NATIONAL OPEN UNIVERSITY OF NIGERIA

FACULTY OF SCIENCES

DEPARTMENT OF COMPUTER SCIENCE

COURSE CODE: CIT310

COURSE TITLE: ALGORITHMS AND COMPLEXITY ANALYSIS

Course Code	CIT 310
Course Title	Algorithms and Complexity Analysis
Course Developer/Writer	Kingsley Chiwuike Ukaoha, *PhD* Department of Computer Science University of Benin Benin City, Nigeria
	Developed/ Written December, 2021.
Programme Leader	
Course Coordinator	

NATIONAL OPEN UNIVERSITY OF NIGERIA

CONTENTS **Page**

Introduction

In writing the Algorithms and Complexity Analysis course, emphasis will be placed on understanding the concept of computer algorithms, how to develop algorithms; test them before translating into viable and workable programs. This course is specifically tailored towards those students who are actually studying computing and interested in developing and testing computer algorithms and in applying them towards developing programs in any programming language.

What you will Learn in this Course

This is a course with theoretical and self-exercises content. Throughout the semester, students will complete 3 modules of 15 units, self-assessment exercises and workable assignments expected to meet specific course criteria.

Course Aims

The aim of the course is to guide learners of Computing and Computer Programs on how to design and test algorithms and also help them in identifying different types of algorithm design paradigms. It is also to help them simplify the task of understanding the theory behind computer algorithms.

Course Objectives

Below are the objectives of the course which are to:

1. Provide sound understanding of computer algorithms.
2. Provide an understanding of algorithm design paradigms.
3. Provide suitable examples of different types of algorithms and why algorithms are very important in computing.

Working through this Course

To complete this course you are required to read each study unit and other materials which may be provided by the National Open University of Nigeria. Each unit contains self-assessment exercises and some units with real-life problems to solve. At the end of every unit, you may be required to submit tutor

marked assignments for assessment and grading. At the end of the course there is a final examination.
To be abreast of this course, you are advised to avail yourself the opportunity of attending the tutorial or online facilitation sessions where you have opportunity of comparing your knowledge with those of your colleagues.

The Course Materials
The main components of the course are:

1.0 The Course Guide
2.0 Study Units
3.0 References/Further Readings
4.0 Assignments
5.0 Presentation Schedule

Study Units
The study units in this course are as follows:

Module 1 Basic Algorithm Analysis
Unit 1 Basic Algorithm Concepts
Unit 2 Analysis and Complexity of Algorithms
Unit 3 Algorithm Design Techniques
Unit 4 Recursion and Recursive Algorithms
Unit 5 Recurrence Relations

Module 2 Searching and Sorting Algorithms
Unit 1 Bubble Sort and Selection Sort Algorithm
Unit 2 Insertion Sort and Linear Search Algorithms
Unit 3 Radix Sort and Stability in Sorting
Unit 4 Divide-and-Conquer Strategies I: Binary Search
Unit 5 Divide-and-Conquer Strategies II: MergeSort and Quicksort Algorithms

Module 3 Other Algorithm Techniques
Unit 1 Binary Search Trees
Unit 2 Dynamic Programming
Unit 3 Computational Complexity
Unit 4 Approximate Algorithms I
Unit 5 Approximate Algorithms II

Presentation Schedule
The course materials assignments have important deadlines for submission. The learners should guide against falling behind stipulated deadlines.

Assessment

There are three ways of carrying out assessment of the course. First assessment is made up of self-assessment exercises, second consists of the tutor marked assignments and the last is the written examination/end of course examination.
You are expected to do all the self-assessment exercises by applying what you have read in the units. The tutor marked assignments should be submitted to your facilitators for formal assessment in accordance with the deadlines stated in the presentation schedule and the assignment files. The total assessment will carry 30% of the total course score. At the end of the course, the final examination will not be more than three hours and will carry 70% of the total marks.

Tutor Marked Assignments

Tutor Marked Assignments (TMA) is a very important component of the course. You are to attempt three TMAs out four given before sitting for final examination. The TMA will be given to you by your facilitator and return same after the assignment is completed. Make sure the TMA reach the facilitator before the deadline given in the presentation schedule.

Final Examination and Grading

The final course examination will not exceed 3 hours which will total 70% of the final score. The examination questions will reflect the self-assessment exercises, tutor marked assignments that are previously encountered in the course units.

Course Marking Scheme

Your grade will be based on assignments and end of course examination. The assignments submitted will be weighted equally.

	Marks
Assignments	**30%**
Examination	**70%**
Total	**100%**

Facilitators/Tutor and Tutorials

There are 16 hours of tutorials provided in support of this course. You will be notified of the dates, times and location of these tutorials as well as the name and phone number of your facilitator, as soon as you are allocated a tutorial group.
You facilitator will mark and comment on your assignments, keep a close watch on your progress and any difficulties you might face and provide assistance to you during the course. You are expected to mail you Tutor Marked Assignments to your facilitator before the schedule date (at least two working days are required). The assignments will be marked by your tutor and returned to you as soon as possible.
Do not delay in contacting your facilitator on telephone or e-mail if you need assistance. Such assistance could be as a result of the followings:

- Having difficulties in understanding any part of the study unit or assigned readings
- Difficulties in the self assessment exercises
- Questions or problems with assignment or grading of assignments

The only way to have face to face contact and to ask questions from your facilitator is to attend tutorials. To gain from the tutorials prepare lists of questions and participate actively in discussions.

Summary

This course is to provide overview of computer algorithms and analysis of its complexity. In particular, we will see know more about the nature and design of algorithms, why they are so important in the field of computing and the several algorithm design paradigms that would be explained.. In fact, the learners will actually learn how do basic run-time and space-complexity analysis of computer algorithms. Some examples of algorithms applied in the fields of Searching and Sorting would also be examined..
I wish you success in the course and I hope you will find the course both interesting and useful.

CIT 310 - Algorithms and Complexity Analysis (3 Units)

Table of Content

CIT 310 - Algorithms and Complexity Analysis

Module 1 Basic Algorithm Analysis

Unit 1 Basic Algorithm Concepts

Unit 2 Analysis of Algorithms

Unit 3 Algorithm Design Techniques

Unit 4 Recursion and Recursive Algorithms

Unit 5 Recurrence Relations

Module 1: Basic Algorithmic Analysis

Unit 1: Basic Algorithm Concepts

1.0 Introduction

The word algorithm which literarily means “a step-by-step procedure used in solving a problem” and is a type of effective method in which a definite list of well-defined instructions for completing a task; that given an initial state, will proceed through a well-defined series of successive states, eventually terminating in an end-state. The concept of an algorithm originated as a means of recording procedures for solving mathematical problems such as finding the common divisor of two numbers or multiplying two numbers.

2.0 Objectives

By the end of this unit, you should be able to:

- Define and describe what an algorithm is
- Enumerate the different characteristics of an algorithm
- Examine some of the advantages of algorithms
- Identify some shortcomings or disadvantages of algorithms
- Look at the the concept of a pseudocode
- Examine some benefits and shortcomings of a pseudocode
- Make a comparison between and algorithm and a pseudocode
- Look at the various reasons why an algorithm is needed

3.0 What is an Algorithm?

An algorithm can be defined as a finite set of steps, which has to be followed while carrying out a particular problem. It is nothing but a process of executing actions step by step.

An algorithm is a distinct computational procedure that takes input as a set of values and results in the output as a set of values by solving the problem. More precisely, an algorithm is correct, if, for each input instance, it gets the correct output and gets terminated.

An algorithm unravels the computational problems to output the desired result. An algorithm can be described by incorporating a natural language such as English, Computer language, or a hardware language.

Algorithms are named for the 9th century Persian mathematician Al-Khowarizmi. He wrote a treatise in Arabic in 825 AD, *On Calculation with Hindu Numerals.* It was translated into Latin in the 12th century as *Algoritmi de numero Indorum*, which title was likely intended to mean "[Book by] Algoritmus on the numbers of the Indians", where "Algoritmi" was the translator's rendition of the author's name in the genitive case; but people misunderstanding the title treated *Algoritmi* as a Latin plural and this led to the word "algorithm" (Latin *algorismus*) coming to mean "calculation method".

3.1 Characteristics of Algorithms

The main Characteristics or features of Algorithms are;

- **Input:** It should externally supply zero or more quantities.
- **Output:** It results in at least one quantity.
- **Definiteness:** Each instruction should be clear and ambiguous.
- **Finiteness:** An algorithm should terminate after executing a finite number of steps.
- **Effectiveness:** Every instruction should be fundamental to be carried out, in principle, by a person using only pen and paper.
- **Feasible:** It must be feasible enough to produce each instruction.
- **Flexibility:** It must be flexible enough to carry out desired changes with no efforts.
- **Efficient:** The term efficiency is measured in terms of time and space required by an algorithm to implement. Thus, an algorithm must ensure that it takes little time and less memory space meeting the acceptable limit of development time.
- **Independent:** An algorithm must be language independent, which means that it should mainly focus on the input and the procedure required to derive the output instead of depending upon the language.

3.1.1 Advantages of Algorithms

- **Effective Communication:** Since it is written in a natural language like English, it becomes easy to understand the step-by-step delineation of a solution to any particular problem.
- **Easy Debugging:** A well-designed algorithm facilitates easy debugging to detect the logical errors that occurred inside the program.

- **Easy and Efficient Coding:** An algorithm is nothing but a blueprint of a program that helps develop a program.
- **Independent of Programming Language:** Since it is a language-independent, it can be easily coded by incorporating any high-level language.

3.1.2 Disadvantages of Algorithms

- Developing algorithms for complex problems would be time-consuming and difficult to understand.
- It is a challenging task to understand complex logic through algorithms.

3.2 Pseudocode

Pseudocode refers to an informal high-level description of the operating principle of a computer program or algorithm. It uses structural conventions of a standard programming language intended for human reading rather than the machine reading.

3.2.1 Advantages of Pseudocode

- Since it is similar to a programming language, it can be quickly transformed into the actual programming language than a flowchart.
- The layman can easily understand it.
- Easily modifiable as compared to the flowcharts.
- Its implementation is beneficial for structured, designed elements.
- It can easily detect an error before transforming it into a code.

3.2.2 Disadvantages of Pseudocode

- Since it does not incorporate any standardized style or format, it can vary from one company to another.
- Error possibility is higher while transforming into a code.
- It may require a tool for extracting out the Pseudocode and facilitate drawing flowcharts.

- It does not depict the design.

3.2.3 Difference between Algorithm and Pseudocode

i. An algorithm is simply a problem-solving process, which is used not only in computer science to write a program but also in our day to day life. It is nothing but a series of instructions to solve a problem or get to the problem's solution. It not only helps in simplifying the problem but also to have a better understanding of it.

ii. However, Pseudocode is a way of writing an algorithm. Programmers can use informal, simple language to write pseudocode without following any strict syntax. It encompasses semi-mathematical statements.

3.2.4 Problem Case/ Example:

Suppose there are 60 students in a class. How will you calculate the number of absentees in the class?

i. Pseudocode Approach:

```
1. Initialize a variable called as Count to zero, absent to
   zero, total to 60
2. FOR EACH Student PRESENT DO the following:
   Increase the Count by One
3. Then Subtract Count from total and store the result in absent
4. Display the number of absent students
```

ii. Algorithmic Approach:

```
1. Count <- 0, absent <- 0, total <- 60
2. REPEAT till all students counted
   Count <- Count + 1
3. absent <- total - Count
4. Print "Number absent is:" , absent
```

3.3 Need of Algorithms (Why do we need Algorithms?)

1. To understand the basic idea of the problem.
2. To find an approach to solve the problem.
3. To improve the efficiency of existing techniques.
4. To understand the basic principles of designing the algorithms.
5. To compare the performance of the algorithm with respect to other techniques.
6. It is the best method of description without describing the implementation detail.
7. The Algorithm gives a clear description of requirements and goal of the problem to the designer.
8. A good design can produce a good solution.
9. To understand the flow of the problem.
10. To measure the behavior (or performance) of the methods in all cases (best cases, worst cases, average cases)
11. With the help of an algorithm, we can also identify the resources (memory, input-output) cycles required by the algorithm.
12. With the help of algorithm, we convert art into a science.
13. To understand the principle of designing.
14. We can measure and analyze the complexity (time and space) of the problems concerning input size without implementing and running it; it will reduce the cost of design.

Self-Assessment Exercise

1. What is an algorithm?
2. Differentiate between an algorithm and a pseudocode

3. Highlight some of the basic reasons why algorithms are needed?
4. How is an algorithm similar to and different from a program?
5. Why must every good computer programmer understand an algorithm first?
6. State an algorithm for adding three numbers A, B, and C

4.0 Conclusion

The concept of understanding and writing computer algorithms is very essential to understanding the task of programming and every computing student has to imbibe the concepts of algorithms. In fact, algorithms are the basic key to understanding the theory and practice of computing.

5.0 Summary

In this unit we have learnt an overview of algorithms and their basic characteristics. In addition, we looked at some of the benefits and shortcomings of algorithms and also examined the concept of a pseudocode as well as some of its benefits and shortcomings. We also made a brief comparison between a pseudocode and an algorithm and finally looked at some of the reasons why an algorithm is needed

6.0 Tutor Marked Assignment

1. Explain the following terms; (a) Algorithms (b) Pseudocode
(c) Computer Programs
2. State five properties or features of an algorithm.
3. State some basic differences between and algorithm and a pseudocode and also between an algorithm and a computer program
4. give four benefits each of an algorithm and a pseudocode

7.0 Further Reading and other Resources

Jena, S. R. and Patro, S. (2018) – Design and Analysis of Algorithms, ISBN 978-93-935274-311-7

Baase, S. and Van Gelder, A. (2008). *Computer Algorithms: Introduction to Design and Analysis*, Pearson Education.

Module 1: Basic Algorithm Analysis

Unit 2: Analysis and Complexity of Algorithms

1.0 Introduction

The analysis is a process of estimating the efficiency of an algorithm. There are two fundamental parameters based on which we can analyze the algorithm and they are Space and Time Complexity. There is also the concept in Time Complexity of estimating the running time of an algorithm and we have the Best-case, Average-case and Worst-case

2.0 Objectives

By the end of this unit, you will be able to

- Understand runtime and space analysis or complexity of algorithms
- Know the different types of analysis
- Understand the typical complexities of an algorithm
- Learn how to approximate the time taken by an algorithm

3.0 Analysis of Algorithm

The analysis is a process of estimating the efficiency of an algorithm and that is, trying to know how good or how bad an algorithm could be. There are two main parameters based on which we can analyze the algorithm:

- **Space Complexity:** The space complexity can be understood as the amount of space required by an algorithm to run to completion.
- **Time Complexity:** Time complexity is a function of input size **n** that refers to the amount of time needed by an algorithm to run to completion.

Let's understand it with an example.

Suppose there is a problem to solve in Computer Science, and in general, we solve a program by writing a program. If you want to write a program in some programming language like C, then before writing a program, it is necessary to write a blueprint in an informal language.

Or in other words, you should describe what you want to include in your code in an English-like language for it to be more readable and understandable before implementing it, which is nothing but the concept of Algorithm.

In general, if there is a problem **P1**, then it may have many solutions, such that each of these solutions is regarded as an algorithm. So, there may be many algorithms such as $\mathbf{A_1, A_2, A_3}$, …, $\mathbf{A_n}$.

Before you implement any algorithm as a program, it is better to find out which among these algorithms are good in terms of time and memory.

It would be best to analyze every algorithm in terms of **Time** that relates to which one could execute faster and **Memory or Space** corresponding to which one will take less memory.

So, the Design and Analysis of Algorithm talks about how to design various algorithms and how to analyze them. After designing and analyzing, choose the best algorithm that takes the least time and the least memory and then implement it as a program in C.

We will be looking more on time rather than space because time is instead a more limiting parameter in terms of the hardware. It is not easy to take a computer and change its speed. So, if we are running an algorithm on a particular platform, we are more or less stuck with the performance that platform can give us in terms of speed.

However, on the other hand, memory is relatively more flexible. We can increase the memory as when required by simply adding a memory card. So, we will focus on time than that of the space.

The running time is measured in terms of a particular piece of hardware, not a robust measure. When we run the same algorithm on a different computer which might be faster or use different programming languages which may be designed to compile faster, we will find out that the same algorithm takes a different time.

3.1 Types of Time Complexity Analysis

We have three types of analysis related to time complexity, which are:

3.1.1 Worst-case time complexity: For '**n**' input size, the worst-case time complexity can be defined as the maximum amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function

defined by the maximum number of steps performed on an instance having an input size of n.

3.1.2 Average case time complexity: For '**n**' input size, the average-case time complexity can be defined as the average amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the average number of steps performed on an instance having an input size of n.

3.1.3 Best case time complexity: For '**n**' input size, the best-case time complexity can be defined as the minimum amount of time needed by an algorithm to complete its execution. Thus, it is nothing but a function defined by the minimum number of steps performed on an instance having an input size of n.

3.2 Complexity of Algorithms

The term algorithm complexity measures how many steps are required by the algorithm to solve the given problem. It evaluates the order of count of operations executed by an algorithm as a function of input data size.

To assess the complexity, the order (approximation) of the count of operation is always considered instead of counting the exact steps.

O(f) notation represents the complexity of an algorithm, which is also termed as an Asymptotic notation or "**Big O**" notation. Here the f corresponds to the function whose size is the same as that of the input data. The complexity of the asymptotic computation O(f) determines in which order the resources such as CPU time, memory, etc. are consumed by the algorithm that is articulated as a function of the size of the input data.

The complexity can be found in any form such as constant, logarithmic, linear, n*log(n), quadratic, cubic, exponential, etc. It is nothing but the order of constant, logarithmic, linear and so on, the number of steps encountered for the completion of a particular algorithm. To make it even more precise, we often call the complexity of an algorithm as "running time".

3.3 Typical Complexities of an Algorithm

We examine the different types of complexities of an algorithm and one or more of our algorithm or program will fall into any of the following categories;

3.3.1 Constant Complexity:

It imposes a complexity of **O(1)**. It undergoes an execution of a constant number of steps like 1, 5, 10, etc. for solving a given problem. The count of operations is independent of the input data size.

3.3.2 Logarithmic Complexity:

It imposes a complexity of **O(log(N))**. It undergoes the execution of the order of log(N) steps. To perform operations on N elements, it often takes the logarithmic base as 2.

For N = 1,000,000, an algorithm that has a complexity of O(log(N)) would undergo 20 steps (with a constant precision). Here, the logarithmic base does not hold a necessary consequence for the operation count order, so it is usually omitted.

3.3.3 Linear Complexity:

It imposes a complexity of **O(N)**. It encompasses the same number of steps as that of the total number of elements to implement an operation on N elements. For example, if there exist 500 elements, then it will take about 500 steps. Basically, in linear complexity, the number of elements linearly depends on the number of steps. For example, the number of steps for N elements can be N/2 or 3*N.

It also imposes a run time of **O(n*log(n))**. It undergoes the execution of the order N*log(N) on N number of elements to solve the given problem. For a given 1000 elements, the linear complexity will execute 10,000 steps for solving a given problem.

3.3.4 Quadratic Complexity:

It imposes a complexity of **$O(n^2)$**. For N input data size, it undergoes the order of N^2 count of operations on N number of elements for solving a given problem.
If N = 100, it will endure 10,000 steps. In other words, whenever the order of operation tends to have a quadratic relation with the input data size, it results in quadratic complexity.
For example, for N number of elements, the steps are found to be in the order of $3*N^2/2$.

3.3.5 Cubic Complexity:

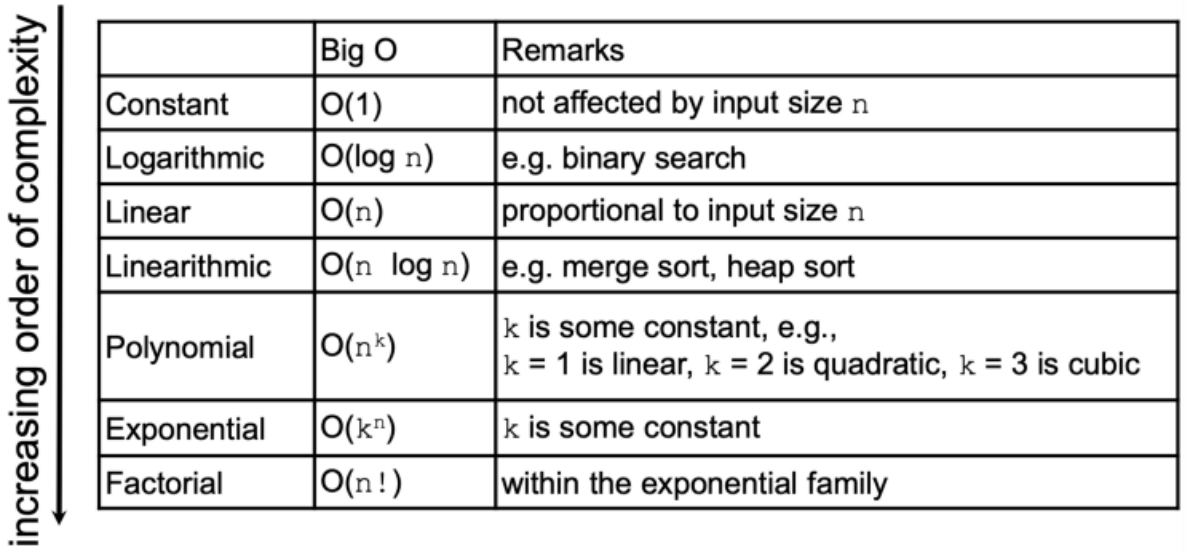
It imposes a complexity of **$O(n^3)$**. For N input data size, it executes the order of N^3 steps on N elements to solve a given problem.
For example, if there exist 100 elements, it is going to execute 1,000,000 steps.

3.3.6 Exponential Complexity:

It imposes a complexity of **$O(2^n)$, $O(N!)$, $O(n^k)$, ….** For N elements, it will execute the order of count of operations that is exponentially dependable on the input data size.
For example, if N = 10, then the exponential function 2^N will result in 1024. Similarly, if N = 20, it will result in 1048 576, and if N = 100, it will result in a number having 30 digits.
The exponential function N! grows even faster; for example, if N = 5 will result in 120. Likewise, if N = 10, it will result in 3,628,800 and so on.

Since the constants do not hold a significant effect on the order of count of operation, so it is better to ignore them.
Thus, to consider an algorithm to be linear and equally efficient, it must undergo N, N/2 or 3*N count of operation, respectively, on the same number of elements to solve a particular problem

A summary of these complexities is given below:



	Big O	Remarks
Constant	O(1)	not affected by input size n
Logarithmic	O(log n)	e.g. binary search
Linear	O(n)	proportional to input size n
Linearithmic	O(n log n)	e.g. merge sort, heap sort
Polynomial	$O(n^k)$	k is some constant, e.g., k = 1 is linear, k = 2 is quadratic, k = 3 is cubic
Exponential	$O(k^n)$	k is some constant
Factorial	O(n!)	within the exponential family

Self Assessment Exercises

1. Compare the Worst-case and the Best-case analysis of an algorithm
2. Why is the Worst-case analysis the most important in algorithm analysis?
3. Among the different complexity types of an algorithm, which do you consider as the worst?
4. Presently we can solve problem instances of size 30 in 1 minute using algorithm A, which is a $\Theta(2n)$ algorithm. On the other hand, we will soon have to solve problem instances twice this large in 1 minute. Do you think it would help to buy a faster (and more expensive) computer?

3.4 How to approximate the time taken by the Algorithm?

So, to find it out, we shall first understand the types of the algorithm we have. There are two types of algorithms:

1. **Iterative Algorithm:** In the iterative approach, the function repeatedly runs until the condition is met or it fails. It involves the looping construct.
2. **Recursive Algorithm:** In the recursive approach, the function calls itself until the condition is met. It integrates the branching structure.

However, it is worth noting that any program that is written in iteration could be written as recursion. Likewise, a recursive program can be converted to iteration, making both of these algorithms equivalent to each other.

But to analyze the iterative program, we have to count the number of times the loop is going to execute, whereas in the recursive program, we use recursive equations, i.e., we write a function of F(n) in terms of F(n/2).

Suppose the program is neither iterative nor recursive. In that case, it can be concluded that there is no dependency of the running time on the input data size, i.e., whatever is the input size, the running time is going to be a constant value. Thus, for such programs, the complexity will be **O(1)**.

3.4.1 Some Examples to Consider

a. For Iterative Programs

Consider the following programs written in simple English and does not correspond to any syntax.

Example1:

In the first example, we have an integer **i** and a **for** loop running from **i** equals **1** to **n**. Now the question arises, how many times does the name get printed?

```
A()
{
int i;
for (i=1 to n)
printf("Abdullahi");
}
```

Since **i** equals **1** to **n**, so the above program will print Abdullahi, **n** number of times. Thus, the complexity will be **O(n)**.

Example2:

```
A()
{
int i, j:
for (i=1 to n)
for (j=1 to n)
printf("Abdullahi");
}
```

In this case, firstly, the outer loop will run **n** times, such that for each time, the inner loop will also run **n** times. Thus, the time complexity will be **$O(n^2)$**.

Example3:

```
A()
{
i = 1; S = 1;
while (S<=n)
{
i++;
SS = S + i;
printf("Abdullahi");
}
}
```

As we can see from the above example, we have two variables; **i**, **S** and then we have while **S<=n**, which means **S** will start at **1**, and the entire loop will stop whenever **S** value reaches a point where **S** becomes greater than **n**.

Here **i** is incrementing in steps of one, and **S** will increment by the value of **i**, i.e., the increment in **i** is linear. However, the increment in **S** depends on the **i**.

Initially;

```
i=1, S=1
```

After 1st iteration;

```
i=2, S=3
```

After 2nd iteration;

```
i=3, S=6
```

After 3rd iteration;

```
i=4, S=10
```

… and so on.

Since we don't know the value of n, so let's suppose it to be k. Now, if we notice the value of S in the above case is increasing; for i=1, S=1; i=2, S=3; i=3, S=6; i=4, S=10; …

Thus, it is nothing but a series of the sum of first n natural numbers, i.e., by the time i reaches k, the value of S will be $\frac{k\,(k+1)}{2}$.

To stop the loop, $\frac{k\,(k+1)}{2}$ has to be greater than n, and when we solve this equation,

we will get $\frac{k^2+k}{2}$ **> n**.

Hence, it can be concluded that we get a complexity of **O(√n)** in this case.

b. For Recursive Program

Consider the following recursive programs.

Example1:

```
A(n)
{
if (n>1)
return (A(n-1))
}
```

Solution;

Here we will see the simple Back Substitution method to solve the above problem.

T(n) = 1 + T(n-1) … Eqn. (1)

Step1: Substitute n-1 at the place of n in Eqn. (1)

T(n-1) = 1 + T(n-2) .. .Eqn. (2)

Step2: Substitute n-2 at the place of n in Eqn. (1)

T(n-2) = 1 + T(n-3) … Eqn. (3)

Step3: Substitute Eqn. (2) in Eqn. (1)

T(n)= 1 + 1+ T(n-2) = 2 + T(n-2) … Eqn. (4)

Step4: Substitute eqn. (3) in Eqn. (4)

T(n) = 2 + 1 + T(n-3) = 3 + T(n-3) = ….. = k + T(n-k) …Eqn. (5)

Now, according to Eqn. (1), i.e. T(n) = 1 + T(n-1), the algorithm will run until n>1. Basically, n will start from a very large number, and it will decrease gradually. So, when T(n) = 1, the algorithm eventually stops, and such a terminating condition is called anchor condition, base condition or stopping condition.

Thus, for k = n-1, the T(n) will become.

Step5: Substitute k = n-1 in eqn. (5)

```
T(n) = (n-1) + T(n-(n-1)) = (n-1) + T(1) = n-1+1
```

Hence, `T(n) = n or O(n).`

4.0 Conclusion

Analysis of algorithms helps us to determine how good or how bad they are in terms of speed or time taken and memory or space utilized. Designing good programs is dependent on how good or how bad the algorithm is and the analysis helps us to determine the efficiency of such algorithms.

5.0 Summary

In the unit, we have learnt the meaning of algorithm analysis and the different types of analysis. We also examined the complexity of algorithms and the different types of complexities.

6.0 Tutor Marked Assignment

1. Between the Worst-case and the Best-case analysis, which is more important to a computer programmer and why?.
2. Why must we avoid exponential complexity at all costs?
3. What do we gain by the analysis of algorithms?
4. Suppose you have a computer that requires 1 minute to solve problem instances of size n = 1,000. Suppose you buy a new computer that runs 1,000 times faster than the old one. What instance sizes can be run in 1 minute, assuming the following time complexities $T(n)$ for our algorithm?

(a) $T(n) = n$ (b) $T(n) = n^3$ (c) $T(n) = 10^n$

7.0 Further Reading and other Resources

Berman, K.A. and Paul, J.L.(2005). *Algorithms: Sequential, Parallel, and Distributed*. Course Technology.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C. (2009). *Introduction to Algorithms*, 3rd ed. MIT Press.

Jena, S. R. and Patro, S. (2018) – Design and Analysis of Algorithms, ISBN 978-93-935274-311-7

Module 1: Basic Algorithm Analysis

Unit 3: Algorithm Design Techniques

1.0 Introduction

The design of any algorithm follows some planning as there are different design techniques, strategies or paradigms that could be adopted depending on the problem domain and a better understanding by the designer. Some of these techniques could be combined also while the limiting behaviour of the algorithm can be represented with asymptotic analysis of which we shall be looking at examples of algorithm design techniques and asymptotic notations.

2.0 Objectives

By the end of this unit, you will be able to

- Understand several design techniques or paradigms of algorithms
- Know the meaning of Asymptotic notations
- Understand some popular Asymptotic notations
- Learn how to apply some of the Asymptotic notations learnt

3.0 Algorithm Design Techniques

An algorithm design technique (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing. Learning these techniques is of utmost importance for the following reasons.

- First, they provide guidance for designing algorithms for new problems, i.e., problems for which there is no known satisfactory algorithm.
- Second, algorithms are the cornerstone of computer science. Every science is interested in classifying its principal subject, and computer science is no exception. Algorithm design techniques make it possible to classify algorithms according to an underlying design idea; therefore, they can serve as a natural way to both categorize and study algorithms.

While the algorithm design techniques do provide a powerful set of general approaches to algorithmic problem solving, designing an algorithm for a particular problem may still be a challenging task. Some design techniques can be simply inapplicable to the problem in question. Sometimes, several techniques need to be combined, and there are algorithms that are hard to pinpoint as applications of the known design techniques.

3.1 Popular Algorithm Design Techniques

The following is a list of several popular design approaches:

3.1.1 Divide and Conquer Approach:

The divide-and-conquer paradigm often helps in the discovery of efficient algorithms. It is a top-down approach. The algorithms which follow the divide & conquer techniques involve three steps:

- Divide the original problem into a set of subproblems.
- Solve every subproblem individually, recursively.
- Combine the solution of the subproblems (top level) into a solution of the whole original problem.

Following are some standard algorithms that are of the Divide and Conquer algorithms variety.

- Binary Search is a searching algorithm. ...
- Quicksort is a sorting algorithm. ...
- Merge Sort is also a sorting algorithm. ...
- Closest Pair of Points The problem is to find the closest pair of points in a set of points in x-y plane.

3.1.2. Greedy Technique:

Greedy method or technique is **an algorithmic paradigm that builds up a solution piece by piece**, always choosing the next piece that offers the most obvious and immediate benefit. So the problems where choosing locally optimal also leads to global solution are best fit for Greedy. The Greedy method is used to solve the optimization problem. An optimization problem is one in which we are given a set of input values, which are required either to be maximized or minimized (known as objective), i.e. some constraints or conditions.

- Greedy Algorithm always makes the choice (greedy criteria) looks best at the moment, to optimize a given objective.

- The greedy algorithm doesn't always guarantee the optimal solution however it generally produces a solution that is very close in value to the optimal.

Examples of Greedy Algorithms

- Prim's Minimal Spanning Tree Algorithm.
- Travelling Salesman Problem.
- Graph – Map Coloring.
- Kruskal's Minimal Spanning Tree Algorithm.
- Dijkstra's Minimal Spanning Tree Algorithm.
- Graph – Vertex Cover.
- Knapsack Problem.
- Job Scheduling Problem.

3.1.3. Dynamic Programming:

Dynamic Programming (DP) is **an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems** and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.

Dynamic programming is both a mathematical optimization method and a computer programming method. The method was developed by Richard Bellman in the 1950s and has found applications in numerous fields, from aerospace engineering to economics

Dynamic programming is used **where we have problems**, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems.

Some examples of Dynamic Programming are;

- Tower of Hanoi
- Dijkstra Shortest Path
- Fibonacci sequence

- Matrix chain multiplication
- Egg-dropping puzzle, etc

3.1.4 Branch and Bound:

The branch and bound method is a **solution approach that parti- tions the feasible solution space into smaller subsets of solutions**. , can assume any integer value greater than or equal to zero is what gives this model its designation as a total integer model.

It is used for solving the optimization problems and minimization problems. If we have given a maximization problem then we can convert it using the Branch and bound technique by simply converting the problem into a maximization problem.

An important advantage of branch-and-bound algorithms is that **we can control the quality of the solution to be expected**, even if it is not yet found. The cost of an optimal solution is only up to smaller than the cost of the best computed one.

Branch and bound is an algorithm design paradigm which is generally used for solving **combinatorial optimization problems**.

Some examples of Branch-and-Bound Problems are:

- Knapsack problems
- Traveling Salesman Problem
- Job Assignment Problem, etc

3.1.5. Backtracking Algorithm:

A backtracking algorithm is a **problem-solving algorithm** that uses a brute force approach for finding the desired output. The Brute force approach tries out all the possible solutions and chooses the desired/best solutions.

Backtracking is a general algorithm **for finding solutions to some computational problems, notably constraint satisfaction problems**, that incrementally builds candidates to the solutions, and abandons a candidate ("backtracks") as soon as it determines that the candidate cannot possibly be completed to a valid solution.

The algorithm works as follows:

Given a problem:

```
Backtrack(s)
    if is not a solution
        return false
    if is a new solution
        add to list of solutions
    backtrack(expand s)
```

It finds a solution by building a solution step by step, increasing levels over time, using recursive calling. A search tree known as the state-space tree is used to find these solutions. Each branch in a state-space tree represents a variable, and each level represents a solution.

A backtracking algorithm uses the depth-first search method. When the algorithm begins to explore the solutions, the abounding function is applied so that the algorithm can determine whether the proposed solution satisfies the constraints. If it does, it will keep looking. If it does not, the branch is removed, and the algorithm returns to the previous level.

In any backtracking algorithm, the algorithm seeks a path to a feasible solution that includes some intermediate checkpoints. If the checkpoints do not lead to a viable solution, the problem can return to the checkpoints and take another path to find a solution

There are the following scenarios in which you can use the backtracking:

- It is used to solve a variety of problems. You can use it, for example, to find a feasible solution to a decision problem.
- Backtracking algorithms were also discovered to be very effective for solving optimization problems.
- In some cases, it is used to find all feasible solutions to the enumeration problem.
- Backtracking, on the other hand, is not regarded as an optimal problem-solving technique. It is useful when the solution to a problem does not have a time limit.

Backtracking algorithms are used in;

- Finding all Hamiltonian paths present in a graph
- Solving the N-Queen problem
- Knights Tour problem, etc

3.1.6. Randomized Algorithm:

A randomized algorithm is **an algorithm that employs a degree of randomness as part of its logic or procedure**. ... In some cases, probabilistic algorithms are the only practical means of solving a problem.

The output of a randomized algorithm on a given input is **a random variable**. Thus, there may be a positive probability that the outcome is incorrect. As long as the probability of error is small for every possible input to the algorithm, this is not a problem.

There are two main types of randomized algorithms: **Las Vegas algorithms and Monte-Carlo algorithms**.

Example 1: In Quick Sort, using a random number to choose a pivot.

Example 2: Trying to factor a large number by choosing a random number as possible divisors.

Self-Assessment Exercise

1. What do you understand by an Algorithm design paradigm?
2. How does the Greedy Technique work and give an example?
3, Give a difference between the Backtracking and Randomized algorithm techniques

3.2 Asymptotic Analysis of algorithms (Growth of function)

Resources for an algorithm are usually expressed as a function regarding input. Often this function is messy and complicated to work. To study Function growth efficiently, we reduce the function down to the important part.

Let f (n) = an^2+bn+c

In this function, the n^2 term dominates the function that is when n gets sufficiently large.

Dominate terms are what we are interested in reducing a function, in this; we ignore all constants and coefficient and look at the highest order term concerning n.

3.2.1 Asymptotic analysis

It is a technique of representing limiting behavior. The methodology has the applications across science. It can be used to analyze the performance of an algorithm for some large data set.

In computer science in the analysis of algorithms, considering the performance of algorithms when applied to very large input datasets

The simplest example is a function $f(n) = n^2+3n$, the term 3n becomes insignificant compared to n^2 when n is very large. The function "$f(n)$ is said to be **asymptotically equivalent** to n^2 as $n \to \infty$", and here is written symbolically as

$$f(n) \sim n^2.$$

Asymptotic notations are used to write fastest and slowest possible running time for an algorithm. These are also referred to as 'best case' and 'worst case' scenarios respectively.

"In asymptotic notations, we derive the complexity concerning the size of the input. (Example in terms of n)"

"These notations are important because without expanding the cost of running the algorithm, we can estimate the complexity of the algorithms."

