Course Book

ALGORITHMS, DATA STRUCTURES, AND PROGRAMMING LANGUAGES

DLBCSL01-01

ALGORITHMS, DATA STRUCTURES, AND PROGRAMMING LANGUAGES

MASTHEAD

Publisher:
IU Internationale Hochschule GmbH
IU International University of Applied Sciences
Juri-Gagarin-Ring 152
D-99084 Erfurt

Mailing address:
Albert-Proeller-Straße 15-19
D-86675 Buchdorf
media@iu.org
www.iu.de

DLBCSL01-01
Version No.: 001-2023-0601
N.N.

© 2023 IU Internationale Hochschule GmbH
This course book is protected by copyright. All rights reserved.
This course book may not be reproduced and/or electronically edited, duplicated, or distributed in any kind of form without written permission by the IU Internationale Hochschule GmbH (hereinafter referred to as IU).
The authors/publishers have identified the authors and sources of all graphics to the best of their abilities. However, if any erroneous information has been provided, please notify us accordingly.

MODULE DIRECTOR

PROF. DR. PAUL LIBBRECHT

Mr. Libbrecht has been a lecturer in Computer Science at IU International University of Applied Sciences since 2020. His main areas are the World Wide Web, data management, and general computer science.

Mr. Libbrecht studied Mathematics at the University of Lausanne (Switzerland) and the Université du Québec à Montréal (Canada). He received his doctorate in Computer Science from Saarland University (Germany). He has been a substitute professor at the University of Education Weingarten (PH Weingarten) and senior developer at the Leibniz Institute for Research and Information in Education. He is a member of the W3C Math Working Group and has been active in the OpenMath Society.

Mr. Libbrecht's research focusses on the technology of learning systems, often with a focus on mathematics. He has published in international conferences and journals. Since 2010, he has worked as a web development consultant for German, French, and US companies.

TABLE OF CONTENTS

ALGORITHMS, DATA STRUCTURES, AND PROGRAMMING LANGUAGES

Module Director . . . 3

Introduction
Signposts Throughout the Course Book . . . 8
Basic Reading . . . 9
Further Reading . . . 10
Learning Objectives . . . 13

Unit 1
Basic Concepts 15

1.1 Algorithms, Data Structures, and Programming Languages as the Basis of Programming . . . 16
1.2 Detailing and Abstraction . . . 20
1.3 Control Structures . . . 24
1.4 Types of Data . . . 30
1.5 Basic Data Structures (List, Chain, Tree) . . . 33

Unit 2
Data Structures 41

2.1 Advanced Data Structures: Queue, Heap, Stack, Graph . . . 42
2.2 Abstract Data Types, Objects, and Classes . . . 55
2.3 Polymorphism . . . 58

Unit 3
Algorithm Design 63

3.1 Induction, Iteration, and Recursion . . . 64
3.2 Methods of Algorithm Design . . . 70
3.3 Correctness and Verification of Algorithms . . . 75
3.4 Efficiency (Complexity) of Algorithms . . . 81

Unit 4
Basic Algorithms 87

4.1 Traversing and Linearization of Trees . . . 89
4.2 Search Algorithms . . . 92
4.3 Sorting Algorithms . . . 94
4.4 Search in Strings . . . 101
4.5 Hash Algorithms . . . 104
4.6 Pattern Recognition . . . 106

Unit 5
Representing Structured Data 111

5.1 Structure of XML documents . . . 114
5.2 Accessing XML Documents with the DOM and SAX Approaches . . . 119
5.3 Transformation of XML documents using XSL . . . 123
5.4 Alternative Document Representations . . . 126

Unit 6
Measuring Programs 129

6.1 Type Inference and IDE Interactive Support . . . 130
6.2 Cyclomatic and Referential Complexity . . . 133
6.3 Digesting Code Documentation . . . 137
6.4 Compiler Optimization . . . 140
6.5 Code Coverage . . . 142
6.6 Unit and Integration Testing . . . 145
6.7 Heap Analysis . . . 148

Unit 7
Programming Languages 153

7.1 Programming Paradigms . . . 154
7.2 Execution of Programs . . . 161
7.3 Types of Programming Languages . . . 163
7.4 Syntax, Semantics, and Pragmatics . . . 167
7.5 Variables and Type Systems . . . 171

Unit 8
Overview of Important Programming Languages 177

8.1 Assembler and Webassembly . . . 178
8.2 C and C++ . . . 183
8.3 Java and C# . . . 186
8.4 Haskell, Lisp . . . 192
8.5 JavaScript and Its Relatives . . . 197
8.6 Other Imperative Programming Languages . . . 200

Appendix

List of References 204
List of Tables and Figures 208

INTRODUCTION

WELCOME

SIGNPOSTS THROUGHOUT THE COURSE BOOK

This course book contains the core content for this course. Additional learning materials can be found on the learning platform, but this course book should form the basis for your learning.

The content of this course book is divided into units, which are divided further into sections. Each section contains only one new key concept to allow you to quickly and efficiently add new learning material to your existing knowledge.

At the end of each section of the digital course book, you will find self-check questions. These questions are designed to help you check whether you have understood the concepts in each section.

For all modules with a final exam, you must complete the knowledge tests on the learning platform. You will pass the knowledge test for each unit when you answer at least 80% of the questions correctly.

When you have passed the knowledge tests for all the units, the course is considered finished and you will be able to register for the final assessment. Please ensure that you complete the evaluation prior to registering for the assessment.

Good luck!

BASIC READING

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2022). *Introduction to algorithms* (4th ed.). MIT Press. http://search.ebscohost.com.pxz.iubh.de:8080/login.aspx?direct=true&db=cat05114a&AN=ihb.51016&site=eds-live&scope=site

Sebesta, R. W. (2016). *Concepts of programming languages* (11th ed.). Pearson. http://search.ebscohost.com.pxz.iubh.de:8080/login.aspx?direct=true&db=cat05114a&AN=ihb.49270&site=eds-live&scope=site

FURTHER READING

UNIT 1

Even, G., & Medina, M. (2012). *Digital logic design: A rigorous approach*. Cambridge University Press. Chapter 8 http://search.ebscohost.com.pxz.iubh.de:8080/login.aspx?direct=true&db=cat05114a&AN=ihb.49269&site=eds-live&scope=site

O'Regan, G. (2018). *The innovation in computing companion:A compendium of select, pivotal inventions*. Springer. Chapter 23 http://search.ebscohost.com.pxz.iubh.de:8080/login.aspx?direct=true&db=cat05114a&AN=ihb.49268&site=eds-live&scope=site

UNIT 2

Andreiana, A.-D., Badica, C., & Ganea, E. (2020). An experimental comparison of implementations of Dijkstra's single source shortest path algorithm using different priority queues data structures. In L.-F. Bărbulescu (Ed.), *2020 24th international conference on system theory, control and computing (ICSTCC)* (pp. 124—129). IEEE. http://search.ebscohost.com.pxz.iubh.de:8080/login.aspx?direct=true&db=edseee&AN=edseee.9259693&site=eds-live&scope=site

Deo, N. (2018). Graphs. In D. P. Mehta & S. Sahni (Eds.), *Handbook of data structures and applications* (2nd ed., pp. 49—68). CRC Press. Chapter 4 http://search.ebscohost.com.pxz.iubh.de:8080/login.aspx?direct=true&db=cat05114a&AN=ihb.49271&site=eds-live&scope=site

UNIT 3

Roughgarden, T. (2021). *Beyond the worst-case analysis of algorithms*. Cambridge University Press. Chapter 1 (Available online)

Sommerville, I. (2016). *Software engineering* (10th ed.). Pearson. Chapter 8 http://search.ebscohost.com.pxz.iubh.de:8080/login.aspx?direct=true&db=cat05114a&AN=ihb.49267&site=eds-live&scope=site

UNIT 4

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). *Data structures and algorithms in Python*. Wiley. Chapter 13 http://search.ebscohost.com.pxz.iubh.de:8080/login.aspx?direct=true&db=cat05114a&AN=ihb.49266&site=eds-live&scope=site

Olukanmi, P., Popoola, P., & Olusanya, M. (2020). Centroid sort: A clustering-based technique for accelerating sorting algorithms. *2020 2nd international multidisciplinary information technology and engineering conference (IMITEC)* (pp. 1—5). IEEE. http://search.ebscohost.com.pxz.iubh.de:8080/login.aspx?direct=true&db=edseee&AN=edseee.9334102&site=eds-live&scope=site

UNIT 5

DOM. (n.d). *Living Document.* WhatWG. (Available online)

JSON. (n.d). *Introducing JSON*. (Available online)

Meggison, David (2004). *About SAX*. SourceForge.(Available online)

Mozilla Corporation (2023) *XSLT Reference*. Mozilla Developer Network. (Available online)

UNIT 6

Reitz, K., & Schlusser, T. (2017). *The hitchhiker's guide to Python*: *Best practices for development*. O'Reilly. Chapter 4 http://search.ebscohost.com.pxz.iubh.de:8080/login.aspx?direct=true&db=cat05114a&AN=ihb.45687&site=eds-live&scope=site

Zhai, H., Casalnuovo, C., & Devanbu, P. (2019). Test coverage in Python programs. *MSR '19 proceedings of the 16th international conference on mining software repositories* (pp. 116—120). IEEE. http://search.ebscohost.com.pxz.iubh.de:8080/login.aspx?direct=true&db=edseee&AN=edseee.8816791&site=eds-live&scope=site

UNIT 7

Hemmendinger, D. (2003). Syntax, semantics, and pragmatics. In A. Ralston, E. D. Reilly, & D. Hemmendinger (Eds.), *Encyclopedia of computer science* (4th ed., pp. 1737—1738). John Wiley and Sons Ltd. (Available online)

Smaragdakis, Y. (2019). Next-paradigm programming languages: What will they look like and what changes will they bring? In H. Masuhara & T. Pietricek (Eds.), *Onward! 2019: Proceedings of the 2019 ACM SIGPLAN international symposium on new ideas, new paradigms, and reflections on programming and software* (pp. 187—197). Association for Computing Machinery. http://search.ebscohost.com.pxz.iubh.de:8080/login.aspx?direct=true&db=edsbas&AN=edsbas.2FFFC074&site=eds-live&scope=site

UNIT 8

Nanz, S., & Furia, C. A. (2015). A comparative study of programming languages in Rosetta code. *ICSE '15: proceedings of the 37th international conference on software engineering* (pp. 778—788). IEEE. http://search.ebscohost.com.pxz.iubh.de:8080/login.aspx?direct=true&db=edsarx&AN=edsarx.1409.0252&site=eds-live&scope=site

Ray, B., Posnett, D., Devanbu, P., & Filkov, V. (2017). A large-scale study of programming languages and code quality in GitHub. *Communications of the ACM*, *60*(10), 91—100. http://search.ebscohost.com.pxz.iubh.de:8080/login.aspx?direct=true&db=edscma&AN=edscma.3126905&site=eds-live&scope=site

LEARNING OBJECTIVES

This course, **Algorithms, Data Structures, and Programming Languages**, will provide students with a basic understanding of algorithms, data structures, and programming languages, which are the foundations of computer programming. It will equip the students with a basic understanding of how to represent algorithms in different ways and how to use control structures, such as loops, conditionals, and recursion, to write programs.

This course will provide the student with a basic understanding of data structures—the building blocks of algorithms. Basic data structures, such as lists, chains, and trees, will be covered. This will be followed by advanced data structures, such as stacks, queues, heaps, and graphs. The concept of abstract data types (ADT) will be introduced for modeling data structures. Students will be taught how to implement data structures using objects and classes.

On completion of the course, the students will have an understanding of basic algorithms and be able to apply them in practical situations. Students will be able to design and analyze basic algorithms and apply suitable algorithms to problems arising in different applications. Additionally, students will have gained a basic understanding of tree traversal, searching, sorting, searching in strings, hashing, and pattern recognition algorithms.

The course will introduce various methodologies for proving the correctness, verification, and testing of programs. On completion of the course, students will understand and be able to apply various program measurement methodologies, as well as explain and compare various programming paradigms and languages.

UNIT 1

BASIC CONCEPTS

STUDY GOALS

On completion of this unit, you will be able to …

- explain the role of algorithms, data structures, and programming languages in programming.
- represent algorithms in various ways.
- understand the role of abstraction and encapsulation in programming.
- define concepts of control structures in programming languages.
- differentiate between different data types.
- create basic data structures, including lists, chains, and trees.

1. BASIC CONCEPTS

Introduction

The stepwise execution of a sequence of instructions to accomplish a given task is ubiquitous in our daily lives. Cooking a dish based on a recipe, searching for a book on a bookshelf in a library, or searching for the shortest route to a destination on a digital map all involve stepwise execution of instructions, with or without the help of a computer. In fact, algorithms and algorithmic computing existed long before the advent of the modern-day digital computer. Ancient civilizations devised systematic methods, or sequences of instructions, to carry out various tasks. Architects in antiquity, such as those in ancient Egypt, devised various systematic geometric constructions using rulers, compasses, and knotted strings. The celebrated algorithm for finding the "greatest common divisor" of two whole numbers was suggested by the Greek mathematician Euclid around 300 BCE. A popular algorithm for finding prime numbers, the "Sieve of Eratosthenes," is attributed to Eratosthenes of Cyrene, an ancient Greek polymath who lived around the second century BCE. Around 250 BCE, the Greek mathematician Archimedes proposed an algorithmic procedure for computing an approximation of π using the ratio of the circumference and diameter of a circle. Although several approximations of π had been proposed earlier, most were merely estimated constant values. Archimedes was the first to suggest an iterative algorithm to compute the value of π, with the accuracy increasing with each iteration. With the advent of digital computers and the increasing complexity of problems solved by them, a systematic paradigm of programming has emerged. For a given problem specification, an algorithm is designed. Data structures will then provide a means of efficient storage, retrieval, and processing of the data encountered by the algorithm. Programming languages then provide the constructs to implement and map the design ideas in the algorithms and data structures to executable code.

1.1 Algorithms, Data Structures, and Programming Languages as the Basis of Programming

The word "algorithm" owes its origin to the Latin translation of the Arabic work of the ninth century Persian mathematician Muhammad ibn Musa al-Khwarizmi (Horowitz et al., 2008). Over the years, computer scientists have come to define an algorithm as a finite sequence of unambiguous instructions that accomplishes a well-defined task in a finite amount of time. Computer algorithms are characterized by the following features (Horowitz et al., 2008):

- input (zero or more input values)
- output (one or more output values)
- definiteness (clear and unambiguous set of instructions)

- termination (ends in a finite number of steps)
- effectiveness (instructions must be feasible)

This definition is machine-independent and would also apply to a pen and paper execution.

The von Neumann Architecture

The Electronic Numerical Integrator and Computer (ENIAC), built in 1946, was one of the first general-purpose digital computers (O'Regan, 2018). Although ENIAC is regarded as the first programmable digital computer, it did not have program storage capabilities. ENIAC's inventors, John Mauchly and John Presper Eckert, proposed its successor, the Electronic Discrete Variable Automatic Computer (EDVAC). Most general-purpose computers and computing as we know them today draw inspiration from the classical **architecture** proposed by John von Neumann (von Neumann, 1945). EDVAC was based on this. This architecture defines what is known as a "stored-program model of computation." In the von Neumann architecture, a computer consists of the following (Liang, 2017):

Architecture
An architecture describes a set of rules and specifications for how software and hardware that comprise a computer system are organized and interact.

- a main memory called random access memory (RAM). Instructions and data reside in the read-write main memory.
- a central processing unit (CPU) consisting of a control unit and an arithmetic and logic unit. The CPU fetches instructions and data from the main memory and performs operations on the data according to the instructions. Results of computations are then written back into the main memory.
- secondary storage units. The data stored in RAM are ephemeral and are no longer available once the system is switched off. Secondary storage units allow us to store data and programs permanently, to be retrieved as required.
- input-output (I/O) units. These include devices such as the keyboard, mouse, monitor, and printer, which allow the user to communicate with the computer.

There are often multiple algorithms for solving a problem. It is good practice to choose one based on efficiency considerations, such as time or space requirements, or ease of implementation.

Programming the Algorithms

Algorithms need to be mapped into instructions in a language that is comprehensible to the computer. This mapped set of instructions is called a "program". Data and programs are stored in memory as "bits" (0s and 1s). The smallest addressable unit of storage is usually a "byte" (8 bits). How each type of data is mapped to bytes depends on the programming language, its version, and the machine concerned. Bytes in memory have unique addresses that can be used to locate, read, and write them.

The language in which the program is written is called a "programming language". Computers have a set of hardware-specific built-in instructions called the "machine language". So, a seemingly obvious choice is to map the algorithm to a machine language program. Programming in machine language involves writing code in a binary number system. That would not only be cumbersome, but such programs would also be hard to read, compre-

hend, debug, and edit. To circumvent such problems, assembly languages were created. An assembly language replaces machine language code with instructions using mnemonics. The assembly language code can be translated into machine language using an assembler. Assembly language is still difficult to work with, while also being machine dependent. Programs are, therefore, more commonly written in platform-independent languages known as “high-level languages”. Examples include Python, Java, C, and C++, but there are many others. Programs written in high-level languages are translated into machine code using “compilers” and “interpreters”. Compilers translate the whole code into machine language, whereas interpreters translate one statement at a time. Java, C, and C++ are examples of compiled languages. Python and Lisp are interpreted languages.

Example Program

Below is a simple Python program fragment for computing the greatest common divisor (GCD) of two positive integers.

Gcd.py

Code

```
first = eval(input('Enter first positive integer:'))
second = eval(input('Enter second positive integer:'))
answer = 1
divisor = 2
while ((divisor <= first) and (divisor <= second)):
   if(((first % divisor)==0) and ((second % divisor)==0)):
       answer = divisor
   divisor += 1
print(answer)
```

This program reads in two positive integers. These are entered as user input from the terminal and stored in main memory as the variables “first” and “second”. The variable names refer to storage locations where these values are stored. The program checks the integers from 2 onwards as possible candidates for GCD. It continues this for as long as the candidate divisor is less than or equal to the smaller of the two input numbers. The variable “answer” is another memory location where the program stores the value of the common divisor found. The assignment statement

```
answer = divisor
```

overwrites the value at this location when a higher valued common divisor is discovered. A common divisor cannot be larger than the minimum of the two numbers. Hence, once this candidate divisor has been checked, the value of the answer as stored in memory is printed out as the GCD of “first” and “second”.

Introduction to Algorithm Analysis

Since there can often be multiple algorithms for the same problem, anyone implementing the algorithm is frequently faced with the problem of deciding which one to choose. Efficiency measures can assist us in making comparisons and arriving at a decision regarding the choice of the algorithm. According to Cormen et al. (2009) two common measures of efficiency are "space complexity", a measure of the amount of memory the algorithm needs, and "time complexity," a measure of how fast the algorithm runs.

There are also other types of complexity measures. The message complexity in distributed algorithms is an example. Programs also use efficient means of structuring data. To measure the efficiency of a data structure, we measure the space used by the data structure (space complexity), the time taken to build the data structure (preprocessing time), the time taken to run a particular query on the data structure (query time), or the time taken to update the data structure (update time).

When measuring the actual time required by an algorithm, there are some challenges. The time required would depend on several factors, such as the machine used, the software environment, and the data set used. Of course, the algorithm must be programmed first. Algorithm analysis involves computing efficiency measures from the**pseudocode** by counting "primitive operations," such as assignments, comparisons, arithmetic operations, function calls, and returns from functions. A basic assumption is that a primitive operation corresponds to, at most, a constant number of instructions on the computer, thus actual times taken by different primitive operations are similar. Hence, the actual time taken by the algorithm is proportional to the number of primitive operations. We call the time taken for an algorithm to run the "running time" or "time complexity", and we measure it in terms of the "input size". Consider the following Python code for **linear search**:

pseudocode
This is text made of words that are understood as elementary programmed operations without being written in a formal programming language.

Linear search
A linear search is a search algorithm to locate a key value in a sequence of unordered elements by comparing the key with elements in the sequence one after the other in the same order.

linearSearch.py

Code

```
def linSearch(numList, keyValue):
   index = 0
   while(index < len(numList)):
       if(keyValue == numList[index]):
           return index
       index += 1
   return -1
```

To analyze the algorithm, note that the two statements `index = 0` and `return -1` are always executed. If the list has n elements (i.e., `numList` is equal to n), the while loop is executed at most n times. The statement `while(index < len(numList))` includes a function call to `len()` and a comparison. The statement `if(keyValue == numList[index])` includes a == operator and is read from a specified position in the list. The statement `return index` adds one more primitive operation. Finally, the increment operation and the assignment `index +=1` may be counted as one or two operations. We

can summarize and claim that the running time of linear search is $n \cdot k + 2$ where k is a small constant and n is the size of the list being searched. There is a better search algorithm, called “binary search”, which takes time proportional to $\log n$. These two time complexities are customarily specified as $O(n)$ and $O(\log n)$ respectively, using the notation for “asymptotic upper bound below”, which guarantees that these bounds hold for all sufficiently large values of n.

Asymptotic upper bound

We define $O(g(n)) = \{f(n)$: There exist positive constants c and n_0 such that $f(n) \leq c \cdot g(n) \, \forall n \geq n_0$. The expression $f(n) = O(g(n))$ denotes the membership of $f(n)$ in the set $O(g(n))$ (Cormen et al., 2009).

Asymptotic lower bound

We define $\Omega(g(n)) = \{f(n)$: There exist positive constants c and n_0 such that $0 \leq c \cdot g(n) \leq f(n) \, \forall n \geq n$. The expression $f(n) = \Omega(g(n))$ denotes the membership of $f(n)$ in the set $\Omega(g(n))$ (Cormen et al., 2009).

Asymptotic tight bound

We define $\Theta(g(n)) = \{f(n)$: There exist positive constants c_1, c_2, and n_0 such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \, \forall n \geq n_0\}$. The expression $f(n) = \Theta(g(n))$ denotes the membership of $f(n)$ in the set $\Theta(g(n))$ (Cormen et al., 2009).

There are situations when more than one parameter is used to specify the time or space complexity. Conventionally, complexity of graph algorithms is specified in terms of the number of vertices $|V|$ and the number of edges $|E|$. Also, there are situations when the complexity may be measured in terms of an input value itself rather than the number of such values. The GCD algorithm presented above runs in time $\mathrm{O(min(m, n))}$, where m and n are the numbers whose GCD is being computed. There is a better algorithm for the GCD problem called Euclid's algorithm that runs in $\mathrm{O(log(min(m, n))}$ time (Cormen et al., 2009).

1.2 Detailing and Abstraction

Once the algorithm has been designed, the designer needs to specify the algorithm clearly and unambiguously. How the data will be organized also needs to be specified. The presence or absence of certain features in a given language influence the algorithm and data structure design, which, in turn, can make the programming effort greater or smaller.

Specifying Algorithms

Natural language

Using natural language to represent algorithms is a seemingly attractive choice. However, natural language is inherently ambiguous and, by definition, algorithms need to be represented in clear unambiguous steps. Therefore, this turns out to be an impractical choice. At the same time, an additional natural language description of an algorithm sometimes complements or augments other forms of representations of the algorithm and improves clarity of understanding for the reader. This is an approach often followed in books, research papers, and technical documents. Despite its natural drawbacks of being ambiguous, natural language is an advantage when the details of an algorithm need to be communicated to people who may not be familiar with programming. Below is a simple natural language description of Euclid's algorithm for finding the GCD of two non-negative numbers.

To find the GCD of two non-negative numbers,

- read the two numbers as input.
- let m be the maximum and n be the minimum of the two numbers.
- if $n = 0$, output m as the answer.
- otherwise, divide m by n and let r be the remainder. Now set $m = n$ and $n = r$ and return to the second step.

Pseudocode

As a method of representation of algorithms, pseudocode comes somewhere in between a natural language description and a program written in a high-level language. It is a more precise representation of the algorithm but usually at a level higher than that of the program. However, since there are no standardized notations to represent pseudocode, people follow their own conventions. It is assumed anyone with some knowledge or background in programming would be able to understand the algorithm. Yet pseudocode has the advantage of being programming language agnostic. The pseudocode describing Euclid's algorithm for finding GCD can take the following form:

GCD

Code

```
begin
    read a, b
    m ← maximum(a, b)
    n ← minimum(a, b)
    while (n ≠0)
        r ← m mod n
        m ← n
        n ← r
```

```
    endwhile
return m
end
```

Flowcharts

Presenting an algorithm as a flowchart was in fashion in the early days of computing. It is still a useful tool to teach or present simple algorithms. The major disadvantage of flowcharts is that they do not scale up well to more complex problems. Below is the flowchart version of Euclid's GCD algorithm.

Figure 1: Euclid's Greatest Common Divisor Algorithm

Source: Created on behalf of IU (2022) based on Euclid.

Choice of Programming Language

Finally, the designed algorithm needs to be mapped to a programming language for execution on a computer. Readability, writability, and reliability are the three most important evaluation criteria for choosing a programming language (Sebesta, 2016). The ease with which software developers can read and understand programs also determines how easily they can be maintained. For example, traditionally, C has been for systems programming, and Lisp or Prolog for artificial intelligence applications. Writability determines how easily the algorithm can be converted to a program. This can be aided by choosing a language more appropriate to the target domain for which it is to be used.

Various attributes of a good language have been identified over the years (Pratt & Zelkowitz, 2001; Sebesta, 2016). These include:

- clarity and simplicity. The simpler the programming language, the easier it is for the algorithm designer to map the algorithms to programs. The algorithm can then be specified with a pseudocode very close to the target language itself.
- expressivity. These include powerful features in the language that allow the programmer to express solutions to problems in clear and natural ways.
- orthogonality. Fewer numbers of primitive and independent constructs and a set of rules for combining them in all possible ways can make a language more convenient to use. If constructs of a language are orthogonal, the language is easy to learn. Exceptions do not need to be learned, since virtually all combinations are allowed.
- support for abstraction. The data structures required for the problem being solved are often different from what is provided in terms of the built-in types. It is the responsibility of the software developer to create appropriate abstractions required for the solution. Implementing these abstractions requires support from the language. For example, Python provides support for object-oriented programming.
- portability or transportability across machines.
- cost of use.

Abstraction and Encapsulation

Procedural or process abstraction in the form of subprograms is a central concept in high-level programming languages. This allows programs to be subdivided into units that are referred to as procedures, functions, or subroutines depending on the language. The units can be authored and used independently by different sets of people. Procedural abstraction allows us to operate the subprograms without knowledge of the low-level implementation details. For example, the function `factorial(x)` in the math module of Python computes the factorial of x. To use the function, we need not worry about how it is actually implemented. The function call is the language feature that supports the abstraction. Other than the name of the function and its parameters, nothing needs to be known to the calling function. The algorithm used by the called function is abstracted out.

Data abstractions allow us to use a data type without details of how it is implemented (Sebesta, 2016). A data abstraction is defined in terms of the associated defining operations.

In data structure design, data abstraction is supported through **abstract data types** (ADTs). The ADT for a data structure specifies what is stored in the data structure and what operations are supported without detailing how the operations are implemented. “Objects” are instances of ADTs.

Abstract data type
An abstract data type is a mathematical model of a data structure that identifies the type of data stored in the data structure and the operations allowed.

Whereas the primary goal of data abstraction is hiding unwanted information, data encapsulation refers to hiding data within an entity along with methods to control access. Since the data organization inside can be manipulated by a controlled set of operations defined by a programmer, only these limited sets of defining operations depend on the internal representation. One often calls the organization of the data the data representation. If the data representation is changed, only the limited set of defining operations needs to

change. Data encapsulation also helps the programmer to ensure that private data rules are enforced. These rules are called "representation invariants." For instance, one may enforce a rule that an ordered list may contain only unique items. If defining operations can only generate objects that follow the representation invariant, then that leads to correct-by-construction implementation—the user cannot create objects that violate these rules.

1.3 Control Structures

The control structures in a programming language facilitate the flow of control inside a program. The control flow, or sequence control, refers to the sequence in which program instructions are executed on a computer. There are three levels at which control structures operate (Pratt & Zelkowitz, 2001):

1. Structures used inside statements, such as those governed by rules of associativity and operator precedence
2. Structures used with groups of statements, such as those associated with conditional statements and iterative loops
3. Structures that facilitate the flow of control between program units

Arithmetic Expressions

Rules and conventions for the evaluation of arithmetic expressions in programming languages usually follow those in mathematics.

Arithmetic expressions are made up of operators, operands, parentheses, and function calls. An operator may be unary, binary, or ternary depending on the number of operands. Binary operators are mostly infix, appearing between their operands, for example, the bitwise ""and"" operation in Python is performed as `a & b`.

Operator evaluation

The control flow in order of evaluation of the operators partly depends on the established "precedence of operators" as defined by the programming language. For instance, consider the Python expression `3+4*2`. Here, the multiplication operation `4*2` is carried out first, since the multiplication operator * has a higher precedence than the addition operator +.

Rules for associativity in the programming language govern the order of evaluation of operators of the same precedence. For example, consider the Python expression `2**3**2`. Here, `**` is the exponentiation operator, which associates right to left. Hence the expression evaluates to `512` and not `64`.

Parentheses can be used to alter the implied order of evaluation as determined by the precedence and associativity rules. The expression `(2**3)**2` will evaluate to `64` in Python because the parentheses override the rules for associativity.

Assignment Operator

The assignment operator, in its simplest variant, takes the following form:

```
variable = expression
```

This requires the expression on the right to be evaluated first before the assignment takes place.

Compound assignments

In many languages, compound assignment operators are supported. Consider the following Python assignments:

Code
```
a = 2
a *= 3 #equivalent to a = a*3
print(a)
```

This will print the value of `a` as `6`.

Multiple assignments

Consider the following assignment statement in Python:

```
a = b = c = 1
```

This is equivalent to the three assignments:

Code
```
c = 1
b = c
a = b
```

Multiple assignments can be done in many languages. Another Python example is

```
a, b, c = 1, 2, 3
```

This is equivalent to:

Code
```
a = 1
b = 2
c = 3
```

This can be used to exchange the values of `x` and `y` as

```
x, y = y, x
```

Comparison Operators and Boolean Expressions

In addition to arithmetic expressions, programming languages also provide constructs for comparison operators and Boolean expressions. The comparison operators supported in Python are `==`, `!=`, `>=`, `<=`, `>`, and `<`.

Boolean expressions also involve the logical operators `or`, `and`, and `not`. A Boolean expression evaluates to `True` or `False`. Logical operators in general work on Boolean operands.

A positive integer is also treated as `True` and `0` is treated as `False`.

Conditional Statements

A conditional statement facilitates branching in programs, allowing the execution to choose between two or more alternate paths. In its simplest form, a Python conditional statement is as follows:

Code

```
if(condition):
    statement

if((x % 3)==0):
   print("Divisible by” 3")
```

Here the intent is to do nothing if the conditional expression is not true. Conditional statements may come with two alternatives. For example, in Python:

Code

```
if(condition):
    Statement 1
Else:
    Statement 2

if(x%2 == 0):
    print("Even")
else:
    print("Odd")
```

Chained conditionals in Python include multiple conditions and statements:

Code

```
if(condition 1):
    Statement 1
Elif(condition 2):
```

```
    Statement 2
.
.
elif(condition n):
    Statement n
else:
    Statement n+1
```

Nested conditionals allow complex logic to be implemented. A Python example is here:

Code

```
if(a == b):
         print("a equals b")
else:
   if(a < b):
       print("a is less than b")
   else:
       print("a is greater than b")
Conditional statements can often be written in many equivalent ways.
The following six statements are equivalent in Python:A) if(x > 0):
   if(x < 100):

B) if((x > 0) and (x < 100)):

C) if x > 0 and x < 100:

D) if (x > 0 and x < 100):

E) if 0 < x < 100:

F) if (0 < x < 100):
```

Iterative Loops

Iterations are repetitive computations of a sequence or a block of statements and form fundamental building blocks of programs. Programming languages support various mechanisms to control how many times the block of statements must be repeated. Two common types of loop control structures provided by programming languages are

- logically controlled loops, e. g., the `while` loop, and
- counter-controlled loops, e. g., the `for` loop.

Function Calls and Recursion

Built-in functions

Languages provide several built-in functions, for example, print(), type(), input(), max(), and min() in Python. Consider the example of the max() function:

Code

```
>>>max(2,3)
3
>>>max(2,3,4)
4
>>>max(max(2,3),4)
4

'c'
>>>max("abc","bcd")
'bcd'
>>>max(2,"two")
TypeError
```

User-defined functions

User-defined functions help in code reuse and in organizing and simplifying code. A simple Python example is

Code

```
def plus5(a):
   return(a+5)
>>>plus5(7)#returns 12
```

Multiple arguments

Functions may have multiple arguments, such as:

Code

```
>>> max(15, 23, 12)
23
>>> max(15, 23.1,12)
23.1
```

Multiple return values

Functions may return multiple values, for example, in the following code:

maxmin1.py

Code
```
def maximinOf3(x, y, z):
   max3 = max(max(x,y),z)
   min3 = min(min(x,y),z)
   return(max3, min3)
print(maximinOf3(15,23,12))
print(maximinOf3(15,23.1,12.5))
```

This prints (23, 12) and `(23.1, 12.5)`, respectively.

No return values

The following example, maxmin2.py, demonstrates a "void" function, which returns nothing. Functions that return something are called "fruitful".

maxmin2.py

Code
```
def maximinOf3(x, y, z):
   max3 = max(max(x,y),z)
   min3 = min(min(x,y),z)
   print (max3, min3)
maximinOf3(15, 23, 12)
```

Recursion

Recursion is a mechanism wherein functions invoke themselves. It often leads to elegant solutions since some problems can be modeled recursively in a natural way.

Recursive functions have a base case that enables us to terminate it. Consider the following factorial function:

$$\text{factorial}\left(n\right) = \begin{cases} 1, n = 0 \\ n*\text{factorial}(n-1), n \geq 1 \end{cases}$$

The corresponding Python function is

fact.py

Code
```
def fact(n):
   if (n==0):
       return 1
   else:
       return n*fact(n-1)
```

Without the base case under the if clause, the function would run indefinitely, causing a runtime error.

1.4 Types of Data

Every programming language provides constructs for structuring data. The types and type system are important characteristics of a programming language and vary from language to language.

Type

A type is defined by a set of values and a set of operations that operate on those values. There are language-specific constraints on the usage of types in a program. A variable of a type can only be operated on by operations defined on the type. A type, in turn, attaches specific meanings to an entity in a program, such as a variable. The hardware would not discriminate between meanings associated with a sequence of bits, that is, whether it is to be interpreted as a string, integer, or character. However, the programming language defines the operations that can be done on the sequence of bits, and the execution of the program translates into microprocessor instructions that manipulate those bits.

Utility of Types

Types have several utilities. Types assist in the hierarchical conceptualization of data. For instance, "employee ID" and "salary" could both be integers. Computing the sum or average is fine for salaries, whereas it would not make sense for the employee ID. Defining separate types for these would require an integer field in both, but a different set of operations could be defined.

Types also ensure correctness. The type system defines rules of usage, which are checked. For instance, the "+" operation in C would represent the addition of numeric types like integers and floating-point numbers. Trying to add two strings would flag an error. In Python, `a + b` would be interpreted as arithmetic addition if both `a` and `b` are numeric types. However, if both `a` and `b` are strings, `a+b` would be interpreted as string concatenation. If `a` is a string and `b` is a numeric type, the Python interpreter flags an error, while in other languages such as JavaScript, both operands are converted to strings and concatenated. A compiler or an interpreter will check if a program is type safe, that is, if all operations are performed in the program with correct types.

Types also define the amount of storage that needs to be allocated. For example, a "char" in C would require one byte of storage. Sometimes, the sizes for different types vary for different implementations of the language.

Type Systems

The type system of a programming language is a logical system defined with a set of constructs to assign types to entities like variables, expressions, or return values of functions (Gabrielli & Martini, 2010). The type system defines the set of built-in types for the language, provides the constructs for defining new types, and defines rules for control of types. There are rules for type compatibility; for example, if a function expects an argument to be a floating-point number, will an integer value for the argument be allowed? Another set of rules defines how the type of an expression is computed from the types of its constituents.

Fundamental Types

Some fundamental types are supported by the language. These usually correspond to the most common and basic ways of structuring data. The set of built-in or fundamental types varies from language to language. For instance, `int` (for integers), `bool` (for Booleans), `char` (for characters), and `float` (single-precision floating-point) are some of the built-in types in C++. There are also more specific types, such as the signed and unsigned variants of `char` or `int`, or the long and short variants of `int`. Python built-in types include `str` (strings), `int`, `float`, `list`, `tuple`, `range`, `dict` (dictionaries), `set`, and `bool` (Booleans).

User-Defined Types

User-defined types allow users of a programming language to extend the fundamental types by creating customized types. Object-oriented languages like Python allow the user to create types called classes. Mechanisms are provided allowing one to create a new class, create objects of that class, and create operations manipulating such objects.

Creating new user-defined types allows the programmer to write programs with the new types closely aligned with the concepts of the application. This helps in writing more concise code and makes the program more readable. Moreover, illegal usage of objects can be detected at compile time, greatly simplifying testing. Type casting is an explicit operation of the programmer to change the type of a variable so that the compiler (and development environment) know which members are accessible. It is necessary, for example, when the function parameter is of a generic type but operations only available to objects of more specific types are necessary.

Strong and Weak Typing

The type system of a programming language lays down a set of rules that the programs written in that language must follow. These rules constrain the set of valid programs that can be written, but are these rules strong enough to ensure that the valid programs do not have type errors? The extent to which this can be guaranteed defines whether the type system is strong or weak. A language with a strong type system is classified as “strongly typed” (Sebesta, 2016). Languages that are not strongly typed are “weakly typed.” Note that these definitions are not precise and there are different viewpoints on the relative strengths of languages. In general, however, an overly restrictive type system may be eas-

ier to check but may severely restrict the set of legal programs. This may also require the user to write more code to ensure type safety, for instance, by explicit conversions using type casting. This is a trade-off that language designers must keep in mind.

Static and Dynamic Type Checking

Statically typed languages obey a static type system, meaning the checking of the type system rules is accomplished at compile time. Declaring all variables with designated types and requiring that expressions have well-defined types are ways to ensure that type safety can be verified at compile time. Examples include Java, C, C++ and Haskell.

In languages with dynamic typing, the checking of the type system rules is conducted at run time. Dynamic checking slows down program execution. In a dynamically typed language, a variable may be bound to an object (but not a type) during compilation, but the binding to is delayed until run time. Examples include Python, Lisp, and JavaScript.

Static typing implies a strongly typed language although the converse is not true. Java is a statically typed language and Python is dynamically typed, but both are regarded as strongly typed language.

