|  |
| --- |
| IUBH |
| Algorithms,Data Structures, and Programming Languages |
| DLBCSL01 |

# Course Learning Objectives

This course, **Algorithms, Data Structures, and Programming Languages,** will provide the student with a basic understanding of algorithms, data structures, and programming languages, which are the foundations of computer programming. It will provide 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 develop an understanding of basic algorithms and be able to apply them in practical situations. The student will be able to design and analyze (basic?) algorithms and will be able to apply suitable algorithms to problems arising in different applications. Additionally, the student will gain 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 apply various program measurement methodologies, as well as explain and compare various programming paradigms and languages.

# Unit 1 – Basic Concepts

**Unit Learning Objectives**

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 the basic data structures list, chain, and tree.

# 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 B.C. 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 B.C. Around 250 B.C., the Greek mathematician Archimedes proposed an algorithmic procedure for computing an approximation of π, 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 Basics of Programming

### Computer Algorithms

The word “algorithm” owes its origin to the Latin translation of the Arabic work of the 9th-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: (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 (Must be feasible)

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

### 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, Mauchly and 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**

This describes a set of rules and specifications for how software and hardware making up 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 obtains 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.
* *I/O (Input-Output) 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 and 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, comprehend, 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. Translation of programs written in high-level languages into machine code is done with *compilers* and *interpreters*. Whereas compilers translate the whole code into machine language, interpreters translate one statement at a time. Java, C, and C++ are examples of compiled languages. Python and Lisp are interpreted.

### Example Program

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

#### Gcd.py

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 are 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 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. Two common measures of efficiency are (Cormen et al., 2009):

*Space Complexity*: A measure of the amount of memory the algorithm needs.

*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 it (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* like 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:**

**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

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 elements (i.e., numList is equal to n), the while loop is executed at most 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 a 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 where is a small constant and is the size of the list being searched. There is a better search algorithm, called *binary search,* which takes time proportional to These two time complexities are customarily specified as and respectively, using the notation for *asymptotic upper bound* below*,* which guarantees that these bounds hold for all sufficiently large values of .

*Asymptotic Upper Bound*: We define there exist positive constants and such that }. The expression denotes the membership of in the set (Cormen et al., 2009).

*Asymptotic Lower Bound*: We define there exist positive constants and such that 0}. The expression denotes the membership of in the set (Cormen et al., 2009).

*Asymptotic Tight Bound*: We define there exist positive constants , , and such that . The expression denotes the membership of in the set (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 and the number of edges . 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 where and are the numbers whose GCD is being computed. There is a better algorithm for the problem due to Euclid that runs in time (Cormen et al. 2009).

### Self-Check Questions

1. Which of the following is a required property of an algorithm?

A. Clear and unambiguous set of instructions.

B. Termination in a finite number of steps.

* A but not B
* B but not A
* Neither A nor B
* *Both A and B*

2. Which of the following languages is interpreted?

* C
* C++
* *Python*
* Java

3. What is the time complexity of the algorithm for finding the GCD of two positive integers and by checking all integers between and ?



## 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 will also need to be specified. The features that are available in a given language influence the algorithm and data structure design, which, in turn, simplifies the programming effort.

### 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. Here 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 be the maximum and be the minimum of the two numbers.
* If output as the answer.
* Otherwise, divide by and let be the remainder. Now set and and return to step 2.

#### 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

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 vogue 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. Here is the flowchart version of Euclid’s GCD algorithm.

Euclid’s GCD Algorithm

![Diagram

Description automatically generated]()

### 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 is, 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 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 variously 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 this is implemented (Sebesta, 2016). A data abstraction is defined in terms of the associated defining operations.

**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.

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.

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 representation 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. If the representation is updated, 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 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.

Self-Check Questions1. Of the following, which one is the closest to a program written in a high-level language?

* *Pseudocode*
* Flowchart
* Natural language description of an algorithm
* Architectural diagram

2. Which of the following characteristics of a programming language relates to powerful features in the language which allow for more succinct programs?

* Orthogonality
* Abstraction
* *Expressivity*
* Portability

3. To use the sqrt(x) function in Python, we need not know its implementation details. This property is best described as:

* Knowledge abstraction
* Procedural approximation
* *Procedural abstraction*
* Data abstraction

## 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):

* Structures used inside statements such as those governed by rules of associativity and operator precedence.
* Structures used with groups of statements such as those associated with conditional statements and iterative loops.
* 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 since 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:

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:

c = 1

b = c

a = b

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

a, b, c = 1, 2, 3

This is equivalent to:

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:

==, !=, >=, <=, >, <

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:

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:

if(condition): #if this is true

Statement 1 #then this happens

Else: #otherwise

Statement 2 #this happens

if(x%2 == 0):

print("Even")

else:

print("Odd")

Chained conditionals in Python include multiple conditions and statements:

if(condition 1): #if this is true

Statement 1 #then this happens

Elif(condition 2): #or else if this is true

Statement 2 #then this happens

. #etc

.

elif(condition n):

Statement n

else: #if all the above are false

Statement n+1 # do this

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

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. All these 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:

>>>max(2,3)

3

>>>max(2,3,4)

4

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

4

>>>max("abc")

‘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:

def plus5(a):

return(a+2)

>>>plus5(7)#returns 12

#### Multiple arguments

Functions may have multiple arguments:

>>> max(15, 23, 12)

23

>>> max(15, 23.1,12)

23.1

#### Multiple return values

Functions may return multiple values:

#### maxmin1.py

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

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 factorial function:

The corresponding Python function is:

#### fact.py

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.

Self-Check Questions1. In Python, which of the relationships is true about a, b, and c after the following assignments?

a=2\*\*2\*\*2

b= (2\*\*2)\*\*2

c = 2\*\*(2\*\*2)

* a < b == c
* b < a == c
* c < a == b
* a == b == c

2. In Python, choose among (A) or (B) which is equivalent to the statement

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

B) if (0 < x < 100):

* A but not B
* B but not A
* *Both A and B*
* Neither A nor B

3. The order of evaluation of operators in an expression like 3\*4+5 depends on which of the following:

A. Rules for operator precedence.

B. Rules for associativity.

* B but not A
* *A but not B*
* Both A and B
* Neither A nor B

## 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 act 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 like 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 a 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. Whereas computing sum or average is fine for salaries, 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 1 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 and 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 more specific types as well, 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 you 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 facilitates explicit type conversions by the user.

### 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 easier 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.

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 a type is delayed till run time.

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: 8 bits make up a byte and multiple bytes make up a word. More significant bits and bytes 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 and Patterson, 2017). There are multiple views on words and bytes:

* Logical: This is viewed as a string of bits. There are bitwise operators which 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.
* 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, in binary is where is the mantissa and the unbiased exponent is (or in decimal). The exponent is usually stored after adding what is called a *bias*. Since the mantissa always starts with a ‘’, 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 (Hennessy and Patterson, 2017).
* Character: Here the view represents a character code like 8-bit ASCII, 16-bit Unicode, or 32-bit Unicode.

#### Big and Little Endian

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

* 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).
* 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).

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

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.

### Self-Check Questions

In which of the following scenarios is a+b a valid operation in Python?

* a *and* b *are both strings or both integers*
* a is an integer and b is a string
* a is string and b is an integer
* a is a floating-point number and b is a string

2. Which of the following describes languages in which the checking of the type system rules happens at run time?

* Statically Typed
* *Dynamically Typed*
* Delayed Typed
* Execution Typed

3. Which of the following languages is dynamically typed?

* C
* C++
* *Python*
* Java

## 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

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

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 time, where is the size of the linked list. The other operations take 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

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, trees can be used to model chemical compounds (Ahamad and 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.

Tree Representation of Butane Isomers

![A picture containing clock

Description automatically generated]()

![Chart

Description automatically generated with medium confidence]()

Trees enable us to naturally organize data in the form of file systems, 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

Whereas trees may be used for representing acyclic relationships connecting entities, 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 on the path from the root to node is called an *ancestor* of. Any node on the path from node to a leaf node is called a *descendant* of (Goodrich et al., 2013)*.*

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

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

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

### Self-Check Questions

1. How many leaves does a binary tree with nodes have?

* *At most*
* Exactly
* Always less than
* Always more than

2. What is the height of a full binary tree with nodes?

3. What is the time complexity of inserting an element at the head of a singly linked list?

Summary

We studied the basic definition of an algorithm, basic concepts of the von Neumann architecture, the idea of what a programming language is, and the complexity of algorithms. We looked at methods of specifying an algorithm including natural language, flowcharts, and pseudocode.

