Debre Tabor University

Gafat institute of technology

Department of Electrical and computer engineering

Design and Analysis of Algorithm

Prepared by: **Misganaw A. (MSC.)**

Email: mail@ethioptec.com **or ethiomisgie@gmail.com**

Website: ethioptec.com **or** info.ethioptec.com

Online modules and references: academics.ethioptec.com

Table of Contents

Chapter 1

1. Introduction to Algorithms

An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks.

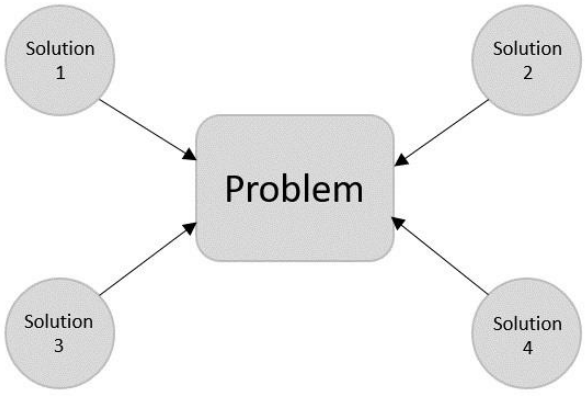
An algorithm is the best way to represent the solution of a particular problem in a very **simple and efficient way**.

If we have an algorithm for a **specific problem**, then we can implement it in any **programming language**, meaning that the **algorithm is independent from any programming languages**.

1.1 Algorithm Design

To solve a problem, different approaches can be followed.

Some of them can be efficient with respect to **time consumption**, whereas other approaches may be **memory efficient**. However, one has to keep in mind that both time **consumption** and **memory** usage cannot be optimized **simultaneously**. If we require an algorithm to run in **lesser time**, we have to **invest** in more memory and if we require an algorithm to run with **lesser memory**, we need to have **more time**.



1.1.1 Problem Development Steps

The following steps are involved in solving computational problems.

- Problem definition
- Development of a model
- Specification of an Algorithm
- Designing an Algorithm
- Checking the correctness of an Algorithm
- Analysis of an Algorithm
- Implementation of an Algorithm
- Program testing
- Documentation

1.1.2 How to Write an Algorithm?

There are no **well-defined standards** for writing algorithms. Rather, it is **problem** and **resource** dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (**do, for, while), flow-control (if-else)**, etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is **well-defined**. That is, we should know the problem domain, for which we are designing a solution.

Example

Let's try to learn algorithm-writing by using an example.

Problem − Design an algorithm to add two numbers and display the result.

Step 1 − START
Step 2 − declare three integers a, b & c
Step 3 − define values of a & b
Step 4 − add values of a & b
Step 5 − store output of step 4 to c
Step 6 − print c
Step 7 − STOP

Algorithms tell the programmers how to **code the program**. Alternatively, the algorithm can be written as −

Step 1 − START ADD
Step 2 − get values of a & b
Step 3 − c ← a + b
Step 4 − display c
Step 5 − STOP

In design and analysis of algorithms, usually the **second method** is used to describe an **algorithm**. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.

1.2 Characteristics of Algorithms

Not all procedures can be called an **algorithm**. An **algorithm** should have the following characteristics

- **Unambiguous** − Algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
- **Input** − An algorithm should have 0 or more well-defined inputs.
- **Output** − An algorithm should have 1 or more well-defined outputs, and should match the desired output.
- **Finiteness** − Algorithms must terminate after a finite number of steps.
- **Feasibility** − Should be feasible with the available resources.
- **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

1.3 Performance of a program

The **performance** of a program is the amount of **computer memory** and **time** needed to run a program. We use **two** approaches to determine the performance of a program.

1. One is analytical, and
2. The other experimental.

In performance **analysis** we use **analytical** methods, while in performance **measurement** we conduct experiments.

1.4 Pseudocode

Pseudocode gives a high-level description of an algorithm without the ambiguity associated with **plain text** but also without the need to know the **syntax** of a particular programming language.

1.5 Difference between Algorithm and Pseudocode

Algorithm	Pseudocode
Generally, the word "**algorithm**" can be used to describe any high-level task in computer science.	On the other hand, **pseudocode** is an informal and (often rudimentary) human readable description of an algorithm leaving many granular details of it. Writing a **pseudocode** has no restriction of styles and its only objective is to describe the high-level steps of algorithm in a much realistic manner in natural language.

For example, following is an algorithm for Insertion Sort.

Algorithm: Insertion-Sort
Input: A list L of integers of length n
Output: A sorted list L1 containing those integers present in L
Step 1: Keep a sorted list L1 which starts off empty
Step 2: Perform Step 3 for each element in the original list L
Step 3: Insert it into the correct position in the sorted list L1.
Step 4: Return the sorted list
Step 5: Stop

Here is a pseudocode which describes how the high-level abstract process mentioned above in the algorithm Insertion-Sort could be described in a more realistic way.

```
for i <- 1 to length(A)
  x <- A[i]
  j <- i
  while j > 0 and A[j-1] > x
    A[j] <- A[j-1]
    j <- j - 1
  A[j] <- x
```

Example

Following are the implementations of the above approach in python programming languages –

Python program

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1

        # Move elements of arr[0..i-1] that are greater than key,
        # to one position ahead of their current position.
        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key  # Insert the current element (key) in the
correct position.

arr = [64, 34, 25, 12, 22, 11, 90]
insertion_sort(arr)
print("Sorted array:", arr)
```

Output

Sorted array: [11, 12, 22, 25, 34, 64, 90]

1.5.1.1.1 Exercise

Write C ++ and Java program for the above python code?

Chapter 2

2 Analysis of Algorithms

---

*The field of computer science, which studies efficiency of algorithms, is known as **analysis of algorithms**.*

Algorithm analysis is an important part of computational **complexity theory**, which provides theoretical estimation for the required resources of an **algorithm** to solve a specific computational problem. **Analysis of algorithms** is the determination of the **amount of time and space resources** required to execute it.

Usually, the efficiency or running time of an algorithm is stated as a **function** relating the **input length** to the **number of steps**, known as **time complexity**, or volume of memory, known as **space complexity**.

2.1 The Need for Analysis

Algorithms are often quite different from one another, though the **objective** of these algorithms are the **same**.

Analysis of algorithm is the process of analyzing the problem-solving capability of the algorithm in terms of the **time** and **size** required (the **size of memory** for storage while implementation). However, the main concern of analysis of algorithms is the **required time** or **performance**.

Generally, we perform the following types of analysis.

- **Worst-case** − The maximum number of steps taken on any instance of size **a**.
- **Best-case** − The minimum number of steps taken on any instance of size **a**.
- **Average case** − An average number of steps taken on any instance of size **a**.
- **Amortized** − A sequence of operations applied to the input of size **a** averaged over time.

To solve a problem, we need to consider **time** as well as **space** complexity as the program may run on a system where memory is limited but adequate space is available

or may be vice-versa. In this context, if we compare **bubble sort** and **merge sort**. Bubble sort does not require additional memory, but merge sort requires additional space. Though time complexity of bubble sort is higher compared to merge sort, we may need to apply bubble sort if the program needs to run in an environment, where memory is very limited.

2.2 Algorithm Design Goals

The three basic design goals that one should strive for in a program are:

1. Try to save Time
2. Try to save Space
3. Try to save Face

A program that runs **faster** is a better program, so saving **time** is an obvious goal. Like wise, a program that saves **space** over a competing program is considered desirable. We want to "**save face**" by preventing the program from locking up or generating **reams** of garbled data.

2.3 Classification of Algorithms

If **"n"** is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc.

2.3.1 1

Next instructions of most programs are executed **once** or at most only a **few times**. If all the instructions of a program have this property, we say that its running time is a **constant**.

2.3.2 Log n

When the running time of a program is **logarithmic**, the program gets slightly **slower** as **n** grows. This running **time** commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction., When **n** is a million, **log n** is a doubled. Whenever **n** doubles, **log n** increases by a constant, but **log n** does not double until n increases to $\mathbf{n^2}$.

2.3.3 n

When the running time of a program is linear, it is generally the case that a **small amount** of processing is done on each input element. This is the optimal situation for an algorithm that must process **n** inputs.

2.3.4 n.log n

This running time arises for algorithms that solve a problem by breaking it up into smaller **sub-problems**, solving then **independently**, and then **combining** the solutions. When **n** doubles, the running time more than doubles.

2.3.5 n^2

When the running time of an algorithm is **quadratic**, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a **double nested loop**) whenever **n** doubles, the running time increases fourfold.

2.3.6 n^3

Similarly, an algorithm that process **triples** of data items (perhaps in **a triple–nested loop**) has a **cubic running time** and is practical for use only on small problems. Whenever **n** doubles, the running time increases eight-fold.

2.3.7 2^n

Few algorithms with exponential running time are likely to be appropriate for practical use, such algorithms arise naturally as “brute–force” solutions to problems. Whenever **n** doubles, the running time squares.

2.4 Complexity of Algorithms

Mainly, algorithmic **complexity** is concerned about its performance, **how fast** or **slow** it works.

The complexity of an algorithm describes the **efficiency** of the algorithm in terms of the amount of the **memory** required to process the data and the processing **time**.

Complexity of an algorithm is analyzed in two perspectives: **Time** and **Space**.

The complexity of an algorithm **M** is the function **f(n)** which gives the **running time** and/or **storage space** requirement of the algorithm in terms of the size **"n"** of the input data. Mostly, the **storage space** required by an algorithm is simply a **multiple of** the **data size "n"**. Complexity shall refer to the **running time** of the algorithm. The function **f(n)**, gives the **running time** of an algorithm, depends not only on the size **"n"** of the input data but also on the particular data. The complexity function **f(n)** for certain cases are:

1. Best Case: The minimum possible value of **f(n)** is called the best case.
2. Average Case: The expected value of **f(n)**.
3. Worst Case: The maximum value of **f(n)** for any key possible input.

2.4.1 Time Complexity

It's a function describing the **amount of time required** to run an algorithm in terms of the size of the input. "Time" can mean

1. the number of memory accesses performed
2. the number of comparisons between integers
3. the number of times some inner loop is executed or
4. some other natural unit related to the amount of real time the algorithm will take.

2.4.2 Space Complexity

Space complexity is a function describing the **amount of memory** (space) an algorithm takes in terms of the amount of input to the algorithm.

Space complexity is sometimes ignored because the **space used is minimal** and/or obvious, however sometimes it becomes as important issue as time complexity

The space need by a program has the following components:

Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.

Data space: Data space is the space needed to store all constant and variable values. Data space has two components:

- Space needed by constants and simple variables in program.

- Space needed by dynamically allocated objects such as arrays and class instances.

Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.

Instruction Space: The amount of instructions space that is needed depends on factors such as:

- The compiler used to complete the program into machine code.
- The compiler options in effect at the time of compilation
- The target computer.

2.5 Rate of Growth

Rate of growth is defined as the rate at which the **running time** of the algorithm is **increased** when the **input size** is increased.

The growth rate could be categorized into two types:

1. **Linear**. If the algorithm is increased in a linear way with an increasing in input size, it is **linear growth rate**.
2. **Exponential**. if the running time of the algorithm is increased exponentially with the increase in input size, it is **exponential growth rate**.

Chapter 3

3 Methodology of Algorithm Analysis

3.1 Asymptotic Analysis

Asymptotic analysis of an algorithm refers to defining the **mathematical foundation**/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the **best case, average case, and worst case** scenario of an algorithm.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a **constant time**.

Asymptotic analysis refers to **computing the running time** of any operation in **mathematical units** of computation.

For example, the running time of one operation is computed as **f(n)** and may be for another operation it is computed as $g(n^2)$. This means the first operation running time will increase **linearly** with the increase in **n** and the running time of the **second** operation will increase **exponentially** when **n** increases. Similarly, the running time of both operations will be nearly the same if **n** is significantly small.

Usually, the time required by an algorithm falls under three types

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

The asymptotic behavior of a function ***f(n)*** refers to the growth of ***f(n)*** as **n** gets large.

We typically ignore small values of **n**, since we are usually interested in estimating how slow the program will be on large inputs.

A good rule of **thumb** is that the **slower** the asymptotic growth rate, the **better** the algorithm. Though it's not always true.

For example, a linear algorithm $f(n)=d*n+k$ is always asymptotically better than a quadratic one, $f(n)=c*n^2+q$.

3.1.1 Asymptotic Notations

Execution time of an algorithm depends on

1. the instruction set
2. processor speed
3. disk I/O speed, etc. Hence, we estimate the efficiency of an algorithm asymptotically.

Time function of an algorithm is represented by **T(n)**, where **n** is the input size.

Following asymptotic notations are used to calculate the running time complexity of an algorithm.

- **O** − Big Oh
- **Ω** − Big omega
- **θ** − Big theta
- **o** − Little Oh
- **ω** − Little omega

Time complexity	Name	Example
O(1)	Constant	Adding an element to the front of a linked list
O(logn)	Logarithmic	Finding an element in a sorted array
O (n)	Linear	Finding an element in an unsorted array
O(nlog n)	Linear	Logarithmic Sorting n items by 'divide-and-conquer'- Mergesort
$O(n^2)$	Quadratic	Shortest path between two nodes in a graph
$O(n^3)$	Cubic	Matrix Multiplication
$O(2^n)$	Exponential	The Towers of Hanoi problem

3.1.1.1 O: Asymptotic Upper Bound

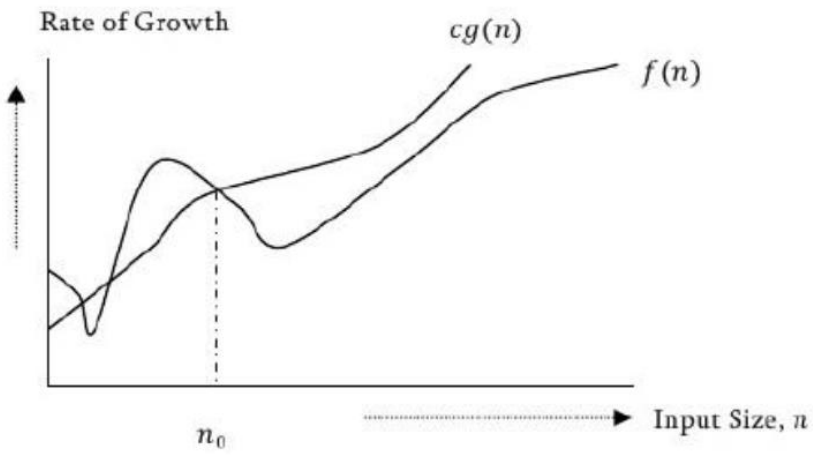
'O' (Big Oh) is the most commonly used notation. A function ***f(n)*** can be represented is the order of ***g(n)*** that is ***O(g(n))***, if there exists a value of positive integer **n** as n_0 and a positive constant **c** such that

$f(n) \leqslant c.g(n)$ **for** $n > n_0$ in all case

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

For example, if f(n) = $n^4 + 100n^2 + 10n + 50$ is the given algorithm, then n_4 is **g(n)**. That means **g(n)** gives the maximum rate of growth for **f(n)** at larger values of **n**

f(n) = O(g(n)), (pronounced order of or **big oh**), says that the growth rate of **f(n)** is less than or equal **(<)** that of **g(n).**



Example

Let us consider a given function, $f(n)=4.n^3+10.n^2+5.n+1$

Considering $g(n)=n^3$, $f(n) \leqslant 5.g(n)$ for all the values of $n>2$

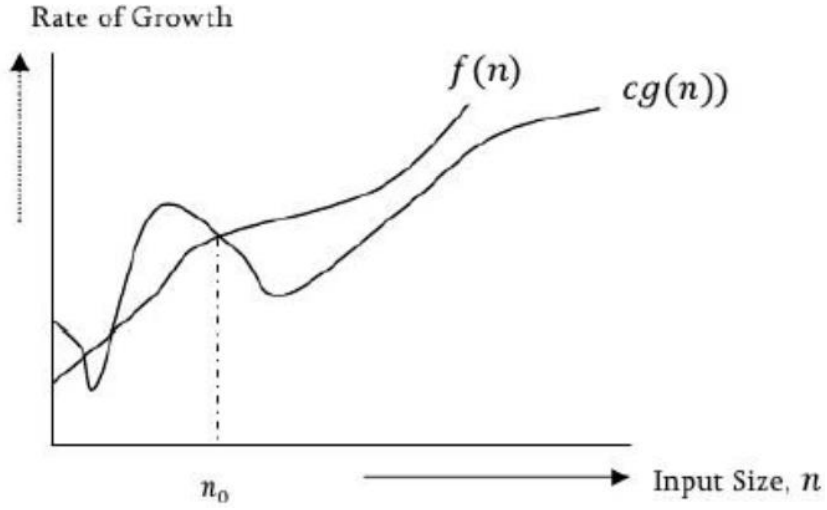
Hence, the complexity of ***f(n)*** can be represented as $O(g(n))$, **i.e.** $O(n^3)$

In general, we do not consider lower values of n. That means the rate of growth at lower values of **n** is not important. In the below figure, n_o is the point from which we consider the rate of growths for a given algorithm. Below n_o the rate of growths may be different

3.1.1.2 Ω: Asymptotic Lower Bound

Similar to above discussion, this notation gives the tighter lower bound of the given algorithm and we represent it as **f(n) = Ω (g(n)).** That means, at larger values of **n,** the tighter lower bound of **f(n) is g.**

For example, if **f(n) = 100n² + 10n + 50, g(n) is Ω (n²).** The . **Ω**. notation as be defined as **Ω (g (n)) = {f(n):** there exist positive constants **c** and **n₀** such **that 0 <= cg (n) <= f(n)** for all **n >= n₀}. g(n)** is an asymptotic lower bound for **f(n). Ω (g (n))** is the set of functions with smaller or same order of growth as **f(n).**



Example

Let us consider a given function, $f(n)=4.n^3+10.n^2+5.n+1$.

Considering $g(n)=n^3$, $f(n)\geqslant 4.g(n)$ for all the values of $n>0$.

Hence, the complexity of ***f(n)*** can be represented as $\Omega(g(n))$, **i.e.** $\Omega(n^3)$

3.1.1.3 θ: Asymptotic Tight Bound

This notation decides whether the upper and lower bounds of a given function are same or not. The average running time of algorithm is always between lower bound and upper bound.

If the upper bound **(O)** and lower bound **(Ω)** gives the same result then **Θ** notation will also have the same rate of growth.

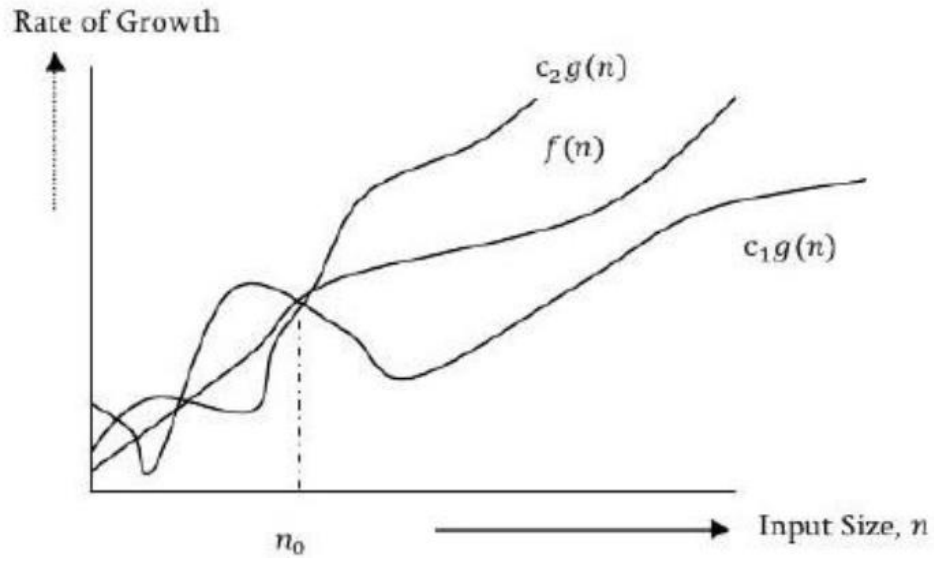
As an example, let us assume that **f(n) = 10n + n** is the expression. Then, its tight upper bound g(n) is **O(n).** The rate of growth in best case is **g (n) =0(n).** In this case, rate of growths in best case and worst are same. As a result, the average case will also be same.

Note: For a given function (algorithm), if the rate of growths (bounds) for **O** and **Ω** are not same then the rate of growth **Θ** case may not be same.

We say that $f(n)=\theta(g(n))$

when there exist constants c_1 and c_2 that $c1.g(n) \leqslant f(n) \leqslant c2.g(n)$ for all sufficiently large value of **n**. Here **n** is a positive integer.

This means function **g** is a tight bound for function **f**.



Example

Let us consider a given function, $f(n)=4.\ n^3+10.n^2+5.n+1$

Considering $g(n)=n^3$, $4.g(n) \leqslant f(n) \leqslant 5.\ g(n)$

for all the large values of **n**.

Hence, the complexity of $f(n)$ can be represented as $\theta(g(n))$, i.e. $\theta(n^3)$.

3.2 Analyzing Algorithm

Suppose **"M"** is an algorithm, and suppose **"n"** is the size of the input data. Clearly the complexity **f(n) of M** increases as **n** increases. It is usually the rate of increase of **f(n)** we

want to examine. This is usually done by comparing **f(n)** with some standard functions. The most common computing times are: **O(1), O($\log_2 n$), O(n), O(n. $\log_2 n$), O(n^2), O(n^3), O(2^n).**

3.2.1 Numerical Comparison of Different Algorithms

The execution time for six of the typical functions is given below:

n	$\log_2 n$	n*$\log_2 n$	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65,536
32	5	160	1024	32,768	4,294,967,296
64	6	384	4096	2,62,144	Note 1
128	7	896	16,384	2,097,152	Note 2
256	8	2048	65,536	1,677,216	????????

3.2.2 The rule of sums

Suppose that **T1(n) and T2(n)** are the running times of two programs fragments **P1 and P2**, and that **T1(n) is O(f(n)) and T2(n) is O(g(n)).** Then **T1(n) + T2(n),** the running time of **P1** followed by **P2** is **O (max f(n), g(n)),** this is called as rule of sums.

For example, suppose that we have three steps whose running times are respectively **O(n^2), O(n^3) and O(n.logn).** Then the running time of the first two steps executed sequentially is **O (max (n^2, n^3))** which is **O(n^3).** The running time of all three together is **O (max (n^3, n. log n))** which is **O(n^3)**

Algorithm	Time complexity	Maximum problem size		
		1 second	1 minute	1 hour
A_1	n	1000	6×10^4	3.6×10^6
A_2	n log n	140	4893	2.0×10^5
A_3	n^2	31	244	1897
A_4	n^3	10	39	153
A_5	2^n	9	15	21

The faster the computer run, the more need are efficient algorithms to take advantage of their power. As the computer becomes faster and we can handle larger problems, it is the complexity of an algorithm that determines the increase in problem size that can be

achieved with an increase in computer speed. Suppose the next generation of computers is ten times faster than the current generation, from the table we can see the increase in size of the problem.

Algorithm	Time Complexity	Maximum problem size before speed up	Maximum problem size after speed up
A_1	n	S1	10 S1
A_2	n log n	S2	≈10 S2 for large S2
A_3	n^2	S3	3.16 S3
A_4	n^3	S4	2.15 S4
A_5	2^n	S5	S5 + 3.3

3.2.3 The Running time of a program

When solving a problem, we are faced with a choice among algorithms. The basis for this can be any one of the following:

1. We would like an algorithm that is easy to understand, code and debug.
2. We would like an algorithm that makes efficient use of the computer's resources, especially, one that runs as fast as possible.

3.2.3.1 Measuring the running time of a program

The running time of a program depends on factors such as:

1. The input to the program.
2. The quality of code generated by the compiler used to create the object program.
3. The nature and speed of the instructions on the machine used to execute the program, and
4. The time complexity of the algorithm underlying the program.

The running time depends not on the **exact input** but **only the size of the input**. For many programs, the running time is really a function of the particular input, and not just of the input size. In that case we define **T(n)** to be the worst-case running time, i.e. the maximum overall input of size **"n",** of the running time on that input. We also consider **T avg(n)** the average, over all input of size **"n"** of the running time on that input. In practice, the average running time is often much harder to determine than the worst-case running time. Thus, we will use worst–case running time as the principal measure of time complexity. Seeing the remarks **(2) and (3)** we cannot express the running time **T(n)** in standard time units such as seconds. Rather we can only make remarks like the running time of such and such algorithm is proportional to n^2. The constant of proportionality will

remain un-specified, since it depends so heavily on the compiler, the machine and other factors.

3.2.4 Rules for using big-O:

The most important property is that **big-O** gives an upper bound only. If an algorithm is **$O(n^2)$,** it doesn't have to take **n^2** steps (or a constant multiple of **n^2**). But it can't take more than **n^2**. So any algorithm that is **O(n),** is also an **$O(n^2)$** algorithm. If this seems confusing, think of **big-O** as being like **"<"**. Any number that is **< n** is also **< n^2**.

1. Ignoring constant factors: **O(c f(n)) = O(f(n)),** where c is a constant; e.g. **$O(20\ n^3)$ = $O(n^3)$**
2. Ignoring smaller terms: If **a<b** then **O(a+b) = O(b),** for example **$O(n^2+n)$ = $O(n^2)$**
3. Upper bound only: If **a<b** then an **O(a)** algorithm is also an **O(b)** algorithm. For example, an **O(n)** algorithm is also an **$O(n^2)$** algorithm (but not vice versa).
4. n and log n are "bigger" than any constant, from an asymptotic view (that means for large enough **n**). So if k is a constant, an **O(n + k)** algorithm is also **O(n)**, by ignoring smaller terms. Similarly, an **O(log n + k)** algorithm is also **O(log n).**
5. Another consequence of the last item is that an **O(n log n + n)** algorithm, which is **O(n(log n + 1)),** can be simplified to **O(n logn).**

3.2.5 Calculating the running time of a program

Let us now look into how big-O bounds can be computed for some common algorithms.
Example 1:
Let's consider a short piece of source code:

```
x = 3*y + 2;
z = z + 1;
```

If y, z are scalars, this piece of code takes a *constant* amount of time, which we write as **O(1).** In terms of actual computer instructions or clock ticks, it's difficult to say exactly how long it takes. But whatever it is, it should be the same whenever this piece of code is executed. **O(1)** means *some* constant, it might be **5, or 1 or 1000.**
Example 2:

$2n^2 + 5n - 6 = O(2^n)$ $2n^2 + 5n - 6 = O(n^3)$ $2n^2 + 5n - 6 = O(n^2)$ $2n^2 + 5n - 6 \neq O(n)$	$2n^2 + 5n - 6 \neq \Theta(2^n)$ $2n^2 + 5n - 6 \neq \Theta(n^3)$ $2n^2 + 5n - 6 = \Theta(n^2)$ $2n^2 + 5n - 6 \neq \Theta(n)$
$2n^2 + 5n - 6 \neq \Omega(2^n)$ $2n^2 + 5n - 6 \neq \Omega(n^3)$ $2n^2 + 5n - 6 = \Omega(n^2)$ $2n^2 + 5n - 6 = \Omega(n)$	$2n^2 + 5n - 6 = o(2^n)$ $2n^2 + 5n - 6 = o(n^3)$ $2n^2 + 5n - 6 \neq o(n^2)$ $2n^2 + 5n - 6 \neq o(n)$

Example 3:

If the first program takes **$100n^2$** milliseconds and while the second takes $5n^3$ milliseconds, then might not $5n^3$ program better than **$100n^2$** program? As the programs can be evaluated by comparing their running time functions, with constants by proportionality neglected. So, **$5n^3$** program be better than the **$100n^2$** program.

$$5\ n^3/100\ n^2 = n/20$$

for inputs **n < 20**, the program with running time **$5n^3$** will be faster than those the one with running time **$100\ n^2$**. Therefore, if the program is to be run mainly on inputs of small size, we would indeed prefer the program whose running time was **$O(n^3)$** However, as **"n"** gets large, the ratio of the running times, which **is n/20**, gets arbitrarily larger. Thus, as the size of the input increases, the **$O(n^3)$** program will take significantly more time than the **$O(n^2)$** program. So it is always better to prefer a program whose running time with the lower growth rate. The low growth rate function's such as **O(n) or O(n log n)** are always better.

Example 4:

Analysis of simple for loop

Now let's consider a simple for loop:

```
for (i = 1; i<=n; i++)
        v[i] = v[i] + 1;
```

This loop will run exactly **n** times, and because the inside of the loop takes constant time, the total running time is proportional to n. We write it as **O(n).** The actual number of instructions might be **50n**, while the running time might be **17n** microseconds. It might even be **17n+3** microseconds because the loop needs some time to start up. The **big-O** notation allows a multiplication factor (like 17) as well as an additive factor (like 3). As

long as it's a linear function which is proportional to **n,** the correct notation is **O(n)** and the code is said to have *linear* running time.