Byte Oriented Representations

The smallest unit of storage on a computer is a single bit, 0 or 1. Computers usually operate in groups of bits—eight bits make up a byte and multiple bytes make up a word. More significant bits and bytes of a word, that is, those written to the left when seen on paper in English, are called "higher order" and less significant ones are referred to as "lower order". Algorithms acting on words are implemented in hardware. Common word sizes are 32 bits and 64 bits, as determined by the manufacturer. The type of an operand determines its size (Hennessy & Patterson, 2017). There are multiple views on words and bytes, such as

- logical. This is viewed as a string of bits. There are bitwise operators that act according to this view.
- integer. This can be operated on according to rules of arithmetic operations. Two's complement representation is the most common representation for signed integers. For example 23 in binary over 8bits is $00010111$and -23in binary over 8bits in two's complement is 11101000(taking minus changes the top bit and inverts all other bits).
- floating-point. The operations are the same as for integers, but the word is divided into the sign bit, the mantissa, and the exponent. The mantissa represents the actual bits of the floating-point number, and the exponent represents the power of the radix (in this case two) in the scientific notation. For instance, 25.375 in binary is $11001.011 = 1.1001011 \cdot 2^4$, where 1.1001011 is the mantissa and the unbiased exponent is 100 (or 4 in decimal). The exponent is usually stored after adding what is called a "bias". Since the mantissa always starts with a 1, often only the rest of it, called the normalized mantissa, is stored. Usually, hardware manufacturers follow IEEE 754, which is the technical standard for floating-point arithmetic. Single-precision floating-point usually uses 32 bits and double precision uses 64 bits for representation.
- character. The view represents a character code like 8-bit ASCII, 16-bit Unicode, or 32-bit Unicode.

Big and Little Endian

There are two ways of storing multi-byte data (Even & Medina, 2012):

1. If the machine is Big Endian, the most significant or the leftmost byte of the multi-byte data is stored first (at the lowest address).
2. If the machine is Little Endian, the least significant or the rightmost byte of the multi-byte data is stored first (at the lowest address).

Below is a Python code snippet to determine the “endianness” of a Windows machine:

Code
```
import sys
print(sys.byteorder)
>>little
```

Endianness becomes important if a file is being read on a machine with a different endianness than the one on which it was written. Software circumvents this problem by including a switch to swap bytes if required.

1.5 Basic Data Structures (List, Chain, Tree)

List

The “list” or the “singly linked list” is an unordered sequence of items. We need to be able to maintain the relative positions of these items, so we call the first and last elements of the list the head and tail of the list, respectively. The location of the head of the list is explicitly known, and the location of the $i+1$-th item in the sequence is stored with the i-th item. There is no next item corresponding to the last item on the list. We will construct a Python implementation of a list data structure below, and, for simplicity, we will assume that our lists cannot contain duplicate items. Note that native Python lists are implemented using arrays and are different from linked lists.

Structure

The linked list is built as a collection of basic building blocks called nodes. Each node stores two fields—a data element and a next node information. A Python implementation is shown below:

sList.py

Code
```
class Node:
   def __init__(self, elem):
       self.element = elem
       self.nextNode = None
   def getElement(self):
       return self.element
   def getNextNode(self):
       return self.nextNode
   def setElement(self, elem):
       self.element = elem
   def setNextNode(self, elem):
       self.nextNode = elem
```

Supported operations

We will construct a list data structure that supports the following operations:

- `LinkedList()` constructs an empty list.
- `isEmpty()` returns `True`/`False` based on whether the list is empty or not.
- `getLength()` returns the number of elements in the list.
- `addNode(element)` adds a new element to the front of the list.
- `deleteNode(element)` removes the element from the list.
- `searchNode(element)` searches for the element's item in the list, returning `True`/`False`.

sList.py

Code
```
class LinkedList:
   def __init__(self):
       self.length = 0
       self.head = None
   def isEmpty(self):
       return (self.length==0)
   def getLength(self):
       return self.length
   def addNode(self,elem):
       temp=Node(elem)
       temp.setNextNode(self.head)
       self.head=temp
       self.length +=1

   def deleteNode(self, elem):
       lastNode = None
```

```
        thisNode = self.head
        found = False
        while not found:
            if(thisNode == None):
                break
            if thisNode.getElement() == elem:
                found = True
            else:
                lastNode = thisNode
                thisNode = thisNode.getNextNode()

        if(thisNode==None):
            print("Element not in list")
        elif lastNode == None: #head node gets deleted
            self.head = thisNode.getNextNode()
            self.length -=1
        else:
            lastNode.setNextNode(thisNode.getNextNode())
            self.length -=1


    def searchNode(self, elem):
        thisNode = self.head
        found = False
        while ((not found) and (thisNode != None)):
            if thisNode.getElement() == elem:
                found = True
            else:
                thisNode = thisNode.getNextNode()
        return found
```

In our implementation, `deleteNode` and `searchNode` take $O(n)$ time, where n is the size of the linked list. The other operations take $O(1)$ time.

Chain

A chain is also known as a doubly linked list. It is similar to a singly linked list, except that each node has a pointer to both its predecessor and successor on the list. The symmetrical nature of the doubly linked list makes it easier to implement certain operations on it. However, the price we pay is an extra pointer per node that not only occupies space, but also needs to be correctly updated during list operations. We maintain two sentinel nodes at the head and tail of the list, which simplifies some special cases. The Python implementation follows:

dList.py

Code

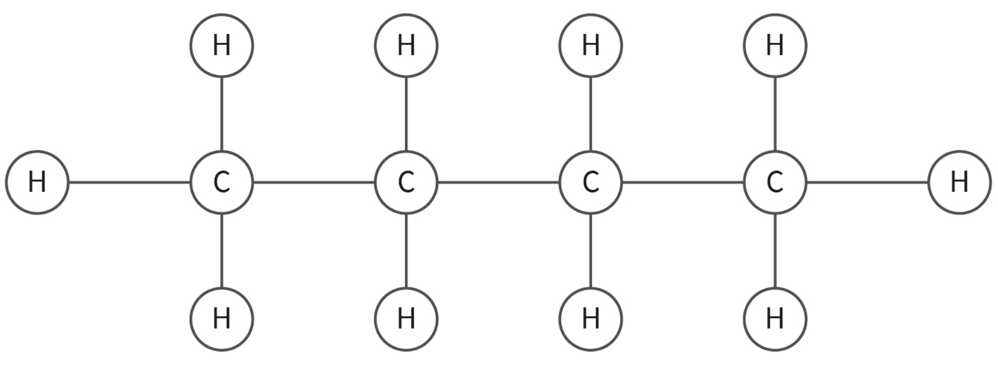
```
class DNode:
   def __init__(self,elem = None, prev=None, next=None):
       self.element = elem
       self.prevNode = prev
       self.nextNode = next
   def getElement(self):
       return self.element
   def getPrevNode(self):
       return self.prevNode
   def getNextNode(self):
       return self.nextNode
   def setElement(self, elem):
       self.element = elem
   def setPrevNode(self, elem):
       self.prevNode = elem
   def setNextNode(self, elem):
       self.nextNode = elem


class DoublyLinkedList:
   def __init__(self):
       self.length = 0
       self.head = DNode(None)
       self.tail = DNode(None)

   def isEmpty(self):
       return (self.length==0)

   def getLength(self):
       return self.length


   def _addNodeIntermediate(self, elem, prev, next):
       #Add element between nodes prev and nextelem
       temp = DNode(elem, prev, next)
       prev.setNextNode(temp)
       next.setPrevNode(temp)
       self.length +=1

   def _deleteNodeIntermediate(self, elem):
       #Remove intermediate node from list
       lastNode = elem.getPrevNode()
       nextNode = elem.getNextNode()
       lastNode.setNextNode(nextNode)
       nextNode.setPrevNode(lastNode)
```

```
        self.length -=1

    def addNodeFront(self,elem):
       #Add element immediately after head node
       self._addNodeIntermediate\
           (self, elem,self.head,self.head.nextNode)

    def addNodeEnd(self,elem):
        #Add element immediately after head node
        self._addNodeIntermediate\
            (self,elem,self.tail.prevNode, self.tail)
```

Trees

The tree is a fundamental data structure that helps to represent connectivity and hierarchy. For example, graphs and trees can be used to model chemical compounds (Ahmad & Koam, 2020) to help visualize their atomic-level connectivity. In 1857, mathematician Arthur Cayley invented the concept of trees when trying to model the problem of counting the number of possible isomers of an alkane (Wilson, 2010). Since then, trees have been widely used to model various problems in chemistry, geology, biology, computer science, and other disciplines.

Figure 2: Tree Representation of Butane Isomers



Source: Created on behalf of IU (2022).

Trees enable us to naturally organize data in the form of file systems, HyperText Markup Language (HTML) pages, organizational structures in companies, and genealogical diagrams called family trees. Trees are also used to represent expressions. In programming language compilers and in natural language processing, parse trees represent the derivation of strings in the language according to the rules of the underlying grammar.

Definitions

Trees may be used for representing acyclic relationships connecting entities, but many trees are rooted. A rooted tree is a collection of nodes storing data elements with the following properties (Goodrich et al., 2013):

- An empty collection is a tree.
- A nonempty tree has a designated node as its root.
- Every node other than the root has a parent.
- A root has no parent.
- If node u is the parent of node v, then node v is the child of node u.

Note that the tree is a recursive structure. A tree T is either empty or consists of a root node r connected to possibly empty subtrees rooted at nodes v where v is a child of r (Goodrich et al., 2013).

If nodes u and v have the same parent w, then u and v are called "sibling nodes".

"External nodes", or "leaf nodes", do not have any children.

Any node a on the path from the root to node v is called an "ancestor" of v. Any node d on the path from node v to a leaf node is called a "descendant" of v (Goodrich et al., 2013).

Nodes with one or more child nodes are called "internal nodes" (Goodrich et al., 2013).

In an m-ary tree, each internal node has at most m child nodes. If each internal node has exactly m children, the tree is called a full m-ary tree. The most common m-ary tree is the binary tree for $m = 2$.

The length of the path, in terms of the number of nodes, from the root to a node v in the tree is called the "level" of v. The maximum of the levels of all the vertices in the tree is called the "height" of the tree (Goodrich et al., 2013).

SUMMARY

An algorithm is a finite sequence of unambiguous instructions that accomplishes a well-defined task in a finite amount of time. Algorithms are used to solve problems, but in order to do so, they must be mapped into instructions in a language that is comprehensible to the computer. This mapped set of instructions is called a "program". Methods for specifying an algorithm including natural language, flowcharts, and pseudocode.

Most general-purpose computers as we know them today draw inspiration from the classical architecture proposed by John von Neumann and consist of RAM, CPU, secondary storage, and I/O.

Procedural abstraction allows us to operate subprograms without knowledge of the low-level implementation details. Data abstractions allow us to use a data type without the details of how it is implemented. Data encapsulation features involve hiding data within classes along with methods to control access.

Control structures facilitate the flow of control through a program. These include structures inside statements governed by rules of operator precedence and associativity, structures for conditional statements and loops, and flow of control between subprograms.

The programming language features of functions and function calls support procedural abstractions. Recursive functions are an elegant yet powerful feature that allows functions to invoke themselves.

Types are programming language features that facilitate the structuring of data. Types include built-in and user-defined types, type systems, and static and dynamic typing. Important types include basic data structures include lists, chains, and trees with their Python implementations.

UNIT 2

DATA STRUCTURES

STUDY GOALS

On completion of this unit, you will be able to …

- implement the data structures: stack, queue, heap, and graph.
- understand the concepts of abstract data types, objects, and classes.
- apply different types of polymorphism.

2. DATA STRUCTURES

Introduction

Data structures represent data and relationships among data for efficient manipulation. Data are much more than collections of bits and bytes. Data are associated with objects and their representations. Objects could be persons, physical objects, events, or abstract concepts. For representations, there are choices to be made regarding what attributes are to represent the objects, what queries we need to ask of the objects, and how frequently. Consider a simple problem of storing and querying a set of integers. The goal is to answer a search query from the user about the presence or absence of an integer of the user's choice in our collection. Suppose the design decision to make is whether we should store it in a sorted array or an unsorted array. In general, search works better in sorted arrays. There are algorithms to execute our search problem on a sorted array within a time that is logarithmic relative to the number of integers stored. A brute-force scan through the array would take linear time. However, if our set is unsorted to start with, we will need to sort it first, but then the time taken to sort followed by a binary search would be expensive compared to a brute-force linear search. So, does that mean that we should use linear search as opposed to binary search for this problem? Yes, if we simply had to search only once. If we had to search several times, the cumulative advantage of the logarithmic searches over linear ones would be significant, even considering the overhead of the sorting step. Now, scale this problem up to web searching. We expect our answers immediately! The search engine is able to satisfy our requirement for speed because of sophisticated preprocessing and data storage ahead of processing our query.

Object-oriented programming allows us to identify the fundamental objects in our design, publish an abstraction with essential methods, and hide the implementation details. Features of languages supporting this paradigm also allow us to encapsulate these objects into classes. In this unit (which describes data structures as classes) these principles are demonstrated as they are applied.

2.1 Advanced Data Structures: Queue, Heap, Stack, Graph

Stacks

A “stack” is a collection of data items following a “Last In, First Out” (LIFO) paradigm (Goodrich et al., 2013). The basic operations supported by a stack are as follows (using Python nomenclature):

- `push(element)` adds an element to the top of the stack.
- `pop()` removes an element from the top of the stack.
- `topOfStack()` reads an element from the top of the stack.

- `isEmpty()` checks if the stack is empty and returns `True`/`False`.
- `size()` returns the number of elements in the stack.

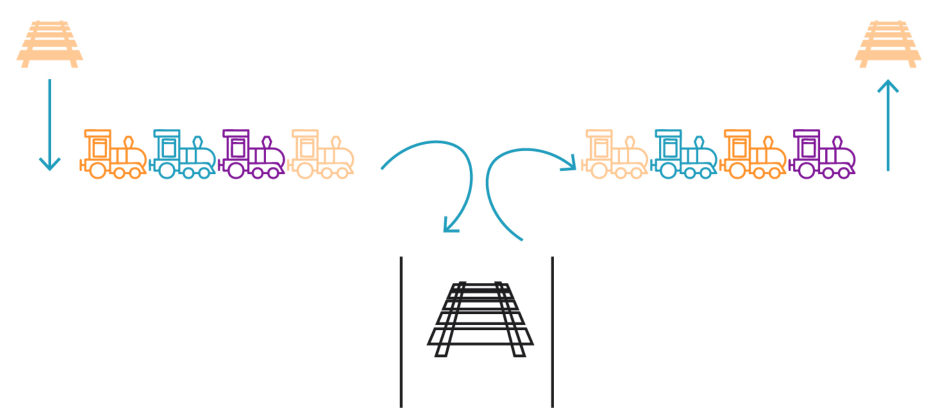
Stacks can be implemented using singly linked lists, although the simpler array implementations are also common. An array implementation of a stack using Python lists is given below:

stack.py

Code
```
class Stack:
   def __init__(self):
       self.elements = [] #Initialized to empty list

   def isEmpty(self):
       return self.elements == []

   def size(self):
       return len(self.elements)

   def topOfStack(self):
       return self.elements[len(self.elements)-1]

   def push(self, newElement):
       self.elements.append(newElement)

   def pop(self):
       return self.elements.pop()#Python's in-built pop
```

One important application of stacks is in stack frames in function invocation. An "activation record" (also called "stack frame") is created when a function is invoked. This stores information about variables and arguments of the function. When functions invoke other functions, new activation records are created and pushed onto the "call stack." Later, as the called function returns control to the calling function, the former's activation record is popped from the call stack. Other applications of stacks include matching parentheses in parsed expressions, depth-first search in graphs, and matching tags in HyperText Markup Language (HTML).

Figure 3: Railroad Car Switching Using Stacks



Source: Created on behalf of IU (2022).

Stacks can be used to switch railroad cars of a train in a switching yard (Knuth, 2013). In the figure above, railroad cars arrive at the switching yard in the order D, C, B, and A. A stack-like structure is used to switch the cars so that they leave the yard in the order C, A, B, and D. The following Python code simulates the process:

Code

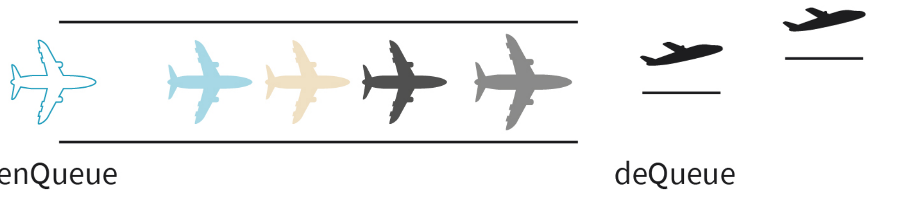
```
s=Stack()
s.push("D")
s.push("C")
print(s.pop())
s.push("B")
s.push("A")
print(s.pop())
print(s.pop())
print(s.pop())
```

This prints `C,A,B,D,` in that order.

Queues

A “queue” is a collection of data items following a “First In, First Out” (FIFO) paradigm (Goodrich et al., 2013). An example of a queue is the departure queue of flights taking off from a particular runway at an airport. An aircraft that is ready to depart enters the queue. Aircraft in the queue wait for their turn. The aircraft at the front of the queue takes off after receiving permission from air traffic control.

Figure 4: Departure Queue of Aircrafts on a Runway



Source: Created on behalf of IU (2022).

The basic supported operations of a queue are as follows:

- `enQueue (element)` adds an element to the rear of the queue.
- `deQueue()` removes an element from the front of the queue.
- `isEmpty()` checks if the stack is empty and returns `True/False`.
- `size()` returns the number of elements in the queue.

A queue can be implemented using linked lists or circular arrays. An implementation using Python lists is given below:

queue.py

Code
```
class Queue:
    def __init__(self):
        self.elements = []

    def isEmpty(self):
        return self.elements == []

    def size(self):
        return len(self.elements)

    def enQueue(self, newElement):
        self.elements.insert(0,newElement)

    def deQueue(self):
        return self.elements.pop()
```

Queues are also used in job scheduling, traversal mechanisms in graphs such as breadth-first search (or BFS, which will be explained later on), as well as other applications.

Heaps

A "heap" is a data structure that is used as a building block in two important problems—**heapsort** and implementation of priority queues. Many algorithms, such as Dijkstra's shortest path algorithm, Prim's minimal spanning tree algorithm, and various job schedul-

Heapsort
A heapsort is a sorting algorithm that builds a heap of the numbers to be sorted and repeatedly removes the maximum (or minimum).

ing and selection problems, use a heap as a fundamental data structure (Cormen et al., 2009). We consider the binary heap here. There are other variants, including the Binomial Heap, Fibonacci Heap, and Leftist Heap, among others, that have their own properties (Brodal, 2013).

Priority queue operations

Let us consider the problem of implementing a priority queue using a heap. The basic item is a <data, priority> pair. The priority queue attempts to keep track of the item with the highest priority (Goodrich et al., 2013).

The supported operations are as follows:

- `insert (element)` adds an element to the priority queue.
- `extractMax ()/extractMin()` removes the element of highest or lowest priority.
- `reportMax()/reportMin()` returns the highest or lowest priority value.

Heap property

Complete binary tree
A complete binary tree is a binary tree with the following properties: (a) all leaves are either at the last level or the last level and the penultimate level and (b) leaves at the last level are packed as far left as possible.

Let T be a **complete binary tree** with nodes v having fields defined as follows:

- $\mathrm{key(v)}$ is the key associated with node v.
- $\mathrm{Left(v)}$ is the left child of v.
- $\mathrm{Right(v)}$ is the right child of v.

Let r be the root of T. T is a heap if

- T is NULL or

b) $\mathrm{Key(r)} \geq \max(\mathrm{key(Left(r))}, \mathrm{key(Right(r))}$.
c) The subtrees rooted at $Left(r)$ and $Right(r)$ are heaps.

This is called a MAX-heap. Analogously, we can define a MIN-heap (Cormen et al., 2009).

Heap implementation

A binary heap can be implemented using a linked structure in the form of a binary tree. A commonly preferred implementation is using an array. A MAX-heap implemented using Python lists is given below:

heaps.py

Code

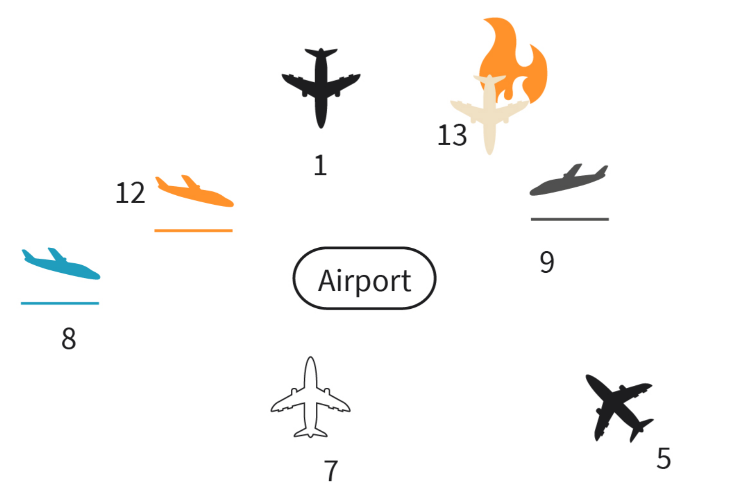
```
class Heap:
   def __init__(self):
       self._X = []

   def isEmpty(self):
```

```
        return self._X == []

    def size(self):
        return len(self._X)

    def _parent(self, i):
        return((i-1)//2)

    def insert(self, newElement):
        #Append at the end
        self._X.append(newElement)
        i = self.size()-1
        #Bubble up
        while(i > 0):
            top = self._parent(i)
            if(self._X[top] < self._X[i]):
                self._X[top],self._X[i] \
                    = self._X[i],self._X[top]
            else:
                break
            i = top

    def _maxChild(self, i):
        if 2*i + 2 >= self.size():
            maxChild = 2*i+1
        elif self._X[2*i+1] > self._X[2*i+2]:
            maxChild = 2*i+1
        else:
            maxChild = 2*i+2
        return(maxChild)


    def extractMax(self):
        #Remove the maximum element from heap and return
        maxElement=self._X.pop(0)
        if(self.size() != 0):
            #Bring last element to front
            lastElement=self._X.pop()
            self._X.insert(0, lastElement)
            #Trickle down
            i = 0
            while(2*i < self.size()-1):
                m = self._maxChild(i)
                if(self._X[m] > self._X[i]):
                    self._X[i],self._X[m] \
                        = self._X[m],self._X[i]
                else:
                    break
```

```
                i =m
            return maxElement

    def reportMax(self):
        return(self._X[0])

    def printHeap(self):
            print(self._X)
```

Of the operations, `reportMax` takes $O(1)$ time while both `extractMax` and `insert` take $O(\log n)$ time, where n is the number of items in the heap when the operation is performed.

Figure 5: Aircraft Landing Problem



Source: Created on behalf of IU (2022).

An application

Consider an aircraft landing problem at an airport, where the air traffic control tries to prioritize the landing of aircraft based on various factors quantified by a priority value. Aircraft with higher priority land earlier. For instance, an airplane that is already low and very close will have a high priority. Consider the scenario shown in the figure, where aircraft with different priority values seek landing permission. These priorities are inserted into a priority queue implemented as a MAX-heap. Then, an aircraft arrives seeking an emergency landing. It has the priority value `13`, which is the maximum. The priority queue returns this value for the `extractMax` operation. These operations are executed using our heap implementation as follows:

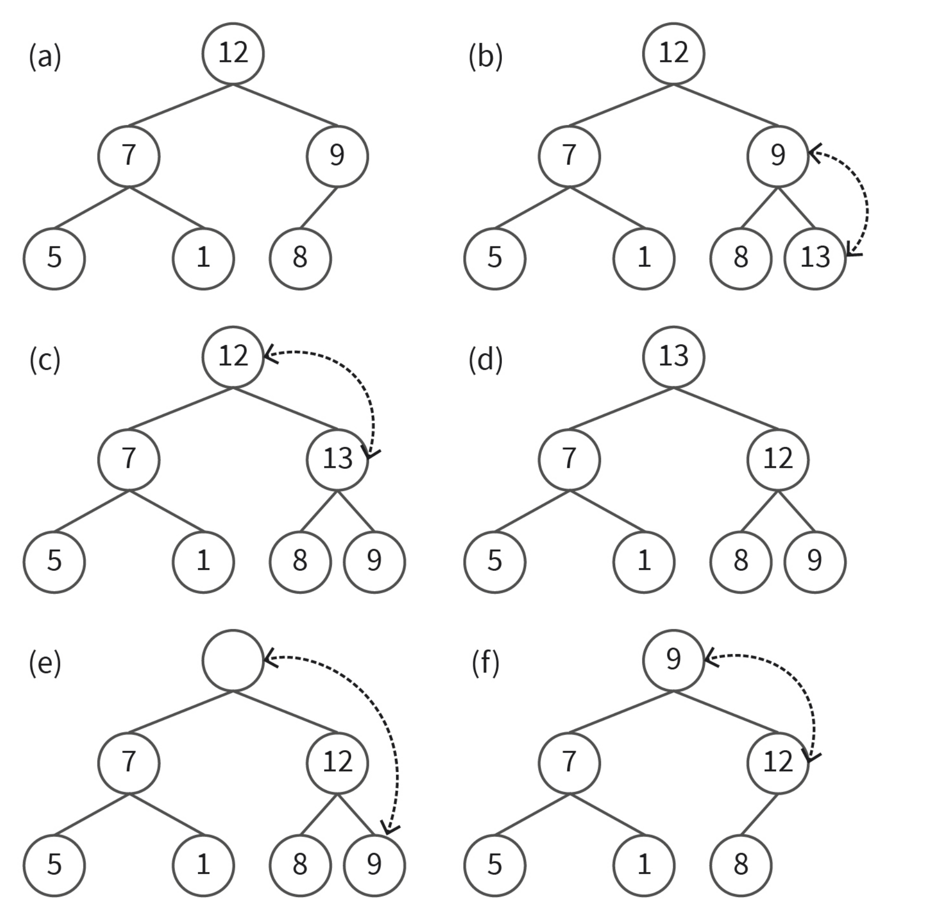
Code

```
H=Heap()
H.insert(5)
H.insert(12)
```

```
H.insert(9)
H.insert(7)
H.insert(1)
H.insert(8)
H.insert(13)
H.extractMax()
```

The structure of the MAX-heap as it evolves while executing an `insert(13)` followed by `extractMax` is shown below.

Figure 6: Heap Operations



Source: Created on behalf of IU (2022).

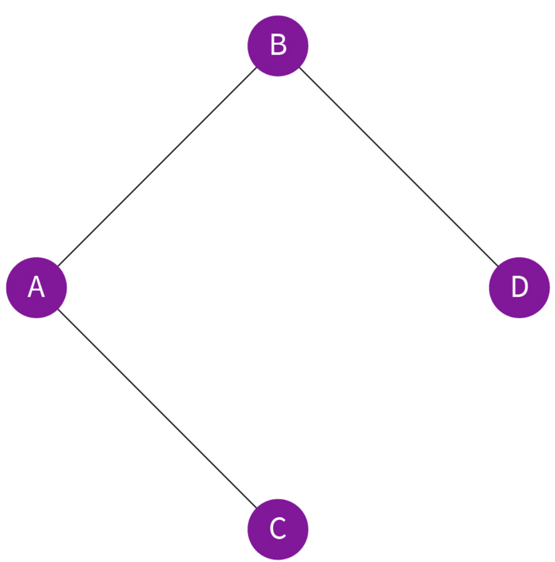
Graphs

The celebrated problem of Königsberg bridges asked whether the seven bridges of the Prussian city of Königsberg, over the river Preger, could all be traversed in a single trip without going through any bridge twice (Rosen, 2019). The additional requirement was that the trip must end in the same place it began. In 1736, Leonhard Euler showed that the Königsberg bridge problem could not be solved. This initiated the study of graph theory, which is central to computer science today (Rosen, 2019).

A graph $G = (V, E)$ consists of a set of vertices V and a set of edges $E \subseteq V \cdot V$.

An edge (A, B), $A \epsilon V$ and $B \epsilon V$ connects vertices A and B.

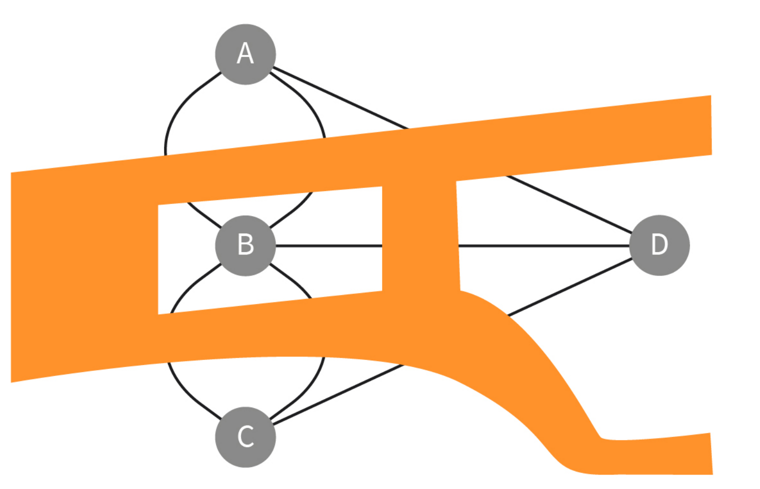
Figure 7: A Graph



Source: Created on behalf of IU (2022).

In the graph representation of the Königsberg bridge problem, each vertex represents a landmass, and each edge represents a bridge.

Figure 8: The Konigsberg Bridge Problem



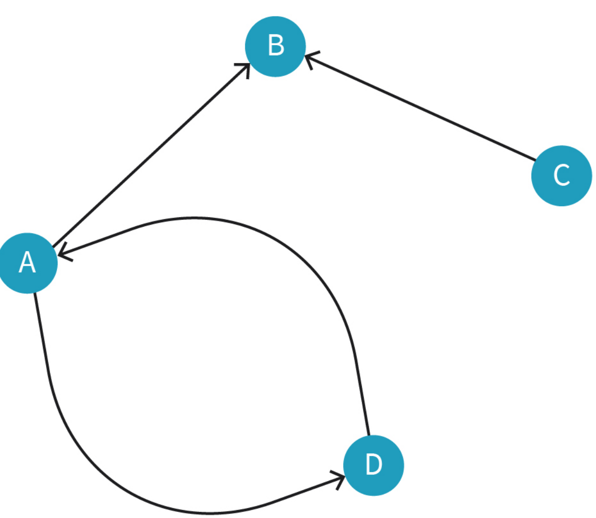
Source: Created on behalf of IU (2022).

Self-loops
These are edges in graphs that start and end at the same vertex.

A “simple graph” has no **self-loops** and does not have multiple edges between vertices. A graph with multiple edges or self-loops is called a pseudo-graph (Rosen, 2019).

A “directed graph” has only directed edges between pairs of vertices. An “undirected graph” has no directed edges (Rosen, 2019).

Figure 9: A Directed Graph



Source: Created on behalf of IU (2022).

Applications

Social networks are often represented as graphs. We sometimes call such graphs “social graphs.” The entities represented by vertices could be individuals, posts, or some comments. The edges connecting the vertices represent some relationship between the entities between the vertices. For example, the edge of a graph showing Facebook connections would represent friendship. Graphs may have different types of vertices. For instance, in a collaboration network, a vertex may be an author or a research paper. In some networks, the edges represent different types of relationships, such as friendship, familial relationships, or acquaintance. In others, such as trust networks, the edge may be weighted. The relationship is non-random, and, often, the entities form clusters of “communities”, which are not necessarily disjointed. The graph representation also depends on what type of data will be mined. A collaboration network in research may be represented with (a) vertices as authors and edges indicating co-authorship, (b) vertices as papers with edges indicating the presence of common authors, or (c) each vertex being either an author or a paper and an edge indicating authorship of a paper.

Other applications of graphs include (Rosen, 2019)

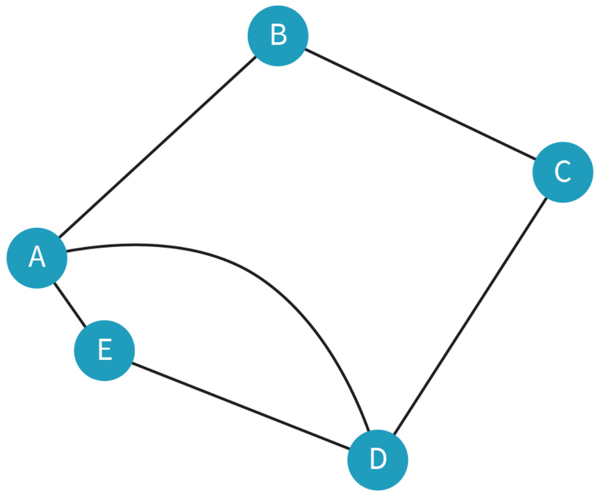
- road networks.
- web pages and their hyperlinks.
- communication networks.
- collaboration networks.
- airline network (connectivity between cities).

Cycles

Cycles
A cycle is a sequence of vertices such that every consecutive pair in the sequence is connected by an edge, and the last vertex in the sequence is connected to the first.

Various problems are modeled using **cycles** in graphs.

Figure 10: An Undirected Graph with Cycles

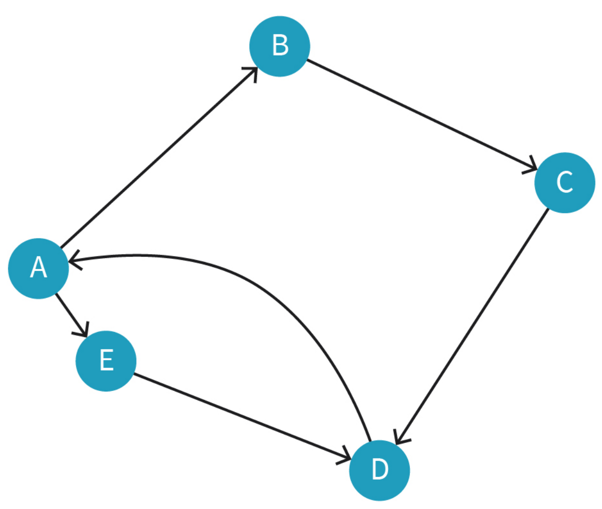


Source: Created on behalf of IU (2022).

In directed graphs, we look for a directional sequence to determine a cycle.

In the graph below, A-E-D-A and A-B-C-D-A are cycles, but A-B-C-D-E-A is not.

Figure 11: A Graph with Directed Cycles



Source: Created on behalf of IU (2022).

DAGs

Directed acyclic graphs (DAGs) are useful for modeling dependencies between tasks.

Directed acrylic graphs
A DAG is a directed graph with no directed cycles.

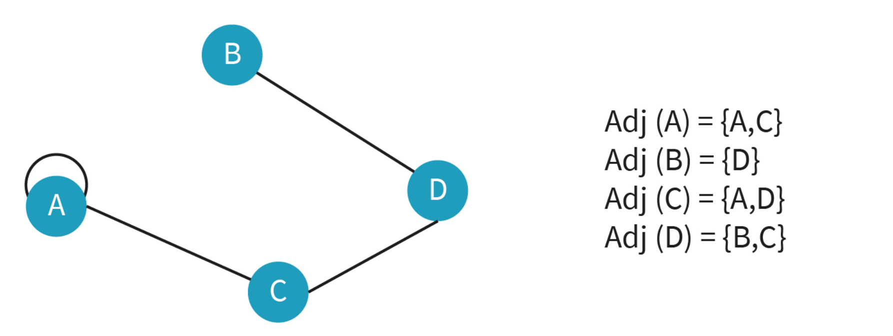
Graph representation

Two popular ways in which graphs may be represented are "adjacency lists" and "adjacency matrices" (Rosen, 2019).

For an undirected graph $G = (V, E)$, the adjacency list for vertex v, $Adj(v)$, stores the list of vertices connected to v, that is: $Adj \quad (v) \quad = \{u \,|\, (v, u)\epsilon E\}(u, v)\epsilon E$ if $(v, u)\epsilon E$. This is usually the preferred representation if the graph is **sparse**. An adjacency list representation of an undirected graph is shown below along with the adjacency lists. For example, the self-loop at vertex A and a straight edge between A and C results in $Adj(A) = \{A, C\}$.

Sparse
A sparse graph is a graph with |V| vertices and O(|V|) number of edges.

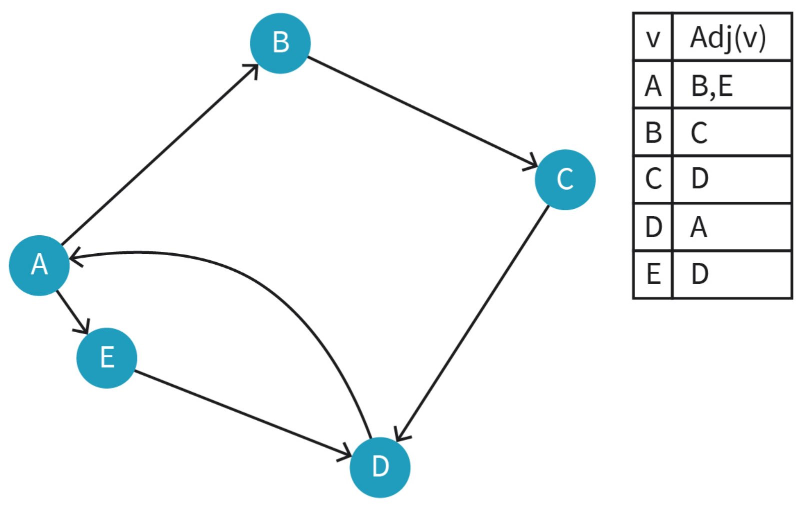
Figure 12: Adjacency List of an Undirected Graph



Source: Created on behalf of IU (2022).

For a directed graph $G = (V, E)$, the adjacency list for vertex v, $Adj(v)$ stores the list of neighbors $\{u \,|\, (v, u)\epsilon E\}$. In some applications, $Adj(v)$ may store incoming edges at v.

Figure 13: Adjacency List of a Directed Graph



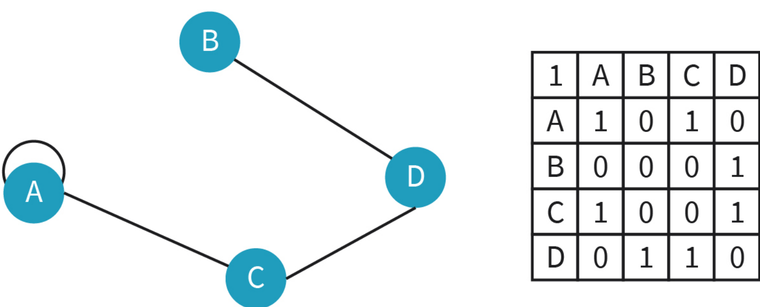
v	Adj(v)
A	B,E
B	C
C	D
D	A
E	D

Source: Created on behalf of IU (2022).

For any simple graph G, directed or undirected, let us assume that the vertices are named $v(i), 0 \leq i \leq n-1$. The adjacency matrix A is a two-dimensional Boolean matrix where $A(i,j) = 1$ if, and only if, $(v(i), v(j))$ is an edge in the graph. This is sometimes the preferred representation if the graph is **dense**.

Dense
A dense graph is a graph with |V| vertices and O(|V| 2) number of edges.