We studied the concept of abstraction. 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. We studied built-in and user-defined types, type systems, and static and dynamic typing. We also studied basic data structures like lists, chains, and trees with their Python implementations.

# 

# Unit 2 – Data Structures

**Unit Learning Objectives**

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. These principles are being applied in this unit which describes data structures as classes.

## 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

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 get created and get 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 HTML.

Railroad Car Switching Using Stacks

![Diagram

Description automatically generated]()

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.

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 airplane that is ready to depart enters the queue. Aircraft in the queue wait for their turn. The airplane at the front of the queue takes off on receiving permission from the air traffic control.

Departure Queue of Aircraft at a Runway

![Text, whiteboard

Description automatically generated]()

The basic supported operations of a queue are:

* 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

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, breadth-first search (BFS) in graphs, 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 like Dijkstra’s shortest path algorithm, Prim’s minimal spanning tree algorithm, and various job scheduling problems 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 like the Binomial Heap, Fibonacci Heap, and Leftist Heap, among others, (Brodal, 2013) that have their own properties.

**Heapsort**

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

#### 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:

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

#### Heap Property

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

**Complete binary tree**

A complete binary tree is a binary tree with the following properties: (a) all leaves are 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.

* is the key associated with node
* ) is the left child of
* is the right child of

Let be the root of . is a heap if:

* is NULL or
* (a)

(b) The subtrees rooted at and 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

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 time while both extractMax and insert take time, where is the number of items in the heap when the operation is performed.

Aircraft Landing Problem

Diagram

Description automatically generated

#### 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:

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 byextractMax is shown below:

Heap Operations

|  |  |
| --- | --- |
| Shape  Description automatically generated | Shape  Description automatically generated |
| Shape, arrow  Description automatically generated | Shape  Description automatically generated |
| Shape  Description automatically generated | Shape  Description automatically generated |

### Graphs

#### Introduction

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 consists of a set of vertices and a set of edges

An edge , and connects vertices and .

A Graph

![A picture containing sword, weapon

Description automatically generated]()

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

A Graph for the Königsberg Bridge Problem

![Diagram

Description automatically generated]()

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).

**Self-loops**

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

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

A Directed Graph

![A close-up of a stethoscope

Description automatically generated with medium confidence]()

#### 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 like friendship, family 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 disjoint. The graph representation also depends on what type of data we wish to mine. 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

**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.

* Airline network (connectivity between cities).

#### Cycles

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

An Undirected Graph with Cycles

![Ein Bild, das Schwert enthält.

Automatisch generierte Beschreibung]()

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.

A Graph with Directed Cycles

![A picture containing sword

Description automatically generated]()

#### DAGs

**DAG**

A DAG is a directed graph with no directed cycles.

*Directed acyclic graphs* or ***DAGs*** are useful for modeling dependencies between tasks.

#### Graph Representation

**Sparse graph**

A sparse graph is a graph with vertices and number of edges.

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

For an undirected graph the adjacency list for vertex stores the list of neighborsiff . 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 and a straight edge between and results in

Adjacency List of an Undirected Graph

![Chart, line chart

Description automatically generated with medium confidence]()

For a directed graph , the adjacency list for vertex stores the list of neighbors In some applications, may store incoming edges at

Adjacency List of a Directed Graph

![Diagram, shape

Description automatically generated]()

**Dense graph**

A dense graph is a graph with vertices and number of edges.

For any simple graph , directed or undirected, let us assume that the vertices are named as . The adjacency matrix is a two-dimensional Boolean matrix where if and only if is an edge in the graph. This is sometimes the preferred representation if the graph is **dense**.

Adjacency Matrix of an Undirected Graph

![Chart, line chart

Description automatically generated]()The adjacency matrix of a non-directed graph is symmetric. This need not be the case for a directed graph:

Adjacency Matrix of a Directed Graph

![Shape

Description automatically generated with low confidence]()  
Both adjacency list and adjacency matrix representations are widely used, however, adjacency matrices have a space requirement of , while adjacency lists have a space requirement of

### Self-Check Questions

1. For a MAX-heap with elements, how much time does an extractMax operation take in the worst case?

2. Which of the following data structure uses a LIFO paradigm?

* Queue
* Heap
* *Stack*
* Tree

3. What is the space complexity of using adjacency lists to store a graph with vertices and edges?

## 2.2 Abstract Data Types, Objects, and Classes

### ADTs

The *abstract data type*, or *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:

Vertex ADT

Vertex(name): creates a vertex with a given city name.

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 cost of the shortest path between cities named a and b.

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 type representation and the 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.

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

#### Heap ADT

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 functions implementing the supported operations.
* Some operations like \_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. 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 is 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.
* 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

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,
* the Square class’s 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.

### Self-Check Questions

1. In the context of data structures, the acronym ADT stands for

* *Abstract Data Type*
* Augmented Data Type
* Abstract Decision Tree
* Auxiliary Data Type

2. Which of the following are features of object-oriented programming?

A. Inheritance

B. Encapsulation

* A but not B
* B but not A
* Neither A nor B
* *Both A and B*

3. If we represent class A to represent all birds and class B to represent all blackbirds, which one of them will be the base class and which one will be the derived class?

* B is the base class and A is the derived class
* A is the base class but nothing can be said about B
* *A is the base class and B is the derived class*
* B is the derived class but nothing can be said about A

## 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 as well as for user-defined classes.

#### The len function

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

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:

a = 23

b = 45

print(a+b)

a = "abc"

b = "def"

print(a+b)

a = [1,2,3]

b=[4,5]

print(a+b)

a = (1,2,3)

b = (3,4,5)

print(a+b)

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).

### Self-Check Questions

1. What is the ability to use the same function for different types known as?

* Polytype
* *Polymorphism*
* Multiple inheritance
* Subclassing

2. Which of the following is correct regarding types of data the len function can be applied to in Python?

* List but not tuples
* List, tuples, dictionaries but not sets
* List and dictionaries but not ranges or strings
* *List, tuples, dictionaries, sets, strings, and ranges*

3. Suppose that in an inheritance hierarchy of classes in Python, A is a subclass of B and B is a subclass of C. A method area() is defined in C and A. If an object b of class B invokes b.area(), the area method defined in which class will be invoked?

* *A*
* B
* C
* Both A and C

Summary

In this unit, we covered important data structures like 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 deque to remove them from the front. We proposed an implementation using Python lists. Queues are useful in BFS and many scheduling algorithms.

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 time and removing the same in time. The heap also supports an addition of a new element in 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 as well as for user-defined classes.

# Unit 3 – Algorithm Design

**Unit Learning Objectives**

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 the correctness of programs,
* apply program verification and testing methodologies,
* understand formal analysis of algorithms, notations, and complexity classes.

# 3. Algorithm Design

## Introduction

To efficiently solve a problem on a computer, we need efficient algorithms and data structures that we then map to efficient programs in a programming language of our choice. 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. Whereas *break* shifts the control out of the loop, *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:

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 and the loop then terminates.

n = 1

while (n <= 25):

n += 1

print(n)

Note that if 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:

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.

**Iterable**

An iterable is an object in Python that allows iteration through a sequence of values.

The general syntax is

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 to inclusively:

#### forloops.py

for i in range(21,35):

print(i)

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

fruitBasket=["apple","banana","mango","cherry","kiwi"]

for i in fruitBasket:

print(i)

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

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:

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 like 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*. Whereas break allows the control to exit the loop, 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 starting from until it encounters an odd multiple of . It prints the sequence , , ..., .

#### break.py

for i in range(501,1000,2):

print(i)

if(i % 37==0):

break

The following Python code fragment prints all odd multiples of in the range :

#### continue.py

for i in range(501,1000,2):

if(i % 37!=0):

continue

print(i)

#### Iterators

*Iterators* are user-defined functions that iterate through 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 for iterating over elements of an iterable object.

