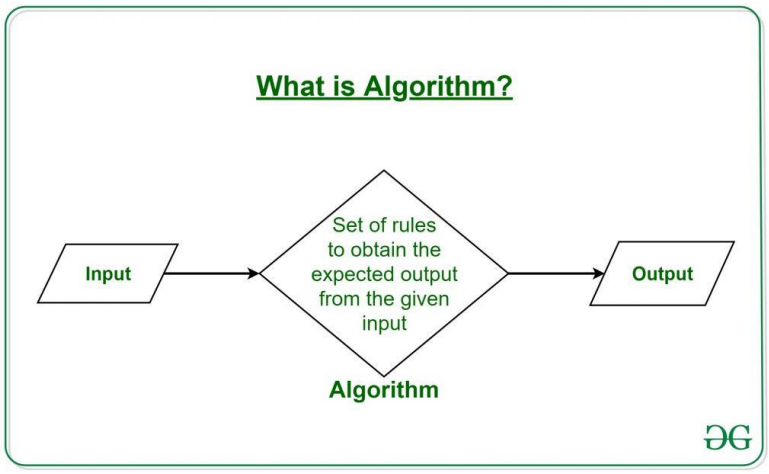
CHAPTER – 1 What is Algorithm? and Basics

1.1 Algorithm

An algorithm is **a set of steps of operations to solve a problem** performing calculation, data processing, and automated reasoning tasks. An algorithm is an efficient method that can be expressed within a finite amount of time and space.



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.

Algorithm is like procedure which does something. For example, preparing the dish of food, some experiment done by the student in physics. Each procedure needs some input and produce output.

Algorithms can be expressed as natural languages, programming languages, pseudocode, flowcharts and control tables. Natural language expressions are rare, as they are more ambiguous. Programming languages are normally used for expressing algorithms executed by a computer.

1.2 What is difference between Algorithm and Program?

Algorithm	Program
It is related to designing the solution of a problem.	It is related to implementation of solution of a problem.
Domain knowledge is required who writes algorithm.	Programmer writes it and also need some domain knowledge.
It is written in any language – English like language or mathematical notation as long as it is understandable by programmer.	It is written in any programming languages like C, C++, Python etc..
It is independent of hardware and software.	It is dependent on hardware and software. We need to select either Linux or Windows.
Analysis is performed on the algorithm once it is written.	Testing is performed once it is implemented.

1.3 Characteristics of Algorithms

The main characteristics of algorithms are as follows −

- **Input** – it may or may not take input – 0 or more input.
- **Output** – It must produce some output – otherwise no meaning of it. At least one output.
- **Definiteness (clear/precise)**– each statement must be unambiguous means it must be having clear meaning. For example, we try to find square root of negative number which is not possible.
- **Finiteness** – it must terminate at some point. It is not like running continuously until we stop it like web server.
- **Effectiveness** – we cannot write statement unnecessarily. It must do something and have some effect. Each operation must be simple and feasible so that one can trace it out using paper and pencil. While preparing a dish, we don't do the things (like cutting vegetables but not used in the dish) which is not part of preparation.
- **Language independent**: algorithms must be independent of any language, and they must be implemented in any language.

1.4 How to write an algorithm?

The algorithm is written without using any language, but it is mostly written in C like language.

Algorithm swap(a,b)

begin

temp = a

a = b

b =temp

end

1.5 What is analysis of algorithm?

In computer science, the analysis of algorithms is the **process of finding the computational complexity of algorithms**—the amount of time, storage, or other resources needed to execute them. Usually, this involves determining a function that relates the size of an algorithm's input to the number of steps it takes (its time complexity) or the number of storage locations it uses (its space complexity).

An algorithm is said to be efficient when this function's values are small or grow slowly compared to a growth in the size of the input.

Different inputs of the same size may cause the algorithm to have different behavior, so best, worst and average case descriptions might all be of practical interest.

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.

1.6 Why Analysis of Algorithms is important?

- In software development, before implementing the software, we first need to prepare the design. Without design, it is not possible to make the software.
- When we construct any house then first, we prepare the drawing and then construct it. It is not based on trial and error.

  The following is the importance of analysis of algorithm.
  1. To predict the behavior of an algorithm without implementing it on a specific computer.
  2. It is much more convenient to have simple measures for the efficiency of an algorithm than to implement the algorithm and test the efficiency every time a certain parameter in the underlying computer system changes.
  3. It is impossible to predict the exact behavior of an algorithm. There are too many influencing factors.
  4. The analysis is thus only an approximation; it is not perfect.
  5. More importantly, by analyzing different algorithms, we can compare them to determine the best one for our purpose.

1.7 How to analyse an algorithm?

Algorithm is analysed based on the following parameters:

- Time: how much time it is taking? time function
- Space: how much memory space it requires?
- Network data transfer: how much data it is transferring.
- Power consumption: as devices are different now adays.
- CPU register it uses. For ex. When we are writing algorithm for device driver.

1.8 Time and Space analysis:

Algorithm swap(a,b)

begin

temp = a	*- 1*
a = b	*- 1*
b =temp	*- 1*

end

Time analysis:

The above algorithm takes 3 units of time so, f(n) = 3 which is O (1).

Each statement takes one unit time as constant.

X = a + 5 * b + c * 9

In machine, it is done using many instructions, but we don't consider it. We just see as one instruction and taking one unit time. It is like going to your friend house no need to do planning how you will go there but if you have to go to mars then detailed planning is needed.

Space analysis:

How much memory space is needed?

