COS 431 TENTATIVE MODULE ARRANGEMENT
(Note: Present this arrangement during revision)

#1 # Introduction (concepts and principles of algorithm)
- Problem solving by algorithm
- Role of algorithm in program development
- Implementation strategies for algorithm.

#2 # Developing algorithm and program/program from algorithm.
- Principles of good programming style (expressions and documentation)
- Structured programming concepts.
- Debugging, testing, verification, code inspection etc
- (Grey areas) cmt: not clear what this section specifies.

#3 # Design of algorithm
- Sorting and Searching
- Data Structure
- Recursion
- String processing (Assignment: Discuss these concepts and present the algorithms to implement them: abide by good programming principles. not more than 4 pages (typed))

#4 # Analysis of algorithms
- Semantic analysis
- Efficiency and performance of algorithms (comparative analysis)

#5 # Algorithmic strategies
- Brute force
- Greedy
- Divide- and- conquer
- Heuristic algorithms
- Numerical algorithms
- Patter matching and string processing algorithms, etc

#6 - Revision

COS 451 - ALGORITHMS

INTRODUCTION

1.1 This initial section of the course Algorithm, apparently not included in the outline, is intended to render insight into the basic concepts of algorithms. A brief discussion shall be made in the areas of history, definition, features and correctness of algorithms, examples and categories of algorithm, specification of algorithms, and scope of the course CS 451 at the end. We shall discuss herein the elementary aspects of the course, in tandem with the curriculum, and look at problem solving with algorithms, algorithms and program development, and attempt to end this section with implementation strategies for algorithms.

- In the perspective of history; the older form of the term "algorithm" first appeared in website's New World Dictionary as late as 1957, the older form is "Algorism".
- The term algorithm has an ancient meaning – The process of doing arithmetic using Arabic numerals.
- During the Middle Ages, abacists computed on the abacus and algorists computed by algorism.
- The term algorism subsequently changed to algorithm through many pseudo-etymological perversions. [Knuth, 1997 vol 1]

- In terms of definition: an algorithm is any well-defined computational procedure that takes some values, or set of values, as input and produces some value, or set of values, as output. [Cormen, et al, 2009].

  Thus, an algorithm is a sequence of computational steps that transform the input into the desired output.

  Alternatively, an algorithm could be seen as a tool for solving a well-specified computational problem.
- Another significant attempt towards defining the notion of algorithm is from the point of view of mathematical set theory by [Knuth, 1997, vol 1] which establishes an algorithm by formally defining a computational method as a

quadruple (Q, I, Ω, f) in which Q is a set containing subsets I and Ω, and f is a function from Q to itself. Furthermore f should leave Ω point wise fixed; that is f(q) should equal q for all elements q of Ω. The four quantities Q, I, Ω, f are intended to represent respectively the states of the computation, the input, the output, and the computational rule…

- Understanding the basic tenets of the above definitions will elucidate the features of algorithms that follow. Nevertheless, algorithms are known to concentrate on the high level design of data structures and methods for using them to solve problems. The subject is highly mathematical, but the mathematics can be compartmentalized and makes provision for emphasis to be placed on what rather than why. The assumed prerequisite is that scholars can take a description of an algorithm and relevant **data structures**, and use a **programming tool** to implement the algorithm.

- Having noted that an algorithm encompasses a finite set of rules that give a sequence of operations for solving a **specific** type of problem, we thus discuss the features and correctness of an algorithm. An algorithm has five important features:

(1) **Finiteness** - an algorithm must terminate after a finite number of steps. There are exceptions, of course, seen in computational methods and reactive processes, automatic teller machines, weather forecasts, and other TSR program.

(2) **Definiteness** – Each step of an algorithm must be precisely defined, the actions to be carried out must be rigorously and unambiguously specified for each case.

(3) **Input** – An algorithm has zero or more inputs: quantities that are given to it initially before the algorithm runs. These inputs are taken from specified sets of objects.

(4) **Output** – An algorithm has one or more outputs: quantities that have a specified relation to the inputs.

(5) **Effectiveness** – This implies that the operations of an algorithm must all be sufficiently basic that they can in principle be done i.e. exactly and in a finite length of time by someone using pencil and paper (manually).

Correctness of Algorithms

- An algorithm is said to be correct it, for every input instance, it halts with the correct output. Then we can assert that a correct algorithm solves the given computational problem.
- An incorrect algorithm might not halt at all on some input instances, a common mistake made by programmers, or it might halt with an incorrect answer. Nevertheless, incorrect algorithms can sometimes be useful, provided we can control their error rate.
- While we have mentioned some of the exceptions for termination in the case of the features of finiteness, we shell mainly concern ourselves with correct algorithms. The next section looks at examples and categories of algorithms.
-

1.2.1 Examples of Algorithms

(1) **Sorting Algorithms**

- Numerous computations and tasks become simple by properly sorting information in advance. Sorting can be with respect to a sequence of characters, say in alphabetical order, or a sequence of numbers into non decreasing order, etc.
- The above problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools.
- Even though we shall subsequently be discussing sorting in detail, here we formally define the problem: consider.

  Input: A sequence of n numbers $[a_1, a_2, \ldots\ldots, a_n]$

  Output: A permutation (recording) $[a_1^1, a_2^1, \ldots\ldots a_n^1]$ of the input sequence such that: $a_1^1 \leq a_2^1 \leq \ldots\ldots \leq a_n^1$.

To illustrate, given the input sequence <31, 41, 59, 26, 41, 58> The sorting algorithm returns as the output the sequence - <26,31,41,41,58,59>.

- Such an input sequence is called an instance of the sorting algorithm.
- In general, an instance of a problem consists of the input (satisfying whatever constraints are imposed in the problem statement) needed to compute a solution to the problem.
- Sorting is a fundamental operation in computer science for the reason that many programs use it as an intermediate step. As a result, there exists a large number of sorting algorithms. However, the choice of sorting algorithm, for any given application, depends mainly/mostly on the following,
- The number of items to be sorted.
- The extent to which the items are already somewhat sorted,
- The possible restrictions on the item values.
- The architectures of the computer, and the kind of storage devices to be used: main memory, disks, or even tapes.
- Examples of sorting algorithm, insertion, bubble sort, merge sort etc.

Other examples, to be mentioned briefly, include:

(2) Searching Algorithms:

- Searching algorithms are very critical in the operation of applications of all vintages: small to large and the World Wide Web. They function by finding the existence of a target item within a collection.
- Given a collection C of elements, there are three fundamental queries that could be asked.
    1. Existence: Does C contain a target element t?
    2. Retrieval: Return the element in C that matches the target element t.
    3. Associative lookup: Return information associated in collection C with the target key element key element k. [Heineman, 2009].

- Examples of searching algorithm include sequential search, binary search, hash-based search, etc.

(3) **Graph Algorithms:**

- Graphs are fundamental structures used in computer science to represent complex structured information. Inherently, a graph contains a set of elements, known as vertices, and relationships between pairs of these elements known as edges.
- Types of graphs that occur commonly in algorithms include; undirected, directed, weighted, and hyper graphs.

(4) **Network flow Algorithms**

- As we see in Operations Research – Network problems, there are numerous problems that can be viewed as a network of vertices and edges, with a capacity associated with each edge over which commodities flow.
- The application of this kind of algorithm is significant in solving problems of shortest path (minimum cost), transportation, transshipment, maximum flow (efficient capacity utilization), etc.

(5) **Computational Geometry Algorithms:**

- In Computer Science, the algorithms are useful for solving geometric problems. A computational geometry problem involves geometric objects, such as points, lines, and polygons. Precisely, a computational geometry problem is defined by;

  (a) The type of input data to be processed

  (b) The computation to be performed, and

  (c) Whether the task is static or dynamic.

  These classifications help identify the techniques that can improve efficiency across families of related problems.
- In modern engineering and mathematics, computational geometry has applications in such diverse fields as computer graphics, robotics, VSLI

design, Computer-aided design, molecular modeling, metallurgy, manufacturing, textile layout, forestry, and statistics.

- Input elements include, set of points, set of line segments, or the vertices of a polygon in counter clockwise order.
- The output is often a response to query about the objects, such as possible intersections, or new geometric objects etc.

1.2.2 Categories of Algorithm

- Algorithms can be categorized in terms of style, application, or complexity.

(a) **Style**

The Commonly used styles are divide-and-conquer, recursion, dynamic programming (bottom-up or memorization) and greedy strategy (do the best thing locally and hope for the best).

(b) **Application**

Common application categories include; mathematics, geometry, graphs, string matching, sorting, and searching.

(c) **Complexity Algorithms**

These algorithms are a larger category including any algorithm that must consider the best result among a large sample space of possibilities. Many complexity, or otherwise called combinational algorithms, are NP – complete. NP – complete problems are those that have shown not to be polynomially solvable (i.e. no better method or more efficient method for solving them).

1.3 Specification of Algorithms:

- Algorithms can be specified in English, as a computer program, or even as a hardware design. The only requirement is that the specification must provide a precise description of the computational procedure to be followed. This provision emphasizes the need to pay attention to the principles of good programming style (to be discussed subsequently).
- Nevertheless, some of the implementations of the algorithms we will encounter will be in the same high level language as contained in the texts: while some appear in C–like language, students should be able to implement some of the algorithm using OOP tools, such as Java or C++.

1.4 The Role of Algorithms in the Problem Solving Process:

This section discusses problem solving by algorithms, by taking a quick glance at some of practical applications of algorithms, and algorithms in the context of program development.

1.4.1 Problems Solving by Algorithms

The Sorting algorithm, discussed as an example in section 1.2.1, represents just a tiny fraction of problems algorithms can solve. There are several practical applications for which algorithm are known for efficacy. These include and are not limited to the following;

(i) The human Genome project – Towards identifying all the "100,000" genes in human DNA, storing this information in databases, and developing tools for data analysis. Each of these steps requires sophisticated algorithms. (The details of these algorithms are beyond the scope of this course).

(ii) The internet – very popular, enabling users to quickly access and retrieve large amounts of information. By using clever algorithms, sites on the internet are able to manage and manipulate large volumes of data. Typical among these include route finders (shortest path problem), search engines, etc.

(iii) Electronic commerce – enabling "goods" and "services" to be negotiated and exchanged electronically, and significantly depends on the privacy of personal and sensitive information, such as credit card numbers, passwords etc. The core technologies used in electronic commerce include public–key cryptography and digital signatures, which are based on numerical algorithms and number theory.

(iv) Manufacturing and other commercial enterprises – these require allocating relatively scarce resources in the most beneficial way, and desire to seek out the opportunity to minimize cost. (Optimization problems). Specifically;

- An oil company may wish to know where to place its wells in order to maximize its expected profit (maximum flow problems)
- A political candidate may want to determine where to spend "more" money buying campaign advertising in order to maximize chances of victory (Game Theory)
- An airline may wish to assign crews to flights in the least expensive way possible, making sure that each flight is covered and that extant regulations regarding crew scheduling are met. (Resource Allocation and Transshipment Problem).
- An Internet service provider may wish to determine where to place additional resources to serve customers better. (Resource Allocation Problem).

* The big question is – where can we not employ algorithms, as systems approach is inevitable for efficient and effective problem solving?

1.4.2 Algorithms and Program Development

- Conventionally, initial attempts to solve problems start with the analysis of the problem and the required data. After the analysis phase, a detailed procedural solution can emerge.
- An algorithm, being a computable set of steps to achieve a designed step, requires precise definition of steps for solving problems.
- Each step of an algorithm must be an unambiguous instruction to facilitate coding, which results in the actual program that can be executed by a computer. Importantly, knowledge of the factors that lead to improved performance of key algorithms is essential for the success of software applications. Needless to mention that implementation of most of the key algorithms can be found in literature and that, in practice, reinventing the wheel is usually not advisable. Nevertheless, the need to develop the skills of algorithms can never be overstated (see end note).
- Operationally speaking, the steps of an algorithm must be ordered, termination of an algorithms is necessary whether or not the algorithm is successful; although there are exceptions, such as weather monitors, automatic teller machines, and other TSRS mentioned earlier. Notably, algorithms are chosen on the basis of efficiency, accuracy, and clarity.

Activity: We can from the foregoing appreciate the place of algorithms in a typical program/software development cycle (Class Discussion).

1.5 Implementation Strategies for Algorithms

There are various ranges of options for implementing algorithms to solve specific problems. For reasons of space and time, we "list" a few;

(i) **Understanding the problem** - problem identification and analysis is highly essential; in the first instance starting with the big picture (clear wholistic view), understanding the problem, identifying potential causes and courses of action, and digging into the details are vital measures.

(ii) **Experimentation** – developing test algorithms and experimenting with them to ascertain suitability to problems being addressed.

(iii) **Practice with real codes** – it is reasonable to ensure that algorithms are translatable to actual codes in target high-level languages to demonstrate effectiveness.

(iv) **Optimization** – as an essential strategy for improving performance of algorithms: determining the quantitative behavior of algorithms is important, as the choice of an algorithm is dependent on the execution time, resource usage, adaptability (platform(s) independence), etc.

NB (end note)

You already know about the essential computer science data structure. Though we shall encounter some of them in due course, such as; array, linked lists, stack, queues, hash table, binary trees, directed and undirected graphs. It may not be necessary to implement these data structures, in practice, since they are typically provided by code libraries.

1.6 Scope of COS 451 – Algorithms

The outline of this course in accordance with NUC's guidelines can be found in the undergraduate prospectus. This highlights the following.

- Principles of good programming styles (expressions and documentation).
- Structured programming concepts
- Debugging, testing, verification, code inspection (Debugging algorithms)
- Semantic analysis
- String processing
- Data Structures
- Recursion
- Efficiency of algorithms

Nevertheless, there are some efforts to review and improve the content of this course at the departmental level. The delivery approach seeks to abide by some improvements and reflects on some contemporary issues in algorithms, while avoiding some grey areas both in the NUC guidelines and in the departmental reviews.
We hope to achieve the best – we can!

Recommend Text

(1) Introduction to Algorithms B
By Thomas H. Cormen, et al. Third Edition, MIT Press, 2009.
Consider Internet – based Resources.

2: A: Principles of Good Programming Style

Preamble

- Quite often, early attempts to programming assume that paying attention to style is unnecessary and amounts to waste of time. In reality, it has been proven that 80% of software costs go into maintenance of existing code, and virtually no modern software is written by only one developer throughout its life time.
- The need for developers to find ways of communicating designs with one another for purposes of software maintenance and reuse offers impetus for adherence to consistent design pattern or style.
- Suitably, style significantly reduces maintenance costs and increases the life time and functionality of software and/or programs. Software failures can most easily be attributed to poor style. Therefore, there is need to place emphasis on style during the program development life cycle.

2.1 General Principles.

- There are two fundamental principles for good programming style. These are readability and maintainability.
- Readability is concerned with writing good codes that can easily be understood even by a novice. Readable programs are usually properly and consistently formatted and uses clear meaningful names for functions and variables.

  In this range of programs (i.e. readable programs) there is explicit provision for **concise and accurate comments**, which describe a natural decomposition of the software's functionality into simple and specific functions. Sections where confusion and/or ambiguity can arise are clearly marked. With these provisions, it becomes easy to point out why a program can work and makes debugging easy as well.
- With respect to maintainability, codes should be written so that they are straight forward for another programmer to fix bugs or make functional changes in future.

  The functions and procedures should be general and assume minimal pre-conditions. All important values should be declared and marked as

constants that can easily be changed. Program codes are required to be robust to handle any possible input and produce reasonable output (result) without crashing.

Relevant messages (exceptions) should be displayed when an illegal or wrong input is made.

Furthermore, the subsequent section discusses the issues of expressions and documentation as relevant aspects of good programming style.