3.2.2 Why is Asymptotic Notation Important?

1. They give simple characteristics of an algorithm's efficiency.

2. They allow the comparisons of the performances of various algorithms.

3.3 Asymptotic Notations:

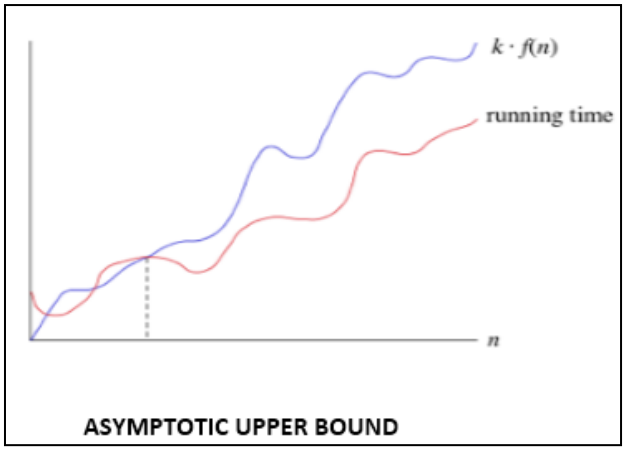
Asymptotic Notation is a way of comparing function that ignores constant factors and small input sizes. Three notations are used to calculate the running time complexity of an algorithm:

3.3.1. Big-oh notation:

Big-oh is the formal method of expressing the upper bound of an algorithm's running time. It is the measure of the longest amount of time. The function **f (n) = O (g (n))** [read as "f of n is big-oh of g of n"] if and only if exist positive constant c and such that

$$f(n) \leqslant k.g(n) \text{ for } n>n0 \text{ in all case}$$

Hence, function g (n) is an upper bound for function f (n), as g (n) grows faster than f (n)



Examples:

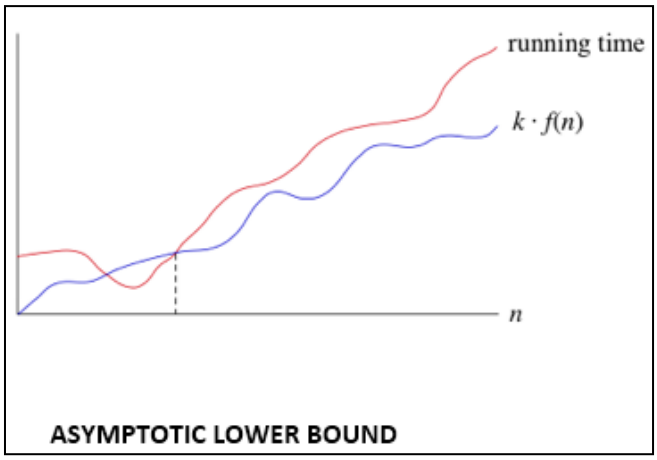
1. $3n+2=O(n)$ as $3n+2 \leq 4n$ for all $n \geq 2$
2. $3n+3=O(n)$ as $3n+3 \leq 4n$ for all $n \geq 3$

Hence, the complexity of **f(n)** can be represented as O (g (n))

3.3.2. Big Omega () Notation:

The function f (n) = Ω (g (n)) [read as "f of n is omega of g of n"] if and only if there exists positive constant c and n_0 such that

$F(n) \geq k^* g(n)$ for all $n, n \geq n_0$



Example:

$f(n) = 8n^2+2n-3 \geq 8n^2-3$

$= 7n^2+(n^2-3) \geq 7n^2$ (g(n))

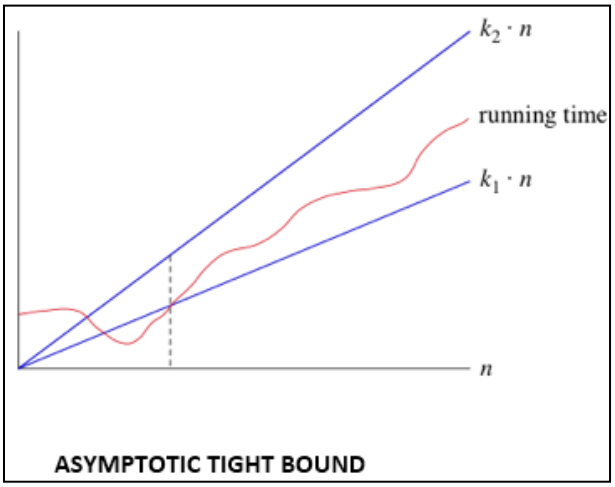
Thus, $k_1=7$

Hence, the complexity of **f (n)** can be represented as **Ω** (g (n))

3.3.3. Big Theta (θ):

The function f (n) = θ (g (n)) [read as "f is the theta of g of n"] if and only if there exists positive constant k_1, k_2 and k_0 such that

$k_1 * g(n) \leq f(n) \leq k_2 g(n)$ for all n, $n \geq n_0$



For Example:

$3n+2 = \theta(n)$ as $3n+2 \geq 3n$ and $3n+2 \leq 4n$, for n
$k_1=3, k_2=4$, and $n_0=2$

Hence, the complexity of f (n) can be represented as θ (g(n)).

The Theta Notation is more precise than both the big-oh and Omega notation. The function f (n) = θ (g (n)) if g(n) is both an upper and lower bound.

Self-Assessment Exercise

1. Which of the Asymptotic notations do you consider more important and why?
2. What do you understand by a Backtracking algorithm?
3. What do you understand by the Upper and Lower bound of an algorithm?

4.0 Conclusion

Algorithm design techniques presents us with different paradigms or methods of representing or designing computer algorithms and as the algorithm executes and grows in bounds (upper or lower), the Asymptotic notations helps us to determine the levels of growth.

5.0 Summary

Several design techniques or paradigms are available for specifying algorithms and they range from the popular Divide-and-Conquer, Greedy techniques and Randomized algorithms amongst others. In the same vein, we have three main notations for carrying out the Asymptotic analysis of algorithms and they are the Big O, Big Omega and Big Theta notations.

6.0 Tutor Marked Assignment

1. Give two examples each of functions that can be represented as
   a. O(f(n)) b. Θ(f(n)) c. Ω(f(n))
2. Mention two types of problems each that can be solved with
   a. Dynamic Programming b. Divide-and-Conquer technique
3. Why is Asymptotic notation considered important?
4. The function $f(x) = n + n2 + 2n + n4$ belongs in which of the following complexity categories: (a) $\theta(n)$ (b) $\theta(n2)$ (c) $\theta(n3)$ (d) $\theta(n \lg n)$ (e) $\theta(n4)$ (f) None of these

7.0 Further Reading and other Resources

Dave, P. H. and Dave, H. B. (2008). Design and Analysis of Algorithms, Pearson Education.

Jena, S. R. and Swain, S. K, (2017). *Theory of Computation and Application*, 1st Edition, University Science Press, Laxmi Publications.

Levitin, A. (2012). Introduction to the Design and Analysis of Algorithms, 3rd Ed. Pearson Education, ISBN 10-0132316811

Module 1: Basic Algorithm Analysis

Unit 4: Recursion and Recursive Algorithms

1.0 Introduction

Recursion is a method of solving problems that involves breaking a problem down into smaller and smaller subproblems until you get to a small enough problem that it can be solved trivially. In computer science, recursion involves a function calling itself. While it may not seem like much on the surface, recursion allows us to write elegant solutions to problems that may otherwise be very difficult to program.

2.0 Objectives

By the end of this unit, you will be able to

- Know the meaning of Recursion and a Recursive algorithm
- Understand the different types of recursive algorithms
- See some examples of recursive algorithms
- Understand how the recursive algorithm works
- Know the difference between recursion and iteration
- Know the reasons why recursion is preferred in programming
- Know the runtime and space complexity of different recursive algorithms

3.0 Recursion and Recursive Algorithms (Definitions):

The Merriam-Webster describes recursion as:

*"*a computer programming technique involving the use of a procedure, subroutine, function, or algorithm that calls itself one or more times until a specified condition is met at which time the rest of each repetition is processed from the last one called to the first.*"*

Recursion is the process of defining something in terms of itself. A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

A recursive algorithm is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input.

There are two main instances of recursion. The first is when recursion is used as a technique in which a function makes one or more calls to itself. The second is when a data structure uses smaller instances of the exact same type of data structure when it represents itself.

3.1 Why use recursion?

Recursion provides an alternative for performing repetitions of the task in which a loop is not ideal. Most modern programming languages support recursion. Recursion serves as a great tool for building out particular data structures.

So now let's start with an example exercise of creating a factorial function.

3.1.1 Factorial Example

The factorial function is denoted with an exclamation point and is defined as the product of the integers from 1 to *n*. Formally, we can state this as:

$$n! = n \cdot (n-1) \cdot (n-2) \ldots 3 \cdot 2 \cdot 1$$

Note, **if n = 0, then n! = 1**. This is important to take into account, because it will serve as our *base case*.

Take this example:

$$4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24.$$

So how can we state this in a recursive manner? This is where the concept of **base case** comes in.

Base case is a key part of understanding recursion, especially when it comes to having to solve interview problems dealing with recursion. Let's rewrite the above equation of 4! so it looks like this:

$$4! = 4 \cdot (3 \cdot 2 \cdot 1) = 24$$

Notice that this is the same as:

$$4! = 4 \cdot 3! = 24$$

Meaning we can rewrite the formal recursion definition in terms of recursion like so:

$$n! = n \cdot (n-1)\ !$$

Note, **if n = 0, then n! = 1**. This means the **base case** occurs once n=0, the *recursive cases* are defined in the equation above. Whenever you are trying to develop a recursive solution it is very important to think about the base case, as your solution will need to return the base case once all the recursive cases have been worked through. Let's look at how we can create the factorial function in Python:

```
def fact(n):
    '''
    Returns factorial of n (n!).
    Note use of recursion
    '''
    # BASE CASE!
    if n == 0:
        return 1

    # Recursion!
    else:
        return n * fact(n-1)
```

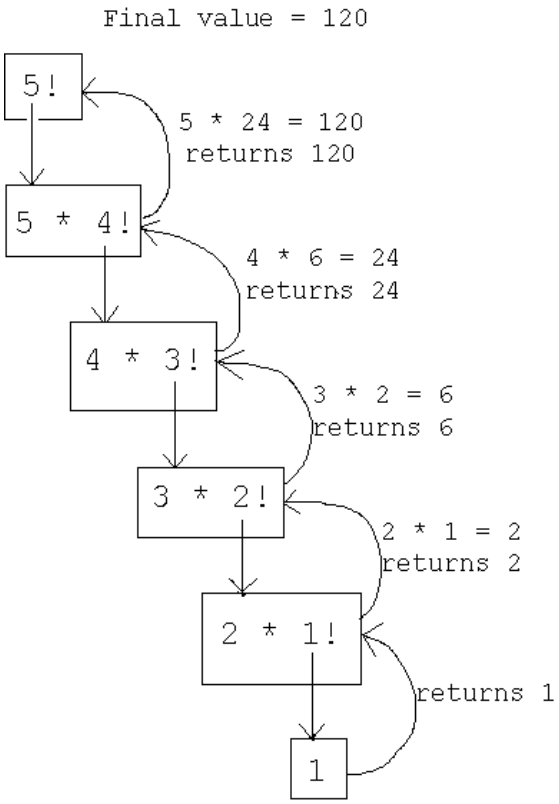
Let's see it in action!

```
fact(5) = 120
```

Note how we had an **if** statement to check if a base case occurred. Without it this function would not have successfully completed running. We can visualize the recursion with the following figure:

We can follow this flow chart from the top, reaching the base case, and then working our way back up.

Recursion is a powerful tool, but it can be a tricky concept to implement.



3.1.2 Purpose of Recursions

Recursive functions have many uses, but like any other kind of code, their necessity should be considered. As discussed above, consider the differences between recursions and loops, and use the one that best fits your needs. If you decide to go with recursions, decide what you want the function to do before you start to compose the actual code.

3.1.3 Conditionals to Start, Continue, and Stop the Recursion

It's important to look at any arguments or conditions that would start the recursion in the first place. For example, the function could have an argument that might be a string or array. The function itself may have to recognize the datatype versus it being recognized before this point (such as by a parent function). In simpler scenarios, starting conditions may often be the exact same conditions that force the recursion to continue.

More importantly, you want to establish a condition where the recursive action stops. These conditionals, known as base cases, produce an actual value rather

than another call to the function. However, in the case of tail-end recursion, the return value still calls a function but gets the value of that function right away.

The establishment of base cases is commonly achieved by having a conditional observe some quality, such as the length of an array or the amount of a number, just like loops. However, there are multiple ways to go about it, so feel free to alter the complexity as needed.

3.1.4 The Three Laws of Recursion

All recursive algorithms must obey three important laws:

1. A recursive algorithm must have a **base case**.
2. A recursive algorithm must change its state and move toward the base case.
3. A recursive algorithm must call itself, recursively.

Self-Assessment Exercises

1. What do you understand by the term “base case”?
2. Why must a stopping criterion be specified in a recursive algorithm?
3. What happens when a recursive algorithm calls itself recursively?

3.2 Types of Recursion

Recursion are mainly of two types depending on whether a function calls itself from within itself or more than one function call one another mutually. The first one is called direct recursion and another one is called indirect recursion.

Thus, the two types of recursion are:

3.2.1. Direct Recursion:

These can be further categorized into four types:

a. **Tail Recursion:**

If a recursive function calling itself and that recursive call is the last statement in the function then it’s known as Tail Recursion. After that call the recursive function performs nothing. The function has to process or perform any operation at the time of calling and it does nothing at returning time.

Example:

```
// Code Showing Tail Recursion
#include <iostream>
using namespace std;

// Recursion function
void fun(int n)
{
    if (n > 0) {
        cout << n << " ";

        // Last statement in the function
        fun(n - 1);
    }
}

// Driver Code
int main()
{
    int x = 3;
    fun(x);
    return 0;
}
```

Output:

```
3 2 1
```

Time Complexity For Tail Recursion : O(n)
Space Complexity For Tail Recursion : O(n)

Lets us convert Tail Recursion into Loop and compare each other in terms of Time & Space Complexity and decide which is more efficient.

```
// Converting Tail Recursion into Loop
#include <iostream>
using namespace std;

void fun(int y)
{
    while (y > 0) {
        cout << y << " ";
        y--;
    }
}

// Driver code
int main()
{
    int x = 3;
    fun(x);
    return 0;
```

```
}
```

Output

```
3 2 1
```

Time Complexity: **O(n)**
Space Complexity: **O(1)**

So it was seen that in case of loop the Space Complexity is O(1) so it was better to write code in loop instead of tail recursion in terms of Space Complexity which is more efficient than tail recursion.

b. **Head Recursion**:

If a recursive function calling itself and that recursive call is the first statement in the function then it's known as **Head Recursion.** There's no statement, no operation before the call. The function doesn't have to process or perform any operation at the time of calling and all operations are done at returning time.

Example:

```
// C++ program showing Head Recursion

#include <bits/stdc++.h>
using namespace std;

// Recursive function
void fun(int n)
{
    if (n > 0) {

        // First statement in the function
        fun(n - 1);

        cout << " "<< n;
    }
}
// Driver code
int main()
{
    int x = 3;
    fun(x);
    return 0;
}
```

Output:

```
1 2 3
```

Time Complexity For Head Recursion: **O(n)**
Space Complexity For Head Recursion: **O(n)**

Let's convert the above code into the loop.

```
// Converting Head Recursion into Loop
#include <iostream>
using namespace std;

// Recursive function
void fun(int n)
{
    int i = 1;
    while (i <= n) {
        cout <<" "<< i;
        i++;
    }
}

// Driver code
int main()
{
    int x = 3;
    fun(x);
    return 0;
}
```

Output:

```
1 2 3
```

c. **Tree Recursion**:

To understand **Tree Recursion** let's first understand **Linear Recursion**. If a recursive function calling itself for one time then it's known as **Linear Recursion**. Otherwise if a recursive function calling itself for more than one time then it's known as **Tree Recursion**.

Example: Pseudo Code for linear recursion

```
fun(n)
{
    // some code
    if(n>0)
    {
        fun(n-1); // Calling itself only once
    }
    // some code
}
```

Program for tree recursion

```
// C++ program to show Tree Recursion
```

```
#include <iostream>
using namespace std;

// Recursive function
void fun(int n)
{
    if (n > 0)
    {
        cout << " " << n;

        // Calling once
        fun(n - 1);

        // Calling twice
        fun(n - 1);
    }
}

// Driver code
int main()
{
    fun(3);
    return 0;
}
```

Output:

```
3 2 1 1 2 1 1
```

Time Complexity For Tree Recursion: **$O(2^n)$**

Space Complexity For Tree Recursion: **O(n)**

.

d. **Nested Recursion**:

In this recursion, a recursive function will pass the parameter as a recursive call. That means **"recursion inside recursion".** Let see the example to understand this recursion.

Example:

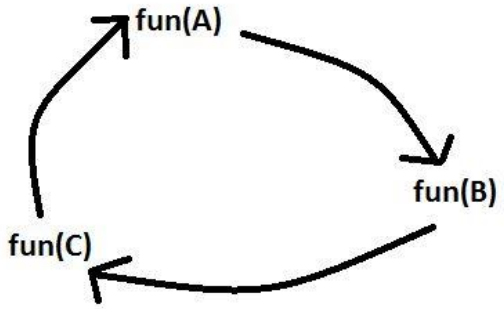
```
// C++ program to show Nested Recursion
#include <iostream>
using namespace std;
int fun(int n)
{
    if (n > 100)
        return n - 10;
    // A recursive function passing parameter
    // as a recursive call or recursion inside
    // the recursion
    return fun(fun(n + 11));
}
 // Driver code
int main()
{
    int r;
    r = fun(95);
     cout << " " << r;
     return 0;
}
```

Output:

```
9  1
```

3.2.2. Indirect Recursion:

In this recursion, there may be more than one functions and they are calling one another in a circular manner.



From the above diagram fun(A) is calling for fun(B), fun(B) is calling for fun(C) and fun(C) is calling for fun(A) and thus it makes a cycle.

Example:

```
// C++ program to show Indirect Recursion
```

```
#include <iostream>
using namespace std;
void funB(int n);
void funA(int n)
{
    if (n > 0) {
        cout <<" "<< n;
        // Fun(A) is calling fun(B)
        funB(n - 1);
    }
}
void funB(int n)
{
    if (n > 1) {
        cout <<" "<< n;
        // Fun(B) is calling fun(A)
        funA(n / 2);
    }
}

// Driver code
int main()
{
    funA(20);
    return 0;
}
```

Output:

```
20  19  9  8  4  3  1
```

3.3 Recursion versus Iteration

The Recursion and Iteration both repeatedly execute the set of instructions. Recursion is when a statement in a function calls itself repeatedly. The iteration is when a loop repeatedly executes until the controlling condition becomes false. The primary difference between recursion and iteration is that recursion is a process, always applied to a function and iteration is applied to the set of instructions which we want to get repeatedly executed.

Recursion

- Recursion uses **selection structure**.
- **Infinite recursion** occurs if the recursion step does not reduce the problem in a manner that converges on some condition (**base case**) and Infinite recursion can crash the system.
- Recursion terminates when a **base case** is recognized.

- Recursion is usually **slower than iteration** due to the overhead of maintaining the stack.
- Recursion uses **more memory than iteration**.
- Recursion makes the **code smaller**.

Iteration

- Iteration uses **repetition structure**.
- An infinite loop occurs with iteration if the loop condition test never becomes false and Infinite looping uses CPU cycles repeatedly.
- An iteration **terminates** when the **loop condition fails**.
- An iteration does not use the **stack** so it's **faster than recursion**.
- Iteration consumes **less memory.**
- Iteration makes the **code longer**.

Self-Assessment Exercises

1. Try and find the Sum of the elements of an array recursively

2. Find the maximum number of elements in an array A of *n* elements using recursion

3. How is recursion different from iteration?

3.4 Some Recursive Algorithms (Examples)

3.4.1 Reversing an Array

Let us consider the problem of reversing the *n* elements of an array, A, so that the first element becomes the last, the second element becomes the second to the last, and so on. We can solve this problem using the linear recursion, by observing that the reversal of an array can be achieved by swapping the first and last elements and then recursively reversing the remaining elements in the array.

```
Algorithm ReverseArray(A, i, j):
    Input: An array A and nonnegative integer indices i and j
```

```
    Output: The reversal of the elements in A starting at
index i and ending at j
    if i < j then
        Swap A[i] and A[j]
        ReverseArray(A, i+1, j-1)
    return
```

3.4.2 Fibonacci Sequence

Fibonacci sequence is the sequence of numbers 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... The first two numbers of the sequence are both 1, while each succeeding number is the sum of the two numbers before it. We can define a function F(n) that calculates the nth Fibonacci number.

First, the base cases are: F(0) = 1 and F(1) = 1.

Now, the recursive case: F(n) = F(n-1) + F(n-2).

Write the recursive function and the call tree for F(5).

```
Algorithm Fib(n) {
    if (n < 2) return 1
    else return Fib(n-1) + Fib(n-2)
}
```

The above recursion is called **binary recursion** since it makes two recursive calls instead of one. How many number of calls are needed to compute the kth Fibonacci number? Let n_k denote the number of calls performed in the execution.

$n_0 = 1$

$n_1 = 1$

$n_2 = n_1 + n_0 + 1 = 3 > 2^1$

$n_3 = n_2 + n_1 + 1 = 5 > 2^2$

$n_4 = n_3 + n_2 + 1 = 9 > 2^3$

$n_5 = n_4 + n_3 + 1 = 15 > 2^3$

...

$n_k > 2^{k/2}$

This means that the Fibonacci recursion makes a number of calls that are exponential in *k*. In other words, using binary recursion to compute Fibonacci numbers is very inefficient. Compare this problem with binary search, which is very efficient in searching items, why is this binary recursion inefficient? The main problem with the approach above, is that there are multiple overlapping recursive calls.

We can compute F(*n*) much more efficiently using linear recursion. One way to accomplish this conversion is to define a recursive function that computes a pair of consecutive Fibonacci numbers F(*n*) and F(*n*-1) using the convention F(-1) = 0.

```
Algorithm LinearFib(n) {
    Input: A nonnegative integer n
    Output: Pair of Fibonacci numbers (F_n, F_n-1)
    if (n <= 1) then
        return (n, 0)
    else
        (i, j) <-- LinearFib(n-1)
        return (i + j, i)
}
```

Since each recursive call to *LinearFib* decreases the argument *n* by 1, the original call results in a series of *n*-1 additional calls. This performance is significantly faster than the exponential time needed by the binary recursion. Therefore, when using binary recursion, we should first try to fully partition the problem in two or, we should be sure that overlapping recursive calls are really necessary.

Let's use iteration to generate the Fibonacci numbers. What's the complexity of this algorithm?

```
public static int IterationFib(int n) {
    if (n < 2) return n;
    int f0 = 0, f1 = 1, f2 = 1;
    for (int i = 2; i < n; i++) {
        f0 = f1;
        f1 = f2;
        f2 = f0 + f1;
    }
    return f2;
}
```

Self-Assessment Exercises

1. Either write the pseudo-code or the Java code for the following problems. Draw the recursion trace of a simple case. What is the running time and space requirement?.

    - Recursively searching a linked list
    - Forward printing a linked list
    - Reverse printing a linked list

4.0 Conclusion

Recursive algorithms are very important in programming as they help us write very good programs and also allow us to understand the concept of computing well. So many programs are naturally recursive and many others can be turned into a recursive algorithm.

5.0 Summary

In computer science, recursion is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem. Such problems can generally be solved by iteration, but this needs to identify and index the smaller instances at programming time. There exist several natural examples of recursive algorithms while other programming algorithms that are iterative can be turned into recursive algorithms.

The concept of recursion is very important to developers of algorithms and also to programmers.

6.0 Tutor Marked Assignment

1 Given the following recursive algorithm:

$F_0 = 1, \quad F_1 = 1$

$F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$

Find F_{10} and F_{15} by simulating it manually

2. Mathematically, the greatest common divisor, gcd is given as:

$$\gcd(p, q) = \begin{cases} p, & \text{if } q = 0 \\ \gcd(q, \text{remainder}(p, q)), & \text{if } p \geq q \text{ and } q > 0 \end{cases}$$

Compute; i. gcd (48, 12) ii. gcd (1035, 759)

3. What makes recursion better than iteration and what makes iteration better than recursion.
4. Give a vital difference between Head recursion and Tail recursion.

7.0 Further Reading and Other References

Cormen, T. H., Leiserson, C., Rivest, R. and Stein, C. (2009). *Introduction to Algorithms*. Third Edition. MIT Press.

Jena, S. R. and Swain, S. K, (2017). *Theory of Computation and Application*, 1st Edition, University Science Press, Laxmi Publications.

Module 1: Algorithm Analysis

Unit 5: Recurrence Relations

1.0 Introduction

A **recurrence** or **recurrence relation** on the other hand defines an infinite sequence by describing how to calculate the n-th element of the sequence given the values of smaller elements, as in:

$$T(n) = T(n/2) + n,\ T(0) = T(1) = 1.$$

In principle such a relation allows us to calculate T(n) for any n by applying the first equation until we reach the base case. To *solve* a recurrence, we will find a formula that calculates T(n) directly from n, without this recursive computation.

2.0 Objectives

By the end of this unit, you will be able to

- Know more about Recurrences and Recurrence relations
- Understand the different methods for resolving recurrence relations
- Know the areas of applications of recurrence relations

3.0 Recurrence Relations

A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs. To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence.

For Example, the Worst Case Running Time T(n) of the MERGE SORT Procedures is described by the recurrence.

$$T(n) = \theta(1) \text{ if } n=1$$

$$2T\left(\frac{n}{2}\right) + \theta(n) \text{ if } n>1$$

3.1 Methods for Resolving Recurrence Relations

Recurrence relations can be resolved with any of the following four methods:

1. Substitution Method (Guess-and-Verify)

2. Iteration Method.
3. Recursion Tree Method.
4. Master Method.

3.1.1. Guess-and-Verify Method:

As when solving any other mathematical problem, we are not required to explain where our solution came from as long as we can prove that it is correct. So the most general method for solving recurrences can be called "guess but verify". Naturally, unless you are very good friends with the existential quantifier you may find it had to come up with good guesses. But sometimes it is possible to make a good guess by iterating the recurrence a few times and seeing what happens.

The Guess-and-Verify Method consists of two main steps:

1. Guess the Solution.
2. Use the mathematical induction to find the boundary condition and shows that the guess is correct.

For Example1 Solve the equation by Substitution Method.

$$T(n) = T\left(\frac{n}{2}\right) + n$$

We have to show that it is asymptotically bound by O (log n).

Solution:

For $T(n) = O(\log n)$

We have to show that for some constant c

$$T(n) \leq c \log n.$$

Put this in given Recurrence Equation.

$$T(n) \leq c \log\left(\frac{n}{2}\right) + 1$$

$$\leq c \log\left(\frac{n}{2}\right) + 1 = c \log n - c \log_2 2 + 1$$

$$\leq c \log n \text{ for } c \geq 1$$

Thus **T (n) =O logn**.

Example2 Consider the Recurrence

$$T(n) = 2T\left(\frac{n}{2}\right) + n \quad n>1$$

Find an Asymptotic bound on T.

Solution:

We guess the solution is O (n (logn)).Thus for constant 'c'.

$$T(n) \le c\, n \log n$$

Put this in given Recurrence Equation.

Now,

$$\begin{aligned} T(n) &\le 2c\left(\frac{n}{2}\right)\log\left(\frac{n}{2}\right) + n \\ &\le cn\log n - cn\log 2 + n \\ &= cn\log n - n(c\log 2 - 1) \\ &\le cn\log n \text{ for } (c \ge 1) \end{aligned}$$

Thus **T (n) = O (n logn)**.

3.1.2. Iteration Methods

It means to expand the recurrence and express it as a summation of terms of n and initial condition.

Example1: Consider the Recurrence

$$T(n) = 1 \quad \textbf{if } n=1$$
$$= 2T(n-1) \quad \textbf{if } n>1$$

Solution:

$$\begin{aligned} T(n) &= 2T(n-1) \\ &= 2[2T(n-2)] = 2^2T(n-2) \\ &= 4[2T(n-3)] = 2^3T(n-3) \\ &= 8[2T(n-4)] = 2^4T(n-4) \quad \text{(Eq.1)} \end{aligned}$$

Repeat the procedure for i times

$T(n) = 2^i\ T(n-i)$
Put $n-i=1$ or $i= n-1$ in (Eq.1)
$T(n) = 2^{n-1}\ T(1)$
$= 2^{n-1} .1$ {T (1) =1 .....given}
$= 2^{n-1}$

Example2: Consider the Recurrence

$T(n) = T(n-1) +1$ and $T(1) = \theta(1)$.

Solution:

$T(n) = T(n-1) +1$
$= (T(n-2) +1) +1 = (T(n-3) +1) +1+1$
$= T(n-4) +4 = T(n-5) +1+4$
$= T(n-5) +5= T(n-k) + k$
where $k = n-1$
$T(n-k) = T(1) = \theta(1)$
$T(n) = \theta(1) + (n-1) = 1+n-1=n= \theta(n)$.

3.1.3 Recursion Tree Method

Recursion Tree Method is a pictorial representation of an iteration method which is in the form of a tree where at each level nodes are expanded.

In general, we consider the second term in recurrence as root. It is useful when the divide & Conquer algorithm is used.

It is sometimes difficult to come up with a good guess. In Recursion tree, each root and child represents the cost of a single subproblem. We sum the costs within each of the levels of the tree to obtain a set of pre-level costs and then sum all pre-level costs to determine the total cost of all levels of the recursion.

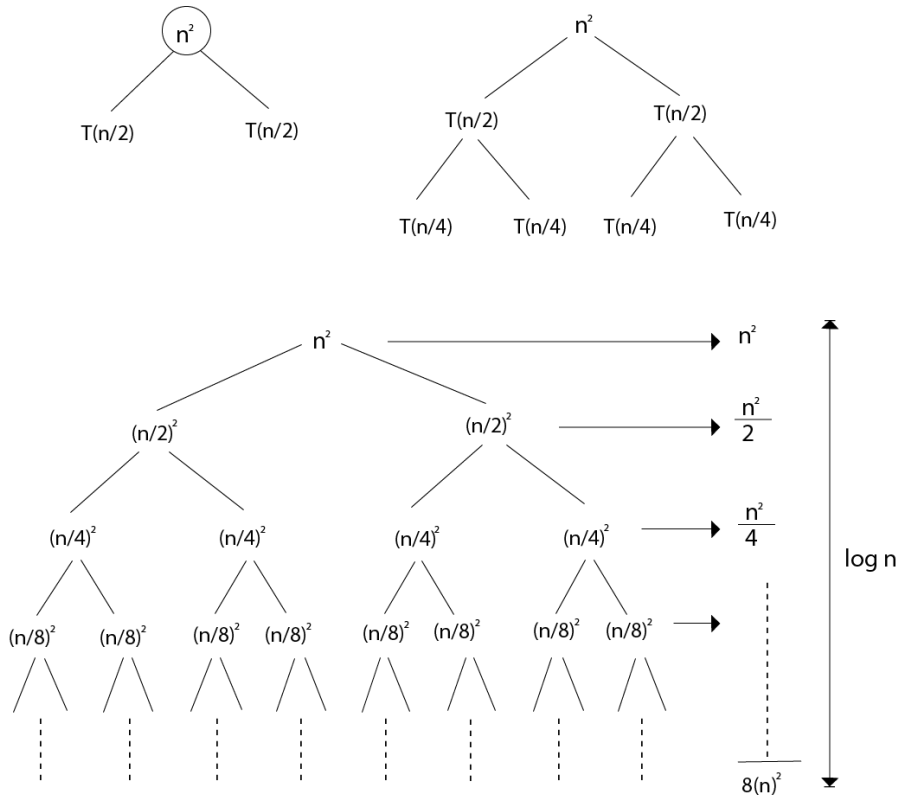
A Recursion Tree is best used to generate a good guess, which can be verified by the Substitution Method.

Example 1

Consider $T(n) = 2T\left(\frac{n}{2}\right) + n^2$

We have to obtain the asymptotic bound using recursion tree method.

Solution: The Recursion tree for the above recurrence is



$$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \ldots\ldots \log n \text{ times.}$$

$$\leq n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2^i}\right)$$

$$\leq n^2 \left(\frac{1}{1-\frac{1}{2}}\right) \leq 2n^2$$

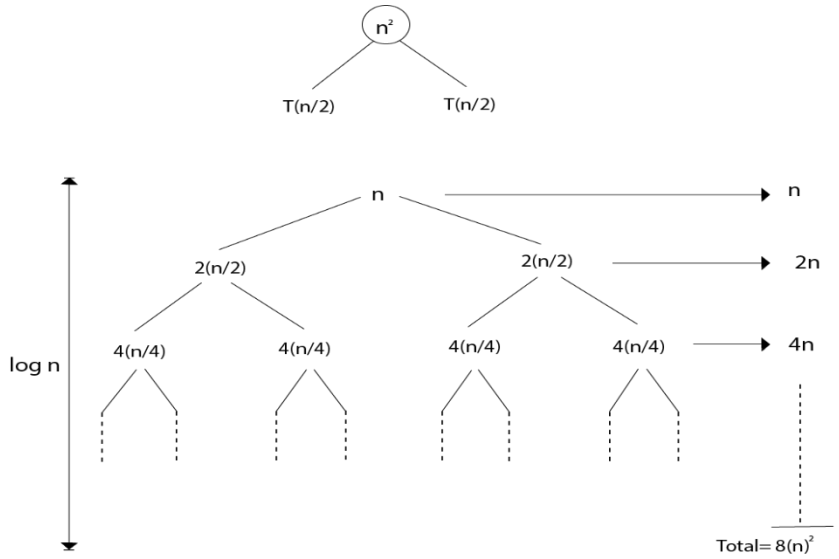
$$\mathbf{T(n) = \theta n^2}$$

Example 2: Consider the following recurrence

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

Obtain the asymptotic bound using recursion tree method.

Solution: The recursion trees for the above recurrence



We have $n + 2n + 4n + \ldots\ldots \log_2 n$ times

$$= n\,(1 + 2 + 4 + \ldots\ldots \log_2 n \text{ times})$$

$$= n\,\frac{(2^{\log_2 n} - 1)}{(2-1)} = \frac{n(n-1)}{1} = n^2 - n = \theta(n^2)$$

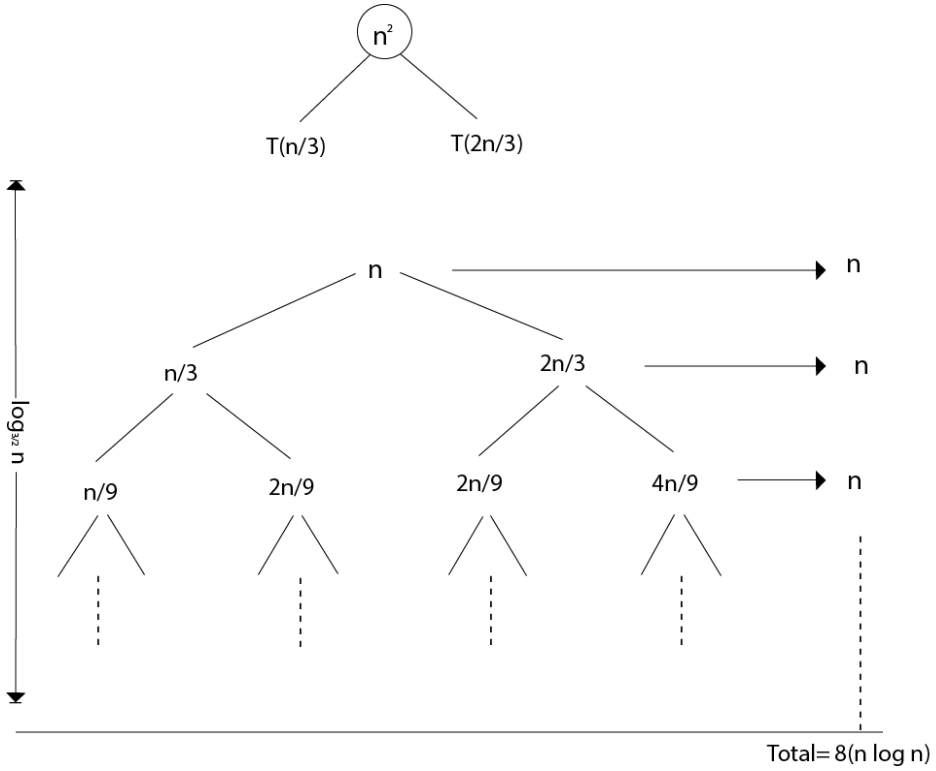
$$\mathbf{T(n) = \theta(n^2)}$$

Example 3: Consider the following recurrence

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$$

Obtain the asymptotic bound using recursion tree method.

Solution: The given Recurrence has the following recursion tree



When we add the values across the levels of the recursion trees, we get a value of n for every level. The longest path from the root to leaf is

$$n \longrightarrow \frac{2}{3}n \longrightarrow \left(\frac{2}{3}\right) n \longrightarrow \ldots 1$$

Since $\left(\frac{2}{3}\right) n=1$ when $i=\log \frac{3}{2}\ n$.

Thus the height of the tree is $\log \frac{3}{2}\ n$.

$$T(n) = n + n + n + \ldots\ldots + \log \frac{3}{2}\ n \text{ times.} = \boldsymbol{\theta(n \log n)}$$

3.1.5 Master Method

The Master Method is used for solving the following types of recurrence

T (n) = a $T\left(\frac{n}{b}\right)$ + f (n) with a≥1 and b≥1 be constant & f(n) be a function and $\frac{n}{b}$ can be interpreted as

Let T (n) is defined on non-negative integers by the recurrence.