```
for (i = 1; i<=n; i++)
        for (j = 1; j<=n; j++)
                a[i,j] = b[i,j] * x;
```

The **outer for loop executes N times**, while the **inner loop** executes **n times** for every execution of the outer loop. That is, the inner loop executes **n × n = n^2 times**. The assignment statement in the inner loop takes constant time, so the running time of the code is **O(n^2) steps**. This piece of code is said to have *quadratic* running time.

Example 6:

Analysis of matrix multiply

Let's start with an easy case. Multiplying two **n × n** matrices. The code to compute the matrix product **C = A * B** is given below.

```
for (i = 1; i<=n; i++)
        for (j = 1; j<=n; j++)
                C[i, j] = 0;
                for (k = 1; k<=n; k++)
                        C[i, j] = C[i, j] + A[i, k] * B[k, j];
```

There are 3 nested *for* loops, each of which runs n times. The innermost loop therefore executes **n*n*n = n^3** times. The innermost statement, which contains a scalar sum and product takes constant **O(1)** time. So the algorithm overall takes **O(n^3)** time.

Example 7:

Analysis of bubble sort

The main body of the code for bubble sort looks something like this:

```
for (i = n-1; i<1; i--)
        for (j = 1; j<=i; j++)
                if (a[j] > a[j+1])
                        swap a[j] and a[j+1];
```

This looks like the double. The **innermost statement**, the if, takes **O(1**) time. It doesn't necessarily take the same time when the condition is true as it does when it is **false**, but both times are bounded by a constant. But there is an important difference here. The outer **loop executes n times**, but the inner loop executes a **number of times that depends on i.** The first time the **inner for executes**, it runs **i = n-1 times**. The second

time it **runs n-2 times**, etc. The total number of times the inner **if statement executes** is therefore:

$$(n-1) + (n-2) + \ldots + 3 + 2 + 1$$

This is the sum of an arithmetic series.

$$\sum_{i=1}^{N-1} (n-i) = \frac{n(n-i)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

The value of the sum is **n(n-1)/2**. So the running time of bubble sort is **O(n(n-1)/2)**, which is **O((n^2-n)/2).** Using the rules for **big-O** given earlier, this bound simplifies to **O((n^2)/2)** by ignoring a smaller term, and to **O(n^2)**, by ignoring a constant factor. Thus, bubble sort is an **O(n^2)** algorithm.

3.2.6 General rules for the analysis of programs

In general the running time of a statement or group of statements may be parameterized by the input size and/or by one or more variables. The only permissible parameter for the running time of the whole program is „n" the input size.

1. The running time of each assignment read and write statement can usually be taken to be O(1). (There are few exemptions, such as in PL/1, where assignments can involve arbitrarily larger arrays and in any language that allows function calls in arraignment statements).
2. The running time of a sequence of statements is determined by the sum rule. I.e. the running time of the sequence is, to with in a constant factor, the largest running time of any statement in the sequence.
3. The running time of an if–statement is the cost of conditionally executed statements, plus the time for evaluating the condition. The time to evaluate the condition is normally O(1) the time for an if–then–else construct is the time to evaluate the condition plus the larger of the time needed for the statements executed when the condition is true and the time for the statements executed when the condition is false.

The time to execute a loop is the sum, over all times around the loop, the time to execute the body and the time to evaluate the condition for termination (usually the latter is O(1)). Often this time is, neglected constant factors, the product of the number of times around the loop and the largest possible time for one execution of the body, but we must consider each loop separately to make sure.

3.3 Solving Recurrence Equations

A recurrence is an equation or inequality that describes a function in **terms of its value on smaller inputs**. Recurrences are generally used in divide-and-conquer paradigm.

Let us consider ***T(n)*** to be the running time on a problem of size **n**.

If the problem size is small enough, say ***n < c*** where **c** is a **constant**, the straightforward solution takes **constant time**, which is written as **θ (1)**. If the **division of the problem** yields a number of **sub-problems** with size ***nb***.

To solve the problem, the required time is ***a.T(nb)***. If we consider the time required for division is ***D(n)*** and the time required for combining the results of sub-problems is ***C(n)***, the recurrence relation can be represented as −

$$T(n)=\{\theta(1) \; a.T(nb)+D(n)+C(n) \; if \; n \leqslant c \; otherwise$$

A recurrence relation can be solved using the following methods −

- **Substitution Method** − In this method, we guess a bound and using mathematical induction we prove that our assumption was correct.
- **Recursion Tree Method** − In this method, a recurrence tree is formed where each node represents the cost.
- **Master's Theorem** − This is another important technique to find the complexity of a recurrence relation.

3.3.1 Substitution Method

The **substitution** method is a **trial and error** method; where the values that we might **think** could be the **solution** to the relation are substituted and check whether the equation is valid. If it is valid, the solution is **found**. Otherwise, another value is checked.

3.3.1.1 Procedure

The steps to solve recurrences using the substitution method are −

- Guess the form of solution based on the **trial-and-error** method
- Use Mathematical Induction to prove the solution is correct for all the cases.

Example

Let us look into an example to solve a recurrence using the substitution method,

T(n) = 2T(n/2) + n

Here, we assume that the time complexity for the equation is **O(nlogn)**. So according the mathematical induction phenomenon, the time complexity for **T(n/2)** will be **O(n/2logn/2)**; substitute the value into the given equation, and we need to prove that **T(n)** must be **greater than** or equal to **nlogn**.

≤ 2n/2Log(n/2) + n
= nLogn - nLog2 + n
= nLogn - n + n
≤ nLogn

3.3.2 Recurrence Tree Method

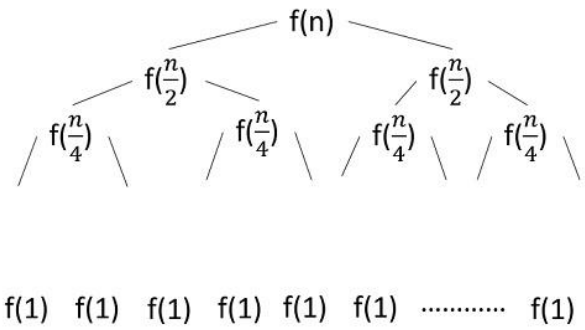
In the recurrence tree method, we draw a recurrence tree until the program cannot be divided into smaller parts further. Then we calculate the time taken in each level of the recurrence tree.

3.3.2.1 Procedure

- Draw the recurrence tree for the program
- Calculate the time complexity in every level and sum them up to find the total time complexity.

Example

Consider the binary search algorithm and construct a recursion tree for it −



Since the algorithm follows divide and conquer technique, the recursion tree is drawn until it reaches the smallest input level $T(n^{2k})$

$T(n^{2k})=T(1)$

$n=2^k$

Applying logarithm on both sides of the equation,

$\log n=\log^{2k}$

$k=\log_2 n$

Therefore, the time complexity of a binary search algorithm is **O(log n)**.

3.3.3 Master's Theorem

Master's theorem is one of the many methods that are applied to calculate time complexities of algorithms. In analysis, time complexities are calculated to find out the best optimal logic of an algorithm. Master's theorem is applied on recurrence relations.

But before we get deep into the master's theorem, let us first revise what recurrence relations are −

Recurrence relations are equations that define the sequence of elements in which a term is a function of its preceding term. In algorithm analysis, the recurrence relations are usually formed when loops are present in an algorithm.

3.3.3.1 Problem Statement

Master's theorem can only be applied on **decreasing** and **dividing recurring functions**. If the relation is not decreasing or dividing, master's theorem must not be applied.

3.3.3.2 Master's Theorem for Dividing Functions

Consider a relation of type −

$T(n) = aT(n/b) + f(n)$

where, **a >= 1** and **b > 1**,

n − size of the problem

a − number of sub-problems in the recursion

n/b − size of the sub problems based on the assumption that all sub-problems are of the same size.

f(n) − represents the cost of work done outside the recursion -> Θ(nk logn p) ,where k >= 0 and p is a real number;

If the recurrence relation is in the above given form, then there are three cases in the master theorem to determine the asymptotic notations −

- If $a > b^k$, then $T(n)= \Theta (n^{\log_b a})$ [$\log_b a = \log a / \log b$.]
- If $a = b^k$
  - If $p > -1$, then $T(n) = \Theta (n^{\log_b a} \log^{p+1} n)$
  - If $p = -1$, then $T(n) = \Theta (n^{\log_b a} \log \log n)$
  - If $p < -1$, then $T(n) = \Theta (n^{\log_b a})$
- If $a < b^k$,
  - If $p >= 0$, then $T(n) = \Theta (n^k \log^p n)$.
  - If $p < 0$, then $T(n) = \Theta (n^k)$

3.3.3.3 Master's Theorem for Decreasing Functions

Consider a relation of type −

$T(n) = aT(n-b) + f(n)$
where, $a >= 1$ and $b > 1$, $f(n)$ is asymptotically positive

Here,

n − size of the problem

a − number of sub-problems in the recursion

n-b − size of the sub problems based on the assumption that all sub-problems are of the same size.

f(n) − represents the cost of work done outside the recursion -> $\Theta(n^k)$, where k >= 0.

If the recurrence relation is in the above given form, then there are three cases in the master theorem to determine the asymptotic notations −

- if a = 1, T(n) = O (n^{k+1})
- if a > 1, T(n) = O ($a^{n/b} * n^k$)
- if a < 1, T(n) = O (n^k)

3.3.3.4 Examples

Few examples to apply master's theorem on *dividing recurrence relations* −

Example 1

Consider a recurrence relation given as $T(n) = 8T(n/2) + n^2$

In this problem, a = 8, b = 2 and f(n) = Θ(nk logn p) = n2,
giving us k = 2 and p = 0.
a = 8 > bk = 22 = 4,
Hence, case 1 must be applied for this equation.
To calculate, T(n) = Θ (nlogb a)
= nlog28
= n(log 8 / log 2)
= n3
Therefore, T(n) = Θ(n3) is the tight bound for this equation.

Example 2

Consider a recurrence relation given as $T(n) = 4T(n/2) + n^2$

In this problem, a = 4, b = 2 and f(n) = Θ(nk logn p) = n2,
giving us k = 2 and p = 0.
a = 4 = bk = 22 = 4, p > -1
Hence, case 2(i) must be applied for this equation.
To calculate, T(n) = Θ (nlogb a logp+1 n)

= nlog24 log0+1n
= n2logn
Therefore, T(n) = Θ(n2logn) is the tight bound for this equation.

Example 3

Consider a recurrence relation given as T(n) = 2T(n/2) + n/log n

In this problem, a = 2, b = 2 and f(n) = Θ(nk logn p) = n/log n,
giving us k = 1 and p = -1.
a = 2 = bk = 21 = 2, p = -1
Hence, case 2(ii) must be applied for this equation.
To calculate, T(n) = Θ (n logb a log log n)
= nlog44 log logn
= n.log(logn)
Therefore, T(n) = Θ(n.log(logn)) is the tight bound for this equation.

Example 4

Consider a recurrence relation given as $T(n) = 16T(n/4) + n^2/\log^2 n$

In this problem, a = 16, b = 4 and f(n) = Θ(nk logn p) = n2/log2n,
giving us k = 2 and p = -2.
a = 16 = bk = 42 = 16, p < -1
Hence, case 2(iii) must be applied for this equation.
To calculate, T(n) = Θ (n logb a)
= nlog416
= n2
Therefore, T(n) = Θ(n2) is the tight bound for this equation.

Example 5

Consider a recurrence relation given as $T(n) = 2T(n/2) + n^2$

In this problem, a = 2, b = 2 and f(n) = Θ(nk logn p) = n2,
giving us k = 2 and p = 0.
a = 2 < bk = 22 = 4, p = 0
Hence, case 3(i) must be applied for this equation.
To calculate, T(n) = Θ (nk logp n)
= n2 log0n
= n2
Therefore, T(n) = Θ(n2) is the tight bound for this equation.

Example 6

Consider a recurrence relation given as $T(n) = 2T(n/2) + n^3/\log n$

In this problem, a = 2, b = 2 and f(n) = Θ(nk logn p) = n3/log n,
giving us k = 3 and p = -1.
a = 2 < bk = 23 = 8, p < 0

Hence, case 3(ii) must be applied for this equation.
To calculate, T(n) = Θ (nk)
= n3
= n3
Therefore, T(n) = Θ(n3) is the tight bound for this equation.

Few examples to apply master's theorem in *decreasing recurrence relations* −

Example 1

Consider a recurrence relation given as $T(n) = T(n-1) + n^2$

In this problem, a = 1, b = 1 and f(n) = O(nk) = n2, giving us k = 2.
Since a = 1, case 1 must be applied for this equation.
To calculate, T(n) = O(nk+1)
= n2+1
= n3
Therefore, T(n) = O(n3) is the tight bound for this equation.

Example 2

Consider a recurrence relation given as T(n) = 2T(n-1) + n

In this problem, a = 2, b = 1 and f(n) = O(nk) = n, giving us k = 1.
Since a > 1, case 2 must be applied for this equation.
To calculate, T(n) = O(an/b * nk)
= O(2n/1 * n1)
= O(n2n)
Therefore, T(n) = O(n2n) is the tight bound for this equation.

Example 3

Consider a recurrence relation given as $T(n) = n^4$

In this problem, a = 0 and f(n) = O(nk) = n4, giving us k = 4
Since a < 1, case 3 must be applied for this equation.
To calculate, T(n) = O(nk)
= O(n4)
= O(n4)
Therefore, T(n) = O(n4) is the tight bound for this equation.

3.4 Exercises on methods of algorithm analysis

1. Find the time complexity for the following program?

```
for(i=1;i<n;i=i+2)
{
      stmt;
}
```

2. Find the time complexity for the following program?

```
p=0;
for(i=1;p<n;i++)
{
      p=p+i;
}
```

3. Find the time complexity for the following program?

```
for(i=1;p<n;i=i*2)
{
      stmt;
}
```

4. Find the time complexity for the following program?

```
a=1;
while(a<b)
{
      stmt;
      a=a*2;
}
```

Chapter 4

4 Sorting and Searching algorithm

4.1 Sorting algorithm

The **main idea** behind performing sorting is to arrange the data in an **orderly way**, making it easier to search for any element within the sorted data.

4.1.1 Bubble Sort Algorithm

Bubble sort is a simple sorting algorithm. This sorting algorithm is **comparison-based** algorithm in which each **pair of adjacent elements** is compared and the elements are **swapped** if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of **$O(n^2)$** where **n** is the number of items.

Bubble Sort is an elementary sorting algorithm, which works by **repeatedly exchanging adjacent elements**, if necessary. When no exchanges are required, the file is sorted.

We assume **list** is an array of **n** elements. We further assume that **swap** function swaps the values of the given array elements.

Step 1 − Check if the first element in the input array is greater than the next element in the array.

Step 2 − If it is greater, swap the two elements; otherwise move the pointer forward in the array.

Step 3 − Repeat Step 2 until we reach the end of the array.

Step 4 − Check if the elements are sorted; if not, repeat the same process (Step 1 to Step 3) from the last element of the array to the first.

Step 5 − The final output achieved is the sorted array.

```
Algorithm: Sequential-Bubble-Sort (A)
For i ← 1 to length [A] do
for j ← length [A] down-to i +1 do
  if A[A] < A[j-1] then
    Exchange A[j] ↔ A[j-1]
```

4.1.1.1 Pseudocode

We observe in algorithm that **Bubble Sort** compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.

To ease-out the issue, we use one flag variable **swapped** which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

Pseudocode of bubble sort algorithm can be written as follows −

```
voidbubbleSort(int numbers[], intarray_size){
  inti, j, temp;
  for (i = (array_size - 1); i>= 0; i--)
  for (j = 1; j <= i; j++)
  if (numbers[j-1] > numbers[j]){
    temp = numbers[j-1];
    numbers[j-1] = numbers[j];
    numbers[j] = temp;
  }
}
```

4.1.1.2 Analysis

Here, the number of comparisons is

$$1 + 2 + 3 + ... + (n - 1) = n(n - 1)/2 = O(n^2)$$

Clearly, the graph shows the n^2 nature of the bubble sort.

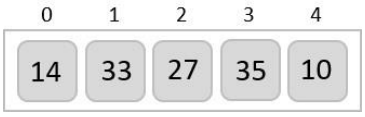
In this algorithm, the number of comparisons is irrespective of the data set, i.e. whether the provided input elements are in sorted order or in reverse order or at random.

4.1.1.3 Memory Requirement

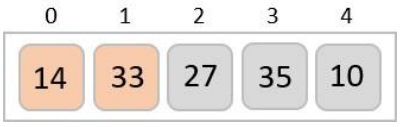
From the algorithm stated above, it is clear that bubble sort does not require extra memory.

Example

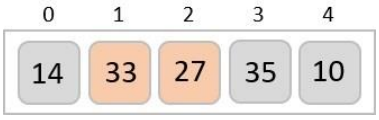
We take an unsorted array for our example. Bubble sort takes $O(n^2)$ time so we're keeping it short and precise.



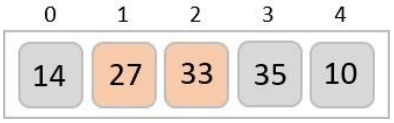
Bubble sort starts with very first two elements, comparing them to check which one is greater.



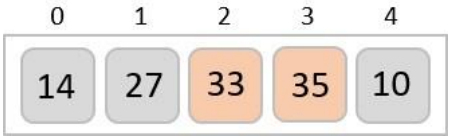
In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.



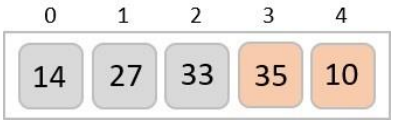
We find that 27 is smaller than 33 and these two values must be swapped.



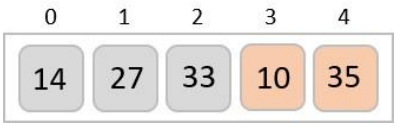
Next we compare 33 and 35. We find that both are in already sorted positions.



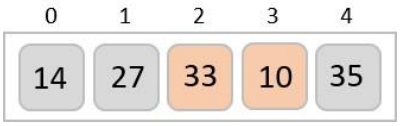
Then we move to the next two values, 35 and 10.

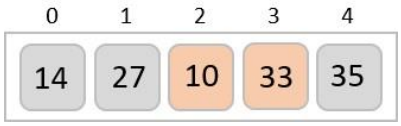


We know then that 10 is smaller 35. Hence they are not sorted. We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this −

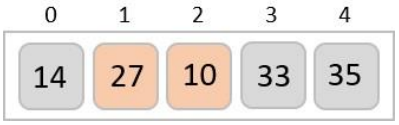


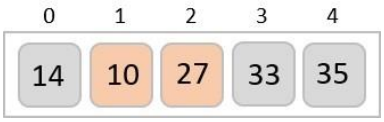
To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this −

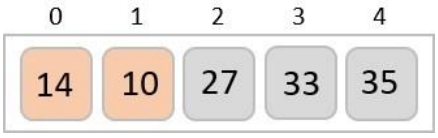


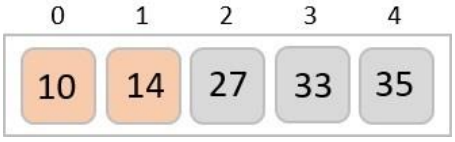


Notice that after each iteration, at least one value moves at the end.

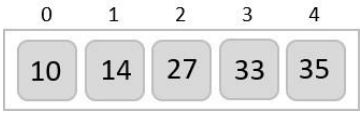








And when there's no swap required, bubble sort learns that an array is completely sorted.



Now we should look into some practical aspects of bubble sort.

4.1.1.4 Implementation

One more issue we did not address in our original algorithm and its improvised pseudocode, is that, after every iteration the highest values settles down at the end of the array. Hence, the next iteration need not include already sorted elements. For this purpose, in our implementation, we restrict the inner loop to avoid already sorted values.

Python code

```
def bubble_sort(array, size):
   for i in range(size):
      swaps = 0;
      for j in range(0, size-i-1):
         if(arr[j] > arr[j+1]):
            temp = arr[j];
            arr[j] = arr[j+1];
            arr[j+1] = temp;
            swaps = 1;
      if(swaps == 0):
         break;

arr = [67, 44, 82, 17, 20]
n = len(arr)
print("Array before Sorting: ")
print(arr)
bubble_sort(arr, n);
print("Array after Sorting: ")
print(arr)
```

Output

Array before Sorting:
[67, 44, 82, 17, 20]
Array after Sorting:
[17, 20, 44, 67, 82]

4.1.2 Insertion Sort Algorithm

Insertion sort is a very simple method to sort numbers in an **ascending or descending** order. This method follows the incremental method. It can be compared with the technique how cards are sorted at the time of playing a game.

This is an **in-place comparison-based** sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'inserted' in this sorted sub-list, has to **find** its appropriate **place** and then it has to be inserted there. Hence the name, **insertion sort**.

The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst-case complexity are of O(**n**^{2}), where **n** is the number of items.

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

Step 1 − If it is the first element, it is already sorted. return 1;

Step 2 − Pick next element

Step 3 − Compare with all elements in the sorted sub-list

Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5 − Insert the value

Step 6 − Repeat until list is sorted

4.1.2.1 Pseudocode

```
Algorithm: Insertion-Sort(A)
for j = 2 to A.length
   key = A[j]
   i = j – 1
   while i > 0 and A[i] > key
      A[i + 1] = A[i]
      i = i -1
   A[i + 1] = key
```

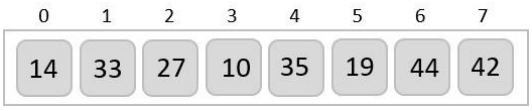
4.1.2.2 Analysis

Run time of this algorithm is very much dependent on the given input.

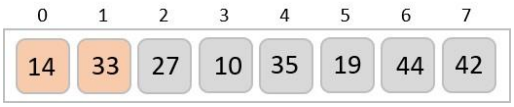
If the given numbers are sorted, this algorithm runs in **O(n)** time. If the given numbers are in reverse order, the algorithm runs in **O(n^2)** time.

Example

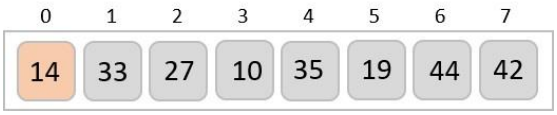
We take an unsorted array for our example.



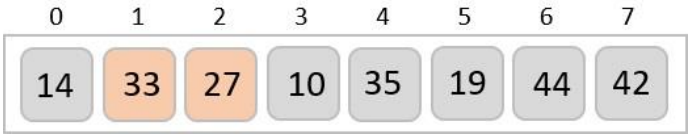
Insertion sort compares the first two elements.



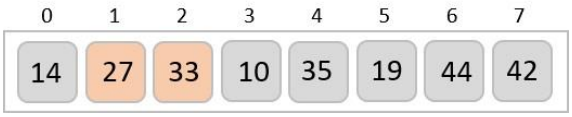
It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.



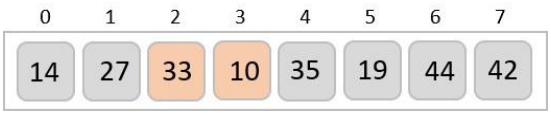
Insertion sort moves ahead and compares 33 with 27.



And finds that 33 is not in the correct position. It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

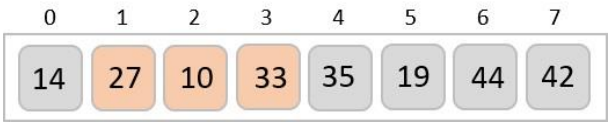


By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10. These values are not in a sorted order.



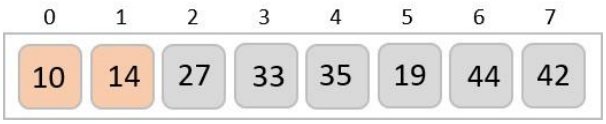
So they are swapped.

However, swapping makes 27 and 10 unsorted.



Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.



By the end of third iteration, we have a sorted sub-list of 4 items.

This process goes on until all the unsorted values are covered in a sorted sub-list. Now we shall see some programming aspects of insertion sort.

4.1.2.3 Implementation

Since insertion sort is an **in-place sorting** algorithm, the algorithm is implemented in a way where the **key element** – which is iteratively chosen as every element in the array – is **compared with it consequent elements** to check its position. If the **key element** is less than its successive element, the swapping is not done. Otherwise, the two elements compared will be swapped and the next element is chosen as the key element.

Insertion sort ac be implemented in Python −

Python code

```
def insertion_sort(array, size):
   for i in range(1, size):
      key = array[i]
      j = i
      while (j > 0) and (array[j-1] > key):
         array[j] = array[j-1]
         j = j-1
      array[j] = key

arr = [67, 44, 82, 17, 20]
n = len(arr)
print("Array before Sorting: ")
print(arr)
insertion_sort(arr, n);
print("Array after Sorting: ")
print(arr)
```

output

```
Array before Sorting:
[67, 44, 82, 17, 20]
Array after Sorting:
```

[17, 20, 44, 67, 82]

4.1.3 Selection Sort Algorithm

Selection sort is a simple sorting algorithm. This sorting algorithm, like insertion sort, is an **in-place comparison-based algorithm** in which the list is divided into two parts, the sorted part at the **left end** and the **unsorted part at the right** end. Initially, the sorted part is empty and the unsorted part is the entire list.

The **smallest** element is selected from the **unsorted array** and swapped with the **leftmost** element, and that element becomes a part of **the sorted array**. This process continues moving unsorted array boundaries by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of **$O(n^2)$**, where **n** is the number of items.

This type of sorting is called Selection Sort as it works by repeatedly sorting elements. That is: we first find the **smallest** value in the array and exchange it with the element in the **first position**, then find the second smallest element and exchange it with the element in the **second position**, and we continue the process in this way until the entire array is sorted.

1. Set MIN to location 0.
2. Search the minimum element in the list.
3. Swap with value at location MIN.
4. Increment MIN to point to next element.
5. Repeat until the list is sorted.

4.1.3.1 Pseudocode