(a) Documentation

- Documentation of programs is the provision of all material that serves primarily to describe a system and make it more readily understandable, rather than contribute in any manner to the actual execution of the program.
- Documentation can be classified according to the purpose that it serves. Thus, for a given system, there may be; requirements and maintenance documents, there is also user documents for end users (manuals). People can and are making careers out of program/system documentation.
- With respect to algorithms and programs, comments are usually the first step towards making computer programs human-readable. Comments are required to explain everything about a program clearly. Nevertheless, abuse of comments, superfluous comments, can be a distraction from a good style: Quality counts.
- In C-like languages, block comments are denoted by "/* comments */" style. They normally appear at the beginning of a program showing name, date, purpose, overall description of program, etc.
- There is also provision for inline comments denoted by "// comments", which should go near important lines of code, within functions, or with variables when they are initialized.

(*Qn: Prior experience of writing programs in C).

[Hint: In any programming environment, try to discover and make use of the convention for comments].

Illustration:

```
/*
*my name
*
*This program
*/
Public styles class Myprogram  extents OurProgram
{
  /*
   *This method does…
    */
    Public Static void run C)
   {int x; //x in an integer representing…
     //some operation or x
    x = x * 7;
        }
}
```

- Another critical aspect of program documentation is naming. The names given to function and subroutines, in the case of algorithms, and to classes, variables, and functions in the actual codes should be unambiguous and descriptive. It is advisable to comply with the naming conventions that are applicable in the target programming languages.
- For instance, function names should start with a verb, makeAlternating Row ( ); createArray ( ), etc. Where: AlternatingRow is a name of a variable.
- The names i, j, and k as variables for loops
- Avoiding one letter names: is also much recommended. Preferable to state area = base * length; instead of a = b * h.

(b) Expressions

- Expressions are the components of a program that are executable. In the case of algorithms, they represent the aspects that must be translated to executable codes for the program(s) to function and perform the required tasks. Here, we shall discuss expressions, within the context of good programming style, in terms of indentations, declarations, spaces, and function usage.

(i) Indentation is used to clearly mark control flow in a program. Within any bracketed block, all code should be indented in one tab. This includes the functions body itself.

Illustration

```
Public static class MyProgram extends OurProgram
{
    Public Static void run (  )
      {
        int number;
        If (number > o)
        { number = number + 1 ;
        }
        Else
            { number = number + 2;
             }
      }
}
```

(ii) Declaration is the introduction of an entity for a part of the program, its scope, giving it a name and establishing its static properties.

- Examples are declarations of variables, procedures, input/output ports or files. Any significant number should be declared as a constant which is easily changeable. This improves readability and ensures that it will be changed where necessary.

(iii) Spaces are used to consistently improve readability. White spaces are meaningless to compilers. Typically, spaces should be used in between functions, and within function, to separate key sections.

(iv) Functions usage should be short and accomplish a clear, specific task, as much as possible they should be self-contained, or in other words, considered as "black boxes" which do not depend on anything except their parameters, and can handle any possible input gracefully. Functions should be as short as possible and usually should not exceed ten lines of code.

In conclusion, attention should be paid to how programs output results and information to users, a very relevant aspect of good programming style. Part of writing professional looking programs is providing clear instructions and results to the users of programs. This also requires proper language, i.e. English, without spelling or punctuation errors, with provisions for clearly explaining error conditions. It is safe to assume that end users are computer illiterate.

 # 2 B: Structured Programming Concepts

- The concept of structured programming started in the late 1960s with an article by Edsger Dijkstra. He proposed a "go to less" method of planning programming logic that eliminated the need for the branching category of control structures. The topic of structured programming was debated for about 20 years.

  However, towards the end of the 20th century, it became widely accepted that it is useful to learn and apply the concepts of structured programming.

Personnel Comments:

/* It is disturbing to note that NUC made the paradigm of structured programming a part of algorithms, whereas predominant issues in literature and researches involving algorithms are those of dynamic programming and linear programming. This in addition to those of debugging, testing, verification, code inspection (Principles of good programming style), etc, in my opinion, should be purged out of algorithm and be treated in modules that deal with programming languages proper. */. These sections will not be tested for.

- [According to oxford dictionary of computing],
- Structured programming is a method of program development that makes extensive use of abstraction in order to factorize the problem and give increased confidence that the resulting program is correct. Here abstraction means concentrating solely on the aspects of a program that are relevant to the current purpose.
- Procedural abstraction has been extensively employed since the early days of computing (in Fortran, Pascal, etc) with the effect that a function, clearly defined, can be treated as a single self-contained entity.
- In general, programming entails some form of control over the flow of program execution. The mechanisms that allow for flow control are regarded as the control structures. There are four main categories of control structures; [you may wish to illustrate, pls]

(1) Sequence - executing instructions as they appear one after the other (ordered execution).

(2) Selection – making a choice between two or more flows depending on some criteria

(3) Iteration - repetition to be executed several times as long as conditions remain true..

(4) Branching - A control structure that allows the flow of execution to jump to a different part of the program.

NB We shall be expecting these in the algorithms that we examine.

In the following section, we examine some of the methodologies, which have been developed for writing structured program. [These are available online].

(1) Edsger Dijkstra's structured programming: in this, the logic of a program is a structure composed of similar substructures in a limited number of ways. This reduces understanding a program to understanding each structure on its own, and in relation to that containing it, a useful separation of concerns.

(2) Entry point structured programming: This is a view derived from Dijkstra and advocates splitting programs into sub-sections with a single point of entry, but in strongly opposed to a single point of exit.

(3) Data structured programming (or Jackson's structured programming). This is based on aligning **data structures** with **program structures**. This approach applied the fundamental structures proposed by Dijkstra, but as constructs that used the high-level structure of a program to be modeled on the underlying data structures being processed.

It is to be noted that years after Dijkstra (1969), object oriented programming was developed to handle very large complex programs. We may therefore, present a comparative analysis of structured programming and object oriented programming. Nevertheless, it is pertinent to mention that low-level structured programs are often composed of simple, hierarchical program flow structures, which are sequence,

selection, and iteration (already discussed). And importantly, there is no provision for the “Go to” statement in structured programs, for which they are known.

Assignment [Class Discussion]

- Compare and contrast between structured programming and object oriented programming.

NB: The object of the assignment is for the students to be able to identify the right language for implementing any algorithm of interest and in general; reason about programming languages (development tools).

3: Design of Algorithms

- Sorting and Searching
- Data Structures

3:1 Introduction

Having gone through the definition, features, and some examples of algorithms, we can identify some relevant processes in the implementation strategy, or the value chain of algorithms, to include those of debugging, testing, verification, and code inspection. It is correct to assert that these processes fall within the remits of the implementation strategies we discussed earlier, which are, understanding the problem, experimentation, practicing with real code, and optimization.

For the purposes of understanding the concepts of algorithm design better, we note that Algorithm design is a specific method to create a mathematical process in solving problems. Applied algorithm design is algorithm engineering [Wikipedia].

Furthermore, Algorithm design is identified and incorporated into many solution theories of operation research, such as dynamic programming and divide-and-conquer. Techniques for designing and implementing algorithm designs are algorithm design patterns, such as template method pattern and decorator pattern, and uses of data structures, and name and sort lists. Some current day uses of algorithm design can be found in searching and internet retrieval processes of web crawling, packet routing and caching.

Mainframe programming languages such as ALGOL (for *Algo*rithmic *l*anguage), FORTRAN, COBOL, PL/I, SAIL, and SNOBOL are computing tools to implement an "algorithm design"... but, an "algorithm design" is not a language. An algorithm design can be a hand written process, e.g. set of equations, a series of mechanical processes done by hand, an analog piece of equipment, or a digital process and/or processor, just as we stated earlier.

One of the most important aspects of algorithm design is creating an algorithm that has an efficient run time, also known as its big Oh and/or the other asymptotic notations, which we shall discuss in the section that deals with efficiency of algorithms.

Steps in development of Algorithms:

1. Problem definition
2. Development of a model
3. Specification of Algorithm
4. Designing an Algorithm
5. Checking the correctness of Algorithm
6. Analysis of Algorithm
7. Implementation of Algorithm
8. Program testing
9. Documentation Preparation

To design algorithms, we shall be describing the steps of the algorithms in pseudocode, sometimes similar in many respects to C, C++, and Java. To make progress in algorithms design, it is necessary to abide by the principles of good programming style (already discussed).

It is pertinent to point out that pseudocode differs from "real" code, as pseudocode employs whatever expressive method that is most clear and concise to specify a given algorithm. Secondly, pseudocode is not typically concerned with some issues of software engineering: issues of data abstraction, modularity, and error handling are often ignored in order to convey the essence of the algorithm more concisely.

Comment:

- We proceed in accordance with the agenda of this section on design of algorithms by looking at sorting and searching algorithms, implementation of data structures, recursion, and string processing.
  Approach: what, why, and how (with examples/illustrations).

3.2 Sorting Algorithm

- Sorting is a topic that arises very frequently in programming: The rearrangement of items into ascending or descending order. Quite remarkably, the order in which items are stored in computer memory often has a profound influence on the speed and simplicity of algorithms that manipulate those items. In contrast to other terminologies of similar meaning, such as ordering or sequencing, sorting has become firmly established in computing parlance in the strict sense of sorting into order.

Some of the most important applications of sorting are;

(i) Grouping items of identical values together: this can be achieved by arranging items in ascending or descending order.

(ii) Matching items in two or more files: If several files have been sorted into the same order, it is possible to find all the matching entries in one sequential pass through them, without backing up.

(iii) Searching for information by key values: sorting is an important aid to searching. It helps, to make computer output, and indeed most texts, more suitable for human consumption. Simply imagine how labyrinthine it can be to search for a word in the dictionary without sorting, etc.

- Before proceeding to design an example sorting algorithm. We define our sorting problem more clearly and look at some terminology [knuth, vol 3, 1998]. Consider N items. R_1, R_2, …, R_N to be sorted; each R^{th} item is a record, and the entire collection of N records is called a file. Each record R_j has a key K_j, which governs the sorting process. Additional data,

besides the key, is usually also present; this entire information has no effect on sorting except that it must be carried along as part of each record.

- An ordering relation "<" is specified on the keys so that the following conditions are satisfied for any key values a, b, c:
  (i) Exactly one of the possibilities; a < b, a = b, or b < a is true (the trichotomy rule).
  (ii) If a < b, and b < c, then a < c (law of transitivity)

- Notably, properties (i) and (ii) characterize the mathematical concept of linear ordering, also called total ordering. Any relationship "<" satisfying these two properties can be sorted by most methods, although some sorting techniques are designed to work only with numerical or alphabetic keys that have the usual ordering.
- It is also important to mention that sorting can be classified generally into internal and external sorting. With internal sorting, the records are kept entirely in the computer's high-speed random-access memory, which allows for more flexibility in the structuring and accessing of the data. While external sorting becomes necessary when there are more records than can be stored in the RAM, implying the need for external storage, but posses some access constraints.

- The first type of sorting algorithm we examine is the insertion sort.

3.2.1 Insertion sort:

- Insertion sort works the way many people sort a hand of playing cards.
- Usually, people start with an empty left hand and the cards face down on the table.
- Then one card is removed at a time from the table and inserted into the correct position in the left hand.
- To find correct position for a card, we compare it with each of the cards already in the hand, from right to left.
- At all times, the cards held in the left hand are sorted, and these cards, were originally the top cards of the pile on the table.
- We therefore, proceed to present our sorting algorithm, called INSERTION SORT.

Recall our already introduced sorting problem where;

Input: A Sequence of numbers $< a_1, a_2, \ldots . a_n >$

Output: A permutation (recording) $< a_1, a_2, \ldots, a_n >$ of the input sequence $\ni a_1^1 \leq a_2^1 \leq \ldots \ldots . \leq a_n^1$

- The numbers to be sorted are known as the keys. Although conceptually, we are sorting a sequence, the input comes to us in the form of an array with n – elements. Thus, insertion sort takes as parameter an array A[1...n] containing a sequence of length n that is to be sorted (i.e cards on table).
- The algorithm sorts the input numbers in place, which implies that it rearranges the numbers within the Array A, with at most a constant number of them stored outside the array at any time.
- The input array A contains the sorted output sequence when the INSERTION-SORT procedure is finished.

INSERTION SORT

```
1       For   j   = 2 to A.length   //it's still ok to say A.n
2             key  =  A [ j ]
3             // insert A[j] into the sorted sequence A[1 …j -1]
4             i    =  j – 1
5             while i > 0 and A [i] > key /*while loop does not stop until A[i]
                                          < key or i = 0 */
6.                  A [i+1]  =  A[i]  // rightward shift (Interim insertion)
7.                       i   =  i  - 1
8             A [i + 1]  =  key //insert then transfer control to the for loop for
    next j
```

NB: Observe that when the while loop terminates, control in transferred to line 8.

To demonstrate the effectiveness of the INSERTION SORT algorithm, consider the array below;

1	2	3	4	5	6	← array indices
5	2	4	6	1	3	← array values

 # Loop Invariants and Correctness of Insertion Sort

First of all, we note that a loop invariant is a condition that is necessarily true immediately before and immediately after each iteration of a loop. [you can study more on loop invariants on your own].

Considering the given figure (algorithm);

- The index j indicates the current card being inserted into the hand.
- At the beginning of each interaction of the "for loop", which is indexed by j, the subarray consisting of elements A [1…j -1] constitutes the currently sorted array (hand), and the remaining subarray A [j + 1…n] corresponds to the pile of cards still on the table.

- In fact, elements A[1… J-1] are the elements originally in positions I through j – 1, but now in sorted order (this is the responsibility of the loop indexed by i : imagine an interim subarray)
- These properties of A[1…j-1] is formally stated as a loop invariant [Cormen, et al, 2009]:
  At the start of each interaction of the "for" loop of lines 1 – 8, the subarray A[1…j -1] consists of the elements originally in A[1…j -1], but in sorted order.

- {{The authors in [Cormen, et al, 2009]}} Loop invariants can be used to understand why an algorithm is correct by considering three properties, initialization, maintenance, and termination.

Where;

- Initialization - is time prior to the first iteration of the loop.
- Maintenance - is true before an iteration of the loop, it remains true before the next iteration.
- Termination - When the loop terminates, the invariant gives a useful property that helps to show that the algorithm is correct.

Considering the example array of values given;

5	2	4	6	1	3
1	2	3	4	5	6

- In the context of the loop invariants; for initialization:
  J = 2 by the first iteration.
- The second property, maintenance, provides that each of the iterations maintains the loop invariant. Informally, the body of the 'for' loop works by moving A[j-1], A[j-2], A[j–3], and so on by one position to the right until it finds the proper position for A [j] (lines 4 – 7), at which point it inserts the value of A [j] (line 8). The subarray A[1… j–1] then consists

of the elements originally in A[1…j–1], but in sorted order. Incrementing j for the next iteration of the 'for' loop then preserves the loop invariant.

- Termination - the loop terminates when for the "for loop", j > A. length = n; at a point where j = n + 1; as each loop iteration increases j by 1 and the sorted array becomes A [1….n] in sorted order.

Next: Merge sort algorithm before searching algorithm.

3.2.2 The Merge Sort Algorithm

- This is another type of sorting algorithm. The design approach used for the insertion sort is described as an incremental approach, where a single element A[j] was inserted into its proper place, yielding the sorted subarray A [1… j – 1].
- An alternative design approach known as the divide-and-conquer is used to design a sorting algorithm whose worst case running time can become much less than that of insertion sort (more efficient).
- The algorithms that follow the divide-and-conquer approach are those that break a problem into several sub problems that are similar to the original problem but smaller in size, solve the sub problems recursively, and then combine these solutions to create a solution to the original problem.

The merge sort algorithm follows the divide-and-conquer approach. The algorithm operates in the following manner;

(i) Divide the n – element sequence to be sorted into two subsequences of n/2 elements each.

(ii) Conquer by sorting the two subsequences recursively using merge sort.

(iii) Combine by merging the two sorted sequences to produce the sorted answer.

- The key operation of the merge sort algorithm is the merging of two sorted sequences in the "combine' step.