In the above algorithm, S(n) = 3 which is constant. O(1)

1.9 Priori Analysis and Posteriori Analysis:

Time complexity of an algorithm can be calculated by using two methods:

1. Posteriori Analysis
2. Priori Analysis

Priori Analysis	Posteriori Analysis
Algorithm	Program
Independent of language	Dependent on language
Hardware independent	Hardware dependent
Time and Space function	Watch time and bytes
It uses asymptotic notation to calculate time and space complexity.	It does not use asymptotic notation to calculate time and space complexity.
The time complexity of an algorithm using a priori analysis is same for every system.	The time complexity of an algorithm using a posteriori analysis differ from system to system.
It is cheaper.	It is costlier because it needs hardware and software.
It is done before execution of an algorithm.	It is done after execution of an algorithm.

1.10 Why analysis of algorithm needed?

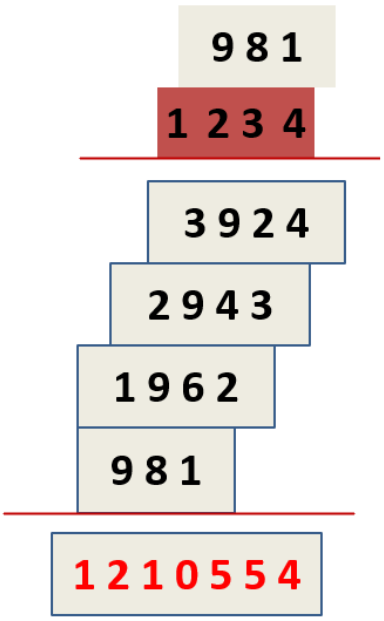
Algorithmics is defined as the study of algorithms. When we set out to solve a problem, there might be a choice of algorithms available. In this case, it is important to decide which one to use. Depending on our priorities and on limits of the equipment available to us, we may want to choose the algorithm that takes least time or least storage or easiest to program and so on.

Now, suppose we have to multiply two positive integers using only pencil and paper.

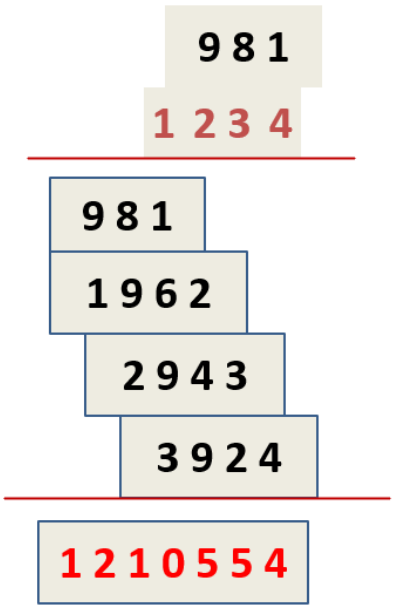
Classical multiplication algorithm:

In North America, they use the American approach of multiplication and in England they use English multiplication approach.

1. American approach



2. English approach



ă la russe approach:

à *la russe* multiplication

1. **Write** the multiplicand and multiplier side by side.
2. **Make two columns**, one under each operand.
3. **Repeat step 4 and 5** until the number in the left column is 1.
4. **Divide** the number in the **left hand column by 2**, ignoring any fractions.
5. **Double** the number in the **right hand column**.
6. Next **cross out** each row where the number in the left hand column is even.
7. Finally **add up** the numbers that remain in the right hand column.

981	1234	1234
490	~~2468~~	
245	4936	4936
122	~~9872~~	
61	19744	19744
30	~~39488~~	
15	78976	78976
7	157952	157952
3	315904	315904
1	631808	631808

Divide and conquer approach of multiplication:

- Both the multiplicand and the multiplier must have the same number of digits and this number be a power of 2.
- If not then it can be done by adding zeros on the left if necessary.

Multiplication by divide and conquer

i. Multiply left half of the multiplicand by left half of multiplier and shift the result by no. of digits of multiplier **i.e. 4.**

ii. Multiply left half of the multiplicand by right half of the multiplier, shift the result by half the number of digits of multiplier **i.e. 2.**

iii. Multiply right half of the multiplicand by left half of the multiplier, shift the result by half the number of digits of multiplier **i.e. 2.**

iv. Multiply right half of the multiplicand by right half of the multiplier the result is **not** shifted at all.

Multiplicand = 0 9 | 8 1

Multiplier = 1 2 | 3 4

Multiply	Shift	Result
(09) * (12)	4	1 0 8 . . . .
(09) * (34)	2	3 0 6 . .
(81) * (12)	2	9 7 2 . .
(81) * (34)	0	2 7 5 4

1 2 1 0 5 5 4

The multiplication of two four-digit numbers is reduced to four multiplications of two-digit numbers.

Again, 2 two-digit numbers' multiplication is given as follows:

Multiply	**Shift**	**Result**
0 * 1	2	00
0 * 2	1	0
9 * 1	1	90
9 * 2	0	18
		108

Each of the above, two-digit multiplication is done using same way with some shifts and addition.

Later, we will see how we can reduce the multiplication from four to three in divide and conquer approach. With these improvements, divide and conquer approach runs faster on computer than any of the preceding methods.