Figure 14: Adjacency Matrix of an Undirected Graph

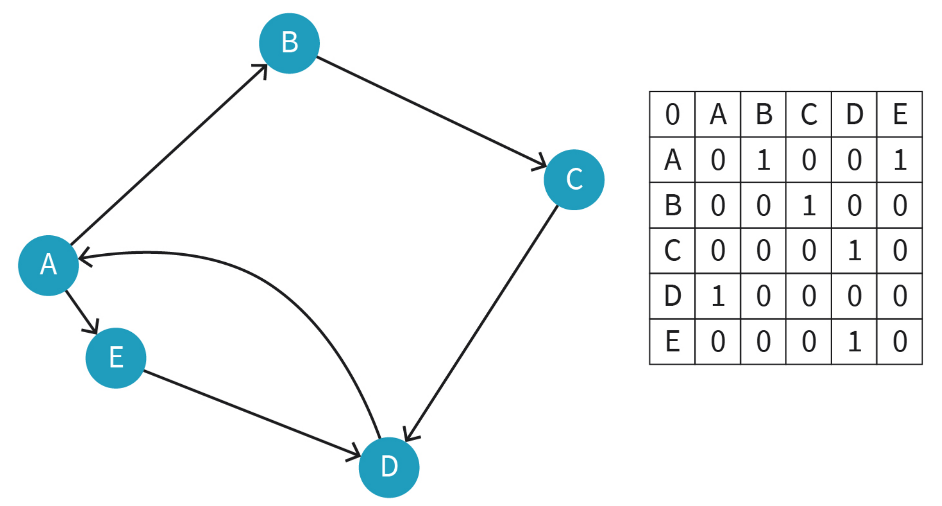


1	A	B	C	D
A	1	0	1	0
B	0	0	0	1
C	1	0	0	1
D	0	1	1	0

Source: Created on behalf of IU (2022).

The adjacency matrix of an undirected graph is symmetric. This need not be the case for a directed graph.

Figure 15: Adjacency Matrix of a Directed Graph



0	A	B	C	D	E
A	0	1	0	0	1
B	0	0	1	0	0
C	0	0	0	1	0
D	1	0	0	0	0
E	0	0	0	1	0

Source: Created on behalf of IU (2022).

Both adjacency list and adjacency matrix representations are widely used; however, adjacency matrices have a space requirement of $O\left(|V|^2\right)$, while adjacency lists have a space requirement of $O\left(|V|^2|E|\right)$.

2.2 Abstract Data Types, Objects, and Classes

ADTs

The abstract data type (ADT) for a data structure specifies what is stored in the data structure and what operations are supported on them, as shown for the stack or the queue data structure above. The ADT does not detail how the operations are implemented.

Defining a graph ADT

Let us define an ADT for the graph data structure. A graph consists of vertices and edges, so we need to define ADTs for these as well. Suppose we use a graph to represent a network of highways connecting a set of cities. Each vertex represents a city. Each edge represents a pair of cities that are connected by a direct highway between them. We assume the edge is undirected.

Our ADTs will include the following:

- Vertex ADT

- VerteVertex(name) creates a vertex with a given city name.x ADT
  - getName() returns the name of the vertex.
- Edge ADT
  - Edge(a, b, w) creates an edge between vertices a and b with weight w.
  - getVertices() returns a pair (a, b) of vertices representing the edge.
- Graph ADT
  - Graph() creates an empty graph.
  - getNumVertices() returns the number of vertices of the graph.
  - getNumEdges() returns the number of edges of the graph.
  - addVertex(a) adds a vertex to the graph for a city named a.
  - deleteVertex(a) deletes the vertex for the city named a, if any, from the graph.
  - addEdge(a, b) adds an edge between vertices with names a and b.
  - deleteEdge(a, b) removes the edge between vertices with names a and b.
  - getVertices() iterates through all vertices of the graph.
  - getEdges() iterates through all edges of the graph.
  - degree(v) returns the degree of vertex v.
  - neighbors(v) returns all neighbors of vertex v.
  - getEdgeBetween(a, b) returns the edge, if any, between vertices a and b.
  - pathCost(a, b) returns the length of the shortest path between cities named a and b (the sum of all weights).

Note that we have not defined details of the representation of the graph, for example, whether we are using adjacency lists or adjacency matrices. The ADT serves as the public interface for those using the graph data structure. The implementation details are hidden.

To implement ADTs in a language, a suitable syntactic structure needs to be provided so that the clients of the abstraction can declare instances of the ADT and operate on them. The data representation and implementations of the operation are hidden from the outside world. Sometimes, support may be required to allow objects of ADTs to be operated on by a few general built-in operations, such as assignment and comparison, since these may need to be redefined for the user-defined type.

Heap ADT

In the heap example, the heap ADT may be defined as follows:

- Heap() constructs an empty heap.
- isEmpty() returns True if the heap is empty, False otherwise.
- size() returns the number of elements in the heap.
- insert(element) adds an element to the heap.
- extractMax() removes and returns the maximum element of the heap.

This is the heap's public interface. Information that is notably not a part of the ADT includes:

- internal representation, whether an array or a linked tree structure is used.
- details of how the functions implement the supported operations.
- some operations, such as _parent and _maxChild, that are for internal computations.

Objects and Classes

A “class” is a fundamental means of abstraction in object-oriented programming. In Python, every data item is an instance of a class. This is true for both built-in types and user-defined types. Classes are instantiated as objects: an object of a given class is called an instance. In the heap example, the statement `H=Heap()` creates an object of class Heap. The class determines how information is represented, while the instance (generally called “object”) stores the concrete information. The basic data represented in our heap example are a sequence of integers. This is implemented using a Python list `_elements`. This list is called a “data member” or “field” of the class. Whether `_elements` is a list or tree is internal to the class and is not of concern to the users of the class. The class also defines behavior in the form of methods or member functions.

Constructors

Once we have defined a class, we can create instances or objects of the class using a “constructor”. For example, we can create an instance of the Heap class by invoking the constructor as `Heap()`. This accomplishes two things: It creates an object in the memory and it calls the `__init__` method of the class to initialize (assign data to) object.

Inheritance

“Inheritance” adds a powerful feature to object-oriented programming that facilitates modular and hierarchical organization. This enables us to define new classes based on the existing class. The new class is called the “derived class” or “subclass”. The existing class from which the subclass is derived is known as the “superclass” or “base class” (Goodrich et al., 2013).

The subclass

- inherits some methods from the base class.
- extends the base class with new methods.
- overrides some methods from the base class.

As an example, consider classes of parallelograms.

parallelograms.py

Code

```
class Parallelogram:
        def __init__(self, p, q):
                self.first = p
                self.second = q
                self.third = p
                self.fourth = q

        def perimeter(self):
                return(self.first + self.second +
```

```
                self.third+self.fourth)

class Rectangle(Parallelogram):
        def __init__(self, p, q):
    super().__init__(p,q)

        def area(self):
                return(self.first*self.second)

class Square(Rectangle):
        def __init__(self, p):
                super().__init__(p,p)

        def area(self):
                return(self.first*self.first)

P=Parallelogram(3,4)
print(P.perimeter())

R=Rectangle(3,4)
print(R.perimeter())
print(R.area())

S=Square(5)
print(S.perimeter())
print(S.area())
```

In the above example, we observe that

- the `Rectangle` class inherits the perimeter method from the base class `Parallelogram`, as does the `Square` class,
- the `Rectangle` class extends the base class `Parallelogram` with an area method, and
- the `Square` class area method overrides the area method from the `Rectangle` class.

Python supports object-oriented programming through the mechanism of "abstract base classes" (Goodrich et al., 2013). One cannot create objects of these classes. Instances are created from concrete classes derived from the abstract base classes.

2.3 Polymorphism

Software reuse implies better productivity. The ability to use the same subprogram for different types of data leads to software reuse and is a powerful facility provided by different languages supporting object-oriented programming. This facility is known as "polymorphism" and manifests itself in different ways in programming languages (Goodrich et al., 2013).

In “ad-hoc polymorphism”, a function can have different implementations depending on the types of its arguments. Function overloading in C++ is an example of ad-hoc polymorphism. C++ also provides support for generic types in the form of templates.

In Python, support for polymorphism exists for both built-in types and for user-defined classes.

The Len Function

The len function in Python works for several types, including ranges, strings, lists, tuples, sets, and dictionaries:

Code

```
A = range(0, 5) #range
print(len(A))

B= [2,3,4,5] #list
print(len(B))

C= (4, 5,6) #tuple
print(len(C))

D = {4,5,6,7,8,9} #set
print(len(D))

E = {'a':2, 'b':3} #dictionaries
print(len(E))
```

The + Operator

The + operator works for a variety of types, such as strings, numeric types, lists, and tuples, but both operands must be of the same type:

Code

```
a = 23
b = 45
print(a+b) # 68

a = "abc"
b = "def"
print(a+b) # abcdef

a = [1,2,3]
b=[4,5]
print(a+b) # [1, 2, 3, 4, 5]
```

```
a = (1,2,3)
b = (3,4,5)
print(a+b) # (1, 2, 3, 3, 4, 5)
```

Python allows the "+" operator to be overloaded with objects of a user-defined class as operands. A method `__add__` needs to be defined for the class. For the expression `obj1 + obj2` where `obj1` and `obj2` are objects of some user-defined class, `obj1.__add__(obj2)` is invoked.

Polymorphism with Inheritance

Let us revisit the `Parallelogram` example. Note that the perimeter is implemented in the base class `Parallelogram`, but not in the derived classes `Rectangle` and `Square`. When we instantiate an object `S` of class Square and invoke `S.perimeter()`, the `perimeter()` method defined in the superclass `Parallelogram` is called. Now, consider the area methods defined in the `Rectangle` and `Square` classes. When we create an object `S` of the `Square` class and invoke `S.area()`, the area method defined in class `S` is called. If we delete this method, since class `Square` is a subclass of the class rectangle, the rectangle's area method will be invoked whenever `S.area()` is called.

In general, an object of a derived class can be passed on as a parameter of a superclass. If a method is implemented in some, but not all, classes in the inheritance hierarchy, the implementation in the nearest superclass to the invoked object's class is invoked (Liang, 2017).

SUMMARY

Important data structures include stacks, queues, heaps, and graphs.

Stacks are structures following the LIFO paradigm. The main operations for a stack are the PUSH and the POP. The PUSH operation involves the addition of an element to the top of the stack. The POP operation involves the removal of an element from the top of the stack. They can be implemented using linked lists or arrays. We proposed an implementation using Python lists. Stacks are useful in many algorithms including parentheses matching in expressions, matching tags in HTML, and supporting stack frames to process function calls.

Queues are FIFO structures that support operations of enqueue to add elements at the end of the queue and dequeue to remove them from the front. We proposed an implementation using Python lists.

Heaps are used in Heapsort and for implementing priority queues. They come in two varieties: MAX-heap and MIN-heap. The basic operations supported include returning the maximum or minimum in $O(1)$ time and removing the same in $O(\log n)$ time. The heap also supports an addition of a new element in $O(\log n)$ time.

Graphs are a fundamental data structure used for modeling relationships such as computer networks, social networks, communication networks, road networks, and biological networks.

We considered ADTs, objects, and classes as fundamental tools to implement abstraction in object-oriented programming. Inheritance adds a powerful feature to object-oriented programming, which facilitates modular and hierarchical organization. This enables us to define new classes based on the existing class. We studied polymorphism as an important concept in object-oriented programming. We saw examples in Python where support for polymorphism exists for both built-in types and user-defined classes.

UNIT 3

ALGORITHM DESIGN

STUDY GOALS

On completion of this unit, you will be able to …

- use iteration and recursion to generate repetition in programs.
- design algorithms using basic algorithm design paradigms.
- prove correctness of programs.
- apply program verification and testing methodologies.
- understand formal analysis of algorithms, notations, and complexity classes.

3. ALGORITHM DESIGN

Introduction

To solve a problem on a computer, we need efficient algorithms and data structures that we then map to programs in a programming language of our choice, which also needs to be efficient. Algorithm design involves mapping the specifications of a problem, possibly in natural language, to an algorithmic pseudocode that can be universally understood. For better understanding later, it may be necessary to create a correctness proof, particularly if some steps of the algorithm are nontrivial.

Since there may be multiple algorithms for the same problem to choose from, the programmer will need some basis for the choice. Hence, it will also be necessary to augment the solution description with an analysis of resource requirements, which mostly translates to the running time and space. Over time, standard methodologies have emerged for all these steps of algorithm design, analysis, and correctness proofs. Although each algorithm is different, over the years, some “design patterns” or templates have also emerged for algorithm design methodologies. These apply to a large class of problems. Finally, standard measures of complexity help in comparing multiple algorithms for the same problem. Once the program is written, we need program verification techniques to ensure the program acts according to specifications. Rigorous testing techniques are then employed to test whether the program meets the requirements and to unearth any bugs.

3.1 Induction, Iteration, and Recursion

Iteration

“Iterations” are repetitive computations of a group of statements. They form fundamental building blocks of algorithms and programs, allowing them to be more compact. Many problems fundamentally depend on repetitive computations; hence, programming languages support various mechanisms for “loops”. These mechanisms differ in how they control the number of times a block of statements must be repeated and the location of the condition check in the code (Sebesta, 2016). Languages also provide different loop control mechanisms, such as “break” and “continue”, for example, that allow the programmer to decide the exact location of the control mechanism within the body of the loop. “Break” shifts the control out of the loop, whereas “continue” shifts it to the beginning of the loop.

The while loop

The “while loop” is a construct in which a loop is controlled using a test condition that evaluates to `True` or `False`. The condition is tested at the beginning of the loop.

The syntax for the while loop in Python is

Code

```
while(condition):
        statements
```

The statements are executed for as long as the condition is true.

The following Python code prints the natural numbers from 1 to 25. The test condition for the while loop fails when $n = 26$, and the loop then terminates.

Code

```
n = 1
while (n <= 25):
                n += 1
                print(n)
```

Note that if n had been initialized to 26, the while loop would not be executed at all.

The do-while loop

The “do-while” loop is a construct that tests a condition at the end of the loop rather than at the beginning. Python does not have a do-while loop.

The syntax in C and C++ is as follows:

Code

```
do {
        statements;
} while (condition);
```

The statements in the body of the do-while loop are executed until the condition is evaluated to be false. The do-while loop is always executed at least once, even if the condition is false throughout.

The for loop

In many programming languages, the “for loop” uses a counter to control the loop. In Python, the for loop is controlled by looping through any objects that are **iterable**(Goodrich et al., 2013). Lists, tuples, strings, and dictionaries are examples of iterables. Equivalent concepts also exist in other languages.

Iterable
An iterable is an object that allows iteration through a sequence of values. The concept exists in various programming languages.

The general syntax is

Code

```
for variable in sequence:
                statements
else:
                statements
```

The else part is optional and executed when the for loop exits normally. It will not execute when the exit is through a break statement.

The following prints all the natural numbers from 21 to 34, inclusively:

forloops.py

Code

```
for i in range(21,35):
   print(i)
```

The following prints the names of the fruits in the list `fruitBasket`:

Code

```
fruitBasket=["apple","banana","mango","cherry","kiwi"]
for i in fruitBasket:
    print(i)
```

The following lines all print 2, 4, 6, 8 in consecutive lines

Code

```
for i in (2,4,6,8):
          print(i)
for i in [2,4,6,8]:
          print(i)
for i in "2468":
          print(i)
```

The for loop in C-based languages has the following general form:

Code

```
for(expression; expression; expression)
                        statements
```

Here, the first expression is used for initialization and the second for the condition to be tested for the loop to continue. The third expression is used for any action at the end of the loop, such as incrementing the loop control variable. All expressions are optional.

User-controlled mechanisms

Programming languages also support loop control mechanisms wherein the exact location of the control mechanism within the body of the loop can be decided by the user. Python supports two such constructs: break and continue. Break allows the control to exit the loop, whereas continue allows the control to skip the rest of the statements in the loop body and return to the start of the loop.

For example, consider the following Python code fragment to print the odd numbers in the range $[501,1000)$ starting from 501 until it encounters an odd multiple of 37. It prints the sequence 501, 503, ..., 555.

break.py

Code

```
for i in range(501,1000,2):
   print(i)
   if(i % 37==0):
       break
```

The following Python code fragment prints all odd multiples of 37 in the range $[501,1000)$:

continue.py

Code

```
for i in range(501,1000,2):
   if(i % 37!=0):
       continue
   print(i)
```

Iterators

“Iterators” are user-defined functions that traverse a data structure in some sequence (Goodrich et al., 2013). Each time it is called, it returns another element of the data structure. For instance, Python allows the creation of iterators that walk through the elements of an iterable object.

In the following example, the Python code snippet creates an iterator for a list of integers and iterates through the list, printing them one by one:

iterator.py

Code

```
alist = list(range(1,21))
i = iter(alist)        #creates iterator
while (1):
   try:
       print(next(i)) #iterates through list
   except StopIteration:
       break
```

Of course, in Python, the for loop would have automated this process of creating an iterator for an object and invoking the next element repeatedly before calling the `StopIteration` exception (Goodrich et al., 2013).

Generators

"Generators" are an alternative to a traditional function and are suitable when we need the results one by one. Here is an example that generates the prime factors of a natural number in Python. Note the use of the "`yield`" construct instead of a "`return`".

generator.py

Code

```
def generatePrimeFactors(num):
   fact = 2
   while fact * fact <= num:
       if num % fact:
           fact += 1
       else:
           num //= fact
           yield fact
   if num > 1:
       yield num
```

This is used as follows to generate the prime factors of 3,000:

Code

```
for i in generatePrimeFactors(3000):
   print(i)
```

Recursion

"Recursion" is an elegant alternative to loops for generating repetition. A function makes one or more calls to itself, trying to express the solution to a problem in terms of solutions to smaller subproblems. Consider the following recursive variant of the Python function to compute the prime factors of a natural number `num`. It must be invoked as `primeFactors(num, 2)`, which returns the prime factors in a list.

primes.py

Code

```
 def primeFactors (num, fact):
   if num < fact*fact:
       return [num]
   if num % fact == 0:
       return [fact] + primeFactors (num // fact, 2)
   return primeFactors (num, fact + 1)
```

Above is an example of linear recursion since only one of the two calls to `primeFactors` in the body of the function will be executed. The number of such calls may be more than one.

The following example illustrates a case of binary recursion in Python. Fibonacci numbers are defined for all non-negative integers n as follows:

$$fib(n) = \begin{cases} n, 0 \leq n < 2 \\ fib(n-1) + fib(n-2), n \geq 2 \end{cases}$$

This definition leads to a straightforward binary recursive implementation.

binFib.py

Code
```
#Binary Recursive Fibonacci
def fib(n):
         if(n==0):
            return 0
         elif(n==1):
            return 1
         else:
            return(fib(n-1)+fib(n-2))
print(fib(6))
```

Although recursion is elegant, it must be used judiciously. Recursive calls have system overheads. Moreover, although a certain way of programming may be "natural", it may not be the most efficient. The above may be improved to a linear recursive version as follows.

linFib.py

Code
```
def linearFibonacci(n):
   #Returns F(n) and F(n-1)
   if (n <= 1):
       return (1,0)
   else:
       (current, prev) = linearFibonacci(n-1)
       return (current+prev, current)
```

The difference between the two implementations is significant. The binary variant runs in exponential time, whereas the linear version takes $O(n)$ time, as we shall see below.

Induction Proofs

Loop invariant
A loop invariant is a property that is true both before and after the execution of a loop.

"Mathematical induction" is a fundamental tool in the design and analysis of algorithms. In proving the correctness of algorithms, we often employ **loop invariants**(Sebesta, 2016). For iterative algorithms, the iterations provide a sequence on which induction can be naturally applied. For recursive algorithms, properties can be proved by applying induction to arguments of the recursive call. In data structure design, too, induction is used for proving properties of recursive structures like heaps or binary trees. Analysis of time or space complexity often uses recurrences, and induction is often useful in asymptotic solutions to recurrences. Induction takes two basic forms: weak and strong.

Weak induction

Suppose we need to prove that a property $P(n)$ is true for all non-negative integers $n \geq 0$. The steps are as follows:

- basis. Show that $P(0)$ is true.
- induction step. Show that for all $n \geq 1$, if $P(n-1)$ is true, then $P(n)$ is true.

Strong induction

In this case, to prove that a property $P(n)$ is true for all non-negative integers $n \geq 0$, the steps are as follows:

- basis. Show that $P(0)$ is true.
- induction step. Show that for all $n \geq 1$, if $P(k)$ is true for all $k < n$, then $P(n)$ is true.

Note that we may use a different basis condition depending on the problem.

3.2 Methods of Algorithm Design

Algorithm design is often guided by analysis in the quest for more efficient solutions. While each problem is different, some basic techniques are useful for a large class of problems.

A Simple Algorithm

A "simple algorithm" is the simplest one for solving the problem; it is usually an obvious one based on the problem statement directly. We consider the Maximum Contiguous Subarray Problem, which is defined as follows: We are given a sequence $A$of $n$integers $A[1\,.\,.\,n]$ and we need to find the largest sum possible in a contiguous subsequence $A[i\,.\,.\,j]$of A. This and similar problems arise in applications, such as bioinformatics, computer vision, and data mining (Brodal, 2013; Bentley, 2000).

For example, consider the sequence $A = [-6, \quad -22, \quad 1, \quad 6, \quad -5, \quad 3, \quad 4]$. Here, the maximum contiguous subsequence is $[1, \quad 6, \quad -5, \quad 3, \quad 4]$with a sum of 9.

A possible **brute-force algorithm** for this problem could simply be to compute the subarray sum for each possible pair (i, j) satisfying $0 \leq i \leq j \leq n-1$ and keep track of the maximum. Below is a Python implementation of this algorithm, with the sequence represented as a Python list.

Brute-force algorithm
This is a straightforward algorithm that typically adopts a simple approach like considering all possible cases.

maxContiguousBF.py

Code
```
def maxContiguousSubseq(A):
   maxSum = 0
   n = len(A)
   for i in range(0,n):
       for j in range(i,n):
           subseqSum = 0
           for k in range(i,j+1):
               subseqSum += A[k]
               maxSum=max(maxSum, subseqSum)
   print("maxSum =", maxSum)
```

The loop with index i is executed n times. The loop with index j is executed $n-i \leq n$ times. The loop with index k is executed $j-i+1 \leq n$ times. So, overall, this is an $O(n^3)$ algorithm for the problem.

Dynamic Programming

In many situations, problems have overlapping subproblems. Solving the overlapping subproblems independently entails wasted resources, such as computing time and space. "Dynamic programming" is an algorithm design technique that involves solving such overlapping subproblems only once and reusing the results. In the Maximum Contiguous Subarray Problem, note that `Sum(i, j)`, the sum for the subsequence `A[i..j]`, can be obtained from `Sum(i, j-1)`, the sum of the subsequence `A[i..j-1]`, by simply adding `A[i]`. The second sum need not be recomputed from scratch, but instead computed from the solution to the subproblem sum.

$$Sum(i,j) = \begin{cases} A[i] \text{if} j = i \, . \\ Sum(i, j-1) + A[j] \text{if} i < j < n \end{cases}$$

This leads to an improved algorithm because we optimize on space by not storing the partial sums. The Python code is displayed below.

maxContiguousDP.py

Code
```
def maxContiguousSubseqDP(A):
   maxSum = 0
   n = len(A)
   for i in range(0,n):
```

```
        subseqSum = 0
        for j in range(i,n):
            subseqSum += A[j] #Compute Sum(i,j)
            maxSum=max(maxSum, subseqSum)
    print("maxSumDP =", maxSum)
listA=[-6, -22, 1, 6, -5, 3, 4]
maxContiguousSubseqDP(listA)
```

The loop with index i is executed n times, and the loop with index j is executed $n-1 \leq n$ times. So, overall, this is an $O(n^2)$ algorithm for the problem, which is an improvement on the brute-force $O(n^3)$ approach.

Divide-and-Conquer

"Divide-and-conquer" is a widely used, and often efficient, design technique. It consists of three steps (Levitin, 2012):

1. The given problem is subdivided into two or more smaller subproblems.
2. The subproblems may be solved recursively or using a different algorithm.
3. To get a solution to the original problem, we combine the solutions to the smaller problems (the "conquer" step).

Let's design a divide-and-conquer algorithm for the Maximum Contiguous Subarray Problem (Bentley, 2000):

1. Divide the array into two parts.
2. Compute the sum of the Maximum Contiguous Subarray residing exclusively on the left.
3. Compute the sum of the Maximum Contiguous Subarray residing exclusively on the right.
4. Compute the sum of the Maximum Contiguous Subarray that crosses the boundary.
5. Return the maximum of the three sums computed.

The Python implementation is as follows:

maxContiguousDC.py

Code

```
def maxContiguousSubseqDC(A, low, high):
    if(low == high):          #single element
        return max(0,A[low]) #if negative return 0
    mid=(low+high)//2


    #find max crossing subsequence to the left
    subseqSum = 0
    maxLeftSum = 0
```

```
    for i in range(mid,low-1,-1):
        subseqSum += A[i]
        maxLeftSum=max(maxLeftSum, subseqSum)

    #find max crossing subsequence to the right
    subseqSum = 0
    maxRightSum = 0
    for i in range(mid+1,high+1):
        subseqSum += A[i]
        maxRightSum=max(maxRightSum, subseqSum)

    #find max subsequence exclusively to the left
    left = maxContiguousSubseqDC(A, low, mid)

    #find max subsequence exclusively to the right
    right = maxContiguousSubseqDC(A, mid+1, high)

    print("low, mid, high, left, right, maxLeft, maxRight", low, mid, high,
    left, right, maxLeftSum, maxRightSum)
    return(max(left, maxLeftSum+maxRightSum, right))

listA=[-6, -22, 1, 6, -5, 3, 4]
print("maxSumDC=", maxContiguousSubseqDC(listA,0,6))
```

Let the running time for this algorithm be expressed as $T(n)$, where n is the size of the input. We subdivide the problem into two parts and recurse on each. Finding the maximum crossing subsequences takes $O(n)$ time.

$$T(n) = \begin{cases} O(1), n \leq 1 \\ 2T\left(\frac{n}{2}\right) + O(n), n > 1 \end{cases}$$

This solves to $T(n) = O(n \log n)$ because there is $O(n)$ amount of work involved in each of $O(\log n)$ levels of recursion.

Greedy Algorithms

Consider an optimization problem that has an associated objective function $F()$. Among multiple candidate solutions, the goal is to find the one that maximizes or minimizes $F()$. We refer to the maximum or minimum value thus found as the “optimal value” and the candidate solution as an “optimal solution”. Whether “optimal” refers to the maximum or minimum depends on the specific problem. An optimal solution in this context is not necessarily unique. Solutions to optimization problems go through steps with choices at each step. A “greedy algorithm” always makes a locally optimal choice. Sometimes the locally optimal choice leads to a globally optimal solution. This works well for several practical problems.

Customers at a grocery store

Consider the following problem of n customers at a grocery store waiting to be served at a single counter. Customer j requires $c(j)$ units of time to be served. The total waiting time for customer j before being served is equal to the total serving time of customers served before customer j. Suppose the grocery store owner wants to minimize the total waiting time of the n customers. In what order should the customers be served to achieve this goal?

It turns out that if the customers are served in the order of non-decreasing $c(j)$, the optimal solution is achieved. Here, the store owner makes the greedy choice by choosing the customer with the minimum $c(j)$ (ties broken arbitrarily) among those still waiting as the next customer to be served. We illustrate this with an example:

Consider a problem instance with $n = 5$ customers.

Let $(c(1), c(2), c(3), c(4), c(5)) = (25{,}21{,}14{,}10{,}5)$.

Consider the schedule $(5{,}14{,}10{,}25{,}21)$.

Cumulative waiting times are $(0{,}5{,}19{,}29{,}54)$.

The sum of waiting times is $5 + 19 + 29 + 54 = 107$.

Now, consider an alternate schedule $(5{,}10{,}14{,}25{,}21)$.

Cumulative waiting times are $(0{,}5{,}15{,}29{,}54)$.

The sum of waiting times is $5 + 15 + 29 + 54 = 103$.

Finally, consider the greedy schedule $(5{,}10{,}14{,}21{,}25)$.

Cumulative waiting times are $(0{,}5{,}15{,}29{,}50)$.

The sum of waiting times is $5 + 15 + 29 + 50 = 99$.

Note that the cumulative waiting time decreases if a customer's position in the queue is exchanged with another customer with a higher service time who is ahead in the queue. The greedy schedule with a non-decreasing order of service times gives the minimal solution.

3.3 Correctness and Verification of Algorithms

Correctness of a Greedy Algorithm

Let us revisit the problem of customers at a grocery store waiting to be served by a single counter. The grocery store owner's goal is to minimize the total waiting time of the n customers. The question we must answer is: what order should the customers be served to achieve this goal? The proposed algorithm tries to achieve this goal by serving the customers in the order of non-decreasing $c(j)$. We prove below that this is correct, that is, if the customers are served in the order of non-decreasing $c(i), 0 \leq i < n$, the solution achieved is an optimal solution.

The condition is $c(1) \geq c(2) \geq \cdots \geq c(n)$.

The goal is to minimize the sum of the waiting time of all customers.

The claim is: The total waiting time is minimized if the customers are processed in the order.

$$S = (s(1), s(2), ..., s(n))\,.$$

We define this with $s(i) = c(i) \quad \text{for} \quad i \quad = 1, \quad . . . n\,.$

Proof

We prove the claim to be correct by the method of contradiction. The claim implies that $s(1) \leq s(2) \leq s(3) \leq ... \leq s(n)$. Let us assume that this order of serving customers, S, adopted by the greedy algorithm is incorrect and does not minimize the total waiting time. Let an optimal schedule minimizing the total waiting time for processing the customers be $T = (t(1), t(2), ..., t(n))$. As T is optimal and S is not, they must differ at one or more indices. Let i be the smallest index such that $t(i) \neq s(i)$:

$$s(1) = t(1), s(2) = t(2), ..., s(i-1) = t(i-1)$$

Since $t(i) \neq s(i)$, $t(i) = s(k)$ for some $k > i$ and $t(j) = s(i)$ for some $j > i$.

Since $s(1) \leq s(2) \leq s(3) \leq \cdots \leq s(n)$, we get $t(j) \leq t(i)$.

Let T' be the schedule obtained from T by swapping $t(i)$ and $t(j)$.

Since j moves $j-i$ places up and i moves $j-i$ places down in the schedule,

$\text{Wait}(T')\text{=}\text{Wait}(T) + t(j) \cdot (j-i) - t(i) \cdot (j-i)$

$\text{Wait}(T')\text{=}\text{Wait}(T) - (j-i) \cdot (t(i) - t(j))$

Since $j > i$ and $t(j) \leq t(i)$, $\mathrm{Wait}(T') \leq \mathrm{Wait}(T)$

$s(1) = t(1), s(2) = t(2), ..., s(i-1) = t(i-1), t(i) \neq s(i)$

$T' = (t'(1), t'(2), ..., t'(n))$

$s(1) = t'(1), s(2) = t'(2), ..., s(i-1) = t'(i-1), s(i) = t'(i)$

$\mathrm{Wait}(T') \leq \mathrm{Wait}(T)$

T matches S up to position $i - 1$.

T' matches S up to position i.

Thus, we can find schedules $T = T_{i+1}, T' = T_i, T_{i+1}, ..., T_n$ where T_j matches S up to position j.

$\mathrm{Wait}(S) = \mathrm{Wait}(T_n) \leq \mathrm{Wait}(T_{n-1}) \leq ... \leq \mathrm{Wait}(T_{i+1}) = \mathrm{Wait}(T)$.

Thus, $\mathrm{Wait}(S) \leq \mathrm{Wait}(T)$.

But, since T is an optimal schedule, $Cost(T) \leq Cost(S)$.

Hence, $\mathrm{Wait}(S) = \mathrm{Wait}(T)$, that is, the optimal schedule has the same cost as the greedy schedule, which we had assumed does not minimize the total waiting time. This is a contradiction. Therefore, we conclude that the greedy schedule must minimize the total waiting time.

The proof methodology used here is general and has been applied for many greedy algorithms. “Matroid theory” provides a mathematical basis to show that a greedy algorithm is correct by using a combinatorial structure called a matroid (Cormen et al., 2009). This has been used for many greedy algorithms but is not necessarily applicable to all.

Correctness of an Iterative Algorithm

Example

Consider the following Python function for computing factorials.

factorial.py

Code

```
def factorial(n):
   index = 0
   value = 1
   while(index < n):
```

```
        index += 1
        value *= index
    return value
```

We would like to prove the following statement $P(r)$ for each r,

$P(r)$: if the while loop executes r times, $r \geq 0$, $\text{value} = r!$.

Proof

The basis $P(0)$ is true. By definition $0! = 1$. Before the while loop executes, $\text{value} = 0! = 1$.

The induction step is as follows: Let $P(k-1)$ be true. Assume that if the while loop executes $r = k - 1$ times and $\text{value} = r! = (k-1)!$. The variable index tracks the number of iterations of the while loop. Thus, at this point $\text{index} = k - 1$. Then, in the next iteration index gets incremented to k and $\text{value} = \text{value} \cdot k = (k-1)! \cdot k$. Hence, $P(k)$ is true.

Correctness of a Recursive Algorithm

Example

Consider the following Python program which computes the highest power of a factor $\text{fact} \geq 2$ that divides a natural number $num \geq 2$.

Code

```
1. def powersOfFactor(num, fact):
2.   if (num < fact):
3.      return 0
4.   if(num == fact):
5.     return 1
6.   elif num % fact == 0:
7.       return(powersOfFactor(num//fact, fact)+1)
8.   return 0
```

Let us prove that this program is correct by using strong induction. Note that we assume that the following inequality always holds because of the nature of the problem: $\text{fact} \geq 2$.

Proof

This is achieved by strong induction on the first function argument num.

The basis is as follows: The function works correctly for $num = 0,1,2$. If $num = 0$ or 1, $num < \text{fact}$, hence the function correctly returns 0 in line 3. If $num = 2$, and $num < \text{fact}$, again the function returns a 0 in line 3, which is correct. If $num = \text{fact} = 2$, the function correctly returns a value of 1 (line 5).

The induction step is as follows: Suppose the function works correctly for all values of num satisfying $0 \leq num < k$. Now consider $num = k$.

Case 1: If $\mathrm{fact} > num$, the function will return 0, which is correct.

Case 2: If $\mathrm{fact} = num$, the function will return 1, which is correct.

Case 3: If $\mathrm{fact} < num$ and $num \,\%\, \mathrm{fact} = 0$, then fact divides num. Then, the highest power of fact that divides num is one more than the highest power of fact that divides num/fact.

Since $k/\mathrm{fact} < k$, according to the induction hypothesis, the function works correctly for $num = k/\mathrm{fact}$ and returns the highest power of fact that divides num/fact. Hence, the function correctly returns one more than the highest power of fact that divides num/fact (line 7).

Hence, by induction on num, the above arguments prove that the function correctly returns the highest power of fact that divides num for all integers $num \geq 0$ and all integers $\mathrm{fact} \geq 2$.

Loop Invariants

Loops are fundamental constructs in programming and proving them to be correct is of paramount importance in program verification. A standard technique used in such cases is the "loop invariant". We revisit the problem of Maximum Contiguous Subarray yet again. The problem can be optimally solved using Kadane's algorithm in $O(n)$ time (Bentley, 2000). Below is a Python implementation of Kadane's algorithm, which may seem unintuitive to start with:

maxContiguousOPT.py

Code

```
def maxContiguousSubseqOpt(A):
   maxSum = 0
   subseqSum = 0
   n = len(A)
   for i in range(0,n):
           subseqSum = max(subseqSum+ A[i], 0)
           maxSum = max(maxSum, subseqSum)
   print("maxSumOpt =", maxSum)
listA=[-6, -22, 1, 6, -5, 3, 4]
maxContiguousSubseqOpt(listA)
```

To see why this is correct, note that the variable $subseqSum$ tracks the maximum sum for a subsequence ending at the most recently processed position, which is $i-1$ at the start of the loop and i at the end. The variable $maxSum$ tracks the maximum sum for the entire processed subsequence, which is $A[0\,..\,i-1]$ at the start of the loop and $A[0\,..\,i]$ at the

end. The values of $subseqSum$ and $maxSum$ are both zero initially, which is correct by definition. If the value $A[i]$ when added to $subseqSum$ is negative, then $A[i]$ cannot be appended to any existing subsequence to create a maximum subsequence and no maximum subsequence ends at position i. Otherwise, $subseqSum$ is updated with the value of $A[i]$. The last line of the loop updates $maxSum$ with the new value of $subseqSum$ if the latter is greater. Thus, the algorithm correctly maintains the meanings associated with $subseqSum$ and $maxSum$ across loops. The formal proof can be done using mathematical induction on the loop variable i.

Program Verification

Today, information technology (IT) systems are increasingly dependent on complex software. Often, checking for faults via manual reviews, rigorous testing, and simulations is not enough. Correctness of a program has been traditionally viewed in three ways (Pratt & Zelkowitz, 2001):

1. Semantic modeling. Given a program, what are its specifications?
2. Correct-by-construction development. Given a specification, develop a program that is correct according to the specification.
3. Program verification. Does the behavior of a program match its specification?

Over the years, research in formal methods in software engineering has focused on the development of rigorous techniques for specification, development, and verification of software systems. Using rigorous specifications and verifying that the implementation meets the specifications can help to detect errors early or to eliminate them. As a limitation, note that formal verification methods only verify whether the system is correct according to the specification, but there is no guarantee that the specification itself is completely correct.

There are different approaches to formal verification (Almeida et al., 2011):

- proof tools. These include “automatic theorem provers,” which automatically construct proofs using axioms and rules of inference, and “proof assistants”, which are interactive theorem provers that can help analyze complex properties and prove expected behaviors based on theoretical deductions.
- model checkers. These use the program’s “state space”, the set of all possible variable states in the programme's memory. The system is specified using logic, and desired properties are validated. A counterexample is provided if a desired property is not valid. Model checkers suffer from the problem of state space explosion and do not scale well to large systems. One way this problem is circumvented is by using higher levels of abstraction. Another way is by using "bounded model checking" (Clarke et al., 2001). Bounded model checkers consider only those states that can be reached within a number of steps below a fixed bound.
- program annotation (Peled & Qu, 2003). These are logical properties to be verified that are placed in the code. These additional instructions are executed during the verification process. The additional code does not alter the behavior of the original program. A common usage is to add simple assertions as preconditions and postconditions to pieces of code.

Testing

Testing programs has two broad goals (Sommerville, 2016):

1. To demonstrate that a program behaves according to the requirements (validation testing)
2. To find inputs for which the program output is incorrect (defect testing)

In practice, commercial software goes through stages of testing including testing done during development, testing done at the time of release, and testing done by users. Testing usually involves both manual and automated processes.

Unit or component testing

Unit or component testing involves testing individual components in isolation. Components could include functions, classes, or class methods (Sommerville, 2016). It usually operates at the level of source files or single classes. A challenge faced in unit testing is that the behavior of the class being tested may depend on other classes that are not present. This requires the creation of mock objects that simulate the behavior of the more complex real objects they represent.

Integration testing

Integration testing involves integrating components in an almost realistic setting and subjecting the integrated system to testing. The goal here is to check that the individual pieces are compatible, integrate smoothly, and transfer data correctly through interfaces (Sommerville, 2016).

Release testing

Release testing is the process of testing a particular release of the software and is intended to ensure that the product or program is ready to be released for general consumption by external users before they receive the release (Sommerville, 2016).

Performance testing

The goal of performance testing is to verify that the system can operate and deliver an adequate service under the intended load. This is carried out after the system is fully integrated (Sommerville, 2016).

User testing

User testing is typically carried out by users and customers to experiment and provide feedback on a new system with the aim of ensuring that interaction with the software under scripted and unscripted conditions yields expected behaviors (Sommerville, 2016).

3.4 Efficiency (Complexity) of Algorithms

Faced with the task of choosing the best algorithm or data structure for a problem, a programmer often depends on the available efficiency measures, such as time and space complexity. Even if the algorithm is correct, if it consumes too much time or space it may not be feasible to deploy it in practice.