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n)$$

In the function to the analysis of a recursive algorithm, the constants and function take on the following significance:

- n is the size of the problem.
- a is the number of subproblems in the recursion.
- n/b is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
- f (n) is the sum of the work done outside the recursive calls, which includes the sum of dividing the problem and the sum of combining the solutions to the subproblems.
- It is not possible always bound the function according to the requirement, so we make three cases which will tell us what kind of bound we can apply on the function.

Master Theorem:

It is possible to complete an asymptotic tight bound in these three cases:

$$T(n)=\begin{cases} \Theta\left(n^{\log_b a}\right) & f(n)=O\left(n^{\log_b a-\varepsilon}\right) \\ \Theta\left(n^{\log_b a}\log n\right) & f(n)=\Theta\left(n^{\log_b a}\right) \\ \Theta(f(n)) & f(n)=\Omega\left(n^{\log_b a+\varepsilon}\right)\text{AND} \\ & af(n/b)<cf(n)\ \text{forlarge}n \end{cases} \quad \begin{matrix} \varepsilon>0 \\ c<1 \end{matrix}$$

Case1: If f (n) = $O\left(n^{\log_b a-\varepsilon}\right)$ for some constant ε >0, then it follows that:

T (n) = Θ $\left(n^{\log_b a}\right)$

Example:

T (n) = 8 T $\left(\frac{n}{2}\right)+1000n^2$ apply master theorem on it.

Solution:

Compare T (n) = 8 T $\left(\frac{n}{2}\right)+1000n^2$ with

T (n) = a T $\left(\frac{n}{b}\right)+f(n)$ *with* $a \geq 1$ *and* $b>1$

a = 8, b=2, f (n) = 1000 n^2, $\log_b a = \log_2 8 = 3$

Put all the values in: f (n) = $O\left(n^{\log_b a-\varepsilon}\right)$

1000 n^2 = O ($n^{3-\epsilon}$)

If we choose ϵ=1, we get: 1000 n^2 = O (n^{3-1}) = O (n^2)

Since this equation holds, the first case of the master theorem applies to the given recurrence relation, thus resulting in the conclusion:

T (n) = Θ $\left(n^{\log_b a}\right)$

Therefore: T (n) = Θ (n^3)

Case 2: If it is true, for some constant k ≥ 0 that:

F (n) = Θ $\left(n^{\log_b a}\log^k n\right)$

then it follows that: T (n) = Θ $\left(n^{\log_b a}\log^{k+1} n\right)$

Example:

T (n) = 2 $T\left(\frac{n}{2}\right)+10n$, solve the recurrence by using the master method.

As compare the given problem with T (n) = a T $\left(\frac{n}{b}\right)+f(n)$ *with* $a \geq 1$ *and* $b>1$

a = 2, b=2, k=0, f (n) = 10n, $\log_b a = \log_2 2 = 1$

Put all the values in f (n) =Θ $\left(n^{\log_b a}\log^k n\right)$, we will get

10n = Θ (n^1) = Θ (n) which is true.

Therefore: T (n) = Θ $\left(n^{\log_b a} \log^{k+1} n\right)$

= Θ (n log n)

Case 3: If it is true f(n) = Ω $\left(n^{\log_b a+\varepsilon}\right)$ for some constant ε >0 and it also true that: a f $\left(\frac{n}{b}\right) \leq cf(n)$ for some constant c<1 for large value of n ,then :

```
T (n) = Θ((f (n))
```

Example: Solve the recurrence relation:

T (n) = 2 $T\left(\frac{n}{2}\right)+n^2$

Solution:

Compare the given problem with

T (n) = a T $\left(\frac{n}{b}\right)+f(n)$ *with* $a \geq 1$ *and* $b>1$

a= 2, b =2, f (n) = n^2, $\log_b a = \log_2 2 = 1$

Put all the values in f (n) = Ω $\left(n^{\log_b a+\varepsilon}\right)$ ..... (Eq. 1)

If we insert all the value in (Eq.1), we will get

$n^2 = \Omega(n^{1+\varepsilon})$ put ε =1, then the equality will hold.

$n^2 = \Omega(n^{1+1}) = \Omega(n^2)$

Now we will also check the second condition:

2 $\left(\frac{n}{2}\right)^2 \leq cn^2 \Rightarrow \frac{1}{2}n^2 \leq cn^2$

If we will choose c =1/2, it is true:

$\frac{1}{2}n^2 \leq \frac{1}{2}n^2$ ∀ n ≥1

So it follows: T (n) = Θ ((f (n))

$T(n) = \Theta(n^2)$

Self-Assessment Exercises

1. How is the Guess-and-Verify method better than the Iteration method
2. Is a recurrence relation similar to a recursive algorithm? Discuss.
3. What is the essence of the base case in every recurrence relation?

3.2 Example of Recurrence Relation: Tower of Hanoi

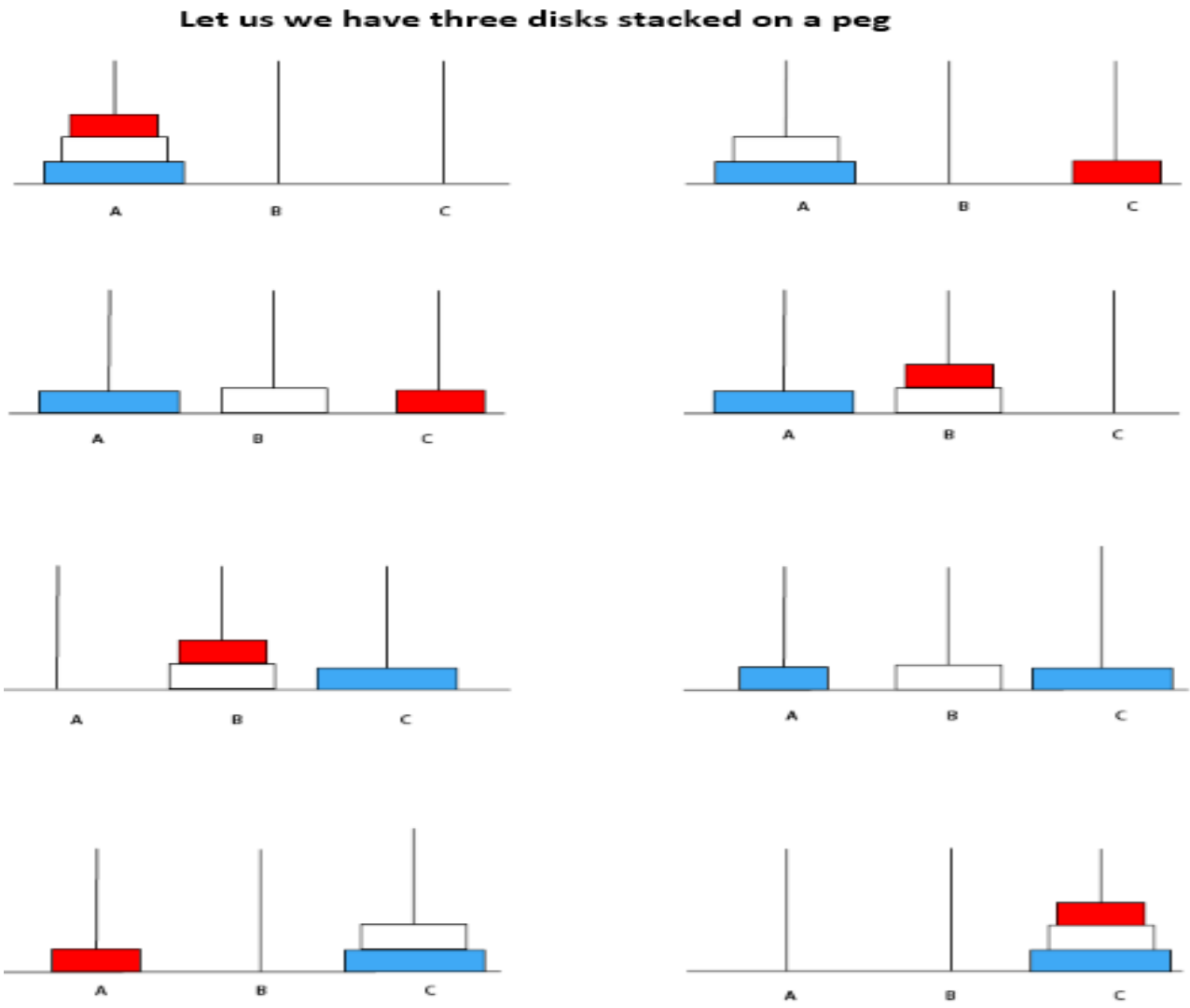
It was invented in 1883 by mathematician Edouard Lucas. He wanted to sell his 8-disk puzzle, and he created the name and the story to make the puzzle more intriguing. The pegs are of diamond, and the 64 disks are of gold. They were put there in an ancient temple at Hanoi, Vietnam by the Creator who gave the Monks the following divine conditions:

1. The disks must all be moved one at a time from one peg to another peg using only three pegs.

2. No larger disk should be placed on top of a smaller disk

3. Only one disk can be transferred at a time.

Once all the disks have been moved, the World will end !!! This problem can be easily solved by Divide & Conquer algorithm



In the above 7 step all the disks from peg A will be transferred to C given Condition:

1. Only one disk will be shifted at a time.
2. Smaller disk can be placed on larger disk.

Let T (n) be the total time taken to move n disks from peg A to peg C

1. Moving n-1 disks from the first peg to the second peg. This can be done in T (n-1) steps.
2. Moving larger disks from the first peg to the third peg will require first one step.
3. Recursively moving n-1 disks from the second peg to the third peg will require again T (n-1) step.

So, total time taken T (n) = T (n-1)+1+ T(n-1)

Relation formula for Tower of Hanoi is:

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

Note: Stopping Condition: T (1) =1

Because at last there will be one disk which will have to move from one peg to another.

T (n) = 2T (n-1) +1.....eq1

Put n = n-1 in eq 1

T (n-1) = 2T (n-2) +1.....eq2

Putting 2 eq in 1 eq

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

T (n) = 2^2T (n-2) +2+1.....eq3

Put n = n-2 in eq 1

T (n-2) = 2T (n-3) +1.....eq4

Putting 4 eq in 3 eq

T (n) = $2^2[2T(n-3)+1]+2+1$

T (n) =$2^3T(n-3)+2^2+2+1$.....eq5

From 1 eq, 3 eq, 5 eq

We get,

$$T(n) = 2^iT(n-i) + 2^{i-1} + 2^{i-2} + \ldots\ldots\ldots\ldots + 2^0$$

Now n-i=1 from stopping condition

And T (n-i) =1

n-1 =i ← A equation

Now, T (n) = $2^i(1) + 2^{i-1} + 2^{i-2} + \ldots\ldots\ldots\ldots\ldots\ldots + 2^0$

It is a Geometric Progression Series with common ratio, r=2.

First term, a=1(2^0)

Sum of n terms in G.P= $S_n = \frac{a(1-r^n)}{1-r}$

So T (n) = $\frac{1(1-2^{i+1})}{(1-2)}$

T (n) = $\frac{2^{i+1}-1}{2-1} = 2^{i+1} - 1$

T (n) = $2^{i+1} - 1$

From A

$T(n) = 2^{n-1+1} - 1$

T (n) = 2^n-1.....B Equation

Because exponents are from 0 to 1 n=i+1

B equation is the required complexity of technique tower of Hanoi when we have to move n disks from one peg to another.

T (3) = 2^3- 1
= 8 - 1

= **7 Ans**

[As in concept we have proved that there will be 7 steps now proved by general equation]

3.2.1 Program for Tower of Hanoi:

```
#include<stdio.h>
void towers(int, char, char, char);
 int main()
 {
       int num;
       printf ("Enter the number of disks : ");
        scanf ("%d", &num);
```

```
printf("The sequence of moves involved in the Tower of Hanoi a
re:\n");
      towers (num, 'A', 'C', 'B');
      return 0;

}
     void towers( int num, char from peg, char topeg, char aux
peg)
 {
           if (num == 1)
 {
     printf ("\n Move disk 1 from peg %c to peg %c", from peg,
 topeg);
           return;
 }
   Towers (num - 1, from peg, auxpeg, topeg);
Printf ("\n Move disk %d from peg %c to peg %c", num, from peg
, topeg);
   Towers (num - 1, auxpeg, topeg, from peg);
 }
```

3.2.2 Applications of Tower of Hanoi problem

It has been used to determine the extent of brain injuries and helps to build/rebuild neural pathways in the brain as attempting to solve, Tower of Hanoi uses parts of the brain that involve managing time, foresight of whether the next move will lead us closer to the solution or not.

The Tower of Hanoi is a simple puzzle game that is used to amuse children. It is also often used as programming challenge when discussing recursion,

3.2.3 Finding a Recurrence (Tower of Hanoi)

To answer how long it will take our friendly monks to destroy the world, we write a recurrence (let's call it $M(n)$) for the number of moves MoveTower takes for an n-disk tower.

The base case - when n is 1 - is easy: The monks just move the single disk directly.

$M(1) = 1$

In the other cases, the monks follow our three-step procedure. First they move the (n-1)-disk tower to the spare peg; this takes $M(n\text{-}1)$ moves. Then the monks move the nth disk, taking 1 move. And finally they move the (n-1)-disk tower again,

this time on top of the *n*th disk, taking $M(n\text{-}1)$ moves. This gives us our recurrence relation,

$M(n) = 2\ M(n\text{-}1) + 1.$

Since the monks are handling a 64-disk tower, all we need to do is to compute $M(64)$, and that tells us how many moves they will have to make.

This would be more convenient if we had $M(n)$ into a *closed-form solution* - that is, if we could write a formula for $M(n)$ without using recursion. Do you see what it should be? (It may be helpful if you go ahead and compute the first few values, like $M(2)$, $M(3)$, and $M(4)$.)

3.2.4 Closed-form solution

Let's figure out values of *M* for the first few numbers.

$$M(1) = 1$$

$$M(2) = 2M(1) + 1 = 3$$

$$M(3) = 2M(2) + 1 = 7$$

$$M(4) = 2M(3) + 1 = 15$$

$$M(5) = 2M(4) + 1 = 31$$

By looking at this, we can guess that

$M(n) = 2^n - 1.$

We can verify this easily by plugging it into our recurrence.

$M(1) = 1 = 2^1 - 1$
$M(n) = 2\ M(n - 1) + 1 = 2\ (2^{n-1} + 1) - 1 = 2^n + 1$

Since our expression 2^n+1 is consistent with all the recurrence's cases, this is the closed-form solution.

So the monks will move $2^{64}+1$ (about 18.45×10^{18}) disks. If they are really good and can move one disk a millisecond, then they'll have to work for 584.6 million years. It looks like we're safe.

Self-Assessment Exercise

1. Simulate the Tower-of-Hanoi problem for N = 7 disks and N = 12 disks.
2. Can we solve the Tower of Hanoi problem for any value of Tn without using a Recurrence relation? Discuss.
3. What are the application areas for the Tower of Hanoi problem?

4.0 Conclusion

Recurrence relation permits us to compute the members of a sequence one after the other starting from one or more initial values.

Recurrence relations apply recursion completely and there exist one or more base cases to help determine the stopping criterion.

5.0 Summary

In mathematics and computer science, a recurrence relation is an equation that expresses the nth term of a sequence as a function of the k preceding terms, for some fixed k, which is called the order of the relation. Recurrence relations can be solved by several methods ranging from the popular Guess-and-Verify method to the Master method and they help us understand the workings of algorithms better.

6.0 Tutor Marked Assignment

1. A new employee at an exciting new software company starts with a salary of ₦50,000 and is promised that at the end of each year her salary will be double her salary of the previous year, with an extra increment of ₦10,000 for each year she has been with the company.
   a) Construct a recurrence relation for her salary for her n-th year of employment.
   b) Solve this recurrence relation to find her salary for her n-th year of employment.
2. Suppose that there are two goats on an island initially. The number of goats on the island doubles every year by natural reproduction, and some goats are either added or removed each year.
   a) Construct a recurrence relation for the number of goats on the island at the start of the n-th year, assuming that during each year an extra 100 goats are put on the island.

b) Solve the recurrence relation from part (a) to find the number of goats on the island at the start of the n-th year.
c) Construct a recurrence relation for the number of goats on the island at the start of the n-th year, assuming that n goats are removed during the n-th year for each $n \geq 3$.
d) Solve the recurrence relation in part (c) for the number of goats on the island at the start of the n-th year.

3. a) Find all solutions of the recurrence relation $a_n = 2a_n - 1 + 2_n 2 . a_n = 2a_n - 1 + 2_n 2$.
b) Find the solution of the recurrence relation in part (a) with initial condition $a_1 = 4$

7.0 Further Reading and other Resources

Jena, S. R. and Swain, S. K, (2017). *Theory of Computation and Application*, 1st Edition, University Science Press, Laxmi Publications.

Michalewicz, Z. and Fogel, D. (2004). *How to Solve It: Modern Heuristics*. Second Edition. Springer.

Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

CIT310: ALGORITHMS AND COMPLEXITY ANALYSIS

Module 2 Searching and Sorting Algorithms

Unit 1 Bubble Sort and Selection Sort Algorithm

Unit 2 Insertion Sort and Radix Sort Algorithms

Unit 3 Linear Search and Stability in Sorting

Unit 4 Divide-and-Conquer Strategies I: Binary Search

Unit 5 Divide-and-Conquer Strategies II: Merge Sort and Quicksort Algorithms

Module 2: Sorting and Searching Algorithms

Unit 1: Bubble Sort and Selection Sort Algorithm

1.0 Introduction

Sorting and searching are two of the most frequently needed algorithms in program design. Common algorithms have evolved to take account of this need.

Since computers were created, users have devised programs, many of which have needed to do the same thing. As a result, common algorithms have evolved and been adopted in many programs.

Two algorithms often used are searches and sorts:

- searches allow a set of data to be examined and for a specific item to be found
- sorts allow a data set to be sorted into order

Methods of searching include:

- linear search
- binary search

Methods of sorting include:

- bubble sort
- merge sort
- insertion sort
- quicksort
- radix sort
- selection sort

2.0 Objectives

By the end of this unit, you should be able to:

- Know some of the techniques for sorting a list containing numbers or texts
- Identify how the bubble sort and Selection sort algorithm works
- Know some benefits and disadvantages of Bubble sort and Selection sort
- Identify the worst case and best case of Bubble sort and selection sort
- Know where the bubble sort and selection sort algorithms are applied

3.0 Bubble Sort

Bubble Sort, also known as Exchange Sort, is a simple sorting algorithm. It works by repeatedly stepping throughout the list to be sorted, comparing two items at a time and swapping them if they are in the wrong order. The pass through the list is duplicated until no swaps are desired, which means the list is sorted.

This is the easiest method among all sorting algorithms.

Algorithm

```
Step 1 ➤ Initialization

     set 1 ← n, p ← 1

Step 2 ➤ loop,

Repeat through step 4 while (p ≤ n-1)
set E ← 0 ➤ Initializing exchange variable.

Step 3 ➤ comparison, loop.

     Repeat for i ← 1, 1, ….. l-1.
     if (A [i] > A [i + 1]) then
     set A [i] ↔ A [i + 1] ➤ Exchanging values
.
     Set E ← E + 1

Step 4 ➤ Finish, or reduce the size.

     if (E = 0) then
     exit
     else
     set l ← l - 1.
```

3.1 How Bubble Sort Works

1. The bubble sort starts with the very first index and makes it a bubble element. Then it compares the bubble element, which is currently our first index element, with the next element. If the bubble element is greater and the second element is smaller, then both of them will swap.

After swapping, the second element will become the bubble element. Now we will compare the second element with the third as we did in the earlier step and swap them if required. The same process is followed until the last element.

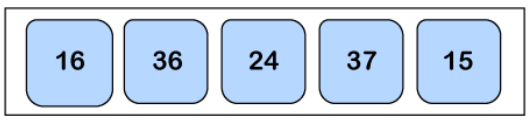
2. We will follow the same process for the rest of the iterations. After each of the iteration, we will notice that the largest element present in the unsorted array has reached the last index.

For each iteration, the bubble sort will compare up to the last unsorted element.

Once all the elements get sorted in the ascending order, the algorithm will get terminated.

Consider the following example of an unsorted array that we will sort with the help of the Bubble Sort algorithm.

Initially,



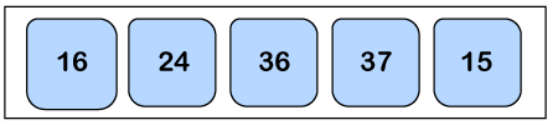
Pass 1:

- **Compare a_0 and a_1**

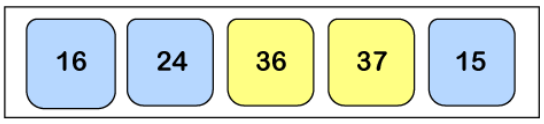
As $a_0 < a_1$ so the array will remain as it is.

- **Compare a_1 and a_2**

Now $a_1 > a_2$, so we will swap both of them.

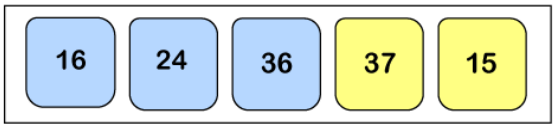


- **Compare a_2 and a_3**

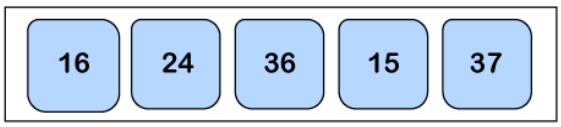


As $a_2 < a_3$ so the array will remain as it is.

- **Compare a_3 and a_4**



Here $a_3 > a_4$, so we will again swap both of them.

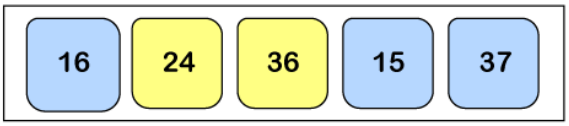


Pass 2:

- **Compare a_0 and a_1**

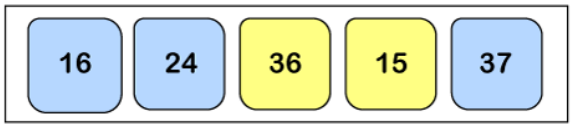
As $a_0 < a_1$ so the array will remain as it is.

- **Compare a_1 and a_2**

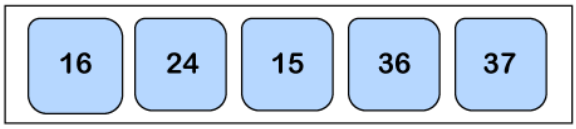


Here $a_1 < a_2$, so the array will remain as it is.

- **Compare a_2 and a_3**



In this case, $a_2 > a_3$, so both of them will get swapped.



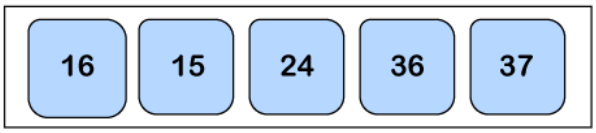
Pass 3:

- **Compare a_0 and a_1**

As $a_0 < a_1$ so the array will remain as it is.

- **Compare a_1 and a_2**

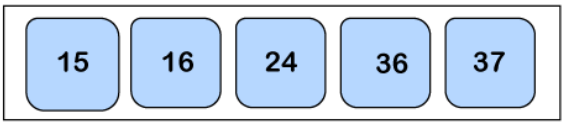
Now $a_1 > a_2$, so both of them will get swapped.



Pass 4:

- **Compare a_0 and a_1**

Here $a_0 > a_1$, so we will swap both of them.



Hence the array is sorted as no more swapping is required.

3.2.1 Complexity Analysis of Bubble Sort

Input: Given n input elements.

Output: Number of steps incurred to sort a list.

Logic: If we are given n elements, then in the first pass, it will do n-1 comparisons; in the second pass, it will do n-2; in the third pass, it will do n-3 and so on. Thus, the total number of comparisons can be found by;

Output;

(n-1) + (n-2) + (n-3) + (n-4) + ... + 1

$$Sum = \frac{n(n-1)}{2}$$

i.e., $O(n^2)$

Therefore, the bubble sort algorithm encompasses a time complexity of **O(n^2)** and a space complexity of **O(1)** because it necessitates some extra memory space for temp variable for swapping.

3.2.2 Time Complexities:

- **Best Case Complexity**: The bubble sort algorithm has a best-case time complexity of **O(n)** for the already sorted array.
- **Average Case Complexity**: The average-case time complexity for the bubble sort algorithm is **O(n^2)**, which happens when 2 or more elements are in jumbled, i.e., neither in the ascending order nor in the descending order.
- **Worst Case Complexity**: The worst-case time complexity is also **O(n^2)**, which occurs when we sort the descending order of an array into the ascending order.

3.2.3 Advantages of Bubble Sort

1. Easily understandable.
2. Does not necessitate any extra memory.
3. The code can be written easily for this algorithm.
4. Minimal space requirement than that of other sorting algorithms.

3.2.4 Disadvantages of Bubble Sort

1. It does not work well when we have large unsorted lists, and it necessitates more resources that end up taking so much of time.
2. It is only meant for academic purposes, not for practical implementations.
3. It involves the n^2 order of steps to sort an algorithm.

Self-Assessment Exercise

1. What exactly do we mean by the concept of “Sorting”
2. Explain the terms “Sorting in Ascending order” and “Sorting in Descending order”.

3. Why do we prefer using the Bubble sort algorithm in teaching Sorting and in sorting small list of numbers?

3.3 Selection Sort Algorithm

The selection sort enhances the bubble sort by making only a single swap for each pass through the rundown. In order to do this, a selection sort searches for the biggest value as it makes a pass and, after finishing the pass, places it in the best possible area. Similarly, as with a bubble sort, after the first pass, the biggest item is in the right place. After the second pass, the following biggest is set up. This procedure proceeds and requires n-1 goes to sort n item since the last item must be set up after the (n-1) th pass.

3.3.1 Algorithm: Selection Sort (A)

```
k ← length [A]
for j ←1 to n-1
smallest ←  j
for I ← j + 1 to k
if A [i] < A [ smallest]
then smallest ←  i
exchange (A [j], A [smallest])
```

3.3.2 How Selection Sort works

1. In the selection sort, first of all, we set the initial element as a **minimum**.
2. Now we will compare the minimum with the second element. If the second element turns out to be smaller than the minimum, we will swap them, followed by assigning to a minimum to the third element.
3. Else if the second element is greater than the minimum, which is our first element, then we will do nothing and move on to the third element and then compare it with the minimum. We will repeat this process until we reach the last element.
4. After the completion of each iteration, we will notice that our minimum has reached the start of the unsorted list.

5. For each iteration, we will start the indexing from the first element of the unsorted list. We will repeat the Steps from 1 to 4 until the list gets sorted or all the elements get correctly positioned.
6. Consider the following example of an unsorted array that we will sort with the help of the Selection Sort algorithm.

   A[] = (7, 4, 3, 6, 5).
   A [] =

1st Iteration:

Set minimum = 7

- Compare a_0 and a_1

As, $a_0 > a_1$, set minimum = 4.

- Compare a_1 and a_2

As, $a_1 > a_2$, set minimum = 3.

- Compare a_2 and a_3

As, $a_2 < a_3$, set minimum= 3.

- Compare a_2 and a_4

As, $a_2 < a_4$, set minimum =3.

Since 3 is the smallest element, so we will swap a_0 and a_2.

2nd Iteration:

Set minimum = 4

- Compare a_1 and a_2

As, $a_1 < a_2$, set minimum = 4.

- Compare a_1 and a_3

As, A[1] < A[3], set minimum = 4.

- Compare a_1 and a_4

Again, $a_1 < a_4$, set minimum = 4.

Since the minimum is already placed in the correct position, so there will be no swapping.

3rd Iteration:

Set minimum = 7

- Compare a_2 and a_3

As, $a_2 > a_3$, set minimum = 6.

- Compare a_3 and a_4

As, $a_3 > a_4$, set minimum = 5.

Since 5 is the smallest element among the leftover unsorted elements, so we will swap 7 and 5.

4th Iteration:

Set minimum = 6

- Compare a_3 and a_4

As $a_3 < a_4$, set minimum = 6.

Since the minimum is already placed in the correct position, so there will be no swapping.

3.3.3 Complexity Analysis of Selection Sort

Input: Given **n** input elements.

Output: Number of steps incurred to sort a list.

Logic: If we are given n elements, then in the first pass, it will do **n-1** comparisons; in the second pass, it will do **n-2**; in the third pass, it will do **n-3** and so on. Thus, the total number of comparisons can be found by;

Output;

(n-1) + (n-2) + (n-3) + (n-4) + ... + 1

$$Sum = \frac{n(n-1)}{2}$$

i.e., $O(n^2)$

Therefore, the selection sort algorithm encompasses a time complexity of **$O(n^2)$** and a space complexity of **O(1)** because it necessitates some extra memory space for temp variable for swapping.

3.3.4 Time Complexities:

- **Best Case Complexity:** The selection sort algorithm has a best-case time complexity of **$O(n^2)$** for the already sorted array.
- **Average Case Complexity:** The average-case time complexity for the selection sort algorithm is **$O(n^2)$**, in which the existing elements are in jumbled ordered, i.e., neither in the ascending order nor in the descending order.
- **Worst Case Complexity:** The worst-case time complexity is also **$O(n^2)$**, which occurs when we sort the descending order of an array into the ascending order.

3.3.5 Advantages of Selection Sort

- It is an in-place algorithm. It does not require a lot of space for sorting. Only one extra space is required for holding the temporal variable.
- It performs well on items that have already been sorted

3.3.6 Disadvantage of selection sort

- As the input size increases, the performance of selection sort decreases.

Self-Assessment Exercise

1. How does the Selection sort algorithm work?

2. What is the Average case and Worst case complexity of the Selection Sort algorithm?

4.0 Conclusion

The sorting problem enables us to find better algorithms that would help arrange the numbers in a list or sequence in any order. Ascending order is when it is arranged from Smallest to Biggest while Descending order is when the list is arranged from biggest item to the smallest item. We looked at the case of the bubble sort and the Selection sort algorithms which are well suited for sorting a small-sized list efficiently.

5.0 Summary

In simple terms, the Sorting algorithm arranges a list from either smallest item consecutively to the biggest item (Ascending order) or from the biggest item consecutively to the smallest item (Descending order).

Two methods of Sorting small-sized lists (Bubble sort and Selection Sort) were introduced and incidentally, they both have the same Worst case runnung time of $O(n^2)$.

6.0 Tutor Marked Assignment

1. Sort the following list [76, 23, 65, 2, 8, 43, 88, 2, 4, 7, 23, 8, 65] in ascending order using Selection Sort.
2. Sort the list given in Question 1 above in descending order using Bubble sort?
3. What are two benefits each of Bubble Sort and Selection Sort algorithms?

7.0 Further Reading and other Resources

Baase, S. and Van Gelder, A. (2008). *Computer Algorithms: Introduction to Design and Analysis*, Pearson Education.

Jena, S. R. and Patro, S. (2018) – Design and Analysis of Algorithms, ISBN 978-93-935274-311-7

Karumanchi, N. (2016). Data Structures and Algorithms, CareerMonk Publications. ISBN-13 : 978-8193245279

Module 2: Sorting and Searching Algorithms
Unit 2: Insertion Sort and Linear Search Algorithm

1.0 Introduction

Insertion sort is one of the simplest sorting algorithms for the reason that it sorts a single element at a particular instance. It is not the best sorting algorithm in terms of performance, but it's slightly more efficient than selection sort and bubble sort in practical scenarios. It is an intuitive sorting technique.

2.0 Objectives

By the end of this unit, you will be able to:

- Know how Insertion sort and Linear search works
- Understand the complexities of both Linear search and Insertion sort
- Know the advantages and disadvantages of Linear search
- Know the advantages and disadvantages of Insertion sort
- Use the Linear Search and Insertion sort algorithms to write good programs in any programming language of your choice.

3.0 Insertion Sort

Insertion sort is one of the simplest sorting algorithms for the reason that it sorts a single element at a particular instance. It is not the best sorting algorithm in terms of performance, but it's slightly more efficient than selection sort and bubble sort in practical scenarios. It is an intuitive sorting technique.

Let's consider the example of cards to have a better understanding of the logic behind the insertion sort.

Suppose we have a set of cards in our hand, such that we want to arrange these cards in ascending order. To sort these cards, we have a number of intuitive ways.

One such thing we can do is initially we can hold all of the cards in our left hand, and we can start taking cards one after other from the left hand, followed by building a sorted arrangement in the right hand.

Assuming the first card to be already sorted, we will select the next unsorted card. If the unsorted card is found to be greater than the selected card, we will simply place it on the right side, else to the left side. At any stage during this whole process, the left hand will be unsorted, and the right hand will be sorted.

In the same way, we will sort the rest of the unsorted cards by placing them in the correct position. At each iteration, the insertion algorithm places an unsorted element at its right place.

Algorithm: Insertion Sort (A)

```
1. for j = 2 to A.length

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
6.      A[i + 1] = A[i]
7.      ii = i -1
8.  A[i + 1] = key
```

3.1 How Insertion Sort Works

1. We will start by assuming the very first element of the array is already sorted. Inside the **key**, we will store the second element.

Next, we will compare our first element with the **key**, such that if **the key** is found to be smaller than the first element, we will interchange their indexes or place the key at the first index. After doing this, we will notice that the first two elements are sorted.

2. Now, we will move on to the third element and compare it with the left-hand side elements. If it is the smallest element, then we will place the third element at the first index.

Else if it is greater than the first element and smaller than the second element, then we will interchange its position with the third element and place it after the first element. After doing this, we will have our first three elements in a sorted manner.

3. Similarly, we will sort the rest of the elements and place them in their correct position.

Consider the following example of an unsorted array that we will sort with the help of the Insertion Sort algorithm.

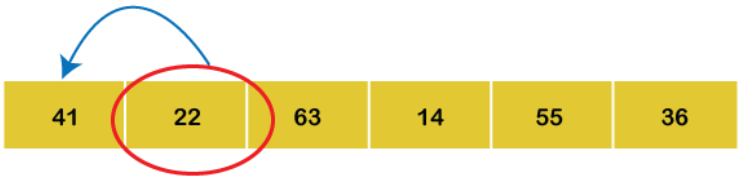
A = (41, 22, 63, 14, 55, 36)

Initially,

1st Iteration:

Set key = 22

Compare a1 with a0

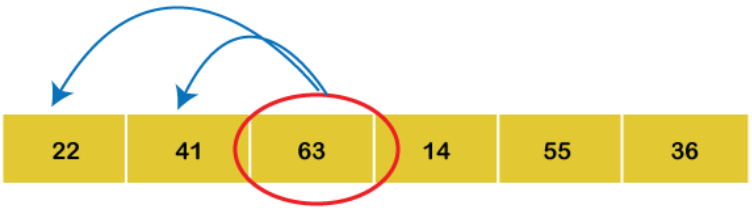


Since a0 > a1, swap both of them.

2nd Iteration:

Set key = 63

Compare a2 with a1 and a0



Since a2 > a1 > a0, keep the array as it is.

3rd Iteration:

Set key = 14

Compare a3 with a2, a1 and a0

Since a3 is the smallest among all the elements on the left-hand side, place a3 at the beginning of the array.

4th Iteration:

Set key = 55

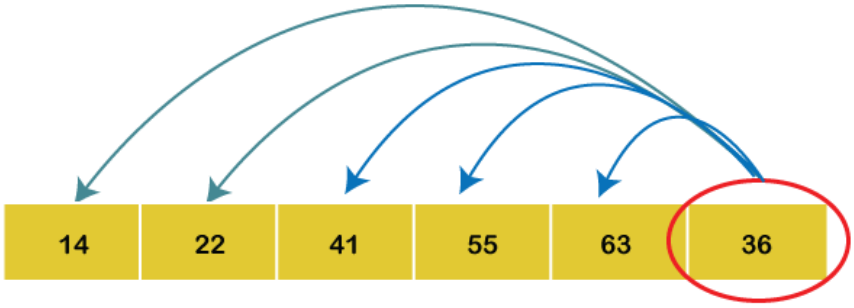
Compare a4 with a3, a2, a1 and a0.

As a4 < a3, swap both of them.

5th Iteration:

Set key = 36

Compare a5 with a4, a3, a2, a1 and a0.



Since a5 < a2, so we will place the elements in their correct positions.

Hence the array is arranged in ascending order, so no more swapping is required.

3.2 Complexity Analysis of Insertion Sort

Input: Given n input elements.

Output: Number of steps incurred to sort a list.

Logic: If we are given **n** elements, then in the first pass, it will make **n-1** comparisons; in the second pass, it will do **n-2**; in the third pass, it will do **n-3** and so on. Thus, the total number of comparisons can be found by;

Output:
(n-1) + (n-2) + (n-3) + (n-4) = …… +1

Sum = i.e…. $O(n^2)$

Therefore, the insertion sort algorithm encompasses a time complexity of **$O(n^2)$** and a space complexity of **O(1)** because it necessitates some extra memory space for a **key** variable to perform swaps.

3.2.1 Time Complexities:

- **Best Case Complexity:** The insertion sort algorithm has a best-case time complexity of **O(n)** for the already sorted array because here, only the outer loop is running n times, and the inner loop is kept still.
- **Average Case Complexity:** The average-case time complexity for the insertion sort algorithm is **$O(n^2)$**, which is incurred when the existing elements are in jumbled order, i.e., neither in the ascending order nor in the descending order.
- **Worst Case Complexity:** The worst-case time complexity is also **$O(n^2)$**, which occurs when we sort the ascending order of an array into the descending order.
  In this algorithm, every individual element is compared with the rest of the elements, due to which n-1 comparisons are made for every n^{th} element.