```
Algorithm: Selection-Sort (A)
fori← 1 to n-1 do
  min j ←i;
  min x ← A[i]
  for j ←i + 1 to n do
    if A[j] < min x then
      min j ← j
      min x ← A[j]
  A[min j] ← A [i]
  A[i] ← min x
```

4.1.3.2 Analysis

Selection sort is among the simplest of sorting techniques and it works very well for **small files**. It has a quite important application as each item is actually moved at the most once.

Selection sort is a method of choice for sorting files with very large objects (records) and small keys. The **worst case** occurs if the array is already sorted in a **descending** order and we want to sort them in an ascending order.

Selection sort spends most of its time trying to find the **minimum** element in the unsorted part of the array. It clearly shows the similarity between Selection sort and Bubble sort.

- Bubble sort selects the maximum remaining elements at each stage, but wastes some effort imparting some order to an unsorted part of the array.
- Selection sort is quadratic in both the **worst** and the average case, and requires no extra memory.

For each **i** from **1** to **n - 1**, there is one exchange and **n - i** comparisons, so there is a total of **n - 1** exchanges and

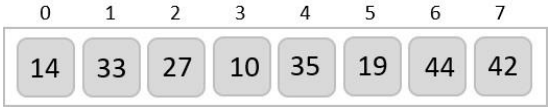
(n − 1) + (n − 2) + ...+2 + 1 = n(n − 1)/2 comparisons.

These observations hold, no matter what the input data is.

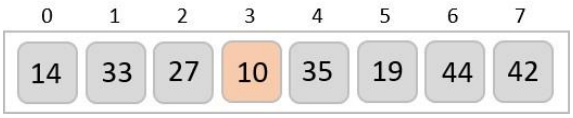
In the worst case, this could be quadratic, but in the average case, this quantity is **O(n log n)**. It implies that the **running time of Selection sort is quite insensitive to the input**.

4.1.4 Example

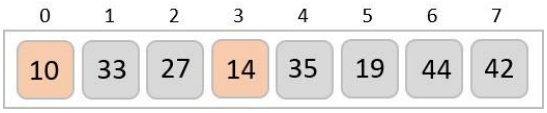
Consider the following depicted array as an example.



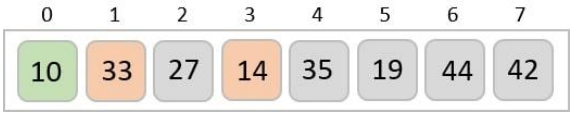
For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



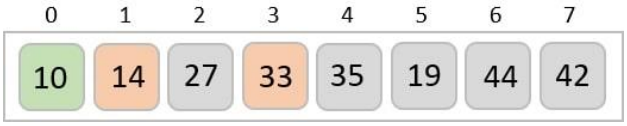
So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.



For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.



We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the beginning in a sorted manner.

The same process is applied to the rest of the items in the array −

4.1.4.1 Implementation

The selection sort algorithm is implemented in four different programming languages below. The given program selects the minimum number of the array and swaps it with the element in the first index. The second minimum number is swapped with the element present in the second index. The process goes on until the end of the array is reached.

Python code

```
def selection_sort(array, size):
   for i in range(size):
      imin = i
      for j in range(i+1, size):
         if arr[j] < arr[imin]:
            imin = j
      temp = array[i];
      array[i] = array[imin];
      array[imin] = temp;

arr = [12, 19, 55, 2, 16]
n = len(arr)
print("Array before Sorting: ")
print(arr)
insertion_sort(arr, n);
print("Array after Sorting: ")
print(arr)
```

output

Array before Sorting:
[12, 19, 55, 2, 16]
Array after Sorting:
[2, 12, 16, 19, 55]

4.1.5 Shell Sort Algorithm

Shell sort is a **highly efficient** sorting algorithm and is based on **insertion sort algorithm**. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

This algorithm uses insertion sort on a widely spread elements, first to sort them and then sorts the less widely spaced elements. This spacing is termed as **interval**. This interval is calculated based on **Knuth's formula** as −

h = h * 3 + 1
where −
h is interval with initial value 1

This algorithm is quite efficient for **medium-sized data** sets as its average and worst-case complexity are of **O(n)**, where **n** is the number of items.

Following is the algorithm for shell sort.

1. Initialize the value of h.
2. Divide the list into smaller sub-list of equal interval h.
3. Sort these sub-lists using insertion sort.
4. Repeat until complete list is sorted.

4.1.5.1 Pseudocode

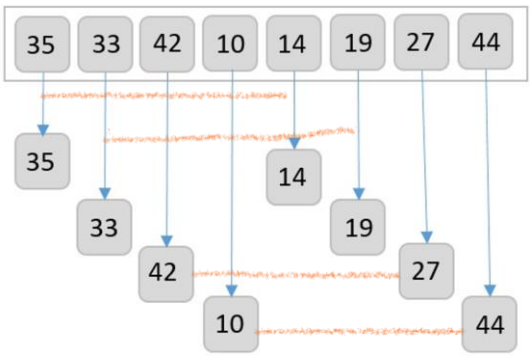
Following is the pseudocode for shell sort.

```
procedure shellSort()
   A : array of items
   /* calculate interval*/
   while interval < A.length /3 do:
      interval = interval * 3 + 1
   end while
   while interval > 0 do:
      for outer = interval; outer < A.length; outer ++ do:
         /* select value to be inserted */
         valueToInsert = A[outer]
         inner = outer
            /*shift element towards right*/
            while inner > interval -1 &&  A[inner - interval]
                                    >= valueToInsert do:
               A[inner] = A[inner - interval]
               inner = inner – interval
            end while
         /* insert the number at hole position */
         A[inner] = valueToInsert
         end for
   /* calculate interval*/
   interval = (interval -1) /3;
   end while
end procedure
```

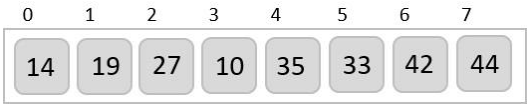
Example

Let us consider the following example to have an idea of how shell sort works. We take the same array we have used in our previous examples. For our example and ease of

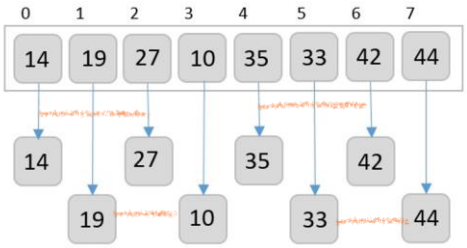
understanding, we take the interval of 4. Make a virtual sub-list of all values located at the interval of 4 positions. Here these values are {35, 14}, {33, 19}, {42, 27} and {10, 44}



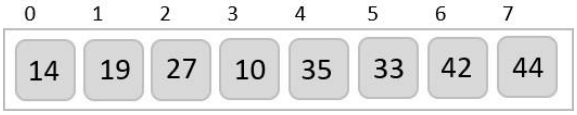
We compare values in each sub-list and swap them (if necessary) in the original array. After this step, the new array should look like this −



Then, we take interval of 2 and this gap generates two sub-lists - {14, 27, 35, 42}, {19, 10, 33, 44}



We compare and swap the values, if required, in the original array. After this step, the array should look like this −



Finally, we sort the rest of the array using interval of value 1. Shell sort uses insertion sort to sort the array.

Following is the step-by-step depiction −

We see that it required only four swaps to sort the rest of the array.

4.1.5.2 Implementation

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

Python code

```
def shell_sort(array,n):
   gap = n//2 #using floor division to avoid float values as result
   while gap > 0:
      for i in range(int(gap),n):
         temp = array[i]
         j = i
         while j >= gap and array[j-gap] >temp:
            array[j] = array[j-gap]
            j -= gap
            array[j] = temp
      gap = gap // 2 #using floor division to avoid float values as
result
```

```
arr = [33, 45, 62, 12, 98]
n = len(arr)
print("Array before Sorting: ")
print(arr)
shell_sort(arr, n);
print("Array after Sorting: ")
print(arr)
```

output

Array before Sorting:
[33, 45, 62, 12, 98]
Array after Sorting:
[12, 33, 45, 62, 98]

4.1.6 Quick sort algorithm

Quick sort is a highly efficient sorting algorithm and is based on **partitioning of array of data into smaller arrays**. A large array is partitioned into two arrays one of which holds values smaller than the specified value, **say pivot**, based on which the partition is made and another array holds values greater than the pivot value.

Quicksort partitions an array and then calls itself recursively twice to sort the two resulting subarrays. This algorithm is quite efficient for large-sized data sets as its average and worst-case complexity are O(n2), respectively.

4.1.6.1 Partition in Quick Sort

Following animated representation explains how to find the pivot value in an array.

The pivot value divides the list into two parts. And recursively, we find the pivot for each sub-lists until all lists contains only one element.

4.1.6.2 Quick Sort Pivot Algorithm

Based on our understanding of partitioning in quick sort, we will now try to write an algorithm for it, which is as follows.

1. Choose the highest index value has pivot
2. Take two variables to point left and right of the list excluding pivot
3. Left points to the low index
4. Right points to the high
5. While value at left is less than pivot move right
6. While value at right is greater than pivot move left
7. If both step 5 and step 6 does not match swap left and right
8. If left ≥ right, the point where they met is new pivot

4.1.6.3 Quick Sort Pivot Pseudocode

The pseudocode for the above algorithm can be derived as −

```
function partitionFunc(left, right, pivot)
   leftPointer = left
   rightPointer = right - 1

   while True do
      while A[++leftPointer] < pivot do
      //do-nothing
      end while

      while rightPointer > 0 && A[--rightPointer] > pivot do
         //do-nothing
      end while

      if leftPointer >= rightPointer
         break
      else
         swap leftPointer,rightPointer
      end if
   end while

   swap leftPointer,right
   return leftPointer
end function
```

4.1.6.4 Quick Sort Algorithm

Using pivot algorithm recursively, we end up with smaller possible partitions. Each partition is then processed for quick sort. We define recursive algorithm for quicksort as follows −

1. Make the right-most index value pivot
2. Partition the array using pivot value
3. Quicksort left partition recursively

4. Quicksort right partition recursively

4.1.6.5 Quick Sort Pseudocode

To get more into it, let see the pseudocode for quick sort algorithm −

```
procedure quickSort(left, right)
   if right-left <= 0
      return
   else
      pivot = A[right]
      partition = partitionFunc(left, right, pivot)
      quickSort(left,partition-1)
      quickSort(partition+1,right)
   end if
end procedure
```

4.1.6.6 Analysis

The worst case complexity of Quick-Sort algorithm is **O(n^2)**. However, using this technique, in average cases generally we get the output in **O (n log n)** time.

4.1.6.7 Implementation

Following are the implementations of Quick Sort algorithm in various programming languages –

Python code

```python
def partition(arr, low, high):
    i = low - 1
    pivot = arr[high]  # pivot element
    for j in range(low, high):
        if arr[j] <= pivot:
            # increment
            i = i + 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

def quickSort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quickSort(arr, low, pi - 1)
        quickSort(arr, pi + 1, high)
```

```
arr = [2, 5, 3, 8, 6, 5, 4, 7]
n = len(arr)
print("Contents of the array: ")
for i in range(n):
   print(arr[i], end=" ")
quickSort(arr, 0, n - 1)
print("\nContents of the array after sorting: ")
for i in range(n):
   print(arr[i], end=" ")
```

output

Contents of the array:
2 5 3 8 6 5 4 7
Contents of the array after sorting:
2 3 4 5 5 6 7 8

4.1.7 Radix Sort Algorithm

Radix sort is a step-wise sorting algorithm that starts the sorting from the least significant digit of the input elements. Like Counting Sort and Bucket Sort, Radix sort also assumes something about the input elements, that they are all k-digit numbers.

The sorting starts with the least significant digit of each element. These least significant digits are all considered individual elements and sorted first; followed by the second least significant digits. This process is continued until all the digits of the input elements are sorted.

Note − If the elements do not have same number of digits, find the maximum number of digits in an input element and add leading zeroes to the elements having less digits. It does not change the values of the elements but still makes them k-digit numbers.

The radix sort algorithm makes use of the counting sort algorithm while sorting in every phase. The detailed steps are as follows −

Step 1 − Check whether all the input elements have same number of digits. If not, check for numbers that have maximum number of digits in the list and add leading zeroes to the ones that do not.

Step 2 − Take the least significant digit of each element.

Step 3 − Sort these digits using counting sort logic and change the order of elements based on the output achieved. For example, if the input elements are decimal numbers, the possible values each digit can take would be 0-9, so index the digits based on these values.

Step 4 − Repeat the Step 2 for the next least significant digits until all the digits in the elements are sorted.

Step 5 − The final list of elements achieved after kth loop is the sorted output.

4.1.7.1 Pseudocode

Algorithm: RadixSort(a[], n):

```
// Find the maximum element of the list
max = a[0]
for (i=1 to n-1):
   if (a[i]>max):
      max=a[i]

// applying counting sort for each digit in each number of
//the input list
For (pos=1 to max/pos>0):
   countSort(a, n, pos)
   pos=pos*10
```

The *countSort* algorithm called would be −

```
Algorithm: countSort(a, n, pos)
  Initialize count[0…9] with zeroes
  for i = 0 to n:
     count[(a[i]/pos) % 10]++
  for i = 1 to 10:
     count[i] = count[i] + count[i-1]
  for i = n-1 to 0:
     output[count[(a[i]/pos) % 10]-1] = a[i]
     i--
  for i to n:
     a[i] = output[i]
```

4.1.7.2 Analysis

Given that there are k-digits in the input elements, the running time taken by the radix sort algorithm would be ***Θ(k(n + b)***. Here, n is the number of elements in the input list while b is the number of possible values each digit of a number can take.

Example

For the given unsorted list of elements, 236, 143, 26, 42, 1, 99, 765, 482, 3, 56, we need to perform the radix sort and obtain the sorted output list −

Step 1

Check for elements with maximum number of digits, which is 3. So we add leading zeroes to the numbers that do not have 3 digits. The list we achieved would be −

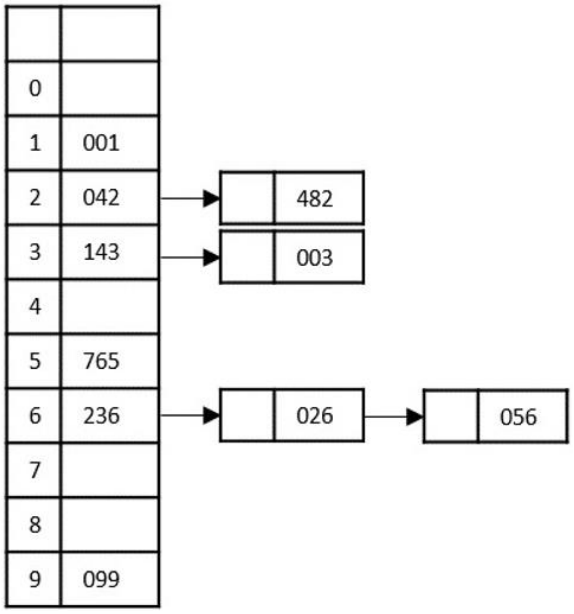
236, 143, 026, 042, 001, 099, 765, 482, 003, 056

Step 2

Construct a table to store the values based on their indexing. Since the inputs given are decimal numbers, the indexing is done based on the possible values of these digits, i.e., 0-9.

Step 3

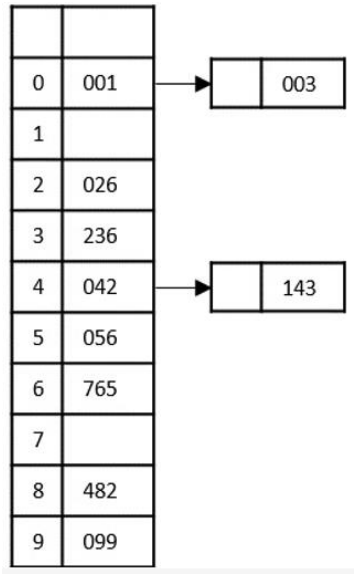
Based on the least significant digit of all the numbers, place the numbers on their respective indices.



The elements sorted after this step would be 001, 042, 482, 143, 003, 765, 236, 026, 056, 099.

Step 4

The order of input for this step would be the order of the output in the previous step. Now, we perform sorting using the second least significant digit.



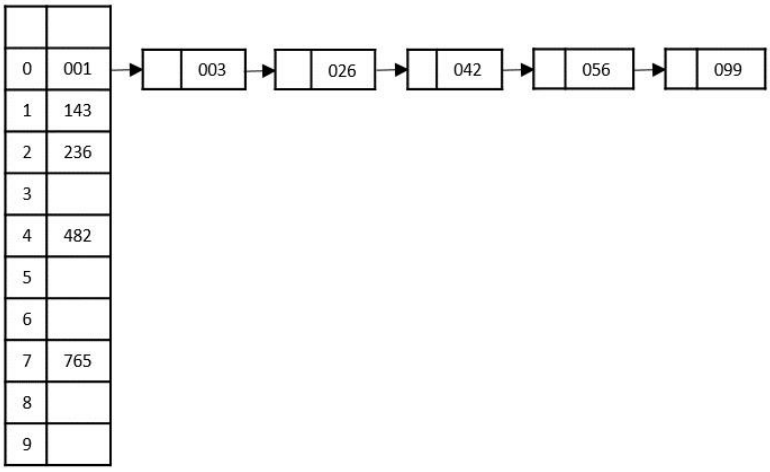
The order of the output achieved is 001, 003, 026, 236, 042, 143, 056, 765, 482, 099.

Step 5

The input list after the previous step is rearranged as −

001, 003, 026, 236, 042, 143, 056, 765, 482, 099

Now, we need to sort the last digits of the input elements.



Since there are no further digits in the input elements, the output achieved in this step is considered as the final output.

The final sorted output is −

1, 3, 26, 42, 56, 99, 143, 236, 482, 765

4.1.7.3 Implementation

The counting sort algorithm assists the radix sort to perform sorting on multiple d-digit numbers iteratively for 'd' loops. Radix sort is implemented in four programming languages in this tutorial: C, C++, Java, Python.

Python

```
def countsort(a, pos):
    n = len(a)
    output = [0] * n
    count = [0] * 10
    for i in range(0, n):
        idx = a[i] // pos
        count[idx % 10] += 1
    for i in range(1, 10):
        count[i] += count[i - 1]
    i = n - 1
    while i >= 0:
        idx = a[i] // pos
        output[count[idx % 10] - 1] = a[i]
        count[idx % 10] -= 1
        i -= 1
    for i in range(0, n):
        a[i] = output[i]

def radixsort(a):
    maximum = max(a)
    pos = 1
    while maximum // pos > 0:
        countsort(a, pos)
        pos *= 10

a = [236, 15, 333, 27, 9, 108, 76, 498]
print("Before sorting array elements are: ")
print (a)
radixsort(a)
print("After sorting array elements are: ")
print (a)
```

output

Before sorting array elements are:
[236, 15, 333, 27, 9, 108, 76, 498]

After sorting array elements are:
[9, 15, 27, 76, 108, 236, 333, 498]

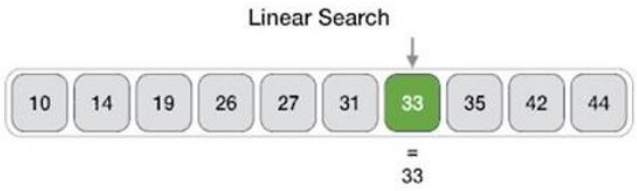
4.2 Searching algorithm

Searching is a process of **finding a particular record**, which can be a **single** element or a **small chunk**, within a huge amount of data. The data can be in various forms: **arrays**, **linked lists**, **trees**, **heaps**, and **graphs** etc. With the increasing amount of data nowadays, there are multiple techniques to perform the searching operation.

4.2.1 Linear Search Algorithm

Linear search is a type of **sequential** searching algorithm. In this method, every element within the input array is traversed and compared with the **key element to be found**. If a match is found in the array the search is said to be successful; if there is no match found the search is said to be unsuccessful and gives the worst-case time complexity.

For instance, in the given animated diagram, we are searching for an element 33. Therefore, the linear search method searches for it sequentially from the very first element until it finds a match. This returns a successful search.



In the same diagram, if we have to search for an element 46, then it returns an unsuccessful search since 46 is not present in the input.

The algorithm for linear search is relatively simple. The procedure starts at the very first index of the input array to be searched.

Step 1 − Start from the 0th index of the input array, compare the key value with the value present in the 0th index.

Step 2 − If the value matches with the key, return the position at which the value was found.

Step 3 − If the value does not match with the key, compare the next element in the array.

Step 4 − Repeat Step 3 until there is a match found. Return the position at which the match was found.

Step 5 − If it is an unsuccessful search, print that the element is not present in the array and exit the program.

4.2.1.1 Pseudocode

```
procedure linear_search (list, value)
   for each item in the list
      if match item == value
         return the item's location
      end if
   end for
end procedure
```

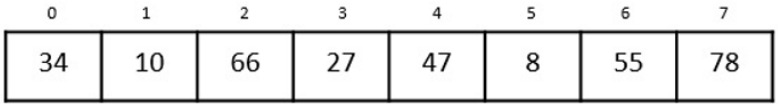
4.2.1.2 Analysis

Linear search traverses through every element sequentially therefore, the best case is when the element is found in the very first iteration. The best-case time complexity would be **O(1)**.

However, the worst case of the linear search method would be an unsuccessful search that does not find the key value in the array, it performs n iterations. Therefore, the worst-case time complexity of the linear search algorithm would be **O(n)**.

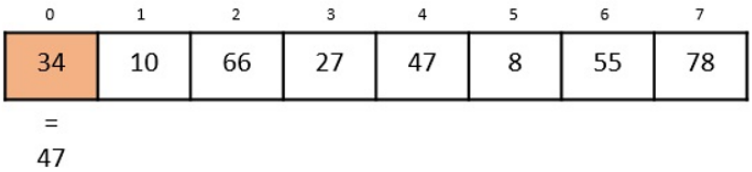
Example

Let us look at the step-by-step searching of the key element (say 47) in an array using the linear search method.



Step 1

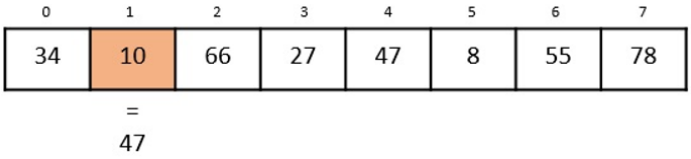
The linear search starts from the 0^{th} index. Compare the key element with the value in the 0^{th} index, 34.



However, 47 ≠ 34. So, it moves to the next element.

Step 2

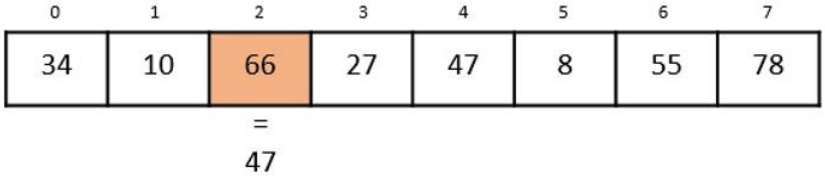
Now, the key is compared with value in the 1st index of the array.



Still, 47 ≠ 10, making the algorithm move for another iteration.

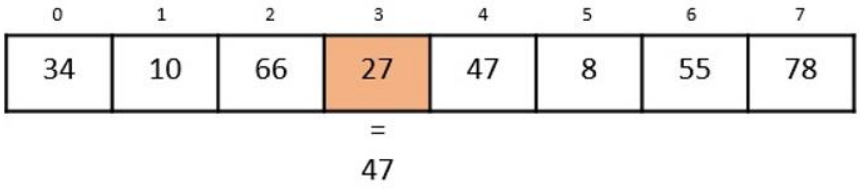
Step 3

The next element 66 is compared with 47. They are both not a match so the algorithm compares the further elements.



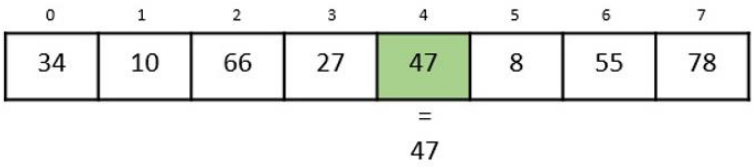
Step 4

Now the element in 3rd index, 27, is compared with the key value, 47. They are not equal so the algorithm is pushed forward to check the next element.



Step 5

Comparing the element in the 4th index of the array, 47, to the key 47. It is figured that both the elements match. Now, the position in which 47 is present, i.e., 4 is returned.



The output achieved is “Element found at 4th index”.

4.2.1.3 Implementation

In this tutorial, the Linear Search program can be seen implemented in four programming languages. The function compares the elements of input with the key value and returns the position of the key in the array or an unsuccessful search prompt if the key is not present in the array.

Python code

```
def linear_search(a, n, key):
   count = 0
   for i in range(n):
      if(a[i] == key):
         print("The element is found at position", (i+1))
         count = count + 1
   if(count == 0):
      print("The element is not present in the array")

a = [14, 56, 77, 32, 84, 9, 10]
n = len(a)
key = 32
linear_search(a, n, key)
key = 3
linear_search(a, n, key)
```

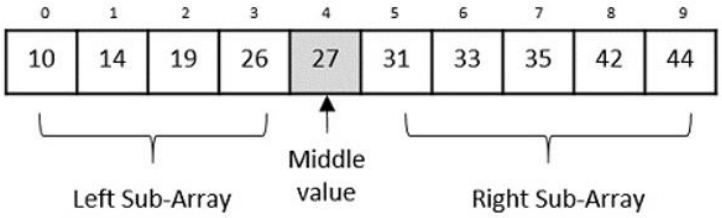
output

The element is found at position 4
The element is not present in the array

4.2.2 Binary Search Algorithm

Binary search is a fast search algorithm with run-time complexity of **O(log n)**. This search algorithm works on the principle of **divide and conquer**, since it **divides** the array into **half before searching**. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular **key** value by comparing the **middle most item** of the collection. If a **match occurs**, then the index of item is **returned**. But if the **middle** item has a value **less** than the **key** value, the **right sub-array** of the middle item is searched. Otherwise, the **left sub-array** is searched. This process continues recursively until the size of a subarray reduces to zero.



Binary Search algorithm is an interval searching method that performs the searching in intervals only. The input taken by the binary search algorithm must always be in a sorted array since it divides the array into subarrays based on the greater or lower values. The algorithm follows the procedure below −

Step 1 − Select the middle item in the array and compare it with the key value to be searched. If it is matched, return the position of the median.

Step 2 − If it does not match the key value, check if the key value is either greater than or less than the median value.

Step 3 − If the key is greater, perform the search in the right sub-array; but if the key is lower than the median value, perform the search in the left sub-array.

Step 4 − Repeat Steps 1, 2 and 3 iteratively, until the size of sub-array becomes 1.

Step 5 − If the key value does not exist in the array, then the algorithm returns an unsuccessful search.

4.2.2.1 Pseudocode

The pseudocode of binary search algorithms should look like this −