To multiply 3141 5975 with 0818 2818

Multiply	Shift	Result
3141 * 0818	8	256933800000000
3141 * 2818	4	**88513380000**
5975 * 0818	4	48875500000
5975 * 2818	0	16837550
		257071205717550

Multiply	Shift	Result
31 * 28	4	8680000
31 * 18	2	55800
41 * 28	2	114800
41 * 18	0	738
		8851338

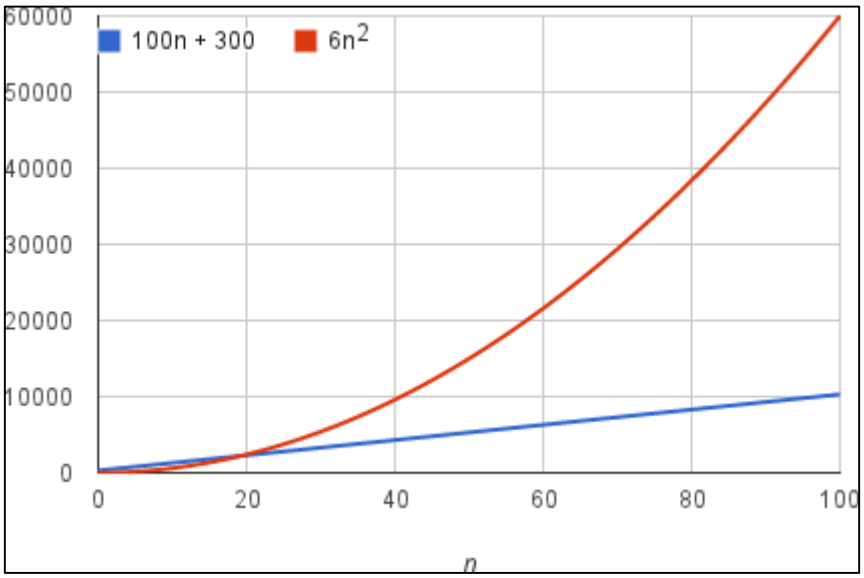
CHAPTER – 2 Asymptotic Notations and Analysis

2.1 Asymptotic Notations

The running time of an algorithm depends on how long it takes a computer to run the lines of code of the algorithm—and that depends on the speed of the computer, the programming language, and the compiler that translates the program from the programming language into code that runs directly on the computer, among other factors.

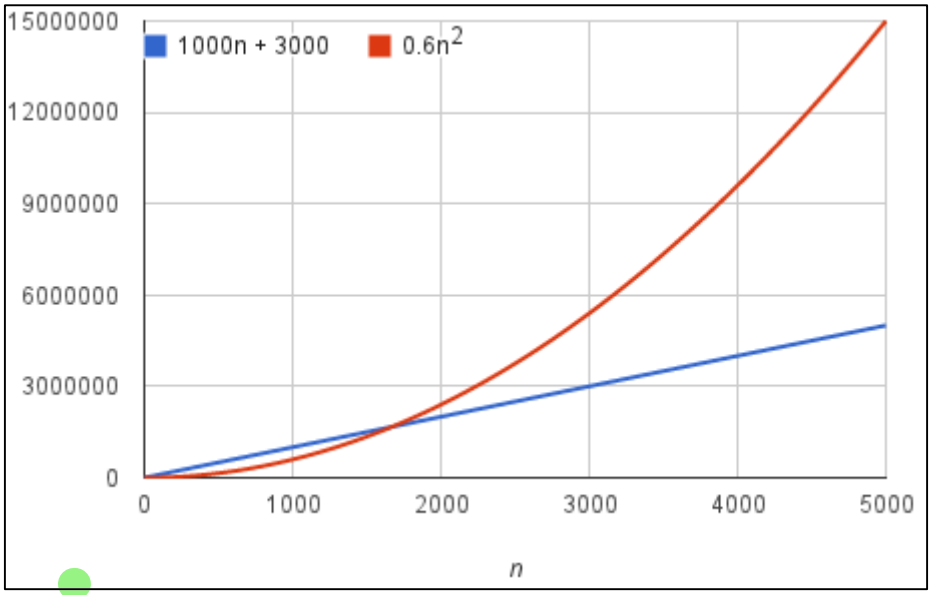
Let's think about the running time of an algorithm more carefully. We can use a combination of two ideas. **First**, we need to determine how long the algorithm takes, in terms of the size of its input. In case of linear search, number of comparisons are increased when number of elements are increased. So, we think about the running time of the algorithm as a function of the size of its input.

Second, an idea is that we must focus on how fast a function grows with the input size. We call this the **rate of growth** of the running time. To keep things manageable, we need to simplify the function to extract the most important part and ignore the less important parts. For example, suppose that an algorithm, running on an input of size n, takes $6\ n^2 + 100\ n + 300$. The $6\ n^2$ term becomes larger than the remaining terms $100\ n + 300$ When n becomes large enough, 20 in this case.



We would say that the running time of this algorithm grows as n^2, dropping the coefficient 6 and the remaining terms $100\ n + 300$.

It doesn't really matter what coefficients we use; as long as the running time $an^2 + bn + c$, for some numbers $a > 0$, b, and c, there will always be a value of n for which an^2 is greater than $bn + c$ and this difference increases as n increases. For example, here's a chart showing values of $0.6\ n^2 + 1000n+3000$, where we reduced the coefficient of n^2 by a factor of 10 and increased the other two constants by a factor of 10.



By dropping the less significant terms and the constant coefficients, we can focus on the important part of an algorithm's running time—its rate of growth—. When we drop the constant coefficients and the less significant terms, we use **asymptotic notation.**

Another example of growth rate:

The order of function growth is critical in evaluating the algorithm's performance. Assume the running times of two algorithms A and B are f(n) and g(n), respectively.

$f(n) = 2n^2 + 5$
$g(n) = 10n$

Here, n represents the size of the problem, while polynomials f(n) and g(n) represent the number of basic operations performed by algorithms A and B, respectively. Running time of both the functions for different input size is shown in following table:

n	1	2	3	4	5	6	7
f(n) - A	7	13	23	37	55	77	103
g(n) - B	10	20	30	40	50	60	70

Algorithm A may outperform algorithm B for small input sizes, however when input sizes become sufficiently big (in this example n = 5), f(n) always runs slower (performs more steps) than g(n). As a result, understanding the growth rate of functions is critical. Asymptotic notations describe the function's limiting behaviour.

What is asymptotic notation?

Asymptotic notations are a mathematical tool that can be used to determine the time or space complexity of an algorithm without having to implement it in a programming language. This measure is unaffected by machine-specific constants. It is a way of describing a significant part of the cost of the algorithm.