The insertion sort algorithm is highly recommended, especially when a few elements are left for sorting or in case the array encompasses few elements.

3.2.2 Space Complexity

The insertion sort encompasses a space complexity of **O(1)** due to the usage of an extra variable **key**.

3.2.3 Insertion Sort Applications

The insertion sort algorithm is used in the following cases:

- When the array contains only a few elements.
- When there exist few elements to sort.

3.2.4 Advantages of Insertion Sort

1. It is simple to implement.
2. It is efficient on small datasets.
3. It is stable (does not change the relative order of elements with equal keys)

4. It is in-place (only requires a constant amount O (1) of extra memory space).
5. It is an online algorithm, which can sort a list when it is received.

3.2.5 Disadvantages of Insertion Sort

1. Insertion sort is inefficient against more extensive data sets.
2. The insertion sort exhibits the worst-case time complexity of O(n2)
3. It does not perform well than other, more advanced sorting algorithms

Self-Assessment Exercise:

1. What is the worst case time complexity of insertion sort where position of the data to be inserted is calculated using binary search?
2. Consider an array of elements arr[5]= {5,4,3,2,1} , what are the steps of insertions done while doing insertion sort in the array.
3. How many passes does an insertion sort algorithm consist of?
4. What is the average case running time of an insertion sort algorithm?
5. What is the running time of an insertion sort algorithm if the input is pre-sorted?

3.3 LINEAR SEARCH

A linear search is the simplest method of searching a data set.

Starting at the beginning of the data set, each item of data is examined until a match is made. Once the item is found, the search ends.

A way to describe a linear search would be:

1. Find out the length of the data set.
2. Set counter to 0.
3. Examine value held in the list at the counter position.
4. Check to see if the value at that position matches the value searched for.
5. If it matches, the value is found. End the search.
6. If not, increment the counter by 1 and go back to step 3 until there are no more items to search.

Consider this list of unordered numbers:

Position in data set	0	1	2	3	4	5	6	7
Data value	3	5	2	9	6	1	8	7

Suppose we were to search for the value 2. The search would start at position 0 and check the value held there, in this case 3.

3 does not match 2, so we move on to the next position.

The value at position 1 is 5.

5 does not match 2, so we move on to the next position.

The value at position 2 is 2 - a match. The search ends.

A linear search in pseudocode might look like this:

```
find = 2
found = False
length = list.length
counter = 0
while found == False and counter < length
     if list[counter] == find then found = True
     print ('Found at position', counter)
else:
     counter = counter + 1
     endif
endwhile
     if found == False then
     print('Item not found')
     endif
```

A linear search, although simple, can be quite inefficient. Suppose the data set contained 100 items of data, and the item searched for happens to be the last item in the set? All of the previous 99 items would have to be searched through first.

However, linear searches have the advantage that they will work on any data set, whether it is ordered or unordered.

3.4 Complexity of Linear Search

The worst case complexity of linear search is **O(n)**.

If the element to be searched lived on the the first memory block then the best case complexity would be: O(1).

3.4.1 Advantages of Linear Search

a. Will perform fast searches of small to medium lists. With today's powerful computers, small to medium arrays can be searched relatively quickly.

b. The list does not need to sorted. ...

c. Not affected by insertions and deletions.

3.4.2 Disadvantages of Linear Search

a. It is less efficient in the case of large-size data sets.
b. The worst- case scenario for finding the element is O(n).

Self-Assessment Exercises

1. Given a list of numbers 12, 45, 23, 7, 9, 10, 22, 87, 45, 23, 34, 56
   a. Use the linear search algorithm to search for the number 10
   b. Comment on the worst-case running time of your algorithm
2. When do we consider the linear search algorithm a better alternative?
3. What is the best case for linear search?

4.0 Conclusion

The Insertion sort is a simple sorting algorithm that builds the final sorted array one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, or merge sort while a linear search or sequential search is a method for finding an element within a list. It sequentially checks each element of the list until a match is found or the whole list has been searched

5.0 Summary

We examined the Insertion sort algorithm and how it can be used to sort or arrange a list in any order while at the same time noting its complexity, advantages and disadvantages. A Linear Search algorithm which is also known as Sequential search is used in finding a given element in a list and returns a

positive answer once the element is located else it returns a negative answer. Linear search is very efficient for searching an item within a small-sized list'

6.0 Tutor Marked Assignment

1. How many linear searches will it take to find the value 7 in the list [1,4,8,7,10,28]?
2. Consider the following lists of partially sorted numbers. The double bars represent the sort marker. How many comparisons and swaps are needed to sort the next number. [**1 3 4 8 9 || 5 2**] using Insertion sort?
3. What is an advantage of the Linear search algorithm?
4. If all the elements in an input array is equal for example {1,1,1,1,1,1}, what would be the running time of the Insertion sort Algorithm?
5. For linear search, describe the "worst case scenario" and the "best case scenario.

7.0 Further Reading and Other Resources

Berman, K. and Paul,J. (2004). *Algorithms: Sequential, Parallel, and Distributed*. Course Technology.

Cormen, T. H., Leiserson,C, Rivest,R. and Stein,C. (2009). *Introduction to Algorithms*. Third Edition. MIT Press.

Trivedi, K. S.(2001). *Probability and Statistics with Reliability, Queueing, and Computer Science Applications*. Second Edition. Wiley-Blackwell Publishing.

Module 2: Searching and Sorting algorithms

Unit 3: Radix Sort and Stability in Sorting

1.0 Introduction

Radix sort is one of the simplest sorting algorithms for the reason that it sorts a single element at a particular instance. It is not the best sorting algorithm in terms of performance, but it's slightly more efficient than selection sort and bubble sort in practical scenarios. It is an intuitive sorting technique.

2.0 Objectives

By the end of this unit, you will be able to:

- Know how to calculate with various data types
- Specify input and output statements
- Differentiate between formatted and unformatted I/O statements.

3.0 Radix Sort

Radix Sort is a Sorting algorithm that is useful when there is a constant 'd' such that all keys are d digit numbers. To execute Radix Sort, for p =1 towards 'd' sort the numbers with respect to the Pth digits from the right using any linear time stable sort.

Radix sort is a sorting technique that sorts the elements digit to digit based on radix. It works on integer numbers. To sort the elements of the string type, we can use their hash value. This sorting algorithm makes no comparison.

The Code for Radix Sort is straightforward. The following procedure assumes that each element in the n-element array A has d digits, where digit 1 is the lowest order digit and digit d is the highest-order digit.

Here is the algorithm that sorts A [1.n] where each number is d digits long.

```
Radix-Sort (array A, int n, int d)
 1 for i ← 1 to d
 2 do stably sort A to sort array A on digit i
```

Example: The first Column is the input. The remaining Column shows the list after successive sorts on increasingly significant digit position. The vertical

arrows indicate the digits position sorted on to produce each list from the previous one.

```
576      49[4]      9[5]4      [1]76      176
494      19[4]      5[7]6      [1]94      194
194      95[4]      1[7]6      [2]78      278
296   → 57[6]   →  2[7]8   → [2]96   → 296
278      29[6]      4[9]4      [4]94      494
176      17[6]      1[9]4      [5]76      576
954      27[8]      2[9]6      [9]54      954
```

3.1 Complexity of the Radix sort algorithm

Worst case time complexity

The worst case in radix sort occurs when all elements have the same number of digits except one element which has significantly large number of digits. If the number of digits in the largest element is equal to n, then the runtime becomes $O(n^2)$. The worst case running time of Counting sort is O(n+b). If b=O(n), then the worst case running time is O(n). Here,the countingSort function is called for d times, where d = ⌊logb(mx)+1⌋⌊logb(mx)+1⌋.
Total worst case complexity of radix sort is $O(\log_b(mx)(n+b))$.

Best case time complexity

The best case occurs when all elements have the same number of digits. The best case time complexity is O(d(n+b)). If b = O(n), then time complexity is O(dn).

Average case time complexity

In the average case, we have considered the distribution of the number of digits. There are **D** passes and each digit can take on up to **b** possible values. Radix sort doesn't depend on the input sequence, so we may keep n as a constant. The running time of radix sort is, T(n) = d(n+b). Taking expectations of both sides and using linearity of expectation,

The average case time complexity of radix sort is O(D*(n+b)).

Space Complexity

In this algorithm, we have two auxiliary arrays **cnt** of size b (base) and **tempArray** of size n (number of elements), and an input array **arr** of size n.

Space complexity: O(n+b)

The base of the radix sort doesn't depend upon the number of elements. In some cases, the base may be larger than the number of elements.

Radix sort becomes slow when the element size is large but the radix is small. We can't always use a large radix cause it requires large memory in counting sort. It is good to use the radix sort when d is small.

3.1.1 Advantages of Radix Sort:

- Fast when the keys are short i.e. when the range of the array elements is less.
- Used in suffix array construction algorithms like Manber's algorithm and DC3 algorithm.
- Radix Sort is stable sort as relative order of elements with equal values is maintained.

3.1.2 Disadvantages of Radix Sort:

- Since Radix Sort depends on digits or letters, Radix Sort is much less flexible than other sorts. ...
- The constant for Radix sort is greater compared to other sorting algorithms.
- It takes more space compared to Quicksort which is in-place sorting.

3.1.3 Applications of Radix Sort

Here are a few applications of the radix sort algorithm:

- Radix sort can be applied to data that can be sorted lexicographically, such as words and integers. It is also used for stably sorting strings.
- It is a good option when the algorithm runs on parallel machines, making the sorting faster. To use parallelization, we divide the input into several

buckets, enabling us to sort the buckets in parallel, as they are independent of each other.

- It is used for constructing a suffix array. (An array that contains all the possible suffixes of a string in sorted order is called a suffix array.

Self-Assessment Exercises

1. If we use Radix Sort to sort n integers in the range $(n^{k/2},n^k]$, for some k>0 which is independent of n, the time taken would be?
2. The maximum number of comparisons needed to sort 9 items using radix sort is? (assume each item is 5 digit octal number):
3. Sort the following list in descending order using the Radix sort algorithm

3.2 Stability in Sorting

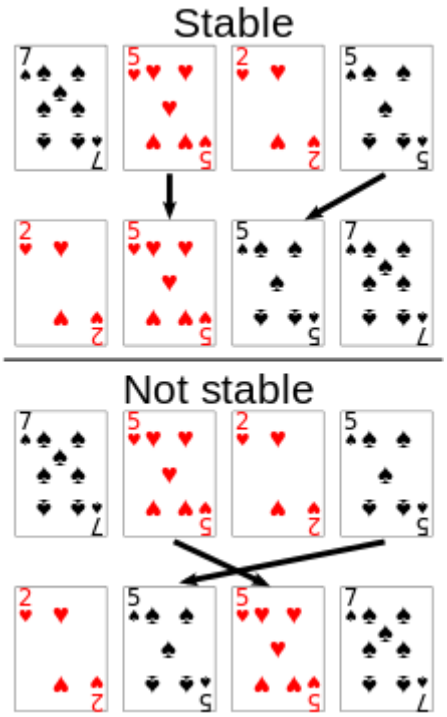
Stable sort algorithms sort equal elements in the same order that they appear in the input. For example, in the card sorting example to the right, the cards are being sorted by their rank, and their suit is being ignored. This allows the possibility of multiple different correctly sorted versions of the original list. Stable sorting algorithms choose one of these, according to the following rule: if two items compare as equal (like the two 5 cards), then their relative order will be preserved, i.e. if one comes before the other in the input, it will come before the other in the output.

Stability is important to preserve order over multiple sorts on the same data set. For example, say that student records consisting of name and class section are sorted dynamically, first by name, then by class section. If a stable sorting algorithm is used in both cases, the sort-by-class-section operation will not change the name order; with an unstable sort, it could be that sorting by section shuffles the name order, resulting in a nonalphabetical list of students.

More formally, the data being sorted can be represented as a record or tuple of values, and the part of the data that is used for sorting is called the *key*. In the card example, cards are represented as a record (rank, suit), and the key is the rank. A sorting algorithm is stable if whenever there are two records R and S with the

same key, and R appears before S in the original list, then R will always appear before S in the sorted list.

When equal elements are indistinguishable, such as with integers, or more generally, any data where the entire element is the key, stability is not an issue. Stability is also not an issue if all keys are different.

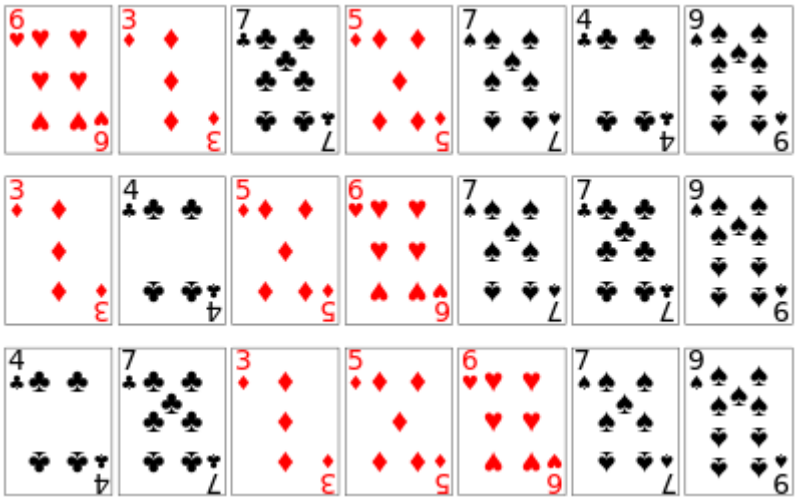


An example of stable sort on playing cards. When the cards are sorted by rank with a stable sort, the two 5s must remain in the same order in the sorted output that they were originally in. When they are sorted with a non-stable sort, the 5s may end up in the opposite order in the sorted output.

Unstable sorting algorithms can be specially implemented to be stable. One way of doing this is to artificially extend the key comparison, so that comparisons between two objects with otherwise equal keys are decided using the order of the entries in the original input list as a tie-breaker. Remembering this order, however, may require additional time and space.

One application for stable sorting algorithms is sorting a list using a primary and secondary key. For example, suppose we wish to sort a hand of cards such that

the suits are in the order clubs (♣), diamonds (♦), hearts (♥), spades (♠), and within each suit, the cards are sorted by rank. This can be done by first sorting the cards by rank (using any sort), and then doing a stable sort by suit:



Within each suit, the stable sort preserves the ordering by rank that was already done. This idea can be extended to any number of keys and is utilized by radix sort. The same effect can be achieved with an unstable sort by using a lexicographic key comparison, which, e.g., compares first by suit, and then compares by rank if the suits are the same.

3.2.1 Why is stable sort useful?

A stable sorting algorithm **maintains the relative order of the items with equal sort keys**. An unstable sorting algorithm does not. In other words, when a collection is sorted with a stable sorting algorithm, items with the same sort keys preserve their order after the collection is sorted.
Suppose you need to sort following key-value pairs in the increasing order of keys:

INPUT: (4,5), (3, 2) (4, 3) (5,4) (6,4)

Now, there is two possible solution for the two pairs where the key is same i.e. (4,5) and (4,3) as shown below:

OUTPUT1: (3, 2), (4, 5), (4,3), (5,4), (6,4)
OUTPUT2: (3, 2), (4, 3), (4,5), (5,4), (6,4)

The sorting algorithm which will produce the first output will be known as stable sorting algorithm because the original order of equal keys are maintained, you can see that (4, 5) comes before (4,3) in the sorted order, which was the original order i.e. in the given input, (4, 5) comes before (4,3) .
On the other hand, the algorithm which produces second output will know as an unstable sorting algorithm because the order of objects with the same key is not maintained in the sorted order. You can see that in the second output, the (4,3) comes before (4,5) which was not the case in the original input.

Self-Assessment Exercise

1. Can any unstable sorting algorithm be altered to become stable? If so, how?
2. What is the use of differentiating algorithms on the basis of stability?
3. When is it definitely unnecessary to look at the nature of stability of a sorting algorithm?
4. What are some stable sorting techniques?
5. What properties of sorting algorithms are most likely to get affected when a typically unstable sorting algorithm is implemented to be stable?

4.0 Conclusion

In computer science, radix sort is a non-comparative sorting algorithm. It avoids comparison by creating and distributing elements into buckets according to their radix. Stable sorting algorithms on the other hand maintain the relative order of records with equal keys (i.e. values). That is, a sorting algorithm is stable if whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the sorted list.

5.0 Summary

We considered another good example of a sorting algorithm known as Radix sort which unconsciously, is the commonest method we use in sorting some items in a list. On the other hand, we looked at stability in sorting algorithms and how to identify stable and unstable sorting algorithms.

6.0 Tutor Marked Assignment

1. In what cases should we prefer using stable sorting algorithms?

2. Assuming that the number of digits used is not excessive, the worst-case cost for Radix Sort when sorting n*n* keys with distinct key values is:
3. If an unstable sorting algorithm happens to preserve the relative order in a particular example, is it said to be stable?
4. The running time of radix sort on an array of n integers in the range [0……..n^5 -1] when using base 10 representation is?
5. How can you convert an unstable sorting algorithm into a stable sorting algorithm?

7.0 Further Reading and other Resources

Dave, P. H. and Dave, H. B. (2008). Design and Analysis of Algorithms, Pearson Education.

Jena, S. R. and Swain, S. K, (2017). *Theory of Computation and Application*, 1st Edition, University Science Press, Laxmi Publications.

Levitin, A. (2012). Introduction to the Design and Analysis of Algorithms, 3rd Ed. Pearson Education, ISBN 10-0132316811

Michalewicz, Z. and Fogel, D. (2004). *How to Solve It: Modern Heuristics*. Second Edition. Springer.

Module 2: Sorting and Searching Algorithms

Unit 4: Divide and Conquer Strategies I: Binary Search Algorithm

1.0 Introduction

Divide-and-Conquer is a useful problem-solving technique that divides a large instance of a problem sixe into smaller and smaller instances and then solves these smaller instances to give a complete solution of the bigger problem. There are several strategies for implementing the Divide-and-Conquer approach and we shall first examine the Binary Search algorithm which first requires that a list be sorted and then proceeds to find any requested item on the list and is very efficient for large lists since it uses logarithmic time.

2.0 Objectives

By the end of this unit, you should be able to:

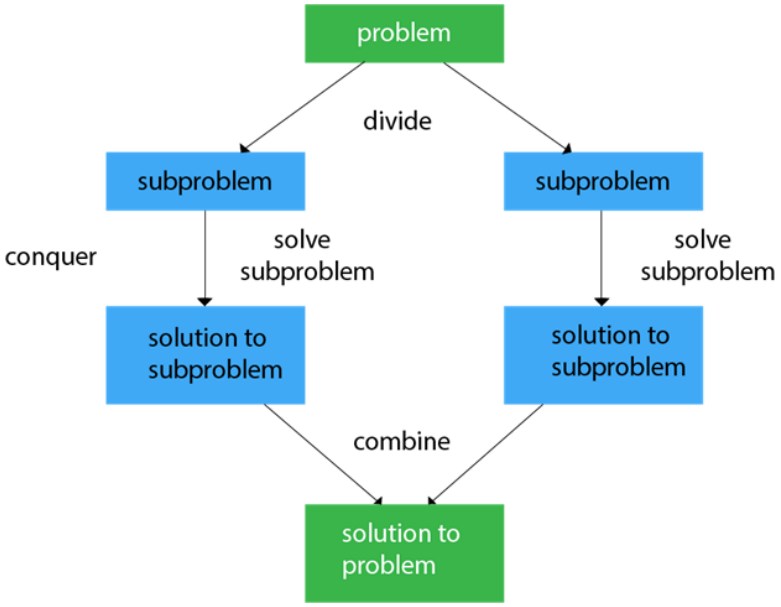
- Know the meaning of a Divide-and-Conquer Algorithm
- Know how to use a Divide-and-Conquer algorithm
- Know the different applications of Divide-and-Conquer algorithms
- Understand the Binary Search algorithm,
- Know why the Binary Search algorithm is useful
- Understand the benefits and shortcomings of Binary search
- Know the different application areas of Binary Search

3.0 Divide and Conquer Algorithms

Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy.

Divide and Conquer algorithm consists of a dispute using the following three steps.

1. **Divide** the original problem into a set of subproblems.
2. **Conquer:** Solve every subproblem individually, recursively.
3. **Combine:** Put together the solutions of the subproblems to get the solution to the whole problem.



Generally, we can follow the **divide-and-conquer** approach in a three-step process.

Examples: The specific computer algorithms are based on the Divide & Conquer approach:

1. Maximum and Minimum Problem
2. Binary Search
3. Sorting (merge sort, quick sort)
4. Tower of Hanoi.

3.1 Fundamental of Divide & Conquer Strategy:

There are two fundamental of Divide & Conquer Strategy:

1. Relational Formula
2. Stopping Condition

1. Relational Formula: It is the formula that we generate from the given technique. After generation of Formula we apply D&C Strategy, i.e. we break the problem recursively & solve the broken subproblems.

2. Stopping Condition: When we break the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So the condition where the need to stop our recursion steps of Divide & Conquer is called as Stopping Condition.

3.1.1 Applications of Divide and Conquer Approach:

Following algorithms are based on the concept of the Divide and Conquer Technique:

1. **Binary Search:** The binary search algorithm is a searching algorithm, which is also called a half-interval search or logarithmic search. It works by comparing the target value with the middle element existing in a sorted array. After making the comparison, if the value differs, then the half that cannot contain the target will eventually eliminate, followed by continuing the search on the other half. We will again consider the middle element and compare it with the target value. The process keeps on repeating until the target value is met. If we found the other half to be empty after ending the search, then it can be concluded that the target is not present in the array.

2. **Quicksort:** It is the most efficient sorting algorithm, which is also known as partition-exchange sort. It starts by selecting a pivot value from an array followed by dividing the rest of the array elements into two sub-arrays. The partition is made by comparing each of the elements with the pivot value. It compares whether the element holds a greater value or lesser value than the pivot and then sort the arrays recursively.

3. **Merge Sort:** It is a sorting algorithm that sorts an array by making comparisons. It starts by dividing an array into sub-array and then recursively sorts each of them. After the sorting is done, it merges them back.
4. **Closest Pair of Points:** It is a problem of computational geometry. This algorithm emphasizes finding out the closest pair of points in a metric

space, given n points, such that the distance between the pair of points should be minimal.

5. **Strassen's Algorithm:** It is an algorithm for matrix multiplication, which is named after Volker Strassen. It has proven to be much faster than the traditional algorithm when works on large matrices.

6. **Cooley-Tukey Fast Fourier Transform (FFT) algorithm:** The Fast Fourier Transform algorithm is named after J. W. Cooley and John Turkey. It follows the Divide and Conquer Approach and imposes a complexity of O(nlogn).

7. **Karatsuba algorithm for fast multiplication:** It is one of the fastest multiplication algorithms of the traditional time, invented by Anatoly Karatsuba in late 1960 and got published in 1962. It multiplies two n-digit numbers in such a way by reducing it to at most single-digit.

3.1.2 Advantages of Divide and Conquer

a. Divide and Conquer tend to successfully solve one of the biggest problems, such as the Tower of Hanoi, a mathematical puzzle. It is challenging to solve complicated problems for which you have no basic idea, but with the help of the divide and conquer approach, it has lessened the effort as it works on dividing the main problem into two halves and then solve them recursively. This algorithm is much faster than other algorithms.
b. It efficiently uses cache memory without occupying much space because it solves simple subproblems within the cache memory instead of accessing the slower main memory.
c. It is more proficient than that of its counterpart Brute Force technique.
d. Since these algorithms inhibit parallelism, it does not involve any modification and is handled by systems incorporating parallel processing.

3.1.3 Disadvantages of Divide and Conquer

a. Since most of its algorithms are designed by incorporating recursion, so it necessitates high memory management.
b. An explicit stack may overuse the space.
c. It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU.

3.1.4 Properties of Divide-and-Conquer Algorithms

Divide-and-Conquer has several important properties.

a. It follows the structure of an inductive proof, and therefore usually leads to relatively simple proofs of correctness. To prove a divide-and-conquer algorithm correct, we first prove that the base case is correct. Then, we assume by strong (or structural) induction that the recursive solutions are correct, and show that, given correct solutions to smaller instances, the combined solution is correct.
b. Divide-and-conquer algorithms can be work efficient. To ensure efficiency, we need to make sure that the divide and combine steps are efficient, and that they do not create too many sub-instances.
c. The work and span for a divide-and-conquer algorithm can be expressed as a mathematical equation called recurrence, which can be usually be solved without too much difficulty.
d. Divide-and-conquer algorithms are naturally parallel, because the sub-instances can be solved in parallel. This can lead to significant amount of parallelism, because each inductive step can create more independent instances. For example, even if the algorithm divides the problem instance into two subinstances, each of those subinstances could themselves generate two more subinstances, leading to a geometric progression, which can quickly produce abundant parallelism.

Self-Assessment Exercise

1. The steps in the Divide-and-Conquer process that takes a recursive approach is said to be?
2. Given the recurrence f(n) = 4 f(n/2) + 1, how many sub-problems will a divide-and-conquer algorithm divide the original problem into, and what will be the size of those sub-problems?
3. Design a divide-and-conquer algorithm to compute k^n for $k > 0$ and integer $n >= 0$.
4. Define divide and conquer approach to algorithm design

3.2 BINARY SEARCH

In computer science, binary search, also known as half-interval search, logarithmic search, or binary chop, is a search algorithm that finds the position of a target value within a sorted array. Binary search compares the target value to the middle element of the array.

A binary search is an efficient method of searching an ordered list. A binary search works like this:

1. Start by setting the counter to the middle position in the list.
2. If the value held there is a match, the search ends.
3. If the value at the midpoint is less than the value to be found, the list is divided in half. The lower half of the list is ignored and the search keeps to the upper half of the list.
4. Otherwise, if the value at the midpoint is greater than the value to be found, the upper half of the list is ignored and the search keeps to the lower half of the list.
5. The search moves to the midpoint of the remaining items. Steps 2 through 4 continue until a match is made or there are no more items to be found.

Consider this list of ordered numbers:

Position in data set	0	1	2	3	4	5	6	7	8
Data value	1	3	4	5	7	9	11	14	16

Suppose we were to search for the value 11.

The midpoint is found by adding the lowest position to the highest position and dividing by 2.

Highest position (8) + lowest position (0) = 8

8/2 = 4

NOTE - if the answer is a decimal, round up. For example, 3.5 becomes 4. We can round down as an alternative, as long as we are consistent.

Check at position 4, which has the value 7.

7 is less than 11, so the bottom half of the list (including the midpoint) is discarded.

Position in data set	0	1	2	3	4	5	6	7	8
Data value	1	3	4	5	7	9	11	14	16

The new lowest position is 5.

Highest position (8) + lowest position (5) = 13

13/2 = 6.5, which rounds up to 7

Check at position 7, which has the value 14.

14 is greater than 11, so the top half of the list (including the midpoint) is discarded.

Position in data set	0	1	2	3	4	5	6	7	8
Data value	1	3	4	5	7	9	11	14	16

The new highest position is 6.

Highest position (6) + lowest position (5) = 11

11/2 = 5.5, which rounds up to 6

Check at position 6.

The value held at position 6 is 11, a match. The search ends.

A binary search in pseudocode might look like this:

```
find = 11
found = False
length = list.length
lowerBound = 0
upperBound = length
while found == False
     midpoint = int((upperBound + lowerBound))/2
     if list[midPoint] == find then
          print('Found at' , midPoint)
     found = True
else
     if list[midPoint]> item then
     upperBound = midpoint-1
else
     lowerBound = midpoint+1
     endif
     endif
endwhile

if found == False then
     print('Not found')
endif
```

A binary search is a much more efficient algorithm than a linear search. In an ordered list of every number from 0 to 100, a linear search would take 99 steps to find the value 99. A binary search would only require seven steps.

However, a binary search can only work if a list is ordered.

3.2.1 Complexity of Binary Search

The time complexity of the **binary search algorithm** is O(log n). The best-case time complexity would be O(1) when the central index would directly match the desired value. The worst-case scenario could be the values at either extremity of the list or values not in the list.

The space complexity of the **binary search algorithm** depends on the implementation of the algorithm. There are two ways of implementing it:

- Iterative method
- Recursive method

Both methods are quite the same, with two differences in implementation. First, there is no loop in the recursive method. Second, rather than passing the new values to the next iteration of the loop, it passes them to the next recursion. In the iterative method, the iterations can be controlled through the looping conditions, while in the recursive method, the maximum and minimum are used as the boundary condition.

In the iterative method, the space complexity would be O(1). While in the recursive method, the space complexity would be O(log n).

3.2.2 Advantages of Binary Search

a. A **binary search algorithm** is a fairly simple search algorithm to implement.
b. It is a significant improvement over linear search and performs almost the same in comparison to some of the harder to implement search algorithms.
c. The **binary search algorithm** breaks the list down in half on every iteration, rather than sequentially combing through the list. On large lists, this method can be really useful.

3.2.3 Disadvantages of Binary Search

a. It employs recursive approach which requires more stack space.
b. Programming binary search algorithm is error prone and difficult.
c. The interaction of binary search with memory hierarchy i.e. caching is poor.

3.2.4 Applications of Binary Search

a. This algorithm is used to search element in a given sorted array with more efficiency.
b. It could also be used for few other additional operations like- to find the smallest element in the array or to find the largest element in the array.

Self-Assessment Exercise

1. Which type of lists or data sets are binary searching algorithms used for?

2. A binary search is to be performed on the list: [3 5 9 10 23]. How many comparisons would it take to find number 9?
3. How many binary searches will it take to find the value 7 in the list [1,4,7,8,10,28]?
4. Given an array arr = {45,77,89,90,94,99,100} and key = 100; What are the mid values(corresponding array elements) generated in the first and second iterations?

4.0 Conclusion

In computer science, divide and conquer is an algorithm design paradigm. A divide-and-conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly.

A binary search algorithm is a widely used algorithm in the computational domain. It is a fat and accurate search algorithm that can work well on both big and small datasets. A binary search algorithm is a simple and reliable algorithm to implement. With time and space analysis, the benefits of using this particular technique are evident.

5.0 Summary

We looked at the meaning of Divide-and-Conquer algorithms and how they work and then considered a very good example of a Divide-and-Conquer algorithm called Binary Search which is very efficient for large lists as its worst case complexity is given in logarithmic time.

6.0 Tutor Marked Assignment

1. Make a brief comparison between Binary Search and Linear Search algorithms.
2. Explain why the complexity of binary search is O(log n)
3. Suppose you have an array, *A*, containing *n* numbers sorted into increasing order. You want to construct a balanced binary tree containing the numbers in *A*. Give a divide-and-conquer algorithm to do so.

4. How many binary searches will it take to find the value 10 in the list [1,4,9,10,11]?

5. Given a real number, x, and a natural number n, x^n can be defined by the following recursive function:

   $x^n = 1$ if n = 0

   $x^n = (x^{n/2})^2$ if $n > 0$ and n is even

   $x^n = x\,(x^{(n-1)/2})^2$ if $n > 0$ and n is odd

   Use this recursion to give a divide-and-conquer algorithm for computing x^n. Explain how your algorithm meets the definition of "divide and conquer."

6. What is the maximum number of comparisons required to find a value in a list of 20 items using a binary search?

7.0 Further Reading and other Resources

Cormen, T. H., Leiserson, C., Rivest, R. and Stein, C. (2009). *Introduction to Algorithms*. Third Edition. MIT Press.

Jena, S. R. and Swain, S. K, (2017). *Theory of Computation and Application*, 1st Edition, University Science Press, Laxmi Publications.

Karumanchi, N. (2016). Data Structures and Algorithms, CareerMonk Publications. ISBN-13 : 978-8193245279

Module 2: Searching and Sorting Algorithms

Unit 5: Merge Sort and Quick Sort Algorithms

1.0 Introduction

We continue with two more examples of Divide-and-Conquer algorithms which incidentally, are sorting algorithms. The Merge sort (also spelled mergesort) is an efficient sorting algorithm that uses **a** divide-and-conquer approach to order elements in an array. It repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.

Like mergesort, Quick Sort (also spelled QuickSort) is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot.

2.0 Objectives

At the end of this unit, you will be able to:

- Understand the Mergesort algorithm
- Know when and where we can apply mergesort
- Understand the complexity of the mergesort approach
- Know the benefits and shortcomings of mergesort
- Know more about the Quicksort algorithm
- Understand how Quicksort works
- Be able to write codes for mergesort and quicksort
- Be able to perform simple sorting of any list using quicksort and mergesort.

3.0 Merge Sort

Merge sort is yet another sorting algorithm that falls under the category of Divide and Conquer technique. It is one of the best sorting techniques that successfully build a recursive algorithm.

Divide and Conquer Strategy

In this technique, we segment a problem into two halves and solve them individually. After finding the solution of each half, we merge them back to represent the solution of the main problem.

Suppose we have an array **A**, such that our main concern will be to sort the subsection, which starts at index **p** and ends at index **r**, represented by **A[p..r]**.

Divide

If assumed **q** to be the central point somewhere in between **p** and **r**, then we will fragment the subarray **A[p..r]** into two arrays **A[p..q]** and **A[q+1, r]**.

Conquer

After splitting the arrays into two halves, the next step is to conquer. In this step, we individually sort both of the subarrays **A[p..q]** and **A[q+1, r]**. In case if we did not reach the base situation, then we again follow the same procedure, i.e., we further segment these subarrays followed by sorting them separately.

Combine

As when the base step is acquired by the conquer step, we successfully get our sorted subarrays **A[p..q]** and **A[q+1, r]**, after which we merge them back to form a new sorted array **[p..r]**.

3.1 Merge Sort algorithm

The MergeSort function keeps on splitting an array into two halves until a condition is met where we try to perform MergeSort on a subarray of size 1, i.e., **p == r**.

And then, it combines the individually sorted subarrays into larger arrays until the whole array is merged.

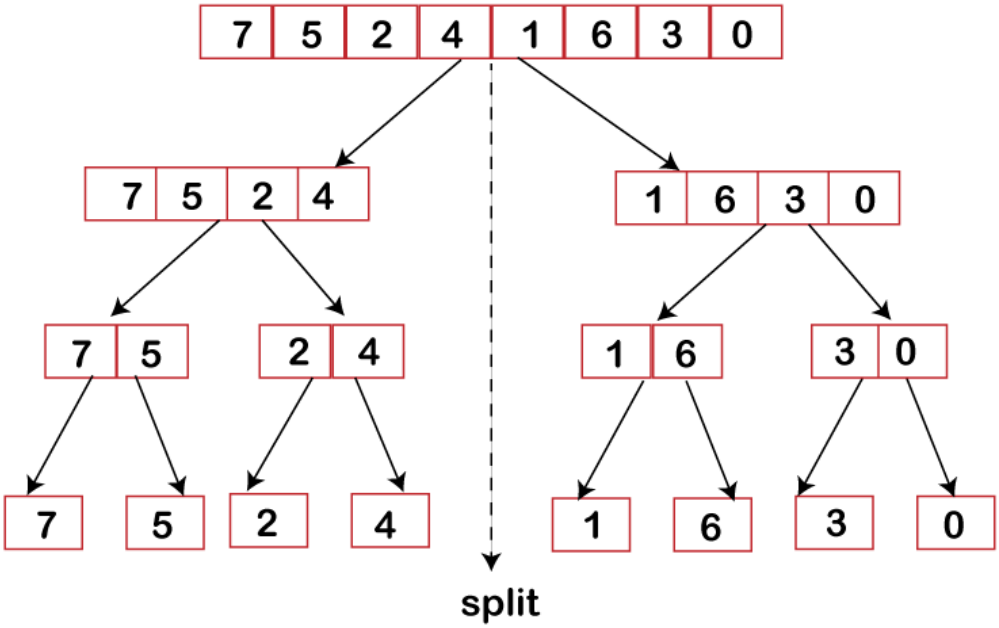
ALGORITHM-MERGE SORT

```
1. If p<r
2. Then q → ( p+ r)/2
3. MERGE-SORT (A, p, q)
4. MERGE-SORT ( A, q+1,r)
5. MERGE ( A, p, q, r)
```

Here we called **MergeSort(A, 0, length(A)-1)** to sort the complete array.

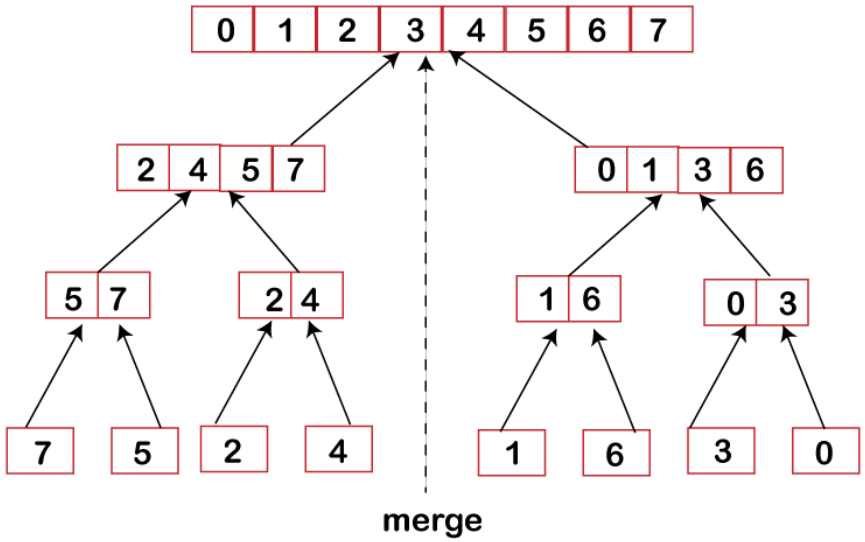
As you can see in the image given below, the merge sort algorithm recursively divides the array into halves until the base condition is met, where we are left with only 1 element in the array. And then, the merge function picks up the sorted sub-arrays and merge them back to sort the entire array.