Thus, the running time is of interest, and, in particular, how the running time grows with the size of the input rather than the exact time, which could depend on the computational resources available to the user. For this, we count some fundamental steps of the algorithm, such as comparisons, arithmetic, and logic operations. Our analysis should yield the order of growth, typically in terms of the input size, and we then choose the algorithm based on this measure.

Asymptotic Complexity

Asymptotic upper bound

We define $O(g(n)) = \{f(n)$: There exist positive constants c and n_0 such that $f(n) \leq c \cdot g(n) \forall n \geq n_0\}$. The expression $f(n) = O(g(n))$ denotes the membership of $f(n)$ in the set $O(g(n))$ (Cormen et al., 2009).

Asymptotic lower bound

We define $\Omega(g(n)) = \{f(n)$: There exist positive constants c and n_0 such that

$0 \leq c \cdot g(n) \leq f(n) \forall n \geq n$. The expression $f(n) = \Omega(g(n))$ denotes the membership of $f(n)$ in the set $\Omega(g(n))$ (Cormen et al., 2009).

Asymptotic tight bound

We define $\Theta(g(n)) = \{f(n)$: There exist positive constants c_1, c_2, and n_0 such that $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \forall n \geq n_0\}$. The expression $f(n) = \Theta(g(n))$ denotes the membership of $f(n)$ in the set $\Theta(g(n))$ (Cormen et al., 2009).

Note that $f(n) = \Theta(g(n))$ if, and only if, $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ (Cormen et al., 2009).

An example follows. Suppose the running time of an algorithm in terms of its input is given as

$$T(n) = 5\mathrm{n}^2 - 20\mathrm{n} + 1$$

$$\Longrightarrow T(n) \leq 5\mathrm{n}^2 + n^2 + n^2 \leq 7\mathrm{n}^2 \forall n \geq 1$$

$$\Longrightarrow T(n) = O(n^2)$$

$T(n) = 5\mathrm{n}^2 - 20\mathrm{n} + 1$

$\Longrightarrow T(n) = n^2 + 4\mathrm{n}(n - 5) + 1$

$\Longrightarrow T(n) \geq n^2 \forall n \geq 5$

$\Longrightarrow T(n) = \Omega(n^2)$

Since $T(n) = O(n^2)$ and $T(n) = \Omega(n^2)$, $T(n) = \Theta(n^2)$

Little oh and little omega

These are used to describe upper and lower bounds which are strict.

We define $o(g(n)) = \{f(n)$: $\forall$ constants $c > 0$, $\exists$ constant $n_0 > 0$ such that

$f(n) < cg(n) \forall n \geq n_0\}$ (Cormen et al., 2009).

We define $\omega(g(n)) = \{f(n)$: $\forall$ constants $c > 0$, $\exists$ constant $n_0 > 0$ such that

$c \cdot g(n) < f(n) \forall n \geq n_0\}$ (Cormen et al., 2009).

Properties based on limits

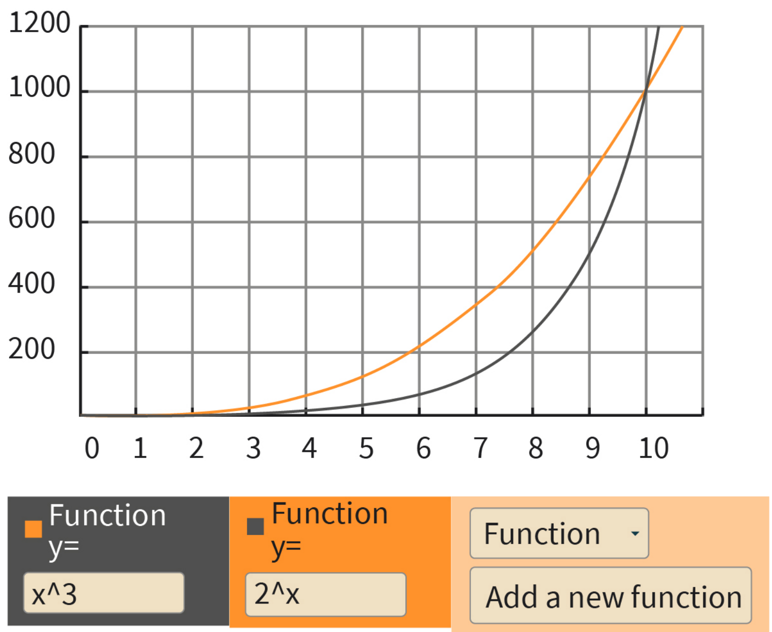
Using limits provides an alternative way to determine the above memberships (Cormen et al., 2009).

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \Longrightarrow f(n) = O(g(n))$$
$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \Longrightarrow f(n) = \Omega(g(n))$$
$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = c > 0 \Longrightarrow f(n) = \Theta(g(n))$$

Asymptotic comparison

When faced with a choice of algorithms, we choose the asymptotically faster algorithm. Note that the asymptotic complexity captures the practical notion that we are interested in comparisons for sufficiently large n. A case in point is the comparison between the functions $f(n) = n^3$ and $g(n) = 2^n$. Notice that while $g(n) < f(n)$ for $1 < n < 10$, $f(n)$ is asymptotically smaller.

Figure 16: Complexity Comparison



Source: Created on behalf of IU (2022).

Common complexity measures

Certain asymptotic complexity measures occur frequently in algorithm analysis. These are listed in order of increasing complexity for sufficiently large n:

- constant: $O(1)$
- logarithmic: $O(\log n)$
- log: $O(\log \log n)$
- linear: $O(n)$
- n-logn: $O(n \cdot \log n)$
- quadratic: $O(n^2)$
- cubic: $O(n^3)$
- polynomial: $O(n^k)$
- exponential: $O(2^n)$
- factorial: $O(n!)$

Polynomial Time Solvability

Many problems we encounter are solvable in polynomial time, that is, there is an algorithm for the problem that runs in time bounded by $O(n^k)$, where n is the size of the input, for some constant $k \geq 0$. One natural question that arises is whether all problems are, in fact, solvable in polynomial time. In the early days of the study of algorithmic complexity, it was observed that, although several problems are solvable within a time that is a low-

degree polynomial in n, such solutions were elusive for many problems. So, it seems there is some fine line dividing problems that are solvable in polynomial time from those that are not. There are two complexity classes of utmost importance:

1. The class P. All problems in this class are solvable in polynomial time.
2. The class NP (NP-complete). No polynomial time algorithms are known for these problems. Moreover, no one has been able to prove that such algorithms do not exist (Cormen et al., 2009). Also, if any one of these problems is solvable in polynomial time, all of these problems would be solvable in polynomial time!

The $P \neq NP$ question is one of the long-standing open problems in computer science. NP-complete problems are important since many of them occur in real-life scenarios. The knowledge that a problem is NP-complete, and thus does not have a known polynomial time solution, lets the algorithm designer attempt other means like heuristics and approximation algorithms to get reasonable solutions to the problem (Cormen et al., 2009).

SUMMARY

Well-known and fundamental algorithm design techniques include divide-and-conquer, greedy algorithms, and dynamic programming. These techniques apply to a large class of problems.

As an example, the Maximum Contiguous Subarray sum can be found using a simple brute-force approach, while improved algorithms using dynamic programming and divide-and-conquer methodologies can be designed.

The greedy technique can be applied to multiple domains, for example, to a scheduling problem.

For correctness proofs, mathematical induction. We proved the correctness of iterative and recursive algorithms. The proofs employed both weak and strong variants of induction. There are specialized techniques for greedy algorithms that show the correctness of a greedy solution by the successive transformation of any other optimal solution to the greedy solution and proving that the greedy solution is no worse than the optimal. We studied the correctness of the greedy scheduling algorithm using the same framework.

Having mapped the algorithm to a program, we need to verify and test it. Formal verification of programs involves formally checking if the program matches its specification. Techniques include proof tools, model checkers, and program annotations. Program testing goals include validation testing to check if the program meets the requirements and

defect testing to unearth bugs. These are accomplished through unit or component testing, integration testing, and release testing. The program is also tested for performance.

UNIT 4

BASIC ALGORITHMS

STUDY GOALS

On completion of this unit, you will be able to …

- implement algorithms for traversal and linearization of trees.
- apply basic algorithms for searching.
- differentiate between various algorithms for sorting.
- utilize the trie data structure for searching for a word in a string.
- apply various hashing techniques to search problems.
- understand fundamental algorithms for pattern recognition.

4. BASIC ALGORITHMS

Introduction

Of the many algorithms that we encounter, some are ubiquitous in practical applications. These common algorithms often serve as fundamental building blocks in algorithmic solutions to more complex problems.

Trees are useful for representing acyclic relationships and connectivity information in numerous applications. In many such applications, it is necessary to visit all the nodes and process them in some systematic order.

Whether we are arranging a list of names in a directory, a list of books in a bibliography, a list of files in a folder on our desktop, or applying painter's algorithm in computer graphics to render objects in reverse order of their distances from a viewer, we use a sorting algorithm in the process.

In the twenty-first century, we are faced with a data deluge and are thus building applications that are increasingly data dependent. Therefore, it is imperative that we be able to locate data efficiently when needed. Hence, search algorithms are fundamental to many such applications.

Whether the data are ordered or not is a basic distinction that we need to make while deciding on the type of search algorithm to apply. We can find a word quickly in a dictionary because the words are ordered. Many hotels and restaurants across the world have a valet parking service, wherein a valet parks the customer's car. When the car needs to be retrieved later, the valet knows exactly where it is.

Hash algorithms try to generalize this idea of finding objects by computing a data item's location from a table or a series of possible locations.

Text processing remains a major application area today, despite the increase in multimedia content. Locating a series of words in a preprocessed text is common to many applications. Data structures like "tries" support such string searches. On the other hand, pattern matching algorithms solve a complementary problem by preprocessing a pattern to speed up searching in a text document.

We will be looking at some fundamental algorithms within the following categories:

- tree traversal algorithms
- searching algorithms including linear search, binary search, and hashing
- basic sorting algorithms
- string-based algorithms

4.1 Traversing and Linearization of Trees

In many applications, such as natural language processing (NLP), we need to visit, list, or print the vertices of a tree in some required order (Filippova & Strube, 2009). These problems are classified under "linearization" or "tree traversal" problems. We assume that the trees are **rooted**. In our examples, the traversal problems merely visit the node and print the data contained in the node. In applications, other complex computations may replace the simple print.

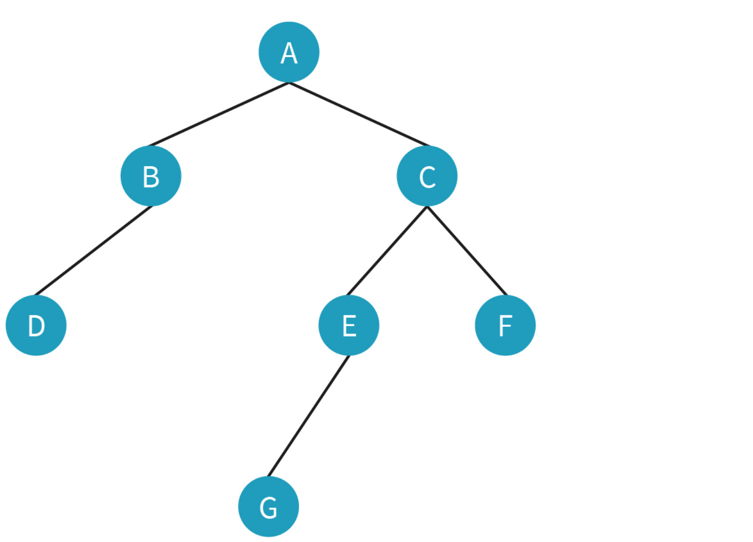
Rooted
Trees that are rooted have a designated vertex as their root.

Representation

We consider a list-of-lists representation of trees in Python (Miller & Ranum, 2013). In this representation, the tree is a Python list T. Although this representation can be generalized to represent any tree, we illustrate its usage on binary trees here. The root is represented by $T(0)$. The left subtree is a Python list $T(1)$ and the right subtree is another Python list $T(2)$.

For example, consider the following binary tree.

Figure 17: Binary Tree



Source: Created on behalf of IU (2022).

Below is a Python list-of-lists representation of this tree:

Code

```
aTree=['A',              #Root
      ['B',             #Left Subtree
       ['D',[],[]],
       []],
      ['C',             #Right Subtree
```

```
        ['E',
         ['G',[],[]],
         []],
        ['F',[],[]]
       ]]

def treeRoot(aTree):
   if(aTree):
       return aTree[0]

def leftSubTree(aTree):
   if(aTree):
       return aTree[1]

def rightSubTree(aTree):
   if(aTree):
       return aTree[2]
```

We can print the root, left subtree, and right subtree:

Code

```
print(treeRoot(aTree))
print(leftSubTree(aTree))
print(rightSubTree(aTree))
```

The following is printed:

Code

```
A
['B', ['D', [], []], []]
['C', ['E', ['G', [], []], []], ['F', [], []]]
```

In-order Traversal

In inorder traversal, we recursively perform an in-order traversal of the left subtree, followed by a visit to the root node. This is followed by a recursive in-order traversal of the right subtree (Cormen et al., 2009). The Python implementation is shown below.

Code

```
def inorder(aTree):
   if aTree:
       inorder(leftSubTree(aTree))
       print(treeRoot(aTree))
       inorder(rightSubTree(aTree))
```

If we invoke `inorder(atree)` with the tree above, the characters stored in the nodes are printed in the following order: D B A G E C F.

Preorder Traversal

In preorder traversal, we visit the root node first. This is followed by recursive preorder traversals of each of the subtrees (Cormen et al., 2009). While this applies to any tree, we illustrate it for a binary tree below.

Code

```
def preorder(aTree):
   if aTree:
       print(treeRoot(aTree))
       preorder(leftSubTree(aTree))
       preorder(rightSubTree(aTree))
```

For our example, a call to `preorder(atree)` prints the characters stored in the nodes in the following order: `A B D C E G F`.

Postorder Traversal

In postorder traversal, we first visit the subtrees from the leaves upwards to the root. This is followed by a visit to the root node (Cormen et al., 2009). A Python implementation is shown below. Like preorder traversal, this can also be extended to other types of trees.

Code

```
def postorder(aTree):
   if aTree:
       postorder(leftSubTree(aTree))
       postorder(rightSubTree(aTree))
       print(treeRoot(aTree))
```

For the example above, a call to `postorder(atree)` prints the characters stored in the nodes in the following order: `D B G E F C A`.

Breadth-First Traversal

The breadth-first traversal (BFS) is also called "level-order traversal" since the nodes are visited level-by-level starting from the root. Within a level, the nodes may be visited in any order (Goodrich et al., 2013). A Python implementation is displayed below.

Code

```
def bfSearch(aTree):
   if aTree:
       qList=[aTree]
       while qList:
```

```
        nextNode = qList.pop(0)
        if(nextNode):
            print(treeRoot(nextNode))
            qList.append(leftSubTree(nextNode))
            qList.append(rightSubTree(nextNode))
```

For our example, a call to `bfSearch(aTree)` prints the characters stored in the nodes in the following order: `A B C D E F G`.

4.2 Search Algorithms

Searching is a fundamental problem in computer science, and problems arising in many applications can be formulated as search problems. In a simple generic instance of such a problem, we have a table of records. Each record is a collection of attribute values. One such attribute is the search key. For a user-defined search key value x, the goal is to find a record whose key value is exactly x. For simplicity, we assume that each element consists of only the corresponding key value. We also assume a Python list is the storage structure for the table.

Sequential Search

In a linear or sequential search, we walk through the list, comparing each element in turn to the user-given key value x, until we find an element equal to x, or until we reach the end of the list. A Python implementation is as follows:

Code

```
def linearSearch(numList, keyValue):
   index = 0
   listLen = len(numList)
   while(index < listLen):
       if(keyValue == numList[index]):
           return index
       index += 1
   return -1
```

The above implementation implicitly assumes that the list is unordered. If the list being searched is ordered, we can take advantage of this by terminating the search early, that is, upon finding a value in the list greater than the key being searched for.

Code

```
def orderedLinearSearch(numList, keyValue):
   index = 0
   success = False
   stop = False
   listLen = len(numList)
```

```
    while index < listLen and not success and not stop:
        if(keyValue == numList[index]):
            success = True
        else:
            if(numList[index] > keyValue):
                stop = True
            else:
                index+=1
    return success
```

A linear search takes $O(n)$ time in the worst case. On an ordered list, the unsuccessful searches are faster than in the unordered case when there is an early termination. However, it is still $O(n)$ in the worst case.

Binary Search

For an ordered sequence, there is a better algorithm to search for a key than linear search. A binary search is based on gradual refinement of the possible interval of indices within which we need to search. The algorithm first compares the user-defined search key `keyValue` with the middle element. If they are equal, it terminates successfully, returning the index of the middle element. If `keyValue` is larger than the middle element, the lower half of the list is removed from consideration. If `keyValue` is smaller than the middle element, the upper half of the list is removed from consideration. In either case, we continue with another iteration. Since the size of the interval within which we need to search is halved in each iteration, the algorithm either terminates successfully or the size of the interval is reduced to one; thus, the algorithm has $O(\log n)$ iterations. The Python code is given below.

Code

```
def binarySearch(numList, keyValue):
    left = 0
    right = len(numList) - 1

    found = -1
    while left <= right:
        mid = (left + right) // 2
        if numList[mid] == keyValue:
            found = mid
            break
        else:
            if keyValue < numList[mid]:
                right = mid - 1
            else:
                left = mid + 1
    return found
```

Consider the list `aList` = [-17, -1, 12, 13, 27, 45, 57, 82]. The call `binarySearch(aList, 13)` returns 3 since the key 13 is in index position 3. The call `binarySearch(aList,28)` returns -1 because 28 is not present.

Binary search is a divide-and-conquer algorithm whose running time satisfies the recurrence

$$T(n) = \begin{cases} 1 \text{if} n = 1 \\ T\left(\frac{n}{2}\right) + 1 \text{otherwise.} \end{cases}$$

This solves to $T(n) = O(\log n)$.

4.3 Sorting Algorithms

Given a table of n elements, $x_1, x_2, ...x_n$, the sorting problem involves finding a permutation $(\pi(1), \pi(2), ..., \pi(n))$ of the integers $(1,2, ..., n)$ such that $x_{\pi(1)} \leq x_{\pi(2)} ... \leq x_{\pi(n)}$. Sorting has many applications, and many algorithms require sorting as a preprocessing step. Some broad categories of applications include the following (Knuth, 1998):

- solution to the "togetherness" problem. Sorting helps in grouping items with the same key value together.
- matching two sets of items. Comparisons become easier if the items are sorted.
- searching for information by key values. For instance, a binary search is only applicable on a sorted sequence.

Insertion Sort

Insertion sort addresses the problem of inserting a new element into a subsequence of elements that is already sorted. Assuming that the subsequence is already sorted in non-decreasing order, the algorithm starts from the end of the subsequence and moves backward, looking for the correct place to insert the new element (Cormen et al., 2009). If the input is stored in an array `aList`, the idea is successively applied to `aList[0..i]` for `0≤ i ≤ n−2`. Once the subsequence `aList[0..i]` is sorted, we try to insert `aList[i+1]` at an appropriate position, shifting elements to make space. A Python implementation is as follows:

Code

```
def insertionSort(aList):
    seqLen = len(aList)
    for index in range(1, seqLen):
        toInsert = aList[index]
        j = index
        while j > 0:
            if(toInsert >= aList[j-1]):
                break
```

```
            aList[j] = aList[j-1]
            j -= 1
        aList[j] = toInsert

aList = [12,3,22,44,15,13,7,45,77,33]
insertionSort(aList)
print(aList)
```

Figure 18: Insertion Sort

- [3, 12, 22, 44, 15, 13, 7, 45, 77, 33]
- [3, 12, 22, 44, 44, 13, 7, 45, 77, 33]
- [3, 12, 22, 22, 44, 13, 7, 45, 77, 33]
- [3, 12, 15, 22, 44, 13, 7, 45, 77, 33]
- [3, 12, 15, 22, 44, 44, 7, 45, 77, 33]
- [3, 12, 15, 22, 22, 44, 7, 45, 77, 33]
- [3, 12, 15, 15, 22, 44, 7, 45, 77, 33]
- [3, 12, 13, 15, 22, 44, 7, 45, 77, 33]

Source: Created on behalf of IU (2022).

The figure illustrates some intermediate steps of "insertion sort" on an example. The subsequence `[3, 12, 22, 44]` is already sorted. The algorithm first tries to insert `15` into this subsequence. The positions where the algorithm tries to place the number are circled. This is followed by insertion of `13` into the subsequence `[3,12,15,22,44]`. Insertion sort takes $O(n^2)$ comparisons and $O(n^2)$ exchanges in the worst case, where n is the size of the input. The number of comparisons can be reduced to $O(n \log n)$ by using a binary search to locate the position where the insertion would take place. The overall running time is still dominated by the number of exchanges and hence is $O(n^2)$.

Bubble Sort

In a "bubble sort," we make a pass through the sequence comparing consecutive elements and swapping them if they are not in order. After the first pass, the largest element ends up in the last position. If we repeat the process, after the second iteration, the second largest element ends up in the penultimate position. If we repeat this $n-1$ times, where n is the number of elements, the array will be sorted. Also, if during an iteration we notice that no interchanges take place, we can conclude that the sequence is already in order and terminate the algorithm (Cormen et al., 2009). A Python implementation is as follows:

Code

```
def bubbleSort(aList):
    seqLen = len(aList)
    swapped = True
    for lastIndex in range(seqLen-1, 0, -1):
        if not swapped:
            break
```

```
        swapped = False
        for k in range(0, lastIndex):
            if aList[k] > aList[k+1]:
                aList[k],aList[k+1]=aList[k+1],aList[k]
                swapped = True
aList = [12,3,22,44,15,13,7,45,77,33]
bubbleSort(aList)
print(aList)
```

Figure 19: Bubble Sort

- [3, 12, 15, 13, 7, 22, 44, 33, 45, 77]
- [3, 12, 13, 15, 7, 22, 44, 33, 45, 77]
- [3, 12, 13, 7, 15, 22, 44, 33, 45, 77]
- [3, 12, 13, 7, 15, 22, 33, 44, 45, 77]
- [3, 12, 7, 13, 15, 22, 33, 44, 45, 77]
- [3, 7, 12, 13, 15, 22, 33, 44, 45, 77]

Source: Created on behalf of IU (2022).

The figure shows some intermediate steps of running a bubble sort on an example. Adjacent pairs of elements to be exchanged are marked. Also marked are the "locked" elements, which will no longer be moved because they are already sorted. Bubble sort takes $O(n^2)$ comparisons and $O(n^2)$ exchanges in the worst case, where n is the size of the input.

Selection Sort

"Selection sort" is similar to bubble sort in that the i-th largest element is located in the i-th iteration and moved to its correct destination. It differs from bubble sort in that selection sort performs exactly one exchange per iteration. It locates the element to be moved first and moves it to its correct destination with a single swap (Goodrich et al., 2013). A Python implementation is depicted below:

Code

```
def selectionSort(aList):
    seqLen = len(aList)
    for lastIndex in range(seqLen-1, 0, -1):
        maxIndex =0
        for k in range(1, lastIndex + 1):
            if aList[k] > aList[maxIndex]:
                maxIndex = k
        aList[lastIndex], aList[maxIndex] \
            = aList[maxIndex],aList[lastIndex]
aList = [12,3,22,44,15,13,7,45,77,33]
selectionSort(aList)
print(aList)
```

```
aList = [12,3,22,44,15,13,7,45,77,33]
selectionSort(aList)
print(aList)
```

Figure 20: Selection Sort

- [12, 3, 22, 33, 15, 13, 7, 44, 45, 77]
- [12, 3, 22, 7, 15, 13, 33, 44, 45, 77]
- [12, 3, 13, 7, 15, 22, 33, 44, 45, 77]
- [12, 3, 13, 7, 15, 22, 33, 44, 45, 77]
- [12, 3, 7, 13, 15, 22, 33, 44, 45, 77]
- [7, 3, 12, 13, 15, 22, 33, 44, 45, 77]
- [3, 7, 12, 13, 15, 22, 33, 44, 45, 77]

Source: Created on behalf of IU (2022).

The figure above shows some of the steps in running a selection sort on an example. Elements to be exchanged are circled. Note that, unlike bubble sort, selection sort exchanges pairs of elements that may or may not be adjacent. Selection sort takes $O(n^2)$ comparisons and $O(n)$ exchanges in the worst case, where n is the size of the input.

Quicksort

Quicksort is one of the most popular sorting algorithms. It works by choosing a pivot element p and partitioning the sequence into two groups of elements: the elements $x \leq p$ and elements $x \geq p$. The algorithm then recurses into the two partitions. A Python implementation using the first element of the subsequence being sorted as the pivot is given below:

Code

```
def partition(aList, left, right):
        pivot = aList[left]
        i=left + 1
        j=right
        while True:
                while (i <= j) and (aList[i] <= pivot):
                        i+=1
                while (i <=j) and (aList[j] >= pivot):
                        j-=1
                if(i <= j):
                        aList[i],aList[j] = aList[j],aList[i]
                else:
                        break
        aList[left],aList[j]= aList[j],aList[left]
        return j

def qSort(aList, left, right):
```

```
        if(left >= right):
                return
        partIndex = partition(aList, left, right)
        qSort(aList,left,partIndex-1)
        qSort(aList,partIndex+1,right)

def quickSort(aList):
        seqLen = len(aList)
        qSort(aList, 0, seqLen-1)

aList = [12,3,22,44,15,13,7,45,77,33]
quickSort(aList)
print(aList)
```

Figure 21: Quicksort

- [12, 3, 22, 44, 15, 13, 7, 45, 77, 33] Choose pivot
- [7, 3, 12, 44, 15, 13, 22, 45, 77, 33] Partition
- [3, 7, 12, 44, 15, 13, 22, 45, 77, 33] Sort left
- [3, 7, 12, 13, 15, 22, 33, 44, 45, 77] Sort right

Source: Created on behalf of IU (2022).

The figure shows some steps of a quicksort on an example. If the pivot element always creates a balanced partition, the recurrence for the running time $T(n)$ is as follows, with a and b constants:

$$T(n) = \begin{cases} a \text{if} n = 1 \\ T\left(\frac{n}{2}\right) + bn \text{otherwise.} \end{cases}$$

This solves to $T(n) = O(n \log n)$.

Median
The median is the middle element of a set if *n* is odd, and one of the two middle elements if *n* is even.

However, the partition may not always be balanced, and quicksort has a worst-case running time of $O(n^2)$. We could get an $O(n \log n)$ worst-case algorithm if we could always generate equal-sized partitions, which is theoretically possible by using the $O(n)$ **median**-finding algorithm (Cormen et al., 2009). However, this algorithm is complex and is never used in practical scenarios. Using a random pivot, however, the $O(n \log n)$ expected running time can be achieved for a randomized quicksort algorithm (Cormen et al., 2009). However, the generation of pseudo-random numbers is an expensive process and slows down the algorithm. So, a compromise often used in practice is to use the median of the three elements (Miller & Ranum, 2013).

Merge Sort

Like quicksort, mergesort is also a divide-and-conquer algorithm. The sequence is divided into two equal parts, which are then sorted recursively. The two sorted subsequences are then merged to create a sorted version of the original sequence (Cormen et al., 2009). The Python implementation below first defines the merge function for merging two sorted Python lists. The `merge` function is invoked within the recursive `mergeSort` function.

Code

```
def merge(A,B,C):
        a=b=0
        la, lb, lc = len(A), len(B), len(C)
        while(a+b < lc):
                if((b==lb) or ((a < la) and (A[a]<B[b]))):
                        C[a+b],a,b=A[a],a+1,b #Select from A
                else:
                        C[a+b],a,b = B[b],a,b+1 #Select from B
        return C

def mergeSort(aList):
        seqLen = len(aList);
        if seqLen <= 1:
                return
        mid = seqLen//2
        lower = aList[:mid] #Copy lower half
        upper = aList[mid:] #Copy upper half
        mergeSort(lower) #Sort lower half
        mergeSort(upper) #Sort upper half
        aList = merge(lower,upper,aList)

aList = [12,3,22,44,15,13,7,45,77,33]
mergeSort(aList)
print(aList)

bList = [3,12,15,22,44,7,13,33,45,77]
merge(bList[:5],bList[5:10],bList)
print(bList)
```

Figure 22: Merge Sort

- [12, 3, 22, 44, 15, 13, 7, 45, 77, 33]
- [12, 3, 22, 44, 15, 13, 7, 45, 77, 33]
- [3, 12, 22, 44, 15, 13, 7, 45, 77, 33]
- [3, 12, 22, 44, 15, 13, 7, 45, 77, 33]
- [3, 12, 22, 15, 44, 13, 7, 45, 77, 33]
- [3, 12, 15, 22, 44, 13, 7, 45, 77, 33]
- [3, 12, 15, 22, 44, 7, 13, 45, 77, 33]
- [3, 12, 15, 22, 44, 7, 13, 45, 77, 33]
- [3, 12, 15, 22, 44, 7, 13, 45, 33, 77]
- [3, 12, 15, 22, 44, 7, 13, 33, 45, 77]
- [3, 12, 15, 22, 44, 7, 13, 33, 45, 77]
- [3, 7, 15, 13, 15, 22, 33, 44, 45, 77]

Source: Created on behalf of IU (2022).

The figure shows steps of merge sort applied to an example. The shaded subsequences are being merged into bigger subsequences. Merge sort creates almost balanced partitions, where the sizes of the two partitions differ by at most one. Its running time is $O(n \log n)$ in the worst case.

Using the Spyder Integrated Development Environment

Integrated development enivronment
This is an application that combines text-editors and language analysis with a running and debugging environment.

The “Scientific PYthon Development EnviRonment”, or Spyder **integrated development enivronment** (IDE), is a development environment for the Python language that is free, open-source, interactive, and powerful. It has advanced features for interactive testing, editing, debugging, and introspection.

Useful features of Spyder include the following:

- There is an IPython (Qt) console as an interactive window.
- The console can also display plots inline.
- The user may execute code snippets from the editor in the console.
- Files in the editor can be parsed partially or fully.
- Visual warnings about potential errors are provided.
- Step-by-step execution is possible.
- There is a variable explorer to show attributes of variables, such as value and size.

Using the Spyder IDE to step through the code, one can study the sorting algorithms in detail. In the figure, the contents of the Python list being sorted are shown in the variable explorer (top-right pane) and are also printed in the Python console (bottom-right pane). The editor is on the left pane.

Figure 23: Spyder ID

Source: Prosenjit Gupta (2022), based on Spyder IDE (2021).

4.4 Search in Strings

We consider the following broad problem: How can a given set of strings S be stored efficiently, such that for a given query string q, it can be quickly determined whether q is in S. An example is a user searching for a specific word in a set of words in a fixed text. Additional queries of interest are prefix queries wherein we search for all words that start with the query prefix. Here, the text will be preprocessed to make the searches faster (Goodrich et al., 2013).

Tries

Also known as digital search trees, “tries” are important data structures in information retrieval. Instead of a search method based on comparisons between elements, tries attempt to take advantage of the representations of the elements as a sequence of characters or digits (Goodrich et al., 2013).

Standard tries

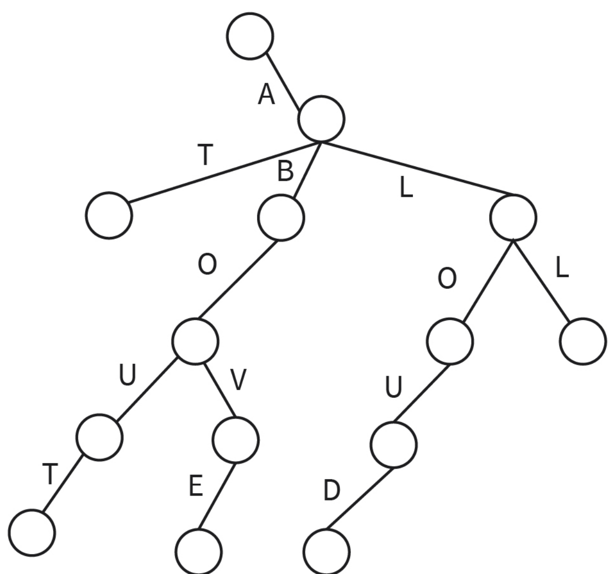
Let Σ be an alphabet. Let S be a set of strings from Σ with total length n satisfying the **prefix property**. We define a trie over S to be a tree satisfying the following properties (Goodrich et al., 2013):

Prefix property
The prefix property states that no string is a proper prefix of another.

- Each edge is labeled with a character from Σ.
- Each node has, at most, $|\Sigma|$ children.
- Edges connecting a node to its child nodes are all labeled differently.
- The number of leaf nodes is exactly $|S|$.
- Each leaf node v is associated with a string that is the concatenation of the characters on the path from the root to v.

- The total number of nodes in the trie is $n+1$.
- The height of the trie is the same as the size of the longest string in S.

Figure 24: A Trie of Some English Words



Source: Created on behalf of IU (2022).

A Python implementation follows.

Code

```
class Trie:
  def __init__(self):
    self._top = dict() #Create top level dictionary

  def buildTrie(self,aList):
    for word in aList:
      d = self._top
      for letter in word:
        if letter not in d:#no entry for letter
          d[letter] = dict() #create entry
        d = d[letter]#descend subtree by letter

  def searchTrie(self,word):
    d = self._top
    for letter in word:
      if letter not in d:#no entry for letter
        print("Not Found")
        return False
      d = d[letter]#descend subtree by letter
    print("Match Found")
    return True
```

```
  def printTrie(self):
    print(self._top)


aList = ["all","aloud","above","at","about"]
trial = Trie()
trial.buildTrie(aList)
trial.printTrie()
trial.searchTrie("aloud")
trial.searchTrie("albeit")
trial.searchTrie("abo")
```

The output is:

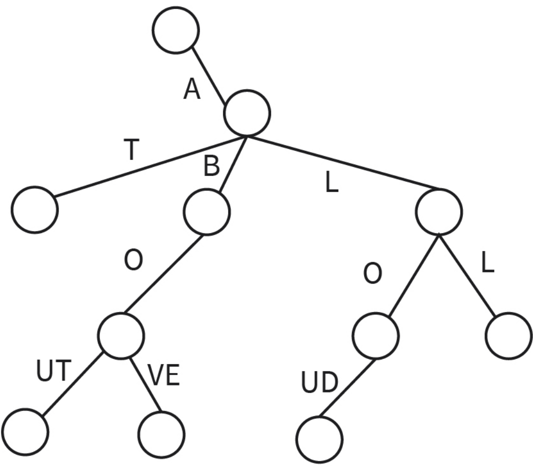
Code

```
{'a': {'l': {'l': {}, 'o': {'u': {'d': {}}}}, 'b': {'o': {'v': {'e': {}},
'u': {'t': {}}}}, 't': {}}}
Match Found
Not Found
Match Found
```

Other Structures

Searching on a set of strings can also use standard search algorithms, such as linear search and binary search. We need to define the comparison operator suitably. In Python, the usual operators <, >, ==, >=, and <= work with strings in the sense of lexicographic comparison. To apply binary search on a set of strings stored as a Python list, we can sort them lexicographically using any of the standard sorting algorithms and then apply binary search. Likewise, search structures like hash tables and binary search trees can be used with strings just as they are used with numeric data (Cormen et al., 2009). There is a modified trie structure called Patricia trie that uses a simple compression idea to reduce a redundant chain of edges into a single edge (Goodrich et al., 2013). The Patricia trie takes $O(|S|)$ space as opposed to the $O(n)$ space required by the standard variant, where n is the total size of all the strings. The Patricia trie for our example is shown below.

Figure 25: Patricia Trie Example



Source: Created on behalf of IU (2022).

4.5 Hash Algorithms

Dictionary
A dictionary is an abstract data type that supports insert, delete, and search operations.

Being efficient and easy to implement, hash tables are a popular structure for **dictionaries**. The algorithms for search queries are content-based as opposed to being comparison-based. The data are stored in locations that are computed by simple functions using the data themselves and are typically based on one or more attributes of the values called keys (Cormen et al., 2009).

Hashing is the mapping of keys to locations of a one-dimensional array, which we will refer to as the hash table T of size m. The mapping is computed by a hash function. If K is the set of keys, the hash function h maps $k \in K$ to $h(k) \in 0,1, ..., m-1$. A "collision" is said to occur if two keys map to the same location: $k_1 \in K, k_2 \in K, h(k_1) = h(k_2), k_1 \neq k_2$.

The basic questions we encounter when designing a hashing scheme from a source of n elements to a table of m locations are (Knuth, 1998):

- What should be the hash function?
- What should be the collision resolution algorithm?

The pair (hash function and collision resolution algorithm) together define a hashing scheme. Note that in hashing, the hash function generates the key values, which are table indices. The search algorithm simply looks up the table at those indices. The insert and delete operations also need to search with the key first and make use of the same hash function.

Hash Functions

Two desirable properties of hash functions are that they should be (a) easy to compute and (b) able to distribute the keys into table locations with approximately equal probability (Cormen et al., 2009). In practice, the distribution is difficult to estimate.

Division method

If there are m locations in the hash table numbered $0\,.\,.\,m-1$, a simple hash function is $h(k) = kmodm$. This is called the "division" method. This can be computed quickly, and the distribution of keys into locations is well-spread for m prime.

Multiplication method

In the "multiplication" method, we define $h(k) = \lfloor k\Theta mod 1 \rfloor$, where $0 < \Theta < 1$ and $\lfloor x \rfloor$ is the largest integer greater than or equal to x. Results indicate that a value of $\Theta = (\sqrt{5} - 1)/2$ and $\Theta = 1 - (\sqrt{5} - 1)/2$ works well (Knuth, 1998).

Universal hashing

One potential problem with hashing is that if someone chooses all or several keys such that $h(k)$ is the same for each key, severe collision and consequent performance degradation takes place. To counter that, the universal hashing scheme chooses a hash function randomly from a collection of **universal hash functions** in a way that is independent of the keys being stored (Cormen et al., 2009). Although universal hashing distributes the keys satisfactorily on average, they are also expensive to compute.

Universal hash functions
These are a collection H of hash functions such that for any pair of keys a and b, the number of hash functions for which they map to the same location is at most |H|/m, where m is the number of memory locations.

Collision Resolution Schemes

When keys are mapped to the same location, the collision needs to be resolved. A variety of collision resolution schemes have been proposed to address this (Cormen et al., 2009).

Chaining

In chaining, each location in the hash table is a linked list of keys that has been mapped to that address by the hash function. We create m lists $L(i), 0 \leq i < m$. List $L(i)$ stores all the keys that get mapped to location i. Under the "simple unform hashing assumption", any element is equally likely to be mapped by the hash function onto any of the table locations (Cormen et al., 2009). Under this assumption, the search takes $\Theta(1 + \alpha)$ where $\alpha = n/m$ is the load factor, m the number of locations, and n is the number of elements. If we maintain $\alpha < 2\mathrm{m}$, the search time is constant. A way to maintain this is to increase the table size and rehash once $n = 2\mathrm{m}$.

Open addressing

Under "open addressing," all items are stored in the hash table directly. For collisions, we search for alternative positions within the table itself. To generate this probe sequence, we must find any empty slots and design a sequence that is a permutation of $\{0,1,2,...,m-1\}$.