```
Procedure binary_search
   A ← sorted array
   n ← size of array
   x ← value to be searched
   Set lowerBound = 1
   Set upperBound = n
   while x not found
      if upperBound < lowerBound
         EXIT: x does not exists.
      set midPoint = lowerBound + (upperBound - lowerBound ) / 2
      if A[midPoint] < x
         set lowerBound = midPoint + 1
      if A[midPoint] > x
         set upperBound = midPoint - 1
      if A[midPoint] = x
         EXIT: x found at location midPoint
   end while
end procedure
```

4.2.2.2 Analysis

Since the binary search algorithm performs searching iteratively, calculating the time complexity is not as easy as the linear search algorithm.

The input array is searched **iteratively** by **dividing into multiple sub-arrays** after every unsuccessful iteration. Therefore, the recurrence relation formed would be of a dividing function.

To explain it in simpler terms,

- During the first iteration, the element is searched in the entire array. Therefore, length of the array = **n**.
- In the second iteration, only half of the original array is searched. Hence, length of the array = **n/2**.
- In the third iteration, half of the previous sub-array is searched. Here, length of the array will be = n/4.
- Similarly, in the **i^{th}** iteration, the length of the array will become **$n/2^i$**

To achieve a successful search, after the last iteration the length of array must be 1. Hence,

$n/2^i = 1$

That gives us

$n = 2^i$

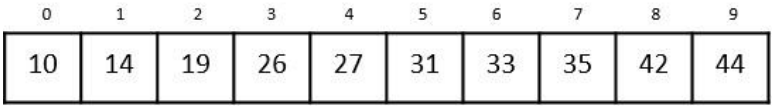
Applying log on both sides,

$\log n = \log 2^i$
$\log n = i. \log 2$
$i = \log n$

The time complexity of the binary search algorithm is **O(log n)**

Example

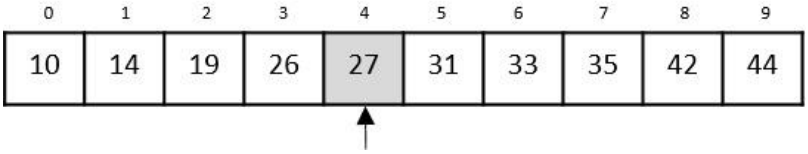
For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.



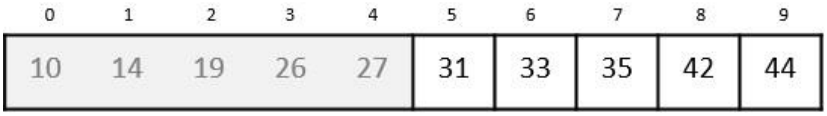
First, we shall determine half of the array by using this formula −

mid = low + (high - low) / 2

Here it is**, 0 + (9 - 0) / 2 = 4** (integer value of **4.5**). So, **4** is the mid of the array.



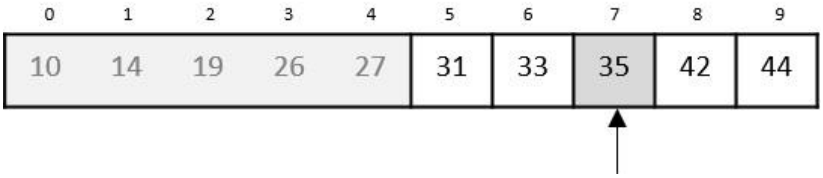
Now we compare the value stored at **location 4**, with the value being searched, i.e. **31**. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the **upper portion** of the array.



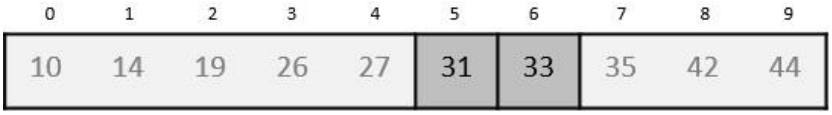
We change our low to mid + 1 and find the new mid value again.

low = mid + 1
mid = low + (high - low) / 2

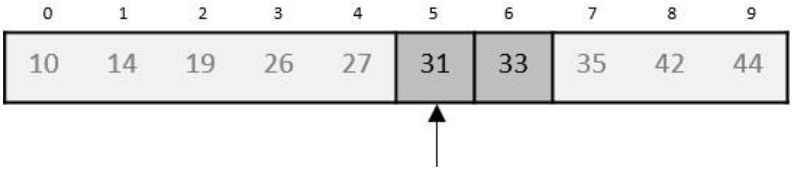
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



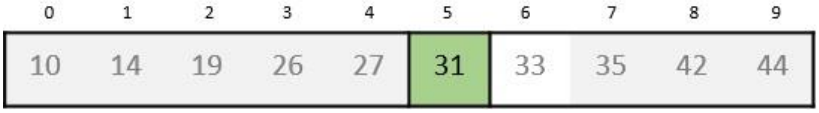
The value stored at location 7 is not a match, rather it is less than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

4.2.2.3 Implementation

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in a sorted form.

Python code

```
def binary_search(a, low, high, key):
   mid = (low + high) // 2
   if (low <= high):
      if(a[mid] == key):
         print("The element is present at index:", mid)
      elif(key < a[mid]):
         binary_search(a, low, mid-1, key)
      elif (a[mid] < key):
         binary_search(a, mid+1, high, key)
   if(low > high):
      print("Unsuccessful Search")
a = [6, 12, 14, 18, 22, 39, 55, 182]
n = len(a)
low = 0
high = n-1
key = 22
binary_search(a, low, high, key)
```

```
key = 54
binary_search(a, low, high, key)
```

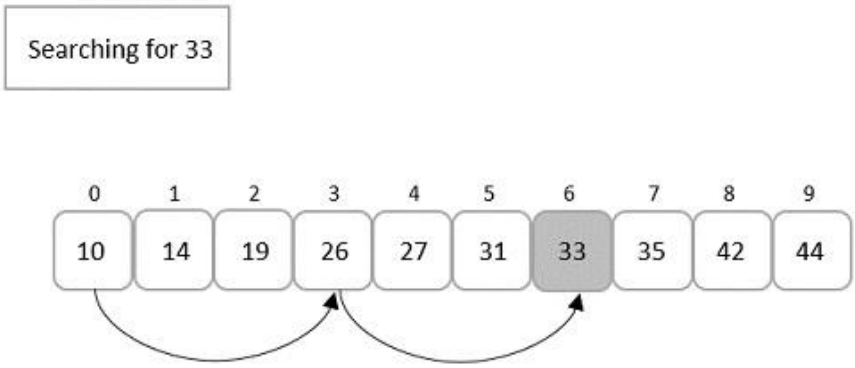
output

The element is present at index: 4
Unsuccessful Search

4.2.3 Jump Search Algorithm

Jump Search algorithm is a slightly **modified** version of the **linear** search algorithm. The main idea behind this algorithm is to reduce the time complexity by comparing lesser elements than the linear search algorithm. The input array is hence sorted and divided into blocks to perform searching while jumping through these blocks.

For example, let us look at the given example below; the sorted input array is searched in the blocks of 3 elements each. The desired key is found only after 2 comparisons rather than the 6 comparisons of the linear search.



Here, there arises a question about **how to divide these blocks**. To answer that, if the input array is of size **'n'**, the blocks are divided in the intervals of √**n**. **First** element of every block is compared with the **key** element until the key element's value is less than the **block element**. Linear search is performed only on that previous block since the input is sorted. If the element is found, it is a successful search; otherwise, an unsuccessful search is returned.

The jump search algorithm takes a sorted array as an input which is divided into **smaller blocks** to make the search **simpler**. The algorithm is as follows −

Step 1 − If the size of the input array is **'n'**, then the size of the block is √**n**. Set **i = 0**.

Step 2 − The key to be searched is compared with the **ith** element of the array. If it is a match, the position of the element is returned; otherwise **i** is incremented with the block size.

Step 3 − The Step 2 is repeated until the **ith** element is greater than the key element.

Step 4 − Now, the element is figured to be in the previous block, since the input array is sorted. Therefore, linear search is applied on that block to find the element.

Step 5 − If the element is found, the position is returned. If the element is not found, unsuccessful search is prompted.

4.2.3.1 Pseudocode

```
Begin
  blockSize := √size
  start := 0
  end := blockSize
  while array[end] <= key AND end < size do
    start := end
    end := end + blockSize
    if end > size – 1 then
      end := size
  done
  for i := start to end -1 do
    if array[i] = key then
      return i
  done
  return invalid location
End
```

4.2.3.2 Analysis

The time complexity of the jump search technique is **O(√n)** and space complexity is **O(1)**.

Example

Let us understand the jump search algorithm by searching for element 66 from the given sorted array, A, below −

0	1	2	3	4	5	6	7	8	9	10	11
0	6	12	14	19	22	48	66	79	88	104	126

Step 1

Initialize **i = 0**, and size of the input array **'n' = 12**

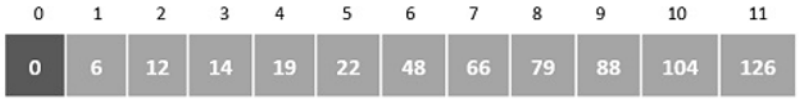
Suppose, block size is represented as 'm'. Then, **m = √n = √12 = 3**

Step 2

Compare **A[0]** with the key element and check whether it matches,

A[0] = 0 ≠ 66

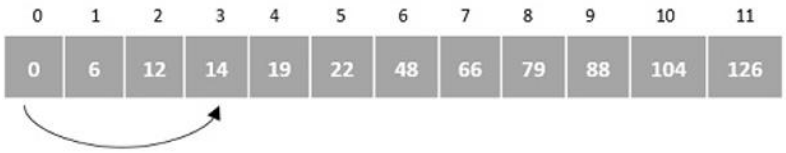
Therefore, **i** is incremented by the block **size = 3**. Now the element compared with the key element is **A[3].**



Step 3

A[3] = 14 ≠ 66

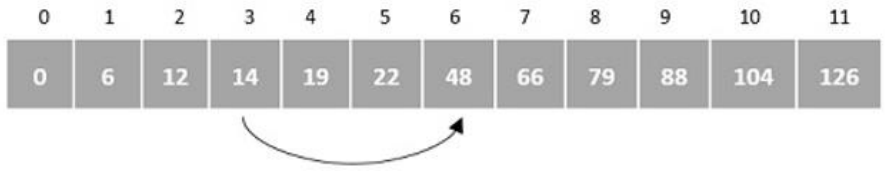
Since it is not a match, i is again incremented by 3.



Step 4

A[6] = 48 ≠ 66

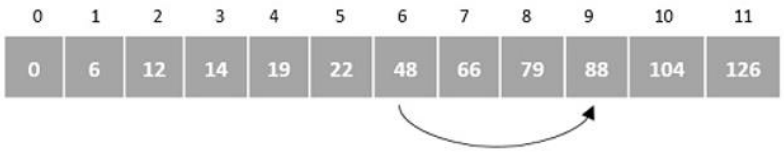
i is incremented by **3** again. **A[9]** is compared with the key element.



Step 5

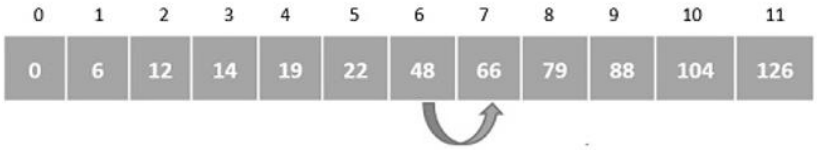
A [9] = 88 ≠ 66

However, 88 is greater than 66, therefore linear search is applied on the current block.



Step 6

After applying linear search, the pointer increments from **6th** index to **7th**. Therefore, A[7] is compared with the key element.



We find that **A[7**] is the required element, hence the program returns 7th index as the output.

4.2.3.3 Implementation

The jump search algorithm is an extended variant of linear search. The algorithm divides the input array into multiple small blocks and **performs the linear search on a single block** that is assumed to contain the element. If the element is not found in the assumed blocked, it returns an unsuccessful search.

The output prints the **position** of the element in the array instead of its index. Indexing refers to the index numbers of the array that start from 0 while position is the place where the element is stored.

Python code

```
import math
def jump_search(a, n, key):
    i = 0
    m = int(math.sqrt(n))
    k = m
    while(a[m] <= key and m < n):
        i = m
        m += k
        if(m > n - 1):
            return -1
    for j in range(m):
```

```
      if(arr[j] == key):
         return j
   return -1
arr = [0, 6, 12, 14, 19, 22, 48, 66, 79, 88, 104, 126]
n = len(arr);
print("Array elements are: ")
for i in range(n):
   print(arr[i], end = " ")
key = 66
print("\nThe element to be searched: ", key)
index = jump_search(arr, n, key)
if(index >= 0):
   print("The element is found at position: ", (index+1))
else:
   print("\nUnsuccessful Search")
```

output

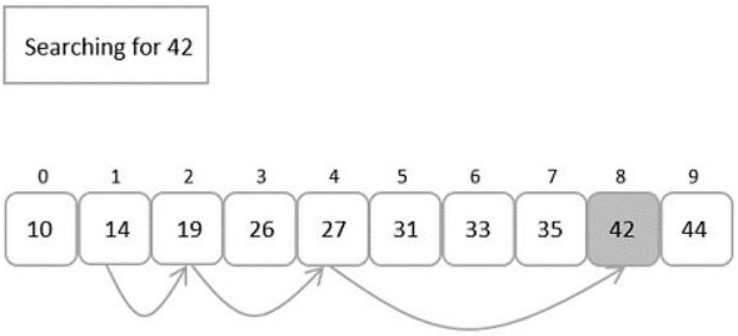
```
Array elements are:
0 6 12 14 19 22 48 66 79 88 104 126
The element to be searched:  66
The element is found at position:  8
```

4.2.4 Exponential Search Algorithm

Exponential search algorithm targets a range of an input array in which it assumes that the required element must be present in and performs a **binary search on that particular small range**. This algorithm is also known as doubling search or finger search.

It is similar to jump search in **dividing** the **sorted input into multiple blocks** and conducting a smaller scale search. However, the difference occurs while performing **computations** to divide the **blocks** and the type of smaller scale search applied (jump search applies linear search and exponential search applies binary search).

Hence, this algorithm jumps exponentially in the powers of **2**. In simpler words, the search is performed on the blocks divided using **pow(2, k**) where **k** is an integer greater than or equal to 0. Once the element at position pow(2, n) is greater than the key element, binary search is performed on the current block.



In the exponential search algorithm, the jump starts from the **1st index** of the array. So we manually compare the first element as the first step in the algorithm.

Step 1 − Compare the first element in the array with the key, if a match is found return the 0th index.

Step 2 − Initialize i = 1 and compare the **ith** element of the array with the key to be search. If it matches return the index.

Step 3 − If the element does not match, jump through the array exponentially in the powers of **2**. Therefore, now the algorithm compares the element present in the incremental position.

Step 4 − If the match is found, the index is returned. Otherwise Step **2** is repeated iteratively until the element at the incremental position becomes greater than the key to be searched.

Step 5 − Since the next increment has the higher element than the key and the input is sorted, the algorithm applies binary search algorithm on the current block.

Step 6 − The index at which the key is present is returned if the match is found; otherwise it is determined as an unsuccessful search.

4.2.4.1 Pseudocode

```
Begin
  m := pow(2, k) // m is the block size
  start := 1
  low := 0
  high := size – 1 // size is the size of input
  if array[0] == key
    return 0
  while array[m] <= key AND m < size do
    start := start + 1
    m := pow(2, start)
    while low <= high do:
      mid = low + (high - low) / 2
      if array[mid] == x
```

```
        return mid
      if array[mid] < x
        low = mid + 1
      else
        high = mid - 1
  done
  return invalid location
End
```

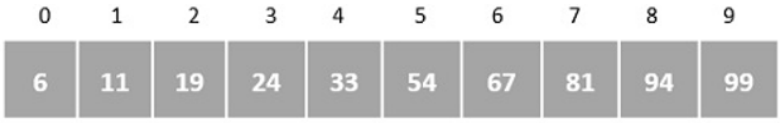
4.2.4.2 Analysis

Even though it is called Exponential search it does not perform searching in exponential time complexity. But as we know, in this search algorithm, the basic search being performed is binary search. Therefore, the time complexity of the exponential search algorithm will be the same as the binary search algorithm's, **O(log n)**.

Example

To understand the exponential search algorithm better and in a simpler way, let us search for an element in an example input array using the exponential search algorithm −

The sorted input array given to the search algorithm is −

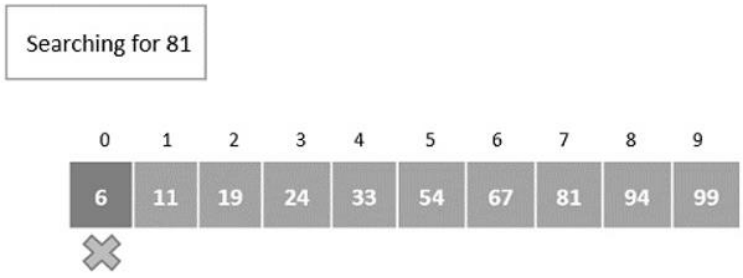


Let us search for the position of element 81 in the given array.

Step 1

Compare the first element of the array with the key element 81.

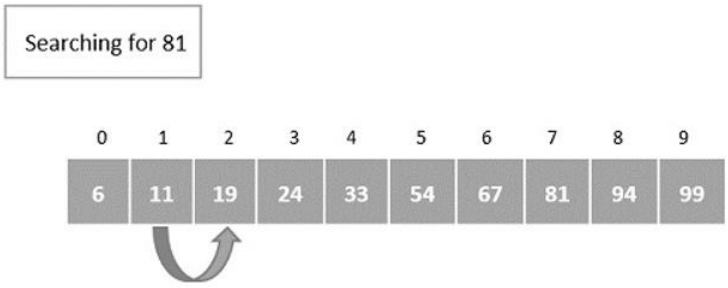
The first element of the array is 6, but the key element to be searched is 81; hence, the jump starts from the **1st** index as there is no match found.



Step 2

After initializing **i = 1**, the key element is compared with the element in the first index. Here, the element in the 1st index does not match with the key element. So it is again incremented exponentially in the powers of 2.

The index is incremented to **$2^m = 2^1$ = the element in 2^{nd}** index is compared with the key element.

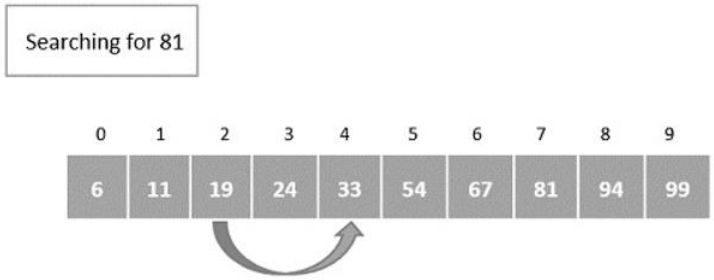


It is still not a match so it is once again incremented.

Step 3

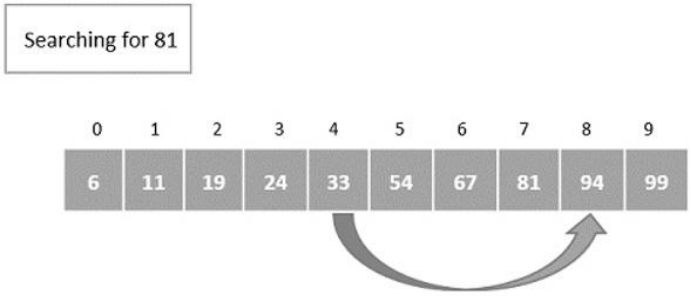
The index is incremented in the powers of 2 again.

$2^2 = 4$ = the element in 4^{th} index is compared with the key element and a match is not found yet.



Step 4

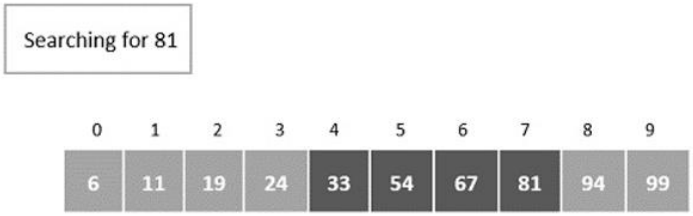
The index is incremented exponentially once again. This time the element in the **8^{th}** index is compared with the key element and a match is not found.



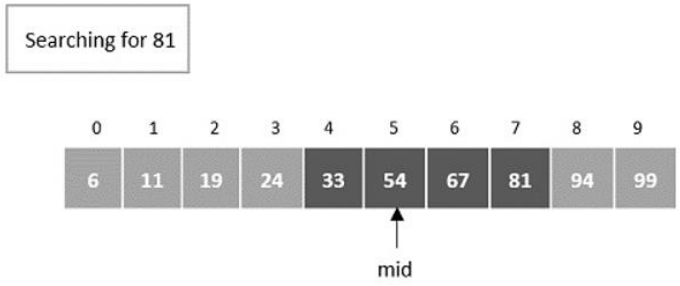
However, the element in the **8th** index is greater than the key element. Hence, the binary search algorithm is applied on the current block of elements.

Step 5

The current block of elements includes the elements in the indices [4, 5, 6, 7].



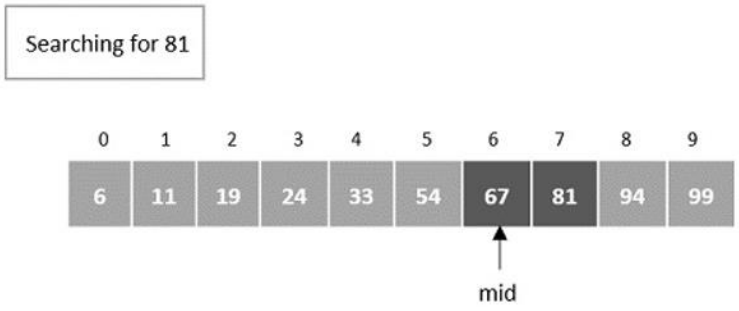
Small scale binary search is applied on this block of elements, where the **mid** is calculated to be the **5th** element.



Step 6

The match is not found at the mid element and figures that the desired element is greater than the mid element. Hence, the search takes place is the right half of the block.

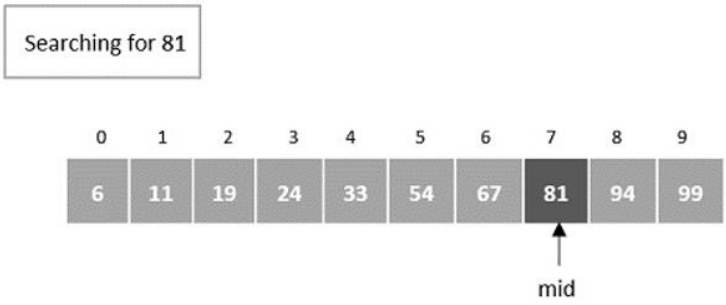
The mid now is set as **6th** element



Step 7

The element is still not found at the **6th** element so it now searches in the right half of the mid element.

The next mid is set as **7th** element.



Here, the element is found at the 7th index.

4.2.4.3 Implementation

In the implementation of the exponential search algorithm, the program checks for the matches at every exponential jump in the powers of 2. If the match is found the location of the element is returned otherwise the program returns an unsuccessful search.

Once the element at an exponential jump becomes greater than the key element, a binary search is performed on the current block of elements.

In this chapter, we will look into the implementation of exponential search in four different languages.

Python code

```
import math
def exponential_search(a, n, key):
    i = 1
```

```
   m = int(math.pow(2, i))
   if(a[0] == key):
      return 0
   while(a[m] <= key and m < n):
      i = i + 1
      m = int(math.pow(2, i))
      low = 0
      high = n - 1
      while (low <= high):
         mid = (low + high) // 2
         if(a[mid] == key):
            return mid
         elif(a[mid] < key):
            low = mid + 1
         else:
            high = mid - 1
   return -1
arr = [6, 11, 19, 24, 33, 54, 67, 81, 94, 99]
n = len(arr);
print("Array elements are: ")
for i in range(len(arr)):
   print(arr[i], end = " ")
key = 67
print("\nThe element to be searched: ", key)
index = exponential_search(arr, n, key)
if(index >= 0):
   print("The element is found at index: ", (index))
else:
   print("\nUnsuccessful Search")
```

output

```
Array elements are:
6 11 19 24 33 54 67 81 94 99
The element to be searched:  67
The element is found at index:  6
```

4.2.5 Hash Table

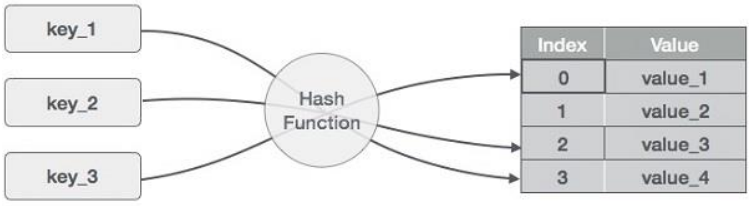
Hash Table is a data structure which stores data in an **associative** manner. In a hash table, data is stored in an **array** format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which **insertion** and **search** operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and

uses hash technique to generate an index where an element is to be inserted or is to be located from.

4.2.5.1 Hashing

Hashing is a technique to convert a **range of key** values into a **range of indexes** of an array. We're going to use **modulo** operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.



- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

Sr.No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

4.2.5.2 Linear Probing

As we can see, it may happen that the hashing technique is used to create an already **used index of the array**. In such a case, we can search the **next empty location** in the array by looking into the next cell until we find an empty cell. This technique is called **linear probing**.

Sr.No.	Key	Hash	Array Index	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2
3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

4.2.5.3 Basic Operations

Following are the basic primary operations of a hash table.

- **Search** − Searches an element in a hash table.
- **Insert** − Inserts an element in a hash table.
- **Delete** − Deletes an element from a hash table.

4.2.5.4 Data Item

Define a data item having some data and key, based on which the search is to be conducted in a hash table.

```
struct DataItem {
```

```
   int data;
   int key;
};
```

4.2.5.5 Hash Method

Define a hashing method to compute the hash code of the **key** of the data item.