The following figure illustrates the dividing (splitting) procedure.



FUNCTIONS: MERGE (A, p, q, r)

```
1. n 1 = q-p+1
2. n 2= r-q
3. create arrays [1.....n 1 + 1] and R [ 1.....n 2 +1 ]
4. for i ← 1 to n 1
5. do [i] ← A [ p+ i-1]
6. for j ← 1 to n2
7. do 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. Do if L [i] ≤ R[j]
14. then A[k] ← L[ i]
15. i ← i +1
16. else A[k] ← R[j]
17. j ← j+1
```



The merge step of Merge Sort

Mainly the recursive algorithm depends on a base case as well as its ability to merge back the results derived from the base cases. Merge sort is no different algorithm, just the fact here the **merge** step possesses more importance.

To any given problem, the merge step is one such solution that combines the two individually sorted lists(arrays) to build one large sorted list(array).

The merge sort algorithm upholds three pointers, i.e., one for both of the two arrays and the other one to preserve the final sorted array's current index.

```
Did you reach the end of the array?
    No:
    Firstly, start with comparing the current elements of
 both the arrays.
    Next, copy the smaller element into the sorted array.

    Lastly, move the pointer of the element containing a
smaller element.
    Yes:
    Simply copy the rest of the elements of the non-
empty array
```

Merge( ) Function Explained Step-By-Step

Consider the following example of an unsorted array, which we are going to sort with the help of the Merge Sort algorithm.

```
A= (36,25,40,2,7,80,15)
```

Step1: The merge sort algorithm iteratively divides an array into equal halves until we achieve an atomic value. In case if there are an odd number of elements in an array, then one of the halves will have more elements than the other half.

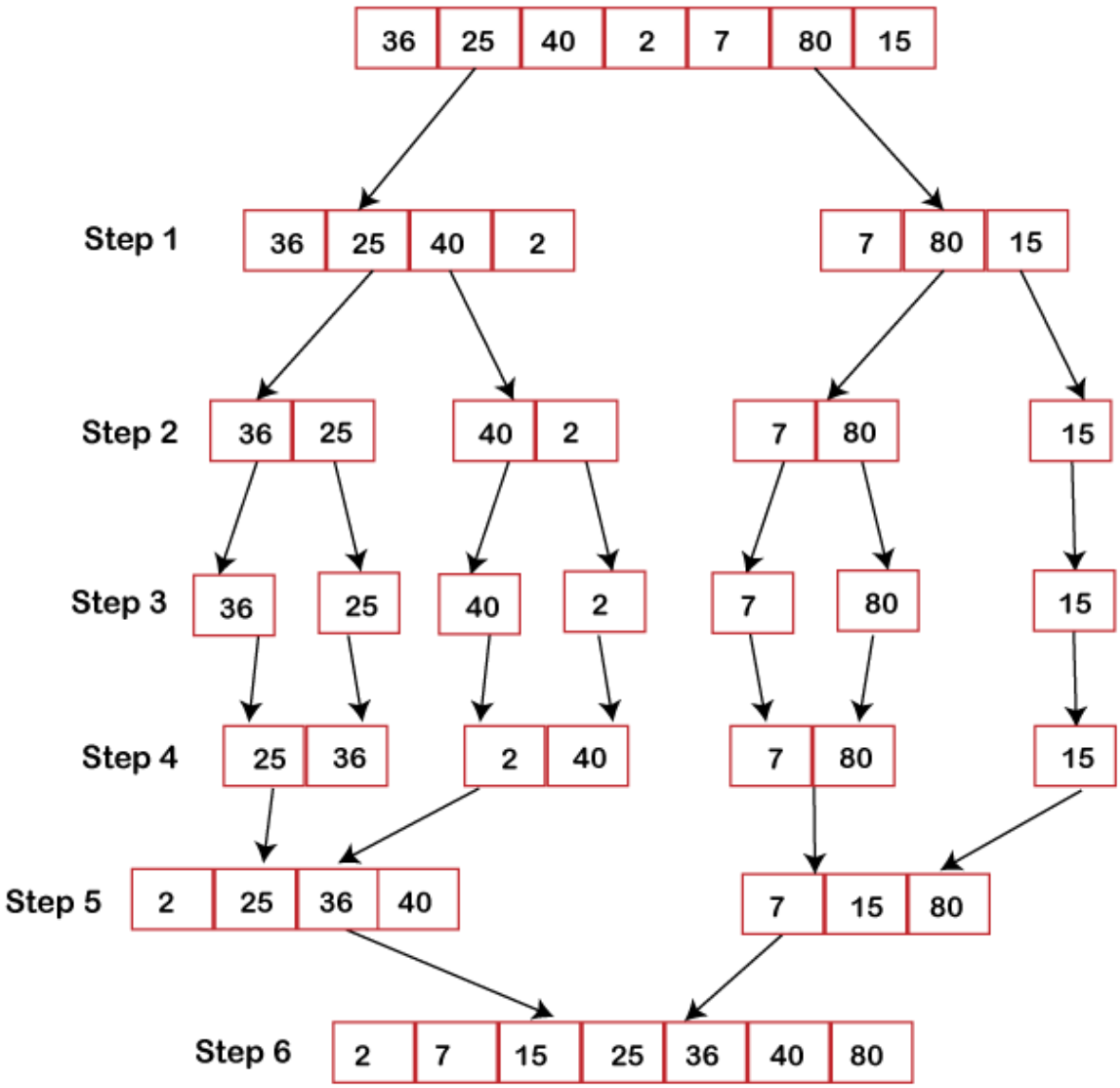
Step2: After dividing an array into two subarrays, we will notice that it did not hamper the order of elements as they were in the original array. After now, we will further divide these two arrays into other halves.

Step3: Again, we will divide these arrays until we achieve an atomic value, i.e., a value that cannot be further divided.

Step4: Next, we will merge them back in the same way as they were broken down.

Step5: For each list, we will first compare the element and then combine them to form a new sorted list.

Step6: In the next iteration, we will compare the lists of two data values and merge them back into a list of found data values, all placed in a sorted manner.



Hence the array is sorted.

3.1.1 Complexity Analysis of Merge Sort:

Best Case Complexity: The merge sort algorithm has a best-case time complexity of **O(n*log n)** for the already sorted array.

Average Case Complexity: The average-case time complexity for the merge sort algorithm is **O(n*log n)**, which happens when 2 or more elements are jumbled, i.e., neither in the ascending order nor in the descending order.

Worst Case Complexity: The worst-case time complexity is also **O(n*log n)**, which occurs when we sort the descending order of an array into the ascending order.

Space Complexity: The space complexity of merge sort is **O(n)**.

3.1.2 Merge Sort Applications

The concept of merge sort is applicable in the following areas:

- Inversion count problem
- External sorting
- E-commerce applications

3.1.3 Advantages of Merge Sort

a. Merge sort can efficiently sort a list in O(n*log(n)) time.
b. Merge sort can be used with linked lists without taking up any more space.
c. A merge sort algorithm is used to count the number of inversions in the list.
d. Merge sort is employed in external sorting.

3.1.4 Disadvantages of Merge Sort

a. For small datasets, merge sort is slower than other sorting algorithms.
b. For the temporary array, mergesort requires an additional space of O(n).
c. Even if the array is sorted, the merge sort goes through the entire process.

Self-Assessment Exercise

1. A list of n string, each of length n, is sorted into lexicographic order using the merge-sort algorithm. The worst case running time of this computation is?
2. What is the average case time complexity of merge sort?
3. A mergesort works by first breaking a sequence in half a number of times so it is working with smaller pieces. When does it stop breaking the list into sublists (in its simplest version)?

3.2 Quick Sort

A sorting technique developed by British computer scientist Tony Hoare in 1959 and published in 1961, that sequences a list by continuously dividing the list into two parts and moving the lower items to one side and the higher items to the other. It starts by picking one item in the entire list to serve as a pivot point. The entire process takes place in the following three steps:

Divide: Rearrange the elements and split arrays into two sub-arrays and an element in between search that each element in left sub array is less than or equal to the average element and each element in the right sub- array is larger than the middle element.

Conquer: Recursively, sort two sub arrays.

Combine: Combine the already sorted array.

Algorithm:

```
QUICKSORT (array A, int m, int n)
 1 if (n > m)
 2 then
 3 i ← a random index from [m,n]
 4 swap A [i] with A[m]
 5 o ← PARTITION (A, m, n)
 6 QUICKSORT (A, m, o - 1)
 7 QUICKSORT (A, o + 1, n)
```

Partition Algorithm:

Partition algorithm rearranges the sub arrays in a place.

```
PARTITION (array A, int m, int n)
 1 x ← A[m]
 2 o ← m
 3 for p ← m + 1 to n
 4 do if (A[p] < x)
 5 then o ← o + 1
 6 swap A[o] with A[p]
 7 swap A[m] with A[o]
 8 return o
```

Example of Quick Sort. Given the following list;

44 33 11 55 77 90 40 60 99 22 88

Let **44** be the **Pivot** element and scanning done from right to left

Comparing **44** to the right-side elements, and if right-side elements are **smaller** than **44**, then swap it. As **22** is smaller than **44** so swap them.

22 33 11 55 77 90 40 60 99 **44** 88

Now comparing **44** to the left side element and the element must be **greater** than 44 then swap them. As **55** are greater than **44** so swap them.

22 33 11 **44** 77 90 40 60 99 **55** 88

Recursively, repeating steps 1 and steps 2 until we get two lists one left from pivot element **44** & one right from pivot element.

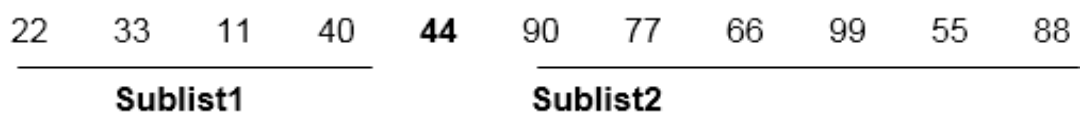
22 33 **40** 77 90 **44** 60 99 55 88

Swap with 77:

22 33 11 40 **44** 90 **77** 60 99 55 88

Now, the element on the right side and left side are greater than and smaller than **44** respectively.

Now we get two sorted lists:



And these sublists are sorted under the same process as above done.

These two sorted sublists side by side.

22	33	11	40	**44**	**90**	77	60	99	55	88
11	33	**22**	40	**44**	88	77	60	99	55	**90**
11	**22**	33	40	**44**	88	77	60	**90**	55	**99**

First sorted list

88	77	60	**55**	**90**	99
Sublist3				**Sublist4**	

55	77	60	**88**	90	99
				Sorted	

55	**77**	**60**
55	60	77
Sorted		

Merging Sublists:

11	**22**	**33**	**40**	**44**	**55**	**60**	**77**	**88**	**90**	**99**

SORTED LISTS

3.2.1 Complexity of Quicksort

Worst Case Analysis: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.

Worst Case Complexity of Quick Sort is **T (n) =O (n^2)**

Average Case Analysis: T(n) = O(n log n) is the average case complexity of quick sort for sorting n elements.

Best Case Analysis: In any sorting, best case is the only case in which we don't make any comparison between elements that is only done when we have only one element to sort.

T(n) = O(n log n)

3.2.2 Advantages of Quick Sort

a. It is in-place since it uses only a small auxiliary stack.
b. It requires only n (log n) time to sort n items.
c. It has an extremely short inner loop.
d. This algorithm has been subjected to a thorough mathematical analysis, a very precise statement can be made about performance issues.

3.2.3 Disadvantages of Quick Sort

a. It is recursive. Especially, if recursion is not available, the implementation is extremely complicated.
b. It requires quadratic (i.e., n2) time in the worst-case.
c. It is fragile, i.e. a simple mistake in the implementation can go unnoticed and cause it to perform badly.

3.2.4 Applications of QuickSort

a. It is used **for information searching** since it is the fastest and is widely used as a better way of searching.
b. It is used everywhere where a stable sort is not needed.
c. Quicksort is a cache-friendly algorithm as it has a good locality of reference when used for arrays.

Self-Assessment Exercises

1. What is recurrence for worst case of QuickSort and what is the time complexity in Worst case?
2. Sort the following list in descending order of magnitude using QuickSort [23, 65, 8, 78, 3, 65, 21, 9, 4, 43, 76, 1, 6, 4, 8, 56]. You can pick any element as your pivot.

3. Apply Quick sort on a given sequence 7 11 14 6 9 4 3 12. What is the sequence after first phase, pivot is first element?

4.0 Conclusion

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Quicksort, is a sorting algorithm that makes **n log n** comparisons in average case for sorting an array of n elements. It is a fast and highly efficient sorting algorithm and follows the divide-and-conquer approach.

5.0 Summary

In this Unit, we examined two sorting algorithm examples of Divide-and-Conquer algorithms. The Mergesort which is also and external sorting algorithm was considered with its complexity analysis explained as well as its benefits and shortcomings.

The QuickSort algorithm which is another example of a Divide-and-Conquer algorithm was also looked at as well as its advantages and disadvantages.

6.0 Tutor Marked Assignment

1. Quicksort works by choosing a pivot value and moving list elements around. Each element less than the pivot will be closer to the beginning of the list than the pivot, and each element greater than the pivot will be closer to the end of the list. By doing this operation many times with different pivots, the list will become sorted. For the fastest operation, which would be the best pivot value?
2. Sort the following list in ascending order. [8, 1, 4, 9, 6, 3, 5, 2, 7, 0] using

   a. MergeSort b. Quicksort

3. To sort the array [5, 4, 3, 2, 1, 0] in ascending order, the first merge in MergeSort will result in?
4. Write a program in any language of your choice to implement the Quick Sort and the Mergesort Algorithms.

7.0 Further Reading and other Resources

Karumanchi, N. (2016). Data Structures and Algorithms, CareerMonk Publications. ISBN-13 ： 978-8193245279

Sen, S. and Kumar, A, (2019). Design and Analysis of Algorithms. A Contemporary Perspective. Cambridge University Press. ISBN: 1108496822, 9781108496827

Vermani, L. R. and Vermani, S.(2019). An Elementary Approach To Design And Analysis Of Algorithms. World Scientific. ISBN: 178634677X, 9781786346773

CIT 310 – Algorithms and Complexity Analysis
Module 3: Other Algorithm Techniques

Unit 1 Binary Search Trees
Unit 2 Dynamic Programming
Unit 3 Computational Complexity
Unit 4 Approximate Algorithms I
Unit 5 Approximate Algorithms II

Module 3: Other Algorithm Techniques

Unit 1: Binary Search Trees

1.0 Introduction

We introduce here a special search tree called the Binary Search Tree and a derivative of it known as the Red Black Tree.

A binary search tree, also known as ordered binary tree is a binary tree wherein the nodes are arranged in a order. The order is : a) All the values in the left sub-tree has a value less than that of the root node. b) All the values in the right node have a value greater than the value of the root node.

On the other hand, a red-black tree is a Binary tree where a particular node has color as an extra attribute, either red or black. By check the node colors on any simple path from the root to a leaf, red-black trees secure that no such path is higher than twice as long as any other so that the tree is generally balanced.

2.0 Objectives

At the end of this unit, you will be able to:

- Understand the meaning of a Binary Search Tree.
- Know the different methods of traversing a Binary Search Tree
- List and explain the different ways a Binary Search Tree can be queried
- Understand the Red Black Trees
- Learn the different properties of Red Black Trees
- Know the different operations done on Red Black Trees

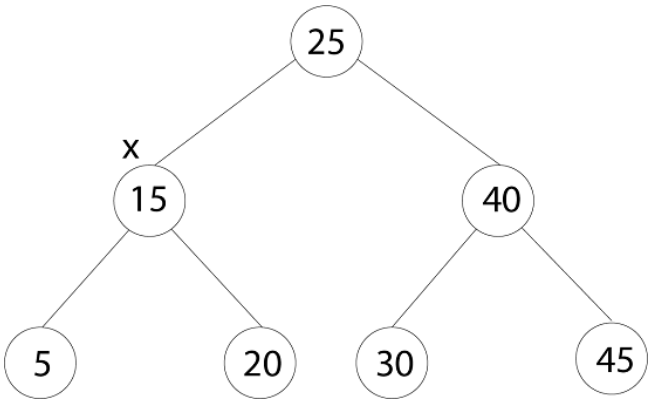
3.0 Binary Search Trees

A Binary Search tree is organized in a Binary Tree. Such a tree can be defined by a linked data structure in which a particular node is an object. In addition to a key field, each node contains field left, right, and p that point to the nodes corresponding to its left child, its right child, and its parent, respectively. If a child or parent is missing, the appropriate field contains the value Nil. The root node is the only node in the tree whose parent field is Nil.

3.0.1 Binary Search Tree Property

Let x be a node in a binary search tree.

- If y is a node in the left subtree of x, then key [y] ≤key [k].
- If z is a node in the right subtree of x, then key [x] ≤ key [y].



In this tree key [x] = 15

If y is a node in the left subtree of x, then key [y] = 5.

i.e. key [y] ≤ key[x].

If y is a node in the right subtree of x, then key [y] = 20.

i.e. key [x] ≤ key[y].

3.1 Traversal in Binary Search Trees:

3.1.1 In-Order-Tree-Walk (x):

In Inorder Tree walk, we always print the keys in the binary search tree in a sorted order.

INORDER-TREE-WALK (x) - Running time is θ(n)

```
1. If x ≠ NIL.
2. then INORDER-TREE-WALK (left [x])
```

```
3. print key [x]
4. INORDER-TREE-WALK (right [x])
```

3.1.2. PREORDER-TREE-WALK (x):

In Preorder Tree walk, we visit the root node before the nodes in either subtree.

PREORDER-TREE-WALK (x):

```
1. If x ≠ NIL.
2. then print key [x]
3. PREORDER-TREE-WALK (left [x]).
4. PREORDER-TREE-WALK (right [x]).
```

3.1.3. POSTORDER-TREE-WALK (x):

In Postorder Tree walk, we visit the root node after the nodes in its subtree.

POSTORDER-TREE-WALK (x):

```
1. If x ≠ NIL.
2. then POSTORDER-TREE-WALK (left [x]).
3. POSTORDER-TREE-WALK (right [x]).
4. print key [x]
```

3.2 Querying a Binary Search Trees:

3.2.1. Searching:

The TREE-SEARCH (x, k) algorithm searches the tree node at x for a node whose key value equal to k. It returns a pointer to the node if it exist otherwise NIL.

TREE-SEARCH (x, k)

```
1. If x = NIL or k = key [x].
2. then return x.
```

```
3. If k < key [x].
4. then return TREE-SEARCH (left [x], k)
5. else return TREE-SEARCH (right [x], k)
```

Clearly, this algorithm runs in O (h) time where h is the height of the tree. The iterative version of the above algorithm is very easy to implement

ITERATIVE-TREE- SEARCH (x, k)

```
1. while x ≠ NIL and k ≠ key [k].
2. do if k < key [x].
3. then x ← left [x].
4. else x ← right [x].
5. return x.
```

3.2.2. Minimum and Maximum:

An item in a binary search tree whose key is a minimum can always be found by following left child pointers from the root until a NIL is encountered. The following procedure returns a pointer to the minimum element in the subtree rooted at a given node x.

TREE- MINIMUM (x)

```
1. While left [x] ≠ NIL.
2. do x←left [x].
3. return x.
```

TREE-MAXIMUM (x)

```
1. While left [x] ≠ NIL
2. do x←right [x].
3. return x.
```

3.2.3. Successor and predecessor:

Given a node in a binary search tree, sometimes we used to find its successor in the sorted form determined by an in order tree walk. If all keys are specific, the successor of a node x is the node with the smallest key greater than key[x]. The structure of a binary search tree allows us to rule the successor of a node without ever comparing keys. The following action returns the successor of a node x in a binary search tree if it exists, and NIL if x has the greatest key in the tree:

TREE SUCCESSOR (x)

```
1. If right [x] ≠ NIL.
2. Then return TREE-MINIMUM (right [x]))
3. y←p [x]
4. While y ≠ NIL and x = right [y]
5. do x←y
6. y←p[y]
7. return y.
```

The code for TREE-SUCCESSOR is broken into two cases. If the right subtree of node x is nonempty, then the successor of x is just the leftmost node in the right subtree, which we find in line 2 by calling TREE-MINIMUM (right [x]). On the other hand, if the right subtree of node x is empty and x has a successor y, then y is the lowest ancestor of x whose left child is also an ancestor of x. To find y, we quickly go up the tree from x until we encounter a node that is the left child of its parent; lines 3-7 of TREE-SUCCESSOR handle this case.

The running time of TREE-SUCCESSOR on a tree of height h is O (h) since we either follow a simple path up the tree or follow a simple path down the tree. The procedure TREE-PREDECESSOR, which is symmetric to TREE-SUCCESSOR, also runs in time O (h).

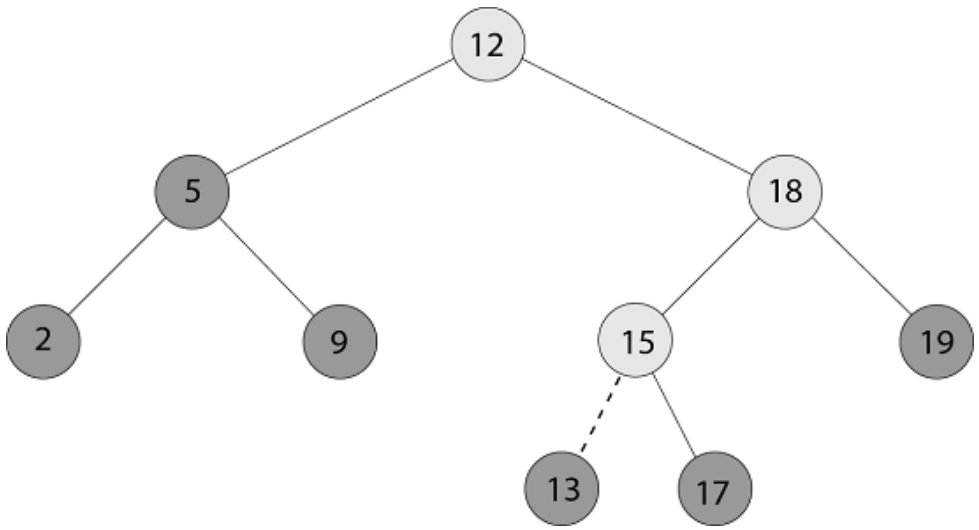
3.2.4. Insertion in Binary Search Tree:

To insert a new value into a binary search tree T, we use the procedure TREE-INSERT. The procedure takes a node ´ for which key [z] = v, left [z] NIL, and right [z] = NIL. It modifies T and some of the attributes of z in such a way that it inserts into an appropriate position in the tree.

TREE-INSERT (T, z)

```
1. y ←NIL.
2. x←root [T]
3. while x ≠ NIL.
4. do y←x
5. if key [z]< key [x]
6. then x←left [x].
7. else x←right [x].
8. p [z]←y
9. if y = NIL.
10. then root [T]←z
11. else if key [z] < key [y]
12. then left [y]←z
```

For Example:



Working of TREE-INSERT

Suppose we want to insert an item with key 13 into a Binary Search Tree.

```
  x = 1
  y = 1 as x ≠ NIL.
  Key [z] < key [x]
      13 < not equal to 12.
  x ←right [x].
  x ←3
Again x ≠ NIL
  y ←3
  key [z] < key [x]
      13 < 18
  x←left [x]
  x←6
Again x ≠ NIL, y←6
  13 < 15
  x←left [x]
  x←NIL
  p [z]←6
```

Now our node z will be either left or right child of its parent (y).

```
key [z] < key [y]
   13 < 15
Left [y] ← z
Left [6] ← z
```

So, insert a node in the left of node index at 6.

3.2.5. Deletion in Binary Search Tree:

When Deleting a node from a tree it is essential that any relationships, implicit in the tree can be maintained. The deletion of nodes from a binary search tree will be considered:

There are three distinct cases:

1. **Nodes with no children:** This case is trivial. Simply set the parent's pointer to the node to be deleted to nil and delete the node.
2. **Nodes with one child:** When z has no left child then we replace z by its right child which may or may not be NIL. And when z has no right child, then we replace z with its right child.
3. **Nodes with both Childs:** When z has both left and right child. We find z's successor y, which lies in right z's right subtree and has no left child (the

successor of z will be a node with minimum value its right subtree and so it has no left child).

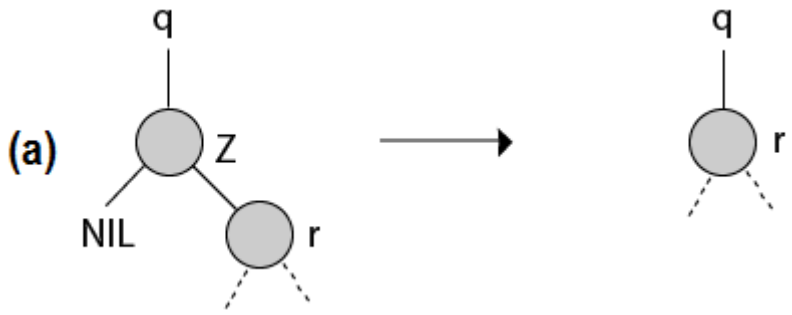
- If y is z's right child, then we replace z.
- Otherwise, y lies within z's right subtree but not z's right child. In this case, we first replace z by its own right child and the replace z by y.

TREE-DELETE (T, z)

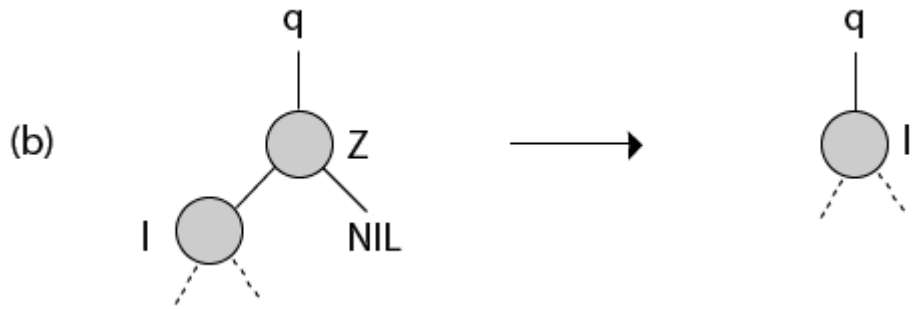
```
If left [z] = NIL or right [z] = NIL.
Then y ← z
Else y  ← TREE- SUCCESSOR (z)
If left [y] ≠ NIL.
Then x ← left [y]
Else x ← right [y]
If x ≠NIL
Then p[x] ← p [y]
If p[y] = NIL.
Then root [T] ← x
Else if y = left [p[y]]
Then left [p[y]] ← x
Else right [p[y]] ← y
If y ≠ z.
Then key [z] ← key [y]
If y has other fields, copy them, too.
Return y
```

The Procedure runs in O (h) time on a tree of height h.

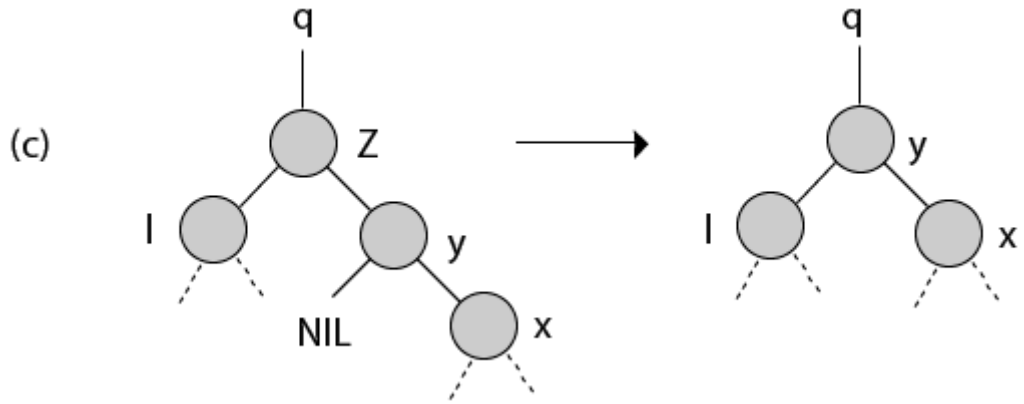
For Example: Deleting a node z from a binary search tree. Node z may be the root, a left child of node q, or a right child of q.



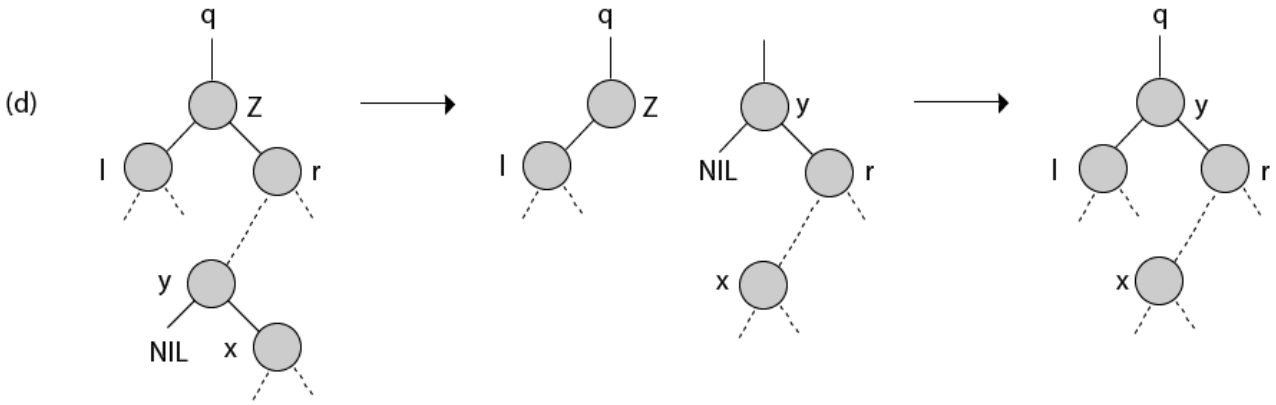
Z has the only right child.



Z has the only left child. We replace z by l.



Node z has two children; its left child is node l, its right child is its successor y, and y's right child is node x. We replace z by y, updating y's left child to become l, but leaving x as y's right child.



Node z has two children (left child l and right child r), and its successor y ≠ r lies within the subtree rooted at r. We replace y with its own right child x, and we set y to be r's parent. Then, we set y to be q's child and the parent of l.

Self-Assessment Exercises

1. What is the worst case time complexity for search, insert and delete operations in a general Binary Search Tree?
2. We are given a set of n distinct elements and an unlabelled binary tree with n nodes. In how many ways can we populate the tree with the given set so that it becomes a binary search tree?
3. How many distinct binary search trees can be created out of 4 distinct keys?
4. Suppose the numbers 7, 5, 1, 8, 3, 6, 0, 9, 4, 2 are inserted in that order into an initially empty binary search tree. The binary search tree uses the usual ordering on natural numbers. What is the in-order traversal sequence of the resultant tree?

3.3 Red Black Tree

A Red Black Tree is a category of the self-balancing binary search tree. It was created in 1972 by Rudolf Bayer who termed them **"symmetric binary B-trees**."

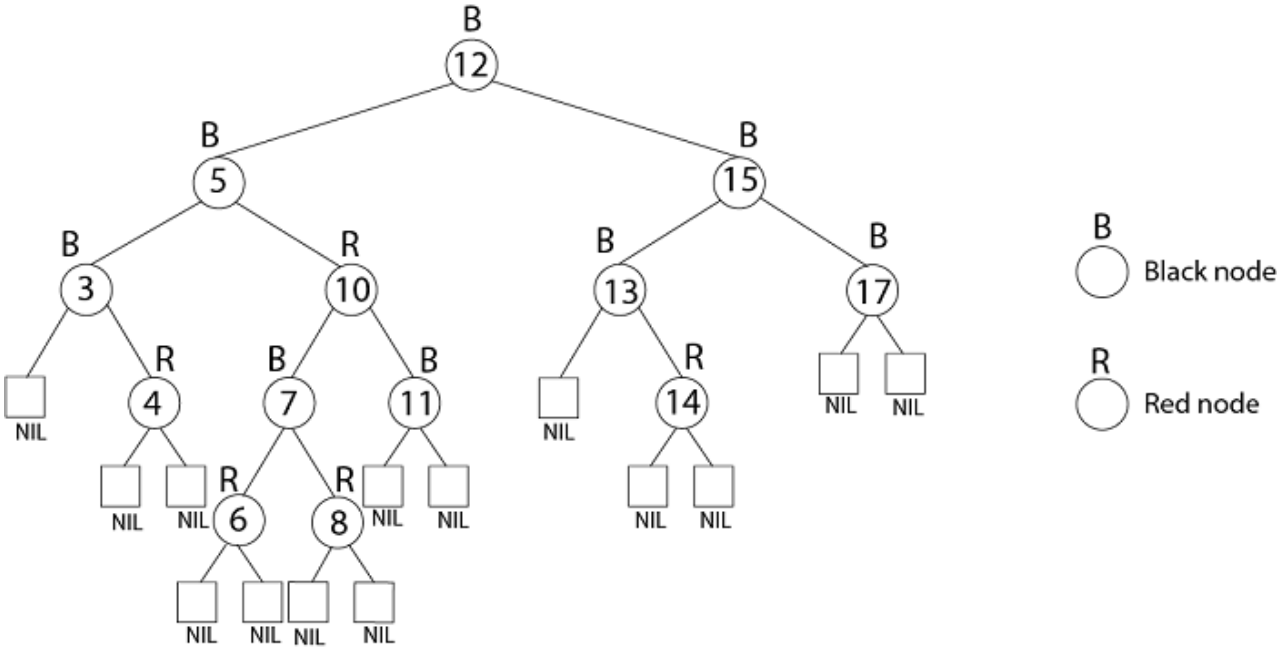
A red-black tree is a Binary tree where a particular node has color as an extra attribute, either red or black. By check the node colors on any simple path from the root to a leaf, red-black trees secure that no such path is higher than twice as long as any other so that the tree is generally balanced.

3.3.1 Properties of Red-Black Trees

A red-black tree must satisfy these properties:

1. The root is always black.
2. A nil is recognized to be black. This factor that every non-NIL node has two children.
3. **Black Children Rule:** The children of any red node are black.
4. **Black Height Rule:** For particular node v, there exists an integer bh (v) such that specific downward path from v to a nil has correctly bh (v) black real (i.e. non-nil) nodes. Call this portion the black height of v. We determine the black height of an RB tree to be the black height of its root.

A tree T is an almost red-black tree (ARB tree) if the root is red, but other conditions above hold.

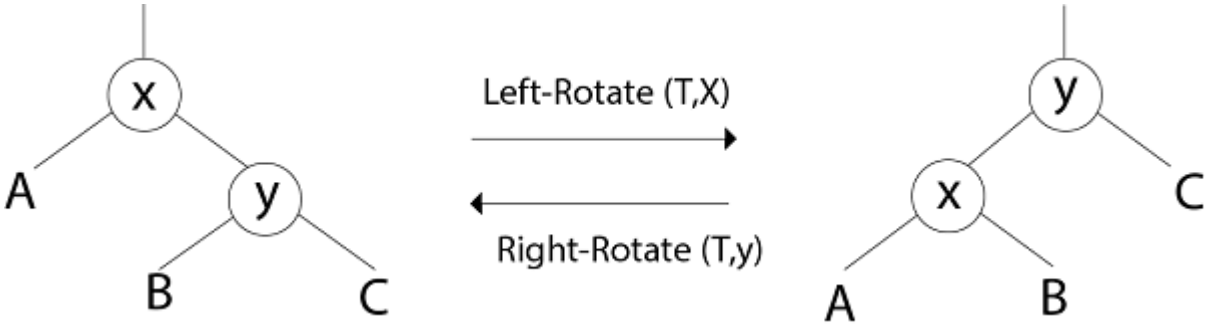


3.4 Operations on RB Trees:

The search-tree operations TREE-INSERT and TREE-DELETE, when runs on a red-black tree with n keys, take O (log n) time. Because they customize the tree, the conclusion may violate the red-black properties. To restore these properties, we must change the color of some of the nodes in the tree and also change the pointer structure.

3.4.1. Rotation:

Restructuring operations on red-black trees can generally be expressed more clearly in details of the rotation operation.



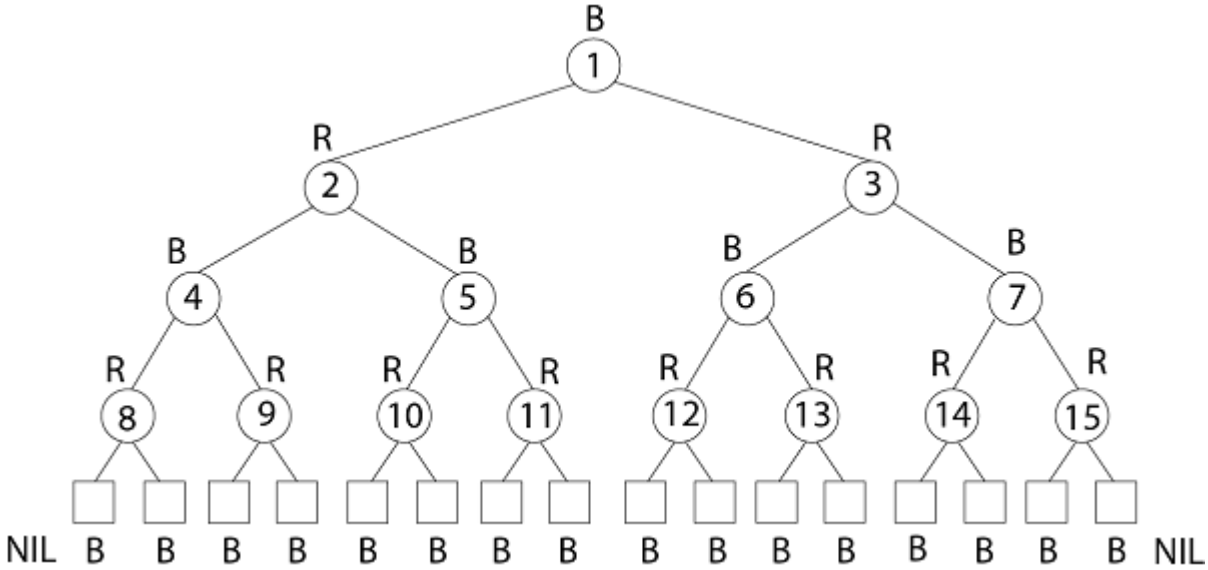
Clearly, the order (Ax By C) is preserved by the rotation operation. Therefore, if we start with a BST and only restructure using rotation, then we will still have a BST i.e. rotation do not break the BST-Property.