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

#### iterator.py

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

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 :

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

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)

The 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.

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

This definition leads to a straightforward binary recursive implementation:

#### binFib.py

#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

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. Whereas the binary variant runs in exponential time, the linear version takes time, as we shall see below.

### Induction Proofs

*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 induction* and *strong induction*.

**Loop invariant**

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

#### Weak Induction

Suppose we need to prove that a property is true for all non-negative integers . We follow these steps:

* Basis: Show that is true.
* Induction Step: Show that for all , if is true, then is true.

#### Strong induction

In this case, to prove that a property  is true for all non-negative integers , we follow these steps:

* Basis: Show that is true.
* Induction Step: Show that for all , if is true for all , then is true.

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

### Self-Check Questions

1. What do we call a property that is true before and after the execution of a loop?

* *Loop invariant*
* Loop property
* Loop constraint
* Assertion

2. Which of the following languages has a while loop but not a do-while loop?

* C
* C++
* *Python*
* Java

3. What is the time complexity of the algorithm for finding Fibonacci numbers using binary recursion?

* Logarithmic
* *Exponential*
* Linear
* Quadratic

## 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 like Bioinformatics, Computer Vision, and Data Mining (Brodal, 2007), (Bentley, 2000).

**Brute-force algorithm**

This is a straightforward al-gorithm that typically adopts a simple approach like considering all possible cases.

*Example*: Consider the sequence A=[-6, -22, 1, 6, -5, 3, 4]. Here the maximum contiguous subsequence is [1, 6, -5, 3, 4] with a sum of 9.

A possible **brute-force algorithm** for this problem could simply be to compute the subrarray sum for each possible pair satisfying and keep track of the maximum. Here is a Python implementation of this algorithm, with the sequence represented as a Python list.

#### maxContiguousBF.py

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 is executed times. The loop with index is executed times. The loop with index k is executed times. So, overall, this is an 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.

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

#### maxContiguousDP.py

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 is executed times, and the loop with index is executed times. So, overall, this is an algorithm for the problem, which is an improvement over the brute-force 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

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 where is the size of the input. We subdivide the problem into two parts and recurse on each. Finding the maximum crossing subsequences takes time.

This solves to because there is amount of work involved in each of levels of recursion.

### Greedy Algorithms

Consider an optimization problem that has an associated objective function . 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 customers at a grocery store waiting to be served at a single counter. Customer requires units of time to be served. The total waiting time for customer before being served is equal to the total serving time of customers served before customer . Suppose the grocery store owner wants to minimize the total waiting time of the 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 , the optimal solution is achieved. Here, the store owner makes the greedy choice of choosing as the next customer to be served the one with the minimum (ties broken arbitrarily) among those still waiting. We illustrate this with an example.

Example: Consider a problem instance with customers.

Let .

Consider the schedule: .

Cumulative waiting times:

Sum of waiting times: .

Now consider an alternate schedule:

Cumulative waiting times:

Sum of waiting times: .

Finally consider the greedy schedule

Cumulative waiting times:

Sum of waiting times: .

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 non-decreasing order of service times gives the minimal solution.

### Self-Check Questions

1. In which algorithm design strategy do we select a locally optimal choice at each stage?

* *Greedy*
* Dynamic programming
* Brute force
* Divide-and-conquer

2. The analysis for the brute-force solution to the Maximum Contiguous Subarray shows that it takes time. When does the algorithm take that much time?

* In the best case but not always
* On average but not always
* Sometimes but not always
* *Always*

3. What is the time complexity of the divide-and-conquer algorithm for the Maximum Contiguous Subarray Problem?

## 3.3 Correctness and Verification of Algorithms

### Correctness of a Greedy Algorithm

We 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 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 . We prove below that this is correct, that is, if the customers are served in the order of non-decreasing the solution achieved is an optimal solution.

Condition:

Goal: To minimize the sum of the waiting time of all customers.

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

Proof:

We prove the claim to be correct by the method of contradiction. The claim implies that . 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 As is optimal and S is not, they must differ at one or more indices. Let be the smallest index such that .

Since , for some and for some .

Since , we get .

Let be the schedule obtained from by swapping and .

Since moves places up and moves places down in the schedule,

Since and ,

matches up to position .

matches up to position .

Thus, we can find schedules where matches up to position .

Thus .

But since is an optimal schedule, .

Hence , 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* (Cormen et al., 2009) provides a mathematical basis to show that a greedy algorithm is correct by using a combinatorial structure called a *matroid*. 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

def factorial(n):

index = 0

value = 1

while(index < n):

index += 1

value \*= index

return value

We would like to prove the following:

: If the while loop executes times, ,

*Proof*:

*Basis*: is true. By definition Before the while loop executes, .

*Induction Step*: Let be true. Assume that if the while loop executes times and . The variable index tracks the number of iterations of the while loop. Thus, at this point . Then, in the next iteration index gets incremented to k and . Hence is true.

### Correctness of a Recursive Algorithm

Example: Consider the following Python program which computes the highest power of a factor that divides a natural number .

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: .

*Proof*: By strong induction on the first function argument .

*Basis*: The function works correctly for If or , hence the function correctly returns in line . If , and , again the function returns a in line which is correct. If , the function correctly returns a value of (line 5).

Induction Step: Suppose the function works correctly for all values of satisfying . Now consider .

Case 1: If , the function will return , which is correct.

Case 1: If , the function will return , which is correct.

Case 2: If and , then divides . Then the highest power of that divides is one more than the highest power of that divides .

Since , according to the induction hypothesis, the function works correctly for and returns the highest power of that divides . Hence the function correctly returns one more than the highest power of that divides (line 7).

Hence by induction on num, the above arguments prove that the function correctly returns the highest power of *fact* that divides for all integers and all integers .

### 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 time (Bentley, 2000). Here is a Python implementation of Kadane’s algorithm, which may seem unintuitive to start with:

#### maxContiguousOPT.py

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 tracks the maximum sum for a subsequence ending at the most recently processed position, which is at the start of the loop and i at the end. The variable tracks the maximum sum for the entire processed subsequence, which is at the start of the loop and at the end. The values of and are both zero initially, which is correct by definition. If the value when added to is negative, then cannot be appended to any existing subsequence to create a maximum subsequence and no maximum subsequence ends at position . Otherwise, is updated with the value of . The last line of the loop updates with the new value of if the latter is better. Thus, the algorithm correctly maintains the meanings associated with and across loops. The formal proof can be done using mathematical induction on the loop variable .

### Program Verification

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

* *Semantic Modeling*: Given a program, what are its specifications?
* *Correct-by-Construction Development*: Given a specification, develop a program that is correct according to the specification.
* *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 study complex properties and prove expected behaviors based on theoretical deductions.
* Model Checkers. These use the program’s *state space*. 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 and 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 goals

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

* to demonstrate that a program behaves according to the requirements (validation testing), and
* 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 c*omponent 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 satisfy 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 (Sommerville, 2016).

### Self-Check Questions

1. What do we call the checking of whether the behavior of a program meets the specifications?

* *Program Verification*
* Testing
* Semantic Modeling
* Correctness proof

2. Which of the following is useful in proving the correctness of recursive algorithms?

A. Weak Induction.

B. Strong Induction.

* A but not B
* B but not A
* Neither A nor B
* *Both A and B*

3. What do we call the demonstration that a program behaves according to the requirements?

* Unit Testing
* Integration Testing
* Defect Testing
* *Validation Testing*

## 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.

In general, of interest is 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 there exist positive constants and such that }. The expression denotes the membership of in the set (Cormen et al., 2009).

#### Asymptotic Lower Bound

We define there exist positive constants and such that

0}. The expression denotes the membership of in the set (Cormen et al., 2009).

#### **Asymptotic Tight Bound**

We define there exist positive constants , , and such that . The expression denotes the membership of in the set (Cormen et al., 2009).

Note that if and only if and (Cormen et al., 2009).

Example:

Suppose the running time of an algorithm in terms of its input is given as:

Since and ,

#### Little oh and Little Omega

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

We define :constants  constant such that   
 (Cormen et al., 2009).