- The merge operation becomes effective by calling an auxiliary procedure;

  MERGE (A, P, q, r)

  Where; A is an array and p, q, and r are indices into the array such that.

  $P \leq q < r$.
- The procedure assumes that the subarrays A [P…q] and A [q + 1…r] are in sorted order..
- The procedure therefore merges them to form a single sorted subarray that replaces the current subarray A[P…r]
- We shall come to see in the section on efficiency that the procedure MERGE takes time $\Theta(n)$, where $n = r - p + 1$ is the total number of elements being sorted.
- To understand how MERGE SORT works, consider our card sorting example, but this time suppose we have two piles of cards face up on a table
- Each card is sorted with the smallest cards on top.
- We wish to merge the two piles into a single sorted output pile, which is to be faced down on the table.
- The basic step is to choose the smaller of the two cards on top of the face-up-piles, removing it from; its pile (which exposes a new top card), and placing this card face down into the output pile.
- Computationally, each basic step takes a constant time, albeit a simplifying assumption, since we are comparing just the two top cards. As we perform at most “n” basic steps, merging takes time $\Theta(n)$.

- The following pseudocode implements the above idea, but with an additional twist that avoids having to check whether either pile is empty in each basic step.
- We place at the bottom of each file a sentinel card (i.e. a guard), denoted by ∞, for reasons of simplicity.
- Once both sentinel cards are exposed, all the nonsentinel cards have already been placed on to the output pile.

- Since we know in advance that exactly r –p + 1 number of cards will be placed on to the output file, we can stop once we have performed that many basic steps.

MERGE (A, P, q, r)

1. $n_1 = q - P + 1$
2. $n_2 = r - q$
3. Let L[1… n_1+1] and R[1...n_2+1] //be new arrays each with space for the sentinels.
4. for i = 1 to n_1
5. L [i] = A[P + i-1]
6. for j = 1 to n_2
7. R [j] = A [q + j]
8. L [n_1 + 1] = ∞
9. R [n_2 + 1] = ∞
10. i = 1
11. j = 1
12. for k = P to r
13. if L[i] ≤ R[j]
14. A[k] = L[i]
15. i = i + 1
16. else A[k] = R[j]
17. j = j + 1

Description (in detail)

- Line 1 computes the length n_1 of the subarray A [P...q]
- Line 2 computes the length n_2 of the subarray A [q + 1…r]
- Line 3 creates the arrays L and R of lengths n_1 + 1 and n_2 + 1 respectively to accommodate the sentinels.
- Lines 4 – 5: the “for” loop copies subarray A[P...q] into L[1…n_1]

- Lines 6 – 7: the "for" Loop copies the subarray A [q + 1…r] into R[1…n_2]`
- Lines 8 – 9: put the sentinels at the ends of the arrays L and R.
- Lines 10 – 17: perform the r – p + 1 basic steps by maintaining the loop invariant.

Assignment 1

- Illustrate the merge sort algorithm with an array of not more than 15 elements in total and discuss how the merge sort maintains the loop invariant [10marks].

Important note; While we have considered the basic family of sorting: insertion sorting and merge sorts, it is important to note that other examples of insertion sorting include; the binary insertion, two-way insertion, the shellsort, list insertion etc. other sorting algorithms include the bubble sort, Quick sort, Radix exchange sort, Heap sort, etc; all of the which cannot be discussed in this course owing to limitations of time and space. Nevertheless, further reading/studying of these concepts will be of immense significance.

3.3 Searching

- Searching is another important and fundamental operation in computing. A searching algorithm operates by examining a set of data elements to find a designated (a given identification) target element.
- For example, in a numerical application, we might want to find f(x), given x and a table of the values of f. In a non numerical application, we might want to find the English translation of a given French/Greek/Russian word.
- For searching purposes, we suppose that a set of N records has been stored, with distinct N keys that uniquely identify the elements of the records. The collection of records is called a file or a table. A large file or group of files is frequently called a database.

- Algorithms for searching are presented with an argument, say K, and the problem is to find which record has K as its key. The result of a search operation will usually be any of the two possibilities: successful, having located the record containing K; or unsuccessful having determined that K is nowhere to be found. Searching involves the use of three fundamental queries, existence, retrieved, and associative lookup.

- The classification of searching algorithms is dependent on their operational nature: Internal or external; static or dynamic; based on comparisons between keys or on digital properties of keys; those that work with actual keys or those that work with transformed keys.

- Examples of searching algorithm include; the sequential search (brute force), binary search, and hash-based searches, which can be based on arithmetic transformations of the actual keys. We here attempt to discuss only sequential search and binary search owing to the constraints of time. Further studies on hash-based searches can be made with the support of ideas from the two treated.

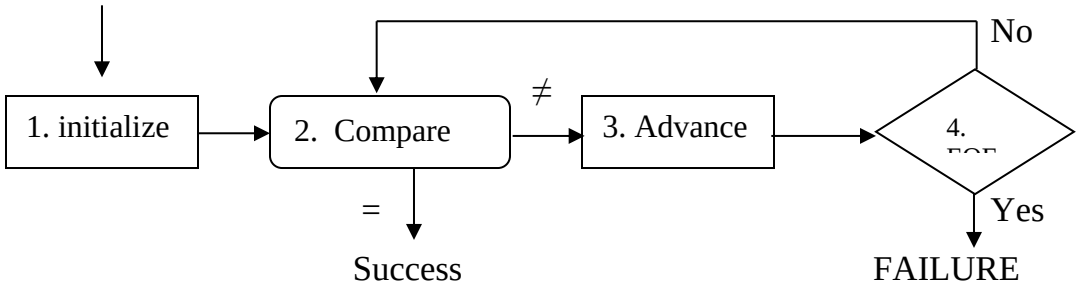
(1) Sequential Search.

This is also called linear search. “Begin at the Beginning”, and go on till you find the right key; then stop. Considering a target element, say of value t in some collection C; there are two conditions which terminate the search.

(i) The element is found, i.e. $a_i = t$ where $t \varepsilon C$.

(ii) The entire array has been scanned, and no match was found.

This can be illustrated with the following flow diagram:



- Input: The input consists of a collection, C, of $n \geq 0$ elements and a target item, t, being sought.
- Output: The output is a return of true if t belongs to C, and false otherwise.

The SEQUENTIAL SEARCH ALGORITHM

```
Search (A, t)
1. for I =  0 to  n -1  do
2.         If (A [ I ]  = t) then
3.              return true
4.    return false
   End
```

(2) Binary Search

Binary search, a popular divide-and-conquer searching strategy, is known to deliver better performance than sequential search over a table with a seemingly large amount of entries.

It operates by first of all dividing the entries into halves and determining the middle entry. If the middle entry corresponds to the target element, it terminates successfully; else it goes to determine the corresponding half with the greater probability of containing the target element. It terminates with failure, if the target element cannot be found in both halves. This drastically reduces the search time when compared to searching sequentially.

Binary search can be illustrated with the following flow diagram;

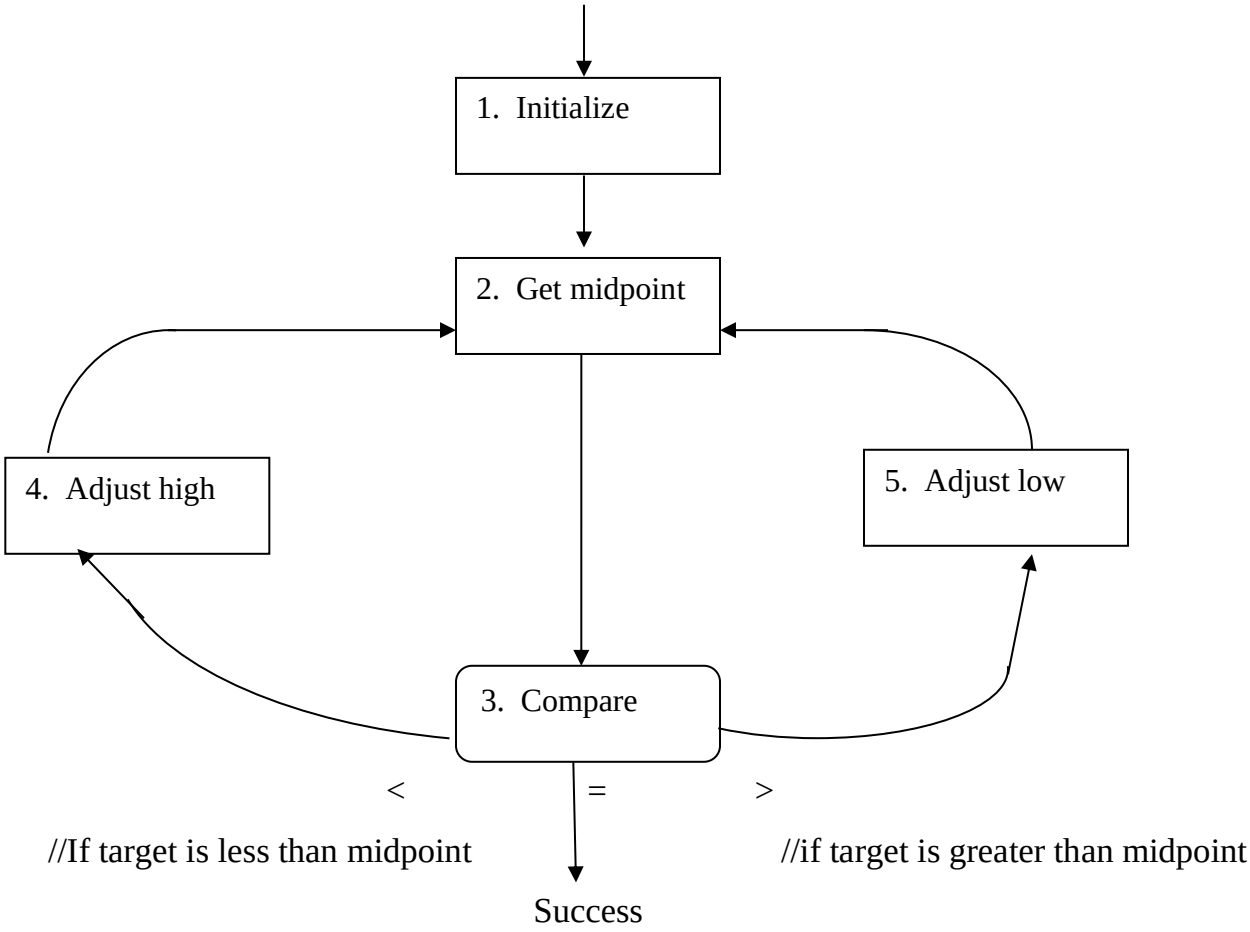


Figure: Binary Search

Binary search, therefore, is assumed to sort the elements in a collection in advance of query. Thus, Binary search is useful for searching ordered set:

Ak : $1 \leq k < N$ such that $a_{k-1} \leq a_k$

- It is important to note that the expected number of comparisons in binary search is N/2, while that of sequential search is N. (Refer to the section on efficiency for further details of their performance evaluation)

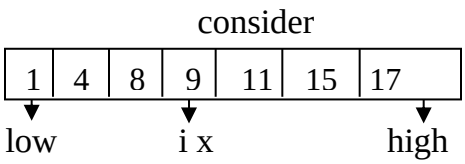
- Input: The input to binary search is an indexed collection, say A of elements. Each element A[i] has a key value K_i that uniquely identifies the elements. These keys are totally ordered, as stated, which means that given two keys K_i and K_j we have that $K_i < K_j$, $K_i = K_j$, or $K_i > K_j$. The data structure is constructed and preserves the ordering of the keys.

- Output: the output is either true or false.

BINARY SERARCH ALGORITHM

Search (A, t)

```
1     low  =  0
2     high   =  n – 1
3     while  (low ≤ high) do
4       ix  =  (low  + high)/2
5       If (t = A [ix]) then
6           return true
7     else if (t < A[ix]) then
8           high   =  ix – 1
9     else low   =  ix + 1
10    return false
End.
```



See the following comments.

Comment: we assume even numbered array for the above algorithm to be implementable. The actual arithmetic of other cases is beyond our discussion for now. There are other suggestions in literature for computing the mid point, such as those of ix = (R-L)/2+L; in order to fix the bugs associated with the actual implementation of the algorithm, as a matter of the problem of representing the finite bounds of the chosen integer types.

3.2 Data Structures

Preamble

- Computer programs usually operate on tables of data (information). To realize efficient algorithms and programs, it is very important to supply pre-knowledge of data types to the compiler in order to avoid storage allocation problems (i.e. problem of dynamic storage allocation).

- The amount of storage allocated, say to a variable, will have to be chosen according to the size of the range of values, and their relationships, that the variables may assume. Therefore, to use a computer properly, we need to understand the structural relationships present within data, as well as the basic techniques for representing and manipulating such structure within a computer.

- The data elements to be manipulated by a program are required to be associated with a type, which in turn is required to be made explicit by a declaration from the very beginning. Recall that the standard primitive data types are; integer, real, Boolean, characters, and set. The data type characterizes the set of values to which elements, such as constants, variables, expressions, or functions, belong to.

- In general the design of computer representations depends on the desired functions of the data as well as on its intrinsic properties (relationships). This section takes a look at some means for storage allocations and representation of structured data, in terms of the algorithms for creating, accessing, and destroying structural information.

Elementary Data Structures

- The elementary data structures, which form many complex data structures, are identified on the basis of the principal operations to be performed, just as it is notable that computer memories are distinguished by their intended applications.
- Here, the elementary data structures of interest are those of stacks, queues, linked lists, and rooted trees. We, therefore, consider some of the basic operations involving these data structures.

(1) Stack

- A stack is a linear data structure, can be implemented with an array, for which all insertions and deletions (and usually all accesses) are made at one end of the array or list. The most recent item (youngest) currently in the list is normally the first to be removed. This mode of operation is popularly tagged (LIFO).
- The implementation of stacks is useful in diverse areas of computing: in the process of entering and leaving subroutines; for the processing of languages with a nested structure, like programming languages, arithmetic expressions, etc. In general, stacks occur most frequently in connection with explicitly or implicitly recursive algorithms.
- A Stack is typically identified with a top (accessible item) and a bottom (least accessible item). The INSERT operation on a stack is usually referred to as PUSH, while the delete operation, which does not take an element argument, is regarded as POP. These terminologies (PUSH and POP) come from an analogy with the stacks of plates often found in cafeterias. They don't portend a physical motion in computer memory, as in principle, items are only added onto the top.
- Consider a stack with at most n – elements, i.e. an array S[1…n]. The array has an attribute S.top that indexes the most recently inserted element. Thus, the stack consist of elements S [1,., S.top], where S[1] is the bottom element and S[S.top] is the top element.

- When S.top = 0 – The stack is empty (comment: imagine a spring-based stack , where the top, a pointer, is always occupied as long as there is an item in the stack: Thus, empty stack implies nothing on top). An empty stack is tested with a query operation - STACK-EMPTY.
- An attempt to pop an empty stack, underflows, is an error, as well as attempt to exceed the top of the stack, overflows.
- We can, therefore, proceed to implement the stated operations on stack with the following pseudocode:

  Considering an array S [1…n].

STACK – EMPTY (S)

1. if S.top = = 0
2. return TRUE
3. else return FALSE

PUSH (S,x)

1. S.top = S.top + 1
2. If S.top > S[n], then OVERFLOW
3. else S [S.top] = x

POP (S)

1. if S.top = = 0, then UNDERFLOW;
2. else Y = S [S.top]
3. S.top = S.top - 1
4. return Y

// Earlier Pseudocode (Alternative)

POP (S)

1. IF STACK – EMPTY(S)
2. error " underflow''
3. else S.top = S.top - 1
4. return S [S.top + 1]

(2) **Queues**

- A queue is another dynamic set, a linear data structure, for which all insertions are made at one end of the list, and all deletions (and usually all accesses) are made at the other end. In a queue, the element deleted is always the one that has been in the set for the longest time (FIFO).