LEFT ROTATE (T, x)

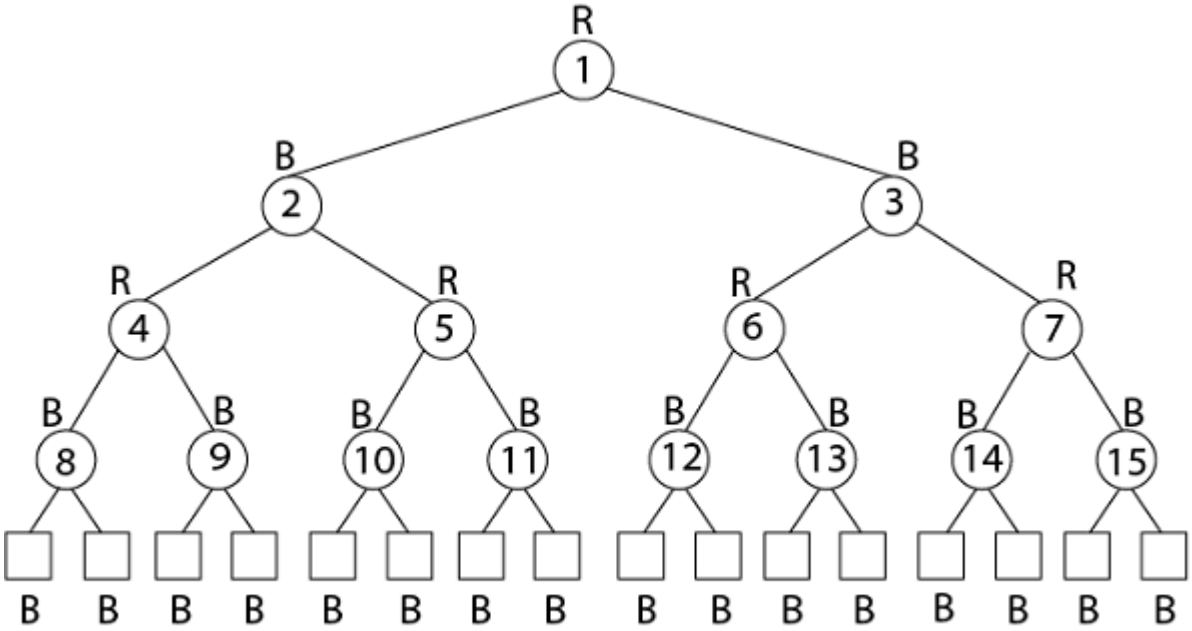
```
1. y ← right [x]
1. y ← right [x]
2. right [x] ← left [y]
3. p [left[y]] ← x
4. p[y] ← p[x]
5. If p[x] = nil [T]
  then root [T] ← y
   else if x = left [p[x]]
    then left [p[x]] ← y
   else right [p[x]] ← y
6. left [y] ← x.
7. p [x] ← y.
```

Example: Draw the complete binary tree of height 3 on the keys {1, 2, 3... 15}. Add the NIL leaves and color the nodes in three different ways such that the black heights of the resulting trees are: 2, 3 and 4.

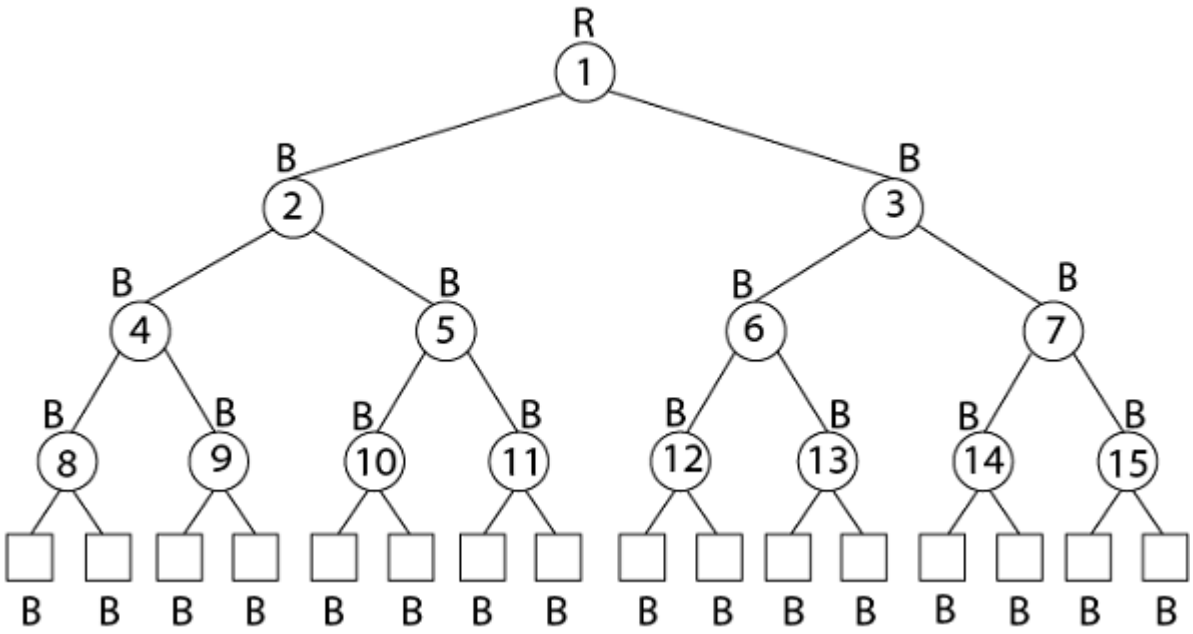
Solution:



Tree with black-height-2



Tree with black-height-3



Tree with black-height-4

3.4.2. Insertion:

- Insert the new node the way it is done in Binary Search Trees.
- Color the node red
- If an inconsistency arises for the red-black tree, fix the tree according to the type of discrepancy.

A discrepancy can be a decision from a parent and a child both having a red color. This type of discrepancy is determined by the location of the node concerning grandparent, and the color of the sibling of the parent.

RB-INSERT (T, z)

```
y ← nil [T]
x ← root [T]
while x ≠ NIL [T]
do y ← x
if key [z] < key [x]
then x  ← left [x]
else x ←  right [x]
p [z] ← y
```

```
if y = nil [T]
then root [T] ← z
else if key [z] < key [y]
then left [y] ← z
else right [y] ← z
left [z] ← nil [T]
right [z] ← nil [T]
color [z] ← RED
RB-INSERT-FIXUP (T, z)
```

After the insert new node, Coloring this new node into black may violate the black-height conditions and coloring this new node into red may violate coloring conditions i.e. root is black and red node has no red children. We know the black-height violations are hard. So we color the node red. After this, if there is any color violation, then we have to correct them by an RB-INSERT-FIXUP procedure.

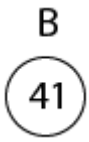
RB-INSERT-FIXUP (T, z)

```
      while color [p[z]] = RED
      do if p [z] = left [p[p[z]]]
      then y ← right [p[p[z]]]
      If color [y] = RED
5. then color [p[z]] ← BLACK     //Case 1
6. color [y] ← BLACK             //Case 1
7. color [p[z]] ← RED            //Case 1
8. z  ← p[p[z]]                  //Case 1
else if z= right [p[z]]
10. then z ← p [z]               //Case 2
11. LEFT-ROTATE (T, z)           //Case 2
12. color [p[z]] ← BLACK         //Case 3
13. color [p [p[z]]] ← RED       //Case 3
14. RIGHT-ROTATE  (T,p [p[z]])   //Case 3
15. else (same as then clause)
With "right" and "left" exchanged
16. color [root[T]] ← BLACK
```

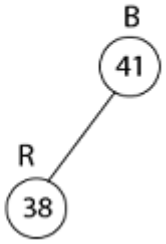
Example: Show the red-black trees that result after successively inserting the keys 41,38,31,12,19,8 into an initially empty red-black tree.

Solution:

Insert 41

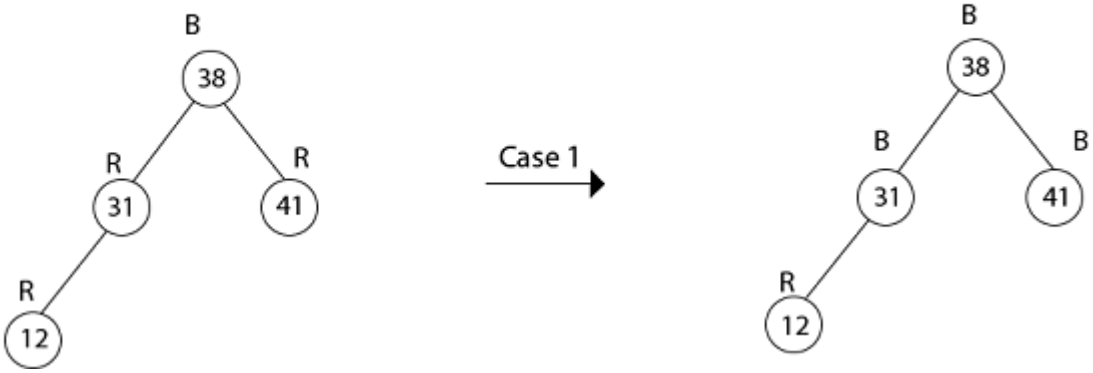


- Insert 38

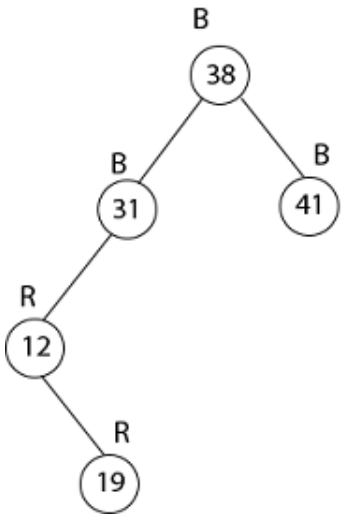


- Insert 31

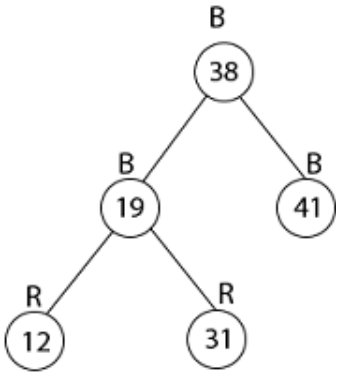
- Insert 12



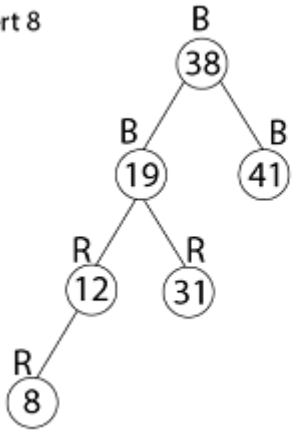
Insert 19



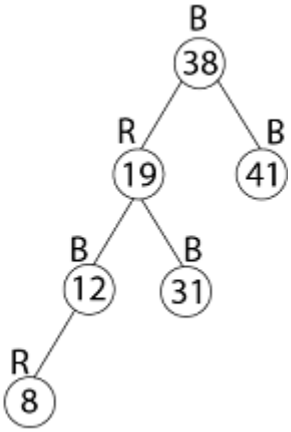
Case 2,3



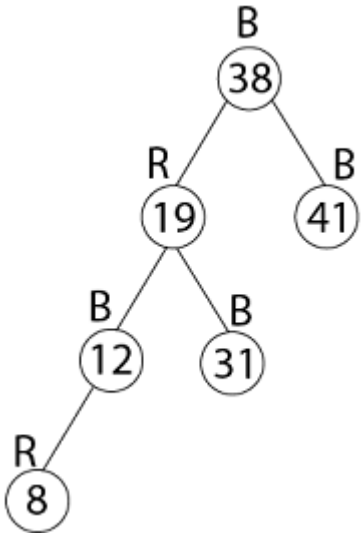
Insert 8



Case-1



Thus the final tree is



3.4.3. Deletion:

First, search for an element to be deleted

- If the element to be deleted is in a node with only left child, swap this node with one containing the largest element in the left subtree. (This node has no right child).
- If the element to be deleted is in a node with only right child, swap this node with the one containing the smallest element in the right subtree (This node has no left child).
- If the element to be deleted is in a node with both a left child and a right child, then swap in any of the above two ways. While swapping, swap only the keys but not the colors.
- The item to be deleted is now having only a left child or only a right child. Replace this node with its sole child. This may violate red constraints or black constraint. Violation of red constraints can be easily fixed.
- If the deleted node is black, the black constraint is violated. The elimination of a black node y causes any path that contained y to have one fewer black node.
- Two cases arise:
    - The replacing node is red, in which case we merely color it black to make up for the loss of one black node.
    - The replacing node is black.

The strategy RB-DELETE is a minor change of the TREE-DELETE procedure. After splicing out a node, it calls an auxiliary procedure RB-DELETE-FIXUP that changes colors and performs rotation to restore the red-black properties.

RB-DELETE (T, z)

```
1. if left [z] = nil [T] or right [z] = nil [T]
2. then y ← z
3. else y ← TREE-SUCCESSOR (z)
4. if left [y] ≠ nil [T]
5. then x ← left [y]
6. else x ← right [y]
7. p [x] ←  p [y]
8. if p[y] = nil [T]
```

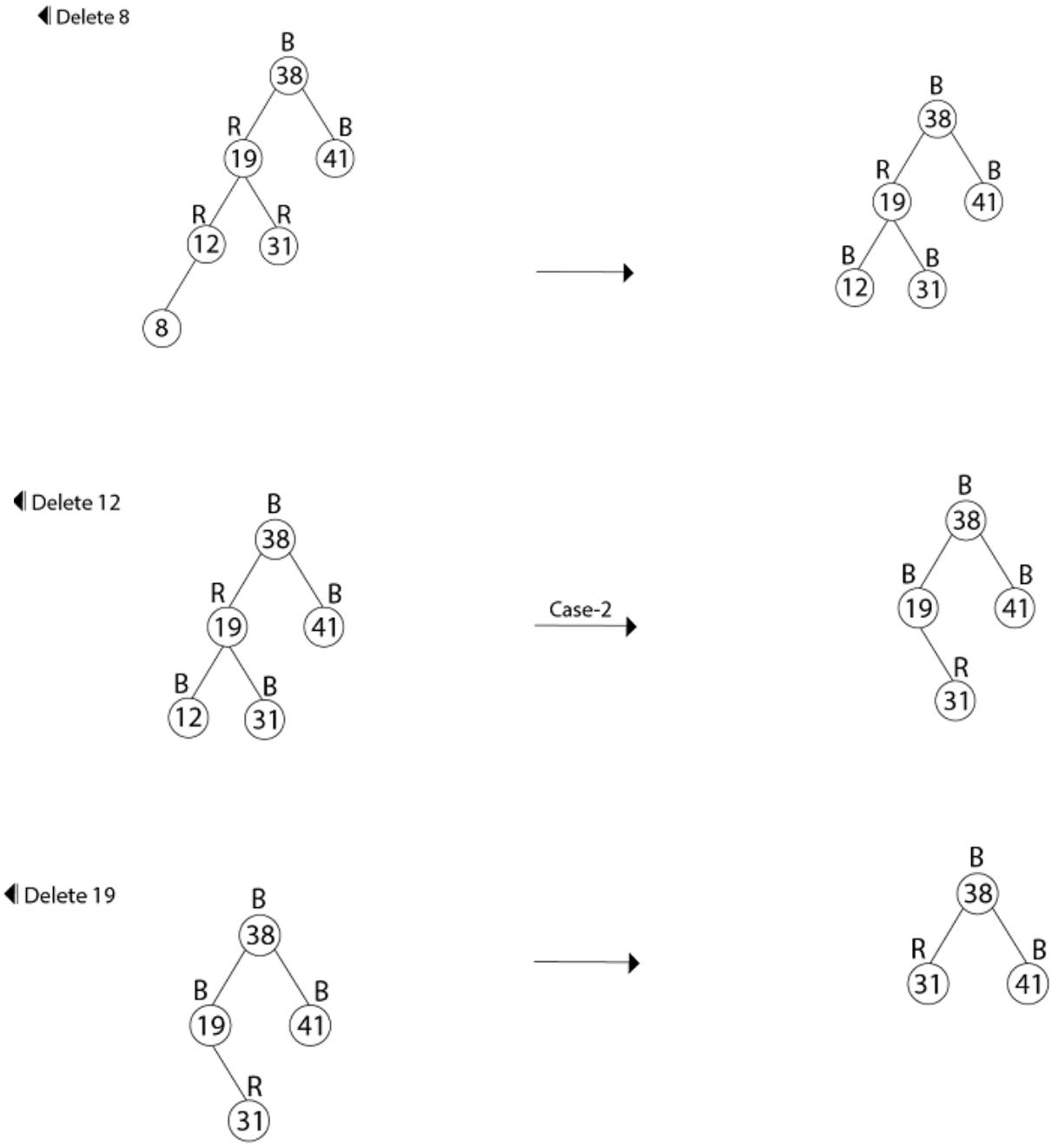
```
9. then root [T]  ← x
10. else if y = left [p[y]]
11. then left [p[y]] ← x
12. else right [p[y]] ← x
13. if y≠ z
14. then key [z] ← key [y]
15. copy y's satellite data into z
16. if color [y] = BLACK
17. then RB-delete-FIXUP (T, x)
18. return y
```

RB-DELETE-FIXUP (T, x)

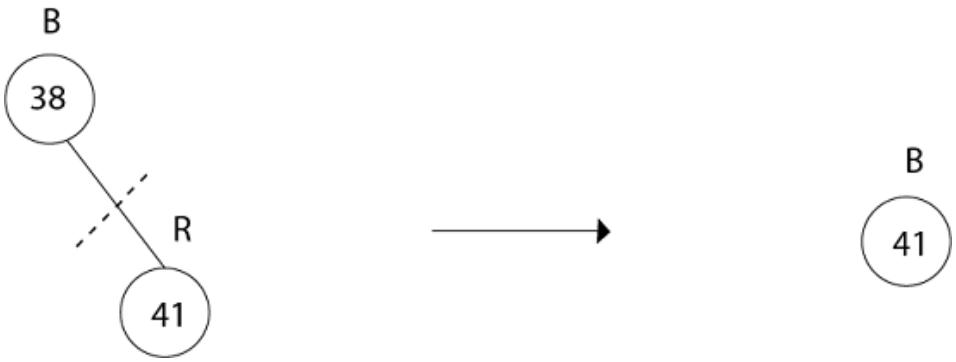
```
1. while x ≠ root [T] and color [x] = BLACK
2. do if x = left [p[x]]
3. then w ← right [p[x]]
4. if color [w] = RED
5. then color [w] ← BLACK        //Case 1
6. color [p[x]] ← RED            //Case 1
7. LEFT-ROTATE (T, p [x])        //Case 1
8. w ← right [p[x]]              //Case 1
9. If color [left [w]] = BLACK and color [right[w]] =
BLACK
10. then color [w] ← RED         //Case 2
11. x ← p[x]                     //Case 2
12. else if color [right [w]] = BLACK
13. then color [left[w]] ← BLACK //Case 3
14. color [w] ← RED              //Case 3
15. RIGHT-ROTATE (T, w)          //Case 3
16. w ← right [p[x]]             //Case 3
17. color [w] ← color [p[x]]     //Case 4
18. color p[x] ← BLACK           //Case 4
19. color [right [w]] ← BLACK    //Case 4
20. LEFT-ROTATE (T, p [x])       //Case 4
21. x ← root [T]                 //Case 4
22. else (same as then clause with "right" and "left"
exchanged)
23. color [x] ← BLACK
```

Example: In a previous example, we found that the red-black tree that results from successively inserting the keys 41,38,31,12,19,8 into an initially empty tree. Now show the red-black trees that result from the successful deletion of the keys in the order 8, 12, 19,31,38,41.

Solution:



Delete 38



Delete 41

No Tree.

Self-Assessment Exercises

1. When deleting a node from a red-black tree, what condition might happen?
2. What is the maximum height of a Red-Black Tree with 14 nodes? (Hint: The black depth of each external node in this tree is 2.) Draw an example of a tree with 14 nodes that achieves this maximum height.
3. Why can't a Red-Black tree have a black node with exactly one black child and no red child?

4.0 Conclusion

A binary search tree, also called an ordered or sorted binary tree, is a rooted binary tree data structure whose internal nodes each store a key greater than all the keys in the node's left subtree and less than those in its right subtree. On the other hand, a red–black tree is a kind of self-balancing binary search tree. Each node stores an extra bit representing "color", used to ensure that the tree remains balanced during insertions and deletions

5.0 Summary

In this unit, we considered the Binary Search Tree and looked at how such trees could be traversed while also examining the various methods of querying or accessing information from a Binary Search Tree. In addition, we looked at a special derivative of the Binary Search Tree called Red Black Trees, its properties and also some operations that could be carried out on Red Black Tress.

6.0 Tutor-Marked Assignments

1. What is the special property of red-black trees and what root should always be?
   a) a color which is either red or black and root should always be black color only
2. The following numbers are inserted into an empty binary search tree in the given order: 10, 1, 3, 5, 15, 12, 16. What is the height of the binary search tree (the height is the maximum distance of a leaf node from the root)?
3. What are the operations that could be performed in O(logn) time complexity by red-black tree?
4. The preorder traversal sequence of a binary search tree is 30, 20, 10, 15, 25, 23, 39, 35, 42. Give the postorder and inorder traversal sequence of the same tree.
5. How can you save memory when storing color information in Red-Black tree?
6. Which of the following traversals is sufficient to construct BST from given traversals 1) Inorder 2) Preorder 3) Postorder

7.0 Further Reading and other Resources

Baase, S. and Van Gelder, A. (2020). Computer Algorithms: Introduction to Design and Analysis. Addison-Wesley. ISBN: 0201612445, 9780201612448

Bhasin, H. (2015). Algorithms: Design and Analysis. Oxford University Press. ISBN: 0199456666, 9780199456666

Sen, S. and Kumar, A, (2019). Design and Analysis of Algorithms. A Contemporary Perspective. Cambridge University Press. ISBN: 1108496822, 9781108496827

Vermani, L. R. and Vermani, S.(2019). An Elementary Approach To Design And Analysis Of Algorithms. World Scientific. ISBN: 178634677X, 9781786346773

Module 3: Other Algorithm Techniques

Unit 2: Dynamic Programming

1.0 Introduction

Dynamic programming is both a mathematical optimization method and a computer programming method. The method was developed by Richard Bellman in the 1950s and has found applications in numerous fields, from aerospace engineering to economics. We look at some of the techniques of Dynamic Programming in this unit as well as some benefits and applications of Dynamic Programming

2.0 Objectives

At the end of this unit, you should be able to

- Explain better the concept of Dynamic Programming
- Know the different methods for resolving a Dynamic Programming problem
- Know when to use either of the methodologies learnt
- Understand the different areas of applications of Dynamic Programming
- Evaluate the basic differences between Dynamic Programming and the Divide-and-Conquer paradigm.

3.0 Dynamic Programming

Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The subproblems are optimized to optimize the overall solution is known as optimal substructure property. The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem. The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.

From the definition of dynamic programming, it is a technique for solving a complex problem by first breaking it into a collection of simpler subproblems,

solving each subproblem just once, and then storing their solutions to avoid repetitive computations.

Let's understand this approach through an example.

Consider an example of the Fibonacci series. The following series is the Fibonacci series:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ,…
```

The numbers in the above series are not randomly calculated. Mathematically, we could write each of the terms using the below formula:

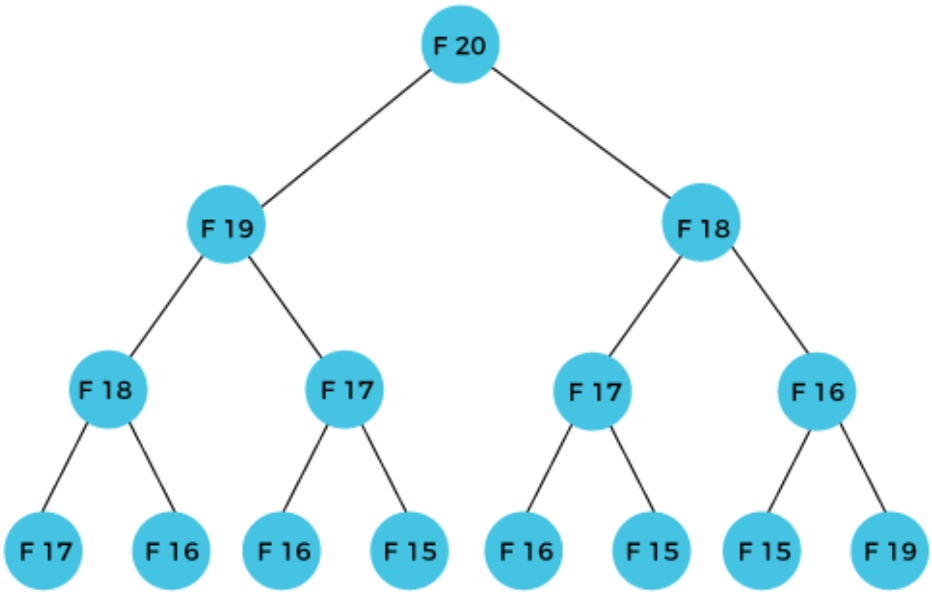
```
F(n) = F(n-1) + F(n-2),
```

With the base values F(0) = 0, and F(1) = 1.

To calculate the other numbers, we follow the above relationship. For example, F(2) is the sum f(0) and f(1), which is equal to 1.

How can we calculate F(20)?

The F(20) term will be calculated using the nth formula of the Fibonacci series. The below figure shows that how F(20) is calculated.



As we can observe in the above figure that F(20) is calculated as the sum of F(19) and F(18).

In the dynamic programming approach, we try to divide the problem into the similar subproblems. We are following this approach in the above case where F(20) into the similar subproblems, i.e., F(19) and F(18). If we revisit the definition of dynamic programming that it says the similar subproblem should not be computed more than once. Still, in the above case, the subproblem is calculated twice. F(18) is calculated two times; similarly, F(17) is also calculated twice. However, this technique is quite useful as it solves the similar subproblems, but we need to be cautious while storing the results because we are not particular about storing the result that we have computed once, as it can lead to a wastage of resources.

In the above example, if we calculate the F(18) in the right subtree, then it leads to the tremendous usage of resources and decreases the overall performance.

The solution to the above problem is to save the computed results in an array. First, we calculate F(16) and F(17) and save their values in an array. The F(18) is calculated by summing the values of F(17) and F(16), which are already saved in an array. The computed value of F(18) is saved in an array. The value of F(19) is calculated using the sum of F(18), and F(17), and their values are already saved in an array. The computed value of F(19) is stored in an array. The value of F(20) can be calculated by adding the values of F(19) and F(18), and the values of both F(19) and F(18) are stored in an array. The final computed value of F(20) is stored in an array.

3.1 How Dynamic Programming Works

The following are the steps that the dynamic programming follows:

- It breaks down the complex problem into simpler subproblems.
- It finds the optimal solution to these sub-problems.
- It stores the results of subproblems (memoization). The process of storing the results of subproblems is known as memorization.
- It reuses them so that same sub-problem is calculated more than once.

- Finally, calculate the result of the complex problem.

The above five steps are the basic steps for dynamic programming. The dynamic programming is applicable that are having properties such as:

- Those problems that are having overlapping subproblems and optimal substructures. Here, optimal substructure means that the solution of optimization problems can be obtained by simply combining the optimal solution of all the subproblems.

In the case of dynamic programming, the space complexity would be increased as we are storing the intermediate results, but the time complexity would be decreased.

3.2 Approaches of dynamic programming

There are two approaches to dynamic programming:

- Top-down approach
- Bottom-up approach

3.2.1 Top-down approach

The top-down approach follows the memorization technique, while bottom-up approach follows the tabulation method. Here memorization is equal to the sum of recursion and caching. Recursion means calling the function itself, while caching means storing the intermediate results.

Advantages of Top-down aproach

- It is very easy to understand and implement.
- It solves the subproblems only when it is required.
- It is easy to debug.

Disadvantages of Top-down approach

- It uses the recursion technique that occupies more memory in the call stack. Sometimes when the recursion is too deep, the stack overflow condition will occur.
- It occupies more memory that degrades the overall performance.

Let's understand dynamic programming through an example.

```
int fib(int n)
{
   if(n<0)
   error;
 if(n==0)
 return 0;
 if(n==1)
return 1;
sum = fib(n-1) + fib(n-2);
}
```

In the above code, we have used the recursive approach to find out the Fibonacci series. When the value of 'n' increases, the function calls will also increase, and computations will also increase. In this case, the time complexity increases exponentially, and it becomes $O(2^n)$.

Another solution to this problem is to use the dynamic programming approach. Rather than generating the recursive tree again and again, we can reuse the previously calculated value. If we use the dynamic programming approach, then the time complexity would be O(n).

When we apply the dynamic programming approach in the implementation of the Fibonacci series, then the code would look like:

```
static int count = 0;
int fib(int n)
{
if(memo[n]!= NULL)
return memo[n];
count++;
if(n<0)
```

```
error;
if(n==0)
return 0;
if(n==1)
return 1;
sum = fib(n-1) + fib(n-2);
memo[n] = sum;
}
```

In the above code, we have used the memorization technique in which we store the results in an array to reuse the values. This is also known as a top-down approach in which we move from the top and break the problem into sub-problems.

3.2.2 Bottom-Up approach

The bottom-up approach uses the tabulation technique to implement the dynamic programming approach. It solves the same kind of problems but it removes the recursion. If we remove the recursion, there is no stack overflow issue and no overhead of the recursive functions. In this tabulation technique, we solve the problems and store the results in a matrix.

The bottom-up is the approach used to avoid the recursion, thus saving the memory space. The bottom-up is an algorithm that starts from the beginning, whereas the recursive algorithm starts from the end and works backward. In the bottom-up approach, we start from the base case to find the answer for the end. As we know, the base cases in the Fibonacci series are 0 and 1. Since the bottom approach starts from the base cases, so we will start from 0 and 1.

Key points of Bottom-up approach

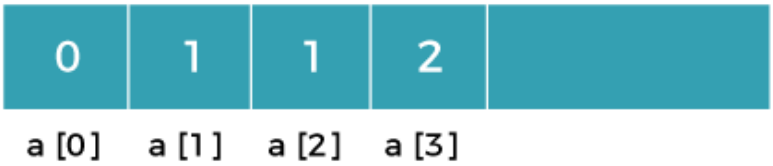
- We solve all the smaller sub-problems that will be needed to solve the larger sub-problems then move to the larger problems using smaller sub-problems.
- We use for loop to iterate over the sub-problems.
- The bottom-up approach is also known as the tabulation or table filling method.

Let's understand through an example.

Suppose we have an array that has 0 and 1 values at a[0] and a[1] positions, respectively shown as below:

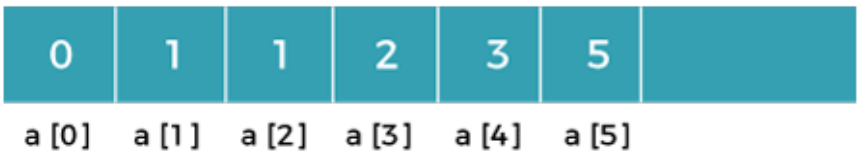
Since the bottom-up approach starts from the lower values, so the values at a[0] and a[1] are added to find the value of a[2] shown as below:

The value of a[3] will be calculated by adding a[1] and a[2], and it becomes 2 shown as below:



The value of a[4] will be calculated by adding a[2] and a[3], and it becomes 3 shown as below:

The value of a[5] will be calculated by adding the values of a[4] and a[3], and it becomes 5 shown as below:



The code for implementing the Fibonacci series using the bottom-up approach is given below:

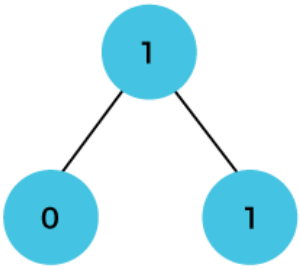
```
int fib(int n)
{
    int A[];
    A[0] = 0, A[1] = 1;
    for( i=2; i<=n; i++)
    {
         A[i] = A[i-1] + A[i-2]
    }
    return A[n];
}
```

In the above code, base cases are 0 and 1 and then we have used for loop to find other values of Fibonacci series.

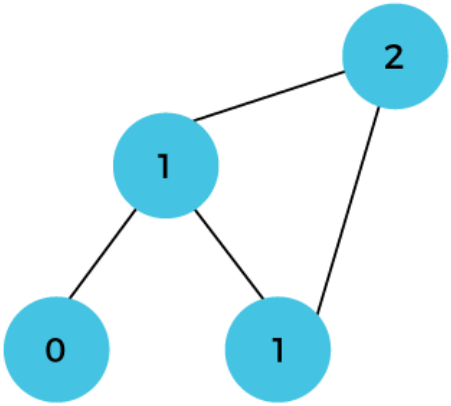
Let's explain better using the following diagrammatic representation:

Initially, the first two values, i.e., 0 and 1 can be represented as:

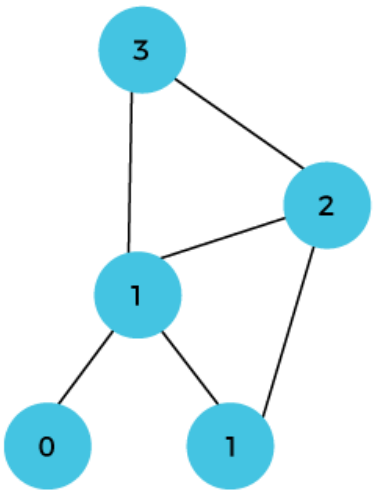
When i=2 then the values 0 and 1 are added shown as below:



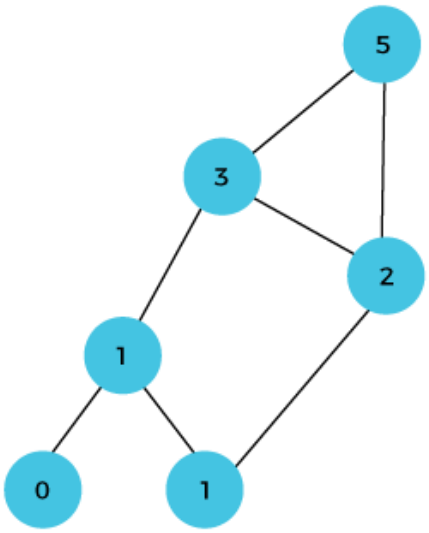
When i=3 then the values 1and 1 are added shown as below:



When i=4 then the values 2 and 1 are added shown as below:



When i=5, then the values 3 and 2 are added shown as below:



In the above case, we are starting from the bottom and reaching to the top

3.3 Divide and Conquer Method versus Dynamic Programming:

We highlight some of the differences between Divide-and-Conquer approach and Dynamic Programming.

Divide and Conquer Method	**Dynamic Programming**
1.It deals (involves) three steps at each level of recursion: **Divide** the problem into a number of subproblems. **Conquer** the subproblems by solving them recursively. **Combine** the solution to the subproblems into the solution for original subproblems.	**1.**It involves the sequence of four steps: • Characterize the structure of optimal solutions. • Recursively defines the values of optimal solutions. • Compute the value of optimal solutions in a Bottom-up minimum. • Construct an Optimal Solution from computed information.
2. It is Recursive.	**2.** It is non Recursive.

3. It does more work on subproblems and hence has more time consumption.	**3.** It solves subproblems only once and then stores in the table.
4. It is a top-down approach.	**4.** It is a Bottom-up approach.
5. In this subproblems are independent of each other.	**5.** In this subproblems are interdependent.
6. For example: Merge Sort & Binary Search etc.	**6. For example:** Matrix Multiplication.

3.4 Techniques for Solving Dynamic Programming Problems

To solve any dynamic programming problem, we can use the FAST method.

Here, FAST stands for:

- **'F'** stands for **Find the recursive solution:** Whenever we find any DP problem, we have to find the recursive solution.
- **'A'** stands for **Analyse the solution:** Once we find the recursive solution then we have to analyse the solution and look for the overlapping problems.
- **'S'** stands for **Save the results for future use:** Once we find the overlapping problems, we store the solutions of these sub-problems. To store the solutions, we use the n-dimensional array for caching purpose.

The above three steps are used for the top-down approach if we use 'F', 'A' and 'S', which means that we are achieving the Top-down approach and since it is not purely because we are using the recursive technique.

- **'T'** stands for **Tweak the solution** to make it more powerful by eliminating recursion overhead which is known as a Bottom-up approach. Here we remove the recursion technique and use the iterative approach to achieve the same results, so it's a pure approach. Recursion is always an overhead as there are chances of getting a stack overflow error, so we should use the bottom-up approach to avoid this problem.

Above are the four steps to solve a complex problem.

Problem Statement: Write an efficient program to find the nth Fibonacci number?

As we know that Fibonacci series looks like:

0, 1, 1, 2, 3, 5, 8, 13, 21,...

First, we find the recursive solution,

$$\text{Fib}(n) = \begin{cases} n & n<2 \\ \text{Fib}(n-1) + \text{Fib}(n-2) & \text{otherwise} \end{cases}$$

The below is the code of the above recursive solution:

```
Fib(n)
{
    if(n<2)
    return n;
   return fib(n-1) + fib(n-2);
}
```

The above recursive solution is also the solution for the above problem but the time complexity in this case is $O(2^n)$. So, dynamic programming is used to reduce the time complexity from the exponential time to the linear time.

Second step is to Analyse the solution

Suppose we want to calculate the fib(4).