We define constants , constant such that

} (Cormen et al., 2009).

#### Properties Based on Limits

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

#### 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 . A case in point is the comparison between the functions and . Notice that while for , is asymptotically smaller.

Complexity Comparison

![Chart

Description automatically generated]()

#### Common Complexity Measures

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

* constant:
* logarithmic:
* log log:
* linear:
* n-logn:
* quadratic:
* cubic:
* polynomial:
* exponential:
* factorial:

### 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 , where n is the size of the input, for some constant . 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, whereas several problems are solvable within a time that is a low-degree polynomial in , 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:

* The class . All problems in this class are solvable in polynomial time.
* The class (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 these problems would be solvable in polynomial time!

The question is one of the long-standing open problems in computer science. -complete problems are important since many of them occur in real-life scenarios. The knowledge that a problem is -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).

### Self-Check Questions

1. If we compare two algorithms and with running time complexity of and respectively, how do they compare asymptotically?

* *is asymptotically faster than*
* is asymptotically slower than
* Both are of equivalent complexity
* They are incomparable

2. If an algorithm’s running time is given by what is its asymptotic complexity?

* but not
* but not )
* but not
* *, and*

3. If the complexities below are for algorithms for the same problem, which one would be the fastest?

Summary

This unit covered some well-known and fundamental algorithm design techniques including divide-and-conquer, greedy algorithms, and dynamic programming. These techniques apply to a large class of problems.

We showed how to solve the problem of finding the Maximum Contiguous Subarray sum using a simple brute-force approach, and then how to design improved algorithms using dynamic programming and divide-and-conquer methodologies.

We also illustrated the greedy technique on a scheduling problem.

For correctness proofs, we illustrated the power of 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.

|  |
| --- |
| IUBH |

# Unit 4 – Basic Algorithms

**Unit Learning Objectives**

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 21st 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 is 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, and
* String-based algorithms.

## 4.1 Traversing and Linearization of Trees

In many applications, such as natural language processing, 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**

Rooted trees are trees that 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 Although this representation can be generalized to represent any tree, we illustrate its usage on binary trees here. The root is represented by . The left subtree is a Python list and the right subtree is another Python list .

For example, consider the following binary tree:

Binary Tree

![Shape

Description automatically generated]()

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

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:

print(treeRoot(aTree))

print(leftSubTree(aTree))

print(rightSubTree(aTree))

The following is printed:

A

['B', ['D', [], []], []]

['C', ['E', ['G', [], []], []], ['F', [], []]]

### Inorder 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.

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 here:

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 in postorder recursively. 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.

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 shown below:

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.

### Self-Check Questions

1. Which of the following tree traversal algorithms visits the root node before traversals of the subtrees and uses a queue?

* *BFS*
* Postorder
* Inorder
* Preorder

2. Which of the following tree traversal algorithms visits the root node after the recursive traversals of the subtrees?

* BFS
* *Postorder*
* Inorder
* Preorder

3. Which of the following tree traversal algorithms is also known as level-order traversal?

* Inorder
* Preorder
* Postorder
* *BFS*

## 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 , the goal is to find a record whose key value is exactly. 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 , until we find an element equal to , or until we reach the end of the list. A Python implementation is as follows:

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.

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 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 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 iterations. The Python code is given below:

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 since the key is in index position . The call binarySearch(aList,28) returns since is not present.

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

This solves to

### Self-Check Questions

1. What is the worst-case complexity of sequential search?

2. What is the worst-case complexity of binary search?

3. Among binary search and linear search, which one requires the sequence to be ordered? (A) Binary Search (B) Linear Search.

* Both A and B
* *A but not B*
* B but not A
* Neither A nor B

## 4.3 Sorting Algorithms

Given a table of n elements, the sorting problem involves finding a permutation of the integers such that . 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:

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)

Insertion Sort

![Text, letter

Description automatically generated]()

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 comparisons and exchanges in the worst case, where is the size of the input. The number of comparisons can be reduced to 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

### 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 times, where 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:

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)

Bubble Sort

![Text, letter

Description automatically generated]()

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 comparisons and exchanges in the worst case, where is the size of the input.

### Selection Sort

*Selection sort* is similar to bubble sort in that the -th largest element is located in the -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).

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)

Selection Sort

![Ein Bild, das Text enthält.

Automatisch generierte Beschreibung]()

The figure shows some 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 comparisons and exchanges in the worst case, where is the size of the input.

### Quicksort

Quicksort is one of the most popular sorting algorithms. It works by choosing a pivot element and partitioning the sequence into two groups of elements: the elements and elements . 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:

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)

Quicksort

![Text, letter

Description automatically generated]()

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 is as follows:

This solves to

However, the partition may not always be balanced, and quicksort has a worst-case running time of We can get a worst-case algorithm if we could always generate equal-sized partitions, which is theoretically possible by using the **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 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).

**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.

### MergeSort

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.

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)

MergeSort

![Text

Description automatically generated]()

The figure shows steps of mergeSort applied to an example. The shaded subsequences are being merged into bigger subsequences. MergeSort creates almost balanced partitions, where the sizes of the two partitions differ by at most one. Its running time is in the worst case.

### Using Spyder IDE

The "*Scientific PYthon Development EnviRonment*" (Spyder) 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 also printed in the Python console (bottom-right pane). The editor is on the left pane.

Spyder IDE

![Graphical user interface, text, application

Description automatically generated]()

### Self-Check Questions

1. Among bubble sort, insertion sort, selection sort, and merge sort, asymptotically which is the fastest?

* Insertion sort
* Bubble sort
* Selection sort
* *Merge sort*

2. What is the asymptotic time complexity of randomized quicksort?

* worst-case
* worst-case
* *expected*
* expected

3.Which of the following two algorithms follows a divide-and-conquer paradigm?

A. Quicksort

B. MergeSort

* B but not A
* A but not B
* *Both A and B*
* Neither A nor B

## 4.4 Search in Strings

We consider the following broad problem: How can a given set of strings be stored efficiently, such that for a given query string , it can be quickly determined whether is in . 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 an important data structure 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 be an alphabet. Let be a set of strings from Σ with total length satisfying the **prefix property**. We define a trie over 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 children.
* Edges connecting a node to its child nodes are all labeled differently.
* The number of leaf nodes is exactly .
* Each leaf node is associated with a string that is the concatenation of the characters on the path from the root to
* The total number of nodes in the trie is
* The height of the trie is the same as the size of the longest string in .

An example is shown below:

A Trie of Some English Words

![Diagram

Description automatically generated]()

A Python implementation follows:

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:

{'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 (Cormen et al., 2009) and binary search trees can be used with strings just as they are used with numeric data. There is a modified trie structure called Patricia trie (Goodrich et al., 2013) that uses a simple compression idea to reduce a redundant chain of edges into a single edge. The Patricia trie takes space as opposed to the space required by the standard variant, where is the total size of all the strings. The Patricia trie for our example is shown below:

Patricia Trie Example

![Diagram

Description automatically generated]()

### Self-Check Questions

1. If we build a trie over a set of strings made up of a total of characters, how many nodes at most will the standard trie have?

2. Which of the following queries can be supported by a trie built over a set of strings ?

A. Searching with a string to check if

B. Searching with a string to check if matches prefix of any word in

* A but not B
* B but not A
* *Both A and B*
* Neither A nor B

3. If we build a standard trie and a Patricia trie with the same set of strings each of size at least , which has fewer nodes?

* Standard trie
* *Patricia trie*
* Both have an equal number of nodes
* Depends on the exact data set

## 4.5 Hash Algorithms

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 is stored in locations that are computed by simple functions using the data itself, and are typically based on one or more attributes of the values called keys (Cormen et al., 2009).

**Dictionary**

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

Hashing is the mapping of keys to locations of a one-dimensional array, which we will refer to as the hash table of size . The mapping is computed by a *hash function*. If is the set of keys, the hash function h maps to . A *collision* is said to occur if two keys map to the same location:

The basic questions we encounter when designing a hashing scheme are (Knuth, 1998):

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

The pair (hash function, 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 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 locations in the hash table numbered , a simple hash function is This is called the *division* method. This can be computed quickly, and the distribution of keys into locations is reasonable for prime.