- Note that there is another variant of the linear data structure called Deque (double–ended–queue), pronounced as deck and has some properties in common with a deck of cards. A deque is a linear list (data structure) for which all insertions and deletions (and usually all accesses) are made at the ends of the list. We shall ignore the details and operations on deque in this course, in order to keep things simple.

- Thus, continuing with queues; the Insert operations on a queue is usually regarded as ENQUEUE, while the delete operation is DEQUEUE (don't confuse this with deque)

- The FIFO property of a queue causes it to operate like a line of customers waiting to pay a cashier. The queue has a head and a tail attributes associated with it. A newly arriving element is at the tail, while the element dequeued is always the head.
- The following pseudocodes can be used to implement the basic operations of a queue. Considering a queue Q, an array Q [1,…, n] where n is the length of queue.

(1) ENQUEUE (Q, x)

1. If Q. tail = = Q. length //checking of Q is empty
2. Q. tail = 1 //better to start at position 1
3. else Q.tail = Q.tail + 1
4. If Q.tail > Q.length, the OVERFLOW
5. Q [Q.tail] = x

Compare this with the earlier pseudocode (Alternative)

ENQUEUE = (Q, x)

1. Q [Q.tail] = x
2. If Q.tail = = Q. length
3. Q.tail = 1
4. else Q.tail = Q.tail + 1

(2) DEQUEUE (Q)

1. If Q.head = Q.tail, then UNDERFLOW;
2. If Q.head = = Q.length
3. Q.head = 1
4. else Q.head = Q.head + 1
5. x = Q [Q.head]

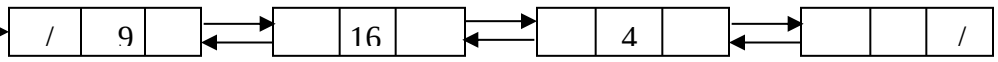
NB

- The elements in the queue reside in locations Q.head, Q.head + 1 ,…, Q.tail – 1, where we "wrap around" in the sense that location I immediately follows location n in circular order.
- When Q.head = Q.tail, the queue is empty.
- Initially, we have Q.head = Q.tail = 1
- An attempt to dequeue an empty queue causes underflows.
- When Q.head = Q.tail + 1, the queue is full; an attempt to enqueue a full queue causes overflows.

(3) Linked Lists.

- A linked list is a data structure in which the objects are arranged in a linear order. Unlike an array, however, in which the linear order is determined by the array indices, the order in a linked list is determined by a pointer in each object.
- Linked lists provide simple and flexible representation for dynamic data sets.

Illustration

(a) L. head 

- As shown in the above figure, each element in a doubly linked list (as the illustration portrays) is an object with an attribute key and two pointer attributes, next and previous.
- Given an element x in the list, where x. previous = NIL implies no predecessor and thus the first element or the head of the list.
- IF x . next = NIL – no successor or tail of the list.
- A list may be singly linked or doubly linked, it may be sorted or not and circular or not. In singly linked list, previous pointers are absent. In sorted linked list, the linear order of the lists corresponds to the linear order of the keys. In circular linked list, the previous pointer of the head points to the tail while the next pointer of the tail points to the head.
- The following pseudocodes may be used to implement search, insertion, and deletion operation on a linked list.

(1) LIST – SEARCH (L,k)

1 x = L.head
2. while x ≠ NIL and x.key ≠ k
3 x = x . next
4. return x.

(2) LIST – INSERT (L, x)

1. x.next = L.head
2. If L.head ≠ NIL
3. L.head. prev = x
4. L.head = x
5. x . prev = NIL

Comment

Given an element x whose key attribute has already been set, the LIST-INSERT procedure joins (splices) x unto the front of the linked list. The attribute notation can cascade, so that L.head previous denotes the previous attribute of the object that L.head points to. The running time of LIST-INSERT on a list of n-elements is **O**(1) (big Oh of 1).

(3) LIST-DELETE (L, x)

1. If x . prev ≠ NIL
2. x.prev.next = x.next
3. else L.head = x.next
4. If x.next ≠ NIL
5. x.next.prev = x.prev.

- The above code reassigns the next pointer of x to the previous adjacent element and the previous pointer to the next adjacent (i.e. succeeding) element. Thereby removes all pointers to x that was intended to be deleted.
  LIST – DELETE runs in **O**(i), but to delete an element with a given key, the worst case requires Θ(n) time.

(4) ROOTED TREES

- Rooted trees can be represented by linked data structures. Infact, trees are the most important non-linear structures that arise in computer algorithms. Generally speaking, tree structure means a "branching" relationship between nodes, much like that found in the trees of nature.

- The nodes of the tree are treated as objects, in which it is assumed that each node contains a key attribute. The remaining attributes are pointers to other nodes, and they vary according to the type of tree.

- Formally, a tree can be defined [Knuth, vol 1, 1997] as a finite set T of one or more modes such that

  (a) There is one specially designated node called the root of the tree, root (T), and

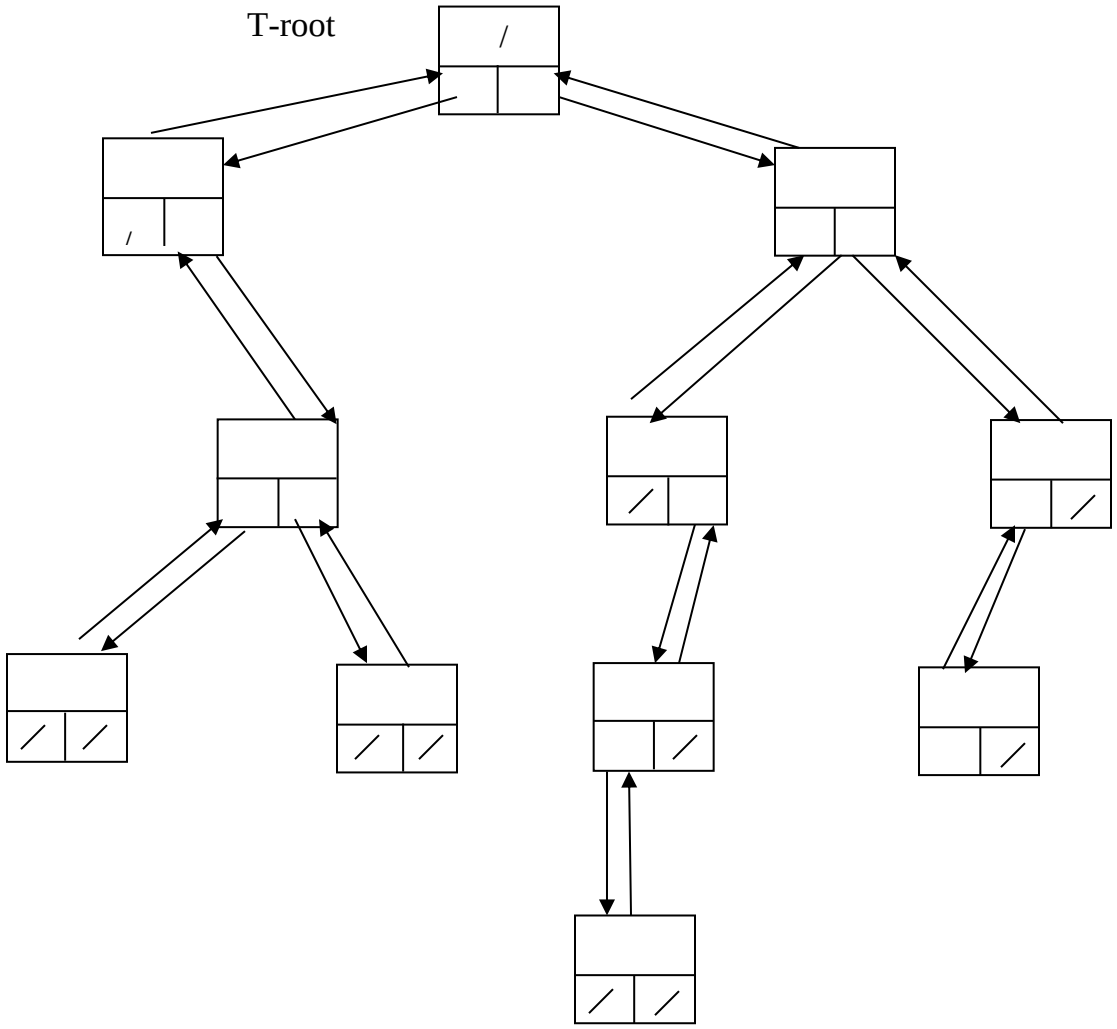
  (b) The remaining node (excluding the root) are partitioned into $m \geq 0$ disjoint sets $T_1,\ldots,T_m$, and each of these sets in turn is a tree. The trees $T_1\ldots,T_m$ are called the subtrees of the root.

- The above definition simply implies recursion, defining a tree in terms of trees. Recursion is an innate characteristic of tree structures.
- The recursive character of trees is such that every node of a tree is the root of some subtree contained in the whole tree. The number of subtrees of a node is called the degree of that node. A node of degree zero is called a terminal node, or sometimes a leaf. A non terminal node is often called a branch node.

(a) Binary Trees

Binary trees will be our only tree type of interest owing to limitations of time. In a binary tree each node has at most two subtrees; and when only one subtree is present, we distinguish between the left and right subtree. Formally, a binary tree can be defined as a finite set of nodes that either is empty, or consists of a root and the elements of two disjoint binary trees called the left and right subtrees of the root.

Illustration



In binary trees, the attributed P, left, and right are used to store pointers to the parent, left child, and right child.

- If x.P = NIL, then x is the root
- If x.left = NIL ≡ no left child and so on.
- The root of the entire tree T is pointed to by the attribute T. root, if T.root = NIL, then the tree is empty.

(b) Rooted trees with unbounded branching:

- The number of children of each node is at most some constant k; such that the left and right attributes are replaced by child 1, child 2,…, child k.
- This scheme no longer works when the number of children of a node is unbounded: A lot of memory can be wasted when k is bounded by a large constant.
- In conclusion students may wish to study more about rooted trees and heap data structure on their own. It is okay to conclude this section of data structures with focus on stack, queues, and linked lists.

4: Analysis of Algorithms

- Efficiency and performance of algorithm
- Semantic Analysis.

4.0 Introduction

- The topics of the design and analysis of algorithms is very important, as programming and program development can become complex and complicated tasks. Such a scenario, considering large projects, especially, require coordinated efforts of many people, storing and accessing large quantities of data efficiently, and solving complex computational problems.
- Intuitively, quite a number of solutions can be designed to solve a particular problem. The set of solutions, usually regarded as the set of feasible solutions, provide the basis from which the optimal solution (if it exists) can emerge through the process of stepwise refinement.
- In algorithms' design and analysis, the process of refinement considers the objective and/or quantitative properties of the algorithms in terms of efficiency, correctness, and sometimes semantic analysis of the target algorithm.
- In this section of the course, we therefore discuss efficiency of algorithms, and briefly discuss the issue of semantic analysis.

4.1 Efficiency and Performance of Algorithms

- From the foregoing introduction, we can picture that analysis of algorithms in quite important in computer programming, since we have argued that there are usually several algorithms available for a particular application, the important and sensible task is to determine which option is the best for a particular problem.
- The analysis framework for algorithms in the first instance, considers the efficiency and correctness of algorithms.
- Efficiency considerations usually investigate the dual issues of time and space. Where time efficiency deals with how much time an algorithm

uses, and space efficiency deals with how much space (memory allocation) the algorithm uses, in addition to the space needed for the input and output.

- For a better understanding of the topic of efficiency and the motivation for choosing one algorithm over the other (objectively or quantitatively), let us consider a comparison between sequential and binary search, which we have already discussed.
- Suppose we have a million records to search with sequential search, on average we would search half of the list – 500,000 records before we find what we are looking for.
- If the item we are looking for is the first one in the list, then we pull a single record; if it is the last one, then we end of pulling all one million records.
- With binary search, we look at one record the middle one – per "turn", then split the list in half; so, we go from 1 million to ½ million, to ¼ million, to 125,000, to 62,500,…, 30, to 15, 7, 3, 1, having a maximum of 20 "turns" to go through all million records.
- Thus, the best case, optimistic instance, for binary search can be with the first record (single record) while the worst case is to pull a maximum of 20.
- The above, and similar exercises, is the concern of analysis of algorithms – where, in the case of sequential and binary searches, we looked at the number of "turns" which translates into time. Another measure we may wish to consider is the respective space (memory) each requires.
- We also look at each algorithm in general, over a range of instances. We note that there are best, worst, and average cases for an algorithm. These situations may vary widely depending on the class(es) of algorithms.
- Considering, again, the sequential and binary search, their respective best, worst, and average cases can be profiled for one million items as follows;

	Best case	**Worst Case**	**Average Case**

Sequential Search	First item in list (one turn)	Last item in list (one million turns)	Middle of list (500,000 turns)
Binary Search	Middle of list (one turn)	Reduce list to one or less: 20 “turns”	Halfway through the reduction, 10 turns.

- Note that the table is specific to a list of one million items. Nevertheless, the table suggests that binary search is the faster algorithm. But as the table is just for 1 million records, it is not quite general (cannot be generalizable).
- We therefore consider a more suitable representation as the size of the input, say n, grows arbitrarily. This will lead us to a more generalized table (relation).
- The above scenario leads us to the task of generalisation by considering the growth of functions as discussed in most literature, given that certain algorithms will perform better than their counterparts at some values of input but behave otherwise as input values exceed certain threshold.
- For reasons of simplicity, we replace the one million records in our table by n; and the table becomes.

	Best case	**Worst Case**	**Average Case**
Sequential Search	First item in list (one turn)	Last item in list n “turns”	Middle of list : n/2 “turns”
Binary Search	Middle of list (one turn)	Reduce list to one or less: $\log_2 n$ “turns”	Halfway through: $\log_2 (n/2)$ “turns”.

Comment

We can easily extend this analysis to insertion sort and merge sort.

4.1.2 Approaches (methods) of Analysis of Algorithms

- There are fundamentally two approaches for analysis of algorithms. These are; the theoretical analysis – using proof of correctness and asymptotic representations (big – oh and related notations); and using empirical analysis – involving testing and measurement over a range of instances (benchmark). In this category, from literature, we have the random access machine (RAM) model [Cormen, et al, 2009], the obsolete MIX machine [Knuth, 1997], etc.
- Irrespective of the approach and as stated earlier, considerations of efficiency can be by: as a function of Input size, by measuring running time, or by worst-case, best-case, and average-case analysis.

(1) Efficiency as a function of input size: Typically, more time and space are needed to run an algorithm on bigger inputs (e.g, more numbers, larger strings, larger graphs). To analyse efficiency as a function of n = size of input;

- Searching/sorting – n = number of items in list
- String processing – n = length of string(s)
- Matrix operations – n = dimension of matrix

  n x n matrix has n^2 elements.
- Graph processing - n_v = number of vertices and

  N_g = number of edges.

(2) Measuring Running Time:

Measuring time efficiency of algorithms involves:

- Identifying the basic operation(s) contributing the most to running time,
- Characterizing the number of times it is performed as a function of input size.
- Basically, we can estimate running time T(n) by : $T(n) \approx Cop * C(n)$.

  Where:

  T(n) - running time as a function of n

  Cop - running time of a single basic operation, and

  C(n) - number of basic operations as a function of n.

(3) **Worst-case, Best-case, and Average-case Analysis:**

- Considering sequential search, for instance, which searches for a target element t in an array A of n elements, the number of turns to make will run from 0 to n – 1 [or can we say from 1 to n?].

We can state the case analysis as follows:

- Basic operations: The comparison in the loop
- Worst case: n comparisons
- Best case: 1 comparison
- Average case: (n + 1)/2 comparisons

  assuming each element equally likely to be searched.