Machine-specific constants include the machine's hardware architecture, RAM, supported virtual memory, processor speed, available instruction set (RISC or CISC), and so on. The asymptotic notations examine algorithms that are not affected by any of these above factors.

When doing complexity analysis, the following assumptions are assumed.

Assumptions for finding the growth rate:

- The actual cost of operation is not considered.
- Abstract cost c is ignored: $O(c.\ n^2)$ reduces to $O(n^2)$
- Only leading term of a polynomial is considered: $O(n^3 + n)$ reduces to $O(n^3)$
- Drop multiplicative or divisive constant if any: $O(2n^2)$ and $O(n^2/2)$ both reduces to $O(n^2)$.

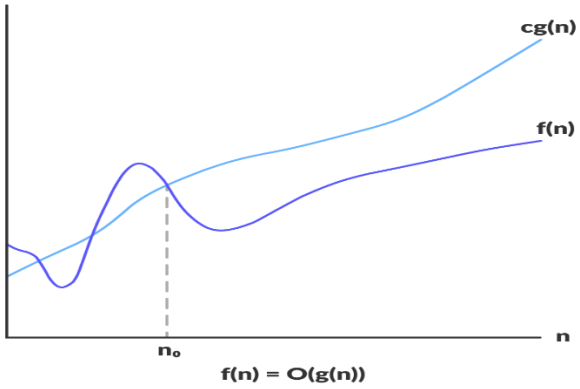
There are mainly three asymptotic notations:

- Big-O notation – upper bound of the function
- Omega notation - lower bound of the function
- Theta notation – average bound of the function.

2.2 Big- O notation:

We write f(n) = O(g(n)), If there are positive constants n_0 and c such that, to the right of n_0 the f(n) always lies on or below c*g(n).

O(g(n)) = { f(n) : There exist positive constant c and n_0 such that $0 \leq f(n) \leq c\ g(n)$, for all $n \geq n_0$}



For example:

f(n) = 2 n + 3

2 n + 3 < = ????

2 n + 3 < = 10 n for any value of n > no, no =1 and c = 10.

We can write anything like 7 n, 1000 n.

Simply, we can do the following.

2 n + 3 <= 2 n + 3 n

2 n + 3 < = 5 n , n >=1

Here, f(n) = 2 n + 3 and g(n) = n

So, f(n) = O(g(n))

Can we write the following?

2 n + 3 <= 2 n^2 + 3 n^2

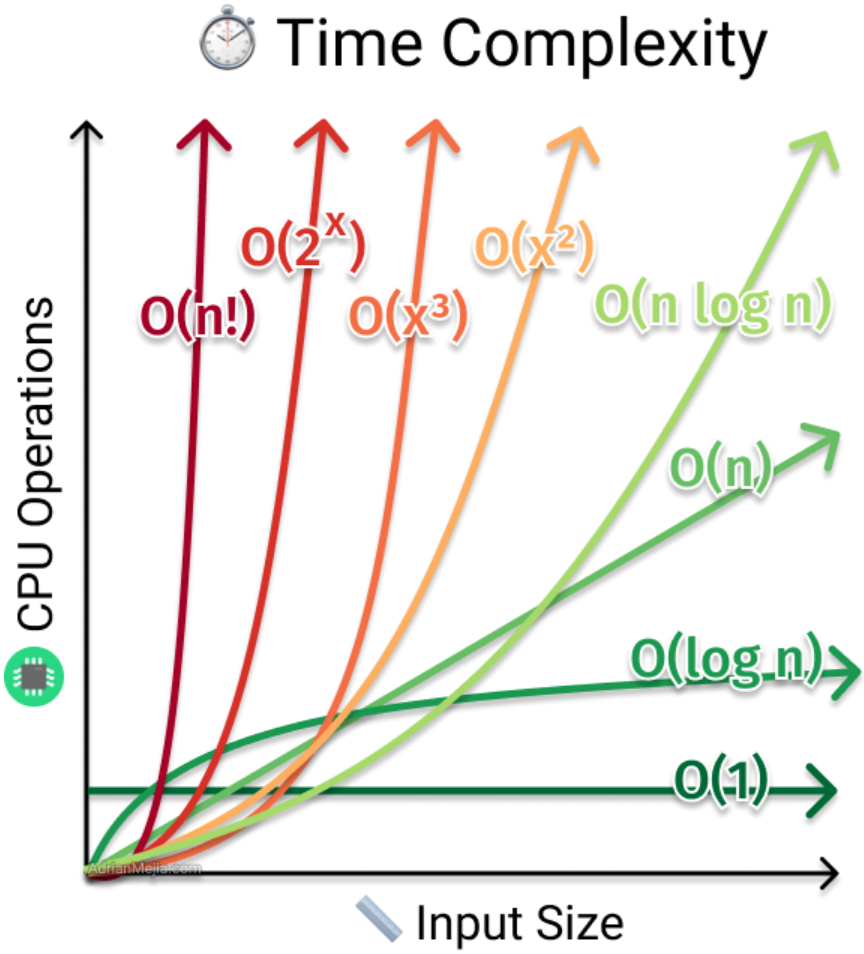
Yes, we can also write and f(n) = O(n^2)

f(n) = O(n) , f(n) = O(n^2)

Actually, f(n) belongs to linear class and all the classes on its right are upper bound of it. All functions on its left are lower bound. So, we can take any function on its right only.