```
Fib(4)= fib(3) + fib(2)

Fib(3) = fib(2) + fib(1)

Fib(2) = fib(1) + fib(0)
```

As we can observe in the above figure that fib(2) is calculated two times while fib(1) is calculated three times. So, here overlapping problem occurs. In this step, we have analysed the solution.

Third step is to save the result.

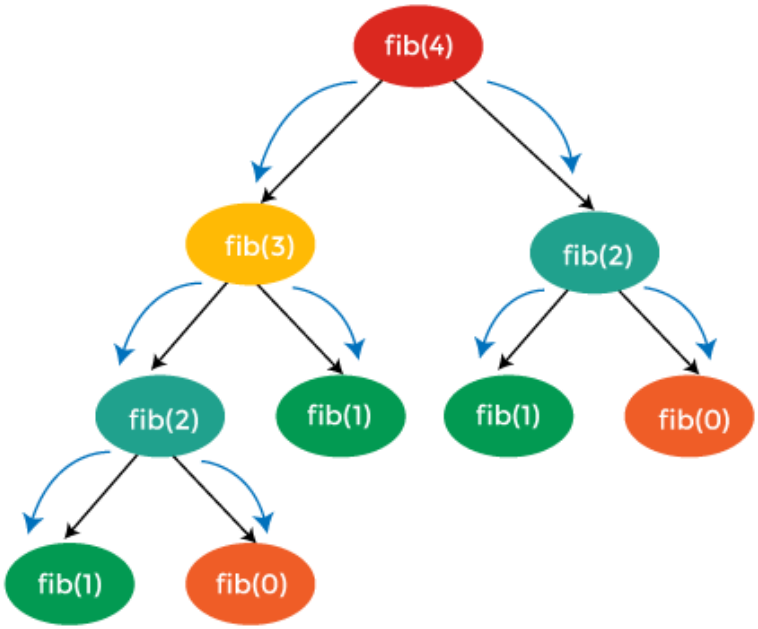
The process of saving the result is known as memoization. In this step, we will follow the same approach, i.e., recursive approach but with a small different that we have used the cache to store the solutions so that it can be re-used whenever required.

Below is the code of memorization.

```
Fib(n)
{
     int cache = new int[n+1];
     if(n<2)
     return n;
    if(cache[n]!= 0)
    return cache[n];
  return cache[n] =  fib(n-1) + fib(n-2);
}
```

In the above code, we have used a cache array of size n+1. If cache[n] is not equal to zero then we return the result from the cache else we will calculate the value of cache and then return the cache. The technique that we have used here is top-down approach as it follows the recursive approach. Here, we always look for the cache so cache will be populated on the demand basis. Suppose we want to calculate the fib(4), first we look into cache, and if the value is not in the cache then the value is calculated and stored in the cache.

Visual representation of the above code is:



Fourth step is to Tweak the solution

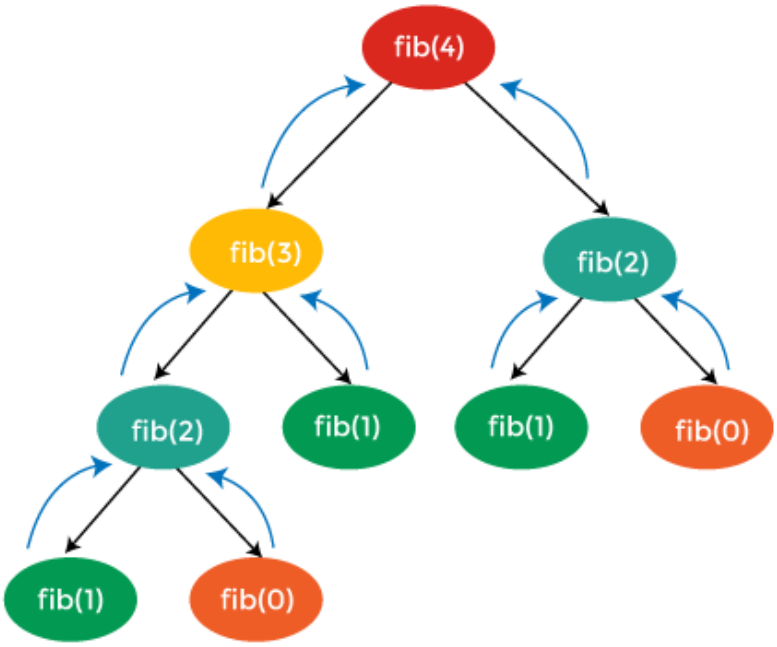
In this step, we will remove the recursion completely and make it an iterative approach. So, this technique is known as a bottom-up approach.

```
Fib(n)
{
    int cache[] = new int[n+1];
    // base cases
    cache[0] = 0;
    cache[1] = 1;
    for(int i=2; i<=n; i++)
    {
      cache[i] = cache[i-1] + cache[i-2];
    }
return cache[n];
}
```

In the above code, we have followed the **bottom-up** approach. We have declared a cache array of size n+1. The base cases are cache[0] and cache[1] with their values 0 and 1 respectively. In the above code, we have removed the recursion completely. We have used an iterative approach. We have defined a for loop in which we populate the cache with the values from the index i=2 to n, and from the cache, we will return the result. Suppose we want to calculate f(4), first we will calculate f(2), then we will calculate f(3) and finally, we we calculate the

value of f(4). Here we are going from down to up so this approach is known as a bottom-up approach.

We can visualize this approach diagrammatically:



As we can observe in the above figure that we are populating the cache from bottom to up so it is known as bottom-up approach. This approach is much more efficient than the previous one as it is not using recursion but both the approaches have the same time and space complexity, i.e., O(n).

In this case, we have used the **FAST** method to obtain the optimal solution. The above is the optimal solution that we have got so far but this is not the purely an optimal solution.

Efficient solution:

```
fib(n)
{
    int first=0, second=1, sum=0;
    if(n<2)
   {
      return 0;
 }
for(int i =2; i<=n; i++)
{
    sum = first + second;
    first = second;
```

```
        second = sum;
    }
    return sum;
    }
```

The above solution is the efficient solution as we do not use the cache.

The Following are the top 10 problems that can easily be solved using Dynamic programming:

a. Longest Common Subsequence.
b. Shortest Common Supersequence.
c. Longest Increasing Subsequence problem.
d. The Levenshtein distance (Edit distance) problem.
e. Matrix Chain Multiplication.
f. 0–1 Knapsack problem.
g. Partition problem.
h. Rod Cutting.

Self-Assessment Exercises

1. When do we consider using the dynamic programming approach?

2. Four matrices M1, M2, M3 and M4 of dimensions pxq, qxr, rxs and sxt respectively can be multiplied is several ways with different number of total scalar multiplications. For example, when multiplied as ((M1 X M2) X (M3 X M4)), the total number of multiplications is pqr + rst + prt. When multiplied as (((M1 X M2) X M3) X M4), the total number of scalar multiplications is pqr + prs + pst. If p = 10, q = 100, r = 20, s = 5 and t = 80, then the number of scalar multiplications needed is?

3. Consider two strings A = "qpqrr" and B = "pqprqrp". Let x be the length of the longest common subsequence (not necessarily contiguous) between A and B and let y be the number of such longest common subsequences between A and B. Then x + 10y =?

4. In dynamic programming, the technique of storing the previously calculated values is called?

5. What happens when a top-down approach of dynamic programming is applied to a problem?

4.0 Conclusion

Dynamic programming is nothing but recursion with memoization i.e. calculating and storing values that can be later accessed to solve subproblems that occur again, hence making your code faster and reducing the time complexity (computing CPU cycles are reduced). Dynamic programming is used where we have problems, which can be divided into similar sub-problems, so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems.

5.0 Summary

In this Unit, we considered a very important algorithm design paradigm known as Dynamic programming and compared it with another useful method known as Divide-and-Conquer technique. Several ways for resolving the Dynamic Programming problem were considered.

6.0 Tutor Marked Assignments

1. For each of the following problems, explain whether they could be solved or not using dynamic programming?
   A : Mergesort B : Binary search
   C : Longest common subsequence D : Quicksort
2. Give at least three properties of a dynamic programming problem
3. You are given infinite coins of denominations 1, 3, 4. What is the total number of ways in which a sum of 7 can be achieved using these coins if the order of the coins is not important?
4. What is the main difference between the Top-down and Bottom-up approach for solving Dynamic Programming problems?

7.0 Further Reading and Other Resources

Baase, S. and Van Gelder, A. (2020). Computer Algorithms: Introduction to Design and Analysis. Addison-Wesley. ISBN: 0201612445, 9780201612448

Bhasin, H. (2015). Algorithms: Design and Analysis. Oxford University Press. ISBN: 0199456666, 9780199456666

Sen, S. and Kumar, A, (2019). Design and Analysis of Algorithms. A Contemporary Perspective. Cambridge University Press. ISBN: 1108496822, 9781108496827

Vermani, L. R. and Vermani, S.(2019). An Elementary Approach To Design And Analysis Of Algorithms. World Scientific. ISBN: 178634677X, 9781786346773

Module 3: Other Algorithm Techniques

Unit 3: Computational Complexity

1.0 Introduction

In general, the amount of resources (or cost) that an algorithm requires in order to return the expected result is called computational complexity or just complexity. ... The complexity of an algorithm can be measured in terms of time complexity and/or space complexity.

Computational complexity theory focuses on classifying computational problems according to their resource usage, and relating these classes to each other. A computational problem is a task solved by a computer. A computation problem is solvable by mechanical application of mathematical steps, such as an algorithm.

A problem is regarded as inherently difficult if its solution requires significant resources, whatever the algorithm used.

We shall be looking at the famous P (Polynomial Time) and NP (Non Polynomial Time) as well as NP-complete problems.

2.0 Objectives

By the end of this unit, you should be able to:

- Know the meaning and focus of Computational Complexity theory
- Identify the different cases of P and NP problems
- Differentiate between Tractable and Intractable problems
- Know what we mean by Deterministic and Non-Deterministic problems
- Understand the differences between Deterministic and Non Deterministic algorithms

3.0 Computational Complexity Theory

An algorithm's performance is always important when you try to solve a problem. An algorithm won't do you much good if it takes too long or requires too much memory or other resources to actually run on a computer.

Computational complexity theory, or just *complexity theory*, is the study of the difficulty of computational problems. Rather than focusing on specific algorithms, complexity theory focuses on problems.

For example, the mergesort algorithm can sort a list of N numbers in O(N log N) time. Complexity theory asks what you can learn about the task of sorting in general, not what you can learn about a specific algorithm. It turns out that you can show that any sorting algorithm that sorts by using comparisons must use at least $N \times \log(N)$ time in the worst case.

Complexity theory is a large and difficult topic, so there's no room here to cover it fully. However, every programmer who studies algorithms should know at least something about complexity theory in general and the two sets P and NP in

particular. This module introduces complexity theory and describes what these important classes of problems are.

3.0.1 Notations Used

The **Big O notation** describes how an algorithm's worst-case performance increases as the problem's size increases.
For most purposes, that definition is good enough to be useful, but in complexity theory Big O notation has a more technical definition.
If an algorithm's run time is f (N) , then the algorithm has Big O performance of g(N) if f(N) < g(N) x k for some constant k and for N large enough. In other words, the function g(N) is an upper bound for the actual run-time function f(N).
.
Two other notations similar to Big O notations are sometimes useful when discussing algorithmic complexity.

Big Omega notation, written Ω(g(N)), means that the run-time function is bounded *below* by the function g(N) . For example, as explained a moment ago, N log(N) is a lower bound for algorithms that sort by using comparisons, so those algorithms are Ω(N logN) .

Big Theta notation, written Θ(g(N)) , means that the run-time function is bounded both above and below by the function g(N) . For example, the mergesort algorithm's run time is bounded above by O(N log N), and the run time of any algorithm that sorts by using comparisons is bounded below by Ω(N log N), so mergesort has performance Θ(N log N).

In summary,
Big O notation gives an upper bound,
Big Omega gives a lower bound, and
Big Theta gives an upper and lower bound.

Some algorithms however, have different upper and lower bounds.
For example, like all algorithms that sort by using comparisons, quicksort has a lower bound of Ω(N log N).
In the best and expected cases, quicksort's performance actually is Ω(N log N).
In the worst case, however, quicksort's performance is $O(N^2)$.
The algorithm's lower and upper bounds are different, so no function gives quicksort a Big Theta notation.
In practice, however, quicksort is often faster than algorithms such as mergesort that are tightly bounded by Θ(N log N), so it is still a popular algorithm.

3.1 Deterministic Algorithms

A Deterministic algorithm is an algorithm which, given a particular input will always produce the same output, with the underlying machine always passing through the same sequence of states.

In other words, Deterministic algorithm will always come up with the same result given the same inputs.

Deterministic algorithms are by far the most studied and familiar kind of algorithm as well as one of the most practical, since they can be run on real machines efficiently.

Formally, a deterministic algorithm computes a mathematics function; a function has a unique value for any input in its domain, and the algorithm is a process that produces this particular value as output.

Deterministic algorithms can be defined in terms of a state machine: a *state* describes what a machine is doing at a particular instant in time. State machines pass in a discrete manner from one state to another. Just after we enter the input, the machine is in its *initial state* or *start state*. If the machine is deterministic, this means that from this point onwards, its current state determines what its next state will be; its course through the set of states is predetermined. Note that a machine can be deterministic and still never stop or finish, and therefore fail to deliver a result.

Examples of particular abstract machines which are deterministic include the **deterministic Turing machine** and **deterministic finite automat**

3.1.1 Facts about Deterministic Algorithms

i. Deterministic algorithm is the algorithm which, given a particular input will always produce the same output, with the underlying machine always passing through the same sequence of states.
ii. In deterministic algorithm the path of execution for algorithm is same in every execution.

iii. On the basis of execution and outcome in case of Deterministic algorithm, they are also classified as reliable algorithms as for a particular input instructions the machine will give always the same output.
iv. In Deterministic Algorithms execution, the target machine executes the same instruction and results same outcome which is not dependent on the way or process in which instruction get executed.
v. As outcome is known and is consistent on different executions so deterministic algorithm takes polynomial time for their execution.

3.2 Non-deterministic Algorithms

A nondeterministic algorithm is an algorithm that, even for same input, can exhibit different behaviors on different runs.

In other words, it is an algorithm in which the result of every algorithm is not uniquely defined and result could be random.

An algorithm that solves a problem in nondeterministic polynomial time can run in polynomial time or exponential time depending on the choices it makes during. The nondeterministic algorithms are often used to find an approximation to a solution, when the exact solution would be too costly using a deterministic one.

A nondeterministic algorithm is different from its more familiar deterministic counterpart in its ability to arrive at outcomes using various routes. If a deterministic algorithm represents a single path from an input to an outcome, a nondeterministic algorithm represents a single path stemming into many paths, some of which may arrive at the same output and some of which may arrive at unique outputs.

3.2.1 What Makes An Algorithm Non-deterministic?

A variety of factors can cause an algorithm to behave in a way which is not deterministic, or non-deterministic:

i. If it uses external state other than the input, such as user input, a global variable, a hardware timer value, a random value, or stored disk data.
ii. If it operates in a way that is timing-sensitive, for example if it has multiple processors writing to the same data at the same time. In this case, the precise order in which each processor writes its data will affect the result.
iii. If a hardware error causes its state to change in an unexpected way.

3.2.2 Facts About Non-deterministic Algorithms

i. A Non-deterministic algorithm is the algorithms in which the result of every algorithm is not uniquely defined and result could be random.
ii. In a Non-Deterministic algorithm the path of execution is not same for algorithm in every execution and could take any random path for its execution.
iii. Non deterministic algorithms are classified as non-reliable algorithms for a particular input the machine will give different output on different executions.
iv. In Non-Deterministic Algorithms, the machine executing each operation is allowed to choose any one of these outcomes subjects to a determination condition to be defined later.
v. As outcome is not known and is non-consistent on different executions so Non-Deterministic algorithm could not get executed in polynomial time.

3.2.3 Deterministic versus Non-deterministic Algorithms

The following table gives some vital differences between a Deterministic and a Non Deterministic algorithm.

BASIS OF COMPARISON	DETERMINISTIC ALGORITHM	NON-DETERMINISTIC ALGORITHM
Description.	Deterministic algorithm is the algorithm which, given a particular input will always produce the same output, with the underlying machine always passing through the same sequence of states.	Non-deterministic algorithm is the algorithms in which the result of every algorithm is not uniquely defined and result could be random.
Path Of Execution	In deterministic algorithm the path of execution for algorithm is same in every execution.	In Non-Deterministic algorithm the path of execution is not same for algorithm in every execution and could take any random path for its execution.
Basis Of Comparison	On the basis of execution and outcome in case of Deterministic algorithm, they are also classified as reliable algorithms as for a particular input instructions the machine will give always the same output.	Non deterministic algorithms are classified as non-reliable algorithms for a particular input the machine will give different output on different executions.
Operation	In Deterministic Algorithms execution, the target machine executes the same instruction and results same outcome which is not dependent on the way or process in which instruction get executed.	In Non-Deterministic Algorithms, the machine executing each operation is allowed to choose any one of these outcomes subjects to a determination condition to be defined later.
Output	As outcome is known and is consistent on different executions so deterministic algorithm takes polynomial time for their execution.	As outcome is not known and is non-consistent on different executions so Non-Deterministic algorithm could not get executed in polynomial time.

3.3 NP (Non-Deterministic Polynomial) Problem

The set of all decision-based problems came into the division of NP Problems who can't be solved or produced an output within polynomial time but verified in

the **polynomial time**. NP class contains P class as a subset. NP problems are very hard to solve.
Note: The term “NP” does not mean “Not Polynomial”. Originally, the term meant “Non-Deterministic Polynomial. It means according to the one input number of output will be produced.

3.3.1 Definition of P Problems

Definition of P class Problem: - The set of decision-based problems come into the division of P Problems who can be solved or produced an output within polynomial time. P problems being easy to solve

Definition of Polynomial time: - If we produce an output according to the given input within a specific amount of time such as within a minute, hours. This is known as Polynomial time.

Definition of Non-Polynomial time: - If we produce an output according to the given input but there are no time constraints is known as Non-Polynomial time. But yes output will produce but time is not fixed yet.

3.2 Decision Based Problems

A problem is called a decision problem if its output is a simple "yes" or "no" (or you may need to represent it as true/false, 0/1, accept/reject.) We will phrase many optimization problems as decision problems.

For example, Greedy method, D.P., given a graph G= (V, E) if there exists any Hamiltonian cycle.

3.4.1 NP-hard Problems

A problem is NP-hard if an algorithm for solving it can be translated into one for solving any NP- problem (nondeterministic polynomial time) problem. NP-hard therefore means "at least as hard as any NP-problem," although it might, in fact, be harder.

A problem must satisfy the following points to be classified as NP-hard

1. If we can solve this problem in polynomial time, then we can solve all NP problems in polynomial time
2. If you convert the issue into one form to another form within the polynomial time

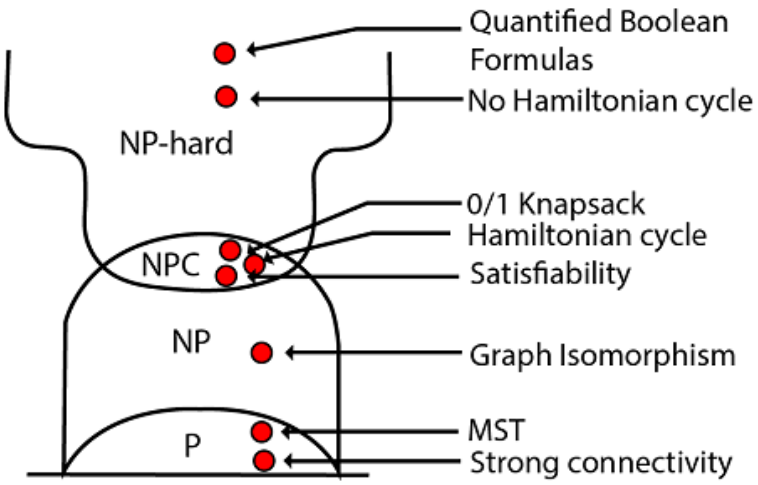
3.4.2 NP-complete Problems:

A problem is NP-complete when: it is a problem for which the correctness of each solution can be verified quickly and a brute-force search algorithm can find a solution by trying all possible solutions.

A problem is in the class NP-complete if it is in NP and is as hard as any problem in NP. A problem is NP-hard if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself. These problems are called NP-complete.

Many significant computer-science problems belong to this class—e.g., the traveling salesman problem, satisfiability problems, and graph-covering problems.

3.4.3 Pictorial representation of all NP classes



3.5 Tractable and Intractable Problems

3.5.1 Tractable Problem:

A problem that is solvable by a polynomial-time algorithm. The upper bound is polynomial.
Here are **examples** of tractable problems (ones with known polynomial-time algorithms):
– Searching an unordered list
– Searching an ordered list
– Sorting a list
– Multiplication of integers (even though there's a gap)
– Finding a minimum spanning tree in a graph (even though there's a gap)

3.5.2 Intractable Problem:

A problem that cannot be solved by a polynomial-time algorithm. The lower bound is exponential.

From a computational complexity stance, intractable problems are problems for which there exist no efficient algorithms to solve them. Most intractable problems have an algorithm that provides a solution, and that algorithm is the brute-force search.

This algorithm, however, does not provide an efficient solution and is, therefore, not feasible for computation with anything more than the smallest input.

Examples
Towers of Hanoi: we can prove that any algorithm that solves this problem must have a worst-case running time that is at least $2^n - 1$.
* List all permutations (all possible orderings) of n numbers.

3.5.3 IS P = NP?

The **P versus NP problem** is a major unsolved problem in computer science. It asks whether every problem whose solution can be quickly verified can also be solved quickly.

An answer to the P versus NP question would determine whether problems that can be verified in polynomial time can also be solved in polynomial time.

If it turns out that P ≠ NP, which is widely believed, it would mean that there are problems in NP that are harder to compute than to verify: they could not be solved in polynomial time, but the answer could be verified in polynomial time.

If P=NP, then all of the NP problems can be solved deterministically in Polynomial time.
The Clay Mathematics Institute has offered a $1,000,000 prize to anyone who proves or disproves P = NP.

Self Assessment Exercise

1. Differentiate between a P problem and an NP problem
2. What are NP-hard problems?
3. Give three examples of Tractable problems
4. What are the features of Deterministic problems?

4.0 Conclusion

Computational complexity theory focuses on classifying computational problems according to their resource usage, and relating these classes to each other. A computational problem is a task solved by a computer. A computation problem is solvable by mechanical application of mathematical steps, such as an algorithm. Several areas considered in this Unit were P and NP problems, Deterministic versus Non Deterministic problems as well as Tractable versus Intractable problems.

5.0 Summary

In this Unit we looked at the meaning and nature of Computational Complexity theory and also examined the notion of Deterministic as well as Non Deterministic algorithms. Several examples of the algorithms were listed and we also treated P, NP, NP-hard and NP-complete problems while also mentioning Tractable and Intractable problems. On a final note, we also looked at the unsolvable problem of P = NP.

6.0 Tutor Marked Assignment

1. What would be the implication of having P = NP?

2. What again would happen if P≠NP?
3. What makes exponential time and algorithms with factorials more difficult to solve?
4. How many stages of procedure does a Non deterministic algorithm consist of?

7.0 Further Reading and other Resources

Baase, S. and Van Gelder, A. (2020). Computer Algorithms: Introduction to Design and Analysis. Addison-Wesley. ISBN: 0201612445, 9780201612448

Bhasin, H. (2015). Algorithms: Design and Analysis. Oxford University Press. ISBN: 0199456666, 9780199456666

Sen, S. and Kumar, A, (2019). Design and Analysis of Algorithms. A Contemporary Perspective. Cambridge University Press. ISBN: 1108496822, 9781108496827

Vermani, L. R. and Vermani, S.(2019). An Elementary Approach To Design And Analysis Of Algorithms. World Scientific. ISBN: 178634677X, 9781786346773

Module 3: Other Algorithm Techniques

Unit 4: Approximate Algorithms I

1.0 Introduction

In computer science and operations research, approximation algorithms are efficient algorithms that find approximate solutions to optimization problems (in particular NP-hard problems) with provable guarantees on the distance of the returned solution to the optimal one. Approximation algorithms are typically used when finding an optimal solution is intractable, but can also be used in some situations where a near-optimal solution can be found quickly and an exact solution is not needed.

2.0 Objectives

At the end of this Unit, you should be able to;

- Know the meaning of an Approximate algorithm
- Understand the performance ratio of approximate algorithms
- Learn more about the Vertex Cover and Traveling Salesman problems
- Understand the concept of Minimal Spanning Trees
- Understand more of the concept of Performance Ratios

3.0 Approximate Algorithms

An Approximate Algorithm is a way of approach **NP-COMPLETENESS** for the optimization problem. This technique does not guarantee the best solution. The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at the most polynomial time. Such algorithms are called approximation algorithm or heuristic algorithm.

- For the traveling salesperson problem, the optimization problem is to find the shortest cycle, and the approximation problem is to find a short cycle.
- For the vertex cover problem, the optimization problem is to find the vertex cover with fewest vertices, and the approximation problem is to find the vertex cover with few vertices.

3.0.1 Performance Ratios

Suppose we work on an optimization problem where every solution carries a cost. An Approximate Algorithm returns a legal solution, but the cost of that legal solution may not be optimal.

For Example, suppose we are considering for a **minimum size vertex-cover (VC)**. An approximate algorithm returns a VC for us, but the size (cost) may not be minimized.

Another Example is we are considering for a **maximum size Independent set (IS)**. An approximate Algorithm returns an IS for us, but the size (cost) may not be maximum. Let C be the cost of the solution returned by an approximate algorithm, and C* is the cost of the optimal solution.

We say the approximate algorithm has an approximate ratio P (n) for an input size n, where

$$\max\left(\frac{C}{C^*}, \frac{C^*}{C}\right) \leq P(n)$$

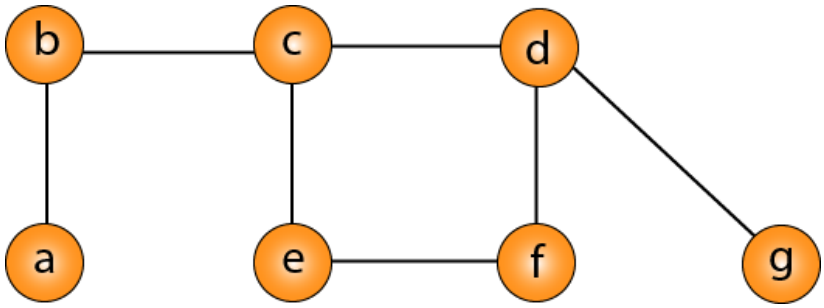
Intuitively, the approximation ratio measures how bad the approximate solution is distinguished with the optimal solution. A large (small) approximation ratio measures the solution is much worse than (more or less the same as) an optimal solution.

Observe that P (n) is always ≥ 1, if the ratio does not depend on n, we may write P. Therefore, a 1-approximation algorithm gives an optimal solution. Some problems have polynomial-time approximation algorithm with small constant approximate ratios, while others have best-known polynomial time approximation algorithms whose approximate ratios grow with n.

3.1 Vertex Cover

A Vertex Cover of a graph G is a set of vertices such that each edge in G is incident to at least one of these vertices.

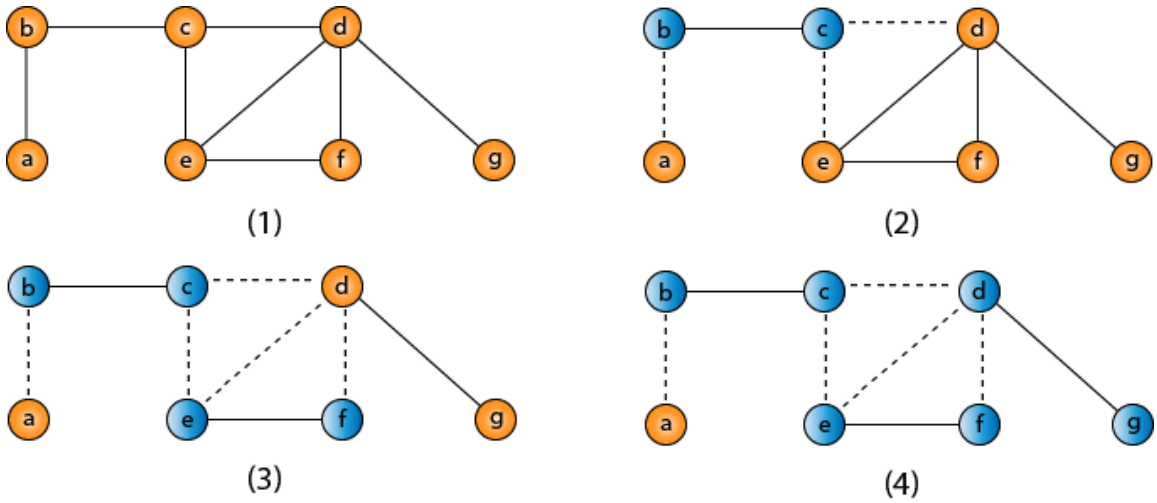
The decision vertex-cover problem was proven NPC. Now, we want to solve the optimal version of the vertex cover problem, i.e., we want to find a minimum size vertex cover of a given graph. We call such vertex cover an optimal vertex cover C*.



An approximate algorithm for vertex cover:

```
Approx-Vertex-Cover (G = (V, E))
{
       C = empty-set;
    E'= E;
    While E' is not empty do
      {
    Let (u, v) be any edge in E': (*)
    Add u and v to C;
    Remove from E' all edges incident to
       u or v;
      }
    Return C;
}
```

The idea is to take an edge (u, v) one by one, put both vertices to C, and remove all the edges incident to u or v. We carry on until all edges have been removed. C is a VC. But how good is C?



VC = {b, c, d, e, f, g}

3.2 Traveling-salesman Problem

In the traveling salesman Problem, a salesman must visits n cities. We can say that salesman wishes to make a tour or Hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. There is a non-negative cost c (i, j) to travel from the city i to city j. The goal is to find a tour of minimum cost. We assume that every two cities are connected. Such problems are called Traveling-salesman problem (TSP).

We can model the cities as a complete graph of n vertices, where each vertex represents a city.

It can be shown that TSP is NPC.

If we assume the cost function c satisfies the triangle inequality, then we can use the following approximate algorithm.

Triangle inequality

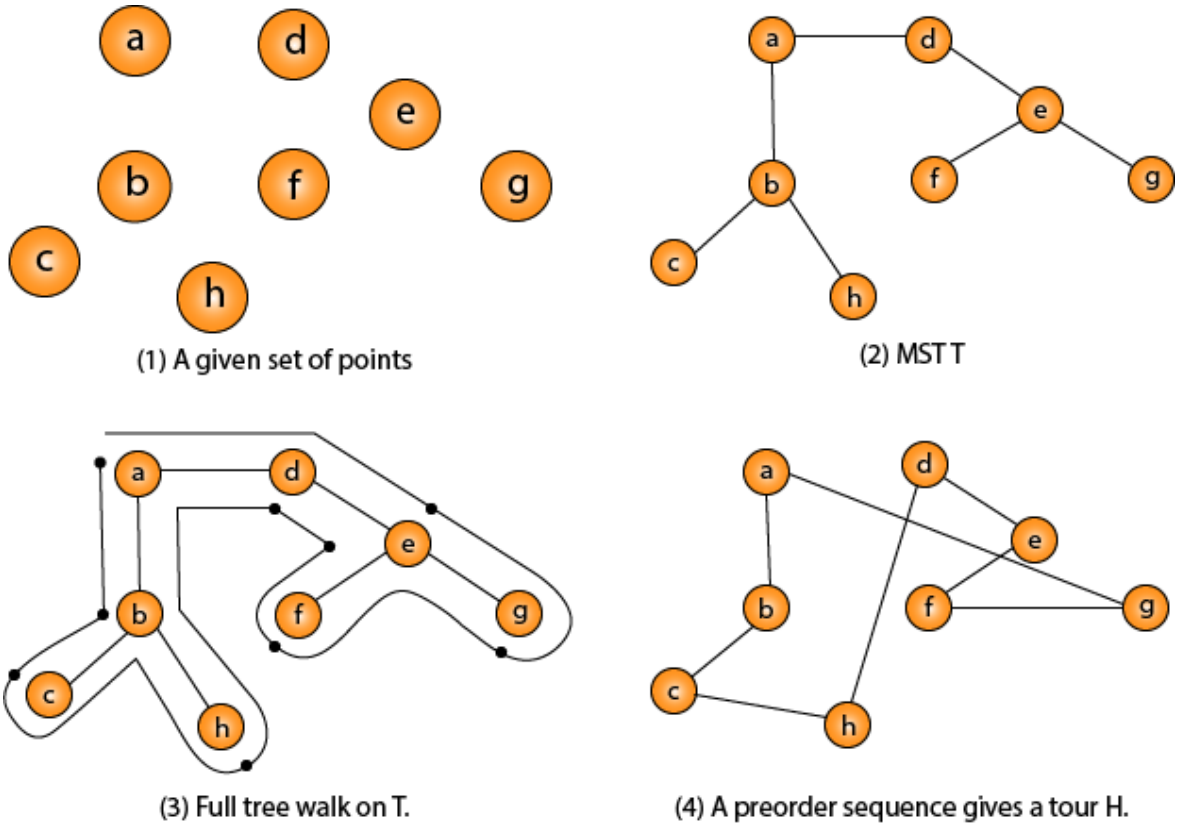
Let u, v, w be any three vertices, we have

$$c(u, w) \leq c(u, v) + c(v, w)$$

One important observation to develop an approximate solution is if we remove an edge from H*, the tour becomes a spanning tree.