```
int hashCode(int key){
   return key % SIZE;
}
```

4.2.5.6 Search Operation

Whenever an element is to be searched, compute the **hash code** of the **key** passed and locate the element **using that hash code as index** in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

```
struct DataItem *search(int key) {
   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty
   while(hashArray[hashIndex] != NULL) {

      if(hashArray[hashIndex]->key == key)
         return hashArray[hashIndex];

      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   return NULL;
}
```

Example

Following are the implementations of this operation in various programming language –

Python code

```python
SIZE = 10 # Define the size of the hash table
class DataItem:
    def __init__(self, key):
        self.key = key
hashMap = {} # Define the hash table as a dictionary
def hashCode(key):
    # Return a hash value based on the key
```

```
    return key % SIZE

def search(key):
    # get the hash
    hashIndex = hashCode(key)

    # move in map until an empty slot is found or the key is found
    while hashIndex in hashMap:
        # If the key is found, return the corresponding DataItem
        if hashMap[hashIndex].key == key:
            return hashMap[hashIndex]

        # go to the next cell
        hashIndex = (hashIndex + 1) % SIZE

    # If the key is not found, return None
    return None
# Initializing the hash table with some sample DataItems
item2 = DataItem(25) # Assuming the key is 25
item3 = DataItem(64) # Assuming the key is 64
item4 = DataItem(22) # Assuming the key is 22
# Calculate the hash index for each item and place them in the hash
table
hashIndex2 = hashCode(item2.key)
hashMap[hashIndex2] = item2

hashIndex3 = hashCode(item3.key)
hashMap[hashIndex3] = item3

hashIndex4 = hashCode(item4.key)
hashMap[hashIndex4] = item4

# Call the search function to test it
keyToSearch = 64 # The key to search for in the hash table
result = search(keyToSearch)
print("The element to be searched: ", keyToSearch)
if result:
    print("Element found")
else:
    print("Element not found")
```

Output

The element to be searched: 64
Element found

4.2.5.7 Insert Operation

Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

```
void insert(int key,int data) {
   struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
   item->data = data;
   item->key = key;

   //get the hash
   int hashIndex = hashCode(key);

   //move in array until an empty or deleted cell
   while(hashArray[hashIndex] != NULL && hashArray[hashIndex]->key != -1) {
      //go to next cell
      ++hashIndex;

      //wrap around the table
      hashIndex %= SIZE;
   }

   hashArray[hashIndex] = item;
}
```

Example

Following are the implementations of this operation in various programming languages −

Python code

```python
SIZE = 4  # Define the size of the hash table
class DataItem:
    def __init__(self, key):
        self.key = key
hashArray = [None] * SIZE  # Define the hash table as a list of
DataItem pointers
def hashCode(key):
    # Return a hash value based on the key
    return key % SIZE

def insert(key):
    # Create a new DataItem
    newItem = DataItem(key)
    # Initialize other data members if needed
    # Calculate the hash index for the key
    hashIndex = hashCode(key)
    # Handle collisions (linear probing)
```

```
    while hashArray[hashIndex] is not None:
        # Move to the next cell
        hashIndex += 1
        # Wrap around the table if needed
        hashIndex %= SIZE

    # Insert the new DataItem at the calculated index
    hashArray[hashIndex] = newItem
# Call the insert function with different keys to populate the hash
table
insert(42)  # Insert an item with key 42
insert(25)  # Insert an item with key 25
insert(64)  # Insert an item with key 64
insert(22)  # Insert an item with key 22
# Output the populated hash table
for i in range(SIZE):
    if hashArray[i] is not None:
        print(f"Index {i}: Key {hashArray[i].key}")
    else:
        print(f"Index {i}: Empty")
```

Output

Index 0: Key 64
Index 1: Key 25
Index 2: Key 42
Index 3: Key 22

4.2.5.8 Delete Operation

Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

```
struct DataItem* delete(struct DataItem* item) {
  int key = item->key;

  //get the hash
  int hashIndex = hashCode(key);

  //move in array until an empty
  while(hashArray[hashIndex] !=NULL) {

    if(hashArray[hashIndex]->key == key) {
      struct DataItem* temp = hashArray[hashIndex];

      //assign a dummy item at deleted position
```

```
      hashArray[hashIndex] = dummyItem;
      return temp;
    }
    //go to next cell
    ++hashIndex;

    //wrap around the table
    hashIndex %= SIZE;
  }
  return NULL;
}
```

Example

Following are the implementations of the deletion operation for Hash Table in various programming languages –

Python code

```python
SIZE = 5  # Define the size of the hash table

class DataItem:
    def __init__(self, key):
        self.key = key

def hashCode(key):
    # Implement your hash function here
    # Return a hash value based on the key
    return key % SIZE

def insert(key):
    global hashArray  # Access the global hashArray variable
    # Calculate the hash index for the key
    hashIndex = hashCode(key)

    # Handle collisions (linear probing)
    while hashArray[hashIndex] is not None:
        # Move to the next cell
        hashIndex = (hashIndex + 1) % SIZE
    # Insert the new DataItem at the calculated index
    hashArray[hashIndex] = DataItem(key)
    # Print the inserted item's key and hash index
    print(f"Inserted key {key} at index {hashIndex}")

def delete(key):
    global hashArray  # Access the global hashArray variable
    # Find the item in the hash table
    hashIndex = hashCode(key)
```

```
    while hashArray[hashIndex] is not None:
        if hashArray[hashIndex].key == key:
            # Mark the item as deleted (optional: free memory)
            hashArray[hashIndex] = None
            return
        # Move to the next cell
        hashIndex = (hashIndex + 1) % SIZE
    # If the key is not found, print a message
    print(f"Item with key {key} not found.")
# Initialize the hash table as a list of None values
hashArray = [None] * SIZE
print("Hash Table Contents before deletion:")
# Call the insert function with different keys to populate the hash
table
insert(1)  # Insert an item with key 1
insert(2)  # Insert an item with key 2
insert(3)  # Insert an item with key 3
insert(4)  # Insert an item with key 4
ele1 = 2
ele2 = 4
print("The keys to be deleted: ", ele1, " and ", ele2)
delete(2)  # Delete an item with key 2
delete(4)  # Delete an item with key 4

# Print the hash table's contents after delete operations
print("Hash Table Contents after deletion:")
for i in range(1, SIZE):
    if hashArray[i] is not None:
        print(f"Index {i}: Key {hashArray[i].key}")
    else:
        print(f"Index {i}: Empty")
```

Output

Hash Table Contents before deletion:
Inserted key 1 at index 1
Inserted key 2 at index 2
Inserted key 3 at index 3
Inserted key 4 at index 4
The keys to be deleted: 2 and 4
Hash Table Contents after deletion:
Index 1: Key 1
Index 2: Empty
Index 3: Key 3
Index 4: Empty

4.2.6 Exercises

4.2.7 Exercises on sorting algorithm

1. Implement bubble sort using C++ & Java code.?
2. Implement insertion sort using C++ & Java code.?
3. Implement selection sort using C++ & Java code.?
4. Implement shell sort using C++ & Java code.?

4.2.7.1 Exercises on Searching algorithm

1. Read about Hash Table?
2. Implement linear search using C++ & Java code.?
3. Implement binary search using C++ & Java code.?
4. Implement jump search using C++ & Java code.?
5. Implement exponential search using C++ & Java code.?

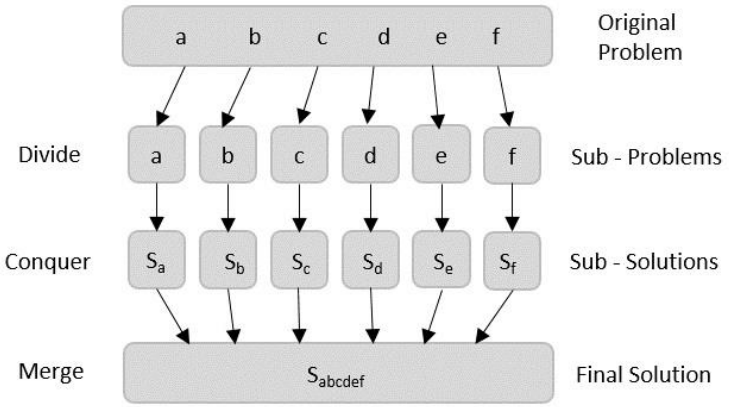
Chapter 5

5 Algorithm Design Techniques

5.1 Divide & Conquer Algorithm

Using divide and conquer approach, the problem in hand, is **divided into smaller sub-problems** and then each problem is solved **independently**.

- When we keep dividing the **sub-problems** into even **smaller sub-problems**, we may eventually reach a stage where **no more division** is possible.
- Those **smallest** possible sub-problems are solved using **original solution** because it takes lesser time to compute.
- The solution of all **sub-problems** is finally **merged** in order to obtain the solution of the original problem.



Broadly, we can understand **divide-and-conquer** approach in a three-step process.

5.1.1 Divide/Break

This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in size but still represent some part of the actual problem.

5.1.2 Conquer/Solve

This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

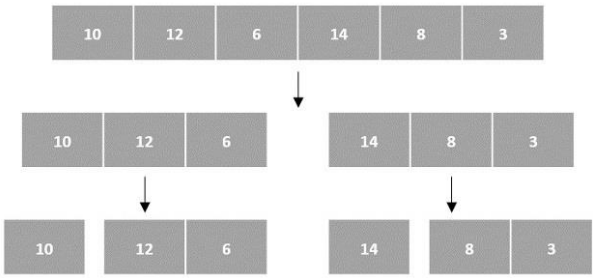
5.1.3 Merge/Combine

When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.

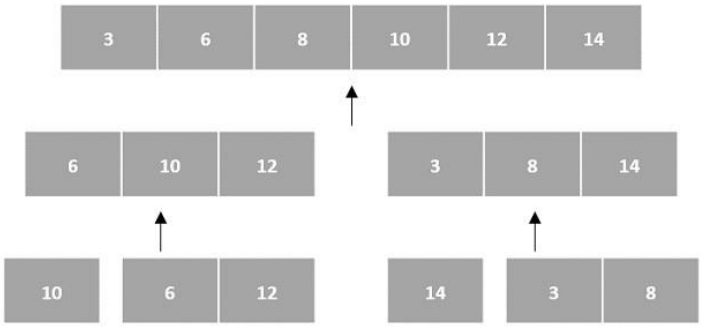
5.1.4 Arrays as Input

There are various ways in which various algorithms can take input such that they can be solved using the **divide and conquer** technique. Arrays are one of them. In algorithms that require input to be in the form of a list, like various sorting algorithms, array data structures are most commonly used.

In the input for a sorting algorithm below, the array input is **divided into subproblems** until they cannot be divided further.



Then, the subproblems are sorted (the conquer step) and are merged to form the solution of the original array back (the combine step).

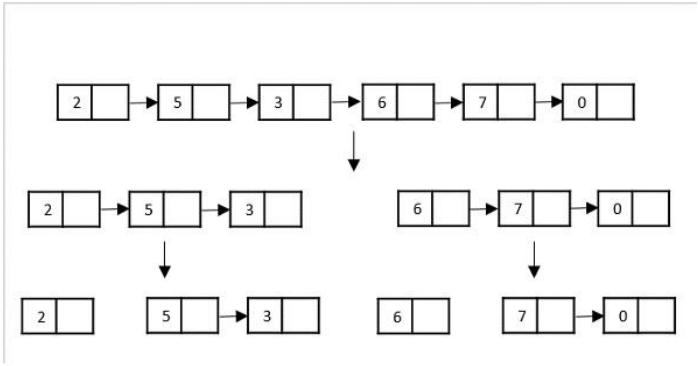


Since arrays are indexed and linear data structures, sorting algorithms most popularly use array data structures to receive input.

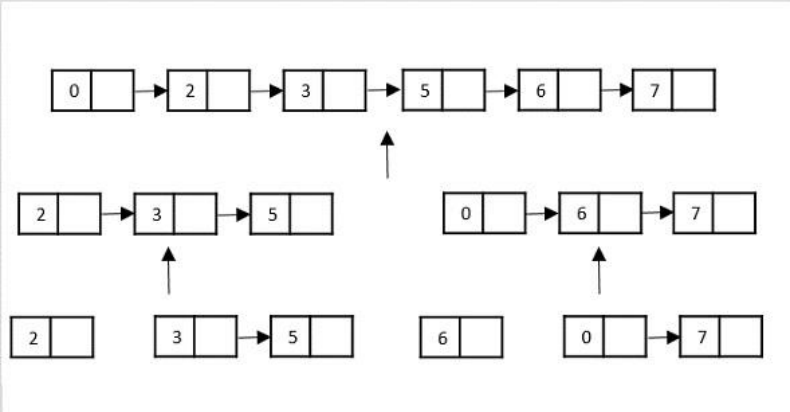
5.1.5 Linked Lists as Input

Another data structure that can be used to take input for divide and conquer algorithms is a linked list (for example, merge sort using linked lists). Like arrays, linked lists are also linear data structures that store data sequentially.

Consider the merge sort algorithm on linked list; following the very popular tortoise and hare algorithm, the list is divided until it cannot be divided further.



Then, the nodes in the list are sorted (conquered). These nodes are then combined (or merged) in recursively until the final solution is achieved.



Various searching algorithms can also be performed on the linked list data structures with a slightly different technique as linked lists are not indexed linear data structures. They must be handled using the **pointers available in the nodes** of the list.

5.1.6 Pros and cons of Divide and Conquer Approach

Divide and conquer approach supports parallelism as sub-problems are independent. Hence, an algorithm, which is designed using this technique, can run on the multiprocessor system or in different machines simultaneously.

In this approach, most of the algorithms are designed using recursion, hence memory management is very high. For recursive function stack is used, where function state needs to be stored.

5.1.7 Examples of Divide and Conquer Approach

The following computer algorithms are based on divide-and-conquer programming approach.

- **Merge Sort**
- **Quick Sort**
- **Binary Search**
- **Strassen's Matrix Multiplication**
- **Closest pair (points)**
- **Karatsuba**

5.1.8 Max-Min Problem

The Max-Min Problem in algorithm analysis is finding the maximum and minimum value in an array.

To find the maximum and minimum numbers in a given array ***numbers[]*** of size **n**, **divide and conquer approach** can be used.

5.1.8.1 Divide and Conquer Approach

In this approach, the array is divided into two halves. Then using recursive approach maximum and minimum numbers in each halves are found. Later, return the maximum of two maxima of each half and the minimum of two minima of each half.

In this given problem, the number of elements in an array is $y-x+1$, where **y** is greater than or equal to **x**.

Max−Min(x,y)

will return the maximum and minimum values of an array *numbers[x...y]*.

Algorithm: Max - Min(x, y)
if y – x ≤ 1 then
  return (max(numbers[x],numbers[y]),min((numbers[x],numbers[y]))
else
  (max1, min1):= maxmin(x, ⌊((x + y)/2)⌋)
  (max2, min2):= maxmin(⌊((x + y)/2) + 1)⌋,y)
return (max(max1, max2), min(min1, min2))

Example

Following are implementations of the above approach in various programming languages

Python code

```
def max_min_naive(arr):
    max_val = arr[0]
    min_val = arr[0]
    # Loop through the array to find the maximum and minimum values
    for i in range(1, len(arr)):
        if arr[i] > max_val:
            max_val = arr[i]  # Update the maximum value if a larger
element is found
        if arr[i] < min_val:
            min_val = arr[i]  # Update the minimum value if a smaller
element is found
    return max_val, min_val  # Return the pair of maximum and minimum
```

```
values
arr = [6, 4, 26, 14, 33, 64, 46]
max_val, min_val = max_min_naive(arr)
print("Maximum element is:", max_val)
print("Minimum element is:", min_val)
```

Output

Maximum element is: 64
Minimum element is: 4

5.1.8.2 Analysis

Let ***T(n)*** be the number of comparisons made by *Max−Min(x,y)* , where the number of elements $n=y-x+1$

If ***T(n)*** represents the numbers, then the recurrence relation can be represented as

$$T(n) = \begin{cases} T\left(\lfloor \frac{n}{2} \rfloor\right) + T\left(\lceil \frac{n}{2} \rceil\right) + 2 & for\ n > 2 \\ 1 & for\ n = 2 \\ 0 & for\ n = 1 \end{cases}$$

Let us assume that ***n*** is in the form of power of **2**. Hence, $\mathbf{n = 2^k}$ where **k** is height of the recursion tree. So,

$$T(n) = 2.T(\frac{n}{2}) + 2 = 2.\left(2.T(\tfrac{n}{4}) + 2\right) + 2.\ldots. = \frac{3n}{2} - 2$$

5.1.9 Merge Sort Algorithm

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

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

5.1.9.1 How Merge Sort Works?

To understand merge sort, we take an unsorted array as the following −

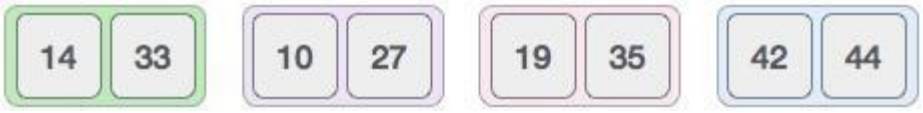
We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.

We further divide these arrays and we achieve atomic value which can no more be divided.

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.



In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

After the final merging, the list becomes sorted and is considered the final solution.

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is considered sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Step 1: If it is only one element in the list, consider it already sorted, so return.
Step 2: Divide the list recursively into two halves until it can no more be divided.
Step 3: Merge the smaller lists into new list in sorted order.

5.1.9.2 Pseudocode

We shall now see the pseudocodes for merge sort functions. As our algorithms point out two main functions − divide & merge.

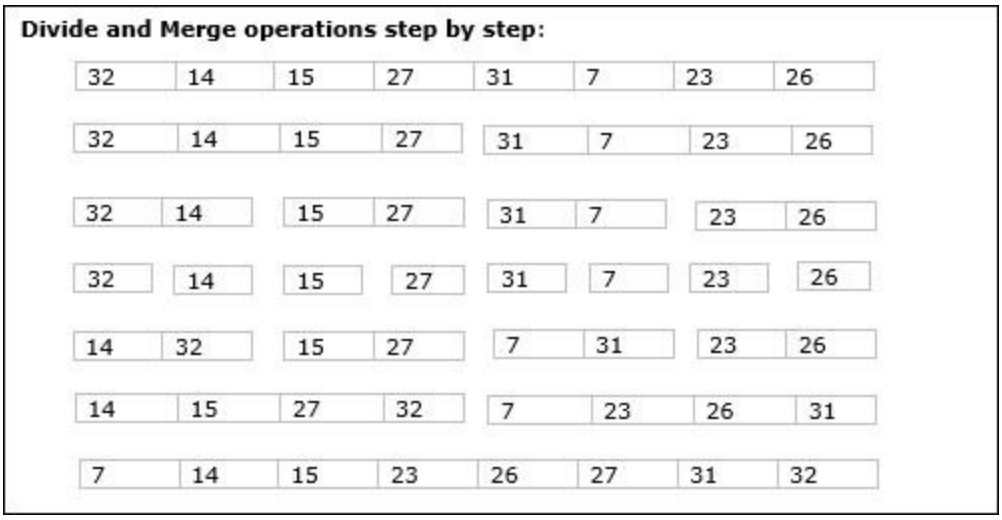
Merge sort works with recursion and we shall see our implementation in the same way.

```
procedure mergesort( var a as array )
   if ( n == 1 ) return a
      var l1 as array = a[0] ... a[n/2]
      var l2 as array = a[n/2+1] ... a[n]
      l1 = mergesort( l1 )
      l2 = mergesort( l2 )
      return merge( l1, l2 )
end procedure
procedure merge( var a as array, var b as array )
   var c as array
   while ( a and b have elements )
      if ( a[0] > b[0] )
         add b[0] to the end of c
         remove b[0] from b
      else
         add a[0] to the end of c
         remove a[0] from a
      end if
   end while
   while ( a has elements )
      add a[0] to the end of c
      remove a[0] from a
   end while
   while ( b has elements )
      add b[0] to the end of c
      remove b[0] from b
   end while
   return c
```

end procedure

Example

In the following example, we have shown Merge-Sort algorithm step by step. First, every iteration array is divided into two sub-arrays, until the sub-array contains only one element. When these sub-arrays cannot be divided further, then merge operations are performed.



5.1.9.3 Analysis

Let us consider, the running time of Merge-Sort as ***T(n)***. Hence,

$$\mathrm{T}(\mathrm{n}) = \begin{cases} c & if\, n \le 1 \\ 2\,xT\left(\frac{n}{2}\right) + dxn & otherwise \end{cases} \quad where\ c\ and\ d\ are\ constants$$

Therefore, using this recurrence relation,

$$T(n) = 2^i\, T\left(n/2^i\right) + i \cdot d \cdot n$$

$$As,\ i = \log n,\ T(n) = 2^{\log n} T\left(n/2^{\log n}\right) + \log n \cdot d \cdot n$$

$$= c \cdot n + d \cdot n \cdot \log n$$

$$Therefore,\ T(n) = O(n \log n).$$

Example

Following are the implementations of the above approach in various programming languages −

Python code

```
def merge_sort(a, n):
   if n > 1:
      m = n // 2
      #divide the list in two sub lists
      l1 = a[:m]
      n1 = len(l1)
      l2 = a[m:]
      n2 = len(l2)
      #recursively calling the function for sub lists
      merge_sort(l1, n1)
      merge_sort(l2, n2)
      i = j = k = 0
      while i < n1 and j < n2:
         if l1[i] <= l2[j]:
            a[k] = l1[i]
            i = i + 1
         else:
            a[k] = l2[j]
            j = j + 1
         k = k + 1
      while i < n1:
         a[k] = l1[i]
         i = i + 1
         k = k + 1
      while j < n2:
         a[k]=l2[j]
         j = j + 1
         k = k + 1

a = [10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0]
n = len(a)
print("Array before Sorting")
print(a)
merge_sort(a, n)
print("Array after Sorting")
print(a)
```

Output

Array before Sorting
[10, 14, 19, 26, 27, 31, 33, 35, 42, 44, 0]
Array after Sorting
[0, 10, 14, 19, 26, 27, 31, 33, 35, 42, 44]

5.2 Greedy Algorithms

The simplest and straightforward approach is the **Greedy** method. In this approach, the decision is taken on the basis of **current available** information without worrying about the effect of the **current decision** in future.

Greedy algorithms build a solution part by part, choosing the **next part** in such a way, that it gives an immediate benefit. This approach never reconsiders the choices taken previously. This approach is mainly used to solve optimization problems. Greedy method is easy to implement and quite efficient in most of the cases. Hence, we can say that Greedy algorithm is an algorithmic paradigm based on **heuristic** that follows local optimal choice at each step with the hope of finding global optimal solution.

In many problems, it does not produce an optimal solution though it gives an approximate (near optimal) solution in a reasonable time.

5.2.1 Components of Greedy Algorithm

Greedy algorithms have the following five components −

- **A candidate set** − A solution is created from this set.
- **A selection function** − Used to choose the best candidate to be added to the solution.
- **A feasibility function** − Used to determine whether a candidate can be used to contribute to the solution.
- **An objective function** − Used to assign a value to a solution or a partial solution.
- **A solution function** − Used to indicate whether a complete solution has been reached.

5.2.2 Areas of Application

Greedy approach is used to solve many problems, such as

- Finding the shortest path between two vertices using **Dijkstra'**s algorithm.
- Finding the **minimal spanning tree** in a graph using Prim's /Kruskal's algorithm, etc.

5.2.2.1 Counting Coins Problem

The Counting Coins problem is to count to a desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of 1, 2, 5 and 10 and we are asked to count 18 then the greedy procedure will be −

- 1 − Select one 10 coins, the remaining count is 8
- 2 − Then select one 5 coin, the remaining count is 3
- 3 − Then select one 2 coin, the remaining count is 1
- 4 − And finally, the selection of one 1 coins solves the problem

Though, it seems to be working fine, for this count we need to pick only 4 coins. But if we slightly change the problem then the same approach may not be able to produce the same optimum result.

For the currency system, where we have coins of 1, 7, 10 value, counting coins for value 18 will be absolutely optimum but for count like 15, it may use more coins than necessary. For example, the greedy approach will use 10 + 1 + 1 + 1 + 1 + 1, total 6 coins. Whereas the same problem could be solved by using only 3 coins (7 + 7 + 1)

Hence, we may conclude that the greedy approach picks an immediate optimized solution and may fail where global optimization is a major concern.

5.2.2.2 Where Greedy Approach Fails

In many problems, Greedy algorithm **fails** to **find an optimal solution**, moreover it may produce a worst solution. Problems like **Travelling Salesman and Knapsack** cannot be solved using this approach.

Examples

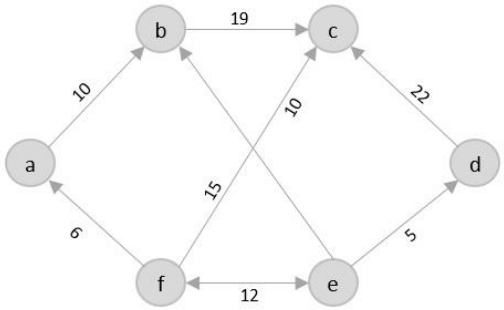
Most networking algorithms use the greedy approach. Here is a list of few of them −

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

5.2.3 Travelling Salesman Problem

The travelling salesman problem is a **graph computational problem** where the salesman needs to visit all **cities** (represented using nodes in a graph) in a list just **once** and the **distances** (represented using edges in the graph) between all these **cities** are known. The solution that is needed to be found for this problem is the **shortest possible route** in which the salesman visits all the cities and returns to the origin city.

If you look at the graph below, considering that the salesman starts from the **vertex 'a'**, they need to travel through all the remaining vertices **b, c, d, e, f** and get back to '**a'** while making sure that the cost taken is minimum.



There are various approaches to find the solution to the **travelling salesman problem**: **naïve approach**, **greedy approach**, **dynamic programming approach**, etc. In this tutorial we will be learning about solving travelling salesman problem using **greedy** approach.

5.2.4 Travelling Salesperson Algorithm

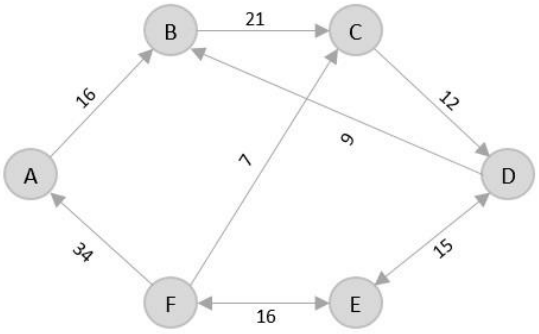
As the definition for greedy approach states, we need to find the **best optimal solution** locally to **figure out** the global optimal solution. The inputs taken by the algorithm are the graph **G {V, E},** where **V** is the set of vertices and **E** is the set of edges. The shortest path of graph **G** starting from **one vertex** returning to the same vertex is obtained as the **output**.

5.2.4.1 Algorithm

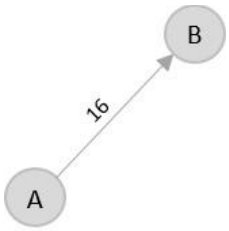
- Travelling salesman problem takes a graph G {V, E} as an input and declare another graph as the output (say G') which will record the path the salesman is going to take from one node to another.
- The algorithm begins by sorting all the edges in the input graph G from the least distance to the largest distance.
- The first edge selected is the edge with least distance, and one of the two vertices (say A and B) being the origin node (say A).
- Then among the adjacent edges of the node other than the origin node (B), find the least cost edge and add it onto the output graph.
- Continue the process with further nodes making sure there are no cycles in the output graph and the path reaches back to the origin node A.
- However, if the origin is mentioned in the given problem, then the solution must always start from that node only. Let us look at some example problems to understand this better.

Examples

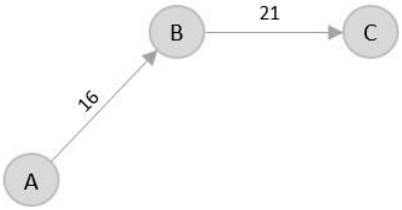
Consider the following graph with six cities and the distances between them −



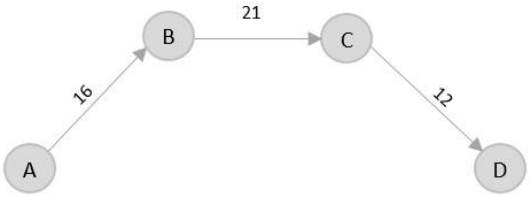
From the given graph, since the origin is already mentioned, the solution must always start from that node. Among the edges leading from A, A → B has the shortest distance.



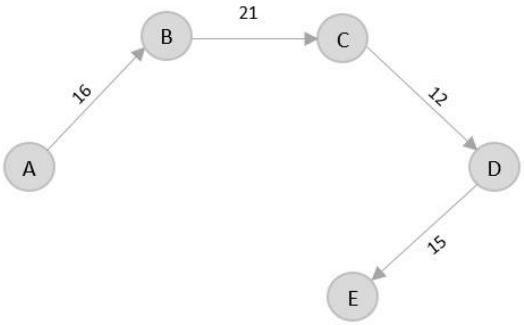
Then, B → C has the shortest and only edge between, therefore it is included in the output graph.



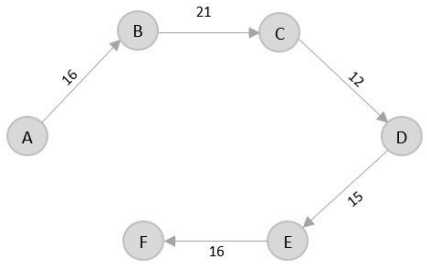
There's only one edge between C → D, therefore it is added to the output graph.



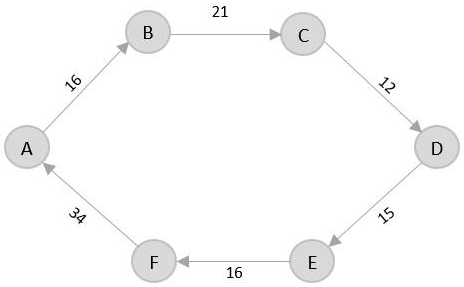
There's two outward edges from **D.** Even though, **D → B** has lower distance than D → E, **B is already visited once** and it would form a cycle if added to the output graph. Therefore**, D → E** is added into the output graph.



There's only one edge from e, that is E → F. Therefore, it is added into the output graph.



Again, even though F → C has lower distance than F → A, F → A is added into the output graph in order to avoid the cycle that would form and C is already visited once.



The shortest path that originates and ends at A is A → B → C → D → E → F → A

The cost of the path is: **16 + 21 + 12 + 15 + 16 + 34 = 114**.

Even though, the cost of path could be decreased if it originates from other nodes but the question is not raised with respect to that.

Example

The complete implementation of Travelling Salesman Problem using Greedy Approach is given below –

Python code