#### Multiplication Method

**Universal Hash Functions**

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

In the *multiplication* method, we define where and is the largest integer greater than or equal to . Results indicate that a value of and works well (Knuth, 1998).

#### Universal Hashing

One potential problem with hashing is that if someone chooses all or several keys such that 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.

### 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 by the hash function to that address. We create lists . List stores all the keys that get mapped to location 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 where is the load factor and *n* is the number of elements. If we maintain , the search time is constant. A way to maintain this is to increase the table size and rehash once .

#### 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 must design a sequence that is a permutation of

To search, we follow the same probe sequence as the one 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 is the original hash function, let denote the -th location probed. The common algorithms for generating the probe sequence are (Cormen et al., 2009):

* Linear Probing. Here for . This leads to *primary clustering* where several adjacent locations can be filled up.
* Quadratic Probing. Here for . If ), then for all This leads to *secondary clustering,* which is relatively less severe than primary clustering (Cormen et al., 2009).
* Double Hashing. Let mod where and are auxiliary hash functions and must be relatively prime to 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 is prime or a power of two.

### Self-Check Questions

1. In hashing with open addressing, if we use a probe sequence for , which scheme are we using?

* *Linear probing*
* Quadratic probing
* Double hashing
* Universal hashing

2. If elements are to be stored in a hash table with locations, what is the load factor ?

3. Which hashing scheme chooses a hash function randomly from a family of hash functions?

* Linear probing
* Quadratic probing
* Double hashing
* *Universal hashing*

## 4.6 Pattern Recognition

In the classical pattern matching problem, we have an alphabet and we are given a pattern and a text where both and are strings over We wish to find all occurrences of in (Cormen et al., 2009). Other variants of the problem include finding either the first or any occurrence (Goodrich et al., 2013). We denote to be a *valid shift* if occurs in with a shift , starting at position . Otherwise, the shift is deemed to be *invalid* (Cormen et al., 2009). One occurrence begins within another one. In the example shown below, occurs at , , and .

Pattern Matching

![Graphical user interface

Description automatically generated with low confidence]()

### Naïve Pattern Matching

*Naïve pattern matching* is a simple brute-force algorithm that tries every possible value of the shift and checks whether it is a valid shift. There are possible choices for. Here is a Python implementation:

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','cbabababaa')

naiveMatch('abc','cbabababaa')

With the two nested loops, the running time is . 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) by Knuth, Morris, and Pratt (Cormen et al., 2009) corrects the problem associated with the naïve algorithm. If a prefix of of size has matched with , followed by a mismatch, we try to determine the longest suffix of the matched part that is also a prefix of . 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. An example follows:

The Prefix Function

![Chart

Description automatically generated with low confidence]()

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 and . 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 shown below:

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.

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','cbabababaa')

The running time for the KMP algorithm is (Cormen et al., 2009).

### Self-Check Questions

1. What is the complexity of the naïve algorithm for pattern matching in terms of and , the sizes of the pattern and the text respectively?

* O(n)
* O(m)
* O(m+n)
* *O((n-m+1)m)*

2. What is the running time of the KMP algorithm in terms of and , the sizes of the pattern and the text respectively?

* O(m)
* O(n)
* O(mn)
* *O(m+n)*

3. Among the naïve and the KMP algorithms for pattern matching, which algorithm computes a prefix table?

A. KMP algorithm.

B. Naïve algorithm.

* *A only*
* B only
* Both A and B
* Neither A nor B

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.

In this unit, we also covered 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 – Measuring Programs

**Unit Learning Objectives**

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

* apply type inference mechanisms
* understand tools to generate documentation and apply knowledge of the best practices in documentation sharing
* demonstrate an awareness of compiler optimization techniques and difficulties
* compare several tools for code coverage analysis
* understand the principles of unit and integration testing and apply a selection of them
* apply heap analysis tools for the discovery of bugs.

# 5. 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. 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 that 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.

## 5.1 Type Inference and IDE Interactive Support

The types and **type system** are important characteristics of a programming language and vary from language to language. 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).

**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.

### Difficulties in Python

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

from random import randint

def typeCheck(num):

if(num%2):

a = 123

else:

a = "123"

print(type(x))

typeCheck(randint(1,1000))

What is the type of ain line #6? Here, if numis 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.

### 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 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. SOSML is the online 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.

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));

SOSML IDE

![Ein Bild, das Text enthält.

Automatisch generierte Beschreibung]()

![Table

Description automatically generated]()

If none of the types are specified, all the types default to integer, and the program reports an error when invoked with real parameters.

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 is too conservative and comes at a price. Consider the following Python code fragment:

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. Thus, static type checking turns out to be more conservative.

### Self-Check Questions

1. How can the automatic type derivation of expressions in a program at compilation time be classified?

* Type system
* *Type inference*
* Type reasoning
* Type compilation

2. Why is type inference difficult in Python?

* Python is weakly typed
* Python is strongly typed
* *Python is dynamically typed*
* Python is statically typed

3. In S ML, in which of the following cases is the result of calling *print(sum(2.1, 1.2))* different?

* fun sum(a, b:real):real = a + b
* fun sum(a, b:real) = a + b
* fun sum(a, b):real = a + b
* *fun sum(a, b) = a + b*

## 5.2 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 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 of a control flow graph with edges, vertices, and 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.

.

If the number of components ,

This also represents the minimum number of paths whose linear combination can generate all possible paths in the graph. Complexity is a major cause of software errors. Studies have shown that cyclomatic complexity has a high correlation with errors in software (Watson & McCabe, 1996).

Consider the following Python function:

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 below:

Control Flow Graph G1

![Diagram, radar chart

Description automatically generated with medium confidence]()

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

else:

maxNum = num2

Control Flow Graph G2

![Diagram

Description automatically generated with medium confidence]()

For , the cyclomatic complexity .   
For , .

#### Simplified Calculations

A straight-line control flow graph with one start and one exit node has a complexity of . If we add binary decision predicates, they add 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 where is the number of binary decision predicates.

For a **planar graph**, Euclid’s formula gives us the number of edges , the number of vertices and the number of regions r in the planar embedding of the graph (Rosen, 2019): . Since , . 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:

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 API.

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):

[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, 2021).

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:

where is the Heavyside step function. The rank in turn is converted to a letter grade with A for a and F for , with A indicative of a simple block and F 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.

A definition of a variable occurs in a statement whenever there is an assignment of a value to . A definition clear path to is a path in which is not reassigned. The definition of a variable reaches the top of a block of code if and only if there is a definition clear path from the definition of to the top of block . Analogously, we can describe the notion of the definition of reaching the bottom of . The definition of is live at the top of if the definition reaches the top of and it is referenced later. Variable is live at the top or bottom of a block if there is a live definition of . 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.

### Self-Check Questions

1. In Cyclomatic Complexity implementation in Radon, what is the rank corresponding to a score of 23?

* A
* B
* C
* *D*

2. What is the cyclomatic complexity of a program whose control flow graph has 8 edges and 6 vertices and 1 component?

* 3
* *4*
* 6
* 8

3. Cyclomatic complexity is an example of which type of software metric?

* *Product metric*
* Control metric
* Process metric
* Data referencing metric

## 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 also. It can generate an 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-Tober, 2017), which is the basis of the readthedocs.io 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

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.

**RESTful API**

A RESTful API is an API conforming to the REST software architectural style, which defines a set of rules for creating web services.

#### 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

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 the 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\_\_).

**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.

Here is an example using help(obj):

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 Python 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):

* What the documentation contains. These include the correctness, completeness, and currentness of the documentation.
* 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 is useful to the readers or from the point of view of expectations set on the code.
* The documentation generation tool and documentation processes.