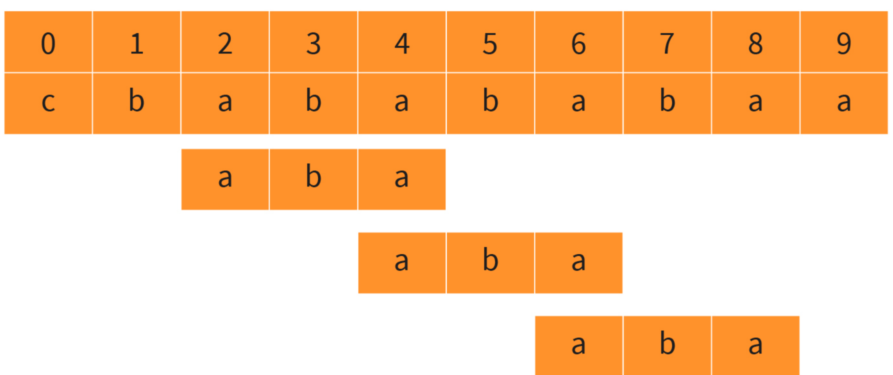
To search, we follow the same probe sequence as that used for insertion. If we encounter an empty slot during the search, we immediately conclude that the element being searched for is not present in the table because the insert operation followed the same probe sequence and would not have missed the empty slot. If $h(k)$ is the original hash function, let $g(k, i)$ denote the i-th location probed. The common algorithms for generating the probe sequence are as follows (Cormen et al., 2009):

- linear probing. Here, $g(k, i) = (h(k) + i) mod m$ for $0 \leq i < m$. This leads to "primary clustering" where several adjacent locations can be filled up.
- quadratic probing. Here, $g(k, i) = \left(h(k) + c_1 \cdot i + c_2 \cdot i^2\right) mod m$ for $0 \leq i < m$. If $h(a) = h(b)$, then $g(a, i) = g(b, i)$ for all i. This leads to "secondary clustering," which can be seen as less severe than primary clustering as more spots are used.
- double hashing. Let $g(k, i) = (h_1(k) + i \cdot h_2(k)) \mathrm{mod} m$ where h_1 and h_2 are auxiliary hash functions and $h_2(k)$ must be relatively prime to m for the probe sequence to explore all locations. The performance of double hashing is more efficient and therefore faster than that of linear or quadratic probing when m is prime or a power of two.

4.6 Pattern Recognition

In the classical pattern matching problem, we have an alphabet Σ , and we are given a pattern $P[0\,..\,m-1]$ and a text $T[0\,..\,n-1]$ where both P and T are strings over Σ. We wish to find all occurrences of P in T (Cormen et al., 2009). Other variants of the problem include finding either the first or any occurrence (Goodrich et al., 2013). We denote s to be a "valid shift" if P occurs in T with a shift s, starting at position s. Otherwise, the shift is deemed to be invalid (Cormen et al., 2009). One occurrence begins within another one. In the example shown below, $P = \mathrm{aba}, T = \mathrm{cbababababaa}, P$ occurs at $s = 2$, $s = 4$, and $s = 6$.

Figure 26: Pattern Matching



Source: Created on behalf of IU (2022).

Naïve Pattern Matching

Naïve pattern matching is a simple brute-force algorithm that tries every possible value of the shift s and checks whether it is a valid shift. There are $n - m + 1$ possible choices for s. Below is a Python implementation.

Code

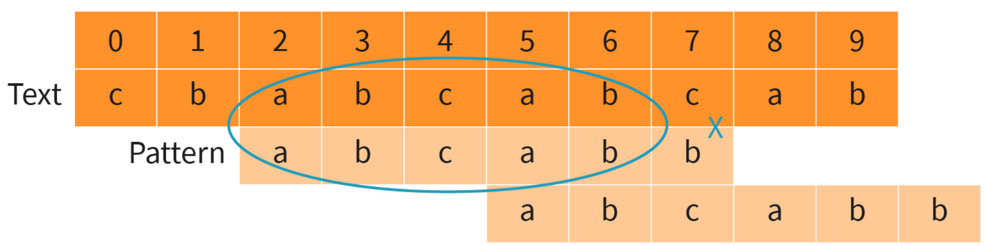
```
def naiveMatch(p,t):
   if not p or not t:
       return 0
   m = len(p)
   n = len(t)
   found = False
   for i in range(n-m+1):
       j=0
       k=i
       while j < m and i < n and p[j]==t[k]:
           j+=1
           k+=1
       if j== m:
           print("Found valid shift", i, "for", p)
           found = True
   if not found:
       print("No match for",p)

naiveMatch('aba','cbababaa')
naiveMatch('abc','cbababaa')
```

With the two nested loops, the running time is $O((n - m + 1)m)$. The algorithm's inefficiency has a reason; in the event of a mismatch, partial matches between the pattern and the text are not taken advantage of later.

The Knuth-Morris-Pratt Algorithm

The Knuth-Morris-Pratt algorithm (KMP algorithm; Cormen et al., 2009) corrects the problem associated with the naïve algorithm. If a prefix of p of size r has matched with s, followed by a mismatch, we try to determine the longest suffix of the matched part that is also a prefix of p. The key observation here is that this portion is already a matched part of text and need not be matched again. Additionally, some preprocessing of the pattern can support this computation, as depicted in the figure below.

Figure 27: The Prefix Function



Source: Created on behalf of IU (2022).

In this example, the substring `abcab` of the text matches a prefix of the pattern `abcabb`. When the last character of the pattern fails to match, the brute-force algorithm would try to shift the pattern by one position and attempt a rematch. However, the suffix `ab` of the matching substring `abcab` is also a prefix of the pattern. This substring is already matched with the substring `ab` of text at positions five and six. The pattern is now realigned (as shown) for a new attempted match, without a required rematch of the prefix `ab`. This saves comparisons over the brute-force algorithm. We pre-compute a prefix function in a table based on the pattern without knowledge of the text. Essentially, `table[j]=k` tells us that if the pattern fails to match at position `j+1`, we can assume that the first k characters of the pattern are already matched and proceed. A Python implementation of the prefix table computation is depicted below.

Code

```
def prefix(p):
   m = len(p) #size of pattern
   table = [0]*m #Creates a list of m zeros.
   i = 0
   for j in range(1,m):
       while i > 0 and p[i] != p[j]:
           i=table[i-1]
       if p[i] == p[j]:
           i+=1
       table[j]=i
   return table
```

A call to `prefix('abcabb')` returns `[0, 0, 0, 1, 2, 0]`. Here, `Table[4] = 2` tells us that when a failure to match occurs at position 5 of the pattern, as in our example, a prefix of the pattern of size two is matched up when the pattern is realigned as shown in the figure.

Code

```
def kmp(p,t):
    m = len(p) #size of pattern
    n =len(t)
    table=prefix(p)
    j=0
    for i in range(n):
        while j > 0 and p[j] != t[i]:
             j=table[j-1]
        if(p[j] == p[i]):
            j+=1
        if j == m:
            print("Match found at", i)
            return i-m+1

kmp('aba','cbababab aa')
```

The running time for the KMP algorithm is $\Theta(m+n)$ (Cormen et al., 2009).

SUMMARY

Tree traversal algorithms involve visiting all the nodes of a tree in a systematic order. There are four fundamental tree traversal algorithms: inorder, preorder, postorder, and level-order.

Searching and sorting are fundamental algorithmic problems with broad applications. A basic linear search comes in two variants: those for unordered and those for ordered sequences. For ordered sequences, a more efficient algorithm is the binary search algorithm. Some fundamental sorting algorithms include insertion sort, bubble sort, selection sort, quicksort, and mergesort. Preprocessing a set of strings to facilitate efficient search with strings is a common problem in text processing applications. The trie is an example of a data structure that stores such preprocessed strings.

Hash tables support content-based search. The basic challenges in designing a good hashing scheme include designing a good hash function and building a good collision resolution scheme. Examples of hashing schemes include the multiplication method, division method, and universal hashing.

Common collision resolution schemes are chaining and open addressing. Under open addressing, the different algorithms for generating the probe sequence are linear probing, quadratic probing, and double hashing.

There are different approaches to the problem of searching for a fixed preprocessed pattern string in a block of text. The naïve algorithm runs a sliding window of the pattern across the text trying to discover matches. The Knuth-Morris-Pratt algorithm, which constructs a prefix table to record information about prefixes of the pattern that occurs within it, enables us to get a faster solution than the naïve one.

UNIT 5

REPRESENTING STRUCTURED DATA

STUDY GOALS

On completion of this unit, you will be able to …

- understand the background and purpose of extensible markup language (XML).
- understand how to store and share data across platforms.
- use XML to automatically present data in the form of a web page.
- use JSON (an XML alternative) to process data.

5. REPRESENTING STRUCTURED DATA

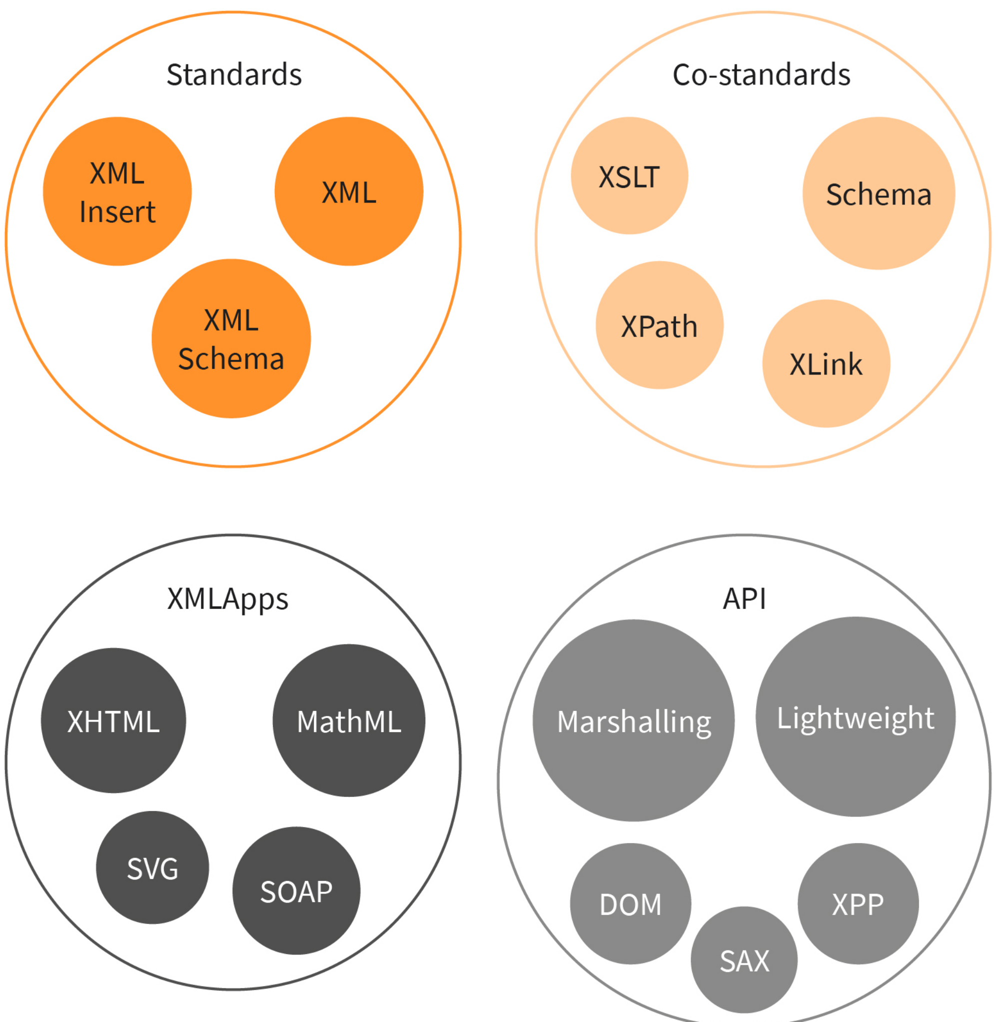
Introduction

In this unit, we will look at a common approach to storing data in a way that is machine-readable and platform-independent. XML, which was introduced in 1998, is a markup language that enables the automated storage and retrieval of data. Several other markup languages are also commonly used, including JSON and YAML, both of which we will explore later.

A markup language describes the properties, membership, and hierarchy of a set of data. The most well-known markup language is hypertext markup language (HTML), which is the basis for the entire presentation of all web pages. HTML primarily aims to represent the content of a web page and therefore describes, for example, the placing of an image or text. XML describes how the data to be stored is to be structured in terms of its properties and hierarchy alone. XML therefore, is not the most precise description of a web page, nor is it a universal representation of data. Moreover, XML is not to be understood as a programming language, as it is not made up of commands or of programing flow control.

XML can be seen as an extension of the comma separated values (CSV) format for organizing data. However, it was intended to offer far more than this format in terms of structuring possibilities. Later, XML evolved into an entire family of languages. An overview of this, including its core building blocks, is shown below.

Figure 28: Overview of the Core Family of Languages around XML



Source: Created on behalf of the IU (2023) based on Vonhoegen (2018, p.34).

At the center is the core specification, which is set by the XML 1.0 standard. This was extended by XML namespaces and finally, in 2001, by XML Schema, which together allow for the description of different content models. Later XML 1.1 was defined, with additional clarifications and restrictions. The supplementary specifications currently include:

- extensible stylesheet language (XSLT), which is used to transform XML documents
- XML Schema, which is used to formally specify the structure a document is allowed to have
- XPath, which is used to express ways to navigate between nodes of the XML-tree
- XLink, which defines how relative and absolute links are expressed and resolved.

Furthermore, the XML language family contains XML applications, which can be understood as an XML vocabulary for predefined application areas. These include, for example:

- the XML serialization of HTML: an XML-compliant reformulation of HTML
- scalable vector graphics (SVG): a language for describing two-dimensional graphics applications)
- mathematical markup language (MathML): a language for representing mathematical formulas
- simple object access protocol (SOAP): a language used for communication between web-services

The XML language family also offers a number of programming interfaces to enable other programs or websites to store data in XML, or to access data stored in XML. These programming interfaces are similar in many different programming languages. They include, for example (Vonhoegen, 2018):

- the document object model (DOM), which has been standardized by the W3C
- SAX , which is a simple application programming interface (API) for XML that offers a streaming approach to parsing
- lightweight in-memory representations of the XML tree as objects (in Python: minidom, in Java: JDOM)
- pull-based parsing approaches (in XPP, XML pull-parsing; pulldom in Python; and StAX in Java)
- “marshalling” approaches where XML elements are mapped to and from instances of classes (objects) in the programming languages (such as JAXB for Java)

5.1 Structure of XML documents

Any document that is to hold data in XML can only function as such if it is "well-formed" according to the XML standard. An XML document consists of elements of the form `<tag>content</tag>`, where `<tag>` denotes the start of an element and `</tag>` denotes the end of the element. The document is framed by what is called a root element, and all other elements are part of a parent element. In other words, XML describes a tree structure. For example, a person can be described in XML as follows:

```
<person>
  <name>Last</name>
  <first name>First</first name>
  <residence>Exampletown</residence>.
  <email>first@example.com</email>
</person>
```

A start tag can contain additional attributes, as in `<email type="private">`.

An XML document is well-formed if it satisfies the following conditions:

- For each start tag there must be a corresponding end tag.
- Different elements must be switched correctly (i.e. before an element can be closed, all sub-elements must be closed).

- Tag names are case-sensitive (i.e. `<tag>` is different from `<Tag>`).
- There must not be two attributes with the same name within one element.
- The values of attributes of an element must be enclosed in quotes.
- Within an XML document, there must be exactly one root element that is not contained in the content of another element.

The XML standard takes into account both the physical and the logical form of the XML document at hand (Sperberg-McQueen, 2008). Let's look at an example of the structure of an XML document created to manage the members of an interdisciplinary team:

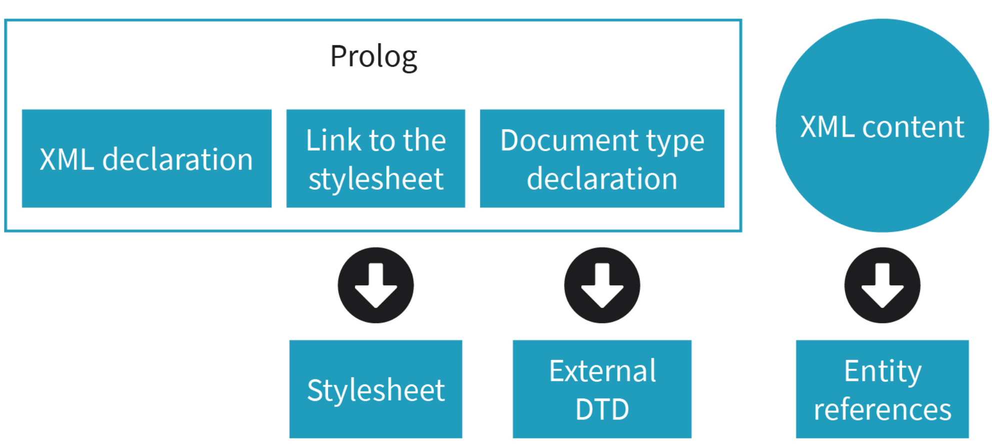
```
<? xml version="1.0" encoding="UTF-8"?>
<team>
  <person>Hanna Meier</person>
       <branch>Software Development</branch>
       <phone>1234</phone>
  <person>Max Mustermann</person>
       <branch>Accounting</branch>
       <phone>7890</phone>
</team>
```

An XML document starts with the formal definition of the XML standard used: in this case, version 1.0 with encoding in 8-bit UCS transformation format (UTF-8). We then introduce that we want to declare a team by opening the `<team>` tag. On the next lower outline level, we then devote ourselves to the individual people, in this case, Hanna Meier and Max Mustermann. The next outline level describes the attributes of the two team members (in our example, their field of activity and the corresponding telephone number). We can already see that XML uses a tree structure for data management. The logical view of the data is called the information set.

Physically, an XML document is nothing more than a document made up of characters, with an XML processor reading in these individual characters and interpreting them as structured data. The latter are referred to as "entities", while an entire document is referred to as a "document entity". Each entity has content and a specific name. The document entity, on the other hand, contains all individual entities and is regarded by an XML parser as a container for the entities (Vonhoegen, 2018).

The most important units into which an XML document can be divided are called "elements". An XML document can thus be understood as a document in a tree structure that is made up of elements. The function that these elements ultimately fulfill depends on the tag introduced in each case. The structure of an XML document is shown in the following figure.

Figure 29: Representation of the Structure of an XML Document



Source: Created on behalf of the IU (2023) based on Vonhoegen (2018, p.50).

The XML version used is specified with the help of the prolog: this is recommended, but not mandatory. This can be extended with information about the encoding used, which indicates how characters are represented in bytes and bits (i.e. how they are are to be encoded), and whether the XML document at hand is to be extended by an external document definition. Here is a common prolog:

```
<? xml version="1.0" encoding=utf-8"?>
```

The information represented in the XML document contains the content of a root element which, in turn, can contain other elements. Each element can contain text, attributes, and further elements. To illustrate this, let's look at the example of a team containing different members. The corresponding XML section would look as follows:

```
<team>
  <person>Hanna Meier</person>
  <person>Max Mustermann</person>
</team>
```

In this example, `<team>` would be the element and the two `<person>` tags would be the associated child elements. It is also clear that an end tag always has the outer form `</tag>`, while the start tag is introduced by `<tag>`. Some tags can be written in the self-closing form if they have no content such as `<special-needs/>` to indicate that the content of this child is empty. Furthermore, elements can have any number of attributes. Let's assume that it is essential for us to also store the number of members in our team. We can realize this as shown below:

```
<team>
  <person size = "170">Hanna Meier</person>
  <person size = "185">Max Mustermann</person>
</team>
```

If we would like to additionally provide our XML document with a comment, which is only captured as data for those looking at the source, we achieve this as follows:

```
<!--
This is a comment field
-->
```

In the previous section, we looked at building a small team using XML tags. Let's now turn to a more complicated example, where we consider the contents of a warehouse containing various products. Again, these can be described using different information such as color, material, material group, and so on. Moreover, a document type declaration (DTD), for example, can be used to define a formal language definition for the correct description of the inventory of a warehouse. DTDs define what the XML document in question should look like in concrete terms. This defines a specialization of XML but is written in a syntax that is different from XML syntax. We will now look at an example of a DTD to understand its structure in more detail (Vonhoegen 2018, p. 36).

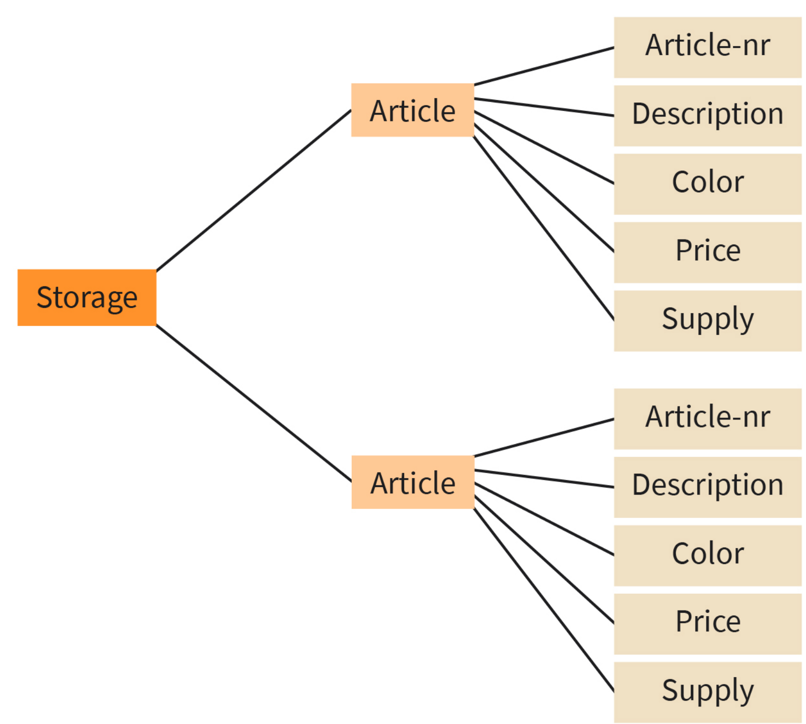
```
<?xml version="1.0" encoding="UTF-8"?>
    <!DOCTYPE bearing[
        <!ELEMENT order_number (#PCDATA)>
        <!ELEMENT name (#PCDATA)>
        <!ELEMENT color (#PCDATA)>
        <!ELEMENT price (#PCDATA)>
        <!ELEMENT supply (#PCDATA)>
        <!ELEMENT item (order_number,color,price,supply)>
        <!ELEMENT storage (item+)>
        ]>
```

This minimal example of a DTD defines the dialect to be used for a concrete implementation of an XML document, in this case one designed to manage stock. Let's look at this with an example:

```
<storage>
  <article>
    <order_number>1234</order_number>
    <supply>4</supply>
    <price>14,99</price>
  </article>
  <article>
    <order_number>4567</order_number>
    <supply>3</supply>
    <price>19,99</price>
  </article>
</storage>
```

Here we have introduced a minimal example of a warehouse, consisting of two products (1 and 2), which can be uniquely described by the attributes: item no., description, color, price and stock. We can also represent this structure of an XML document graphically in the form of a tree structure, as can be seen in the figure below.

Figure 30: Tree Diagram of the Document in the Example



Source: Created on behalf of the IU (2023).

The root element is the warehouse itself, containing two child elements "item", which in turn are represented by their attributes (also represented as child elements in XML).

In addition to the original DTDs, there are now various other mechanisms for defining the structure of XML documents for a specific application purpose. The most widespread are XML Schema and Relax NG. A well-formed document that also complies with such a structure definition is called valid.

XML has been adopted broadly and can be used to serialize data of many different types. For example, if you want to transfer a data type of object over a network connection, this object must first be converted into a transferable form and XML can be used. In this way, it is also possible to store an object in XML format in a file and, if necessary, to read it out again to restore the original object.

5.2 Accessing XML Documents with the DOM and SAX Approaches

For a program to be able to process an XML document, a parser that will convert the stream of bytes to the structured information that the document represents is needed. There are multiple parsing approaches that are different in their APIs and have varying degrees of effectiveness in terms of ease-of-programming and performance.

The DOM approach enables the creation of a uniform programming interface to manipulate the data of an XML document, including reading and writing. An XML parser loads an arbitrary XML document and makes it available in the form of structured elements. The typical structure of a DOM document is shown in the following figure and is available for most programming languages.

Figure 31: XML Document Accessed Through a DOM Parser

Source: Created on behalf of the IU (2023) based on Vonhoegen (2018, p.416).

The basic idea is that the **class** of the elements of an XML document provides functions of the type `setAttribute()` or `getAttribute()`, with which the values of an element of an XML document can be manipulated or read.

Class
This is a summary of various functions that can be used to manipulate an XML document.

The DOM API is a standardized API with the same method and **class** names across all languages. Contrary to this approach, lightweight approaches such as ElementTree in Python or JDOM in Java leverage the special approaches of the language to offer an easier programmatic representation. However they are not translatable to other languages.

Let's now try to read the XML document at hand using Python's minidom, a reduced implementation of the DOM interface in Python.

The basis for this is to first parse a document.

```
from xml.dom.minidom import parse, Node
xmldoc = parse("data/sample.xml")
```

Then, we obtain the root node of the DOM tree. This is done using the following command:

```
stockNode = xmldoc.documentElement
```

In the next step we want to get a list of all child elements of this root node (i.e. all articles in the warehouse). We can achieve this as follows:

```
nodeList = stockNode.childNodes
```

This gives us a list whose elements we can access by means of an index `[i]` as follows:

```
articleNode = nodeList[i]
```

or through a loop like the one below:

```
for itemNode in nodeList:
    print(itemNode.nodeType)
```

Note that a node is a generic object that can be of any nature: element, text (including whitespace text), processing-instructions, and comments. The type can be checked with `nodeType`. In order to access the elements, one needs to filter through the list. Let's define `articleNode` to be `nodeList[1]`, the first article.

Debugging can be done with the string representation of the object as follows:

```
articleNode = nodeList[1]
print(str(articleNode))
## outputs: <DOM Element: article at 0x1079c3490>
```

We will now face the task of searching through the tree for the price of an article with the given `order_number`. We can loop through the children of the root node, then loop through each of their children and collect the price and order-number, returning the value if the order-number matches:

```
def searchPrice(searchedOrderNum):
  for itemNode in nodeList:
    if (itemNode.nodeType == Node.ELEMENT_NODE):
      foundNum = 0
      foundPrice = 0
      for attNode in itemNode.childNodes:
        if attNode.nodeType == Node.ELEMENT_NODE \
            and attNode.nodeName == "order_number":
          foundNum = attNode.childNodes[0].data.strip()
        if attNode.nodeType == Node.ELEMENT_NODE \
            and attNode.nodeName == "price":
          foundPrice = attNode.childNodes[0].data.strip()
      if foundNum == searchedOrderNum:
        return foundPrice
```

This function uses a simple traversal of the tree, avoids using non-element nodes, then collects the price and the order number. At the end of this collection it matches the price to the expected number and, if found, returns it. This technique is as follows: one builds a **reading head** that walks through the in-memory representation of the document.

Reading head
This is a data-structure that imitates the reading head of vinyl-discs' or magnetic tapes' readers: it collects information at the position it is pointed to and transmits it.

DOM allows not only the parsing and reading of XML documents but also their creation and modification. After modifications, one can write the document back using `node.writexm`. However, for these operations to happen, the object graph of the complete document needs to go into memory, making it inappropriate for large documents.

SAX as an Alternative to DOM

The storage of DOM elements can pose a problem in terms of memory, however there are alternative strategies available. Whereas in DOM, data is typically stored in a tree structure, other approaches follow the paradigm of XML parsing as a stream. In stream-based approaches, it is not necessary for a large collection of nodes to be loaded in memory. The most established example of this is called "simple API for XML" (SAX). SAX delivers the parsing result as events to a registered handler. The handler then collects the necessary information according to the application requirements.

SAX is available in most programming languages in which the definition of objects that are of a subclass of a SAX handler are allowed. In Python, this is done by creating a subclass of `ContentHandler`of the module `xml.sax.handler`. The subclass overrides methods to collect the information, or simply ignores the events: `startDocument`, `startElement`, `endElement`, or `characters` are among the widespread methods.

The task of searching the price of an article for a given order-number can be done using SAX: Here, the reading-head approach explained above is also used however, the reading head needs to be activated only at the right place, within the appropriate elements.

```
from xml.sax import parse
from xml.sax.handler import ContentHandler

class StockSearchHandler(ContentHandler):

  _searchedNum = ""
  _readPrice = ""; _readNum = ""
  _inOrderNum = False; _inArticle = False
  _inPrice = False
  _foundPrice = ""

  def __init__(self, searchedOrderNum):
    super().__init__()
    self._searchedNum = searchedOrderNum

  def startElement(self, name, attrs):
    if name == "order_number":
      self._inOrderNum = True
    if name == "article":
      self._inArticle = True
      self._readPrice = ""; self._readNum = ""
    if name == "price": self._inPrice = True

    print(f"BEGIN: <{name}>, {attrs.keys()}")

  def endElement(self, name):
    if name == "order_number":
      self._inOrderNum = False
    if name == "article":
      self._inArticle = False
      ## now check if this was the right item
      if self._searchedNum == self._readNum:
        self._foundPrice = self._readPrice
    if name == "price": self._inPrice = False

    print(f"END: </{name}>")

  def characters(self, content):
    if content.strip() != "":
      if self._inOrderNum: self._readNum += content
      if self._inPrice: self._readPrice += content
      print("CONTENT:", repr(content))
```

```
  def getFoundPrice(self):
    return self._foundPrice
```

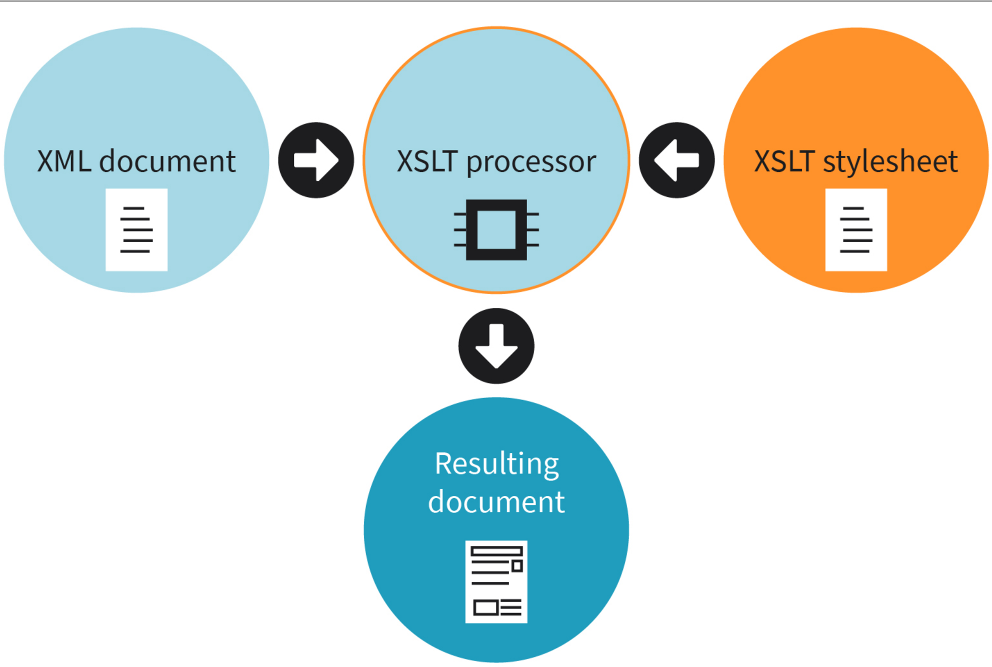
The resulting code is a single pass that includes parsing and collecting information. It would be inappropriate if it had to be carried out multiple times per second: In this case in-memory representation would be better. However, this code is able to tolerate collections of billions of articles and still remain effective in terms of memory.

Alternatives to DOM and SAX parsing have continue to emerge. Language-specific versions such as JDOM or ElementTree have appeared, claiming to offer ease of use for development, especially for inexperienced developers. Latecomers have contributed pull-parsing to the landscape: Instead of streaming using a handler that is called, the streaming is directed by a reading-head that the application pilots itself. It can then give such instructions such as “read till the end of this element”, an elementary action which can only be done by manipulating flags or stacks using the SAX approach.

5.3 Transformation of XML documents using XSL

Although an XML document is generally readable by humans, it is ultimately somewhat unwieldy if larger amounts of data stored in XML documents are to be processed manually. An automated way to render an XML document more user-friendly is to use the XSLT language to transform the data into a document that can be understood or processed more easily, for example, in an HTML document to be shown to the browser. XSLT applies “stylesheets”, which are a pack of templates that each process a small part of the source XML, with each template being selected by a given condition.

Figure 32: Process of Creating a New Document Using XSLT



Source: Created on behalf of the IU (2023) based on Vonhoegen (2018).

The conversion is not limited to a conversion to HTML. It can also be to another XML document or to pure text.

```
01 <?xml version="1.0"?>
02 <xsl:stylesheet version=”1.0”
03     xmlns:xsl ="http://www.w3.org/1999/XSL/Transform">
04  <xsl:output method = "html"/>
05  <xsl:template match="/">
06    <html>
07      <head>
08        <title>Stock</title>
09      </head>
10      <body>
11        <h1>The warehouse currently contains:</h1>
12        <xsl:for-each select="storage/article">
13          <p><xsl:value-of select="order_number"/></p>
14          <p><xsl:value-of select="label"/></p>
15          <p><xsl:value-of select="color"/></p>
16          <p><xsl:value-of select="price"/></p>
17          <p><xsl:value-of select="stock"/></p>
18        </xsl:for-each>
19      </body>
20    </html>
21  </xsl:template>
22 </xsl:stylesheet>
```

Line 2 defines the document as an XSL template, while line 3 specifies that the output document should be an HTML document. We tell the XSL processor by means of line 4 that it should first locate itself in the root of the XML document. This is followed by outputting the start of the HTML document with the title "Stock". In line 12, we systematically step through the article elements that are below the storage elements in our XML document and then output the associated stored data. An example of a possible result of this output (applied to the previous example) is shown below:

```
The warehouse currently contains:
<h1>12345</h1>
<p>Product 1</p>
<p>Blue</p>
<p>25.90</p>
<p>22</p>
<p>65432</p>
<p>Product 2</p>
<p>Black</p>
<p>9.99</p>
<p>34</p>
```

XSLT can be applied by calling XSLT-processors. Currently Xalan and xsltproc are two classic XSLT-processors. However, web-browsers can also process XSLT. In order for a web browser to know how to convert the XML document into an HTML document at the end using the XSL template when it traverses an XML document, we open the XML document with the following lines:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl"
href="StorageStylesheet.xsl"?>
```

The actual reference to the XSL template is made by means of the `href` attribute. XSLT can operate on a pull basis, where, for example, children are extracted and transform (for example, by using `xsl:value-of`). It is also in a position to push the processing by letting any matching template be called, for example, using the `xsl:apply-templates` element which will search for any template matching the currently selected element. The flexibility of this push and pull mechanism has made XSLT into a universal tool for manipulating XML.

5.4 Alternative Document Representations

The popularity of XML has established the feasibility of representing structured information using a text based language. However, XML has also been criticized for its rich structure. Among the greatest concerns are the fact that namespaces have introduced a complexity that prevents simple parsers from operating.

Some alternative languages aim for more simplicity while still retaining the flexibility of XML structures. We will present two notable examples: JSON and YAML. Both of these languages have supporting parsers, producers and validators for most programming languages, some using streaming approaches.

JavaScript object notation emerged from the web-browser community where a simple hypertext transfer protocol (HTTP) request could obtain "object data" simply by loading an extra JavaScript file. The JavaScript syntax used to write the data of objects shown here

```
[
  {"order_num": 1234, "supply":  4, "price":  14.99},
  {"order_num": 4567, "supply":  3, "price":  19.99}
]
```

has been the basis for the JSON specification which allows array encoding (between `[ and ]`), objects (between `{ and }`), object properties in the form `"name": value` as well as atomic constants as strings (between double quotes), Booleans, null or numbers. Contrary to XML, JSON is always encoded in UTF-8. Note that JSON is more strict than JavaScript in its syntax: property names need to be quoted and comments are not supported. The minimalism of the JSON model and its compact appearance, however, has convinced many programmers and the syntax is used widely in software projects (JSON, n.d.).

Both XML and JSON allow whitespace to be used freely. This provides the flexibility to decide when to put two attributes beside each other or one on each line: Other characters can be used to mark the separation of data. YAML follows the Python approach, employing newlines and indentation in order to structure data (Ben-Kiki et al., 2021). Compared to JSON, YAML is richer in its notations, allowing almost all of JSON syntax aside from more normal indent-based structures. YAML also includes producers, parsers and validators in multiple programming languages.

The code below shows our storage example in YAML. Note that more compact notations would be possible too.