```
import numpy as np
def travellingsalesman(c):
    global cost
    adj_vertex = 999
    min_val = 999
    visited[c] = 1
    print((c + 1), end=" ")
    for k in range(n):
        if (tsp_g[c][k] != 0) and (visited[k] == 0):
```

```
            if tsp_g[c][k] < min_val:
                min_val = tsp_g[c][k]
                adj_vertex = k
    if min_val != 999:
        cost = cost + min_val
    if adj_vertex == 999:
        adj_vertex = 0
        print((adj_vertex + 1), end=" ")
        cost = cost + tsp_g[c][adj_vertex]
        return
    travellingsalesman(adj_vertex)
n = 5
cost = 0
visited = np.zeros(n, dtype=int)
tsp_g = np.array([[12, 30, 33, 10, 45],
                  [56, 22, 9, 15, 18],
                  [29, 13, 8, 5, 12],
                  [33, 28, 16, 10, 3],
                  [1, 4, 30, 24, 20]])
print("Shortest Path:", end=" ")
travellingsalesman(0)
print()
print("Minimum Cost:", end=" ")
print(cost)
```

Output

Shortest Path: 1 4 5 2 3 1
Minimum Cost: 55

5.2.5 Kruskal's Minimal Spanning Tree

Kruskal's minimal spanning tree algorithm is one of the efficient methods to find the minimum spanning tree of a graph. A minimum spanning tree is a **subgraph** that connects all the vertices present in the **main graph** with the **least possible edges** and minimum cost (sum of the weights assigned to each edge).

The algorithm first starts from the **forest** – which is defined as a **subgraph** containing only **vertices of the main graph** – of the graph, adding the **least cost edges** later until the minimum spanning tree is created without forming cycles in the graph.

Kruskal's algorithm has easier implementation than prim's algorithm, but has higher complexity.

5.2.5.1 Kruskal's Algorithm

The inputs taken by the **kruskal's** algorithm are the graph **G {V, E},** where **V** is the set of vertices and **E** is the set of edges, and the source vertex **S** and the minimum spanning tree of graph G is obtained as an output.

5.2.5.2 Algorithm

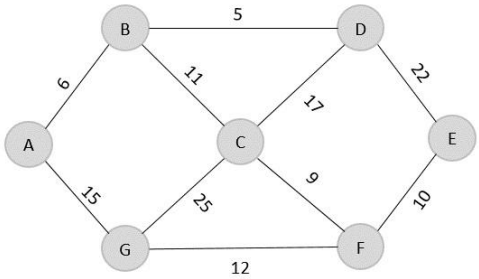
- Sort all the edges in the graph in an ascending order and store it in an array edge[].

Edge								
Cost								

- Construct the forest of the graph on a plane with all the vertices in it.
- Select the least cost edge from the edge[] array and add it into the forest of the graph. Mark the vertices visited by adding them into the visited[] array.
- Repeat the steps 2 and 3 until all the vertices are visited without having any cycles forming in the graph
- When all the vertices are visited, the minimum spanning tree is formed.
- Calculate the minimum cost of the output spanning tree formed.

Examples

Construct a minimum spanning tree using kruskal's algorithm for the graph given below

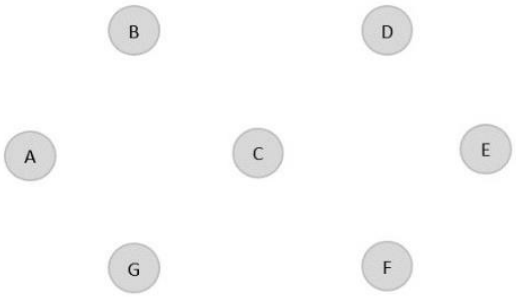


Solution

As the first step, sort all the edges in the given graph in an ascending order and store the values in an array.

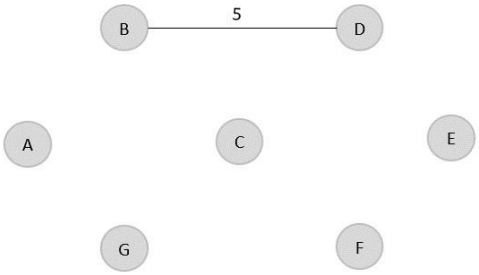
Edge	B→D	A→B	C→F	F→E	B→C	G→F	A→G	C→D	D→E	C→G
Cost	5	6	9	10	11	12	15	17	22	25

Then, construct a forest of the given graph on a single plane.



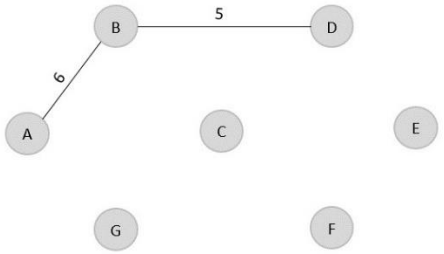
From the list of sorted edge costs, select the least cost edge and add it onto the forest in output graph.

B → D = 5
Minimum cost = 5
Visited array, v = {B, D}



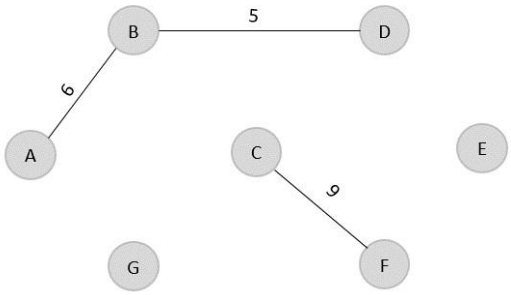
Similarly, the next least cost edge is B → A = 6; so we add it onto the output graph.

Minimum cost = 5 + 6 = 11
Visited array, v = {B, D, A}



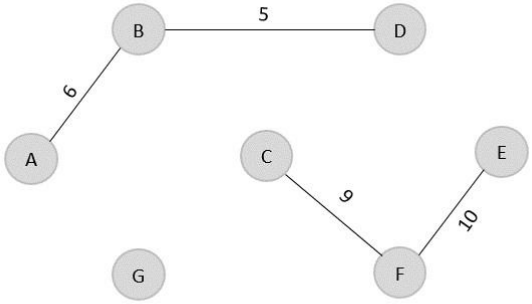
The next least cost edge is C → F = 9; add it onto the output graph.

Minimum Cost = 5 + 6 + 9 = 20
Visited array, v = {B, D, A, C, F}



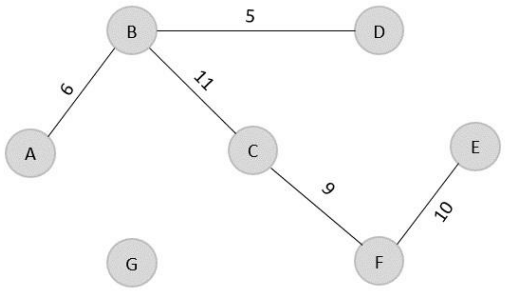
The next edge to be added onto the output graph is F → E = 10.

Minimum Cost = 5 + 6 + 9 + 10 = 30
Visited array, v = {B, D, A, C, F, E}



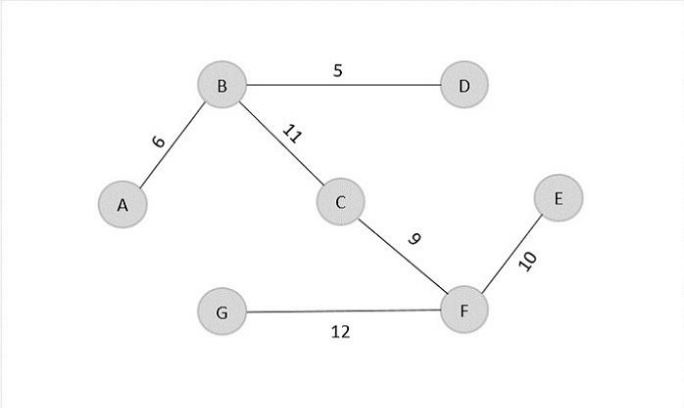
The next edge from the least cost array is B → C = 11, hence we add it in the output graph.

Minimum cost = 5 + 6 + 9 + 10 + 11 = 41
Visited array, v = {B, D, A, C, F, E}



The last edge from the least cost array to be added in the output graph is F → G = 12.

Minimum cost = 5 + 6 + 9 + 10 + 11 + 12 = 53
Visited array, v = {B, D, A, C, F, E, G}



The obtained result is the minimum spanning tree of the given graph with cost = 53.

Example

The final program implements the Kruskal's minimum spanning tree problem that takes the cost adjacency matrix as the input and prints the shortest path as the output along with the minimum cost.

Python code

```
inf = 999999
k, a, b, u, v, n, ne = 0, 0, 0, 0, 0, 0, 1
mincost = 0
cost = [[0, 10, 20], [12, 0, 15], [16, 18, 0]]
p = [0] * 9
def applyfind(i):
    while p[i] != 0:
        i = p[i]
    return i
```

```
def applyunion(i, j):
    if i != j:
        p[j] = i
        return 1
    return 0
n = 3
for i in range(n):
    for j in range(n):
        if cost[i][j] == 0:
            cost[i][j] = inf
print("Minimum Cost Spanning Tree:")
while ne < n:
    min_val = inf
    for i in range(n):
        for j in range(n):
            if cost[i][j] < min_val:
                min_val = cost[i][j]
                a = u = i
                b = v = j
    u = applyfind(u)
    v = applyfind(v)
    if applyunion(u, v) != 0:
        print(f"{a} -> {b}")
        mincost += min_val
    cost[a][b] = cost[b][a] = 999
    ne += 1
print(f"Minimum cost = {mincost}")
```

Output

```
Minimum Cost Spanning Tree:
0 -> 1
1 -> 2
Minimum cost = 25
```

5.2.6 Dijkstra's Shortest Path Algorithm

Dijkstra's shortest path algorithm is similar to that of Prim's algorithm as they both rely on **finding the shortest path locally** to achieve the global solution. However, unlike prim's algorithm, the dijkstra's algorithm does not **find the minimum spanning tree**; it is designed to find the **shortest path in the graph from one vertex to other remaining vertices** in the graph. Dijkstra's algorithm can be performed on both directed and undirected graphs.

Since the shortest path can be calculated from **single source vertex to all the other vertices** in the graph, Dijkstra's algorithm is also called **single-source shortest path algorithm**. The output obtained is called **shortest path spanning tree**.

In this chapter, we will learn about the greedy approach of the dijkstra's algorithm.

5.2.7 Dijkstra's Algorithm

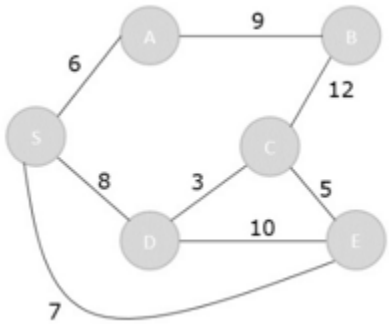
The dijkstra's algorithm is designed to find the shortest path between **two vertices** of a graph. These two vertices could **either be adjacent or the farthest points** in the graph. The algorithm starts from the source. The **inputs** taken by the algorithm are the graph **G {V, E}**, where V is the set of vertices and E is the set of edges, and the source vertex S. And the output is the shortest path spanning tree.

5.2.7.1 Algorithm

- Declare two arrays – distance[] to store the distances from the source vertex to the other vertices in graph and visited[] to store the visited vertices.
- Set distance[S] to '0' and distance[v] = ∞, where v represents all the other vertices in the graph.
- Add S to the visited[] array and find the adjacent vertices of S with the minimum distance.
- The adjacent vertex to S, say A, has the minimum distance and is not in the visited array yet. A is picked and added to the visited array and the distance of A is changed from ∞ to the assigned distance of A, say d_1, where $d_1 < \infty$.
- Repeat the process for the adjacent vertices of the visited vertices until the shortest path spanning tree is formed.

Examples

To understand the dijkstra's concept better, let us analyze the algorithm with the help of an example graph −



Step 1

Initialize the distances of all the vertices as ∞, except the source node S.

Vertex	S	A	B	C	D	E
Distance	0	∞	∞	∞	∞	∞

Now that the source vertex S is visited, add it into the visited array.

visited = {S}

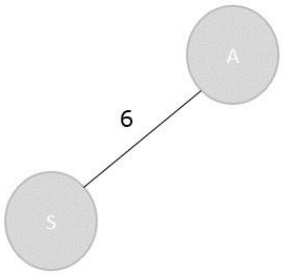
Step 2

The vertex S has three adjacent vertices with various distances and the vertex with minimum distance among them all is A. Hence, A is visited and the dist[A] is changed from ∞ to 6.

S → A = 6
S → D = 8
S → E = 7

Vertex	S	A	B	C	D	E
Distance	0	6	∞	∞	8	7

Visited = {S, A}



Step 3

There are two vertices visited in the visited array, therefore, the adjacent vertices must be checked for both the visited vertices.

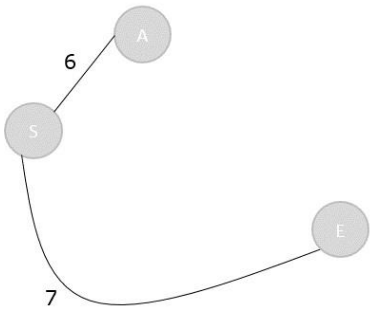
Vertex S has two more adjacent vertices to be visited yet: D and E. Vertex A has one adjacent vertex B.

Calculate the distances from S to D, E, B and select the minimum distance −

S → D = 8 and S → E = 7.
S → B = S → A + A → B = 6 + 9 = 15

Vertex	S	A	B	C	D	E
Distance	0	6	15	∞	8	7

Visited = {S, A, E}



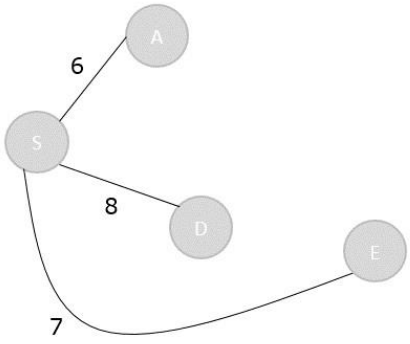
Step 4

Calculate the distances of the adjacent vertices – S, A, E – of all the visited arrays and select the vertex with minimum distance.

S → D = 8
S → B = 15
S → C = S → E + E → C = 7 + 5 = 12

Vertex	S	A	B	C	D	E
Distance	0	6	15	12	8	7

Visited = {S, A, E, D}



Step 5

Recalculate the distances of unvisited vertices and if the distances minimum than existing distance is found, replace the value in the distance array.

S → C = S → E + E → C = 7 + 5 = 12
S → C = S → D + D → C = 8 + 3 = 11

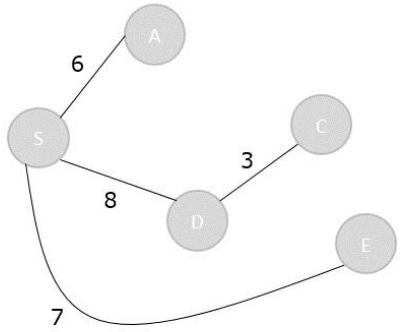
dist[C] = minimum (12, 11) = 11

S → B = S → A + A → B = 6 + 9 = 15
S → B = S → D + D → C + C → B = 8 + 3 + 12 = 23

dist[B] = minimum (15,23) = 15

Vertex	S	A	B	C	D	E
Distance	0	6	15	11	8	7

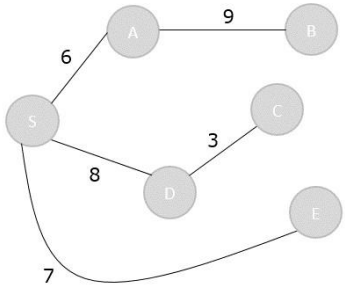
Visited = { S, A, E, D, C}



Step 6

The remaining unvisited vertex in the graph is B with the minimum distance 15, is added to the output spanning tree.

Visited = {S, A, E, D, C, B}



The shortest path spanning tree is obtained as an output using the dijkstra's algorithm.

Example

The program implements the dijkstra's shortest path problem that takes the cost adjacency matrix as the input and prints the shortest path as the output along with the minimum cost.

Python code implementation

```
import sys
def min_dist(dist, visited):  # finding minimum dist
    minimum = sys.maxsize
    ind = -1
    for k in range(6):
        if not visited[k] and dist[k] <= minimum:
            minimum = dist[k]
            ind = k
    return ind
def greedy_dijkstra(graph, src):
    dist = [sys.maxsize] * 6
    visited = [False] * 6
    dist[src] = 0  # Source vertex dist is set 0
    for _ in range(6):
        m = min_dist(dist, visited)
        visited[m] = True
        for k in range(6):
            #  updating the dist of neighbouring vertex
            if not visited[k] and graph[m][k] and dist[m] !=
sys.maxsize and dist[m] + graph[m][k] < dist[k]:
                dist[k] = dist[m] + graph[m][k]
    print("Vertex\t\tdist from source vertex")
    for k in range(6):
        str_val = chr(65 + k)  # Convert index to corresponding
character
```

```
        print(str_val, "\t\t\t", dist[k])
# Main code
graph = [
    [0, 1, 2, 0, 0, 0],
    [1, 0, 0, 5, 1, 0],
    [2, 0, 0, 2, 3, 0],
    [0, 5, 2, 0, 2, 2],
    [0, 1, 3, 2, 0, 1],
    [0, 0, 0, 2, 1, 0]
]
greedy_dijkstra(graph, 0)
```

output

Vertex	dist from source vertex
A	0
B	1
C	2
D	4
E	2
F	3

5.2.8 Map Coloring Algorithm

Map colouring problem states that given a **graph G {V, E}** where V and E are the set of **vertices and edges** of the graph, all vertices in V need to be coloured in such a way that no two adjacent vertices must have the same colour.

The real-world applications of this algorithm are – assigning **mobile radio frequencies**, making schedules, designing Sudoku, allocating registers etc.

With the map colouring algorithm, a graph **G** and the colours to be added to the graph are taken as an **input** and a coloured graph with no two adjacent vertices having the same colour is achieved.

5.2.8.1 Algorithm

- Initiate all the vertices in the graph.
- Select the node with the highest degree to colour it with any colour.
- Choose the colour to be used on the graph with the help of the selection colour function so that no adjacent vertex is having the same colour.
- Check if the colour can be added and if it does, add it to the solution set.
- Repeat the process from step 2 until the output set is ready.

Examples

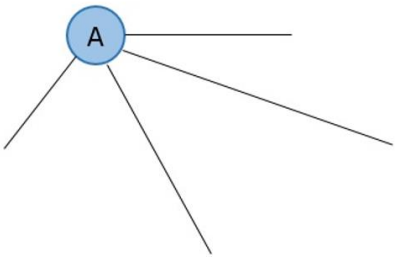
Step 1

Find degrees of all the vertices −

A – 4
B – 2
C – 2
D – 3
E – 3

Step 2

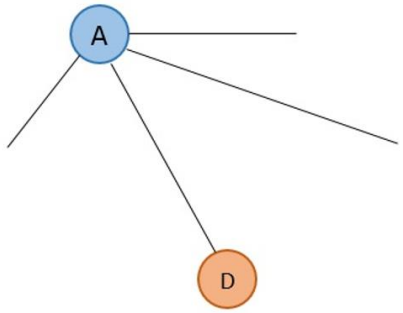
Choose the **vertex with the highest degree** to colour first, i.e., **A** and choose a colour using selection colour function. Check if the colour can be added to the vertex and if yes, add it to the solution set.



Step 3

Select any vertex with the next highest degree from the remaining vertices and colour it using selection colour function.

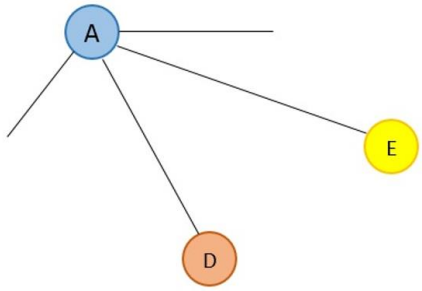
D and E both have the next highest degree 3, so choose any one between them, say D.



D is adjacent to A, therefore it cannot be coloured in the same colour as A. Hence, choose a different colour using selection colour function.

Step 4

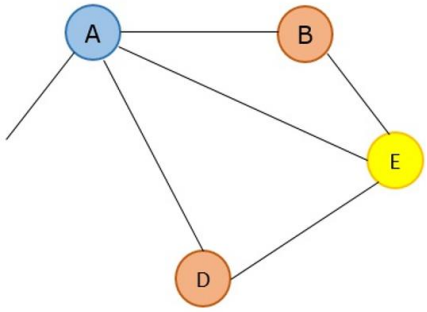
The next highest degree vertex is E, hence choose E.



E is adjacent to both A and D, therefore it cannot be coloured in the same colours as A and D. Choose a different colour using selection colour function.

Step 5

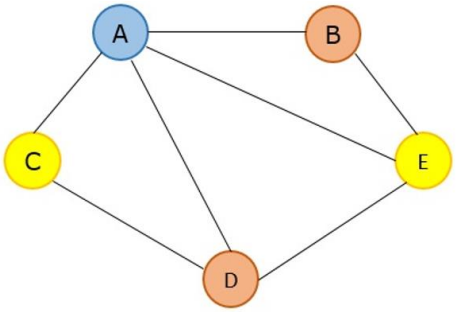
The next highest degree vertices are B and C. Thus, choose any one randomly.



B is adjacent to both A and E, thus not allowing to be coloured in the colours of A and E but it is not adjacent to D, so it can be coloured with D's colour.

Step 6

The next and the last vertex remaining is C, which is adjacent to both A and D, not allowing it to be coloured using the colours of A and D. But it is not adjacent to E, so it can be coloured in E's colour.



Example

Following is the complete implementation of Map Colouring Algorithm in various programming languages where a graph is coloured in such a way that no two adjacent vertices have same colour.

Python code implementation of graph coloring

```
V = 4
graph = [[0, 1, 1, 0], [1, 0, 1, 1], [1, 1, 0, 1], [0, 1, 1, 0]]
def isValid(v, color, c):  # check whether putting a color valid for v
    for i in range(V):
        if graph[v][i] and c == color[i]:
            return False
    return True
def mColoring(colors, color, vertex):
    if vertex == V:  # when all vertices are considered
        return True
    for col in range(1, colors + 1):
        if isValid(vertex, color,
                   col):  # check whether color col is valid or not
            color[vertex] = col
            if mColoring(colors, color, vertex + 1):
                return True  # go for additional vertices
            color[vertex] = 0
    return False  # when no colors can be assigned
```

```
colors = 3  # Number of colors
color = [0] * V  # make color matrix for each vertex
if not mColoring(
        colors, color,
        0):  # initially set to 0 and for Vertex 0 check graph
coloring
    print("Solution does not exist.")
else:
    print("Assigned Colors are:")
    for i in range(V):
        print(color[i], end=" ")
```

Output

Assigned Colors are:
1 2 3 1

5.3 Dynamic Programming

Dynamic programming approach is similar to **divide and conquer** in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike divide and conquer, these **sub-problems are not solved independently**. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

Mostly, dynamic programming algorithms are used for **solving optimization problems**. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best optimal final solution. This paradigm is thus said to be using Bottom-up approach.

So we can conclude that −

- The problem should be able to be divided into smaller overlapping sub-problem.
- Final optimum solution can be achieved by using an **optimum solution of smaller sub-problems.**
- Dynamic algorithms use memorization.

However, in a problem, two main properties can suggest that the given problem can be solved using Dynamic Programming. They are

5.3.1 Overlapping Sub-Problems

Similar to Divide-and-Conquer approach, Dynamic Programming also **combines solutions to sub-problems**. It is mainly used where the solution of one sub-problem is needed repeatedly. The computed solutions are stored in a table, so that these don't have

to be re-computed. Hence, this technique is needed where overlapping sub-problem exists.

For example, Binary Search does not have overlapping sub-problem. Whereas recursive program of **Fibonacci** numbers have many overlapping sub-problems.

5.3.2 Optimal Sub-Structure

A given problem has Optimal Substructure Property, if the optimal solution of the given problem can be obtained using optimal solutions of its **sub-problems**.

For example, the **Shortest Path problem** has the following optimal substructure property

If a **node x** lies in the shortest path from a source node **u** to destination node **v**, then the shortest path from **u** to **v** is the combination of the shortest path from **u** to **x**, and the shortest path from **x** to **v**.

The standard All Pair Shortest Path algorithms like **Floyd-Warshall** and **Bellman-Ford** are typical examples of Dynamic Programming.

5.3.3 Steps of Dynamic Programming Approach

Dynamic Programming algorithm is designed using the following four steps

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from the computed information.

Dynamic algorithms use the output of a **smaller sub-problem** and then try to optimize a bigger sub-problem. Dynamic algorithms use **memorization** to remember the output of **already solved sub-problems**.

Examples

The following computer problems can be solved using dynamic programming approach

- **Fibonacci number series**
- **Knapsack problem**
- **Tower of Hanoi**

- **All pair shortest path by Floyd-Warshall and Bellman Ford**
- **Shortest path by Dijkstra**
- **Project scheduling**
- **Matrix Chain Multiplication**

Dynamic programming can be used in both top-down and bottom-up manner. And of course, most of the times, referring to the previous solution output is cheaper than re-computing in terms of CPU cycles.

5.3.4 Matrix Chain Multiplication

Matrix Chain Multiplication is an algorithm that is applied to determine the **lowest cost way** for **multiplying matrices**. The actual multiplication is done using the standard way of multiplying the matrices, i.e., it follows the basic rule that the number of rows in one matrix must be equal to the number of columns in another matrix. Hence, multiple scalar multiplications must be done to achieve the product.

To brief it further, consider matrices **A, B, C**, and **D**, to be multiplied; hence, the multiplication is done using the standard matrix multiplication. There are multiple combinations of the matrices found while using the standard approach since matrix multiplication is associative. For instance, there are five ways to multiply the **four matrices** given above −

- (A(B(CD)))
- (A((BC)D))
- ((AB)(CD))
- ((A(BC))D)
- (((AB)C)D)

Now, if the size of matrices A, B, C, and D are **l × m, m × n, n × p, p × q** respectively, then the number of scalar multiplications performed will be ***lmnpq***. But the cost of the matrices change based on the rows and columns present in it. Suppose, the values of **l, m, n, p, q** are 5, 10, 15, 20, 25 respectively, the cost of (A(B(CD))) is 5 × 100 × 25 = 12,500; however, the cost of (A((BC)D)) is 10 × 25 × 37 = 9,250.

So, dynamic programming approach of the matrix chain multiplication is adopted in order to find the combination with the lowest cost.

5.3.5 Matrix Chain Multiplication Algorithm

Matrix chain multiplication algorithm is only applied to find the minimum cost way to **multiply a sequence of matrices**. Therefore, the **input** taken by the algorithm is the **sequence of matrices** while the output achieved is the **lowest cost parenthesization**.

5.3.5.1 Algorithm

Count the number of parenthesizations. Find the number of ways in which the input matrices can be multiplied using the formulae –

$$P(n) = \begin{cases} 1 & if\ n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & if\ n \geq 2 \end{cases}$$

(or)

$$P(n) = \begin{cases} \frac{2(n-1)C_{n-1}}{n} & if\ n \geq 2 \\ 1 & if\ n = 1 \end{cases}$$

 Once the parenthesization is done, the optimal substructure must be devised as the first step of dynamic programming approach so the final product achieved is optimal. In matrix chain multiplication, the optimal substructure is found by dividing the sequence of matrices ***A[i….j]*** into two parts ***A[i,k] and A[k+1,j]***. It must be ensured that the parts are divided in such a way that optimal solution is achieved.

 Using the formula

$$C[i,j] = \begin{cases} 0 & if\ i = j \\ \min\limits_{i \leq k < j} \{ C[i,k] + C[k+1,j] + d_{i-1}d_k d_j & if\ i < j \end{cases}$$

find the lowest cost parenthesization of the sequence of matrices by constructing cost tables and corresponding ***k*** values table.

 Once the lowest cost is found, print the corresponding parenthesization as the output.

5.3.5.2 Pseudocode

Pseudocode to find the lowest cost of all the possible parenthesizations −