4.1.3 Order of Growth Classifications (Growth of Functions)

Some of the values of functions important for analysis of algorithms are summarized in the following table in order of magnitude:

n	**Log $_2$ n**	**N**	**n log$_2$ n**	**n ^{2}**	**n ^{3}**	**2 ^{n}**	**n!**
10	3.3	10^1	3.3 x 10^1	10 ^{2}	10 ^{3}	10^3	3.6 x 10^6
10^2	6.6	10^2	6.6 x 10^2	10 ^{4}	10 ^{6}	1.3 x 10^{30}	9.3 x 10 ^{157}
10^3	10	10^3	1.0 x 10^4	10 ^{6}	10 ^{9}		
10^4	13	10^4	1.3 x 10^5	10 ^{8}	10 ^{12}		
10^5	17	10^5	1.7 x 10^6	10 ^{10}	10 ^{15}		
10^6	20	10^6	2.0 x 10^7	10 ^{12}	10 ^{18}		

- **Constant:** a program whose running time's order of growth is a constant executes a fixed number of operations to finish its job; consequently its running time does not depend on n, the size of input.
- **Logarithm:** a program with this running time is barely slower than a constant-time program. The classic example of a program whose running time is logarithmic in the problem size is binary search. The base of the logarithm is not relevant with respect to the order of growth (since all logarithms with a constant base are related by a constant factor), so we can use log n when referring to order of growth.
- **Linear:** programs that spend a constant amount of time processing each piece of input data, or that are based on a single 'for loop' are quite common. The order of growth of such program is said to be linear: its running time is proportional to n.
- **Linearithmic:** this is also seen texts, and used to describe programs whose running time for a problem of size n has order of growth of nlog n. Again the base of the logarithm is irrelevant with respect to the order of growth. The prototypical examples of linearithmic algorithms are merge-sort and quick-sort algorithms.

- **Quadratic:** a typical program whose running time has order of growth of n^2 has two nested 'for loops', used for some calculation involving all pairs of n elements. The elementary sorting algorithms, such as Selection-Sort and Insertion-Sort are prototypes of the programs in this classification.
- **Cubic:** a typical program whose running time has the order of growth of n^3 has three nested 'for loops', used for some calculation involving the triples of n elements. An example is a program to calculate three-sums.
- **Exponential:** the term exponential is used to refer to algorithms whose order of growth is b^n for any constant $b > 1$, even though different values of b lead to vastly different running times. Exponential algorithms are extremely slow: you will never run one of them to completion for a large problem. Still, exponential algorithms play a critical role in the theory of algorithms: because there exists a large class of problems for which it seems that an exponential algorithm is the best choice.

The above stated classifications are most common, but certainly not the complete set. The order of growth of an algorithm's cost might be $n^2 \log n$ or $n^{3/2}$ or some similar function. Indeed, the detailed analysis of algorithms can require the full gamut of mathematical tools that have been developed over the centuries.

It is extremely important to note that one of the primary reasons to study the order of growth of a program is to help design a faster algorithm to solve the same problem: by simply taking advantage of the implication of their running times on specified input size, n.

#4.1.4: Empirical Analysis of Efficiency of Algorithm

- With the foregoing discussion on theoretical method of analysis, we can deduce that the task of analyzing an algorithm can easily be reduced to the determination of the resources that the algorithm requires.

  Occasionally, resources such as memory, communication bandwidth, or hardware are of primary concern. However, computational time remains the usual candidate for measurement. We reiterate that analyzing several algorithmic options for a given problem presents the opportunity to identify the most efficient algorithmic solution.
- In practical or empirical terms, models of implementations technology for algorithmic options are known to be useful for comparative analysis of algorithms. There can, of course, be models for the resources of a particular technology and their attendant costs. One viable option (empirical method) for comparative analysis of algorithms is the random access machine model (RAM) [Cormen, et al, 2009]. Another is the mix machine model, which was nearly abandoned but there are indications of modernization [knuth, 1997]. Mix will not be discussed here.
- In the RAM model, instructions are executed, by assumption, one after another, with no provisions for concurrency. The RAM model contains instructions commonly found in real computers such as; arithmetic operations of addition, subtraction, multiplication, division, remainder, floor, ceiling; data movement operations of load, store, copy; and control operations of conditional and unconditional branch, subroutine call and return.
- Each of such instructions takes a constant amount of time. For reasons of simplicity, the data types in the RAM model are integer and floating point. Also, an assumed limit is placed on the size of each word of data, so that the word size does not grow arbitrarily.
- Continuing, the issue of exponentiation is regarded as having a constant time, even though in practice it may not be so especially considering operations such as x^y where x and y are real numbers.

- Whereas RAM model analysis is useful for performance prediction on actual machines, analyzing a simple algorithm in the RAM model can still pose a challenge. Nevertheless, the need for a method that is simple to write and manipulate, which shows the important characteristic of an algorithm's resources requirements, even by suppressing tedious details, can never be over-emphasised.

 # Illustration:

(1) Analysis of Insertion Sort Using RAM Model

- Insertion sort is one of the critical algorithms that we discussed in the preceding section of this course (Design of Algorithms).We re-note that the time taken by the inserting-sort procedure depends or the input: Sorting a million numbers takes longer time than sorting a hundred; two input sequences of same size can sort at different times depending on how nearly sorted they are. Nevertheless, traditionally, the running time of a program is regarded as a function of the size of its input.
- Thus, we distil out the two parameters of interest; "running time" and "input size". For some problems, the best measure of input size is the total number of bits needed to represent the input in ordinary binary notation. While in others such as a graph, the input size can be represented by a number such as the number of vertices and edges, in the graph. The running time is a function of the number of primitive operations or "steps" or "turns" executed.
- In consonance with the RAM model, it is acceptable to assume that a constant time is required to execute each line of the INSERTION-SORT algorithm. One line may take a different amount of time than another line, but it can be assumed that each execution of the ith line takes time Ci, where Ci is a constant
- In our Insertion-sort procedure, j = 2,3,..., n; where n = A.length, let tj denote the number of times the while loop test in line 5 is executed for that value of j.

When a “for” or “while” loop exists in the usual way (i.e. due to the test in the loop header), the test is executed one time more than the loop body. Comments are not executable statements, and therefore, take no time.

INSERTION-SORT (A)		Cost	Times
1.	for j = 2 to A.length	C_1	n
2.	key = A [j]	C_2	n - 1
3.	/* Insert A [j] into the sorted seq A[1…j-1] */	0	n - 1
4.	i = j -1	C_4	n - 1
5.	While i > o and A[I] > key	C_5	$\Sigma^{n}_{j=2}$ tj
6.	A[i + 1] = A [i]	C_6	$\Sigma^{n}_{j=2}$ (tj -1)
7.	i = i – 1	C_7	$\Sigma^{n}_{j=2}$ (tj-1)
8.	A [i + 1] = key	C_8	n - 1

- The running time of the algorithm is the sum of the running times for each statement executed: a statement that takes C_i steps to execute and executes n times will contribute C_i n to the total running time.

  Thus, to compute the total time of insertion sort

  T (n);

$$T(n) = C_1 n + C_2 (n-1) + C_4 (n-1) + C_5 \sum_{j=2}^{n} Ej + C_6 \sum_{j=2}^{n} (tg - 1) + C_7 \Sigma (t j - 1) + C_8 (n - 1)$$

NB Bear in mind that the actual running times may differ for best – case; where input sequence is somewhat sorted, and worst –case; without any initial sorting. The worst – case running time of an algorithm gives an upper bound (known as the O-notation) on the running time for any input size. Determination of this value provides a guarantee that the algorithm will never take a time longer.

Illustration 2:

(2): Analyzing Divide and Conquer Algorithms

- When an algorithm contains a recursive call to itself, the running time can be described by a recurrence equation or recurrence, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs.

Consider:

```
MERGE-SORT (A, P, r)
1.    if  p  <  r
2.        q  =  [(p + r)/2]  {use the floor function for odds input sizes}
3.         MERGE-SORT (A, P, q)
4.         MERGE-SORT  (A, q + 1, r)
5          MERGE  (A, P, q, r)
```

- A recurrence for the running time of a divide –and – conquer algorithm falls out from the three basic steps of the paradigm (i.e. divide, conquer, and combine).
- As usual, let T(n) be the running time on a problem of size n. if the problem size is small enough, say $n \leq C$ for some constant C, the straight forward solution takes constant time, which is denoted as $\Theta(1)$ (i.e. $\Theta(1)$: big theta of 1).
- Suppose that the division of the problem yields "a" sub problems, each of which is 1/b the size of the original (for merge-sort both "a" and "b" are 2, but there are some divide-and-conquer algorithms in which $a \neq b$).
- It takes time T (n/b) to solve one sub problem of size n/b, and so it takes time aT (n/b) to solve "a" of them.

  Where; a = no of sub problems (divisions);

  b = no of elements in each sub problem
- If we take D(n) time to divide the problem into sub problems and C(n) time to combine the solutions to the sub problems into the solution to the original problem, we get the recurrence.

$$T(n) = \begin{cases} \theta(1) & \text{If } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise.} \end{cases}$$

Analysis of Merge-Sort in the RAM Model

- The pseudocode for merge-sort works correctly when the number of elements is not even [Cormen, et al 2009]. For simplicity, the original problem is assumed to be the power of 2. Thus, each divide step yields two subsequences of size exactly n/2.
- For the worst-case running time of merge-sort on n numbers, where merge-sort on just one element takes a constant time, when there are n > 1 elements, the running time is as follows;
- Divide: This computes the middle of the sub array and takes a constant time.

  Thus; $D(n) = \theta(1)$.
- Conquer: This recursively solves two sub problems of size n/2 each and contributes time 2 T (n/2) to the running time.
- Combine: The merge procedure on an n-element sub array takes time
- $\theta(n)$ and so $C(n) = \theta(n)$
- When we add the functions D(n) and C(n) for the merge-sort analysis, we are adding a function that is $\theta(n)$ and a function that is $\theta(1)$. This sum is a linear function of n, that is , $\theta(n)$. Adding this result to the 2 T(n/2) term from the conquer step gives the recurrence for the worst-case running time T(n) of merge-sort as:

$$T(n) = \begin{cases} \theta(1) & \text{if } n = 1 \\ 2\,T(n/2) + \theta(n) & \text{if } n > 1 \end{cases}$$

- It has been proven, just as we saw in the time analysis of binary and sequential search, that merge-sort, with its $\theta(n \log n)$ running time, out performs insertion-sort, whose worst-case running time is $\theta(n^2)$, as n becomes increasingly large (i.e $n \to \infty$).

Comment

- Before we introduce the asymptotic notations, including the Θ-notation, it may be necessary for us to justify the essence of the asymptotic notations.
- Considering the obtained values from RAM model, as empirical measurements, and again consider herewith a certain result of the analysis of time complexity of insertion sort algorithm; where
- Worst – case T(n): $K^1(n-1) + k_1 n (n-1)/2$;
- Average – case T (n): $K^1(n-1) + k_1 n (n-1)/4$; and
- Best – case T(n): $K^1(n-1)$.

- It is obvious, from the expressions of the running times, that it can be quite complicated and difficult to keep track of or memorize these running times. More difficulty still will be the case of the running time of rather more complicated algorithms. Therefore, any scientific approach that can ameliorate this difficulty is greatly desirable. It is in this regard, and for relative ease of comparative analysis of similar algorithms, that the asymptotic notations are required {look-alikes]
- Bearing in mind that the asymptotic notations provide more convenient expression and classification of performance parameters (efficiency: Time & Space) for algorithms with large input sizes (i.e. $n \rightarrow \infty$), by ignoring lower order terms and constant coefficients, as they are parameters that are machine (or environment) dependent.
- At such large values of n, one of the terms in the expression for running time (the one with the highest order, power) will usually dominate over (lower-order) terms, so that the effects of the lower-order terms become negligible. (NB lower-order means having smaller power of n)

Illustration (example)

- With respect to the running times of the insertion sort algorithm given.
- Looking at the best–case T(n);

  T(n): $k^1(n-1)$

  $= k^1n - k$
- As $n \rightarrow \infty$: k^1n will dominate; we thus ignore other terms and consider only the term with n, which is k^1n.

  We state, as it is by convention and formal presentation of Θ-notation.

  The Best – case running time of insertion sort $T(n) = \Theta(n)$
- Similarly:
- Worst – case $T(n) = k^1 (n-1) + k_1 n (n -1) /2$

  $= k^1 n - k + k_1 n^2 - kn/2$

The dominant term $= k_1 n^2$

The lower order terms k^1n and (-kn) can be ignored as their relative contribution to the general expression (ratio) with n^2, as $n \rightarrow \infty$, become infinitesimal (negligible).

Comment

[Remember that in the real world, the input sizes can indeed be very large – google searches, and other parameters with large distribution]

Thus the Θ-notation for the worst – case running time $T(n) = \Theta(n^2)$

- The average – case is also $\Theta(n^2)$
- These expressions will more conveniently replace the otherwise more difficult, but exact, expressions for the running times of insertion sort. Intuitively speaking, the essence of the running time expressions is to categorize the efficiency of the algorithms effortlessly for reasons of making informed design decisions.

NB

As $n \rightarrow \infty$, a $\Theta(n^2)$ algorithm will always out-perform a $\Theta(n^3)$ algorithm.

- The background given leads us to formally define the Θ-notation.

4.1.5 The Asymptotic Notations (Growth of Functions):

- In our consideration (comparative analysis) of sequential search and binary search, we saw that, intuitively, we could compare their performances over an input size of one million records. However, we remarked that such simplistic comparisons will be difficult as the input size(s) grow(s), which makes generalization difficult.
- In order to overcome the aforementioned constraints as input sizes grow, an analytical and theoretical approach called asymptotic efficiency of algorithms is most commonly used.
- Asymptotic efficiency of algorithms makes it possible to simply consider input sizes large enough to make only the order of growth of the running time relevant. In this regard, emphasis is just on how the running time of an algorithm increases with the size of the input in the limit, as the size of the input increases without bound (i.e. within limited time, as input size is unbounded). Usually, an algorithm that is asymptotically more efficient will be the best choice for all but very small inputs. [Cormen, et al, 2009].
- Asymptotic analysis of efficiency of algorithms therefore is a method of describing limiting behaviour by using asymptotic notations.
- Asymptotic notations, also called Bachmann–Landau notation, describe limiting behaviours of functions as arguments of the functions grow or tends to infinity.
- Succinctly, the asymptotes provide a way of comparing functions by ignoring constant factors and small input sizes.
- Typically, the basic operation count can be approximated as $Cg(n)$, where $g(n)$ is the order of growth.

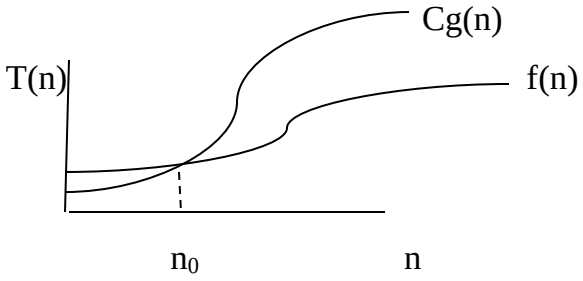
Comment

- In this section, we formally define some of the asymptotic notations, as briefly as possible, owing to our limits. These are the big 0 (O - notation), the big omega (Ω - notations), the theta (Θ - notation), the little o (o - notation), and the little omega (ω - notation) respectively.

(1) The O - notation

- The O – notation (pronounced big Oh) is used when there is only an asymptotic upper bound. It is one of the standard notations in computer science for characterizing the efficiency of an algorithm, in terms of running time or space requirement for a problem of size n written as O (expression), where expression is the expression in terms of n.
- The O-notation is formally defined as follows; for a given function with value g(n), $O(g(n)) = \{f(n)$: there exist positive constants C and n_0 such that $0 \leq f(n) \leq Cg(n)$ for all $n \geq n_0 \}$

* Illustrated graphically as:



- The O-notation is used to give an upper bound to the set of values of a function, to within a constant factor.
- Alternatively, we can view the O(g(n)) as follows;

  O(g(n)): functions $\leq$ C g(n)

In terms of time;

T(n) ε O(g(n)) if there are positive constants C and n_0 ∋ T(n) $\leq$ Cg(n) for all $n \geq n_0$

- Different functions with the same growth rate may be represented using the same O–notation. It is commonly used to describe how closely a finite series approximates a given function, by omitting constant factors and lower order terms. The O-notation specifically describes worst-case scenario.