### Self-Check Questions

1. In Python, documentation of a function f entered with docstrings can be retrieved using which of the following?

A. repr(f.\_\_doc\_\_)

B. help(f)

* A but not B
* B but not A
* Neither A nor B
* *Both A and B*

2. Which of the following is correct regarding the usage of the tool *Doxygen*?

* It is used for code profiling
* It is used for documentation generation for C++ code only
* It is used for documentation generation for Python code only
* *It is used for documentation generation for code written in various languages*

3. Which of the following is a built-in attribute for accessing documentation associated with an object in Python?

* \_\_document\_\_
* \_\_docstring\_\_
* *\_\_doc\_\_*
* \_\_pydoc\_\_

## 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

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).

**Peephole optimization**

A peephole optimization is a code optimization technique applied on a small amount of code appearing in a sliding window.

#### **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:

pi=3.14

for index in range(0,20):

alist.append(2\*pi\*index)

print(alist)

Here the computation 2\*pi is being done once for each iteration of the loop. This can be made more efficient as follows:

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 Python code snippet:

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.

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.

### Self-Check Questions

1. What is the goal of compiler optimization?

A. Generate code that takes less time.

B. Generate code that takes less space.

* Neither A nor B
* *Both A and B*
* B but not A
* A but not B

2. What do we call the transformation of a for loop executing 5 times to a block of code wherein the code within the loop is written 5 times?

* *Loop unrolling*
* Loop hoisting
* Loop duplication
* Loop elimination

3. In compiler optimization, what does *elimination of redundant store* involve?

* A variable that is used but never initialized
* *A variable that is assigned but never used*
* A variable that is assigned and then used
* A variable that is neither assigned nor used

## 5.5 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. Usually, 70% to 80% 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, a module, a library, or a 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, 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.

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 as follows:

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% and the remaining 43% 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%. Finally adding the third test case powerOfFactors(10,3), gives us a 100% test coverage.

Test Coverage Report

![Graphical user interface, text, application

Description automatically generated]()

### Self-Check Questions

1. Which of the following is a possible code coverage metric?

A. Lines of code covered.

B. Number of statements covered.

* Neither A nor B
* *Both A and B*
* A but not B
* B but not A

2. Which of the following is a popular tool for measuring code coverage in Python?

* Codecov.py
* *Coverage.py*
* CodeCover.py
* PITest.py

3. In Coverage.py, what command generates an HTML code coverage report?

* *Coverage html*
* Coverage report
* Coverage html report
* Coverage run html

## 5.6 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 testing and integration testing.

### Unit Testing

Unit testing includes testing individual functions, classes, and methods with different parameters. 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 should also discover bugs.

Some accepted best practices for creating test cases are (Whittaker, 2009):

* 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. We present one example based on UnitTest here.

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. During one such promotion, the supermarket decides to award points to all customers in the age group of whose point balance is currently nil. 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.

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 customers. We add points to the balance of the customer with ID .

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 age group that is the target for the current promotion, but the second one already has a non-zero balance and hence will not be getting the offer. The fourth customer is not in the target group for the offer. 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.

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:

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.

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, already 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 via component interfaces. Assuming unit testing on individual objects has been done, our focus is then on testing the interfaces and the components as a group to determine if they work together as required (Sommerville, 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.

### Self-Check Questions

1. Which of the following is a unit test framework in Python?

* jUnit
* *UnitTest*
* SnapTest
* Testthat

2. Testing individual functions comes under which type of testing?

* Integration Testing
* Component Testing
* *Unit Testing*
* System Testing

3. Which of the following is the correct way to run the UnitTests in Python?

* *python -m unittest*
* python unittest -r
* unittest -m python
* python unittest -m

## 5.7 Heap Analysis

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.

**Heap**

A heap is a block of storage wherein portions get allocated and freed dynamically.

### 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

With the decorator @profile, the functions being profiled can be marked. Here is an example usage:

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 shown below:

Using the Memory Profiler in Python

![A picture containing table

Description automatically generated]()

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:

mprof run filename.py

mprof plot

The resultant plot for the above program is shown below:

Memory Usage Over Time

![Chart, line chart

Description automatically generated]()

Consider another profiled program:

@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

Another Example

![A picture containing calendar

Description automatically generated]()

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.

### Self-Check Questions

1. Which of the following is a memory profiling tool for Python?

* *Memory Profiler*
* Memory Usage
* DotMemory
* Memstat

2. What is the decorator used for marking a Python function to be profiled by Memory Profiler?

* @memory
* *@profile*
* @mprof
* @memory\_profile

3. What is the command to plot memory usage over time?

A. mprof run filename.py

B. mprof plot

* *A followed by B*
* B followed by A
* Only A
* Only B

Summary

In this unit, we began by covering type inferencing, which 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.

Also in this unit, we covered a software metric called cyclomatic complexity. This measures the testability of a program by counting the number of linearly independent paths in the control flow graph of the program. An example of computing cyclomatic complexity by using a Python tool called Radon was also presented.

This unit contains an overview of various tools to generate software documentation, including an example usage for Python using Pydoc. Various compiler optimization techniques were also discussed. This unit also includes a study of code coverage metrics; an example usage of the Coverage.Py tool that generates code coverage measurements in Python was presented. An overview of unit and integration testing methodologies in the software development process is included. Usage of the UnitTest testing framework was illustrated using example Python code. Finally, heap analysis or memory profiling was discussed, and example usage of the Memory Profiler tool was shown. This tool is used to monitor memory consumption of a process and generates analysis line-by-line as well as over time.

# Unit 6 – Programming Languages

**Unit Learning Objectives**

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.

# 6. 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.

## 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. 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. 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). Here is a Python implementation of this algorithm using a simple imperative style:

#### partition1.py

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)

We can put this code inside a function, with the list and pivot as parameters:

#### partition2.py

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’s 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, 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().

#### partition3.py

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:

lambda a : a + 13

We apply this to a parameter 11 to get a result 24:

(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:

list(filter(lambda x: x<= p,L))

This is followed by another lambda function to filter out elements x > pivot:

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:

#### partition4.py

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:

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 data structures such 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

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 GUI (Graphical User Interface) 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

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”.

Tkinter Label and Button

Graphical user interface, application

Description automatically generated

### Self-Check Questions

1. Which style of programming uses the lambda construct?

* Object-Oriented
* Procedural
* Logical
* *Functional*

2. Which paradigm of programming is supported by the kanren package of Python?

* *Logic Programming*
* Functional Programming
* Imperative Programming
* Procedural Programming

3. Which is the oldest paradigm of programming?

* Logic Programming
* Functional Programming
* *Imperative Programming*
* Procedural Programming

## 6.2 Execution of Programs

The operating system controls and monitors various system-level activities. It also allocates shared resources to programs, including CPU, RAM, disks, and 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.

**Process**

A process is a program in execution.

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.

Typical Layout of a Process in Memory



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, and
* Multiprocessing, where 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 various states:

* New: the process is being created,
* Ready: the process is ready to be scheduled,
* Running: the process is executing, and
* 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. 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: the CPU is allocated to the process that requests it first,
* Shortest Job First: the job 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, and
* Priority Scheduling: each process has a priority and the process with the highest priority is selected to be scheduled.

### Self-Check Questions

1. In which CPU scheduling algorithm is the CPU allocated to the process that requests it first?

* First Request
* *First-Come-First-Served*
* Round-Robin
* Early Request First

2. Which of these relates to multiple programs working on the same processor?

* Multithreading
* Multiprogramming
* *Multitasking*
* Multiprocessing

3. Which module(s) of the operating system select(s) the next job to be scheduled?

A. CPU scheduler

B. Dispatcher.

* Both A and B
* *A but not B*
* B but not A
* Neither A nor B

## 6.3 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 forming the heart of it. 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 form:

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 this Python code fragment:

#### alias.py

a = [2,3,4,5]

b = a

b[0]=-1

print(a)

print(b)

This prints [-1, 3, 4, 5] twice since b is just an alias for a.

Now consider a modified version:

#### copy.py

a = [2,3,4,5]

b = a[:]

b[0]=-1

print(a)

print(b)

This 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 like a = b = c.

### Object-Oriented Languages

In general, an abstraction is a representation of an entity that includes a subset of significant attributes. A data structure has an associated abstract data type that specifies what data is stored within the data structure and what operations are supported 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 is represented is determined by the class. The actual data is 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 clas*s 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 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 like 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 is through the 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.