We should try to write the closet function to f(n).

$$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots < 2^n < 3^n < n^n$$

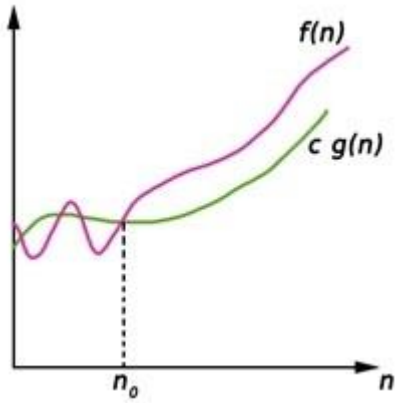


2.3 Big-Omega - Ω

Big-Omega (Ω) notation gives a lower bound for a function f(n) to within a constant factor.

We write f(n) = Ω(g(n)), If there are positive constants n_0 and c such that, to the right of n_0 the f(n) always lies on or above c*g(n).

Ω(g(n)) = { f(n) : There exist positive constant c and n_0 such that $0 \leq c\ g(n) \leq f(n)$, for all $n \geq n_0$}



f(n) = 2 n + 3

2 n + 3 >= 1 * n

c=1, g(n) = n

For all n >=1, f(n) = Ω(n)

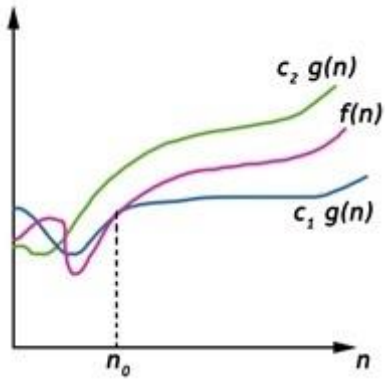
We can also say, f(n) = Ω(log n) or any lower bound on left side in the order of the class.

2.4 Theta - Θ

Big-Theta(Θ) notation gives bound for a function f(n) to within a constant factor.

We write f(n) = Θ(g(n)), If there are positive constants n_0 and c1 and c2 such that, to the right of n_0 the f(n) always lies between c1*g(n) and c2*g(n) inclusive.

Θ(g(n)) = {f(n) : There exist positive constant c1, c2 and n_0 such that $0 \leq c1\ g(n) \leq f(n) \leq c2\ g(n)$, for all $n \geq n_0$}



f(n) = 2 n + 3

1 * n <= 2 n + 3 <= 5 * n

c1 = 1, g(n) = n , c2 = 5, n >= 1

f(n) = Θ(n) but not Θ(n^2) or Θ(log n)

Don't misunderstand about worst case and best case with notation. Any notation can be used to represent the best case, worst case of an algorithm.

Examples – 1:

$f(n) = 2\ n^2 + 3\ n + 4$

$2\ n^2 + 3\ n + 4 <= 2\ n^2 + 3\ n^2 + 4\ n^2$

$2\ n^2 + 3\ n + 4 <= 9\ n^2$ for any value of $n >= 1$, $c = 9$, $n_0 = 1$

So, $f(n) = O(n^2)$

Example – 2:

$F(n) = 2\ n^2 + 3\ n + 4$

$2\ n^2 + 3\ n + 4 >= 1 * n^2$

$f(n) = \Omega(n^2)$

Example-3:

$f(n) = 2\ n^2 + 3\ n + 4$

$1 * n^2 <= 2\ n^2 + 3\ n + 4 <= 9\ n^2$

$O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$

Example-4:

$f(n) = n^2 \log n + n$

$1 * n^2 \log n <= n^2 \log n + n <= 10\ n^2 \log n$

$O(n^2 \log n)$, $\Omega(n^2 \log n)$, $\Theta(n^2 \log n)$

Example-5:

$f(n) = n\ !$

$1 <= n\ ! <= n^n$

Upper bound - $O(n^n)$, lower bound - $\Omega(1)$

There is not average bound for n!.

When it is not possible to find ɵ then Ω and O are used to find lower and upper bound respectively.

We can not put n^{10} as lower bound and n^{14} as upper bound.

Example-6:

f(n) = log n !

log (1 * 1 * 1 ..) <= log (1 * 2 * 3 * 4 *…n) <= log (n * n * n * n) (log n^n) means n log n

Upper bound - O(n log n) , lower bound - Ω(1)

2.5 General properties of asymptotic notations

General property:

If f(n) = O(g(n)) then a * f(n) = O(g(n)) where a is some constant value.

f(n) = 2 n^2 + 5 is O(n^2)

7 * (2 n^2 + 5) = O(n^2)

Also, similar for Ω and Θ

If F(n) = Θ(g(n)) then a * f(n) = Θ(g(n))

If F(n) = Ω(g(n)) then a * f(n) = Ω(g(n))

Reflexive property:

If f(n) is given, then f(n) is O(f(n)).

e.g f(n) = n^2 then O(n^2)

Any function is a lower bound of itself.

Transitive property:

If f(n) is O(g(n)) and g(n) is O(h(n)) then f(n) is O(h(n)).

Example: n is O(n^2) and n^2 is O(n^3) then n is O(n^3)

Symmetric:

Only for ɵ notation only

If f(n) is ɵ(g(n)) then g(n) is ɵ(f(n))

f(n) = n^2 and g(n) = n^2

Transpose symmetric:

If f(n) is O(g(n)) then g(n) is Ω(f(n))

f(n) = n

g(n) = n^2

f(n) = O(n^2)

g(n) = Ω(n)

Another property:

If f(n) = O(g(n))

f(n) = Ω(g(n))

Then f(n) = ɵ(g(n))

Means c1 * g(n) <= f(n) <= c2 * g(n)

Another property:

If f(n) = O(g(n))

And d(n) = O(h(n))

Then f(n) + d(n) = O(max(g(n),h(n)))

Another property:

If f(n) = O(g(n))

And d(n) = O(h(n))

Then f(n) * d(n) = O(g(n) * h(n))

2.6 Comparison of functions

Value of n	n^2	n^3
2	4	8
3	9	81

Apply log on both the side.