Examples:

(1) Using O-notation to simplify functions:

Consider: $f(x) = 6x^4 - 2x^3 + 5$

- Recall that the asymptotic notation ignores small input values and constant factors. Of the terms in f(x) above, the one with the highest growth rate is the one with the largest exponent as a function of x, namely $6x^4$, which is the product of 6 and x^4. The first factor does not depend on x. This leaves us with the simplified form as x^4.
- Thus, $f(x) = O(x^4)$ (i.e. f(x) is big O of (x^4))
- Applying formal definition of the O-notation, the statement that $f(x) = O(x^4)$ is equivalent to its expansion;

  $/f(x)/ \leq m /g(x)/$ for some suitable choices of x_0 and M for all $x > x_0$

  [The burden of proof and application is on students].

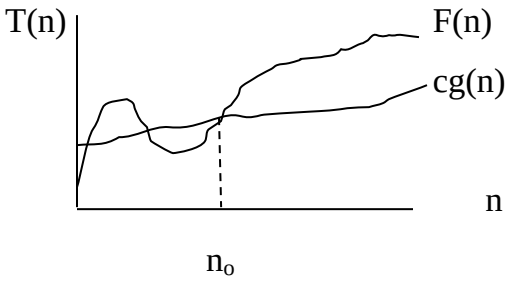
(2) Suppose an algorithm is being developed to operate on a set of n-elements, where it is required to determine time T(n) for the algorithm to run in terms of the number of elements in the input size.

- The algorithm works by first calling a subroutine to sort elements in the set and then performs its own operations.
- The sort procedure has a known time complexity of $O(n^2)$, and after the subroutine runs, the algorithm must take an additional $55n^3 + 2n + 10$ time before it terminates.
- Thus the overall time complexity of the algorithms is $T(n) = O(n^2) + 55n^3 + 2n + 10$. Using the O-notation, simplify T(n).

2. The Ω – notation:

- The Ω – notation (pronounced "big-omega of g of n") provides an asymptotic lower bound, just as the O-notation provides an asymptotic upper bound on a function.
- For a given function g (n), the Ω – notation is usually denoted at Ω (g (n)), and formally defined as:

  Ω (g (n)) = { f(n): There exist positive constants c and n_o such that $o \leq c\ g(n) \leq f(n)\ \ n \geq n_o$}
- This can be illustrated graphically as follows;



- from the above figure, for all values n at or to the right of n_o (some threshold) the value of f(n) is on or above cg(n).

Example

(1) Prove if an algorithm with time complexity T (n) = $an^2 + bn + c$ is Ω (n^2).

Hint: This implies that there exists a constant $C_1 > 0$

such that $C_1n^2 \leq an^2 + bn + c$.

Assignment: Prove (1).

NB Asymptote - A line that tends towards a curve without meeting it at any finite time.

(2) Recall that 0(f(n)) or 0(g (n)), O – notation, is the set of all functions with the same or a smaller order of growth than f(n)/g(n) e.g. $0(n^2) = \{n^2, 100\ n + 5, \log n, \ldots.\}$

Thus,

$n^3 ¢ 0\ (n^2)$

In contrast, Ω(f (n)) in the set of all functions with a larger or the same order or growth as f(n)

where

$\Omega(n) = \{n^2, n^3, n^4 + 10, \ldots\}$

but

$6n + 100 \notin \Omega(n^2)$

as $\Omega(f(n)) = \{T(n)$: there exists $c_I > o \ni c_I f(n) \leq T(n)$ for all $n \geq n_o\}$

Proof:

(1) for $6n + 100$ to be a member of $\Omega(n^2)$ i.e.

$6n + 100 \in \Omega(n^2)$

Then there must be a constant multiple ($c_1 > o$) that will bind T(n), which in this case is $6n + 100$, from below:

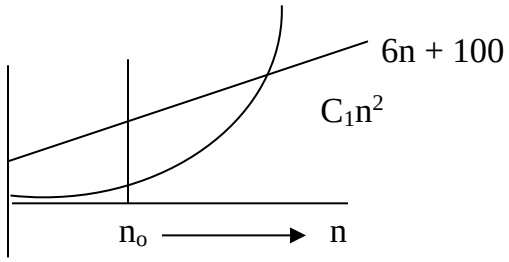
$\longrightarrow C_1 n^2 \leq 6n + 100$ which should imply that $C_1 n^2 - 6n - 100 \leq 0$,

$\forall$ large $n \geq n$ o. But considering the dormant term n^2 (as $n \to \infty$) for $c > 0$;

$C_1 n^2 \neq 0$ { cannot be less than zero }

Thus, $6n + 100 \notin \Omega(n^2)$

- A look at the graph will show that $C_1 n^2$ will cross the line of $6n + 100$ and thus will (i.e. $6n + 100$) fail to represent a lower bound. The graph follows:



(ii) Showing that $n^3 \in \Omega(n^2)$

Implies that there exists a constant $C_1 > 0$ (a positive constant)

such that $n^3 \geq C_1 n^2$

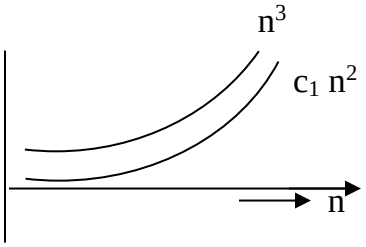
i.e. $C_1 n^2 \leq n^3$

$\to C_1 \leq n$

The above can hold true always no matter the value of C_1 we choose as $n \to \infty$, i.e. as n becomes increasingly large – (C_1 is some constant, don't forget)

Thus $n^3 \in \Omega(n^2)$

Graphically:



(3) The Θ - Notation

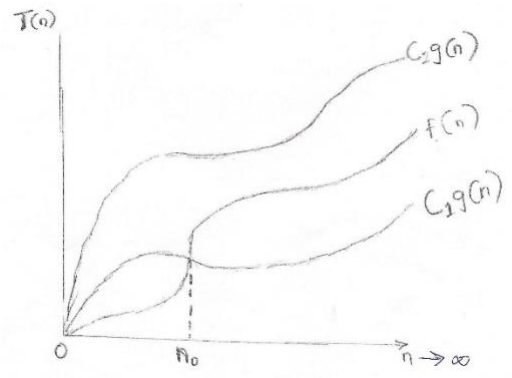
- The big Theta notation.
- **The Θ-notation**

  For a given function g(n), we denote by Θ(g(n)) the set of functions:

  $\Theta(g(n)) = \{f(n)$: there exist positive constants

  C_1, C_2, and n_o such that

  $\theta \leq C_1 g(n) \leq f(n) \leq C_2 g(n)$ for all $n \geq n_o\}$
- This implies that the function f(n) belongs to the set Θ(g(n)) if there exist positive constants C_1 and C_2 such that it can be "sandwiched" between $C_1 g(n)$ and $C_2 g(n)$, for sufficiently large n.
- Θ-notation is illustrated graphically:



- The Θ(f(n)) or Θ(g(n)) is the set of all functions with the same order of growth as f(n) or g(n), as the case may be.

Note: T(n) = Θ(f(n)) is equivalent to stating that T(n) = Ω(f(n)) but not the converse.

- If T(n) = Ω(f(n)) and T(n) = O(f(n)).

Then T(n) = Θ(f(n)).

- {Onus of proof and example on students}.
- The Θ-notation allows us to immediately determine the algorithm that can run faster, all things being equal.
- Generally, asymptotic notations are the machine–independent (i.e. actual machine, compiler, and programming language) notation for the running times, and perhaps other efficiency parameters, of an algorithm.
- We take a quick look at the remainder; the o-notation and the ω-notation.

(4) The o -notation

- The little o-notation is relevant as the asymptotic upper bound provided by O-notation may or may not be asymptotically tight. The bound $2n^2 = O(n^2)$ is asymptotically tight, but the bound $2n = O(n^2)$ is not. [Cormen, et al, 2009]
- The o-notation (little – oh of g of n) → o(g (n))

  is formally defined as; O (g(n) = {f(n): for any positive constant c > o, there exists a constant $n_o > o$ such that $0 \leq f(n) < cg(n)$ $\forall\ n \geq n_o$}
- For example, $2n = o(n^2)$, but $2n^2 \neq o(n^2)$
- Observe that the definition of O-notation and o-notation are similar. The main difference is that in f(n) = O(g(n)), the bound $o \leq f(n) \leq cg(n)$ holds for some constant c > o, but in the case of o-notation;
- f(n) = o(g(n)), the bound $0 \leq f(n) < cg(n)$ holds for all constants c > o. Intuitively, in o-notation, the function f(n) becomes insignificant relative to g(n) as n approaches infinity; that is,

$$\lim_{n \to \infty} f(n)/g(n) = 0$$

This is sometimes used as the definition for o-notation.

(5) The ω -notation

- The ω-notation (little – omega notation) is used to denote a lower bound that is not asymptotically tight. Just as o-notation is to O-notation, ω-notation is to Ω-notation.
- We formally define ω-notation as;

  $\omega(g(n))$ = { f(n): for any positive constant c > o,

  there exists a constant n_o> o, such that $0 \leq c\, g(n) < f(n)$, for all $n \geq n_o$}

  Alternative definition is given by;

  $f(n) \in \omega(g(n))$ iff $g(n) \in o(f(n))$
- For example $n^2/2 = \omega(n)$, but $n^2/2 \neq \omega(n^2)$.

- The relation $f(n) = \omega(g(n))$ implies that

  $$\lim_{n \to \infty} f(n)/g(n) = \infty$$

  The above means that f(n) becomes increasingly larger than g(n) as n becomes larger and tends to infinitely

4.2 Semantic Analysis of Algorithms:

- The subject of semantic analysis is very popular in compiler construction where it intrinsically deals with the issues of parsing and semantic analysis among others.
- Whereas parsing only verifies that a program consists of tokens that are syntactically correct, semantic analysis checks whether the program tokens form a sensible set of instructions in programming language or algorithms, as the case may be.
- Semantic analysis, therefore, is a vital part of initial phase of program correctness verification that enables the compiler to weed out incorrect programs. It is absolutely necessary to ensure that a program is sound enough to carry on to code generation.
- A major part of semantic analysis consists of tracking variable, function, type declarations, and type checking. In the majority of languages, identifiers are declared before they are used. Whenever a compiler encounters a new declaration, it records the type of information assigned to that identifier and then continues to examine the rest of the program, verifying that the identifier type is respected in the context of the operations being performed.
- For example; types of both sides of an assignments statement, type match for parameters of a function, ambiguity arising from two global declarations sharing the same name, arithmetic operands being numeric (no automatic int-to-double conversion). These are all checked in the semantic analysis phase.
- Fundamentally, therefore, semantic analysis involves type–checking, scope checking, and a few other steps. The phase is all about verifying the language rules, especially those that are two complex or difficult to contain in the grammar.
- With Scope checking, a language offers some control for scopes by constraining the visibility of an identifier to all or some subsection of a

program. Global variables are functions that are available anywhere, while local variables are only visible within certain sections.

- In C, a scope is a section of a program text enclosed by basic program delimiters; “{ }”. There can be nested scope. The innermost scope is the current scope.
- Scope checking is, therefore, the process of determining the points of accessibility of an identifier in a program. The results are such that an error occurs when attempt is made to access a local variable declared in one function in another function.

NB - It may be sufficient to end the discussion of semantic analysis at this Point (definition) while details of implementation method are recommended as further work (to be continued).

Lecture 5: Algorithmic Strategies:

Here we examine the main algorithmic paradigms as the basis for categorization according to the strategy that they employ.

5.1 Simple Recursive Algorithms.

- Quite generally, recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem. The significance of recursion evidently lies on the possibility of defining an infinite set of objects by a finite statement.
- A recursive procedure/function definition has one or more base cases, meaning input(s) for which the function produces a result trivially (without recurring), and one or more recursive cases, meaning input (s) for which the program recurs (i.e. calls itself). Recursive algorithms are known to do some extra work to convert the solution to the simpler sub problem(s) into a solution to the given problem.

Examples

(1) The factorial function

Factorial (n)

(1) $0! = 1$

(2) for $n > 0$

(3) $n! = n (n - 1)!$

Neither equation by itself constitutes a complete definition; the first is the base case, and the second in the recursive case. Because the base cases break the chain of recursion, they are sometimes called “terminating cases”.

(2) To Count the number of elements in a list

(1) If the list is empty, return zero; otherwise

(2) Skip past the first element, and count the remaining elements in the list

(3) Add one to the result.

(3) To test if a value occurs in a list

(1) If the list is empty, return false; otherwise

(2) If the first thing in the list is the given value, return true; otherwise.

(3) Step past the first element, and test whether the value occurs in the remainder of the list.

5.2 Backtracking Algorithms

- Backtracking is a general algorithm for finding all (or some) solutions to some computational problem that incrementally builds candidates to the solution, and abandons each partial candidate *c* ("backtracks") as soon as it determines that *c* cannot possibly be completed to a valid solution.
- A backtracking algorithm is based on a depth-first recursive search. The algorithm proceeds as follows:

(1) Tests to see if a solution has been found, and if so, returns it; otherwise.

(2) For each choice that can be made at this point

- make that choice
- Recur
- If the recursion returns a solution, return if

(3) If no choices remain, return failure.

Example of backtracking algorithm

- Colour a map with no more than 4 colours:

Colour (Country n)

(1) If all countries have been coloured (n > no of countries)
    return success; otherwise,

(2) for each colour c of four colours,

(3) if country n is not adjacent to a country that has been coloured C

(4) - Colour country n with colour C

(5) - recursively colour country n + 1

(6) - If successful, return success

(7) Return failure (if loop exits)

5.3 Divide- and- conquer Algorithms

- Typically, a divide-and-conquer algorithm consists of two parts.
  (1) A divide part: that divides the problem into smaller subproblems of the same type and recursively solves the subproblems
  (2) A combine step: That combines the solutions to the sub problems into a solution to the original problem.
- Traditionally, an algorithm is only called divide-and-conquer if it contains two or more recursive calls.

Examples of divide-and-conquer algorithms:

(1) Quicksort:
  - Partition the array into two parts, and quicksort each of the parts
  - No additional work is required to combine the two sorted parts.

(2) Mergesort:
  - Cut the array in half, and mergesort each half
  - Combine the two sorted arrays into a single sorted array by merging them.

(3) Binary Search:
  - Divide the array in two halves, and binary search each half for the target element.

- No additional work is required to combine the two searched halves.

5.4 Dynamic Programming Algorithms

- Dynamic programming is still a method of solving complex problems by breaking them down to simpler subproblems. The method is applicable to problems exhibiting the properties of overlapping subproblems and optimal substructure. The method is more efficient than other methods that do not take advantage of the subrountine overlap (like depth-first search).
- Dynamic programming is especially useful when the number of repeating subproblems grows exponentially as a function of the input size. The approach seeks to solve each subproblem only once, thus reducing the number of computations.
- In effect, a dynamic programming algorithm remembers past results (memo-ized) and uses them to find new results. Dynamic programming is generally used for optimization problems in which the following apply;

(i) Multiple solutions exist: need to find the best one

(ii) Requires optimal substructure and overlapping subproblem

(iii) Optimal substructure: Optimal solution contains optimal solutions to subproblemss

(iv) Overlapping subproblems: solutions to subproblems can be stored and re-used in a bottom-up fashion.

- Dynamic programming differs from divide-and-conquer as the subproblems generally need not overlap-divide-and-conquer.

Examples of Dynamic Programming Algorithms:

- Several examples exist, even in literature, as there is no standard formulation for DPPs. Some are'

(1) Rod-cutting: Deciding where to cut steel nods; where an enterprise wishes to know the best way to cut up the rods. [Cormen, et al, 2010, page 360].