```
- order_num: 1234
  supply: 4
  price: 14.99
```

```
- order_num: 4567 # this is the second item
  supply: 3
  price: 19.99
```

SUMMARY

XML makes it possible to store data in a structured manner and to load and process it as required. Processing such documents involves parsing and producing them, and may also involve validating them against a set of rules. There are several APIs for XML processing, each with their own advantages: Some are standardized, others are easy to manipulate, or they may be well-suited to performing at high volumes. Classical approaches include DOM (in memory) and SAX (event-based) parsing.

XML is one of the most advanced structured data languages (often called markup languages). However, alternatives have been popularized that offer other possibilities such as light weight or the use of indents: YAML and JSON are notable examples.

UNIT 6

MEASURING PROGRAMS

STUDY GOALS

On completion of this unit, you will be able to …

- apply type inference mechanisms.
- understand tools used to generate documentation
- apply knowledge of best practices in documentation sharing.
- develop an awareness of compiler optimization techniques and difficulties.
- compare several tools for code coverage analysis.
- understand and apply a range of principles of unit and integration testing.
- discover bugs using the apply heap analysis tools.

6. MEASURING PROGRAMS

Introduction

Measurements are of paramount importance in any scientific process. Observations based on measurements lead to generalizations and the development of theories that facilitate the implementation of the process. The software development process, including design, coding, debugging, testing, verification, and integration, has also benefited from the development and application of measurement methodologies.

Different types of metrics have evolved to capture and measure various aspects of programs. Some metrics attempt to capture features of the product such as complexity, size, or performance. One such measure is "cyclomatic complexity." Based on the cyclomatic number in graph theory, this metric tries to measure the difficulties involved in testing and understanding a program.

Quality improvements can be made by focusing on the reduction of complexity. Product quality metrics include "defect density," which tries to measure defects relative to the size of the software, for example, lines of code. The "mean time to failure" tries to measure the average time between encountering two defects in the program.

"Process metrics" are those that target improvements of the software development process and maintenance. A primary goal in the software development process is to ensure that the implementation meets the requirement specifications. To achieve this goal, "code coverage" was one of the first metrics developed for software testing. It tries to measure to what extent the program is covered by the test cases. This is defined in terms of various criteria, such as lines of code, instructions, functions, function calls, or branches, which are expected to resemble a representative usage. A combination of instruction coverage and branch coverage is commonly used today, and test coverage is an important consideration in equipment certification in the avionics and automotive industries.

6.1 Type Inference and IDE Interactive Support

Type system
A type system is a logical system defined with a set of constructs to assign types to entities, such as variables, expressions, or return values of functions.

The types and **type system** are important characteristics of a programming language and vary between languages. A type is defined by a set of values and a set of operations that operate on those values. There are language-specific constraints on the usage of types in a program. The type system defines the set of built-in types for the language, provides the constructs for defining new types, and defines rules for control of types. In some languages, the types of variables may need to be specified by the programmer completely. "Type inference" involves the derivation of the types of expressions in a programming language, usually performed at the time of compilation. Logical inference algorithms then derive these unspecified types (Sebesta, 2016).

Difficulties in Python

Python is dynamically typed. To illustrate the problem of type inferencing in Python, consider the following code fragment:

Code

```
from  random import randint
def typeCheck(num):
   if(num%2):
       a = 123
   else:
       a = "123"
   print(type(a))
typeCheck(randint(1,1000))
```

What is the type of a in line number six? Here, if num is odd, the type of a is "integer". If num is even, the type of a is "string". Since the argument to the function typeCheck is randomly generated, its parity becomes known only at runtime. Another difficulty is that Python allows new code generation at runtime: any code optimization done with knowledge of the full code is at risk of becoming invalid.

Type Inferencing in ML

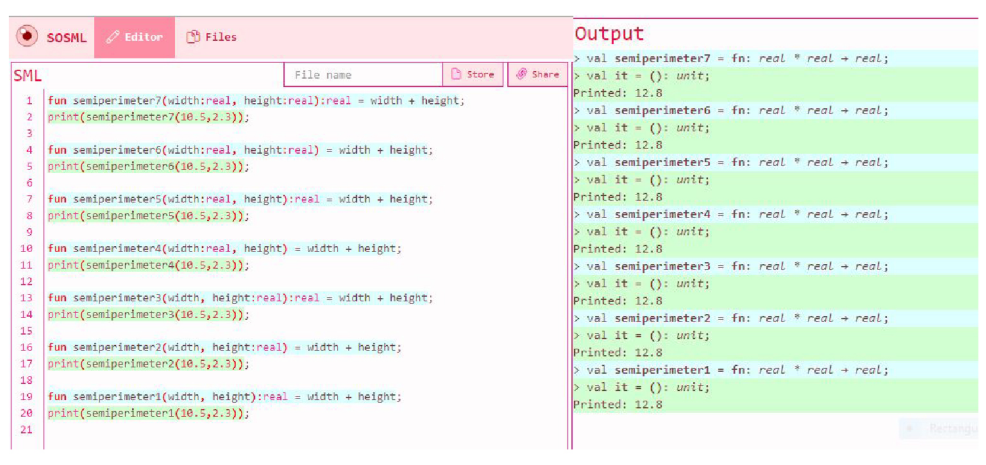
Type inference has a long history in the context of functional programming languages. Practical type inferencing was applied to the programming language, meta language (ML) by Robin Milner (Sebesta, 2016). ML is primarily a functional programming language with support for the imperative style of programming. It has a syntax similar to many imperative languages and is strongly typed, with all types being statically inferred. In ML, type declarations are not required if the types can be derived unambiguously. Standard ML (S ML) is a modern dialect of ML. Saarland Online S ML (SOSML) is the online integrated development environment (IDE) for S ML developed at Saarland University. Consider the computation of the semi-perimeter of a rectangle as the sum of its width and height in SOSML. The following definitions are all equivalent, and all produce the correct result whenever the type (real) of at least one of either width, height, or the function, is specified. The type inference mechanism infers the missing types as real.

Code

```
fun semiperimeter7(width:real, height:real):real = width + height;
print(semiperimeter7(10.5,2.3));

fun semiperimeter6(width:real, height:real) = width + height;
print(semiperimeter6(10.5,2.3));

fun semiperimeter5(width:real, height):real = width + height;
print(semiperimeter5(10.5,2.3));

fun semiperimeter4(width:real, height) = width + height;
```

```
print(semiperimeter4(10.5,2.3));

fun semiperimeter3(width, height:real):real = width + height;
print(semiperimeter3(10.5,2.3));

fun semiperimeter2(width, height:real) = width + height;
print(semiperimeter2(10.5,2.3));

fun semiperimeter1(width, height):real = width + height;
print(semiperimeter1(10.5,2.3));
```

Figure 33: SOSML IDE



Source: Prosenjit Gupta (2022), based on Jujuedv & PHP Wellnitz (n.d.).

If none of the types are specified, all the types default to integer, and the program reports an error when invoked with real parameters. An example is the code below which computes the half of the perimeter of the rectangle given the lengths of the sides:

Code

```
fun semiPerimeter0(width, height) = width + height;
print(semiPerimeter0(10.5,2.3));
```

Statically Typed Languages

Statically typed languages obey a static type system, and the type system rules can be checked at compile time. Declaring all variables with designated types and requiring that expressions have well-defined types are ways to ensure that type system rules can be verified at compile time. However, this can be interpreted as too conservative and comes at a price. Consider the following Python code fragment:

Code

```
x=1
if(0==1):
   x="2+3"
else:
   x=x+2
print(x)
```

This executes without error in Python and the value of x is correctly printed as 3. The if branch is not executed and so does not interfere with the rest of the computation. However, the types of x in the two branches of the conditional statement being different, a static type checker would have flagged an error.

6.2 Cyclomatic and Referential Complexity

Cyclomatic Complexity

"Cyclomatic complexity" is an example of a predictor, or **product metric**. It is the measure of complexity in a program. While there are many complexity measures, it is important to choose one that is largely independent of implementation characteristics such as source formatting and programming language. This measure was originally proposed by Thomas McCabe (Kan, 2016), and tries to quantify the testability and maintainability of software. For example, to measure the complexity of the control structure of a program, we consider its control flow graph. Cyclomatic complexity is the number of linearly independent paths through this graph. Mathematically, the cyclomatic complexity $CC(G)$ of a control flow graph G with e edges, v vertices, and k components is defined as follows (Kan, 2016):

Product metric
A product metric is a software metric associated with the software itself as opposed to control metrics, which are associated with software processes.

$$CC(G) = e - v + 2\mathrm{k}$$

If the number of components $k = 1, \mathrm{CC(G) = e - v + 2}$.

This also represents the minimum number of paths whose linear combination can generate all possible paths in the graph. High complexity, in general, is a major cause of software errors as it makes it difficult to fully understand the code. Studies have shown that cyclomatic complexity has a high correlation with errors in software (Watson & McCabe, 1996).

Consider the following Python function:

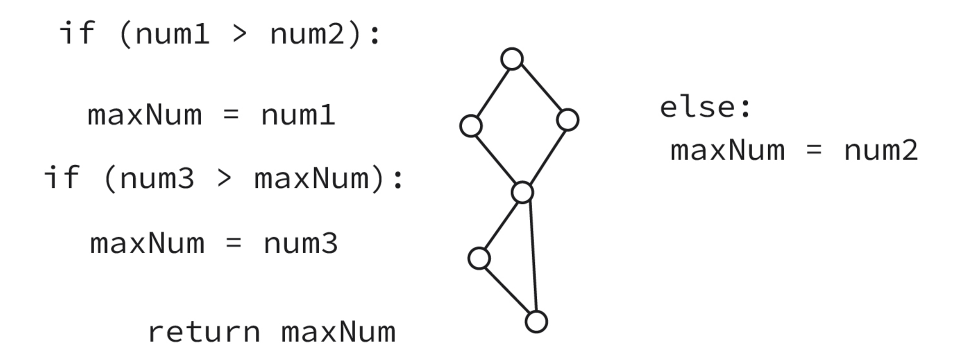
Code

```
def testMax(num1, num2, num3):
   if(num1 > num2):
       maxNum = num1
```

```
else:
    maxNum = num2
if(num3 > maxNum):
    maxNum = num3
    return maxNum
```

The control flow graph for the above is the graph G_1 below:

Figure 34: Control Flow Graph G_1



Source: Created on behalf of IU (2022).

If we add the following (redundant) code just before the return, the control flow graph changes to the graph G_2 below.

Code

```
else:
 maxNum = num2
```

Figure 35: Control Flow Graph G_2

if (num1 > num2):

maxNum = num1

if (num3 > maxNum):

maxNum = num3

return maxNum

else:
maxNum = num2

else:
maxNum = num2

Source: Created on behalf of IU (2022).

For G_1, the cyclomatic complexity $\mathrm{CC(G_1) = e - v + 2 = 7 - 6 + 2 = 3}$.

For G_2, $\mathrm{CC(G_2) = e - v + 2 = 8 - 7 + 2 = 3}$.

Simplified calculations

A straight-line control flow graph with one start and one exit node has a complexity of $e - v + 2 = 1$. If we add p binary decision tests, they add p to the cyclomatic complexity since each decision predicate adds two edges and one vertex, which adds one to the cyclomatic complexity. Thus, a control flow graph with all predicates being binary predicates has a cyclomatic complexity of $p + 1$ where p is the number of binary decision predicates.

For a **planar graph**, Euclid’s formula gives us the number of edges e, the number of vertices v, and the number of regions r in the planar embedding of the graph (Rosen, 2019): $r = e - v + 2$. Since $\mathrm{CC(G) = e - v + 2}$, $\mathrm{CC(G) = r}$. Thus, if the control flow graph is planar, the cyclomatic complexity is the number of regions in the planar drawing of the graph.

Planar graph
A planar graph is a graph that can be drawn on the plane without any edge crossings.

Cyclomatic complexity using Radon

Radon is a Python tool that computes various software metrics from source code. Metrics computed include cyclomatic complexity, raw metrics related to the number of lines, Halstead metrics, and maintainability index, among others.

In a file radonTest.py, we define the functions to be tested:

Code

```
aList = [23,34,2,13,11,-1,33,-44]
def linSearch(numList, keyValue):
        index = 0
        while(index < len(numList)):
                if(keyValue == numList[index]):
                        return index
                index += 1
        return -1
def typeCheck(num):
        if(num%2):
                x = 123
        else:
                x = "123"
print(type(x))
def testMax(num1, num2, num3):
        if(num1 > num2):
                maxNum = num1
        else:
                maxNum = num2
        if(num3 > maxNum):
                maxNum = num3
        return maxNum
```

The contents of the above file are fed to Radon's `ComplexityVisitor` **application programming interface** (API).

Application programming interface
An application programming interface, or API, serves as an intermediate layer that allows applications to communicate.

Code

```
from radon.visitors import ComplexityVisitor
f=open("radonTest.py","r")
v = ComplexityVisitor.from_code(f.read())
f.close()
print(v.functions)
```

This prints the output below. The complexity value refers to cyclomatic complexity (Watson & McCabe, 1996):

Code

```
[Function(name='linSearch', lineno=3, col_offset=0, endline=9,
is_method=False, classname=None, closures=[], complexity=3),
Function(name='typeCheck', lineno=10, col_offset=0, endline=14,
is_method=False, classname=None, closures=[], complexity=2),
Function(name='testMax', lineno=16, col_offset=0, endline=23,
is_method=False, classname=None, closures=[], complexity=3)]
```

Cyclomatic complexity helper functions are available in `radon.complexity` (Python Software Foundation, 2021a).

Code

```
from radon.complexity import cc_rank
  print(cc_rank(5), cc_rank(8), cc_rank(13), \
   cc_rank(26), cc_rank(36), cc_rank(45))
```

This prints (A, B, C, D, E, F).

In Radon, the cyclomatic complexity score is converted to a rank using the following equation:

$$\text{rank} = \lceil \text{score}/10 \rceil - H(5 - \text{score})$$

where H is the Heaviside step function (which gives $0$for negative numbers and 1 for positive numbers). The rank in turn is converted to a letter grade with A for a $\text{rank} = 0$ and for $\text{rank} \geq 5$, with A indicative of a simple block and indicating a very high-risk block. The higher the cyclomatic complexity is, the more complex the code will be. Such code is likely to be prone to coding errors and may be unstable, requiring frequent modifications and bug fixes.

Data Referencing Metrics

The data dependency complexity, within and across modules, is captured by the data flow metrics. These measures are useful in practice. Dunsmore's "data flow complexity" is defined as the average number of **live variables** per statement in a block of code (Chung, 1990).

Live variable
A variable is said to be live between its first and last references within a function.

Chung (1990) redefined data flow complexity based on live variable referencing (places where a variable is used later than its declaration).

A definition of a variable v occurs in a statement whenever there is an assignment of a value to v. A definition clear path to v is a path in which v is not reassigned. The definition of a variable v reaches the top of a block b of code if, and only if, there is a definition clear path from the definition of v to the top of block b. Analogously, we can describe the notion of the definition of v reaching the bottom of b. The definition of v is live at the top of b if the definition reaches the top of b and it is referenced later. Variable v is live at the top or bottom of a block b if there is a live definition of v. The total number of live variables in a block or the total of live definitions of all live variables in the block are suitable complexity measures.

6.3 Digesting Code Documentation

Documentation consists of explanatory remarks and comments that assist in better understanding the code. However, software documentation is often plagued by numerous issues, such as poor content or ambiguous information. This has led to research and development in automatic generation and recommendation of documentation.

Tools

Context-aware recommendation tools generate documentation that is both context-sensitive and of high quality (Aghajani et al., 2020). These include tools that use text summarization algorithms to create summaries from bug reports, code snippets and changes, classes and methods, and unit tests (Aghajani et al., 2020).

Doxygen

Originally developed for C++, Doxygen can be used to generate documentation for C, C#, Java, Python, and PHP. It can generate a HyperText Markup Language (HTML) file for online browsing or a LaTeX file for creating an offline manual. It can also be used to derive structure from code and to generate dependency graphs and collaboration diagrams.

Sphinx

Sphinx is a popular and comprehensive document generator for Python but is also used for other languages. It generates automatic cross-referencing links for functions and classes and creates indices. It also allows customization through user-defined indices. It uses the powerful reStructuredText markup language (Garcia-Tobar, 2017), which is the basis of the Read the Docs (n.d.) website.

Javadoc

Javadoc is used to generate HTML pages from Java source files and to parse declaration and documentation comments in Java source files. The HTML pages describe the public and protected classes, interfaces, nested classes, methods, and constructors. The `javadoc` command can be run on entire packages or individual source files.

Swagger

RESTful APIs
A RESTful API is an API conforming to the REST software architectural style, which defines a set of rules for creating web services.

Swagger is an "interface description language" for describing **RESTful APIs** used to communicate with web services. Swagger Core generates an OpenAPI interface from existing Java code. The documentation can be generated automatically from the API definition.

pdoc

pdoc is used for generating Python documentation; it is simpler than Sphinx and has minimal setup requirements. The documentation is simply entered as a markdown language. Moreover, pdoc automatically links identifiers in Python docstrings to corresponding documentation. Source code of functions and classes can be viewed in HTML.

Pydoc

Docstring
A docstring is any string appearing as the first statement of a class, a member function of a class, a function, or a module in Python.

Pydoc is an online help system and document generator in Python. The document may be created as text or HTML and is derived from **docstrings**. In Python, docstrings help to embed documentation into the source code (Goodrich et al., 2013), and are demarcated by triple quotes (“””) at the beginning and end. There are various ways to retrieve the documentation. For example, `help(obj)` for any object `obj` generates the corresponding documentation. Alternatively, the documentation could also be retrieved using `repr(linSearch.__doc__)`.

Below is an example using help(obj):

Code

```
aList = [23,34,2,13,11,-1,33,-44]
def linSearch(numList, keyValue):
   """Search for keyValue in numList.
   Args:
       numList: a list of values

       keyValue: a value being searched for in numList.
```

```
    Returns:
        index of keyValue in numList if found.
        -1 otherwise.
    """
    index = 0
    while(index < len(numList)):
        if(keyValue == numList[index]):
            return index
        index += 1
    return -1

print(linSearch(aList,11))
help(linSearch)
```

Best Practices

Various best practices have been established by the developers' community over time, facilitating code maintainability. Similar guidelines exist for any documentation generator based on code. Guidelines for using docstrings in Python, for example, were documented in PEP 257 (Goodger & van Rossum, 2010). Recommended best practices include

- documenting modules. Each module should start with a top-level docstring that outlines the purpose of the module. The subsequent paragraphs should describe the module operations.
- documenting classes. There should be a class-level docstring for every class, describing the purpose and operations. It should also describe the public attributes and methods, and provide guidance for deriving subclasses, including information on attributes and methods.
- documenting functions. This is similar to modules and classes. Additionally, there should be explanatory entries for function arguments, return values, and any special behaviors.

Issues in Sharing

Good documentation facilitates collaboration. However, various issues arise in the process that need to be tackled effectively (Aghajani et al., 2019), including

- what the documentation contains. These include the qualities of being correct, complete, and current.
- how the content is written and organized. These include ease of use, readability, and usefulness for the intended purpose. For instance, a relevant issue is whether the documentation can be understood by the intended audience.
- documentation generation tool and documentation processes.

6.4 Compiler Optimization

When we write programs in a high-level language, our efforts to make the program more efficient are focused on using algorithms that take less time or memory. However, the code generated by compilers can often be optimized. The optimized code may take less time, less memory, or both. Increasingly, energy efficiency is also an objective, and optimizing compilers generate transformations that make the code more efficient. Some optimizations are machine-independent while others are not.

Code Optimization Techniques

Local and global optimization

Peephole optimization
A peephole optimization is a code optimization technique applied on a small amount of code appearing in a sliding window.

Several optimizations are classified under **peephole optimizations**, which are local. In global code optimization, improvements are based on analysis, usually of data flow, across blocks (Aho et al., 2007). These are usually based on data flow analysis.

Making small functions inline

Often, we can replace function calls with the code for the function itself. This can improve performance in the case of small functions.

Taking repetitive computations outside loops

Consider the following Python example:

Code
```
pi=3.14
for index in range(0,20):
   alist.append(2*pi*index)

print(alist)
```

Here, the computation $2 \times \pi$ is being done once for each iteration of the loop. This can be made more efficient as follows:

Code
```
alist=[]
pi=3.14
twoPi=2*3.14
for index in range(0,20):
   alist.append(twoPi*index)

print(alist)
```

Elimination of common subexpressions

Consider the following Python code snippet:

Code
```
a=2
b=3
y=3**a + 3**a*b
print(y)
```

Computing `y` as `y=3**a*(1+b)` is more efficient, since it eliminates one exponentiation operation.

Eliminating redundant stores

If a variable appears on the left side of an assignment statement but is never used again, the assignment operation is redundant and may be removed.

Eliminating unreachable code

Some parts of the code may be unreachable and hence may be removed. This could happen as in the example below if `flag` is never false and the `else` part is never reached.

Code
```
flag=True
if flag:
   a += 1
else:
   b+=1
```

Reduction in strength

Often operations can be replaced by more efficient alternatives (Aho et al., 2007). For instance, `x**5` is more efficient than making a function call `pow(x,5)`.

Loop unrolling

Since condition checking of a loop is an overhead, if the loop runs for a small constant number of times, it is more efficient to eliminate the loop construct and instead repeat the code the required number of times (Aho et al., 2007).

Difficulties

Compiler optimization involves solving problems such as instruction scheduling, loop fusion, and register allocation. On many machines, the order of execution of instructions strongly influences their total execution time. Compilers need to take advantage of the inherent parallelism that can be exploited in scheduling while simultaneously ensuring

correctness. Loop fusion merges two or more loops resulting in a merged loop that often runs faster. Register allocation maps values to hardware registers during code generation. All these processes include problems that are NP-complete. Heuristics are applied to obtain practical solutions in a reasonable time.

6.5 Code Coverage

Commercial software goes through several stages of testing, which is resource intensive. Hence, organizations need measurable ways of determining testing completeness. “Code coverage” is a software metric that tries to quantify to what extent the software is verified by measuring the degree to which a suite of tests exercises a software system. It applies to any stage of testing, such as unit or integration testing. Usually, 70 to 80 percent coverage is considered acceptable, though critical applications may demand a higher coverage.

To measure coverage, we need to first identify what part of the software is under consideration: a file, module, library, or system.

The Metrics

The coverage can be counted at various levels of granularity in terms of the following (Qian Yang et al., 2009):

- statements or lines of code. These are commonly used measures.
- blocks. Here, a sequence of non-branching instructions is considered as a unit, and we measure how many of them are covered.
- classes, branches, and functions.
- loops. Coverage analyses how many loops are executed zero, one, or more times.

Python Tool

Let’s look at the popular tool Coverage.py, which provides support for measuring code coverage in Python, and illustrate its usage with an example. Consider the following Python function to find the highest power of a given number that is a factor of a second given number:

Code

```
def powersOfFactor(num, fact):
   if(num == fact):
       return 1
   elif num % fact == 0:
       return(powersOfFactor(num//fact,fact) + 1)
   return 0
```

Let us assume that this function is in a file `factors.py`. Let us create another file `test_factors.py` with the following code:

Code

```
from factors import powersOfFactor
powersOfFactor(8,8)
```

We execute the command `coverage run test_factors.py` followed by the commands `coverage report` and `coverage html`. A file `test_factors_py.html` is generated in the folder `htmlcov`. Coverage is 57 percent and the remaining 43 percent not covered by the test case is marked. Then, we add another test case `powersOfFactors(1024,2)` and repeat the process. The code coverage now improves to 86 percent. Finally, adding the third test case `powerOfFactors(10,3)` gives us a 100 percent test coverage.

Figure 36: Test Coverage Report

Coverage for **factors.py** : 57%

7 statements | 4 run | 3 missing | 0 excluded

```
1  # -*- coding: utf-8 -*-
2  """
3  Spyder Editor
4
5  This is a temporary script file.
6  """
7
8
9
10 def powersOfFactor(num, fact):
11     if(num == fact):
12         return 1
13     elif num % fact == 0:
14         return(powersOfFactor(num//fact,fact) + 1)
15     return 0
```

Coverage for **factors.py** : 86%

7 statements | 6 run | 1 missing | 0 excluded

```
1  # -*- coding: utf-8 -*-
2  """
3  Spyder Editor
4
5  This is a temporary script file.
6  """
7
8
9
10 def powersOfFactor(num, fact):
11     if(num == fact):
12         return 1
13     elif num % fact == 0:
14         return(powersOfFactor(num//fact,fact) + 1)
15     return 0
16
```

Coverage for **factors.py** : 100%

7 statements | 7 run | 0 missing | 0 excluded

```
1  # -*- coding: utf-8 -*-
2  """
3  Spyder Editor
4
5  This is a temporary script file.
6  """
7
8
9
10 def powersOfFactor(num, fact):
11     if(num == fact):
12         return 1
13     elif num % fact == 0:
14         return(powersOfFactor(num//fact,fact) + 1)
15     return 0
16
```

Source: Prosenjit Gupta (2022), based on Batchelder (2022).

6.6 Unit and Integration Testing

During the process of software development, a software system needs to be regularly tested to discover bugs and defects (Sommerville, 2016). The testing process includes unit and integration testing.

Unit Testing

Unit testing includes testing individual functions, classes, and methods with different parameters. The wording "unit" testing does not imply that a notion of unit is defined in the programming languages, but simply that it refers to a small part that can be tested. When testing a class, all parameters need to be checked, all attributes need to be set, and all values verified. When using inheritance, operations need to be verified in subclasses as well. Unit tests have a dual role: They should demonstrate the correct expected behavior and also discover bugs.

Some accepted best practices for creating test cases (Whittaker, 2009) are as follows:

- those leading to the generation of all possible error messages
- those causing input buffers to overflow
- those having the same sequence of inputs several times
- those resulting in invalid outputs being generated
- those generating extremely small or extremely large numeric outputs

The UnitTest framework

Programming languages have supporting "unit testing frameworks" that make it easier to build, maintain and automate unit tests. Python is supported by frameworks like Pytest, UnitTest, and Nose. One example based on UnitTest is presented below.

Consider a supermarket that maintains a list of their customers and awards points to them from time to time during promotional offers. The points can be redeemed against purchases. Consider an example: During one such promotion, the supermarket, Green and Fresh, decides to award 50 points to all customers in the age group of $[18{,}25]$ whose point balance is currently nil. In the example below, we define a Python class for the customer data, and also define a method `offer()` to calculate if an offer is being made to a customer and, if so, update their points balance. The code is as follows:

Code

```
import unittest

class CustData:
   def __init__(self, ID, age):
       self._ID = ID
       self._age = age
       self._points = 0
```

```
    def get_ID(self):
        return self._ID

    def get_age(self):
        return self._age

    def get_points(self):
        return self._points

    def update_points(self,r):
        self._points+=r

    def offer(self, low, high, amt):
        if(self._age in range(low,high+1)):
            if(self._points == 0):
                self.update_points(amt)
            else:
                return False
        return True
```

For testing purposes, we create a dataset of four customers. We add 50 points to the balance of the customer with the ID `1322`.

Code

```
custList=[]
custList.insert(0,CustData(1555,18))
custList.insert(1,CustData(1322,23))
custList[1].update_points(50)
custList.insert(2,CustData(1687,25))
custList.insert(3,CustData(3231,53))
```

The first three customers are in the target age group for the current promotion, but the second customer already has a non-zero balance and hence will not receive an offer. The fourth customer is not in the offer's target group. We create four tests to capture this behavior, using Python "assertions". Assertions are statements that must be true in a program. The Python assert statement has an associated condition and an optional error message. If the condition is not satisfied, the program halts and reports an `AssertionError`. The optional error message, if specified, is also printed. The code is as follows:

Code

```
def test_offer():
    assert custList[0].offer(18,25,50) == True,\
        "test_offer0_FAIL"

    assert custList[1].offer(18,25,50) == False,\
        "test_offer1_FAIL"
```

```
    assert custList[2].offer(18,25,50) == True,\
        "test_offer2_FAIL"

    assert custList[3].offer(18,25,50) == False,\
        "test_offer3_FAIL"

test_offer()
```

The following statements allow us to run this from the command line with the command `python -m unittest`:

Code

```
if __name__ == '__main__':
    unittest.main()
```

UnitTest reports a `FAIL` with an `AssertionError: test_offer3_FAIL` message. This happens due to a bug in our `offer` method. Once we correct the code as follows, the test passes, and UnitTest reports an `OK`.

Code

```
def offer(self, low, high, amt):
        if(self._age in range(low,high+1)) \
            and (self._points == 0):
                self.update_points(amt)
                return True
        else:
                return False
```

Integration Testing

In integration testing, previously tested individual units are integrated into larger components with the focus being on testing the program with its interfaces (Somerville, 2016). In software development, several interacting objects are combined into larger components. Access to the object functionality is achieved via component interfaces. Assuming unit testing on individual objects has been performed, the focus is then on testing the interfaces and the components as a group to determine if they work together as required (Somerville, 2016). The following interfaces should be tested:

- parameter interfaces through which components exchange data and function references
- shared memory interfaces in which components share a block of memory
- procedural interfaces wherein one component encapsulates procedures or functions that are called by other components
- message passing interface

Errors can result from the calling component passing wrong parameter types, an incorrect number of parameters, or parameters in an incorrect order. Errors can also occur due to the calling component not sending parameters satisfying some required properties. For instance, the calling component may invoke a function with an unordered list when an ordered list is required.

Interface testing can be difficult since any defects may show up only under certain conditions depending on the behavior of other components.

6.7 Heap Analysis

Heap
A heap is a block of storage wherein portions are dynamically allocated and freed.

Managing storage for data is one of the major concerns of programmers and language designers. **Heap** storage is required because of language features allowing storage to be allocated or freed at arbitrary points in the program resulting from the creation, updates, and deletion of data structures. This requires addressing various problems related to allocation, compaction, and reuse of storage.

Python Heap Analysis

Investigation of performance bottlenecks often requires analysis of memory usage. Heap analysis tools enable the programmer to take corrective actions. Python relies on its own memory management.

Memory Profiler

The memory usage of a process is monitored by a Python module called the Memory Profiler. A line-by-line analysis of memory consumption is generated by the Memory Profiler. It is built on top of the Python library `psutil` (process and system utilities), which monitors and retrieves information on running processes and system utilization (Python Software Foundation, 2021b).

To use the Memory Profiler, we can invoke Python from the command line as

```
python -m memory_profiler filename.py
```

Decorator
This is a line added on top of classes or function declarations that annotates them without changing their essential nature. The more general term annotations is present in many programming languages.

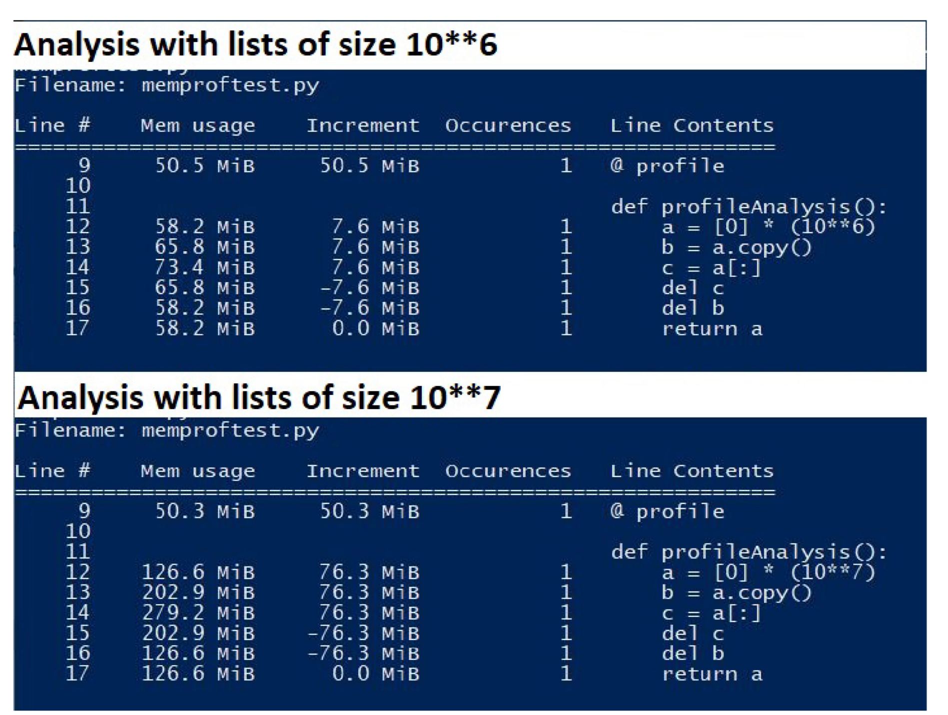
With the **decorator** `@profile`, the functions being profiled can be marked. Here is an example usage:

Code
```
from memory_profiler import profile

@ profile

def profileAnalysis():
   a = [0] * (10**7)
   b = a.copy()
```

```
    c = a[:]
    del c
    del b
    return a

if __name__ == '__main__':
    profileAnalysis()
```

The output is depicted in the figure below.

Figure 37: Using the Memory Profiler in Python: Example One



Analysis with lists of size 106**

```
Filename: memproftest.py

Line #    Mem usage    Increment  Occurences   Line Contents
============================================================
     9     50.5 MiB     50.5 MiB           1   @ profile
    10
    11                                         def profileAnalysis():
    12     58.2 MiB      7.6 MiB           1       a = [0] * (10**6)
    13     65.8 MiB      7.6 MiB           1       b = a.copy()
    14     73.4 MiB      7.6 MiB           1       c = a[:]
    15     65.8 MiB     -7.6 MiB           1       del c
    16     58.2 MiB     -7.6 MiB           1       del b
    17     58.2 MiB      0.0 MiB           1       return a
```

Analysis with lists of size 107**

```
Filename: memproftest.py

Line #    Mem usage    Increment  Occurences   Line Contents
============================================================
     9     50.3 MiB     50.3 MiB           1   @ profile
    10
    11                                         def profileAnalysis():
    12    126.6 MiB     76.3 MiB           1       a = [0] * (10**7)
    13    202.9 MiB     76.3 MiB           1       b = a.copy()
    14    279.2 MiB     76.3 MiB           1       c = a[:]
    15    202.9 MiB    -76.3 MiB           1       del c
    16    126.6 MiB    -76.3 MiB           1       del b
    17    126.6 MiB      0.0 MiB           1       return a
```

Source: Prosenjit Gupta (2022), based on Pedregosa (2021).

Note that in the second case, as we increased all list sizes by an order of magnitude, the corresponding numbers also increased in the “Increment” column, which shows the increase and decrease in memory usage as lists are dynamically created and deleted.

To generate and plot the memory usage over time, we can invoke the Memory Profiler as follows:

Code

```
mprof run filename.py
mprof plot
```

The resultant plot for the above program is depicted in the figure below.

Figure 38: Memory Usage over Time

Source: Created on behalf of IU (2022).

Consider another profiled program, as depicted below.

Code

```
@profile
def profileAnalysis():
    a=[]
    b=[]
    for i in range(0,10**5):
        a.append(0)
        b.append(0)
    c = a.copy()
    d = a[:]
    del d
    del c
    del b
    return a
```

Figure 39: Using the Memory Profiler in Python: Example Two

```
Line #    Mem usage    Increment  Occurences   Line Contents
============================================================
     9     16.9 MiB     16.9 MiB           1   @profile
    10                                         def profileAnalysis():
    11     16.9 MiB      0.0 MiB           1       a=[]
    12     16.9 MiB      0.0 MiB           1       b=[]
    13     18.2 MiB      0.0 MiB      100001       for i in range(0,10**5):
    14     18.2 MiB      0.7 MiB      100000           a.append(0)
    15     18.2 MiB      0.2 MiB      100000           b.append(0)
    16     18.2 MiB      0.0 MiB           1       c = a.copy()
    17     18.6 MiB      0.4 MiB           1       d = a[:]
    18     18.1 MiB     -0.5 MiB           1       del d
    19     17.7 MiB     -0.4 MiB           1       del c
    20     17.3 MiB     -0.4 MiB           1       del b
    21     17.3 MiB      0.0 MiB           1       return a
```

Source: Prosenjit Gupta (2022), based on Pedregosa (2021).

In this case, c is an alias of a, so the creation of c does not require any incremental memory. But d is a copy of a, so the creation of d requires an incremental memory. Finally, d, c, and b all point to independent memory, hence their deletions release memory. Note that here the statistics would vary with the runs. The internal implementation of the Python list does not allocate memory for each append. How the memory is allocated and freed depends on internal memory management algorithms.

SUMMARY

Type inferencing is a feature in programming languages, wherein the type of an expression is derived when not explicitly specified by the programmer. An example of a programming language with strong type inferencing capabilities is ML. There are, however, difficulties with type inferencing in a dynamically typed language like Python.

A software metric called cyclomatic complexity is an important element of the analysis of code. This measures the testability of a program by counting the number of linearly independent paths in the control flow graph of the program. The cyclomatic complexity can be computed using tools, such as the Python tool “Radon”.

Various tools can be used to generate software documentation, including Doxygen, Sphinx, Javadoc, Swagger, pdoc, and Pydoc.

Despite efficiency being a key concern for programmers, code can often be optimized automatically. Compiler optimization techniques include local and global optimization, making small functions inline, taking

repetitive computations outside loops, eliminating common subexpressions, eliminating redundant stores, eliminating unreachable code, reduction in strength, and loop unrolling.

“Code coverage” is a software metric that tries to quantify to what extent the software is verified by measuring the degree to which a suite of tests exercises a software system. The Python tool Coverage.Py can be used for this purpose.

During the process of software development, a software system needs to be regularly tested to discover bugs and defects. The testing process includes unit testing and integration testing.

Finally, heap analysis or memory profiling must be performed. Investigation of performance bottlenecks often requires analysis of memory usage, and heap analysis tools, such as the Python Memory Profiler, enable the programmer to take corrective actions. These tools are used to monitor memory consumption of a process and generates analysis line-by-line, as well as over time.

UNIT 7

PROGRAMMING LANGUAGES

STUDY GOALS

On completion of this unit, you will be able to …

- differentiate between various programming paradigms.
- understand the process of program execution.
- classify programming languages based on paradigms.
- analyze program syntax, semantics, and pragmatics.
- infer types of variables using type system rules.

7. PROGRAMMING LANGUAGES

Introduction

Over the years, many languages have been designed and implemented. The study of programming languages is not just about studying the syntax of individual languages in isolation, though. When mapping a problem's solution to a computer program, the programmer must choose an appropriate programming language and then express the solution efficiently in the chosen language.

While languages support a wide variety of features, a few programming styles, or paradigms, common to many languages have gradually emerged and evolved. Each paradigm has its distinct advantages and disadvantages and is more suitable for certain types of applied algorithms than others. Each paradigm also requires support in the form of certain programming language features for effective usage. There are programming languages that are exclusively meant for programming in one paradigm. Haskell, for example, is a purely functional language. Some languages primarily provide support for one paradigm more than other paradigms. For instance, Lisp is primarily a functional programming language, although modern dialects of Lisp have features of imperative programming. Many languages like Python or C++ are deemed to be multi-paradigm and can be used to implement programs in different paradigms according to the requirement.

There are concepts, such as "lazy evaluation" that are pervasive across languages and can lead to code improvement, if used appropriately. Within the same language, there are often alternative features to choose from, such as multiple loop constructs, for example. A good understanding of different programming paradigms and certain key concepts and features that are common to many programming languages is extremely useful for good programming.

7.1 Programming Paradigms

Different programming patterns, or paradigms, have evolved over the years. For a particular class of problems, programmers often find one programming paradigm more suitable than another. A programming language is deemed to provide support for a particular paradigm if it provides features that facilitate programming in that paradigm. The effort required by the programmer to solve a programming problem in a particular paradigm varies from language to language. The main paradigms of programming include

- imperative programming,
- object-oriented programming,
- functional programming,
- logic programming,
- programming for streaming data, and
- event-driven programming.

Imperative Programming

The imperative paradigm of programming is command-driven, sometimes referred to as statement-oriented. Following the von Neumann model of a computer whereby a program is executed as a sequence of instructions, a program in imperative programming is written as a sequence of statements, expressing commands for the computer to perform. As such, it can be considered to be the oldest paradigm. The statements result in a change of values in one or more memory locations. Most programming languages follow this paradigm, which essentially takes advantage of the von Neumann architecture. The computer's memory stores the instructions of the program, as well as the data on which these instructions act. The instructions include the assignment operator as a central construct. The program is viewed as a list of instructions that continually changes the memory state until a goal state is reached. In its simplest form, the imperative style is difficult to scale. Language support for this style of programming includes features such as variable declarations, expressions, control structures for selection, iteration and branching operations, and procedural abstractions.

Consider the problem of partitioning an array of integers, which involves rearranging them so that integers less than or equal to a pivot appear before those that are greater. This problem is fundamental and forms a building block of other algorithms, such as quicksort and various selection algorithms (Cormen et al., 2009). Below is a Python implementation of this algorithm using a simple imperative style.

partition1.py

Code