```
Approx-TSP (G= (V, E))
{
  1. Compute a MST T of G;
  2. Select any vertex r is the root of the tree;
  3. Let L be the list of vertices visited in a preorder
tree walk of T;
  4. Return the Hamiltonian cycle H that visits the verti
ces in the order L;
}
```

The Traveling-salesman Problem

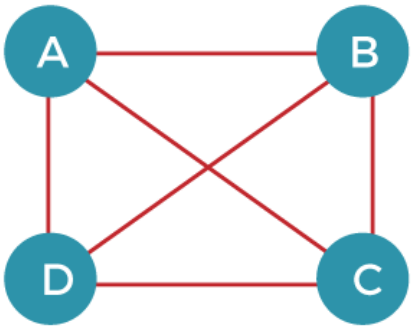


Intuitively, Approx-TSP first makes a full walk of MST T, which visits each edge exactly two times. To create a Hamiltonian cycle from the full walk, it bypasses some vertices (which corresponds to making a shortcut)

3.3 Minimum Spanning Tree

Before knowing about the minimum spanning tree, we should know about the spanning tree.

To understand the concept of spanning tree, consider the graph below:



The above graph can be represented as G(V, E), where 'V' is the number of vertices, and 'E' is the number of edges. The spanning tree of the above graph would be represented as G`(V`, E`). In this case, V` = V means that the number of vertices in the spanning tree would be the same as the number of vertices in the graph, but the number of edges would be different. The number of edges in the spanning tree is the subset of the number of edges in the original graph. Therefore, the number of edges can be written as:

```
E` € E
```

It can also be written as:

```
E` = |V| - 1
```

Two conditions exist in the spanning tree, which is as follows:

- The number of vertices in the spanning tree would be the same as the number of vertices in the original graph.

  V` = V

- The number of edges in the spanning tree would be equal to the number of edges minus 1.

  E` = |V| - 1

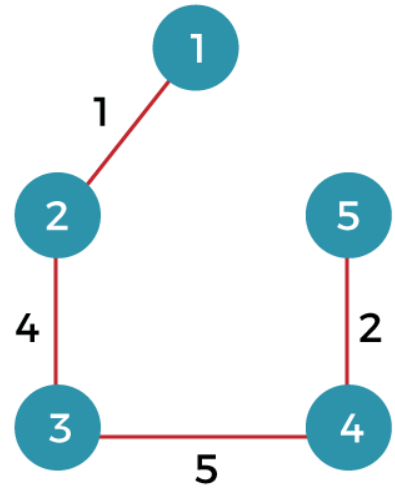
- The spanning tree should not contain any cycle.

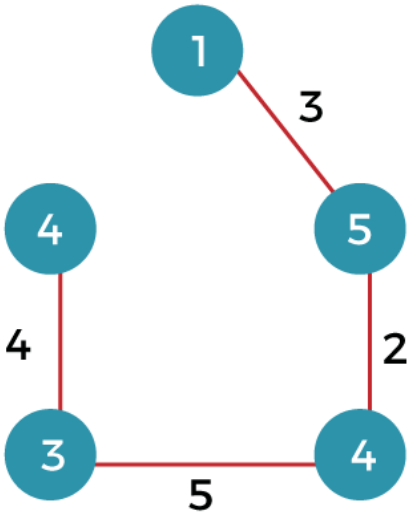
- The spanning tree should not be disconnected.

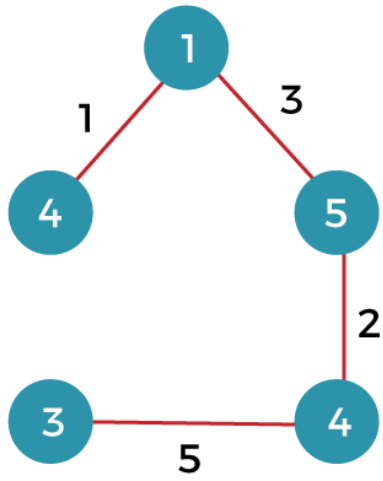
Note: A graph can have more than one spanning tree.

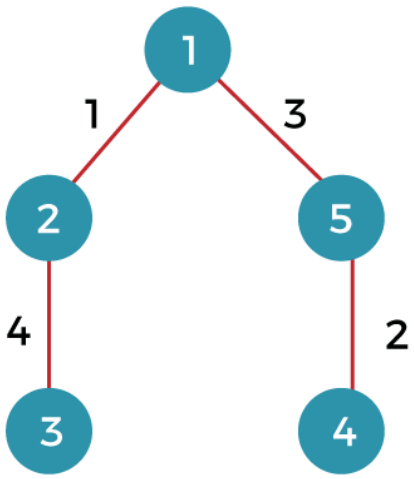
Consider the graph below:

The above graph contains 5 vertices. As we know, the vertices in the spanning tree would be the same as the graph; therefore, V` is equal 5. The number of edges in the spanning tree would be equal to (5 - 1), i.e., 4. The following are the possible spanning trees:









3.3.1 What is a minimum spanning tree?

The minimum spanning tree is a spanning tree whose sum of the edges is minimum. Consider the below graph that contains the edge weight:

The following are the spanning trees that we can make from the above graph.

i. The first spanning tree is a tree in which we have removed the edge between the vertices 1 and 5 shown as below:
The sum of the edges of the above tree is (1 + 4 + 5 + 2): 12

ii. The second spanning tree is a tree in which we have removed the edge between the vertices 1 and 2 shown as below:
The sum of the edges of the above tree is (3 + 2 + 5 + 4) : 14

iii. The third spanning tree is a tree in which we have removed the edge between the vertices 2 and 3 shown as below:
The sum of the edges of the above tree is (1 + 3 + 2 + 5) : 11

iv. The fourth spanning tree is a tree in which we have removed the edge between the vertices 3 and 4 shown as below:
The sum of the edges of the above tree is (1 + 3 + 2 + 4) : 10. The edge cost 10 is minimum so it is a minimum spanning tree.

3.3.2 General properties of minimum spanning tree:

i. If we remove any edge from the spanning tree, then it becomes disconnected. Therefore, we cannot remove any edge from the spanning tree.

ii. If we add an edge to the spanning tree then it creates a loop. Therefore, we cannot add any edge to the spanning tree.

iii. In a graph, each edge has a distinct weight, then there exists only a single and unique minimum spanning tree. If the edge weight is not distinct, then there can be more than one minimum spanning tree.

iv. A complete undirected graph can have an n^{n-2} number of spanning trees.

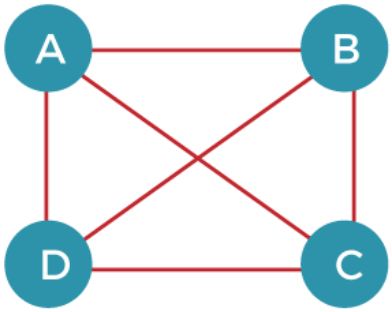
v. Every connected and undirected graph contains atleast one spanning tree.

vi. The disconnected graph does not have any spanning tree.

vii. In a complete graph, we can remove maximum (e-n+1) edges to construct a spanning tree.

Let us understand the last property through an example.

Consider the complete graph which is given below:



The number of spanning trees that can be made from the above complete graph equals to $n^{n-2} = 4^{4-2} = 16$.

Therefore, 16 spanning trees can be created from the above graph.

The maximum number of edges that can be removed to construct a spanning tree equals to e-n+1 = 6 - 4 + 1 = 3.

3.3.3 Application of Minimum Spanning Tree

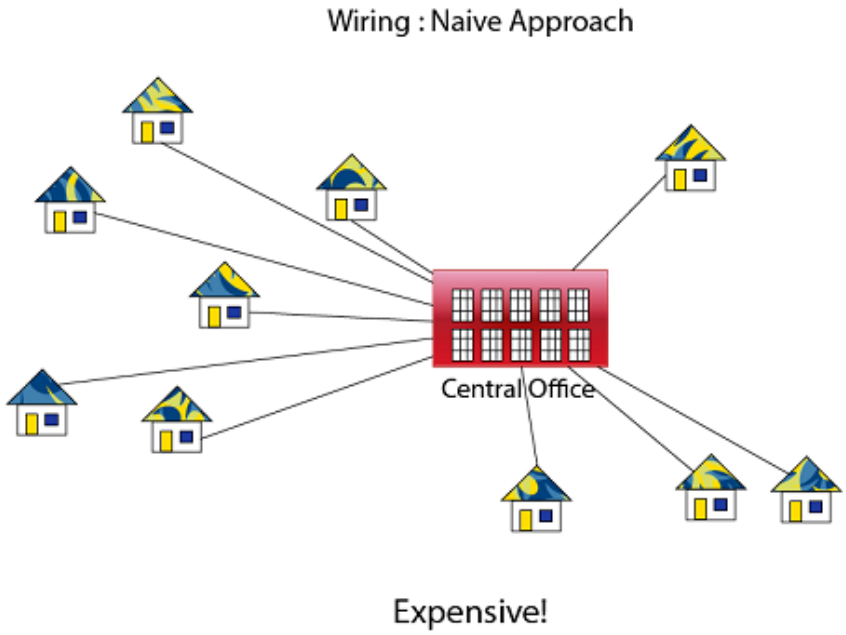
1. Consider n stations are to be linked using a communication network and laying of communication links between any two stations involves a cost. The ideal solution would be to extract a subgraph termed as minimum cost spanning tree.
2. Suppose you want to construct highways or railroads spanning several cities then we can use the concept of minimum spanning trees.
3. Designing Local Area Networks.
4. Laying pipelines connecting offshore drilling sites, refineries and consumer markets.
5. Suppose you want to apply a set of houses with

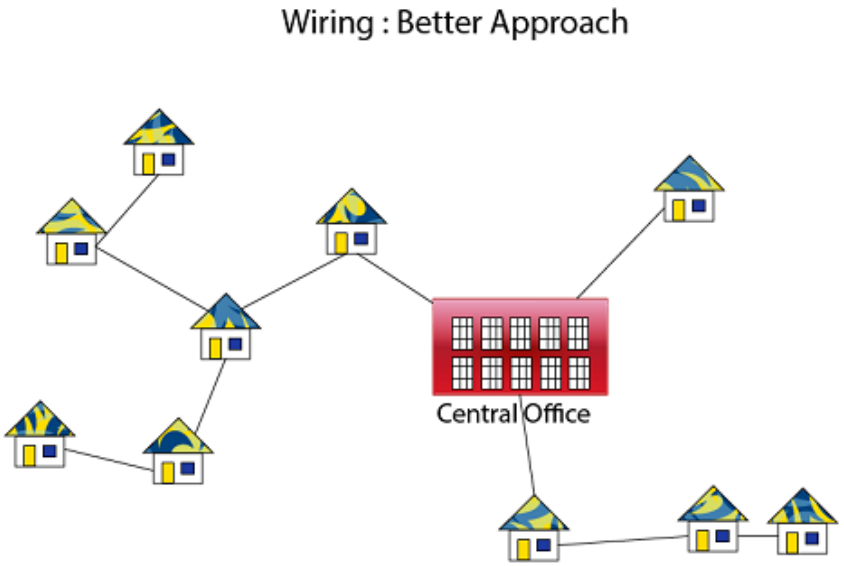
    - Electric Power
    - Water
    - Telephone lines

- Sewage lines

To reduce cost, you can connect houses with minimum cost spanning trees.

For Example, Problem laying Telephone Wire.





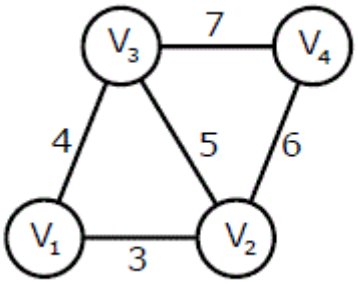
Minimize the total length of wire connecting the customers

Self-Assessment Exercises

1. The traveling salesman problem involves visiting each city how many times?

2. What do you understand by the term MINIMUM SPANNING TREE?

3. An undirected graph G(V, E) contains n (n > 2) nodes named v1 , v2 ,….vn. Two nodes vi , vj are connected if and only if 0 < |i – j| <= 2. Each edge (vi, vj) is assigned a weight i + j. A sample graph with n = 4 is shown below. What will be the cost of the minimum spanning tree (MST) of such a graph with n nodes?



4. What does and approximation ratio measures?

4.0 Conclusion

An approximation or approximate algorithm is a way of dealing with NP-completeness for an optimization problem. The goal of the approximation algorithm is to come close as much as possible to the optimal solution in polynomial time. Examples of Approximation algorithms are the Minimal Spanning tree, Vertex cover and Traveling Salesman problem.

5.0 Summary

In this Unit we considered the meaning of approximate or approximation algorithms and areas of applications like vertex cover, minimum spanning tree and traveling salesman problem amongst others. We also looked at the issue of performance ratios.

6.0 Tutor Marked Assignments

1. How does the practical Traveling Salesman problem differ from the Classical Traveling salesman problem?

2. Consider a complete undirected graph with vertex set {0, 1, 2, 3, 4}. Entry Wij in the matrix W below is the weight of the edge {i, j}. What is the minimum possible weight of a spanning tree T in this graph such that vertex 0 is a leaf node in the tree T?

$$W = \begin{pmatrix} 0 & 1 & 8 & 1 & 4 \\ 1 & 0 & 12 & 4 & 9 \\ 8 & 12 & 0 & 7 & 3 \\ 1 & 4 & 7 & 0 & 2 \\ 4 & 9 & 3 & 2 & 0 \end{pmatrix}$$

4. In the graph given in question (2) above, what is the minimum possible weight of a path P from vertex 1 to vertex 2 in this graph such that P contains at most 3 edges?

5. Consider a weighted complete graph G on the vertex set {v1,v2 ,v} such that the weight of the edge (v,,v) is 2|i-j|. The weight of a minimum spanning tree of G is?

7.0 Further Reading and Other Resources

Dave, P. H. (2007). Design and Analysis of Algorithms. Pearson Education, India. ISBN: 8177585959, 9788177585957

Greenbaum, A. and Chartier, T. P. (2012). Numerical Methods: Design, Analysis, and Computer Implementation of Algorithms. Princeton University Press. ISBN: 1400842670, 9781400842674.

Heineman, G. T., Pollice, G. and Selkow, S. (2016). Algorithms in a Nutshell. O'Reilly Media, Inc. USA.

Karamagi, R. (2020). Design and Analysis of Algorithms. Lulu.com, ISBN: 1716498155, 9781716498152

Module 3: Other Algorithm Techniques

Unit 5: Approximate Algorithms II

1.0 Introduction

An approximation algorithm is a way of dealing with NP-completeness for an optimization problem. The goal of the approximation algorithm is to come close as much as possible to the optimal solution in polynomial time.

We continue our class on Approximate algorithms by looking at some methods of the Minimal Spanning Tree given as Kruskal's algorithm and the Prim's algorithm.

2.0 Objectives

At the end of this Unit, you should be able to:

- Understand the methods of the Minimal Spanning Tree (MST)
- Know more about the Kruskal and the Prim algorithms
-

3.0 Methods of Minimum Spanning Tree

There are two methods to find Minimum Spanning Tree

1. Kruskal's Algorithm
2. Prim's Algorithm

3.1 Kruskal's Algorithm:

An algorithm to construct a Minimum Spanning Tree for a connected weighted graph. It is a Greedy Algorithm. The Greedy Choice is to put the smallest weight edge that does not because a cycle in the MST constructed so far.

If the graph is not linked, then it finds a Minimum Spanning Tree.

3.1.1 Steps for finding MST using Kruskal's Algorithm:

1. Arrange the edge of G in order of increasing weight.
2. Starting only with the vertices of G and proceeding sequentially add each edge which does not result in a cycle, until (n - 1) edges are used.
3. EXIT.

MST- KRUSKAL (G, w)

```
A ← ∅
for each vertex v ∈ V [G]
do MAKE - SET (v)
sort the edges of E into non decreasing order by weight w
for each edge (u, v) ∈ E, taken in non decreasing order by weight
do if FIND-SET (μ) ≠ if FIND-SET (v)
then A  ←  A ∪ {(u, v)}
UNION (u, v)
return A
```

Analysis:

Where E is the number of edges in the graph and V is the number of vertices, Kruskal's Algorithm can be shown to run in O (E log E) time, or simply, O (E log V) time, all with simple data structures. These running times are equivalent because:

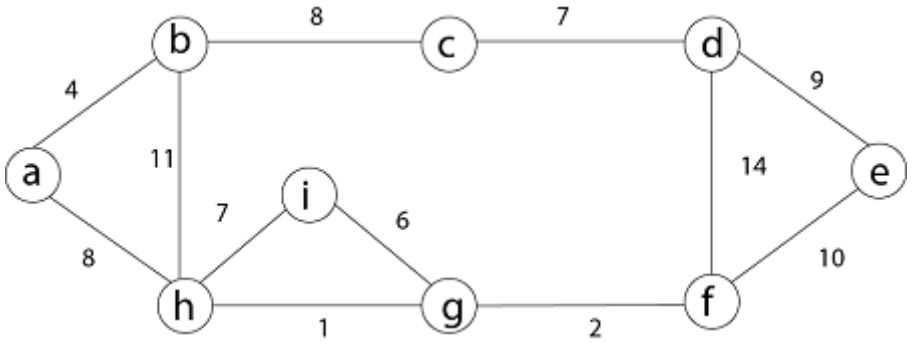
- E is at most V^2 and log V^2= 2 x log V is O (log V).
- If we ignore isolated vertices, which will each their components of the minimum spanning tree, V ≤ 2 E, so log V is O (log E).

Thus the total time is

$$O (E \log E) = O (E \log V).$$

Example:

Find the Minimum Spanning Tree of the following graph using Kruskal's algorithm.



Solution:

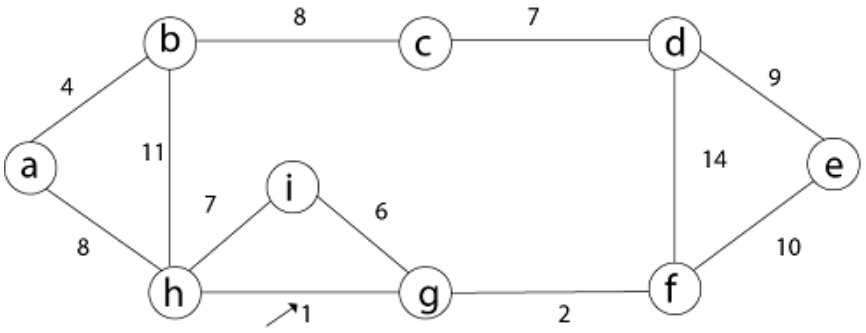
First we initialize the set A to the empty set and create |v| trees, one containing each vertex with MAKE-SET procedure. Then sort the edges in E into order by non-decreasing weight.

There are 9 vertices and 12 edges. So MST formed (9-1) = 8 edges

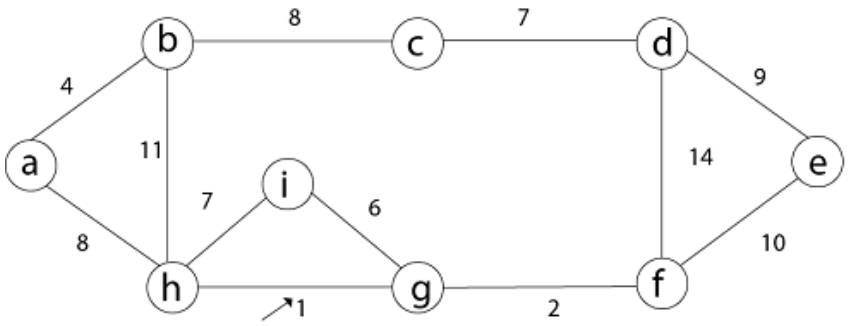
Weight	Source	Destination
1	h	g
2	g	f
4	a	b
6	i	g
7	h	i
7	c	d
8	b	c
8	a	h
9	d	e
10	e	f
11	b	h
14	d	f

Now, check for each edge (u, v) whether the endpoints u and v belong to the same tree. If they do then the edge (u, v) cannot be supplementary. Otherwise, the two vertices belong to different trees, and the edge (u, v) is added to A, and the vertices in two trees are merged in by union procedure.

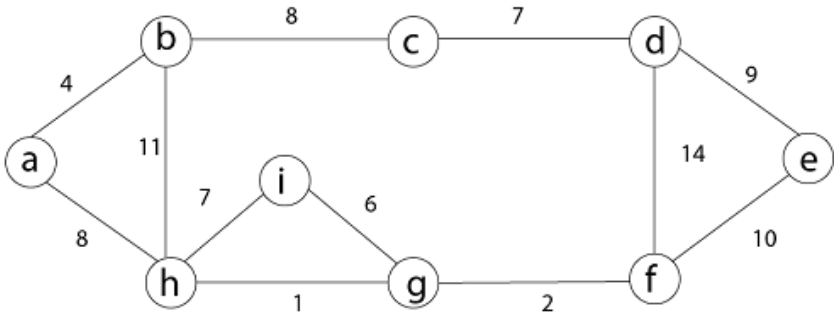
Step1: So, first take (h, g) edge



Step 2: then (g, f) edge.

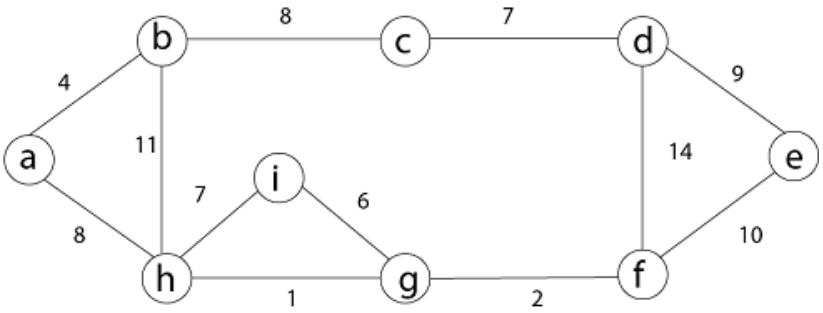


Step 3: then (a, b) and (i, g) edges are considered, and the forest becomes



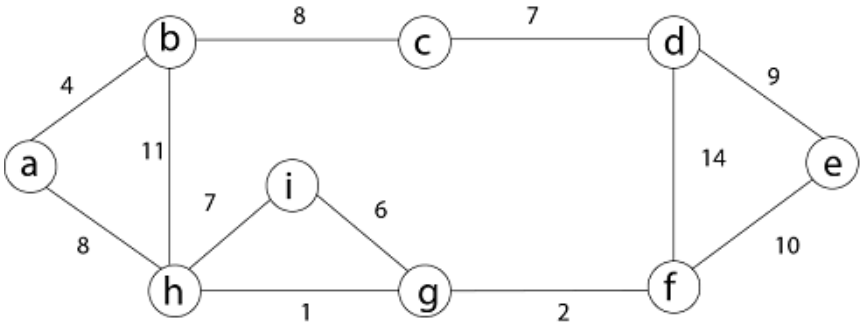
Step 4: Now, edge (h, i). Both h and i vertices are in the same set. Thus it creates a cycle. So this edge is discarded.

Then edge (c, d), (b, c), (a, h), (d, e), (e, f) are considered, and the forest becomes.



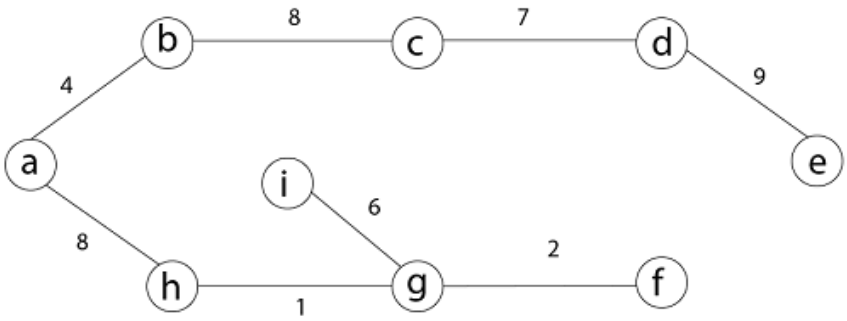
Step 5: In (e, f) edge both endpoints e and f exist in the same tree so discarded this edge. Then (b, h) edge, it also creates a cycle.

Step 6: After that edge (d, f) and the final spanning tree is shown as in dark lines.



Step 7: This step will be required Minimum Spanning Tree because it contains all the 9 vertices and (9 - 1) = 8 edges

e → f, b → h, d → f [cycle will be formed]



Minimum Cost MST

3.2 Prim's Algorithm

It is a greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices:

- Contain vertices already included in MST.
- Contain vertices not yet included.

At every step, it considers all the edges and picks the minimum weight edge. After picking the edge, it moves the other endpoint of edge to set containing MST.

3.2.1 Steps for finding MST using Prim's Algorithm:

1. Create MST set that keeps track of vertices already included in MST.
2. Assign key values to all vertices in the input graph. Initialize all key values as INFINITE (∞). Assign key values like 0 for the first vertex so that it is picked first.
3. While MST set doesn't include all vertices.
    a. Pick vertex u which is not is MST set and has minimum key value. Include 'u'to MST set.
    b. Update the key value of all adjacent vertices of u. To update, iterate through all adjacent vertices. For every adjacent vertex v, if the weight of edge u.v less than the previous key value of v, update key value as a weight of u.v.

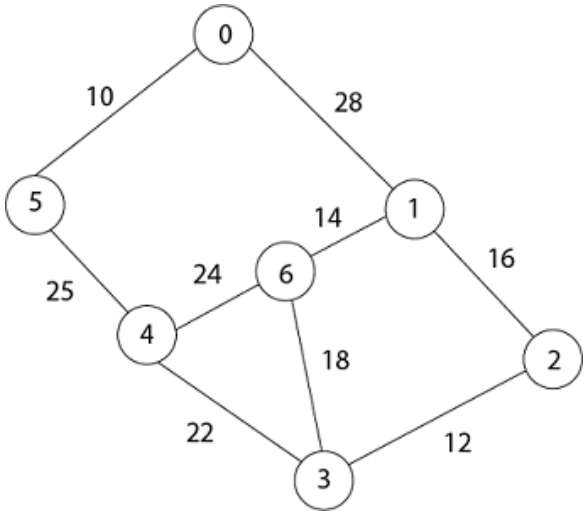
MST-PRIM (G, w, r)

```
for each u ∈ V [G]
do key [u] ← ∞
π [u] ← NIL
key [r] ← 0
Q ← V [G]
While Q ? ∅
do u ← EXTRACT - MIN (Q)
for each v ∈ Adj [u]
```

```
do if v ∈ Q and w (u, v) < key [v]
then π [v] ← u
key [v] ← w (u, v)
```

Example:

Generate minimum cost spanning tree for the following graph using Prim's algorithm.



Solution:

In Prim's algorithm, first we initialize the priority Queue Q. to contain all the vertices and the key of each vertex to ∞ except for the root, whose key is set to 0. Suppose 0 vertex is the root, i.e., r. By EXTRACT - MIN (Q) procure, now u = r and Adj [u] = {5, 1}.

Removing u from set Q and adds it to set V - Q of vertices in the tree. Now, update the key and π fields of every vertex v adjacent to u but not in a tree.

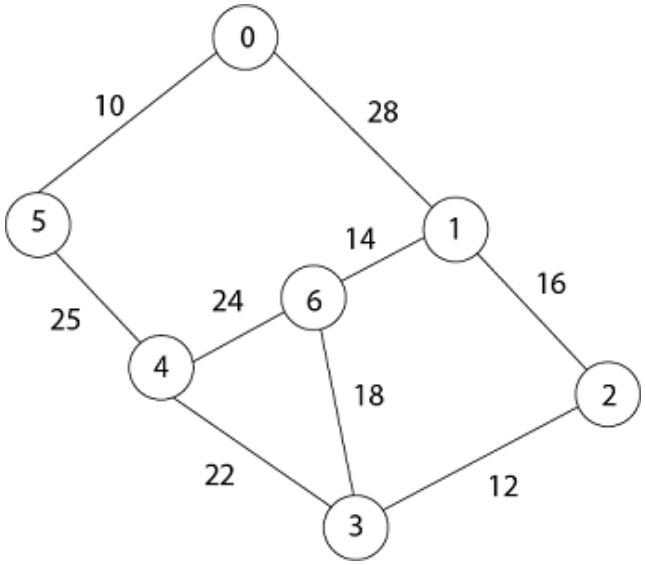
Vertex	0	1	2	3	4	5	6
Key Value	0	∞	∞	∞	∞	∞	∞
Parent	NIL	NIL	NIL	NIL	NIL	NIL	NIL

```
Taking 0 as starting vertex
  Root = 0
```

```
  Adj [0] = 5, 1
Parent, π [5] = 0 and π [1] = 0
    Key [5] = ∞ and key [1] = ∞
w [0, 5) = 10  and w (0,1) = 28
 w (u, v) < key [5] , w (u, v) < key [1]
      Key [5] = 10 and key [1] = 28
```

So update key value of 5 and 1 is:

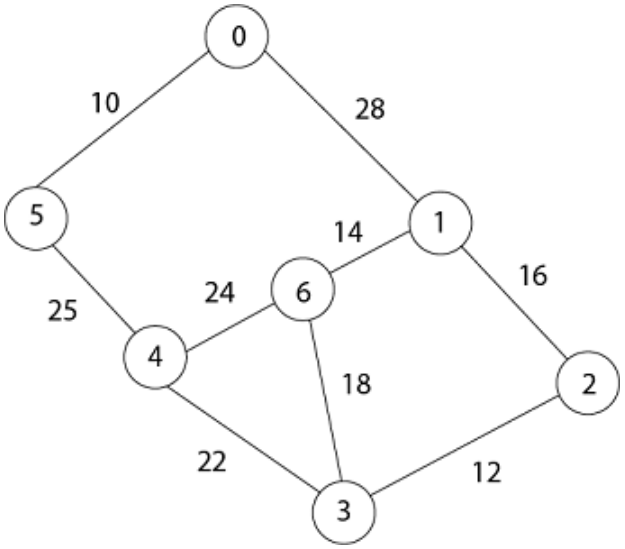
Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	∞	∞	10	∞
Parent	NIL	0	NIL	NIL	NIL	0	NIL



Now by EXTRACT_MIN (Q) Removes 5 because key [5] = 10 which is minimum so u = 5.

```
Adj [5] = {0, 4} and 0 is already in heap
Taking 4, key [4] = ∞      π [4] = 5
(u, v) < key [v] then key [4] = 25
w (5,4) = 25
w (5,4) < key [4]
date key value and parent of 4.
```

Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	∞	25	10	∞
Parent	NIL	0	NIL	NIL	5	0	NIL



Now remove 4 because key [4] = 25 which is minimum, so u =4

```
Adj [4] = {6, 3}
Key [3] = ∞          key [6] = ∞
w (4,3) = 22         w (4,6) = 24
w (u, v) < key [v]     w (u, v) < key [v]
w (4,3) < key [3]       w (4,6) < key [6]
```

Update key value of key [3] as 22 and key [6] as 24.

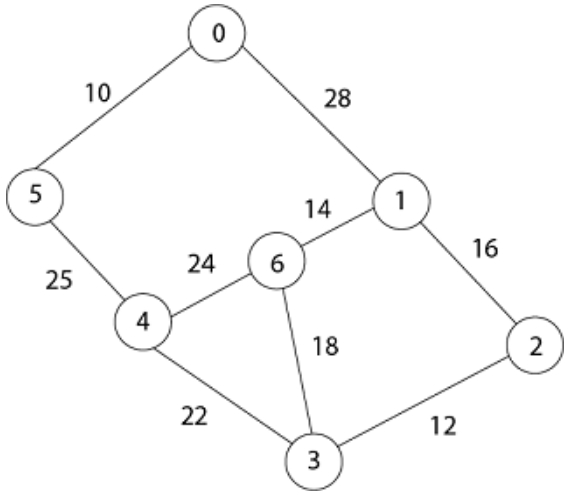
And the parent of 3, 6 as 4.

$\pi[3] = 4$ $\pi[6] = 4$

Vertex	0	1	2	3	4	5	6
Key Value	0	28	∞	22	25	10	24
Parent	NIL	0	NIL	4	5	0	4

u = EXTRACT_MIN (3, 6) [key [3] < key [6]]
u = 3 i.e. 22 < 24

Now remove 3 because key [3] = 22 is minimum so u =3.




```
Adj [3] = {4, 6, 2}
  4 is already in heap
  4 ≠ Q key [6] = 24 now becomes key [6] = 18
  Key [2] = ∞              key [6] = 24
  w (3, 2) = 12            w (3, 6) = 18
  w (3, 2) < key [2]           w (3, 6) < key [6]
```

Now in Q, key [2] = 12, key [6] = 18, key [1] = 28 and parent of 2 and 6 is 3.

π [2] = 3 π[6]=3

Now by EXTRACT_MIN (Q) Removes 2, because key [2] = 12 is minimum.

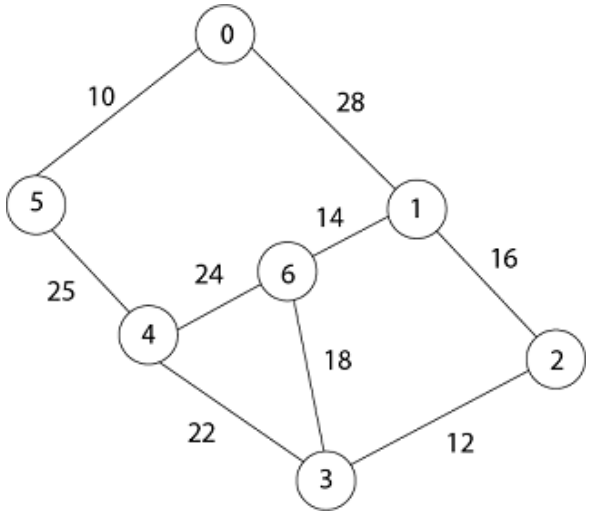
Vertex	0	1	2	3	4	5	6
Key Value	0	28	12	22	25	10	18
Parent	NIL	0	3	4	5	0	3

```
u = EXTRACT_MIN (2, 6)
u = 2           [key [2] < key [6]]
        12 < 18
Now the root is 2
Adj [2] = {3, 1}
   3 is already in a heap
Taking 1, key [1] = 28
   w (2,1) = 16
   w (2,1) < key [1]
```

So update key value of key [1] as 16 and its parent as 2.

π[1]= 2

Vertex	0	1	2	3	4	5	6
Key Value	0	16	12	22	25	10	18
Parent	NIL	2	3	4	5	0	3



Now by EXTRACT_MIN (Q) Removes 1 because key [1] = 16 is minimum.

```
Adj [1] = {0, 6, 2}
    0 and 2 are already in heap.
Taking 6, key [6] = 18
   w [1, 6] = 14
   w [1, 6] < key [6]
```

Update key value of 6 as 14 and its parent as 1.

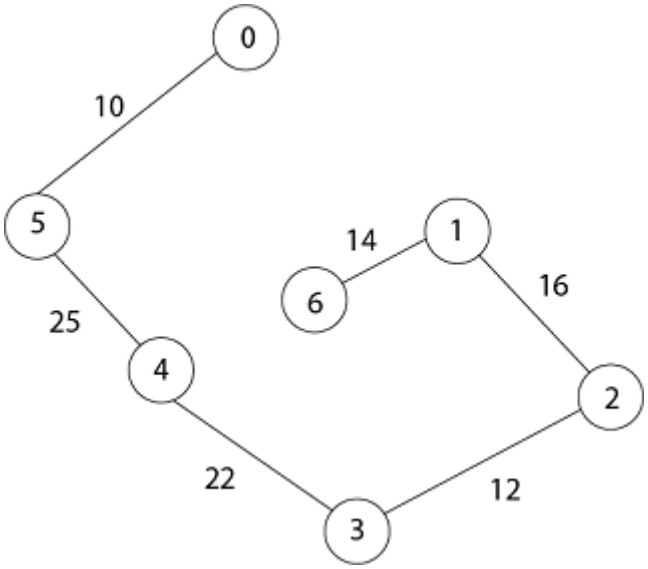
$\Pi [6] = 1$

Vertex	0	1	2	3	4	5	6
Key Value	0	16	12	22	25	10	14
Parent	NIL	2	3	4	5	0	1

Now all the vertices have been spanned, Using above the table we get Minimum Spanning Tree.

$0 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \rightarrow 6$
[Because $\Pi [5] = 0$, $\Pi [4] = 5$, $\Pi [3] = 4$, $\Pi [2] = 3$, $\Pi [1] = 2$, $\Pi [6] = 1$]

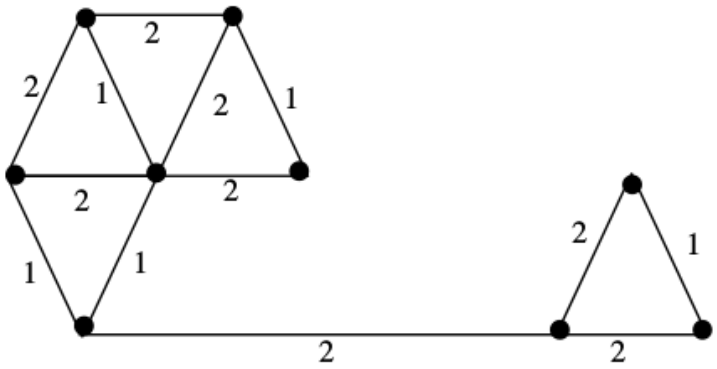
Thus the final spanning Tree is



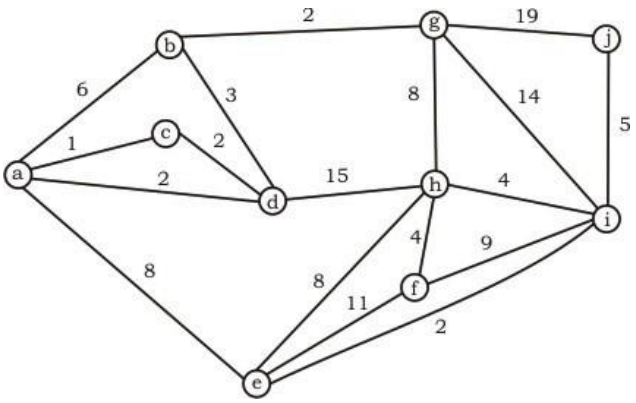
Total Cost = 10 + 25 + 22 + 12 + 16 + 14 = 99

Self-Assessment Exercises

1. The number of distinct minimum spanning trees for the weighted graph below is?



2. What is the weight of a minimum spanning tree of the following graph ?



3. Let G be connected undirected graph of 100 vertices and 300 edges. The weight of a minimum spanning tree of G is 500. When the weight of each edge of G is increased by five, the weight of a minimum spanning tree becomes?

4.0 Conclusion

An approximation algorithm returns a solution to a combinatorial optimization problem that is provably close to optimal (as opposed to a *heuristic* that may or may not find a good solution). Approximation algorithms are typically used when finding an optimal solution is intractable, but can also be used in some situations where a near-optimal solution can be found quickly and an exact solution is not needed.

Many problems that are NP-hard are also non-approximable assuming P≠NP.

5.0 Summary

In this Unit, we concluded our class on Approximate or Approximation algorithms by looking again at the Minimal Spanning Tree and methods for resolving MST problems, we looked at the Prim's and Kruskal's algorithms as well as steps for finding MST using either of the algorithms sonsidered.

6.0 Tutor Marked Assignments

1. Let G be a weighted connected undirected graph with distinct positive edge weights. If every edge weight is increased by the same value, then which of the following statements is/are TRUE?

- P: Minimum spanning tree of G does not change
- Q: Shortest path between any pair of vertices does not change

2. G = (V, E) is an undirected simple graph in which each edge has a distinct weight, and e is a particular edge of G. Which of the following statements about the minimum spanning trees (MSTs) of G is/are TRUE

   I. If e is the lightest edge of some cycle in G,

      then every MST of G includes e

   II. If e is the heaviest edge of some cycle in G,

      then every MST of G excludes e

3. What is the largest integer m such that every simple connected graph with n vertices and n edges contains at least m different spanning trees?

7.0 Further Reading and Other Resources

Dave, P. H. (2007). Design and Analysis of Algorithms. Pearson Education, India. ISBN: 8177585959, 9788177585957

Greenbaum, A. and Chartier, T. P. (2012). Numerical Methods: Design, Analysis, and Computer Implementation of Algorithms. Princeton University Press. ISBN: 1400842670, 9781400842674.

Heineman, G. T., Pollice, G. and Selkow, S. (2016). Algorithms in a Nutshell. O'Reilly Media, Inc. USA.

Karamagi, R. (2020). Design and Analysis of Algorithms. Lulu.com, ISBN: 1716498155, 9781716498152