Problem: Given a rod of length n inches and a table of prizes P_i for i = 1, 2, …, n. Determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Note that if the prize P_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

(2) Some example in bioinformatics, such as

(a) Compute an optimal pairwise alignment.

- Optimal substructure: the alignment of two prefixes contains solutions for the optimal alignments of smaller prefixes.
- Overlapping subproblems: The solution for the optimal alignment of two prefixes can be constructed using the stored solutions of the alignment of three subproblems (in a linear gap model).

(b) Compute a Viterbi path in HMM

- Optimal substructure: The viterbi path for an input prefix ending in a state of an HMM contains shorter Viterbi paths for smaller parts of the input and other HMM states.
- Overlapping subproblems: The solution for the Viterbi path for an input prefix ending in a state of an HMM can be constructed using the stored solutions of Viterbi paths for a shorter input prefix and all HMM states.

5.5 Greedy Algorithms

- Sometimes it becomes inefficient to solve problems of optimization using dynamic programming. A greedy algorithm is more efficient than DP, and makes the choice that looks best at the moment. Though, greedy algorithms do not always yield optimal solutions, but they are effective for many problems.
- Thus, we can assert that a greedy algorithm sometimes works well for optimization problems. A greedy algorithm works in phases.

  At each phase;

  - You take the best you can get right now, without regard to future consequences.
  - You hope that by choosing a local optimum at each step, you will end up at a global optimum.

Examples:

(1) The problem of scheduling several competing activities that require exclusive use of a common resource, with a goal of selecting a maximum-size set of mutually compatible activities.

- Suppose we have a set $S = \{a_1, a_2, \ldots, a_n\}$ of n proposed activities that wish to use a resource, such as a lecture hall, which can serve only one activity at a time. Each activity a_i has a start time s_i and a finish time f_i, where $0 \leq f_i < \infty$. If selected, activity a_i takes place during the half-open time interval $[s_i, f_i)$. Activities a_i and a_j are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap. That is, a_i and a_j are compatible

If $s_i \geq f_j$ or $s_j \geq f_i$.

In the activity – selection problem [cormen, et al, 2010, P415], it is required to select a maximum-size subject of mutually compatible activities. The assumption is that the activities are sorted in monotonically increasing order of finish time.

$f_1 \leq f_2 \leq f_3 \leq \ldots \leq f_{n-1} \leq f_n$

You can continue.

And rpove that greedy algorithm out performs DP.

(2) Counting a certain amount of money, using the fewest possible bills and coins. A greedy algorithm would perform this task by talking the largest possible bill or coin that does not overshoot.

(3) Other greedy algorithms are;

(i) Dijkstra's algorithm for finding the shortest path in a graph: always takes the shortest edge connecting a known node to an unknown node.

(ii) Kruskal's algorithm for finding a minimum cost spanning tree: Always tries the lowest-cost remaining edge.

(iii) Prims algorithm for finding a minimum cost spanning tree: Always takes the lowest cost edge between nodes in the spanning tree and nodes not yet in the spanning tree, etc.

5.6 Branch- and- bound Algorithms

- Branch-and-bound algorithms are generally used for optimisation problems (recall integer programming problems). As the algorithm progresses, a tree of subproblems is formed. The original problem is considered the root problem. A method is used to construct an upper and lower bound for a given problem.
- At each node, apply the bounding methods;
    - If the bounds match, it is deemed a feasible solution to that particular sub problem.
    - If bounds do not match; partition the problem represented by that node, and make the two subproblems into children nodes.
- Continue, using the best known feasible solution to trim sections of the tree, until all nodes have been solved or trimmed.

Example:

(1) The travelling salesman problem (TSP): A Salesman having to visit n – states (at least) once each, and wishes to minimize total distance travelled.

- Consider the root problem to be the problem of finding the shortest route through a set of cites visiting each city once.
- Split the node into two child problems:
    - Shortest route visiting city A first
    - Shortest route not visiting city A first
- Continue Subdividing similarly as the tree.

5.7 Brute Force Algorithm

- A brute force algorithm simply tries all possibilities until a satisfactory solution is found. Such an algorithm can be:

(i) Optimizing: find the best solution. This may require finding all solutions (feasible solutions) or if a value for the best solution is known, it may stop when any best solution is found. (example: finding the best path for a travelling salesman.)

(ii) Satisficing: Stop as soon as a solution is found that is good enough (example: finding a travelling salesman path that is within 10% of optimal.)

NB Brute force is a straightforward approach to solving a problem without regard to efficiency.

Example: A **O**(n) algorithm for a^n:

```
algorithm Power (a, n)
// Input: A real number and an integer  n ≥ 0
// Output: a^n
result  ← 1
for i ← 1  to  n do
        result  ←  result  *  a
return result.
```

- Other examples include string matching, and other numeric computations, such as matrix multiplication, solving recurrences, etc.

NB Despite the inefficiency of brute force algorithms, they are applicable to a wide range of problem, are simple to design, and may be useful in solving small problem instances.

5.8 Pattern Matching Algorithm

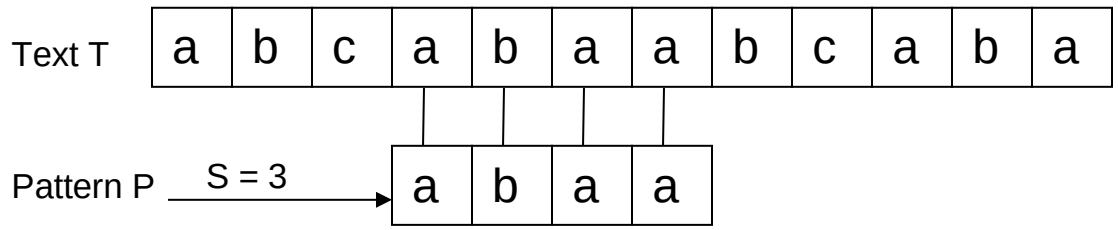
- Pattern matching is a term often used interchangeably with string matching without loss of generality. Text-editing programs, in other words text-processing systems, frequently need to find all occurrences of a pattern in the text. Precisely, the systems must allow their users to search for a given character string within a body of text.

- For the purposes of database management systems, the systems must be capable of searching for records with stated values in specified fields. Such problems typically pose the following string-matching problem: For a specified set {X(i), Y(i)} of pairs of strings, determine, if possible, an r such that X(r) = Y(r). Usually the set is specified not by explicit enumeration of the pairs, but rather by a rule for computing the pairs <X(i), Y(i)> from some given data. There are several algorithms, especially randomized algorithm, for solving problems of this nature, such as those of [Karp and Rabin, 1987].

- There are many other applications for which string-matching is known to be very essential. Some of which are; searching for particular patterns in DNA sequences, finding web pages that are relevant to queries by Internet search engines, etc.

- According to [Cormen, et al, 2009/10] we can formalize a string-matching problem as follows: assuming a text to be an array T[1…n] of length n and that the pattern is an array P[1…m] of length $m \leq n$. With a further assumption that the elements of P and T are characters drawn from a finite alphabet Σ. For example, we may have $\Sigma = \{0,1\}$ or $\Sigma = \{a, b, \ldots, z\}$. The character arrays P and T are often referred to as strings of characters.

- Considering the following figure:



- As indicated in the above string matching problem, the pattern P can be said to occur with shift S in the text T. Equivalently, the pattern P occurs beginning at position S+1 in text T. This implies that $0 \leq S \leq n-m$ and $T[S+1, \ldots S+m] = P[1 .. m]$, that is $T[s+j] = P[j]$, for $1 \leq j \leq m$. If P occurs with shift S in T, then we call S a valid shift; otherwise, we call S an invalid shift. The string-matching problem is the problem of finding all valid shifts, with which a given pattern occurs in a given text T.

- There are several proposals for pattern-matching algorithms, which progressively address the issue of improved performance. These include those of Radin-Karp, Knuth-Morris-Pratt, Bayer-Moore (BM), Horspool (HORSPOOL), Raita (RAITA), [Sheik, et al, 2003].

- Some analytical results of the comparative study of performance of some of the algorithm is shown in the following table:

Algorithm	Preprocessing Time	Matching Time
Naïve	0	0((n-m+1)m)
Rabin-Karp	0(m)	0((n-m+1)m)
Finite automation	0(m $\lvert\Sigma\rvert$)	0(n)
Knuth-Morris-Pratt	0(m)	0(n)

- However, we limit our discussion to the Naïve algorithm. The naïve algorithm represents the direct way of doing the business of string-matching. It is also referred to as the naïve brute-force algorithm. Being a brute-force method, it operates by comparing the first m-characters of the text and the pattern in some predefined order and, after a match or a mismatch, it slides the entire pattern by one character in the forward direction of the text.
- The above process is repeated until the pattern is positioned at the (n-m+1) position of the text.

NB: Recall that it is for this same task that several algorithms have been proposed, and these have their own advantages and limitations based on the pattern length, periodicity, and the type of text (for example, nucleotide or amino acid sequences or language characters, etc).

- Next, we present the naïve string-matching algorithm, which finds all valid shift, using a loop that checks the condition P[1..m] = T[s+1..s+m] for each of the n-m+1 possible values of s.

NAÏVE-STRING-MATCHER (T, P)

1. n = T. length

2. m = P. length

3. for S = 0 to n-m

4. if P[1..m] = = T [s+1 .. s+m]

5. print “Pattern occurs with shift” S

Comments:

- The “for” loop of lines 3 – 5 considers each possible shift explicitly.

- The text in line 4 determines whether the current shift is valid; this test implicitly loops to check corresponding character positions until all positions match successfully or a mismatch is found.

Exercise(s)

(1) Show the comparisons the naïve string-matcher makes for the pattern P = 001 in the text T = 000010001010001.

(2) Suppose that all characters in the pattern P are different. Show how to accelerate NAÏVE – STRING-MATCHER to run in time O(n) on an n-character text T.

5.9 Graph Algorithms

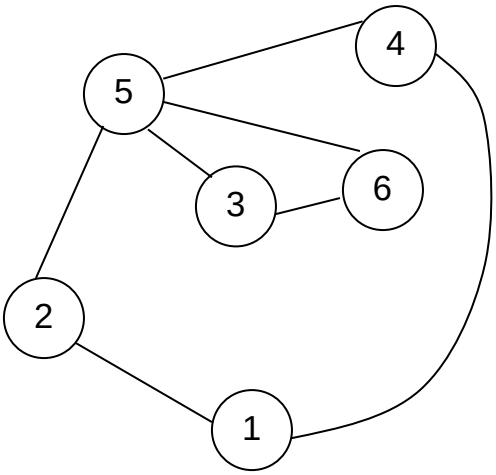
It is notable that several computational problems are expressible in terms of graphs.

- As stated earlier (in examples of algorithm), graphs are fundamental structures used in computer science to represent complex structured information. A graph contains a set of elements, known as vertices, and relationships between pairs of these elements known as edges. The types of graphs that occur commonly in algorithms include; undirected, directed, weighted, and hyper graphs.
- Usually, a graph G is represented as a pair V and E, ie a graph G = (V, E), where V is a finite set of points called vertices and E is a finite set of edges. Actually, there are about two standard ways to represent the graph G: either as a collection of adjacency lists or as an adjacency matrix. Either way applies to directed and undirected graphs.

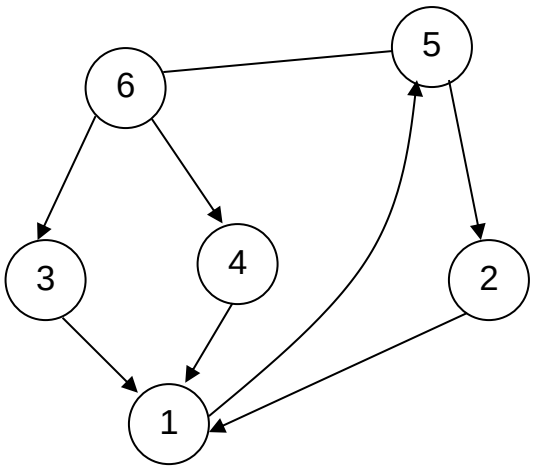
2. Types of Graph

- Bearing in mind that an edge $e \in \Sigma$ is an unordered pair (u, v), where $u, v \in V$: In a directed graph, the edge e is an ordered pair (u, v) where the edge (u, v) is incident from vertex u and is incident to vertex v.

  Recall that a path from a vertex v to a vertex u is a sequence $<v_o, v_1, v_2, \ldots, v_k>$ of vertices where $v_0 = v$, $v_k = u$, and $(v_i, v_{i+1}) \in E$ for i = 0, 1, …, k-1. The length of a path is defined as the number of edges in the path.
- On the other hand, an undirected graph is connected if every pair of vertices is connected by a path.



(a) An undirected Graph

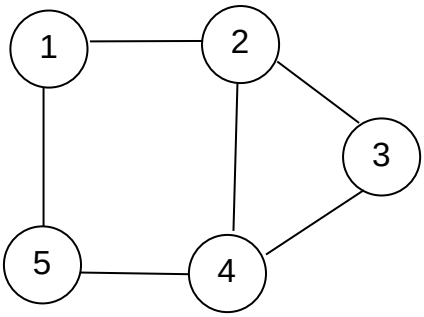


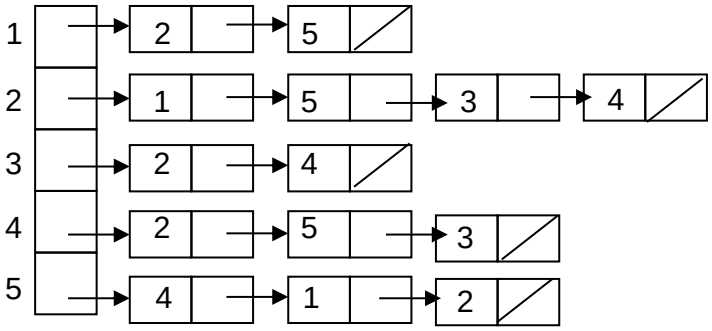
(b) A directed Graph

- Note that a forest can be said to be an acyclic graph, and a tree is a connected acyclic graph. A graph that has weights associated with each edge is called a weighted graph.

3. Representation of Graph

- As mentioned, graphs can be represented by the adjacency matrix or an edge (vertex) list.
- Adjacency matrices have a value $a_{i,j} = 1$ if nodes i and j share an edge; 0 otherwise. In case of a weighted graph, $a_{i,j} = w_{i,j}$, the weight of the edge.
- The adjacency list representation of a graph G = (V, E) consists of an array Adj [1.. $|V|$] of lists. Each list Adj [V] is a list of all vertices adjacent to V in G. Alternatively; it may contain pointers to these vertices.
- We continue with examples of representation of graphs in terms of adjacency lists and adjacency matrices.





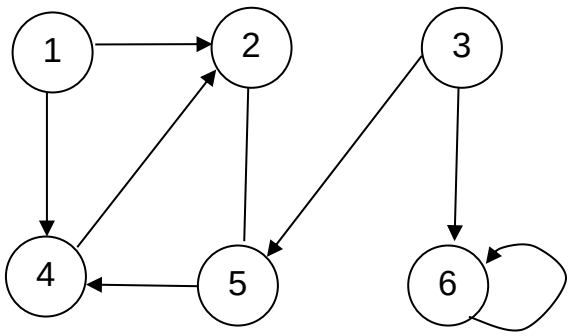
(a) An undirected Graph G=(V,E) => G=(5,7).