```
L=[11,59,26,17,2,1,25,9,3,15]
pivot = 11
i=0
j=len(L)-1
while True:
   while (i <= j) and (L[i] <= pivot):
       i+=1
   while (i <=j) and (L[j] > pivot):
       j-=1
   if(i <= j):
       L[i],L[j] = L[j],L[i]
   else:
       break
print(L)
```

This code can be placed inside a function, with the list and pivot as parameters.

partition2.py

Code
```
def partition(L,p):
   i=1
   j=len(L)-1
   while True:
       while (i <= j) and (L[i] <= p):
           i+=1
       while (i <=j) and (L[j] > p):
           j-=1
       if(i <= j):
           L[i],L[j] = L[j],L[i]
       else:
           break
   return L
aList=[11,59,26,17,2,1,25,9,3,15]
partition(aList,11)
```

Object-Oriented Programming

In the object-oriented paradigm, the first task is to identify the fundamental objects in the design. Then, an abstraction is created, keeping the implementation details hidden. Language features supporting the object-oriented paradigm facilitate the creation of classes to implement these objects.

Let us revisit the same problem of partitioning and consider a Python implementation using the object-oriented paradigm. We encapsulate our solution in a class called `Scores`. Then, as shown in the example below, we invoke the constructor for `Scores` and create an object called `bList`, which is an instance of the class `Scores`. Finally, we invoke the partitioning method by a call to `bList.part()`. The code is as follows:

partition3.py

Code
```
class Scores:
   def __init__(self,L):
       self.S = L

   def part(self,p):
       i=0
       j=self.size()-1
       while True:
           while (i <= j) and (self.S[i] <= p):
               i+=1
           while (i <=j) and (self.S[j] > p):
               j-=1
```

```
            if(i <= j):
                self.S[i],self.S[j] = self.S[j],self.S[i]
            else:
                break
        self.S[0],self.S[j]= self.S[j],self.S[0]
        return self.S

    def isEmpty(self):
        return self.S == []

    def size(self):
        return len(self.S)
bList =Scores([11,59,26,17,2,1,25,9,3,15])
print(bList.part(11))
```

Functional Programming

Functional programming models a problem of computation as a collection of mathematical functions. Here, we consider a Python implementation of the partitioning problem using the functional style. In the functional style of implementation, we can make use of constructs such as `lambda`, `map`, `reduce`, and `filter`.

Python lambda, map, reduce, and filter

The `lambda` construct offers a way to define anonymous functions in Python. Consider the following `lambda` function which adds `13` to an argument:

Code

```
lambda a : a + 13
```

We apply this to a parameter `11` to get a result `24`:

Code

```
(lambda a : a + 13)(11)
```

The `map` function in Python has a syntax map `(f, iter)`, where `iter` represents one or more iterables and `f` is a function that is applied to each element of the iterables.

The `reduce` function has a syntax `reduce(f, iter[,initial])`, where `iter` represents one iterable and `f` is a two-argument function that is cumulatively applied to each of its elements. This can be used, for example, to apply an aggregation function, such as sum, to the elements in a list. The optional argument `[,initial]` can be used to specify an initial value of the aggregation.

The `filter` function in Python has a syntax `filter(f, iter)`, where `iter` represents an iterable and `f` is a Boolean-valued function that is applied to each element of the iterable. Only those items `x` in iter for which `f(x)` is true are output.

Functional style partitioning in Python

Let's apply the Python functional constructs to solve the partitioning problem. First, we define a lambda function to filter out the elements `x <= pivot`:

Code
```
list(filter(lambda x: x<= p,L))
```

This is followed by another lambda function to filter out elements `x > pivot`:

Code
```
list(filter(lambda x: x > p,L))
```

Finally, we concatenate the two results and wrap them around another lambda function, to get our final solution, as follows:

partition4.py

Code
```
aList=[11,59,26,17,2,1,25,9,3,15]
pivot=11
ans=(lambda L,p: list(filter(lambda x: x<= p,L)) + \
     list(filter(lambda x: x> p,L)))(aList,pivot)
print(ans)
```

Logic Programming

Logic programming follows a declarative style of programming. Programs written in such languages specify the goals of the computation rather than details of an algorithm to reach the goal (Tucker & Noonan, 2007). Logic programming is completely non-procedural. The programmer cannot give detailed instructions on how the computation needs to be done. Here, we solve the partitioning problem in Python using the logic programming paradigm, making use of the kanren package in Python. First, we collect all elements less than the pivot in list `a`, elements equal to the pivot in `b`, and elements greater than the pivot in `c`. We concatenate the results into list `d`. Note that we did not give detailed instructions using loop constructs or similar devices to implement the individual steps but rather specified the goals of the computation.

Putting it all together:

Code
```
from kanren import run, eq, membero, var, conde
from kanren.arith import lt,gt
x =var()
L=[11,59,26,17,2,1,25,9,3,15]
pivot = 11
```

```
a=run(0,x,(membero, x, L),(lt,x,pivot))
b=run(0,x,(membero, x, L),(eq,x,pivot))
c=run(0,x,(membero, x, L),(gt,x,pivot))
d=run(0,x,(membero,x,a+b+c))
print(d)
```

Data Stream Programming

Objects that support looping through a sequence of values are called “iterables”. Examples in Python include collection structures, as well as lists, tuples, sets, and dictionaries. Functions that iterate through an iterable object are known as “iterators”. One problem associated with iterables is that all data need to be stored in memory before we can iterate through them in a loop. There are situations where we may not need the entire sequence after all and may break out of the loop after processing a few elements.

A “data stream” is a sequence of data items that are available one at a time. Python supports processing such streams of data by using a construct called the “generator” (Goodrich et al., 2013). Consider the following function to generate the sequence of Fibonacci numbers.

fibGen.py

Code

```
def generateFib():
   one = 0
   other = 1
   while (1):
       yield one
       another = one + other
       one = other
       other = another
gen = generateFib()

def getLessThan(g, n):
   i=next(g)
   while i < n:
       print(i)
       i=next(g)

getLessThan(gen, 100)
```

The statement `gen = generateFib()` creates a generator for all Fibonacci numbers. The function call `next(gen)` gets the next item from the stream of Fibonacci numbers and the function call `getLessThan(gen,100)` gets all the Fibonacci numbers less than 100.

Event-Driven Programming

Event-driven programming is a programming paradigm wherein the control flow is determined by events, such as mouse clicks. It is the principal programming paradigm used in Graphical User Interface (GUI) programming. In this paradigm, usually, there is a main loop that looks for events and calls a special function whenever an event is detected (Liang, 2017). An example in Python is shown below.

event.py

Code

```
from tkinter import *
def clickButton():
   print("Button Clicked")

w = Tk()
l = Label(w, \
         text = "Event-Driven Programming")
b = Button(w, \
          text = "Click here",command=clickButton)
l.pack()
b.pack()
w.mainloop()
```

A label and a button are created and displayed as shown below. We define a function `clickButton()` that is bound to the defined button. When the user clicks the button, the program processes this event via the callback function `clickButton ()`. The program then prints “Button Clicked”.

Figure 40: Tkinter Label and Button

Source: Created on behalf of IU (2022).

7.2 Execution of Programs

The operating system controls and monitors various system-level activities. It also allocates shared resources to programs, including the central processing unit (CPU), random-access memory (RAM), disks, and input/output (I/O) devices. The operating system takes care of scheduling various actions of programs that require the usage of these shared resources.

The Fetch and Execute Cycle

All programs are finally converted to machine language, which is a set of basic instructions in binary code. The CPU runs the program in a sequence of “fetch” and “execute” commands: Fetch the next instruction, execute it, and repeat the process in a cyclic manner. Some instructions are control instructions, which determine the order in which the CPU executes the instruction sequence. This may require the CPU to go back to an earlier instruction in the sequence while executing a loop. It may also skip certain statements while executing a conditional instruction.

The CPU allocates an area in RAM where the program is loaded. It also allocates space for data. The typical layout of a **process** in memory is shown in the figure below.

Process
A process is a program in execution.

Figure 41: Typical Layout of a Process in Memory

Source: Created on behalf of IU (2022).

The execution of a program begins from the first instruction. The CPU outputs the address of the memory location containing the next instruction. This is stored in the program counter. The logic for the address decoding helps select not only the RAM chip, but also the location allocated to the concerned address. The code for the instruction is retrieved from RAM into the CPU via the data bus. This is read into an instruction register. The contents of the register are decoded by the CPU, and instruction processing is initiated. The data or operands on which the instruction must act are fetched from RAM via the data bus, similar to how the instruction was fetched.

Once the operands have been fetched, they are processed by the data processing logic in the CPU. More data items may be required to be fetched, depending on the instruction. The partial results are stored in data registers and may be required to be written back into RAM. The program counter is then incremented to the address of the next instruction. The operating system, program code, and data are all in RAM at the time of execution.

Executing Multiple Programs

The operating system also provides support for concurrency, which allows multiple programs to work together. This includes

- multithreading, which allows multiple tasks belonging to the same program.
- multitasking, which refers to multiple programs working on the same processor.
- multiprocessing, whereby multiple processors are available.

A process with a single thread has a single program counter. The execution of the process proceeds sequentially until termination, one instruction at a time. A program with multiple threads also has multiple program counters, one for each thread. As the program executes, the process changes between the following various states:

- new. The process is being created.
- ready. The process is ready to be scheduled.
- running. The process is executing.
- waiting. The process is waiting for something, for example, I/O.

The CPU executes a scheduling algorithm to select a program to run. CPU scheduling algorithms are at the heart of multiprogramming. In a CPU with a single core, only a single process runs at a time, while other processes wait for the CPU to be free. Typically, a process must wait to complete an I/O request. The CPU would remain idle during this time unless it can be engaged in some other activity. The operating system uses this time to schedule another process.

The scheduling algorithm may be preemptive or non-preemptive. In non-preemptive scheduling, a program continues to run on the CPU until terminated, or while it is forced to wait for an I/O or the termination of another process. Otherwise, scheduling is preemptive: a program can be interrupted after a fixed amount of time in order to allow other programs to run. Once the CPU scheduler has selected the process to be scheduled, the dispatcher gives control of the CPU core to the process. Many CPU scheduling algorithms are chosen based on properties such as optimization of throughput, CPU utilization, turnaround time, and waiting time. Some basic algorithms include

- first-come, first-served. CPU is allocated to the process that requests it first.
- shortest job first. The job selected is that whose next CPU burst is the smallest.
- round-robin. Each process is selected in turn and the CPU is allocated for a quantum of time.
- priority scheduling. Each process has a priority and the process with the highest priority is selected to be scheduled.

7.3 Types of Programming Languages

There is no standard classification of programming languages, but a paradigm-based classification is the most natural. The main paradigms of programming are imperative programming, object-oriented programming, functional programming, and logic programming. Although many languages are **multi-paradigm**, languages often primarily support one paradigm over others.

Multi-paradigm
A multi-paradigm programming language is one that supports multiple programming paradigms.

Imperative Programming Languages

Most programming languages follow this paradigm, which is modeled on the von Neumann architecture. The program is written as a sequence of statements expressing commands for the computer to perform. The statements result in a change of values in one or more memory locations. Imperative programming is the oldest approach and closest to the actual behavior of computers. Some of the early imperative languages were, in fact, close to assembly languages.

Language examples

C, FORTRAN, Pascal, Ada, JavaScript, PHP, and Ruby are examples of imperative languages. Other multi-paradigm languages that also support the imperative style include Python and C++.

Key features

Language support for this style of programming includes features such as procedural abstractions, variable declarations, expressions, control structures for selection, iteration, and branching operations.

The imperative style evolved with procedural abstractions at its core. The programmer starts with a specification of a function along with its input and output parameters. This allows the developer to concentrate primarily on the interface between the function and what it computes, and to ignore the algorithm and details of how it was computed (Tucker & Noonan, 2007). This leads to the development of the program by a process of stepwise refinement. First, the programmer starts with a description of the program to be written along with its input and output specifications. This is then broken down hierarchically into smaller functions to be implemented.

At the heart of the syntax of all imperative languages is the assignment statement, which takes the following form:

Code
```
variable = expression
```

The expression is evaluated, and the output value is copied to the left-hand side. When the right-hand side is also a variable, we need to distinguish between the cases where the assignment merely creates an alias or a separate copy of the right-hand side.

Consider the following Python code fragment:

alias.py

Code
```
a = [2,3,4,5]
b = a
b[0]=-1
print(a)
print(b)
```

This prints `[-1, 3, 4, 5]` twice, because `b` is just an alias for `a`.

Now consider a modified version:

copy.py

Code
```
a = [2,3,4,5]
b = a[:]
b[0]=-1
print(a)
print(b)
```

The above code prints `[2, 3, 4, 5]` followed by `[-1, 3, 4, 5]`. Here, `b` is a copy of `a`.

The former is called "assign by reference" or "reference semantics." The latter is called "assign by copy" or "copy semantics." The latter is more common among imperative languages.

Expressions are composed using arithmetic and logical operators, as well as built-in functions of the language. In C, the assignment is also an operator, which returns a value and hence can appear in an expression. This allows statements such as `a = b = c`.

Object-Oriented Languages

In general, an abstraction is a representation of an entity that includes a subset of significant attributes, while filtering out others, in order to create an abstract idea of the entity appropriate to the program being created. A data structure has an associated abstract data type that specifies what data is stored within the data structure and what operations are supported to be performed on the data. For users of the data structure, the abstract data type serves as a public interface. The implementation details of the data structure are not exposed in the abstract data type specification. To implement abstract data types in a language, an appropriate syntactic structure is required to enable the clients of the abstraction to declare instances of the abstract data type and execute various operations on them. Such a syntactic structure is provided by object-oriented languages.

Language examples

C++, Java, Python, and Ruby are some examples of object-oriented languages.

Key features

In object-oriented programming, a basic means of abstraction that supports the creation of user-defined types is the class. Instantiations of classes are called objects. How data are represented is determined by the class. The actual data are stored by the object. The class also has defined member functions, also called methods.

Inheritance adds a strong feature to object-oriented programming. It helps to modularly and hierarchically organize the classes. We can define new classes using inheritance by deriving them from an existing class, also called the base class or superclass. The new class is called the derived class or subclass. These derivations create a hierarchy of classes.

Another feature of object-oriented programming is polymorphism, which allows us to use the same function or method with arguments of different types. An object of a derived class can be used as a parameter where a parameter of a superclass in the inheritance hierarchy is expected.

Language support for object-oriented languages includes facilities for encapsulation and information hiding. Data defined in a class are accessible through member functions. Access to data across the class hierarchy is controlled. For example, in C++, this access is restricted by classification into public, private, or protected. Data members or member functions of classes labeled public are accessible wherever the class is instantiated. The ones labeled private are accessible only inside the class itself. The protected label extends the access to derived classes as well.

Functional Programming Languages

The development of functional programming languages started in the 1960s. These were widely used for symbolic computation, rule-based systems, theorem proving, natural language processing, and artificial intelligence-related fields. It was felt that the needs of the researchers in these areas were not being met by the imperative languages that were then available (Tucker & Noonan, 2007).

Language examples

Lisp, Haskell, Scheme, ML, OCaml, and F# are some examples of functional programming languages.

Key features

The functional programming style is regarded as the first significant departure from the imperative style of programming. Lisp is the most widely used functional programming language. It began as a pure functional programming language, but its subsequent dialects incorporated various imperative features to improve computational efficiency.

A key feature of the imperative style is the notion of state, which is captured by values of variables. This needs to be tracked during development. A pure functional programming language has no variables or state (Sebesta, 2016). Without variables, iterative loops cannot be implemented as in imperative languages. These are implemented indirectly using recursion (Sebesta, 2016).

In functional programming, a computation is viewed as a mathematical function mapping its arguments to outputs. A functional language typically provides some built-in functions, a mechanism to create more complex functions from the primitive ones, and a function application operation.

Logic Programming Languages

Logic programming emerged as a strong non-imperative paradigm in the 1970s. It is also known as rule-based programming. It has been used in applications, such as expert systems, natural language processing, and database query retrieval.

Language examples

Prolog is the most well-known and widely used logic programming language. Other examples include ALF, Alice, and Datalog.

Key features

Logic programming languages follow a declarative style of programming. Programs written in such languages specify the goals of the computation rather than details of an algorithm to reach the goal (Tucker & Noonan, 2007). The goals are specified as a collection of rules and constraints in symbolic logic, rather than assignments and control flow statements. The language needs to support a mechanism to specify the rules and the goal as logical statements, as well as an inference mechanism to reach the goal. In Prolog, for instance, the representation of the rules and facts uses first-order predicate logic. The inference mechanism employs a process of "resolution". This involves creating a negation of the goal and reaching a contradiction by repeated application of a simple rule: `(A OR B) AND (~A OR C)` implies that `(B OR C)` is true. Whereas the programming effort is reduced in logic programming, logic programming languages can be slow. The efficiency of the solution is dependent on the efficiency of the inference mechanism.

7.4 Syntax, Semantics, and Pragmatics

A programming language should be both concise and clear to be widely adopted. Concise formal definitions may not be understood by all stakeholders. At the same time, simple and informal descriptions may lead to imprecise descriptions and create many variations of the language concerned. One difficulty is the diversity of the audience involved.

The form of the expressions, statements, and procedural units of a programming language is called its “syntax”, and the meaning of these syntactical units is called the “semantics”. “Pragmatics” refers to what the statements achieve in practice (Sebesta, 2016).

Example

Consider the syntax of the following while loop in Python:

Code

```
while boolean_expression:
        statement
```

The semantics of the while loop state that if the Boolean expression is true, the statement will be executed. If there are multiple statements in the same block, they would all be executed in order. Once this is completed, control returns to the Boolean expression for evaluation again. This is repeated until the expression evaluates to false. As an example, consider the following code snippet in Python:

while1.py

Code

```
n=13
while n < 20:
   n+=1
   print(n)
```

The semantics of the while loop states that when the current value of the Boolean expression is true, the statements in the scope of the while loop will be executed. Here, the while loop prints the integers from 14 to 20. When $n = 20$ the condition fails, and control leaves the while loop.

Now consider another variation of the same loop:

while2.py

Code

```
n=13
while n > 20:
   n+=1
   print(n)
```

The semantics of the while loop construct have not changed. However, this loop will not be executed since the Boolean expression will always be false.

A third variation is as follows:

while3.py

Code

```
n=21
while n > 20:
   n+=1
   print(n)
```

Here, the control enters the while loop but never leaves since the Boolean expression is always true. This is an **infinite loop**.

Infinite loop
An infinite loop is a loop that does not terminate.

This example illustrates a simple while loop construct, the syntax of which is well-defined and the semantics well understood. The pragmatics indicate different behavior under different conditions.

Short-Circuit Evaluation

Consider the following Python code fragment:

short1.py

Code

```
x = 20
y = 0
x > 20 and x/y < 5
```

The condition evaluates to `False`. Although the division by zero is not permissible, the Python interpreter is still able to evaluate the truth value of the expression. Since the subexpression `x > 20` is `False`, the result of a logical `and` of any Boolean expression with `False` would be `False`. So, the interpreter evaluates the whole expression to `False` without evaluating the second subexpression `x/y < 5`. This is an example of "short-circuit evaluation" or "lazy evaluation". This provides a special and important opportunity for

code improvement and increased readability. If the first subexpression is a very unlikely condition, and the second subexpression involves a very expensive function call, short-circuit evaluation leads to significant time savings (Scott, 2016).

If we modify the above code fragment by initializing `x` to `21`, `x > 20` is `True`. So, the second subexpression `x/y > 5` is evaluated and the interpreter flags a `ZeroDivisionError`. This can be corrected by introducing a “guard clause” as follows:

short2.py

Code
```
x = 21
x > 20 and y!= 0 and x/y < 5
```

The semantics of the `and` expression is that it is both commutative and associative, so the order of evaluation of the subexpressions should not matter. However, from the point of view of the pragmatics of the short-circuit evaluation, we may reorder such subexpressions to our advantage and improve the code.

Another advantage of short-circuit evaluations is how they evaluate expressions involving pointers. Consider the following example of traversing a linked list in C (Scott, 2016):

short3.c

Code
```
p = my_list;
while (p && p->key != val)
 p = p->next;
```

If `p` is `NULL`, the subexpression `p->key != val` is not evaluated. This works because of short-circuit evaluation.

Specifying Syntax

A language L is defined over an alphabet set Σ. The set of all strings that one can form using characters from Σ is denoted as Σ*. The language L is a subset of Σ*. The syntax rules of the language specify which strings are in L. At the lowest level, such strings, called lexemes, include elements such as numeric literals, operators, and operands. A program written in the language is a string of such lexemes. These lexemes are divided into groups called tokens. For example, for a statement like $i = 2j + 5$, the tokens in a language could include identifiers, integer literals, multiplication operators, and semicolons (Sebesta, 2016).

Languages may be defined by recognition or generation. To define a language by recognition, we need to construct a recognizer, which can be used to test whether a string is in the language or not. Parsers perform such tests. Grammars are used to define languages by generation. The forms of tokens can be described as “regular grammars”. The syntax of a

programming language can mostly be captured by context-free grammars (CFGs; Sebesta, 2016). Given such a grammar, a recognizer for the language concerned can be constructed using standard software. One of the early such systems was the Yet Another Compiler (YACC), which can construct a complier from the CFG specifications of a language. Such tools are useful for generating compilers for new or special-purpose languages.

An Ambiguity

Ambiguity in languages is often a necessary evil to avoid an explosion of rules in the underlying grammar. Although programming languages use mostly unambiguous syntax, there are notable exceptions. Let us consider the following two similar Python code fragments:

if_then_else1.py

Code

```
x=0
i=1
if i >= 0:
   if i==0:
       x=1
else:
   x=2
print(x)
```

The value printed on executing the above code is 0.

if_then_else2.py

Code

```
x=0
i=1
if i >= 0:
   if i==0:
       x=1
   else:
       x=2
print(x)
```

The value printed upon executing the above code is 2. The two `print` statements print differently although the code fragments have identical statements and are similar barring indentation. This is an example of a larger issue of syntactic ambiguity, which is: Which `if` block do we pair the last `else` with? Python resolves this by allowing the user to specify the correspondence by using appropriate indentation.

Languages such as C and C++ resolve this ambiguity by associating such a dangling else with the textually closest `if`, allowing the user to override this default behavior by using explicit braces.

Specifying Semantics

Although the grammar rules of a language can capture most syntactical rules, there are some which cannot be captured. An example is the rule in many languages that requires that variables must be declared before use. There are other rules specified by the type system that require complex rules of grammar to be captured. These rules are covered by static semantic rules of the language. These rules have more to do with the validity of program syntax than the meaning of the program execution. Such static semantic rules are specified and checked using the mechanism of attribute grammars (Sebesta, 2016). Static semantics is so-called because it can be checked at compilation time.

Dynamic semantics deals with the meaning of statements, expressions, blocks, and functions. Describing the dynamic semantics of a language is more difficult than describing static semantics. Precise semantic specifications could potentially lead to correct-by-construction programs and make testing redundant.

In operational semantics, the meaning of a statement, construct, or program is described by specifying what happens when it is executed on a machine. This may consider individual steps of the computation as in structural operational semantics or the overall results as in natural operational semantics (Sebesta, 2016). A rigorous formal method used in describing dynamic semantics is denotational semantics (Tucker & Noonan, 2007).

7.5 Variables and Type Systems

A type in a programming language defines a set of values along with a set of operations that act on those values. When associated with a variable, the type determines what values the variable can take. A function may also have a return type, which determines the set of values it can return. An operation has types associated with its operands and result. The value associated with a type is stored in memory as a sequence of bits. The type of the variable associates an interpretation with the sequence of bits as either a string, integer, or floating-point number. Types may be built-in or user-defined. The set of types varies from one programming language to another.

Scope of Variables

The scope of a variable defines the part of the program where the variable can be assigned or referenced. The scope rules of a language determine how a particular reference to a variable is associated with a declaration. The scope can be static or dynamic. In static scoping, also known as lexical scoping, the scope can be determined once the code is written, prior to execution. In dynamic scoping, the scope of the variable depends on the calling sequence of functions and hence can only be determined at runtime. Most modern languages support static scoping.

Scopes may be nested or disjoint. When the scoping is disjoint, the same name can be used for different entities. In C and C++, a block of statements enclosed within braces "{" and "}" defines a new scope (Tucker & Noonan, 2007). Blocks, but not functions, may be nested in C and C++. Scoping rules in Python, however, are based on functions, so nested functions are possible in Python.

Local and global scoping

Local variable
A variable that is accessible only in a specific part of a program is called a local variable.

In Python, the scope of a **local variable**within a function starts from the point of creation and ends with the last statement of the function (Liang, 2017). Many languages allow variable definitions to appear outside all functions but be visible everywhere. These are called global variables. In Python, global variables can be referenced in functions but can be assigned only if declared to be global (Sebesta, 2016).

global1.py

Code
```
x=1
def A():
   x=2
   print("A", x)
A()
print("Global", x)
```

Here, the two print statements print `2` and `1` respectively because the variable `x` inside function `A()` is local, and the assignment `x=2` does not change the value of the global variable `x`, which is assigned a value of `1`. Next, we change the variable inside function `A()` to be global. Now, both print statements print `2` because the `y` being updated in `A()` is the same variable as the `y` originally assigned to `1`. The code is as follows:

global2.py

Code
```
y=1
def A():
   global y
   y=2
   print("A", y)
A()
print("Global", y)
```

Consider an example of nested functions in Python. The values of `x` printed are `1,2,3,2,1`, in that order. The reassignment `x=3` inside function `B()` does not change the value of `x=2` inside `A` , as can be seen in the code below:

global3.py

Code

```
x=1
print("Global1", x)
def A():
   x=2
   print("A1",x)
   def B():
       x=3
       print("B", x)
   B()
   print("A2",x)
A()
print("Global2", x)
```

Scopes and namespaces

When an identifier is assigned a value in Python, a scope gets defined based on the location of the assignment statement. Each distinct scope is represented by an abstraction called the "namespace". When a variable is referenced in a statement in Python, the name resolution process searches the most locally enclosing scope first and gradually moves outward. We can retrieve information about the most locally enclosing namespace, stored as a dictionary, by using the `dir()` and `vars()` commands. Here, `vars()` returns the whole dictionary, whereas `dir()` returns the keys. The code is as follows:

namespace1.py

Code

```
x=1
def A():
   y=2
   def B():
       z=3
       print(vars())
       print(dir())
   B()
   print(vars())
   print(dir())
A()
```

Type Systems

The type system associated with a programming language defines the built-in types, as well as a set of rules for the creation of user-defined types and the usage of types in the language. A type error results when a type system rule is violated. Type checking, performed either at compile time or run time, is intended to detect type errors.

Type compatibility
The property that allows a value of one type to be acceptable when a value of another type is expected is called type compatibility.

Type checking tests **type compatibility** in various situations, including the following:

- compatibility between operands of an operation. Some rules define how the type of an expression is computed from the types of its constituents.
- assignment statements. Rules govern the relationship between the type of the variable on the left-hand side and the expression on the right-hand side of an assignment statement.
- compatibility between the actual and formal parameters of a function. For example, if a function expects an argument to be of a certain type according to its declaration, would an argument of another type be acceptable in the function call?

Compatibility may be ensured by explicit or implicit conversion to a legal type. Incompatibility results in a type error.

Explicit type conversion

While carrying out computations involving data of mixed type, it often becomes necessary to convert variables from one type to another. Type conversion may be explicit, with support from built-in functions within the language. For example, in Python, built-in functions `int()`, `float()` and `str()` convert arguments to integers, floating-point numbers, and strings, respectively, so we have `int(4.9) returns 4`,`float(4)` returns `4.0` and `str(3.2)` returns the string `'3.2'`.

Implicit type conversion

The type system of the language has rules that govern when values of one type are automatically converted into those of another, compatible, type. These conversions are known as “automatic type promotions” or “coercions”. This allows expressions of mixed but compatible types to be evaluated. For example, Python implicitly converts integers to floating-point numbers whenever required, as follows:

- The result of the expression `a + b` is an integer if both `a` and `b` are integers. However, the type of the result is a floating-point number if the type of at least one of `a` or `b` is a floating-point number.
- Consider the integer division `15//2`, which evaluates to `7`, but `15//2.0`, where `15` is implicitly converted into a floating-point number, is evaluated to `7.5`.
- Also, `bool(1==1)` is `True` and `bool(1==1) + 5` evaluates to `6`. Here, the intermediate expression `True + 5` is evaluated to `6`, since the Boolean value `True` is coerced into `1`.

SUMMARY
Different programming patterns, or paradigms, have evolved over the years, such as imperative, object-oriented, functional, and logic programming. A programming language may primarily support one of the paradigms; however, a multi-paradigm language like Python allows us

to program in different paradigms. The operating system provides support for the execution of programs: The CPU runs the program in a sequence of fetch-execute cycles.

Programming languages are often classified along their paradigm(s). There are certain key features of languages in each category. Language support for the imperative style of programming includes various features, such as procedural abstractions, variable declarations, expressions, control structures for selection, iteration, and branching operations. Object-oriented languages provide a syntactic structure to leverage abstract data types in the language, facilities for classes, and inheritance. Functional programming languages view a computation as a mathematical function mapping its arguments to outputs. A functional language typically provides some built-in functions, a mechanism to create more complex functions from the primitive ones, and a function application operation. Logic programming languages follow a declarative style of programming. Programs written in these languages specify the goals of the computation rather than the details of an algorithm to reach the goal.

Every programming language is characterized by syntax, semantics, and pragmatics, and programs need to be viewed through all three perspectives.

A fundamental aspect of programming is the set of scoping rules for variables in programs. This includes the notions of local scoping, global scoping, and namespaces, and how explicit type conversions and automatic type promotions allow expressions of mixed, but compatible, types to be evaluated.

UNIT 8

OVERVIEW OF IMPORTANT PROGRAMMING LANGUAGES

STUDY GOALS

On completion of this unit, you will be able to …

- develop compiled applications using WebAssembly for execution on the web.
- understand the features that distinguish C++ from C.
- distinguish between generic programming in C# and Java.
- compare and contrast functional programming in Haskell and Lisp.
- use HTML DOM to create a JavaScript application embedded in HTML.
- evaluate unique features of a range of imperative programming languages.

8. OVERVIEW OF IMPORTANT PROGRAMMING LANGUAGES

Introduction

Different programming languages have features that support programming paradigms, such as imperative, object-oriented, functional, or logical. Often, a language can be classified based on the paradigm, but it provides certain features that also support another paradigm. Languages have their unique features and so, over time, they have found usage in some specific application domains. WebAssembly enables the execution of compiled code on the web and supports many languages. C has been popular for systems programming because of its low-level features. C++ is a multi-paradigm language; it includes powerful constructs to support object-oriented programming. Java is an object-oriented language, with support for efficient memory management, multithreading, and distributed computing. The reach and power of Java influenced the design of C#, which was created by Microsoft for its .NET framework (Sebesta, 2016). Haskell and Lisp are two well-known functional programming languages. Haskell is a purely functional, statically typed language with lazy evaluation, list comprehension, and minimalist syntax, whereas Lisp is considered more flexible and is dynamically typed. Originally conceived as a functional alternative to Java, today JavaScript is a central language in web applications. JavaScript has been designed around the idea of the Document Object Model (DOM) with a hierarchy of parent and child objects. Ada, an imperative programming language, was also an important milestone in the development of programming languages because certain important features went on to influence the design of other programming languages. New languages continue to appear regularly. Knowledge of the key features of different languages will help us choose one that suits our needs for a particular requirement.

8.1 Assembler and Webassembly

Assembler

Assembler, or assembly language, is a low-level programming language. The instructions in the assembly language have a close relationship with the machine language instruction in the architecture being used. The assembly language code is converted to the machine code by a utility program, also known as the assembler. Usually, every instruction in the assembly language specifies a single machine language instruction. Assembly language makes low-level programming easier while resulting in more efficient code than what could be achieved through high-level languages. Typically, each line of the assembly language program has an optional label; a mnemonic for the instruction, such as MOV, JMP, or ADD; an optional list of operands; and an optional comment. The operand may be a list of data items or parameters. The assembly language program may also include data directives to hold data and variables. There may also be assembly directives to the assembler (the program translating the assembly code to the machine code) to perform operations

other than assembling. Since the assembly code is dependent on the machine code, every assembly language must be designed for a specific architecture. Below is a small C program and its corresponding assembly code in an 8086-like assembly language generated using the CtoAssembly tool. The comments in the assembly code are summarized from comments generated by the CtoAssembly tool.

CtoAssemblyTest.c

Code

```
int main ()
{
 int a = 1;
 int b = 2;
 int i = 0;
 while (i < 5)
 {
  a = a + b;
  i++;
 }
 return 0;
}

main:
   PUSH %BP ; Push base pointer onto stack
   MOV %SP, %BP; base pointer = stack pointer
@main_body:
   SUB %SP, $4, %SP ; reserve space on stack for a
   MOV $1, -4(%BP); set a = 1
   SUB %SP, $4, %SP; reserve space on stack for b
   MOV $2, -8(%BP); set b = 2
   SUB %SP, $4, %SP; reserve space on stack for i
   MOV $0, -12(%BP); set i = 0
@while0:
   CMP -12(%BP), $5; compare i with 5
   JGE @false0; exit while loop if i >= 5
@true0:
   ADD -4(%BP), -8(%BP), %0; compute a+ b
   MOV %0, -4(%BP); a = a + b
   INC -12(%BP); i++
   JMP @while0; control goes back to while loop start
@false0:
@exit0:
@main_exit:
   MOV %BP, %SP; stack pointer = base pointer
   POP %BP; pops value from stack
   RET
```

WebAssembly

WebAssembly (Wasm) enables execution of compiled code on the web without plug-ins and includes the following components:

- a binary module and format (.wasm format) for executable code
- a human-readable text format (.wat) for assembly code
- a compilation target

WebAssembly supports several compiled and interpreted languages. The syntax of the program is made of symbolic expressions or S-expressions. As an example, consider a C-function computing Fibonacci number as follows:

fib.c

Code

```
int fib(int n)
{
 int curr, next, sum;
 curr = 1;
 next = 1;
 for(int i = 1; i <= n-2; i++) {
    sum = curr + next;
    curr = next;
    next = sum;
 }
 return sum;
}
```

The tool WasmFiddle (Rourke, 2018) was used to generate the WAT code below.

fib.wat

Code

```
(module
 (table 0 anyfunc)
 (memory $0 1)
 (export "memory" (memory $0))
 (export "fib" (func $fib))
 (func $fib (; 0 ;) (param $0 i32) (result i32)
 (local $1 i32); local variables declared
 (local $2 i32)
 (local $3 i32)
 (block $label$0
  (br_if $label$0
   (i32.lt_s
    (get_local $0)
```

```
    (i32.const 3); loop skipped if n < 3
   )
  )
  (set_local $0
   (i32.add
    (get_local $0)
    (i32.const -2); compute n - 2
   )
  )
  (set_local $2
   (i32.const 1); next = 1
  )
  (set_local $3
   (i32.const 1)
  )
  (loop $label$1
   (set_local $2
    (i32.add
     (tee_local $1
      (get_local $2)
     )
     (get_local $3)
    )
   )
   (set_local $3
    (get_local $1)
   )
   (br_if $label$1
    (tee_local $0
     (i32.add
      (get_local $0)
      (i32.const -1)
     )
    )
   )
  )
 )
 (get_local $2); the final result
 )
)
```

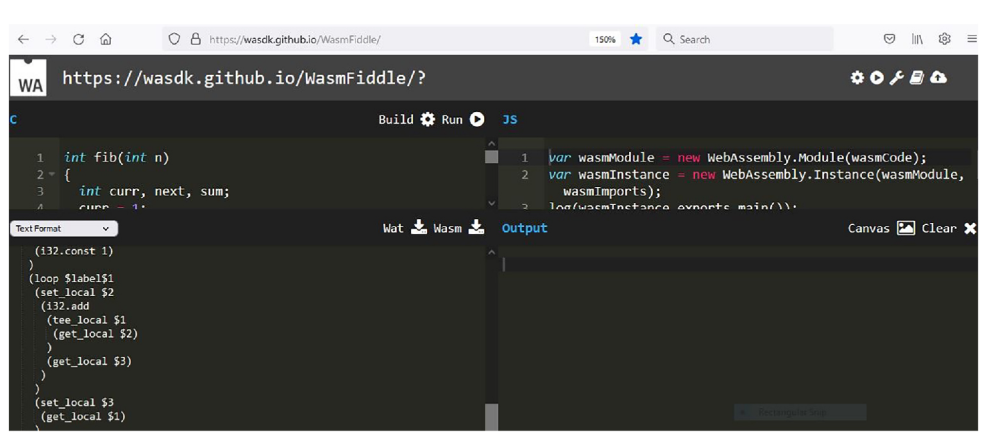
WasmFiddle also creates the Wasm binary. The WAT is the textual representation of this and is extremely useful for development and debugging. In the WAT expression, the function parameter `n` in `fib(int n)` is indicated as `(param $0 i32)`, where the variable `$0` represents `n`, and `i32` represents a 32-bit integer. The type of the return value is indicated by `(result i32)`. The three local variables are declared as `$1`, `$2`, and `$3`. WASM execution is defined in terms of a stack machine. The instruction `get_local` pushes the value of a local variable read onto the stack. The instruction `i32.add` pops the top two values

from the stack, adds them, and pushes the result back onto the stack. The instruction `set_local` pops from the stack into a local variable and `tee_local` reads from the stack into a local variable but does not pop.

WebAssembly is a low-level binary format that is compatible with common web browsers. Neither WebAssembly code nor the text-based WAT code is written by human developers but generated from code written in high-level languages, such as C, C++, Rust, and Go. The resultant code can be made to use memory very carefully, and is, generally, fast. The Wasm code is loaded and executed in the browser using JavaScript WebAssembly application programming interface (API).

The WasmFiddle interface is shown in the figure below. The C code is input by the user. The WAT and JavaScript content is generated by WasmFiddle.

Figure 42: WasmFiddle



Source: Created on behalf of IU (2022).

The following JavaScript code is generated by WasmFiddle:

wasm.js

Code

```
var wasmModule = new WebAssembly.Module(wasmCode);
var wasmInstance = new WebAssembly.Instance(wasmModule, wasmImports);
log(wasmInstance.exports.main());
```

The global WebAssembly object has two child objects `WebAssembly.Module` and `WebAssembly.Instance` that are used to interact with WebAssembly and debug. The `WebAssembly.Module` object contains WebAssembly code that has already been compiled. The `WebAssembly.Instance` object is an instance of a `WebAssembly.Module`, which contains all the exported WebAssembly functions.

Although JavaScript code can be embedded in HyperText Markup Language (HTML), there are heavy applications that are difficult to implement in JavaScript and run in the web browser. WebAssembly offers an alternative route via implementation in C, C++, Rust, or Go, and embedding the Wasm code using JavaScript WebAssembly API.

8.2 C and C++

C was originally designed for the development of the UNIX operating system (Kernighan & Ritchie, 1988) and has strongly influenced the design of many programming languages subsequently (Sebesta, 2016). For example, C introduced the **type casting** operator `(type) expression`, and braces to indicate blocks. Arrays, structures, unions, and pointers all help C to create data structures. C also supports macros and conditional compilation. Embedded software is ubiquitous in electronic devices today and much of it was written in C.

Type casting
An operator that converts a data type into another is said to be type casting.

Later, influenced by languages such as Smalltalk, C++ was created as an extension of C supporting object-orientation and features like iterators, exception handling, templates, and overloading (Tucker & Noonan, 2007). C does not support object-oriented programming, so support for polymorphism and inheritance is absent. Although some of these can be done by the generic nature of pointers, C++ provides all these and much more.

Namespaces

C++ provides a simple data-hiding principle based on **namespaces**. We aggregate related data, functions, and variables into separate namespaces. This facilitates information hiding. It also allows different identifiers with the same names to be used for different purposes. For example, we define a queue data structure in C++ and place that in a queue namespace.

Namespace
A namespace is a region in the program that defines the scope of identifiers declared inside it.

Queue.cpp

Code
```
#include <iostream>
#include "string.h"

using namespace std;
namespace Queue
{
   void enQueue(int);
   int deQueue();
}
void testQ(int n)
{
   Queue::enQueue(n);
   if(Queue::deQueue() == n) cout << "PASS\n";
   else cout << "FAIL\n";
```