$\log n^2 < \log n^3$

2 log n < 3 log n so, 3 log is bigger.

What is logarithms functions? (Watch this video for more details - Relation and Function 04 | All about Logarithmic Function | Class 11 | IIT JEE - YouTube)

Inverse functions of exponentiation functions are called logarithms functions.

$F(n) = 2^3 = 8$

2 is called base, 3 is called exponent and 8 is called result.

Now, same thing can be represented by logarithm as follows:

$Log_2 8 = 3$, here base remains the same, result becomes input to log and exponent becomes result.

Exponent and result are changed in exponentiation and logarithm.

1 $Y = 2^x$

2 $X = \log_2 Y$

In the above, for the 1st function give the input 2, 4 and 16 etc.. and its corresponding output gives as input to the 2nd function so, the result will be those that is given as input to the 1st function.

$\log_b x$ – it is read as log x with base b.

$\log_2 4 = 2$

$\log_2 32 = 5$

$\log_2 1/16 = -4$ $2^? = 1/16 = 2^{-4}$

$\log_{1/4} ½ = ½$

$\log_2 1 = 0$

Output of the logarithmic functions can be positive, zero or negative.

$\log_2 0$ = Not defined

$\log_2 -4$ = Not defined

Input of logarithm functions must not be zero or negative.

Base of logarithm:

$\log_2 4 = 2$

$\log_1 4$ = not defined

$\log_{-2} 4$ = not defined

base of log must not be one or negative.

$F(n) = \log_a x$ where x > 0, output (-infinite,+infinite), a > 0 and a != 1

e = 2.718 – Euler's number (lnx) – $\log_e x$ e = 1/1! + ½! + 1/3! + …+ infinite

Important log formulas:

1. $\log_a x = y$ then $a^y = x$
2. log ab = log a + log b
3. log a/b = log a – log b
4. $\log^b a = b \log a$
5. $a \log^b c = b \log_c a$
6. $a^b = n$ then $b = \log_a n$
7. $\log_{\text{any number for the base}} 1 = 0$

$f(n) = n^2 \log n$ and $g(n) = n (\log n)^{10}$

apply log on both the sides

$\log(n^2 \log n)$ and $\log(n (\log n)^{10})$

$\log n^2 + \text{loglog } n$ and $\log n + \log(\log n)^{10}$

2 log n + loglog n and log n + 10 loglog n

So, f(n) is bigger.

Example:

$F(n) = 3n^{\sqrt{n}}$ and $g(n) = 2^{\sqrt{n} \log n}$

Apply above formula (3) in g(n)

$3n^{\sqrt{n}}$ $2^{log^{n^{\sqrt{n}}}}$ – apply formula no. (4) in this equation

$3n^{\sqrt{n}}$ $(n^{\sqrt{n}})^{log 2^2}$

$3n^{\sqrt{n}}$ > $(n^{\sqrt{n}})^1$

$2 n^2 > n^2$ – this is correct but asymptotically they are same $O(n^2)$

Example:

F(n) = $n^{\log n}$ and g(n) = $2^{\sqrt{n}}$

Apply log on both the sides

log $n^{\log n}$	log $2^{\sqrt{n}}$
log n log n	$\sqrt{n}$ $\log_2 2$ where $\log_2 2 = 1$
$\log^2 n$	$\sqrt{n}$

if you unable to judge then again apply log

$\log\log^2 n$		log $n^{1/2}$
2 loglog n	<	½ log n

Example:

F(n) = 2^n and g(n) = 2^{2n}

Apply log

log 2^n		log 2^{2n}
n $\log_2 2$		2 n $\log_2 2$
n	<	2 n

2.7 Best case, Worst case and Average case analysis

There are three cases to analyze an algorithm: (Types of analysis of algorithms)

1. Worst Case Analysis (Mostly used)

In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x) is not present in the array. When x is not present, the search() function compares it with all the elements of arr[] one by one. Therefore, the worst-case time complexity of the linear search would be **O(n)**.

2. Best Case Analysis (Very Rarely used)

In the best-case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed. In the linear search problem, the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So time complexity in the best case would be Ω(1) or O(1).

3. Average Case Analysis (Rarely used)

In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs. We must know (or predict) the distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in the array). So, we sum all the cases and divide the sum by n which is O(n).

Case analysis is identifying instances for which the algorithm takes the longest or shortest time to complete (i.e., takes the greatest number of steps), then formulating a growth function using this.

Let's understand them using two examples to find their best, worst and average case.

Examples:

1. Linear Search
2. Binary Search

Linear Search (Sequential Search):

0	1	2	3	4	5	6	7	8	9
3	90	56	43	67	12	35	9	78	52

How searching is done?

To search the element in the list, each element is inspected in the list one by one until we get the required element.

Best case: It is a case when algorithm takes minimum time. When the searched key is present on the index 0 (first) then it is best case. Time taken by this is constant time – B(n) = O(1).

Worst case: When algorithm takes maximum time. When searched key is present at last index. Time taken is W(n) – O(n)

Average case: In remaining cases, it may be found on any index.

= all possible case time / no. of cases

= 1 + 2 + 3 + 4 + .. + n / n

= n (n+1) / 2 n

= (n+1)/2

= O(n)

Time taken is A(n) – O(n)

B(n) – 1

B(n) – O(1)

B(n) - Ω(1)

B(n) – Θ(1)

For any constant function, we can write any notation.