### Self-Check Questions

1. What type of language is C++?

* Imperative
* Logical
* Functional
* *Object-Oriented*

2. Which of the following is primarily a functional programming language?

* *Haskell*
* C++
* Python
* Ada

3. Which of the following are logic programming languages?

A. Prolog.

B. Datalog.

* B but not A
* A but not B
* *Both A and B*
* Neither A nor B

## 6.4 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* (Sebesta, 2016), and the meaning of these syntactical units is called the *semantics* (Sebesta, 2016). *Pragmatics* refers to what the statements achieve in practice (Sebesta, 2016).

### An Example

Consider the syntax of the while loop in Python:

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

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 to . When the condition fails and control leaves the while loop.

Now consider another variation of the same loop:

#### while2.py

n=13

while n > 20:

n+=1

print(n)

The semantics of the while loop construct has not changed. However, this loop will not be executed since the Boolean expression will always be false.

A third variation:

#### while3.py

n=21

**Infinite loop**

An infinite loop is one that does not terminate.

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**.

This example illustrates a simple while loop construct whose syntax is well-defined and whose semantics is well understood. The pragmatics indicate different behavior under different conditions.

### Short-Circuit Evaluation

Consider the Python code fragment :

#### short1.py

x = 20

y = 0

x > 20 and x/y < 5

The condition evaluates to False. Although the division by 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

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. But 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

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 is defined over an alphabet set . The set of all strings that one can form using characters from is denoted as . The language is a subset of . The syntax rules of the language specify which strings are in . 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, or 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 CFGs or *context-free grammars* (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 YACC (Yet Another Compiler Compiler), 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

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 .

#### if\_then\_else2.py

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 . 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 the time of compilation.

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).

### Self-Check Questions

1. Context-free grammars are used for describing which of the following?

A. Syntax B. Semantics C. Pragmatics.

* *A*
* B
* C
* B and C

2. Attribute grammars are related to which of the following?

* Pragmatic analysis
* Syntactic Analysis
* *Semantic Analysis*
* Lexical Analysis

3. An error detected by a compiler during program compilation is which type of error?

* *Syntactic error*
* Both syntactic and semantic error
* Resource overflow error
* Neither syntactic nor semantic error

## 6.5 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 to the sequence of bits as either a string, integer, or a 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/C++, a block of statements enclosed within braces ( “{“ and “}” ) defines a new scope (Tucker & Noonan, 2007). Blocks may be nested in C/C++, but not functions. Scoping rules in Python, however, are based on functions, so nested functions are possible in Python.

#### Local and Global Scoping

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 as global (Sebesta, 2016).

**Local variable**

A variable that is accessible only in a specific part of a program is called a local variable.

#### global1.py

x=1

def A():

x=2

print("A", x)

A()

print("Global", x)

Here the two print statements print 2 and 1 respectively since 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.

#### global2.py

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.

#### global3.py

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 dir() and vars() commands. Here vars() returns the whole dictionary, whereas dir() returns the keys.

#### namespace1.py

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 checking tests **type compatibility** in various situations:

**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.

* 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. Some 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:

* 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.

### Self-Check Questions

1. In Python int(5.6) would evaluate to an integer. Which type of conversion is this? A. Explicit. B. Implicit.

* *A but not B*
* B but not A
* Both A and B
* Neither A nor B

2. In a programming language, what do we call the logical system defined with a set of rules to assign types to entities like variables?

* *Type system*
* Type logic
* Programming logic
* Rules system

3. If the scope of a variable is limited to a function, what is it called?

* Global variable
* *Local variable*
* Function variable
* Limited variable

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 using a paradigm-based classification. 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.

This unit also covers scoping rules for variables in programs and 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 7 – Overview of Important Programming Languages

**Unit Learning Objectives**

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 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.

# 7. 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 which support another paradigm also. 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. Whereas Haskell is a purely functional, statically typed language with lazy evaluation, list comprehension, and minimalist syntax, 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 was also an important milestone in the development of programming languages since 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.

## 7.1 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 mnemonicfor 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 |
| 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 text format consists of the syntax of the program as symbolic expressions or S-expressions. As an example, consider a C-function computing Fibonacci numbers:

#### fib.c

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;

}

We use WasmFiddle tool (Rourke, 2018) to generate the WAT code:

#### fib.wat

(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 -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 like 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 API.

The WasmFiddle interface is shown in the figure below. The C code is input by the user. The WAT and JavaScript contents are generated by WasmFiddle.

WasmFiddle

![Graphical user interface, text

Description automatically generated]()

The following JavaScript code is generated by WasmFiddle:

#### wasm.js

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 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.

### Self-Check Questions

1. What is the textual representation format of WebAssembly called?

* WebText
* *WAT*
* WASM
* Assembly Text

2. Which tool can generate WAT code from a C program?

* *WasmFiddle*
* WebAssembly Text
* WASM
* WAT generator

3. In the context of WebAssembly, which of the following is in a binary executable format? A. WASM B. WAT

* *A but not B*
* B but not A
* Both A and B
* Neither A nor B

## 7.2 C and C++

C was originally designed for the development of the UNIX operating system (Kernighan & Ritchie, 2015) 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 such languages 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

**Namespace**

A namespace is a region in the program that defines the scope of identifiers declared inside it.

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.

#### Queue.cpp

#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 customized types. C++ allows us to create 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. 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. The derived class inherits methods from the base class. 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

#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

#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 uses 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.

### Self-Check Questions

1. Which feature in C++ facilitates the use of the same code for objects of different types?

* Overloading
* *Templates*
* Classes
* Inheritance

2. Which of the following is a feature of C++ but not C?

* Structures
* *Namespaces*
* Type Casting
* While loops

3. Which of the following programs support classes?

* *C++ but not C*
* C but not C++
* Neither C nor C++
* Both C and C++

## 7.3 Java and C#

### Java

Java was developed around 1990 in response to the requirement for an architecture-independent language for applications running in consumer electronic devices like 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*.

**Multiple Inheritance**

This is a feature in some object-oriented languages wherein a class may be derived from multiple classes.

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.

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.

Automatic Type Promotion in Java![Diagram

Description automatically generated]()

**Generics**

These are a feature of classes that allows a method to operate on objects of various types.

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).

### 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:

* Its program code in CIL (the Common Intermediate Language),
* Metadata describing every class defined in the assembly and external classes used, and
* A collection of all other assemblies referenced by A, and the version number.

Besides 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 (Sebesta, 2016). C# assemblies are similar to JAR (Java Archive) 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 utilizing the getVal and setVal methods:

#### GetSetTest.java

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

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. Here is an example of using generic stacks in C#:

#### GenericStacks.cs

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:

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

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:

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.

### Self-Check Questions

1. Which of the following supports user-defined overloaded operators?

* Java only
* *C# only*
* Neither C# nor Java
* Both C# and Java

2. Which encapsulation construct of C# is larger than a class?

* *Assemblies*
* Generics
* Parameterized Types
* Packages

3. Which of C# and Java has packages?

* *Java only*
* C# only
* Neither C# nor Java
* Both C# and Java

## 7.4 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 (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 “AI 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*. Integers are real numbers and are examples of numeric atoms. Symbolic atoms consist of strings with different restrictions on allowed characters 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 the ones 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 .

**Lambda Calculus**

This is a formal system in mathematical logic based on function abstractions and applications.

#### fxy.lisp

(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:

((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:

(print ((LAMBDA (x y) (+ (\* 2 x) (\* 3 y) 2)) 3 4))

This prints the answer as 20.

We can define a named function using the defun keyword. Consider the Fibonacci number example:

#### fib.lisp

(defun fib (n)

(if (or (zerop n) (= n 1)) n

(+ (fib (- n 1)) (fib (- n 2)))))

(print (fib 9))

Lisp is extensively used for list processing and has support for list operations. The fundamental list operations CAR, CADR, CONS, and LIST are illustrated here.

The CAR function returns the first element of a list. Some examples are shown below. The single quote indicates that what follows is a list and not a function followed by its arguments.

#### list1.lisp

(print (CAR '(A B C)))

(print (CAR '((A B) (C D))))

(print (CAR '(A (B C))))

These print A, (B C), and A respectively.