```
}
namespace Queue
{
   const int maxSize=100;
   int val[maxSize];
   int num=0, front=0, rear=0;
   bool isFull=0;

   void enQueue(int n)
   {
       if(isFull) return;
       val[rear]=n;
       num++;
       rear=(rear+1)% maxSize;
       if(num==maxSize) isFull=1;
   }

   int deQueue()
   {
       if(num==0) return-1;
       int temp = val[front];
       front =(front+1)% maxSize;
       num--;
       return temp;
   }
}
int main()
{
   testQ(35);
}
```

Classes

User-defined types allow users of a programming language to extend the fundamental types and create their own customized types. C++ allows us to create our own types called classes. For any class, we can create objects of that class and create operations manipulating these objects. Although classes also support information hiding, they are different from namespaces. Classes are datatypes. We can instantiate multiple objects of these types. Namespaces cannot be instantiated as objects.

The class hierarchy, an important feature of object-oriented programming, is supported in C++. Through inheritance, it facilitates modular and hierarchical organization, which enables us to define new classes based on the existing class. The new class is called the derived class or subclass. The existing class from which the subclass is derived is known as the superclass or base class. The derived class inherits methods from the base class, and it may also add new methods or override existing base class methods.

Templates

Consider a C implementation of a function `fun` to compute the sum `a+b+c` of three integer variables `a`, `b`, and `c`.

sum.c

Code

```
#include <stdio.h>
int fun(int i, int j, int k);

int main() {
   int a = 2, b = 3, c=1;
   printf("a=%d, b=%d, c=%d\n",a,b,c);
   printf("Result=%d\n",fun(a,b,c));
   return 0;
}

int fun(int i, int j, int k)
{
  return (i+j+k);
}
```

If we now need a function to operate on arguments that are of type `double`, we need to implement a different function. The template feature of C++ offers a simple solution to this problem. The same function can operate on arguments of different types. In the example below, the function `fun` is called with variables of type `int`, `double`, and `string`. In the first two cases, it adds the arguments. For strings, the function interprets `a+b+c` as the concatenation of strings a, b, and c,

sum.cpp

Code

```
#include <iostream>
#include "string.h"
using namespace std;
template<class T> T fun(T i, T j, T k);
template<class T> T fun(T i, T j, T k)
{
  return (i+j+k);
}

int main() {
   int a = 2, b=3, c=1;
   std::cout << "a=" << a << ", b=" << b << ", c=" << c << "\n";
   std::cout << "Result = " << fun(a,b,c) << "\n";
   float d=2.3, e=2.5, f=1.1;
```

```
    std::cout << "Result = " << fun(d,e,f) << "\n";
    string r = "Apple", s = "Orange", t = "Peach";
     std::cout << "Result = " << fun(r,s,t) << "\n";
    return 0;
}
```

Exception Handling

Often, when an error occurs, the action to be taken depends on the module that invoked the function rather than the function where the error is detected. C++ allows us to define an error handling function that is invoked on detection of the error. The exception handling mechanism is a system stack unwinding mechanism that serves as an alternative return mechanism, which has used beyond exception detection and recovery. Due to the lack of such exception handling mechanisms, C programs return a zero in case of success, or non-zero in case of error, instead of returning a useful value.

8.3 Java and C#

Java was developed around 1990 in response to the requirement for an architecture-independent language for applications running in consumer electronic devices, such as microwave ovens, toasters, and remote controls (Schildt, 2017). These devices used many different CPUs as controllers, and it was expensive to create compilers for languages that were designed to be compiled for specific targets. So, efforts began to design a portable and platform-independent language that could generate code that ran on a variety of CPUs. This led to the creation of Java (Schildt, 2017). The emergence of the World Wide Web, with its associated portability issues, led to the large-scale success of Java. Today, Java is used in a wide variety of application areas.

Java is an object-oriented language, with support for efficient memory management, multithreading, and distributed computing.

The key to the success of Java is the bytecode (Schildt, 2017), which is the output of the Java compiler. It is a highly optimized set of instructions that is executed on the Java Virtual Machine (JVM), Java's runtime system and interpreter for the bytecode. The JVM needs to be implemented for different platforms, but not the Java bytecode (Schildt, 2017). However, now many Java programs are also compiled using a **just-in-time** (JIT) compiler when they start running, which compiles Java bytecodes to machine code at run time.

Just-in-time
This refers to compilation during execution rather than before.

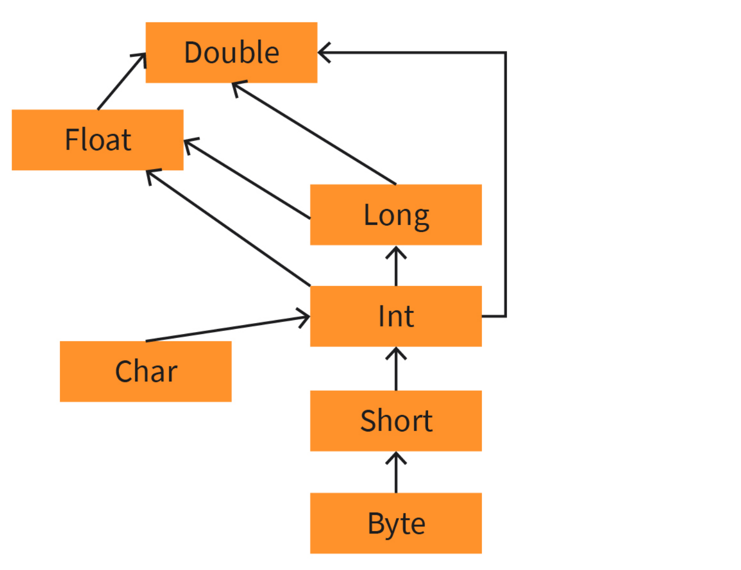
Java has both classes and primitive types. Java arrays are instances of a specific class. Instead of pointers, Java uses a reference type to point to instances of a class. One cannot write stand-alone subprograms in Java, and all subprograms need to be wrapped in classes as methods. In Java, a class can be derived from a single class only, although some benefits of **multiple inheritance** can be achieved through the usage of a feature called interface.

Java supports an elaborate system of type conversions and automatic type promotions that facilitates programming. For example, possible automatic type promotions among numeric types in Java are shown in the figure below. An arrow from type A to type B means that a variable of type A may be promoted to type B.

Multiple inheritance
This is a feature in some object-oriented languages wherein a class may be derived from multiple classes.

The Java package is a naming encapsulation construct. Public and protected variables and methods, as well as those with no access specifiers, are visible to all other classes within the same package.

Figure 43: Automatic Type Promotion in Java



Source: Created on behalf of IU (2022).

Java supports templates, or **generics**, that allow for type parameterized classes. The syntax for a generic class is `className<T>`, where T is a type variable. For generic methods in Java, generic parameters must be user-defined classes and not primitive types. We can instantiate such generic methods multiple times. However, internally, the method operates on `Object` class objects (Sebesta, 2016).

Generics
The feature of classes called generics allow a method to operate on objects of various types.

C#

The reach and the power of Java influenced the design of C#, which was created by Microsoft for its .NET framework. C# is closely related to Java, shares similar syntax and object models, and provides support for distributed computation.

The C# "assembly" is an encapsulation construct that is larger than a class. It consists of one or more files. One or more assemblies, in turn, make up a .NET application. Components of an assembly A include the following:

- its program code in the Common Intermediate Language (CIL).
- metadata describing every class defined in the assembly and external classes used
- a collection of all other assemblies referenced by A and the version number

In addition to public, private, and protected, C# has an additional access modifier called "internal", and a variant of this called "protected internal". A protected internal member is accessible to classes in the current assembly and derived classes in other assemblies. An internal member of a class is accessible only from classes in the current assembly. C# assemblies are similar to Java Archive (JAR) of Java (Sebesta, 2016).

Get Set Methods

"Get" and "set" methods give public access to private variables in a class. In the following Java program, public access to the private field `_value` of class `GetSetTest` is provided by means of the `getVal` and `setVal` methods:

GetSetTest.java

Code

```
class GetSetTest{
        private int _value;

        GetSetTest() {
            _value=0;
        }
        public int getVal()
        {
            return _value;
        }
        public void setVal(int x)
        {
            _value = x;
        }
    }
   public class Test{

    public static void main(String []args){
       GetSetTest t = new GetSetTest();
       t.setVal(25);
       System.out.println("Value=" + t.getVal());
   }
}
```

In C#, the get and set methods do not need to be explicitly invoked. Its mechanism allows us to access private variables with a syntax similar to that for public ones. An example program follows:

Customer.cs

Code

```
using System;

public class Customer
{
    private string _name;
   public string name
   {
       get
       {
           return _name;
       }
       set
       {
           _name = value ;
       }
   }
    int _age;
    public int age {
         get { return _age; }
         set { _age = value; }
    }
    public int ID
    { get; set; }

}

public class Program
{
    public static void Main()
    {
         var t = new Customer();
         t.name = "John Doe";
         Console.WriteLine(t.name);
         t.age = 25;
         Console.WriteLine(t.age);
         t.ID = 111222333;
         Console.WriteLine(t.ID);
    }
}
```

Generics in C#

C# also supports generics or template types. A method can be defined with arguments or return objects of generic type `T`. C# also supports generic collection classes, which allow for the definition of arrays, lists, stacks, queues, and dictionaries of generic type. The following is an example of using generic stacks in C#:

GenericStacks.cs

Code

```
using System;
using System.Collections.Generic;

public class Program
{
    public static void Main()
    {
         Console.WriteLine("Stack of Strings");
         Stack<string> numbers = new Stack<string>();
       numbers.Push("twenty one");
       numbers.Push("thirty two");
       numbers.Push("sixty three");

         foreach( string s in numbers )
       {
           Console.WriteLine(s);
       }

       Console.WriteLine("\nPop", numbers.Pop());
       Console.WriteLine("Top: '{0}'",numbers.Peek());
       Console.WriteLine("Pop", numbers.Pop());

         Console.WriteLine("\nStack of Integers");
         Stack<int> figures = new Stack<int>();
       figures.Push(21);
       figures.Push(32);
       figures.Push(63);

       foreach( int i in figures )
       {
           Console.WriteLine(i);
       }

       Console.WriteLine("\nPop", figures.Pop());
       Console.WriteLine("Top: {0} ",figures.Peek());
       Console.WriteLine("Pop", figures.Pop());
    }
}
```

Generics in Java

In Java, we can define our own classes, variables, or methods with arguments of generic type with parameterized declarations, as follows:

Code
```
class MyClass<T>
public T myData;
public void myMethod<T>(T myArg)
```

However, support for generics is stronger in Java with wildcard types. Wildcard types are not supported in C#.

Test.java

Code
```
import java.util.ArrayList;
import java.util.List;
import java.util.Arrays;
import java.util.Collection;
import java.util.List;
public class Test{
   public static void test(Collection<?> c){
           for (Object n: c) {
               System.out.print(n+" ");
           }
           System.out.println("");
   }
   public static void main(String []args){
       List<Integer> L1=
           Arrays.asList(1,2,3,4,5);
       test(L1);
       List<Float> L2=
           Arrays.asList(1.1f,2.1f,3.1f,4.1f,5.1f);
       test(L2);
       List<String> L3=
           Arrays.asList("a","b","c","d","e");
       test(L3);
   }
}
```

This prints

Code
```
1 2 3 4 5
1.1 2.1 3.1 4.1 5.1
a b c d e
```

The signature of the test method `test(Collection<?> c)` allows us to work with lists of type `Integer`, `Float`, and `String`. If we change this to `test(Collection<? extends Number>)`, it will work with `Integer` and `Float`, but not `String`. In general, `test(Collection<? extends X>)` will work with a subclass of `X`, and `test(Collection<? super X>)` will work with a superclass of `X`. This gives us more flexibility in working with generic types.

8.4 Haskell, Lisp

Haskell and Lisp are two well-known functional programming languages. Whereas Haskell is a purely functional, statically-typed language, Lisp is considered more flexible and is dynamically typed.

Lisp

Lisp (an abbreviation of "list processor") was designed by John McCarthy in 1960 and is regarded as the first functional programming language. It is also regarded as the first "artificial intelligence language" (Tucker & Noonan, 2007). Used primarily for symbolic data processing, Lisp has been used for solving various problems in artificial intelligence, game playing, electronic circuit design, and other areas. Today, many dialects of the original Lisp exist. However, due to portability problems, Common Lisp was created in the 1990s and it combined features of several dialects, including Scheme, while preserving the syntax, primitive functions, and basic features of pure Lisp (Sebesta, 2016).

The two basic data objects in Lisp are atoms and lists. Atoms are indivisible objects and may be either numeric or symbolic. In Lisp, integers are real numbers and are examples of numeric atoms. Symbolic atoms consist of strings with different restrictions on the characters allowed depending on the Lisp dialect being used. The list is a recursive structure consisting of an opening parenthesis "('followed by zero or more atoms or lists, and ending with a closing parenthesis')". The following are valid lists in Lisp:

```
(1 2 3 4 5)

(1 (2 3) (4 (5 6)))
```

The syntax of Lisp is characterized by uniformity and simplicity: Both data and programs take the same form, that of lists. Consider the list `(A B C)`. Interpreted as data, it consists of three atoms `A`, `B`, and `C`. Interpreted as a program, it represents a function named `A`, followed by two arguments `B` and `C` (Sebesta, 2016). Such symbolic expressions, or S-expressions, are similar to those used in the WAT format of WebAssembly. We can define anonymous functions in Lisp using lambda expressions. The term "lambda" owes its origin to **lambda calculus**. Such an expression in Lisp evaluates to function object. Let us define an anonymous function to compute the expression $f(x, y) = 2x + 3y + 2$.

Lambda calculus
This is a formal system in mathematical logic based on function abstractions and applications.

fxy.lisp

Code
```
(LAMBDA (x y) (+ (* 2 x) (* 3 y) 2))
```

To evaluate this, we simply wrap this as the first member in a list, with arguments following as second and third members:

Code
```
((LAMBDA (x y) (+ (* 2 x) (* 3 y) 2)) 3 4)
```

To print the result, wrap the above as the second member in yet another list, with the print function as the first:

Code
```
(print ((LAMBDA (x y) (+ (* 2 x) (* 3 y) 2)) 3 4))
```

This prints the answer `20`.

We can define a named function using the `defun` keyword. Consider the following Fibonacci number example:

fib.lisp

Code
```
(defun fib (n)
 (if (or (zerop n) (= n 1)) n
   (+ (fib (- n 1)) (fib (- n 2)))))
(print (fib 9))
```

Lisp is used extensively for list processing and has support for list operations. The fundamental list operations `CAR`, `CADR`, `CONS`, and `LIST` are illustrated below.

The `CAR` function returns the first element of a list. Some examples are shown below. The single quotation marks indicates that what follows is a list and not a function followed by its arguments.

list1.lisp

Code
```
(print (CAR '(A B C)))
(print (CAR '((A B) (C D))))
(print (CAR '(A (B C))))
```

These print `A`, `(A B)`, and `A` respectively.

The CDR function returns the given list with the first element removed. The CAR and CDR functions may also be combined to create a composite function. CDDR is CDR(CDR) and CADR is CAR(CDR). Most dialects of Lisp allow between two and four such compositions.

list2.lisp

Code

```
(print (CDR '(A B C)))
(print (CDR '(A (B C))))
(print (CADR '(A B C)))
(print (CDDR '(A B C)))
```

These print (B C), ((B C)), B, and (C) respectively.

The CONS and the LIST functions create lists from arguments. CONS is a function with two arguments, that creates a list, with the first argument of the function becoming the first element of the list and the second argument forming the rest of the list. LIST takes any number of arguments and returns a list, the elements of which are the function arguments.

list3.lisp

Code

```
(print (CONS 'A '(B C)))
(print (LIST 'A '(B C)))
(print (LIST 'A '(B C) 'D))
```

These print (A B C), (A (B C)), and (A (B C) D) respectively.

Haskell

Haskell is a purely functional language. Like Lisp, the fundamental data structure in Haskell is the list. Some key features in Haskell include lazy evaluation, list comprehension, and minimalist syntax.

List comprehension

Lists in Haskell can be defined by enumeration, as in [2,3,5,7,9], and using ellipses (..) as in [1,3..11]. Lists can also be defined by list comprehension, which is based on the idea of a function called a generator. We define each element of a list A as a function of the corresponding element of another list B, i.e., A[i]=f(B[i]). For instance, we may define a list as [2*x+1 | x <- [0..10]]. List comprehensions also allow us to define infinite lists, as in [2*x | x <-[0,1..]].

list1.hs

Code

```
print $ [2,3,5,7,9]
print $ [1,3..11]
print $ [2*x+1 | x <- [0..10]]
```

Lazy evaluation

Haskell has **non-strict** semantics, which increases efficiency by using lazy evaluation to avoid some computations (Sebesta, 2016). In lazy evaluation, a parameter of a function is evaluated only if its value is needed for the evaluation of the function. Lazy evaluation allows us to work with infinite lists. An example of a linear search on an ordered list follows. It works not only for finite lists, but also for infinite ones.

Non-strict
Being non-strict is a property of a language that allows a function to be evaluated, even if all actual parameters are not evaluated.

linSearch.hs

Code

```
linSearch x (m:y)
 | m < x = linSearch x y
 | m == x = True
 | otherwise = False
main = do
print $ linSearch 21 [2*x+1 | x <-[12,13,17,22,23,25]]
print $ linSearch 21 [2*x+1 | x <-[0,1..]]
print $ linSearch 22 [2*x+1 | x <-[0,1..]]
```

This prints `False, True, False`.

Minimalist syntax

Consider the simple Haskell program for computing Fibonacci numbers below.

recFib.hs

Code

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
main = do
print $ fib 10
```

The iterative solution to the same problem is as follows:

iterFib.hs

Code
```
f a b = a : f b (a + b)
fib = f 0 1
main = do
   print $ take 10 fib
```

This prints the first ten Fibonacci numbers as `[0,1,1,2,3,5,8,13,21,34]`. Note that there is no keyword to define the function in Haskell. The first line defines the function `f` with parameters `a` and `b`. It outputs `a` to the output list and then implements the pseudo-code: `sum=a+b`; `a = b`; `b = sum`. The last line iterates this ten times. The second line initializes the first two Fibonacci numbers: `0` and `1`.

Haskell programs are succinct. Below is an example of finding the factors of a number:

factors.hs

Code
```
factors n = [f | f <-[1..n], mod n f == 0]
main = do
print $ factors 60
```

This prints the factors of `60`. The first line defines factors of n as the list of numbers $f, 1 \leq f \leq n$ such that $n mod f = 0$.

Now, consider the problem of partitioning an array of integers that involves rearranging them so that integers less than or equal to a pivot appear before those greater than the pivot. This problem forms a building block of other algorithms, such as quicksort and various selection algorithms (Cormen et al., 2009). We again notice the strong declarative style in the short Haskell solution below:

part.hs

Code
```
part (i:j) = [x|x<-j, x <= i]++[i]++[x|x<-j, x > i]
main = do
print $ part [13, 9, 44, 53, 6, 5, 23, 2, 39]
```

This prints the list `[9,6,5,2,13,44,53,23,39]`. As in classical quicksort partitioning, the first element is chosen as the pivot. The first line defines a partial solution as a list containing elements less than or equal to the pivot among the other elements in the list. This is concatenated with a singleton list containing the pivot and a list containing elements greater than the pivot.

8.5 JavaScript and Its Relatives

JavaScript is a central language in web applications. It uses the browser as a platform and first appeared in the Netscape Navigator browser in 1995. It brings a dynamic functionality to websites by facilitating the usage of forms filled in by users. It helps to track all user actions including mouseovers, selecting, scrolling, clicking, and zooming. It forms an interface between the user and the webpage. JavaScript runs inside the browser and has access to the elements in the web document, local file systems, and system resources. It is fully compatible with most browsers and is widely used for client-side front-end scripting (Sebesta, 2016). Since then, JavaScript has found its way as a programming language for servers, data-science, and many other applications. The engine used is the same as that of a browser but without graphical aspects.

Basic Features

The basic syntax of JavaScript has similarities with C, with support for variables, expressions, operators, conditionals, and loops. The code can be embedded in HTML code, as shown in the following example to compute the mean of a sequence of numbers entered by the user.

mean.html

Code
```
<!DOCTYPE html>
<html>
<body>
<h2>Computing Mean</h2>
<p>Mean of a sequence of numbers.</p>
<p id="example"></p>
<script>
var n, i=0, sum = 0;
var body = document.body;
n = prompt("Enter count of numbers, range [1,50]", "");
var p1 = document.createElement('p');
if((n < 1) || (n > 50)){
  p1.appendChild(document.createTextNode("Error!"));
}
else {
  var aList = new Array(50);
  p1.appendChild(document.createTextNode("Sequence: "));
  for(i=0; i < n; i++) {
     aList[i] = prompt("Enter next number","");
     var t1 = document.createTextNode(aList[i]+" ")
     p1.appendChild(t1);
     sum+=parseInt(aList[i]);
  }
  var t2=document.createTextNode("Mean=" + sum/n);
```

```
  p1.appendChild(t2);
}

body.appendChild(p1);
</script>
</body>
</html>
```

The mean is computed for a sequence `1,2,3,4` entered by the user.

Figure 44: Computing Mean

Computing mean
Mean of a sequence of numbers.
Sequence: 1 2 3 4   Mean = 2.5

Source: Created on behalf of IU (2022).

The Document Object Model

Application programming interface
An application programming interface, or API, serves as an intermediate layer that allows applications to communicate.

The Document Object Model (DOM) is a World Wide Web Consortium (WC3) standard defining an **application programming interface** (API) for web documents to manipulate the tree of HTML elements. The HTML DOM is a standard for accessing, adding, deleting, or updating elements of an HTML document. JavaScript has been designed around the idea of the DOM with a hierarchy of parent and child objects. We illustrate this using an example, in which a hierarchy of objects is defined using the `document.createElement()` method with various table objects as arguments. The following (child, parent) relationships are defined among various objects: `(table, body)`, `(tblbody, table)`, `(row, tblbody)`, `(cell,row)`, and `(cellText, cell)`.

table.html

Code

```
<!DOCTYPE html>
<html>
<body>
<p id="example"></p>
<input type="button" value="Create a table" onclick='create_table()'>
<script>
function create_table() {
var n=0, i=0, sum = 0, num=0;
var body = document.body;
n = prompt("Enter count of distinct values, \
             in the range [1,10]", "");
  if((n < 1) || (n > 10)){
    t1=document.createTextNode(" Error, wrong value");
    body.appendChild(t1);
```

```
  }
  else {
     var A = new Array(10);
     var B = new Array(10);
        for(i=0; i < n; i++) {
     A[i] = prompt("Enter next score","");
         B[i] = prompt("Enter next frequency","");
         sum+=parseInt(A[i]*B[i]);
         num+=parseInt(B[i]);
     }

     var table1 = document.createElement("table");
    var tblBody = document.createElement("tbody");
    for (var i = 0; i < 3; i++) {
      var row = document.createElement("tr");
      for (var j = 0; j <= n; j++) {
             var cell = document.createElement("td");
             var m;
             switch(i) {
                 case 1:
                if(j==0) m="Score";
               else m=A[j-1];
            break;
        case 2:
           if(j==0) m="Frequency";
             else m=B[j-1];
           break;
        default:
           if(j==0) m="Index";
               else m=j;
         }
             var cellText =document.createTextNode(m);
             cell.appendChild(cellText);
             row.appendChild(cell);
      }
      tblBody.appendChild(row);
     }
     table1.appendChild(tblBody);
     body.appendChild(table1);
     table1.setAttribute("border", "2");
     t2=document.createTextNode("Mean=" + sum/num);
     body.appendChild(t2);
}
}
</script>
</body>
</html>
```

With a set of five scores and frequencies, the HTML page renders and displays the table as follows:

Figure 45: Displaying Scores and Frequencies

Index	1	2	3	4	5
Score	12	15	19	20	25
Frequency	5	3	5	5	4

Source: Created on behalf of IU (2022).

The Relatives

Despite the popularity of JavaScript, there are similar languages that are more suitable for specific applications and can be easily compiled into JavaScript. These include Typescript, Coffeescript, Elm, Roy, Opal, and Clojurescript (Fogus, 2013). Moreover, JavaScript is often processed by various transformation tools so that it becomes more compact, more contextualized, less readable, or better performing. Finally, JavaScript is, as of May 2022, becoming increasingly popular for server-side applications where the NodeJS environment allows the language to perform extremely well in input-output operations because of its functional aspects.

8.6 Other Imperative Programming Languages

The imperative programming paradigm, based on the von Neumann architecture, is the oldest and most developed paradigm. The imperative programming languages include features such as procedural abstractions, control structures, input/output (I/O), expressions, and assignments.

Ada

Ada was developed as part of an extensive effort in the 1970s by the US Department of Defense (Tucker & Noonan, 2007). In Ada 95, extensions for supporting object-oriented programming were added to the original, largely imperative, Ada 83, making Ada a multi-paradigm language (Tucker & Noonan, 2007). Ada was an important milestone in the development of programming languages since certain important features went on to influence the design of other programming languages (Sebesta, 2016), including the following:

- Ada includes a facility for encapsulation using packages.
- Ada’s usage in critical embedded applications influenced the development of extensive support for user-defined exception handling.

- The idea of generics was introduced allowing procedures to be defined with parameters of unspecified types. The generic procedure can be instantiated for a particular type at compile time.
- Support for concurrency is provided through the **rendezvous** mechanism for synchronization and communication (Tucker & Noonan, 2007).

Rendezvous
This is a mechanism for synchronization between a pair of tasks, allowing data to be exchanged between them and coordinated execution.

Ada 2005 added some more features, such as interfaces and greater control over scheduling algorithms. Ada is widely used in avionics, air traffic control, and rail transportation (Sebesta, 2016).

Perl

Perl found wide usage as a scripting language. It can be compiled into a machine-independent bytecode, which can then be interpreted or compiled into an executable program.

Perl is dynamically typed. Built-in data structures include dynamic arrays with integer indices and associative arrays with string indices. Support for classes was added in Version 5, allowing Perl to be used as a multi-paradigm language. Perl lacks generics, overloading, and exception handling (Tucker & Noonan, 2007), but a strength of Perl lies in its support for regular expressions; it is not surprising that Perl is best known for text processing.

PHP

With the need for dynamic, database-driven content for websites, technologies supporting such content emerged in the mid-1990s. The Hypertext Preprocesssor (PHP), developed by Rasmus Lerdorf, was originally called “Personal Home Page Tools”. It emerged as a general-purpose server-side scripting language that can be embedded in HTML (JavaScript is used for client-side scripting).

PHP is integrated with several database management systems including MySQL, Microsoft SQL Server, Oracle, Informix, and PostgreSQL. It has a simple syntax, is open-source, and is loosely typed, making it easy to use (Nixon, 2018).

SUMMARY

Among the many programming languages that exist, many stand out due to some specific features. At the same time, languages that are closely related, for example, by supporting similar programming paradigms, have certain crucial differences.

The assembly language, or assembler, is designed for specific architectures.

WebAssembly enables the execution of compiled code on the web.

C is a simple and structured imperative programming language that has been popular for embedded systems programming and has strongly influenced the design of many programming languages.

C++, barring some minor exceptions, is a superset of C. It is a multi-paradigm language and includes powerful constructs, such as namespaces, classes and inheritance, operator and function overloading, templates, and features for exception handling.

Java was developed in response to the requirement for an architecture-oblivious language for applications running in consumer electronic devices. The key to its success is the bytecode.

The design of C# was influenced by Java, with which it is closely related. C# and Java share similar syntax and object modeling, and both provide support for distributed computation.

Haskell and Lisp are functional programming languages. Whereas Lisp is considered more flexible and is dynamically typed, Haskell is a purely functional language and statically typed. Some key features in Haskell include lazy evaluation, list comprehension, and a crisp syntax.

JavaScript is a central language in web applications. It uses the browser as a platform but is used increasingly (also as a server environment). Both features bring dynamic functionality to web pages. It facilitates the usage of forms filled in by users. JavaScript has been designed around the idea of the DOM, a W3C standard. This can be used to create JavaScript applications embedded in HTML.

Other imperative programming languages include Perl, Ada, and PHP.

BACKMATTER

LIST OF REFERENCES

Aghajani, E., Nagy, C., Linares-Vásquez, M., Moreno, L., Bavota, G., Lanza, M., & Shepherd, D. C. (2020, July 6—11). Software documentation: The practitioners' perspective. *2020 IEEE/ACM 42nd international conference on software engineering (ICSE)* (pp. 590—601). IEEE.

Aghajani, E., Nagy, C., Vega-Marquez, O. L., Linares-Vasquez, M., Moreno, L., Bavota, G., & Lanza, M. (2019). Software documentation issues unveiled. *2019 IEEE/ACM 41st international conference on software engineering (ICSE)* (pp. 1199—1210). IEEE.

Ahmad, A., & Koam, A. N. A. (2020). Computing the topological descriptors of line graph of the complete m-ary trees. *Journal of Intelligent and Fuzzy Systems*, *39*(1), 1081—1088. https://doi-org.pxz.iubh.de:8443/10.3233/JIFS-191992

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). *Compilers: Principles, techniques, and tools* (2nd ed.). Addison-Wesley.

Almeida, J. B., Frade, M. J., Pinto, J. S., & Melo de Sousa, S. (2011). *Rigorous software development: An introduction to program verification*. Springer.

Batchelder, N. (2022). *Coverage.py* (Version 6.3.2) [Computer software]. https://coverage.readthedocs.io/en/6.3.2/#

Ben-Kiki, O., Evans, C. Net, I.D., (2021) YAML Ain't Markup Language. *YAML Development Team.* https://yaml.org/spec/1.2.2/

Bentley, J. (2000). *Programming pearls* (2nd ed.). Addison-Wesley.

Brodal, G. S. (2013). A survey on priority queues. In A. Brodnik, A. López-Ortiz, V. Raman, & A. Viola (Eds.), Space-efficient data structures, streams, and algorithms. *Lecture notes in Computer Science* (Vol. 8066). Springer. https://doi.org/10.1007/978-3-642-40273-9_11

Chung, C.-M. (1990, September 24—27). Software development techniques—Combining testing and metrics. *IEEE TENCON'90: 1990 IEEE region 10 conference on computer and communication systems: Conference proceedings* (Vol. 1, pp. 424—428). IEEE. https://doi-org.pxz.iubh.de:8443/10.1109/TENCON.1990.152646

Clarke, E., Biere, A., Raimi, R., & Zhu, Y. (2001). Bounded model checking using satisfiability solving. *Formal Methods in System Design*, *19*(1), 7—34. https://doi.10.1023/A:1011276507260

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms* (3rd ed.). MIT Press.

Even, G., & Medina, M. (2012). *Digital logic design: A rigorous approach*. Cambridge University Press. https://.doi.10.1017/CBO9781139226455.009

Filippova, K., & Strube, M. (2009). Tree linearization in English: Improving language model based approaches. *NAACL-short 09': Proceedings of human language technologies: The 2009 annual conference of the North American chapter of the Association for Computational Linguistics, companion volume: Short papers* (pp. 225—228). https://doi-org.pxz.iubh.de:8443/10.3115/1620853.1620915

Fogus, M. (2013). *Functional JavaScript*. O'Reilly.

Gabrielli, M., & Martini, S. (2010). *Programming languages: Principles and paradigms*. Springer.

Garcia-Tobar, J. (2017). *Sphinx as a tool for documenting technical projects*. https://doi-org.pxz.iubh.de:8443/10.5281/zenodo.439242

Goodger, D., & van Rossum, G. (2010). Docstring conventions. In M. Alchin (Ed.), *Pro Python* (pp. 303—307). Apress. https://doi-org.pxz.iubh.de:8443/10.1007/978-1-4302-2758-8_15

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). *Data structures and algorithms in Python*. Wiley.

Hennessy, J. L., & Patterson, D. A. (2017). *Computer architecture: A quantitative approach* (6th ed.). Morgan Kaufmann.

Horowitz, E., Sahni, S., & Rajasekaran, S. (2008). *Computer algorithms/C++.* Universities Press.

JSON (n.d) Introducing JSON. https://www.json.org/json-en.html

Jujuedv, & PHP Wellnitz. (n.d.). *SOSML IDE* (Version 1.6.4) [Computer software]. https://sosml.org/

Kan, S. H. (2016). *Metrics and models in software quality engineering* (2nd ed.). Pearson.

Kernighan, B. W., & Ritchie, D. M. (1988). *The C programming language* (2nd ed.). Pearson.

Knuth, D. E. (1998). *The art of computer programming: Sorting and Searching* (2nd ed., Vol. 3). Pearson.

Knuth, D. E. (2013). *The art of computer programming: Fundamental algorithms* (3rd ed., Vol. 1). Addison-Wesley.

Levitin, A. (2012). *Introduction to the design and analysis of algorithms* (3rd ed.). Pearson.

Liang, Y. D. (2017). *Introduction to programming using Python*. Pearson.

Miller, B. N., & Ranum, D. L. (2013). *Problem solving with algorithms and data structures using Python* (2nd ed.). Franklin Beedle Publishers.

Nixon, R. (2018). *Learning PHP, MySQL and JavaScript* (5th ed.). O'Reilly.

O'Regan, G. (2018). *The innovation in computing companion. A compendium of select, pivotal inventions*. Springer. https://doi.org/10.1007/978-3-030-02619-6_23

Pedregosa, F. (2021). *Memory-profiler* (Version 0.60.0) [Computer software]. https://pypi.org/project/memory-profiler/

Peled D., & Qu, H. (2003). Automatic verification of annotated code. In H. Konig, M. Heiner & A. Wolisz (Eds.), *Formal techniques for networked and distributed systems—FORTE 2003*. Lecture notes in computer science (Vol. 2767). Springer. https://doi.org/10.1007/978-3-540-39979-7_9

Pratt, T. W., & Zelkowitz, M. V. (2001). *Programming languages: Design and implementation* (4th ed.). Prentice-Hall.

Python Software Foundation. (2021a). *Radon* (Version 5.1.0) [Computer software]. https://pypi.org/project/radon/

Python Software Foundation. (2021b). *Memory Profiler* (Version 0.58.0) [Computer software). https://pypi.org/project/memory-profiler/

Qian Yang, J., Li, J., & Weiss, D. M. (2009). A survey of coverage-based testing tools. *The Computer Journal*, *52*(5), 589—597.

Read the Docs. (n.d.). *Read the Docs: documentation simplified.* https://docs.readthedocs.io/en/stable/about/index.html

Rosen, K. H. (2019). *Discrete mathematics and its applications* (8th ed.). McGraw-Hill.

Rourke, M. (2018). *Learn WebAssembly*. Packt.

Scott, M. L. (2016). *Programming language pragmatics* (4th ed.). Morgan Kaufmann.

Schildt, H. (2017). *Java: The complete reference* (10th ed.). Oracle Press.

Sebesta, R. W. (2016). *Concepts of programming languages* (11th ed.). Pearson.

Sommerville, I. (2016). *Software engineering* (10th ed.). Pearson.

Sperberg-McQueen, C. M, (2008). *Extensible Markup Language (XML) 1.0* (5th ed.) W3C. https://www.w3.org/TR/xml/

Spyder IDE. (2021). *Spyder* (Version 5.2.1) [Computer software]. https://www.spyder-ide.org/

Tucker, A. B., & Noonan, R. E. (2007). *Programming languages: Principles and paradigms* (2nd ed.). McGraw Hill.

Vonhoegen, H. (2018): *XML: Einstieg, Praxis, Referenz. Das XML-Handbuch mit vielen Anwendungsbeispielen* [XML: Introduction, praxis, reference: The XML handbook with examples] (9th ed.). Rheinwerk Computing.

von Neumann, J. (1945). *First draft of a report on the EDVAC*. University of Pennsylvania.

Watson, A. H., & McCabe, T. J. (1996). *Structured testing: A testing methodology using the cyclomatic complexity metric* (NIST Special Publication 500—235). NIST. http://www.mccabe.com/iq_research_nist.htm

Whittaker, J. A. (2009). *Exploratory software testing*. Addison-Wesley.

Wilson, R. J. (2010). *Introduction to graph theory* (5th ed.). Pearson.

LIST OF TABLES AND FIGURES

Figure 1: Euclid’s Greatest Common Divisor Algorithm . . . 22

Figure 2: Tree Representation of Butane Isomers . . . 37

Figure 3: Railroad Car Switching Using Stacks . . . 44

Figure 4: Departure Queue of Aircrafts on a Runway . . . 45

Figure 5: Aircraft Landing Problem . . . 48

Figure 6: Heap Operations . . . 49

Figure 7: A Graph . . . 50

Figure 8: The Konigsberg Bridge Problem . . . 50

Figure 9: A Directed Graph . . . 51

Figure 10: An Undirected Graph with Cycles . . . 52

Figure 11: A Graph with Directed Cycles . . . 52

Figure 12: Adjacency List of an Undirected Graph . . . 53

Figure 13: Adjacency List of a Directed Graph . . . 54

Figure 14: Adjacency Matrix of an Undirected Graph . . . 54

Figure 15: Adjacency Matrix of a Directed Graph . . . 55

Figure 16: Complexity Comparison . . . 83

Figure 17: Binary Tree . . . 89

Figure 18: Insertion Sort . . . 95

Figure 19: Bubble Sort . . . 96

Figure 20: Selection Sort . . . 97

Figure 21: Quicksort . . . 98

Figure 22: Merge Sort . . . 100

Figure 23: Spyder ID . . . 101

Figure 24: A Trie of Some English Words . . . 102

Figure 25: Patricia Trie Example . . . 104

Figure 26: Pattern Matching . . . 107

Figure 27: The Prefix Function . . . 108

Figure 28: Overview of the Core Family of Languages around XML . . . 113

Figure 29: Representation of the Structure of an XML Document . . . 116

Figure 30: Tree Diagram of the Document in the Example . . . 118

Figure 31: XML Document Accessed Through a DOM Parser . . . 119

Figure 32: Process of Creating a New Document Using XSLT . . . 124

Figure 33: SOSML IDE . . . 132

Figure 34: Control Flow Graph G_1 . . . 134

Figure 35: Control Flow Graph G_2 . . . 134

Figure 36: Test Coverage Report . . . 144

Figure 37: Using the Memory Profiler in Python: Example One . . . 149

Figure 38: Memory Usage over Time . . . 150

Figure 39: Using the Memory Profiler in Python: Example Two . . . 151

Figure 40: Tkinter Label and Button . . . 160

Figure 41: Typical Layout of a Process in Memory . . . 161

Figure 42: WasmFiddle . . . 182

Figure 43: Automatic Type Promotion in Java . . . 187

Figure 44: Computing Mean 198

Figure 45: Displaying Scores and Frequencies 200

IU Internationale Hochschule GmbH
IU International University of Applied Sciences
Juri-Gagarin-Ring 152
D-99084 Erfurt

Mailing Address
Albert-Proeller-Straße 15-19
D-86675 Buchdorf

media@iu.org
www.iu.org

Help & Contacts (FAQ)
On myCampus you can always find answers
to questions concerning your studies.