W(n) – n

W(n) – O(n)

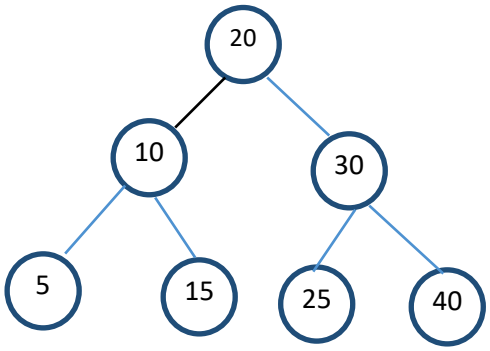
W(n) - Ω(n)

W(n) – Θ(n)

Any notation can be used to show best, average and worst case.

Binary Search:

For any node elements less than are on left and elements greater than or equal to it on its right side.



Number of searches required is equal to height of the binary tree which is – $\log_2 n$. where n is number of elements in the tree.

Best case: The element to be searched for is present in the root. The time taken is constant. B(n) – O(1).

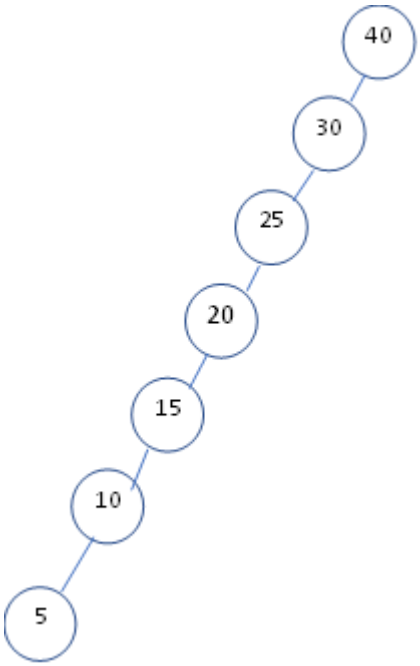
Worst case: The element to be searched is present in the leaf node. So, the time taken is proportional to height of the binary tree which is $\log_2 n$. W(n) – $\log_2 n$.

Average case: It is difficult to find so, mostly it is similar to worst case.

Here, we have balanced tree but if the tree of the following type for same elements where tree is left skewed and height is n so, W(n) – O(n).

In worst case, minimum time is W(n) – $O(\log_2 n)$ and maximum time is W(n) – O(n)

This type of minimum and maximum worst case is not possible for all types of algorithms.



2.8 Frequency count method:

A. Finding sum of array elements

```
Algorithm sum(A[],n)
{
s=0;                    - 1
for(i=0;i<n;i++)        - i=0 # 1, i < n  #  n + 1, i++ #  n
  s = s + A[i];         - n
return s;               - 1
}
```

One unit of time for each statement.

Time complexity:

f(n) = 2 n + 3 which is O(n)

Space complexity:

A # n, s # 1, n # 1

So, S(n) = n + 2 which is O(n)

B. Finding sum of two square matrices.

```
Algorithm mat_add(A[], B[], n)
{
for(i=0;i<n;i++)                - n + 1
  for(j=0;j<n;j++)              - n * (n + 1)
      C[i,j]= A[i,j]+B[i.j];    - n * n
}
```

Time complexity:

$F(n) = 2 n^2 + 2 n + 1$

Which is $O(n^2)$

Space complexity:

A, B and C are n * n and i , j and n are scaler variables.

$S(n) = 3 n^2 + 3$ which is $O(n^2)$

C. Finding multiplication of two matrices.

Algorithm multiply(A, B, C, n)

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

Time complexity:

$F(n) = 2n^3 + 3n^2 + n + 1$ which is $O(n^3)$

Space complexity:

A, B, C – n * n and i, j, k, n – 1

$S(n) = O(n^2)$

Time complexity examples:

- **Example-1**

```
for(i=0;i<n;i+=2) – n /2
  statement;
```

f(n) = n / 2 which is O(n)

- **Example-2**

```
for(i=0;i<n;i+=20) – n /20
  statement;
```

f(n) = n / 20 which is O(n)

- **Example-3**

```
for(i=0;i<n;i++)
   for(j=0;j<i;j++)
       Statement;
```

i	j	No. of times
0	~~0~~	0
1	0 ~~1~~	1
2	0 1 ~~2~~	2
3	0 1 2 ~~3~~	3
……		
n	--	n

F(n) = 1 + 2 + 3 + .. + n

= n (n+1) / 2

= $O(n^2)$

- **Example-4**

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

i	P
1	0 + 1
2	1 + 2
3	1 + 2 + 3
4	1 + 2 + 3 + 4
5	1 + 2 + 3 + 4 + 5
k times	
	1 + 2 + 3 + 4 + 5 + .. + k

The loop stops when,

p > n (1)

p is as follows,

p = 1 + 2 + 3 + .. + k

p = k (k+1) /2

$p = k^2$ (2) //we estimate it to k^2

put the value of p in equation (1)

$k^2 > n$

$k > \sqrt{n}$ which is $O(\sqrt{n})$

- **Example-5**

```
for(i=1;i<n;i*=2)
  statements;
```

$i - 2, 2^2, 2^3, 2^4 \ldots 2^k$

The loop stops when,

$i >= n$ (1)

$i = 2^k$ (2)

$2^k >= n$

$2^k = n$

$k = \log_2 n$ which is $O(\log_2 n)$

When loops get multiplied then it takes log n time.

log may give you float value so, we have to consider the ceil value of the log result.

n = 8 so, log 8 = 3, log 10 = 3.2 so, ceil(3.2) = 4 means we have to take $\lceil log \rceil$

- **Example-6**

```
for(i=n;i>=1;i/=2)
  statements;
```

$i - n/2, n/2^2, n/2^3 \ldots n/2^k$

It stops when $i < 1$

If we equate them then,

$i = n/2^k$

$n/2^k = 1$

$n = 2^k$

$k = \log_2 n$ which is $O(\log_2 n)$

- **Example-7**

```
for(i=0;i * i < n; i++)
  statements;
```

It terminates when i * i >= n

$i^2 = n$

i = $\sqrt{n}$ which is O($\sqrt{n}$)

- **Example-8**

```
for(i=0;i<n;i+=1)
  statement;              - n
for(i=0;i<n;i+=1)
  statement;              - n
```

f(n) = 2n which is O(n)

- **Example-9**

```
p=0
for(i=1;i<n;i=i*2)              - log n
  p++:
for(j=0;j<p;j*=2)               - log p
  Statement;
```

Here, first loop is iterated log n times and value of p becomes log n.