```
MATRIX-CHAIN-MULTIPLICATION(p)
  n = p.length ─ 1
  let m[1…n, 1…n] and s[1…n ─ 1, 2…n] be new matrices
  for i = 1 to n
    m[i, i] = 0
  for l = 2 to n // l is the chain length
    for i = 1 to n - l + 1
      j = i + l - 1
      m[i, j] = ∞
      for k = i to j - 1
        q = m[i, k] + m[k + 1, j] + pi-1pkpj
        if q < m[i, j]
          m[i, j] = q
```

```
            s[i, j] = k
return m and s
```

Pseudocode to print the optimal output parenthesization −

```
PRINT-OPTIMAL-OUTPUT(s, i, j )
if i == j
print “A”i
else print “(”
PRINT-OPTIMAL-OUTPUT(s, i, s[i, j])
PRINT-OPTIMAL-OUTPUT(s, s[i, j] + 1, j)
print “)”
```

Example

The application of dynamic programming formula is slightly different from the theory; to understand it better let us look at few examples below.

A sequence of matrices A, B, C, D with dimensions 5 × 10, 10 × 15, 15 × 20, 20 × 25 are set to be multiplied. Find the lowest cost parenthesization to multiply the given matrices using matrix chain multiplication.

Solution

Given matrices and their corresponding dimensions are −

A5×10×B10×15×C15×20×D20×25

Find the count of parenthesization of the 4 matrices, i.e. n = 4.

Using the formula,

$$P(n) = \begin{cases} 1 & if\ n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & if\ n \geq 2 \end{cases}$$

Since n = 4 ≥ 2, apply the second case of the formula –

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)$$

$$P(4) = \sum_{k=1}^{3} P(k)P(4-k)$$

$$P(4) = P(1)P(3) + P(2)P(2) + P(3)P(1)$$

If P(1) = 1 and P(2) is also equal to 1, P(4) will be calculated based on the P(3) value. Therefore, P(3) needs to determined first.

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

Therefore,

$$P(4) = P(1)P(3) + P(2)P(2) + P(3)P(1)$$
$$= 2 + 1 + 2 = 5$$

Among these 5 combinations of parenthesis, the matrix chain multiplication algorithm must find the lowest cost parenthesis.

Step 1

The table above is known as a **cost table**, where all the cost values calculated from the different combinations of parenthesis are stored.

cost table	1	2	3	4
1				
2				
3				
4				

Another table is also created to store the **k** values obtained at the minimum cost of each combination

k	1	2	3	4
1				
2				
3				
4				

Step 2

Applying the dynamic programming approach formula find the costs of various parenthesizations,

$$C[i,j] = \begin{cases} 0 & if\ i = j \\ \min\limits_{i \le k < j} \{ C[i,k] + C[k+1,j] + d_{i-1} d_k d_j & if\ i < j \end{cases}$$

$C[1,1] = 0$

$C[2,2] = 0$

$C[3,3] = 0$

$C[4,4] = 0$

cost table	1	2	3	4
1	0			
2		0		
3			0	
4				0

Step 3

Applying the dynamic approach formula only in the upper triangular values of the cost table, since i < j always

$C[1,2] = \min\limits_{1 \le k < 2} \{ C[1,1] + C[2,2] + d_0 d_1 d_2 \}$

- $C[1,2] = 0 + 0 + (5 \times 10 \times 15)$
- $C[1,2] = 750$

$C[2,3] = \min\limits_{2 \le k < 3} \{ C[2,2] + C[3,3] + d_1 d_2 d_3 \}$

- $C[2,3] = 0 + 0 + (10 \times 15 \times 20)$
- $C[2,3] = 3000$

$C[3,4] = \min\limits_{3 \le k < 4} \{ C[3,3] + C[4,4] + d_2 d_3 d_4 \}$

- $C[3,4] = 0 + 0 + (15 \times 20 \times 25)$
- $C[3,4] = 7500$

cost table	1	2	3	4
1	0	750		
2		0	3000	
3			0	7500
4				0

Step 4

Find the values of [1, 3] and [2, 4] in this step. The cost table is always filled diagonally step-wise.

$$C[2,4] = \min_{2 \le k < 4} \{ C[2,2] + C[3,4] + d_1 d_2 d_4, C[2,3] + C[4,4] + d_1 d_3 d_4 \}$$

- $C[2,4] = min\{(0 + 7500 + (10 \times 15 \times 20)), (3000 + 5000)\}$
- $C[2,4] = 8000$

$$C[1,3] = \min_{1 \le k < 3} \{ C[1,1] + C[2,3] + d_0 d_1 d_3, C[1,2] + C[3,3] + d_0 d_2 d_3 \}$$

- $C[1,3] = min\{(0 + 3000 + 1000), (1500 + 0 + 750)\}$
- $C[1,3] = 2250$

cost table	1	2	3	4
1	0	750	2200	
2		0	3000	8000
3			0	7500
4				0

Step 5

Now compute the final element of the cost table to compare the lowest cost parenthesization

$$C[1,4] = \min_{1 \le k < 4} \left\{ \begin{array}{c} C[1,1] + C[2,4] + d_0 d_1 d_4, C[1,2] + C[3,4] + d_1 d_2 d_4, C[1,3] \\ + C[4,4] + d_1 d_3 d_4 \end{array} \right\}$$

- $C[1,4] = min\{0 + 8000 + 1250, 750 + 7500 + 1875, 2200 + 0 + 2500\}$
- $C[1,4] = 4700$

cost table	1	2	3	4
1	0	750	2200	4700
2		0	3000	8000
3			0	7500
4				0

Now that all the values in cost table are computed, the final step is to parethesize the sequence of matrices. For that, **k** table needs to be constructed with the minimum value of 'k' corresponding to every parenthesis.

k	1	2	3	4
1		1	2	3
2			2	3
3				3
4				

5.3.5.3 Parenthesizing

Based on the lowest cost values from the cost table and their corresponding k values, let us add parenthesis on the sequence of matrices.

The lowest cost value at [1, 4] is achieved when k = 3, therefore, the first parenthesizing must be done at 3.

(ABC)(D)

The lowest cost value at [1, 3] is achieved when k = 2, therefore the next parenthesization is done at 2.

((AB)C) (D)

The lowest cost value at [1, 2] is achieved when k = 1, therefore the next parenthesization is done at 1. But the parenthesization needs at least two matrices to be multiplied so we do not divide further.

((AB)(C)) (D)

Since, the sequence cannot be parenthesized further, the final solution of matrix chain multiplication is ((AB)C) (D).

5.3.5.4 Implementation

Following is the final implementation of Matrix Chain Multiplication Algorithm to calculate the minimum number of ways several matrices can be multiplied using dynamic programming −

Python code

```
mc = [[-1 for n in range(50)] for m in range(50)]
def DynamicProgramming(c, i, j):
   if (i == j):
      return 0
   if (mc[i][j] != -1):
      return mc[i][j]
   mc[i][j] = 999999
   for k in range (i, j):
      mc[i][j] = min(mc[i][j], DynamicProgramming(c, i, k) +
DynamicProgramming(c, k + 1, j) + c[i - 1] * c[k] * c[j]);
   return mc[i][j]

def Matrix(c, n):
   i = 1
   j = n - 1
   return DynamicProgramming(c, i, j);

arr = [ 23, 26, 27, 20 ]
n = len(arr)
print("Minimum number of multiplications is: ")
print(Matrix(arr, n))
```

Output

```
Minimum number of multiplications is:
26000
```

5.3.6 Floyd Warshall Algorithm

The Floyd-Warshall algorithm is a graph algorithm that is deployed to find the shortest path between all the vertices present in a weighted graph. This algorithm is different from other shortest path algorithms; to describe it simply, this algorithm uses each vertex in the graph as a pivot to check if it provides the shortest way to travel from one point to another.

Floyd-Warshall algorithm works on both directed and undirected weighted graphs unless these graphs do not contain any negative cycles in them. By negative cycles, it is meant that the sum of all the edges in the graph must not lead to a negative number.

Since, the algorithm deals with overlapping sub-problems – the path found by the vertices acting as pivot are stored for solving the next steps – it uses the dynamic programming approach.

Floyd-Warshall algorithm is one of the methods in All-pairs shortest path algorithms and it is solved using the Adjacency Matrix representation of graphs.

Consider a graph, **G = {V, E}** where **V** is the set of all vertices present in the graph and E is the set of all the edges in the graph. The graph, **G**, is represented in the form of an adjacency matrix, **A**, that contains all the weights of every edge connecting two vertices.

5.3.6.1 Algorithm

Step 1 − Construct an adjacency matrix **A** with all the costs of edges present in the graph. If there is no path between two vertices, mark the value as ∞.

Step 2 − Derive another adjacency matrix $\mathbf{A_1}$ from **A** keeping the first row and first column of the original adjacency matrix intact in $\mathbf{A_1}$. And for the remaining values, say $\mathbf{A_1[i,j]}$, if **A[i,j]>A[i,k]+A[k,j]** then replace $\mathbf{A_1[i,j]}$ with **A[i,k]+A[k,j]**. Otherwise, do not change the values. Here, in this step, **k = 1** (first vertex acting as pivot).

Step 3 − Repeat **Step 2** for all the vertices in the graph by changing the **k** value for every pivot vertex until the final matrix is achieved.

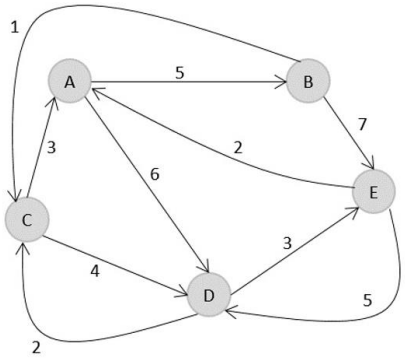
Step 4 − The final adjacency matrix obtained is the final solution with all the shortest paths.

5.3.6.2 Pseudocode

```
Floyd-Warshall(w, n){ // w: weights, n: number of vertices
  for i = 1 to n do // initialize, D (0) = [wij]
    for j = 1 to n do{
      d[i, j] = w[i, j];
    }
    for k = 1 to n do // Compute D (k) from D (k-1)
      for i = 1 to n do
        for j = 1 to n do
          if (d[i, k] + d[k, j] < d[i, j]){
            d[i, j] = d[i, k] + d[k, j];
          }
    return d[1..n, 1..n];
}
```

Example

Consider the following directed weighted graph **G = {V, E}**. Find the shortest paths between all the vertices of the graphs using the Floyd-Warshall algorithm.



Solution

Step 1

Construct an adjacency matrix **A** with all the distances as values.

$$A = \begin{matrix} 0 & 5 & \infty & 6 & \infty \\ \infty & 0 & 1 & \infty & 7 \\ 3 & \infty & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 2 & \infty & \infty & 5 & 0 \end{matrix}$$

Step 2

Considering the above adjacency matrix as the input, derive another matrix A_0 by keeping only first rows and columns intact. Take **k = 1**, and replace all the other values by ***A[i,k]+A[k,j]***.

$$A = \begin{matrix} 0 & 5 & \infty & 6 & \infty \\ \infty & & & & \\ 3 & & & & \\ \infty & & & & \\ 2 & & & & \end{matrix}$$

$$A_1 = \begin{matrix} 0 & 5 & \infty & 6 & \infty \\ \infty & 0 & 1 & \infty & 7 \\ 3 & 8 & 0 & 4 & \infty \\ \infty & \infty & 2 & 0 & 3 \\ 2 & 7 & \infty & 5 & 0 \end{matrix}$$

Step 3

Considering the above adjacency matrix as the input, derive another matrix A_0 by keeping only first rows and columns intact. Take **k = 1**, and replace all the other values by ***A[i,k]+A[k,j]***.

$$A_2 = \begin{matrix} & & 5 & & \\ \infty & 0 & 1 & \infty & 7 \\ & & 8 & & \\ & & \infty & & \\ & & 7 & & \end{matrix}$$

$$A_2 = \begin{matrix} 0 & 5 & 6 & 6 & 12 \\ \infty & 0 & 1 & \infty & 7 \\ 3 & 8 & 0 & 4 & 15 \\ \infty & \infty & 2 & 0 & 3 \\ 2 & 7 & 8 & 5 & 0 \end{matrix}$$

Step 4

Considering the above adjacency matrix as the input, derive another matrix ***A0*** by keeping only first rows and columns intact. Take **k = 1**, and replace all the other values by ***A[i,k]+A[k,j]***.

$$A_3 = \begin{matrix} & & 6 & & \\ & & 1 & & \\ 3 & 8 & 0 & 4 & 15 \\ & & 2 & & \\ & & 8 & & \end{matrix}$$

$$A_3 = \begin{matrix} 0 & 5 & 6 & 6 & 12 \\ 4 & 0 & 1 & 5 & 7 \\ 3 & 8 & 0 & 4 & 15 \\ 5 & 10 & 2 & 0 & 3 \\ 2 & 7 & 8 & 5 & 0 \end{matrix}$$

Step 5

Considering the above adjacency matrix as the input, derive another matrix ***A0*** by keeping only first rows and columns intact. Take **k = 1**, and replace all the other values by ***A[i,k]+A[k,j]***.

$$A_4 = \begin{matrix} & & & & 6 \\ & & & & 5 \\ & & & & 4 \\ 5 & 10 & 2 & 0 & 3 \\ & & & & 5 \end{matrix}$$

$$A_4 = \begin{matrix} 0 & 5 & 6 & 6 & 9 \\ 4 & 0 & 1 & 5 & 7 \\ 3 & 8 & 0 & 4 & 7 \\ 5 & 10 & 2 & 0 & 3 \\ 2 & 7 & 7 & 5 & 0 \end{matrix}$$

Step 6

Considering the above adjacency matrix as the input, derive another matrix ***A0*** by keeping only first rows and columns intact. Take **k = 1**, and replace all the other values by ***A[i,k]+A[k,j]***.

$$A_5 = \begin{matrix} & & & & 9 \\ & & & & 7 \\ & & & & 7 \\ & & & & 3 \\ 2 & 7 & 7 & 5 & 0 \end{matrix}$$

$$A_5 = \begin{matrix} 0 & 5 & 6 & 6 & 9 \\ 4 & 0 & 1 & 5 & 7 \\ 3 & 8 & 0 & 4 & 7 \\ 5 & 10 & 2 & 0 & 3 \\ 2 & 7 & 7 & 5 & 0 \end{matrix}$$

5.3.6.3 Analysis

From the pseudocode above, the Floyd-Warshall algorithm operates using three for loops to find the shortest distance between all pairs of vertices within a graph. Therefore, the **time complexity** of the Floyd-Warshall algorithm is **$O(n^3)$**, where 'n' is the number of vertices in the graph. The **space complexity** of the algorithm is **$O(n^2)$**.

5.3.6.4 Implementation

Following is the implementation of Floyd Warshall Algorithm to find the shortest path in a graph using cost adjacency matrix -

Python code

```
import numpy as np
def floyds(b):
    for k in range(3):
        for i in range(3):
            for j in range(3):
                if (b[i][k] * b[k][j] != 0) and (i != j):
                    if (b[i][k] + b[k][j] < b[i][j]) or (b[i][j] ==
0):
                        b[i][j] = b[i][k] + b[k][j]
    for i in range(3):
        print("Minimum Cost With Respect to Node:", i)
        for j in range(3):
            print(b[i][j], end="\t")
b = np.zeros((3, 3), dtype=int)
b[0][1] = 10
b[1][2] = 15
b[2][0] = 12
#calling the method
floyds(b)
```

Output

```
Minimum Cost With Respect to Node: 0
0       10      25
Minimum Cost With Respect to Node: 1
27      0       15
Minimum Cost With Respect to Node: 2
12      22      0
```

5.3.7 0-1 Knapsack Problem (algorithm)

We discussed the fractional knapsack problem using the greedy approach, earlier in this tutorial. It is shown that Greedy approach gives an optimal solution for Fractional Knapsack. However, this chapter will cover 0-1 Knapsack problem using dynamic programming approach and its analysis.

Unlike in fractional knapsack, the items are always stored fully without using the fractional part of them. Its either the item is added to the knapsack or not. That is why, this method is known as the **0-1 Knapsack problem**.

Hence, in case of 0-1 Knapsack, the value of x_i can be either **0** or **1**, where other constraints remain the same.

0-1 Knapsack cannot be solved by Greedy approach. Greedy approach does not ensure an optimal solution in this method. In many instances, Greedy approach may give an optimal solution.

5.3.7.1 0-1 Knapsack Algorithm

Problem Statement − A thief is robbing a store and can carry a maximal weight of ***W*** into his knapsack. There are ***n*** items and weight of **ith** item is **wi** and the profit of selecting this item is ***pi***. What items should the thief take?

Let **i** be the highest-numbered item in an optimal solution **S** for **W** dollars. Then **S' = S − {i}** is an optimal solution for **W – w_i** dollars and the value to the solution S is V_i plus the value of the sub-problem.

We can express this fact in the following formula: define **c[i, w]** to be the solution for items **1,2, … , i** and the maximum weight **w**.

The algorithm takes the following inputs

- The maximum weight **W**
- The number of items **n**
- The two sequences **v = <v1, v2, …, vn> and w = <w1, w2, …, wn>**

The set of items to take can be deduced from the table, starting at **c[n, w]** and tracing backwards where the optimal values came from.

If ***c[i, w] = c[i-1, w]***, then item i is not part of the solution, and we continue tracing with **c[i-1, w]**. Otherwise, item i is part of the solution, and we continue tracing with **c [i-1, w-W]**.

```
Dynamic-0-1-knapsack (v, w, n, W)
for w = 0 to W do
   c[0, w] = 0
for i = 1 to n do
   c[i, 0] = 0
   for w = 1 to W do
      if wi ≤ w then
         if vi + c[i-1, w-wi] then
            c[i, w] = vi + c[i-1, w-wi]
         else c[i, w] = c[i-1, w]
      else
         c[i, w] = c[i-1, w]
```

The following examples will establish our statement.

Example

Let us consider that the capacity of the knapsack is W = 8 and the items are as shown in the following table.

Item	A	B	C	D
Profit	2	4	7	10
Weight	1	3	5	7

Solution

Using the greedy approach of 0-1 knapsack, the weight that's stored in the knapsack would be A+B = 4 with the maximum profit 2 + 4 = 6. But, that solution would not be the optimal solution.

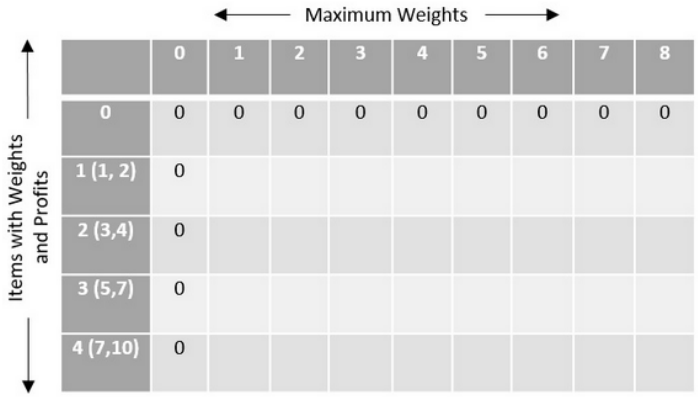
Therefore, dynamic programming must be adopted to solve 0-1 knapsack problems.

Step 1

Construct an adjacency table with maximum weight of knapsack as rows and items with respective weights and profits as columns.

Values to be stored in the table are cumulative profits of the items whose weights do not exceed the maximum weight of the knapsack (designated values of each row)

So we add zeroes to the 0^{th} row and 0^{th} column because if the weight of item is 0, then it weighs nothing; if the maximum weight of knapsack is 0, then no item can be added into the knapsack.

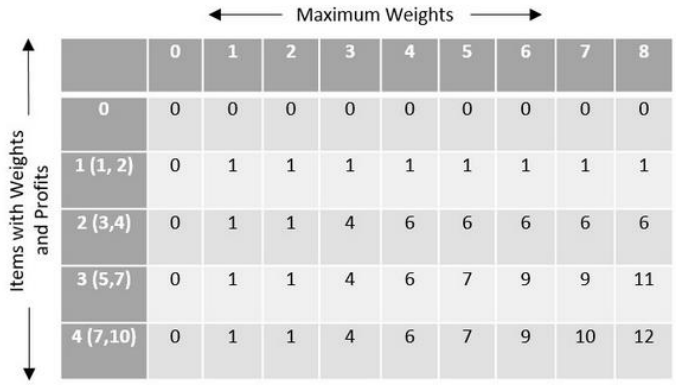


The remaining values are filled with the maximum profit achievable with respect to the items and weight per column that can be stored in the knapsack.

The formula to store the profit values is −

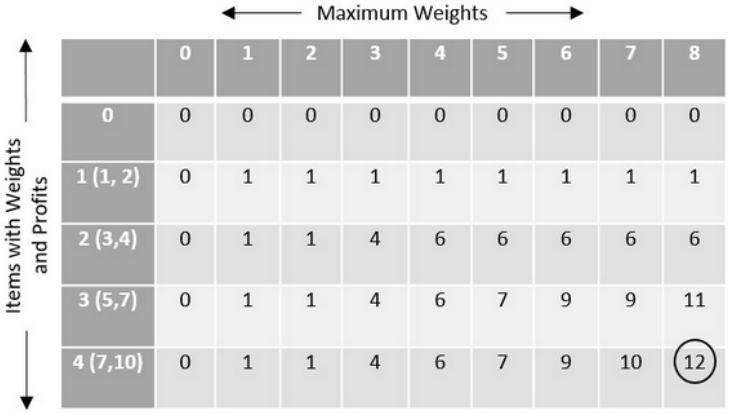
$$c[i,w]=max\{c[i-1,w-w[i]]+P[i]\}$$

By computing all the values using the formula, the table obtained would be −



	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1 (1, 2)	0	1	1	1	1	1	1	1	1
2 (3,4)	0	1	1	4	6	6	6	6	6
3 (5,7)	0	1	1	4	6	7	9	9	11
4 (7,10)	0	1	1	4	6	7	9	10	12

To find the items to be added in the knapsack, recognize the maximum profit from the table and identify the items that make up the profit, in this example, its {1, 7}.



	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1 (1, 2)	0	1	1	1	1	1	1	1	1
2 (3,4)	0	1	1	4	6	6	6	6	6
3 (5,7)	0	1	1	4	6	7	9	9	11
4 (7,10)	0	1	1	4	6	7	9	10	(12)

The optimal solution is {1, 7} with the maximum profit is 12.

5.3.7.2 Analysis

This algorithm takes Θ (n.w) times as table c has (n+1).(w+1) entries, where each entry requires Θ(1) time to compute.

5.3.7.3 Implementation

Following is the final implementation of 0-1 Knapsack Algorithm using Dynamic Programming Approach.