The CDR function returns the given list with the first element removed. The CAR and CDR functions may also be composed. CAAR is CAR(CAR) and CADR is CAR(CDR). Most dialects of Lisp allow between two and four such compositions.

#### list2.lisp

(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 whose elements are the function arguments.

#### list3.lisp

(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]. List comprehension 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

print $ [2,3,5,7,9]

print $ [1,3..11]

print $ [2\*x+1 | x <- [0..10]]

#### Lazy Evaluation

**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.

Haskell has **non-strict** semantics, which makes it more efficient 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. Here is an example of a linear search on an ordered list. It works not only for finite lists but also for infinite ones:

#### linSearch.hs

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 a simple Haskell program for computing Fibonacci numbers.

#### recFib.hs

fib 0 = 0

fib 1 = 1

fib n = fib (n-1) + fib (n-2)

main = do

print $ fib 10

Here is the iterative solution to the same problem.

#### iterFib.hs

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 pseudocode: 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. Here is an example of finding the factors of a number:

#### factors.hs

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 as the list of numbers such that .

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 the ones 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:

#### part.hs

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.

### Self-Check Questions

1. What is the reserved word for function definition in Haskell?

* Function
* Defun
* Define
* *There is no such reserved word*

2. What is the list concatenation operation in Haskell?

* *++*
* +
* +=
* CONS

3. Which of the following is an atom in Lisp?

* (6)
* (
* )
* *6*

## 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).

### 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

<!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:

Computing Mean

![A picture containing text

Description automatically generated]()

**API**

An Application Programming Interface, or API, serves as an intermediate layer that allows applications to communicate.

### The Document Object Model

The Document Object Model (DOM) is a W3C (World Wide Web Consortium) standard defining an **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

<!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:

Displaying Scores and Frequencies

![A picture containing table

Description automatically generated]()

### 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, at the time of writing this text, in increasing fashion for server-side applications where the NodeJS environment allows the language to perform extremely well in input-output operations because of its functional aspects.

### Self-Check Questions

1. Which of the following is a keyword for declaring variables in JavaScript?

* *var*
* int
* float
* char

2. In JavaScript, which of the following declares a variable and initializes it to ?

* *var a = 1;*
* int a = 1;
* integer a = 1;
* int a; a = 1;

3. What is the HTML tag used to define JavaScript code?

* *<script>*
* <javascript>
* <javaScript>
* <java script>

## 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, 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):

* 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.

**Rendezvous**

This is a mechanism for synchronization between a pair of tasks, allowing data exchange between them and coordinated execution.

* Support for concurrency is provided through the **rendezvous** mechanism for synchronization and communication (Tucker & Noonan, 2007).

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 , 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. PHP (*PHP: Hypertext Preprocesssor*), 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).

### Self-Check Questions

1. Which of the following is an imperative programming language?

* *Perl*
* Prolog
* Datalog
* Haskell

2. What programming paradigm does Ada primarily follow?

* *Imperative*
* Logical
* Functional
* Declarative

3. Which of the following languages support concurrency using the rendezvous mechanism?

* C
* C++
* Python
* *Ada*

Summary

Among the many programming languages that exist, many stand out for 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 like 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 whom 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 more and more 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.

The unit concludes with an overview of Perl, Ada, and PHP.

# Appendix 1 – 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 & 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.

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. In *IEEE TENCON’90: 1990 IEEE Region 10 Conference on Computer and Communication Systems. Conference Proceedings* (pp. 424—428, Vol. 1). 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. In *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, *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.

Kan, S. H. (2016). *Metrics and models in software quality engineering* (2nd ed.). Pearson.

Kernighan, B. W., & Ritchie, D. M. (2015). *The C programming language*. (2nd ed.). Pearson.

Knuth, D.E. (2013). *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. (3rd ed.). Addison-Wesley.

Knuth, D. E. (1998). *The art of computer programming. Sorting and Searching* (2nd ed.,Vol. 3).Pearson.

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>

Pawlik, A., Segal, J., Sharp, H., & Petre, M. (2015). Crowdsourcing scientific software documentation: A case study of the NumPy documentation project. *Computing in Science & Engineering*, *17*(1), 28—36.

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 5.1.0.* <https://pypi.org/project/radon/>

Python Software Foundation. (2021b). *Memory profiler 0.58.0*. <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.

Rodhe, I., & Karresand, M. (2015). *Overview of formal methods in software engineering* (Technical report FOI-R--4156—SE). Swedish Defence Research Agency (FOI).

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.

Tucker, A. B., & Noonan, R. E. (2007). *Programming languages: Principles and paradigms* (2nd ed.). McGraw Hill.

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.

# Appendix 2 – List of Tables and Figures

**Euclid’s GCD Algorithm**

Source: Author

--------------------------------------------------------------------------------------

**Tree Representation of Butane Isomers**

Source: Author

--------------------------------------------------------------------------------------

**Railroad Car Switching Using Stacks**

Source: Author

--------------------------------------------------------------------------------------

**Departure Queue of Aircrafts on a Runway**

Source: Author

--------------------------------------------------------------------------------------

**Aircraft Landing Problem**

Source: Author

--------------------------------------------------------------------------------------

**Heap Operations**

Source: Author

--------------------------------------------------------------------------------------

**A Graph**

Source: Author

--------------------------------------------------------------------------------------

**A Graph for the Königsberg Bridge Problem**

Source: Author

--------------------------------------------------------------------------------------

**A Directed Graph**

Source: Author

--------------------------------------------------------------------------------------

**An Undirected Graph with Cycles**

Source: Author

--------------------------------------------------------------------------------------

**A Graph with Directed Cycles**

Source: Author

--------------------------------------------------------------------------------------

**Adjacency List of an Undirected Graph**

Source: Author

--------------------------------------------------------------------------------------

**Adjacency List of a Directed Graph**

Source: Author

--------------------------------------------------------------------------------------

**Adjacency Matrix of an Undirected Graph**

Source: Author

--------------------------------------------------------------------------------------

**Adjacency Matrix of a Directed Graph**

Source: Author

-------------------------------------------------------------------------------------

**Complexity Comparison**

Source: Author

-------------------------------------------------------------------------------------

**Binary Tree**

Source: Author

--------------------------------------------------------------------------------------

**Insertion Sort**

Source: Author

--------------------------------------------------------------------------------------

**Bubble Sort**

Source: Author

--------------------------------------------------------------------------------------

**Selection Sort**

Source: Author

--------------------------------------------------------------------------------------

**Quicksort**

Source: Author

--------------------------------------------------------------------------------------

**Merge Sort**

Source: Author

--------------------------------------------------------------------------------------

**Spyder IDE**

Source: Author

--------------------------------------------------------------------------------------

**A Trie of Some English Words**

Source: Author

--------------------------------------------------------------------------------------

**Patricia Trie Example**

Source: Author

--------------------------------------------------------------------------------------

**Pattern Matching**

Source: Author

--------------------------------------------------------------------------------------

**The Prefix Function**

Source: Author

--------------------------------------------------------------------------------------

**SOSML IDE**

Source: Author

--------------------------------------------------------------------------------------

**Control Flow Graph G1**

Source: Author

--------------------------------------------------------------------------------------

**Control Flow Graph G2**

Source: Author

--------------------------------------------------------------------------------------

**Test Coverage Report**

Source: Author

--------------------------------------------------------------------------------------

**Using the Memory Profiler in Python**

Source: Author

--------------------------------------------------------------------------------------

**Memory Usage Over Time**

Source: Author

-------------------------------------------------------------------------------------

**Another Example**

Source: Author

-------------------------------------------------------------------------------------

**Tkinter Label and Button**

Source: Author

-------------------------------------------------------------------------------------

**Typical Layout of a Process in Memory**

Source: Author

-------------------------------------------------------------------------------------

**WasmFiddle**

Source: Author

--------------------------------------------------------------------------------------

**Automatic Type Promotion in Java**

Source: Author

--------------------------------------------------------------------------------------

**Computing Mean**

Source: Author

-------------------------------------------------------------------------------------

**Displaying Scores and Frequencies**

Source: Author

-------------------------------------------------------------------------------------