(b) An adjacent list representation of G

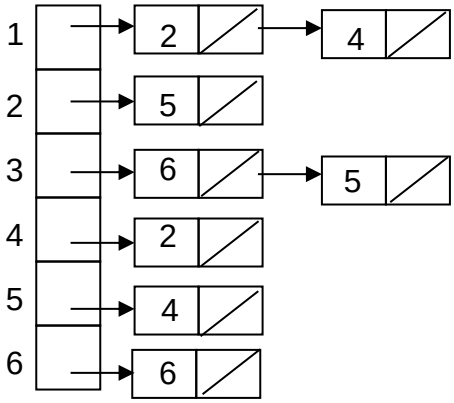
(c) The matrix representation

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

2. Directed Graph



(a) A directed Graph G
G = (V,E) => G = (6,8)



(b) An adjacency list representation of G

(d) The adjacency matrix representation of G

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
5	0	0	0	0	0	1

- We observe that if G is a directed graph, the sum of the lengths of all the adjacency lists is $|E|$, since an edge of the form (u,v) is represented by having V appear in Adj[u]. If G is an undirected graph, the sum of the lengths of all the adjacency lists is $2|E|$, since if [u,v] is an undirected edge, the u appears in v's adjacency list and vice versa. For both directed and undirected graphs, the adjacency list representation has a desirable property that the amount of memory it requires is $\Theta((V+E)$.
- Remember that we can adapt adjacency lists to represent weighted graphs (where each edge are associated with a weight), and given by a function $w:E \rightarrow R$. We

simply store the weight w(u,v) of the edge (u,v) $\in$ E with vertex v in u's adjacency list. The adjacency list representation is quite robust in that we can modify it to support many other graph variants. But the disadvantage of adjacency list representation is that it provides no quicker way to determine whether a given edge (u, v) is present in the graph than to search for v in the adjacency list Adj[u].

- An adjacency matrix representation remedies the disadvantage of the adjacency list representation, but at the cost of using asymptotically more memory. In this case, considering a graph G = (V,E), we assume that the vertices are numbered 1, 2, …, $|V|$ in some arbitrary manner. Then the adjacency matrix representation of a graph G consists of a $|V| \times |V|$ matrix $A = (a_{ij})$ such that

$$Aij = \begin{cases} 1 & if\ (i,j) \varepsilon E \\ 0 & otherwise \end{cases}$$

- Thus, the adjacency matrix of a graph requires $\Theta(V^2)$ memory, independent of the number of edges in the graph.
- An adjacency matrix replacement can also be used to represent a weighted graph. Similarly, we simply store the weight w(u,v) of the edge (u,v) $\in$ E as the entry in row u and column v of the adjacency matrix. If an edge does not exist, we can store a NIL value as its corresponding matrix entry, though for many problems it is convenient to use a value such as 0 or ∞.

Graph Algorithms Examples:

Here we take a look at some of the relevant algorithms related to graphs. These include those for searching graphs, such as Breadth-first search and Depth-first search; and those for solving problems such as the minimum spanning tree and the Dijkstra's algorithm for finding shortest paths (Optimization problems).

5.9.1 Breadth-First Search [Cormen et al, 2009]

- For a given graph G = (V,E) and a distinguished source vertex S, breadth-first search systematically explores the edges of G to "discover" every vertex that is reachable from S. It computes the distance (smallest number of edges) from S to each reachable vertex. It also produces a "bread-first tree" with root S that contains all reachable vertices.
- For any vertex V reachable from S, the simple path in the breadth-first tree from S to V corresponds to a "shortest path" from S to V in G, that is, a path containing the smallest number of edges. The algorithm works for both directed and undirected graphs.
- Breadth-first search, as the name implies, expands the frontier between discovered and undiscovered vertices uniformly across the breadth of the frontier. That is, the algorithm discovers all vertices at distance k from S before discovering any vertices at distance k+1.
- Operationally, therefore, BFS is a strategy for searching in a graph when search is limited to essentially two operations: (a) visit and inspect a node of a graph, (b) gain access to visit the nodes that neighbour the currently visited node.

- In order to fully describe the operation of BFS, Cormen et al used the colour approach/method. In this case, to keep track of progress, BFS colours each vertex white, gray, or black. All vertices start out white and may later become gray and then black. A vertex is "discovered" the first time it is encountered during the search, at which time it becomes non-white.
- In line with the above, gray and black vertices have been discovered, but BFS distinguishes between them to ensure that the search proceeds in breadth-first manner. To this extent, this is such that if $(u,v) \in E$ and vertex u is black, then vertex v is either gray or black: this implies that all vertices adjacent to black vertices have been discovered. Gray vertices may have some adjacent white vertices; they represent the frontier between discovered and undiscovered vertices.
- Proceeding, BFS constructs a breadth-first tree, initially containing only its root, which is the source vertex S. Whenever the search discovers a white vertex v in the course of scanning the adjacency list of an already discovered vertex u, the vertex v and the edge (u,v) are added to the tree. It is therefore said that u is the *predecessor* or *parent* of v in the breadth-first tree. Such a vertex is discovered at most once; it has at most one parent.

  Ancestor and descendant relationships in the breadth-first tree are defined relative to the root s as usual: If u is on the simple path in the tree from the root is to vertex v, then u is an ancestor of v and v is a descendant of u.
- We proceed to present the BFS procedure, which assumes that the input graph G = (V,E) is represented using adjacency lists. It also attaches several additional

attributes to each vertex in the graph, and stores the colour of each vertex u $\in$ v in the attribute u colour and the predecessor of u in the attribute u. π., where u has no predecessor (for example, if u = s or it not been discovered), then u.π = NIL. The attribute u.d holds the distance from the source s to vertex u computed by the algorithm. The algorithm also uses a first-in, first-out queue Q to manage the set of gray vertices:

BFS(G,S)

```
1     for each vertex u ∈ G.V – {s}
2.          u. colour = WHITE
3.          u. d = ∞
4.          u.π = NIL
5.    S.colour = GRAY
6.    S.d = 0
7.    S. π = NIL
8.    Q = ∅
9.    ENQUEUE (Q, S)
10.   while Q ≠ ∅
11.         u = DEQUEUE(Q)
12.   for each v ∈ G. Adj[u]
13.         if v.colour = =  WHITE
14.           v. colour = GRAY
15.           v.d = u.d + 1
16.           v.π = u
17.           ENQUEUE (Q, u)
18.   u.colour = BLACK
```

Explanation/Discussion

The above procedure works as follows:

- With the exception of the source vertex s, lines 1-4 paint every vertex white, set u.d to be infinity for each vertex u, and set the parent of every vertex to be NIL.
- Line 5 paints s gray, since we consider it to be discovered as the procedure begins
- Line 6 initiates s.d. to 0, and line 7 sets the predecessor of the source to be NIL.
- Lines 8 – 9 initialize Q to the queue containing just the vertex s.
- The while loop of lines 10-18 iterates as long as there remain gray vertices, which are discovered vertices that have not yet had the adjacency lists fully examined. The while loop maintains the following invariant:

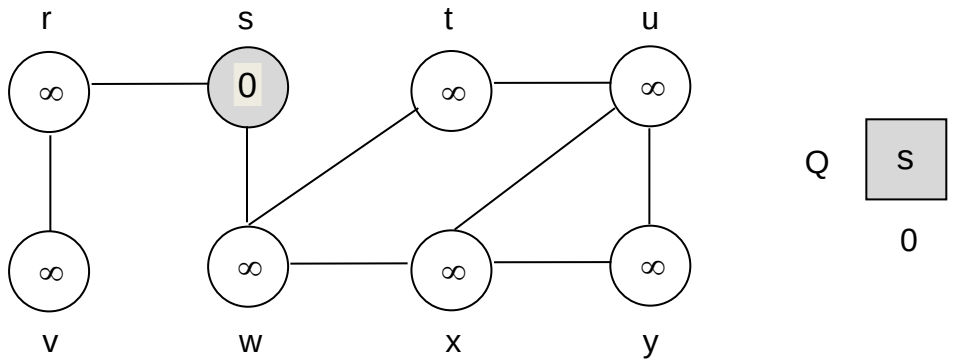
  At the test in line 10, the queue Q consists of the set of gray vertices.

  - Without attempting to use the invariant to prove correctness, it is easy to see that it holds prior to the first iteration and that each iteration of the loop maintains the invariant. Prior to the first iteration, the only gray vertex, and the only vertex in Q, is the source vertex s.
  - Line 11 determines the gray vertex u at the head of the queue Q and removes it from Q.
  - The "for" loop of lines 12 – 17 considers each vertex V in the adjacency list of u. If v is white, then it has not yet been discovered, and the procedure discovers it by executing lines 14 – 17.
  - The procedure paints vertex v gray, sets its distance v.d to u.d + 1, records u as its parent v.π, and places it at the tail of the queue Q.

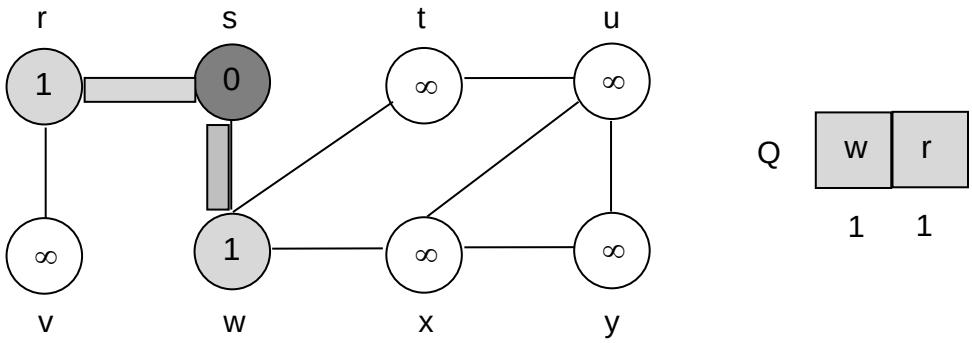
- Once the procedure has examined all the vertices on u's adjacency list, it blackens u in line 18. The loop invariant is maintained because whenever a vertex is painted gray (in line 14) it is also enqueued (in line 17), and whenever a vertex is dequeued (in line 11) it is also painted black (in line 18).
- The results of BFS may depend upon the order in which the neighbours of a given vertex are visited in line 12: The breadth-first tree may vary, but the distances d computed by the algorithm will not.

Illustration of BFS

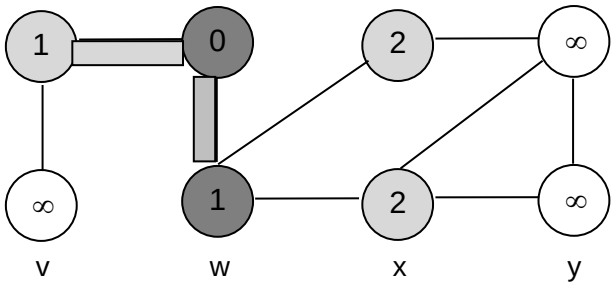
(a)



(b

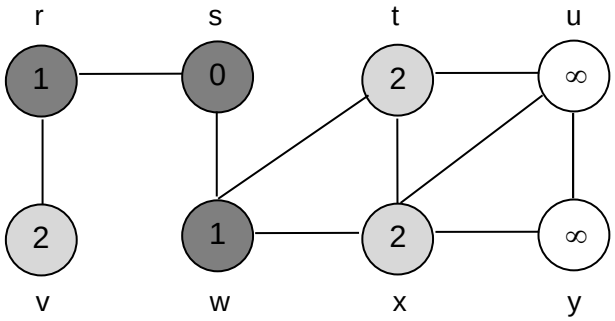


(c)



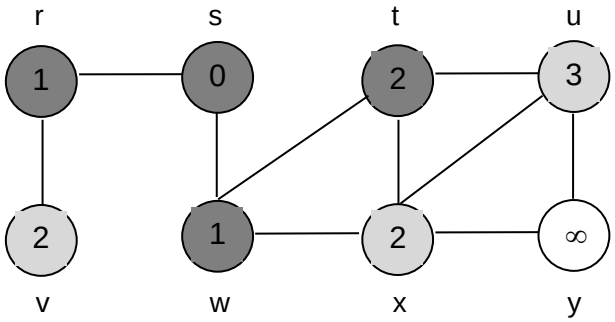
Q | r | t | x |
1 2 2

(d



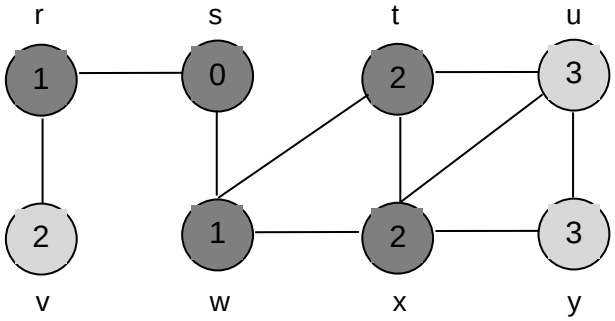
Q | t | x | v |
2 2 2

(e)



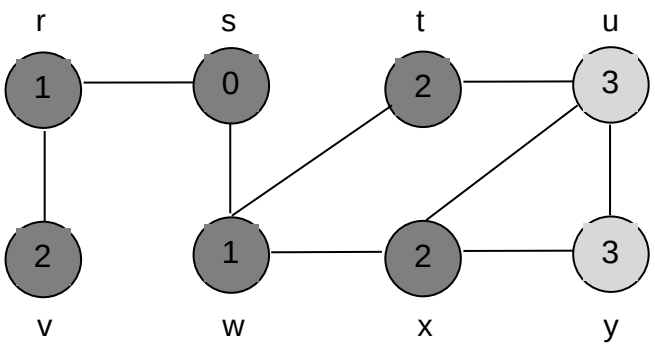
Q | x | v | u |
2 2 3

(f)



Q | v | u | y |
2 3 3

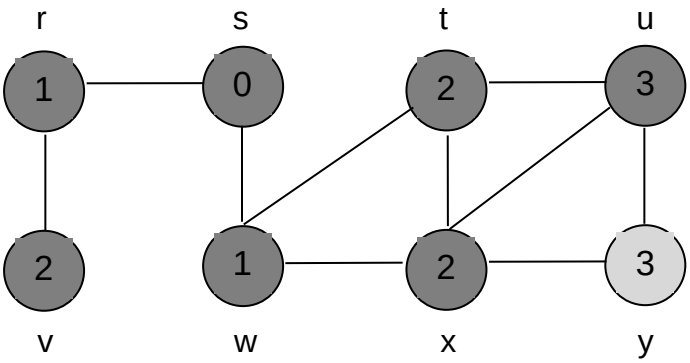
(g



Q u y

3 3

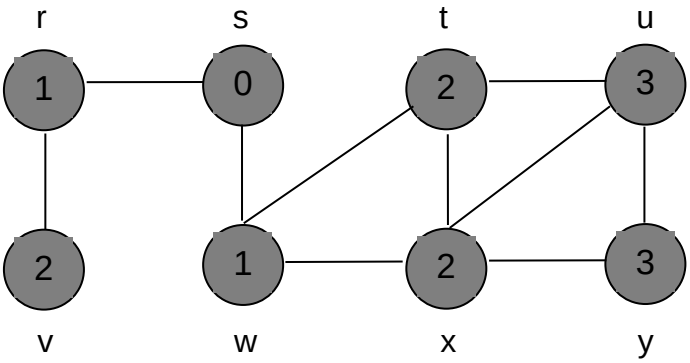
(h



Q y

3

(i



Q ∅

Analysis of BFS

- Both the space and time complexities can be expressed in terms of the asymptotes.
- The time complexity for instance, can be expressed as **O**(|V| + |E|) since every vertex and every edge will be explored in the worst case.

  Note: **O**(|E|) may vary between **O**(|V|) and **O**($|V|^2$) depending on how sparse the input graph is (assuming that the graph is connected).

N/B: BFS can be used for both directed and undirected graphs.

Applications of BFS

- Breadth-first search can be used to solve many problems in graph theory, for example:
- Finding all nodes within one connected component
- Copying collection
- Finding the shortest path between two nodes
- Ford-Fulkerson method for computing the maximum flow in a flow net etc.

N/B: The following are recommended for further work:

- Depth-First search
- Minimum Spanning Tree
- Dijkstra's Algorithm.