```
def knapsack(W, wt, val, n):
    K = [[0] * (W+1) for i in range (n+1)]
    for i in range(n+1):
```

```
      for w in range(W+1):
         if(i == 0 or w == 0):
            K[i][w] = 0
         elif(wt[i-1] <= w):
            K[i][w] = max(val[i-1] + K[i-1][w-wt[i-1]], K[i-1][w])
         else:
            K[i][w] = K[i-1][w]
   return K[n][W]

val = [70, 20, 50];
wt = [11, 12, 13];
W = 30;
ln = len(val);
profit = knapsack(W, wt, val, ln)
print("Maximum Profit achieved with this knapsack: ")
print(profit)
```

Output

Maximum Profit achieved with this knapsack:
120

5.4 Exercises on Algorithm design technique

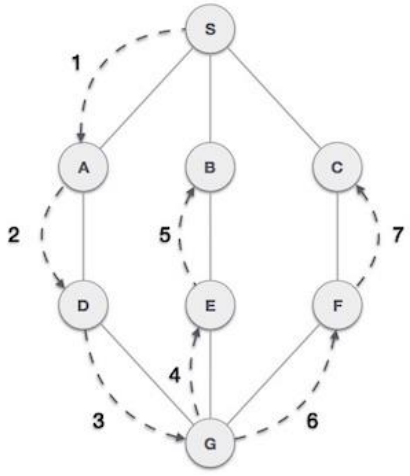
1. Implement merge sort algorithm using Java
2. Implement Travelling Salesperson algorithm using Java
3. Implement Kruskal's minimal spanning tree algorithm using Java
4. Implement Dijkstras's shortest path algorithm using Java
5. Implement map coloring algorithm using java
6. Implement matrix chain multiplication algorithm using java
7. Implement floyed Warshal algorithm using java
8. Implement Knapsack Algorithm using Java

Chapter 6

6 Algorithm for fundamental graphs

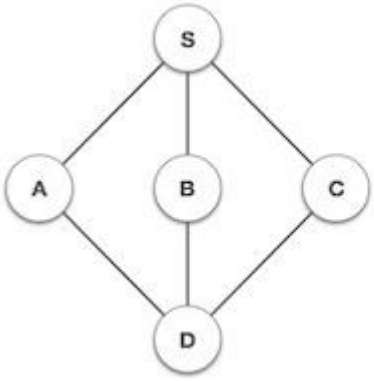
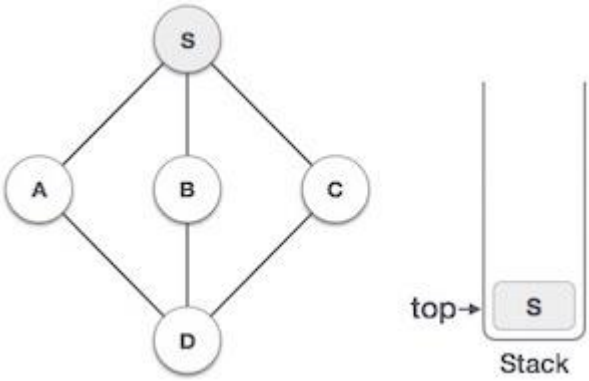
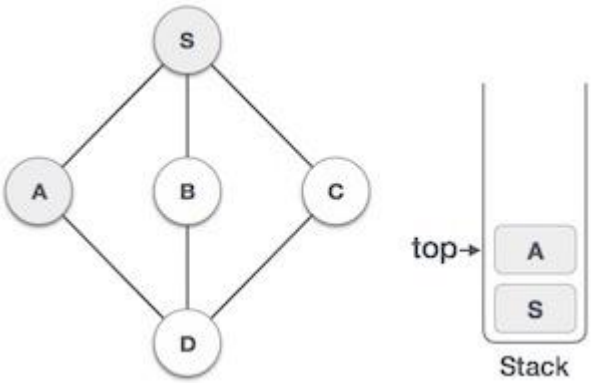
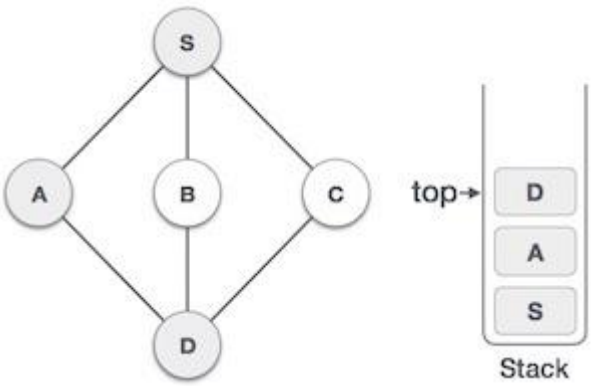
6.1. Depth First Traversal

Depth First Search (DFS) algorithm traverses a graph in a depth ward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

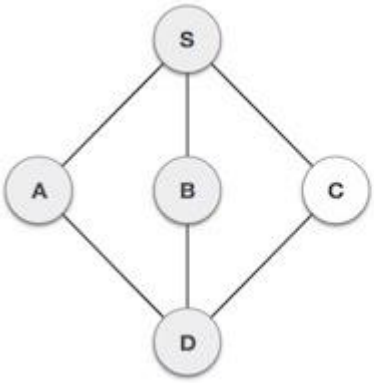


As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** − Repeat Rule 1 and Rule 2 until the stack is empty.

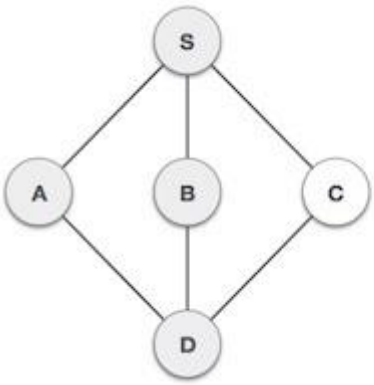
Step	Traversal	Description
1		Initialize the stack.
2		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.
4		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.

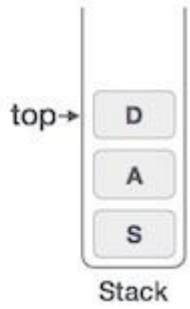
5



We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.

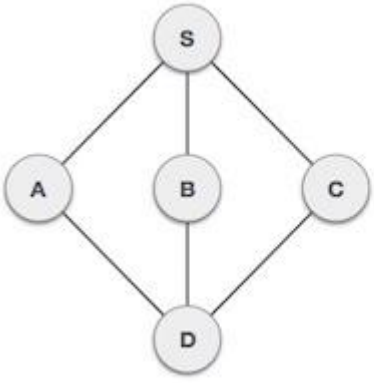
6





We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.

7



Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.

Example

Following are the implementations of this operation in various programming languages –

Python code

```
#Python program for Depth First Traversal
MAX = 5
class Vertex:
    def __init__(self, label):
        self.label = label
        self.visited = False
#stack variables
stack = []
top = -1
#graph variables
#array of vertices
lstVertices = [None] * MAX
#adjacency matrix
adjMatrix = [[0] * MAX for _ in range(MAX)]
#vertex count
vertexCount = 0
#stack functions
def push(item):
    global top
    top += 1
    stack.append(item)
def pop():
    global top
    item = stack[top]
    del stack[top]
    top -= 1
    return item
def peek():
    return stack[top]
def isStackEmpty():
    return top == -1
#graph functions
#add vertex to the vertex list
def addVertex(label):
    global vertexCount
    vertex = Vertex(label)
    lstVertices[vertexCount] = vertex
    vertexCount += 1
#add edge to edge array
def addEdge(start, end):
    adjMatrix[start][end] = 1
    adjMatrix[end][start] = 1
#Display the Vertex
```

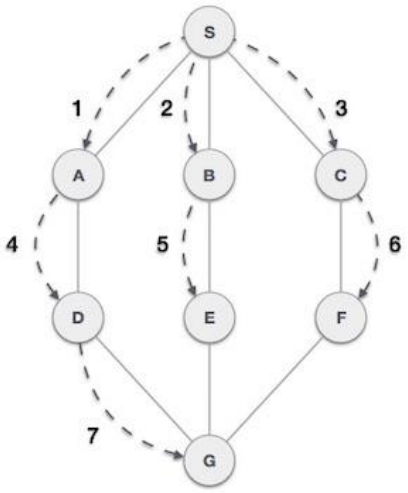
```
def displayVertex(vertexIndex):
    print(lstVertices[vertexIndex].label, end=' ')
def getAdjUnvisitedVertex(vertexIndex):
    for i in range(vertexCount):
        if adjMatrix[vertexIndex][i] == 1 and not
lstVertices[i].visited:
            return i
    return -1
def depthFirstSearch():
    lstVertices[0].visited = True
    displayVertex(0)
    push(0)
    while not isStackEmpty():
        unvisitedVertex = getAdjUnvisitedVertex(peek())
        if unvisitedVertex == -1:
            pop()
        else:
            lstVertices[unvisitedVertex].visited = True
            displayVertex(unvisitedVertex)
            push(unvisitedVertex)
    for i in range(vertexCount):
        lstVertices[i].visited = False
for i in range(MAX):
    for j in range(MAX):
        adjMatrix[i][j] = 0
addVertex('S')   # 0
addVertex('A')   # 1
addVertex('B')   # 2
addVertex('C')   # 3
addVertex('D')   # 4
addEdge(0, 1)    # S - A
addEdge(0, 2)    # S - B
addEdge(0, 3)    # S - C
addEdge(1, 4)    # A - D
addEdge(2, 4)    # B - D
addEdge(3, 4)    # C - D
print("Depth First Search:", end=' ')
depthFirstSearch()
```

Output

```
Depth First Search: S A D B C
```

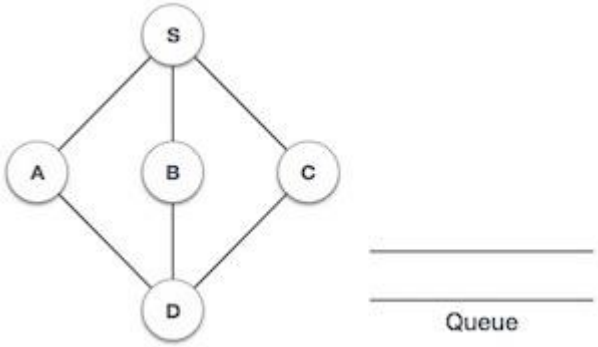
6.1 Breadth first traversal

Breadth First Search (BFS) algorithm traverses a graph in a breadth ward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

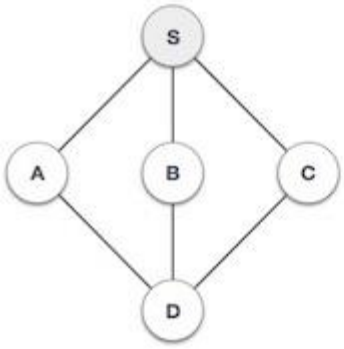


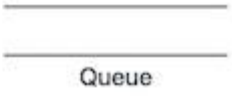
As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- **Rule 2** − If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** − Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1		Initialize the queue.

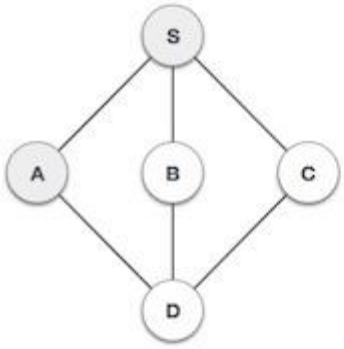
2

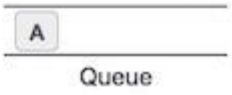




We start from visiting **S** (starting node), and mark it as visited.

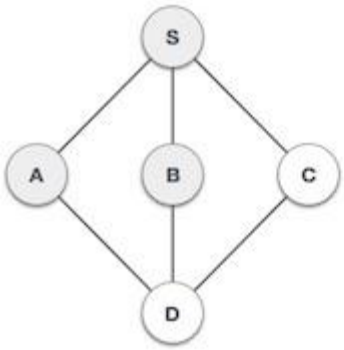
3

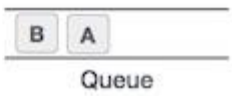




We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.

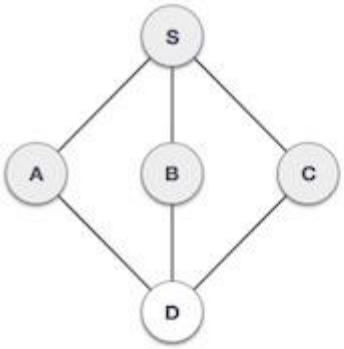
4

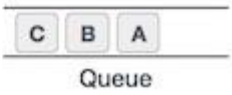




Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.

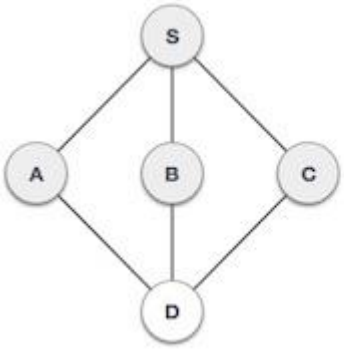
5

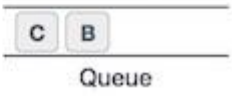




Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.

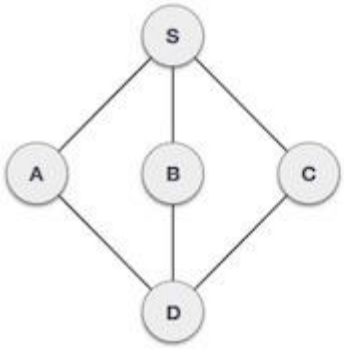
6

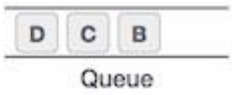




Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.

7





From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

Example

Following are the implementations of this operation in various programming languages −

Python code

```
#Python program for Breadth First Search
# defining MAX 5
MAX = 5
class Vertex:
   def __init__(self, label):
      self.label = label
      self.visited = False
# queue variables
queue = [0] * MAX
rear = -1
front = 0
queueItemCount = 0
# graph variables
#array of vertices
lstVertices = [None] * MAX
```

```
#adjacency matrix
adjMatrix = [[0] * MAX for _ in range(MAX)]
#vertex count
vertexCount = 0
# queue functions
def insert(data):
   global rear, queueItemCount
   rear += 1
   queue[rear] = data
   queueItemCount += 1
def removeData():
   global front, queueItemCount
   queueItemCount -= 1
   data = queue[front]
   front += 1
   return data
def isQueueEmpty():
   return queueItemCount == 0
# graph functions
#add vertex to the vertex list
def addVertex(label):
   global vertexCount
   vertex = Vertex(label)
   lstVertices[vertexCount] = vertex
   vertexCount += 1
#add edge to edge array
def addEdge(start, end):
   adjMatrix[start][end] = 1
   adjMatrix[end][start] = 1
#Display the vertex
def displayVertex(vertexIndex):
   print(lstVertices[vertexIndex].label, end=" ")
#Get the adjacent unvisited vertex
def getAdjUnvisitedVertex(vertexIndex):
   for i in range(vertexCount):
      if adjMatrix[vertexIndex][i] == 1 and not
lstVertices[i].visited:
         return i
   return -1
def breadthFirstSearch():
    #mark first node as visited
   lstVertices[0].visited = True
   #Display the vertex
   displayVertex(0)
   #insert vertex index in queue
```

```
   insert(0)
   while not isQueueEmpty():
    #get the unvisited vertex of vertex which is at front of the queue
      tempVertex = removeData()
      #no adjacent vertex found
      unvisitedVertex = getAdjUnvisitedVertex(tempVertex)
      while unvisitedVertex != -1:
         lstVertices[unvisitedVertex].visited = True
         displayVertex(unvisitedVertex)
         insert(unvisitedVertex)
         unvisitedVertex = getAdjUnvisitedVertex(tempVertex)
    #queue is empty, search is complete, reset the visited flag
   for i in range(vertexCount):
      lstVertices[i].visited = False
# main function
if __name__ == "__main__":
    #set adjacency
   for i in range(MAX):
       #matrix to 0
       for j in range(MAX):
         adjMatrix[i][j] = 0
   addVertex('S')
   addVertex('A')
   addVertex('B')
   addVertex('C')
   addVertex('D')
   addEdge(0, 1)
   addEdge(0, 2)
   addEdge(0, 3)
   addEdge(1, 4)
   addEdge(2, 4)
   addEdge(3, 4)
   print("Breadth First Search: ", end="")
   breadthFirstSearch()
```

Output

```
Breadth First Search: S A B C D
```

6.2 Tree traversal

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree −

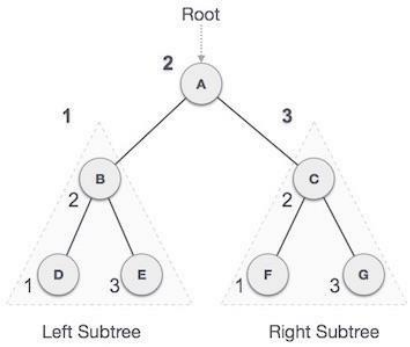
- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

6.2.1 In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself.

If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**.**B** is also traversed in-order. The process goes on until all the nodes are visited. The output of in-order traversal of this tree will be −

D → B → E → A → F → C → G

6.2.1.1 Algorithm

Until all nodes are traversed −

```
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
```

```
Step 3 – Recursively traverse right subtree.
```

Example

Following are the implementations of this operation in various programming languages –

Python code

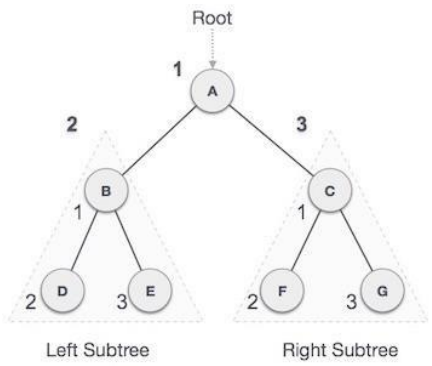
```
class Node:
   def __init__(self, key):
      self.leftChild = None
      self.rightChild = None
      self.data = key

# Create a function to perform inorder tree traversal
def InorderTraversal(root):
   if root:
      InorderTraversal(root.leftChild)
      print(root.data)
      InorderTraversal(root.rightChild)

# Main class
if __name__ == "__main__":
   root = Node(3)
   root.leftChild = Node(26)
   root.rightChild = Node(42)
   root.leftChild.leftChild = Node(54)
   root.leftChild.rightChild = Node(65)
   root.rightChild.leftChild = Node(12)

   # Function call
   print("Inorder traversal of binary tree is")
   InorderTraversal(root)
```

Output

```
Inorder traversal of binary tree is
54
26
65
3
12
42
```

6.2.2 Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

A → B → D → E → C → F → G

6.2.2.1 Algorithm

Until all nodes are traversed −

```
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.
```

Example

Following are the implementations of this operation in various programming languages −

Python code

```
class Node:
   def __init__(self, key):
      self.leftChild = None
      self.rightChild = None
      self.data = key

# Create a function to perform postorder tree traversal
def PreorderTraversal(root):
   if root:
      print(root.data)
      PreorderTraversal(root.leftChild)
      PreorderTraversal(root.rightChild)
```

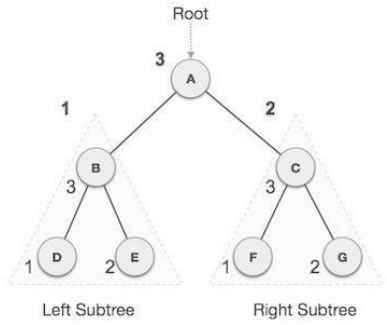
```
# Main class
if __name__ == "__main__":
   root = Node(3)
   root.leftChild = Node(26)
   root.rightChild = Node(42)
   root.leftChild.leftChild = Node(54)
   root.leftChild.rightChild = Node(65)
   root.rightChild.leftChild = Node(12)
   print("Preorder traversal of binary tree is")
   PreorderTraversal(root)
```

Output

```
Preorder traversal of binary tree is
3
26
54
65
42
12
```

6.2.2.2 Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First, we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following pre-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

D → E → B → F → G → C → A

6.2.2.3 Algorithm

Until all nodes are traversed −

```
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.
```

Example

Following are the implementations of this operation in various programming languages −

Python

```
class Node:
   def __init__(self, key):
      self.leftChild = None
      self.rightChild = None
      self.data = key

# Create a function to perform preorder tree traversal
def PostorderTraversal(root):
   if root:
      PostorderTraversal(root.leftChild)
      PostorderTraversal(root.rightChild)
      print(root.data)

# Main class
if __name__ == "__main__":
   root = Node(3)
   root.leftChild = Node(26)
   root.rightChild = Node(42)
   root.leftChild.leftChild = Node(54)
   root.leftChild.rightChild = Node(65)
   root.rightChild.leftChild = Node(12)
   print("Postorder traversal of binary tree is")
   PostorderTraversal(root)
```

output

```
Post order traversal of binary tree is
54
65
26
12
42
3
```

6.3 Binary search tree

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties −

- The left sub-tree of a node has a key less than or equal to its parent node's key.
- The right sub-tree of a node has a key greater than or equal to its parent node's key.

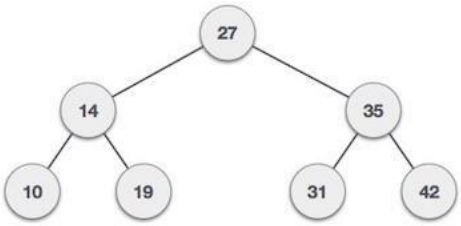
Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as −

```
left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)
```

6.3.1 Binary Tree Representation

BST is a collection of nodes arranged in a way where they maintain BST properties. Each node has a key and an associated value. While searching, the desired key is compared to the keys in BST and if found, the associated value is retrieved.

Following is a pictorial representation of BST −



We observe that the root node key (27) has all less-valued keys on the left sub-tree and the higher valued keys on the right sub-tree.

6.3.2 Basic Operations

Following are the basic operations of a Binary Search Tree −

- **Search** − Searches an element in a tree.
- **Insert** − Inserts an element in a tree.
- **Pre-order Traversal** − Traverses a tree in a pre-order manner.
- **In-order Traversal** − Traverses a tree in an in-order manner.
- **Post-order Traversal** − Traverses a tree in a post-order manner.

6.3.3 Defining a Node

Define a node that stores some data, and references to its left and right child nodes.

```
struct node {
   int data;
   struct node *leftChild;
   struct node *rightChild;
};
```

6.3.4 Search Operation

Whenever an element is to be searched, start searching from the root node. Then if the data is less than the key value, search for the element in the left subtree. Otherwise, search for the element in the right subtree. Follow the same algorithm for each node.

6.3.4.1 Algorithm

```
1. START
2. Check whether the tree is empty or not
3. If the tree is empty, search is not possible
4. Otherwise, first search the root of the tree.
5. If the key does not match with the value in the root,
   search its subtrees.
6. If the value of the key is less than the root value,
   search the left subtree
7. If the value of the key is greater than the root value,
   search the right subtree.
8. If the key is not found in the tree, return unsuccessful search.
9. END
```

Example

Following are the implementations of this operation in various programming languages –

Python code

```
class Node:
   def __init__(self, data):
      self.left = None
      self.right = None
      self.data = data

# Insert method to create nodes
   def insert(self, data):
      if self.data:
         if data < self.data:
            if self.left is None:
               self.left = Node(data)
            else:
               self.left.insert(data)
```

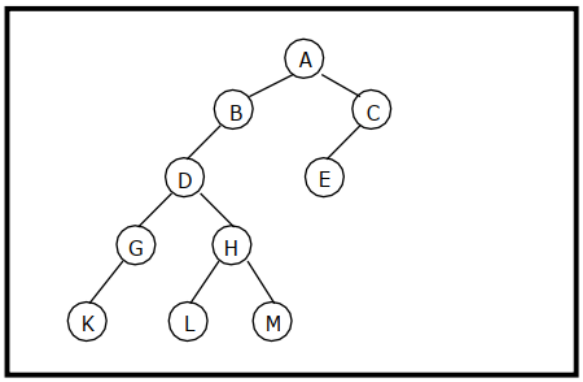
```
         elif data > self.data:
            if self.right is None:
               self.right = Node(data)
            else:
               self.right.insert(data)
         else:
            self.data = data
# search method to compare the value with nodes
   def search(self, key):
      if key < self.data:
         if self.left is None:
            return str(key)+" Not Found"
         return self.left.search(key)
      elif key > self.data:
         if self.right is None:
            return str(key)+" Not Found"
         return self.right.search(key)
      else:
         print(str(self.data) + ' is found')

root = Node(54)
root.insert(34)
root.insert(46)
root.insert(12)
root.insert(23)
root.insert(5)
print(root.search(17))
print(root.search(12))
```

output

```
17 Not Found
12 is found
None
```

Example

Traverse the following binary tree in pre, post and inorder using non-recursive traversing algorithm.



Bin a ry T re e

- Preo rde r t rav e rs al y ie lds: A, B, D, G , K, H, L, M , C , E
- Posto rde r t rav ars al y ie lds: K, G , L, M , H, D, B, E, C , A
- Ino rde r trav ars al yie lds: K, G , D, L, H, M , B, A, E, C

Pre, Post and In ord er Trav ersin g

More detail about the operation

Post order traversal

Current vertex	Stack	Processed nodes	Remarks
A	0		PUSH 0
	0 A -C B D -H G K		PUSH the left most path of A with a -ve for right sons
	0 A -C B D -H	K G	POP all +ve nodes K and G
H	0 A -C B D	K G	Pop H
	0 A -C B D H -M L	K G	PUSH the left most path of H with a -ve for right sons
	0 A -C B D H -M	K G L	POP all +ve nodes L
M	0 A -C B D H	K G L	Pop M
	0 A -C B D H M	K G L	PUSH the left most path of M with a -ve for right sons
	0 A -C	K G L M H D B	POP all +ve nodes M, H, D and B
C	0 A	K G L M H D B	Pop C
	0 A C E	K G L M H D B	PUSH the left most path of C with a -ve for right sons
	0	K G L M H D B E C A	POP all +ve nodes E, C and A
	0		Stop since stack is empty

Preorder traversal

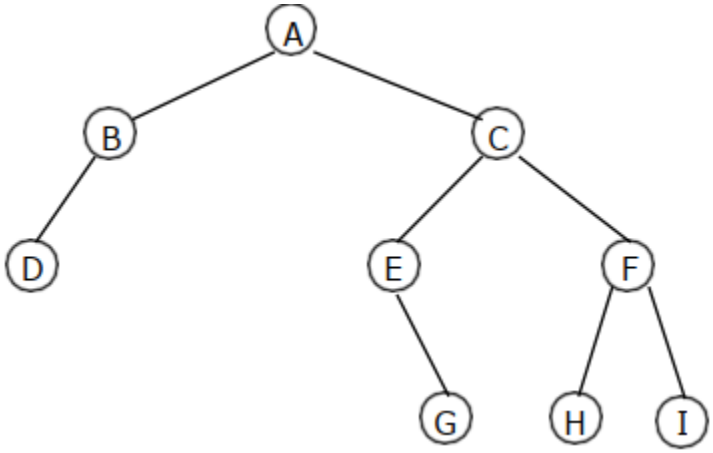
Current vertex	Stack	Processed nodes	Remarks
A	0		PUSH 0
	0 C H	A B D G K	PUSH the right son of each vertex onto stack and process each vertex in the left most path
H	0 C	A B D G K	POP H
	0 C M	A B D G K H L	PUSH the right son of each vertex onto stack and process each vertex in the left most path
M	0 C	A B D G K H L	POP M
	0 C	A B D G K H L M	PUSH the right son of each vertex onto stack and process each vertex in the left most path; M has no left path
C	0	A B D G K H L M	Pop C
	0	A B D G K H L M C E	PUSH the right son of each vertex onto stack and process each vertex in the left most path; C has no right son on the left most path
	0	A B D G K H L M C E	Stop since stack is empty

In order traversal

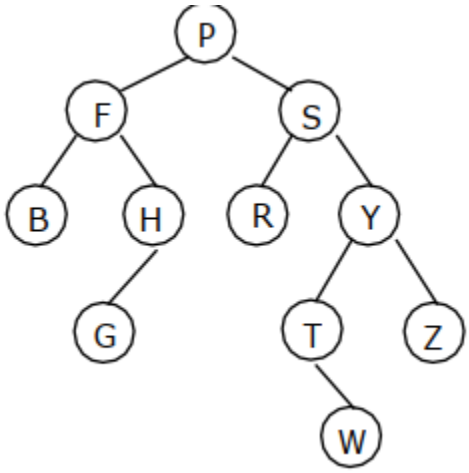
Current vertex	Stack	Processed nodes	Remarks
A	0		PUSH 0
	0 A B D G K		PUSH the left most path of A
K	0 A B D G	K	POP K
G	0 A B D	K G	POP G since K has no right son
D	0 A B	K G D	POP D since G has no right son
H	0 A B	K G D	Make the right son of D as vertex
H	0 A B H L	K G D	PUSH the leftmost path of H
L	0 A B H	K G D L	POP L
H	0 A B	K G D L H	POP H since L has no right son
M	0 A B	K G D L H	Make the right son of H as vertex
	0 A B M	K G D L H	PUSH the left most path of M
M	0 A B	K G D L H M	POP M
B	0 A	K G D L H M B	POP B since M has no right son
A	0	K G D L H M B A	Make the right son of A as vertex
C	0 C E	K G D L H M B A	PUSH the left most path of C
E	0 C	K G D L H M B A E	POP E
C	0	K G D L H M B A E C	Stop since stack is empty

6.4 Exercises on tree traversal

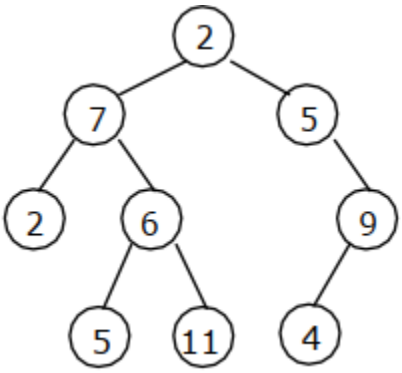
1. **Travers the following tree in pre order, in order and post order? Write a python code for it?**



2. **Travers the following tree in pre order, in order and post order? Write a python code for it?**



3. **Travers the following tree in pre order, in order and post order? Write a python code for it?**



4. Travers the following tree in pre order, in order and post order? Write a python code for it?