Second loop is iterated log p times and replace the value of p as log n.

= log p = log log n

F(n) = loglog n which is O(loglog n)

- **Example-10**

```
for(i=0;i<n;i++)                    -  n
        for(j=0;j<n;j*=2)           - log2 n * n
          Statement;                - log2 n * n
```

F(n) = n log n which is O(n $\log_2$ n)

Summary of some common looping statements' time complexity:

for(i=0;i<n;i++)	O(n)
for(i=0;i<n;i+=2)	n/2 – O(n)
for(i=n;i>1;i--)	O(n)
for(i=0;i<n;i=i*2)	O($\log_2$ n)
for(i=0;i<n;i=i*3)	O($\log_3$ n)
for(i=n;i>1;i=i/2)	O($\log_2$ n)

2.9 Analysis of if and while loop

```
i=1                     - 1
while(i<n)              n + 1
{
statement;              n
i++;                    n
}
```

F(n) = 3n +2 which is O(n)

Almost while and for are similar but we prefer to use for statement.

```
i =1
k=1
while(k < n)
{
statement;
k = k + i;
i++;
}
```

i	k
1	1
2	1 + 1
3	1 + 1 + 2

4	1 + 1 + 2 + 3
5	1+ 1 + 2 + 3 + 4
m times	2 + 2 + 3 + 4 ---- + m

It stops when k > = n

If k repeats m times then m (m+1) / 2

m(m+1) / 2 = n

$m^2 = n$

m = $\sqrt{n}$ which is O($\sqrt{n}$)

if statement inside loop:

```
while(m!=n)
{
if(m > n)
  m = m – n;
else
  n = n – m;
}
```

Take m = 16 and n = 2…

Take m = 4 and n = 2

Maximum time is n / 2 so, O(n) and minimum time is O(1)

```
Algorithm Test(n)
{
if(n < 5)
   Statement;
else
 {
   for(i=0;i<n;i++)
      Statements;
  }
}
```

Statements are executed based on the condition so, best case is O(1) and worst case is O(n)

2.10 Types/classes of time/space functions

Function	Types of class
O(1) Any of the following: F(n) = 3 F(n) = 1000 or any value	Constant
O(log n)	Logarithmic
O(n) Any of the following: F(n) = n + 3 F(n) = 4000 n + 2 n + 4 F(n) = n/2000 + 3	Linear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(2^n)$, $O(3^n)$ or $O(n^n)$	Exponential

Classes of function in order of their weightage:

$$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots < 2^n < 3^n < n^n$$

Let's see in the following example, how exponentiation functions grow much faster than any other functions when n grows.

$\log_2 n$	n	n^2	2^n
0	1	1	2
1	2	4	4
1.3	3	9	8
2	4	16	16
3	8	64	256
. . .	. . .	. . .	. . .
4.3	20	400	10,48,576

After some value of n , 2^n is much greater than all others. So, any power to n is less than 2^n

CHAPTER – 3 Recurrence Relation

3.1 Recurrence Relation

A **recurrence relation** is an equation which represents a sequence based on some rule. It helps in finding the subsequent term (next term) dependent upon the preceding term (previous term). If we know the previous term in a given series, then we can easily determine the next term.

There are different methods to solve the recurrence relations:

1. **Substitution method**
2. **Iterative method**
3. **Recursion tree method**
4. **Master theorem**
5. **Change of variable**
6. **Characteristic equation method**

Substitution Method: We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.

Recurrence Tree Method: In this method, we draw a recurrence tree and calculate the time taken by every level of tree. Finally, we sum up the work done at all levels. To draw the recurrence tree, we start from the given recurrence and keep drawing till we find a pattern among levels. The pattern is typically an arithmetic or geometric series.

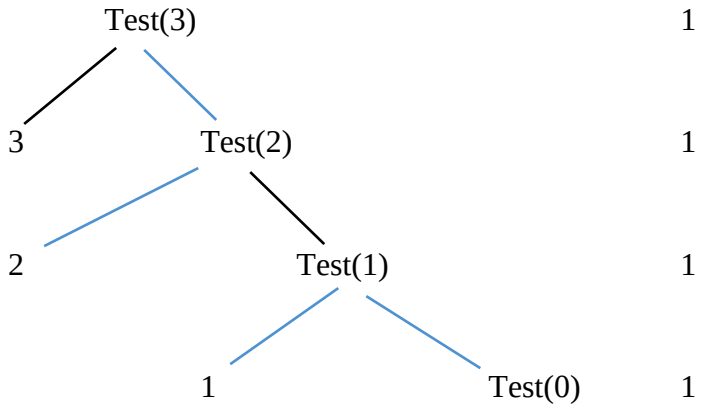
Simple Recursive function:

```
void Test(int)
{
    if(n>0)
    {
        printf("%d",n);
        Test(n-1);
    }
}
```

The major work done is printing of value of n.

Recursive/Recursion tree approach

Calling of Test(3)



Each call prints the value of n and printf statement executes one time per call. Functions calls total 4 times for Test(3) so, for Test(n) it will be called n + 1 times.

Time complexity is = O(n)

How to find recurrence relation for the above function:

```
void Test(int)   - T(n)
{
        if(n>0)                 - 1  // we may add or not. It will not make any
                                     difference.
      {
            printf("%d",n);     - 1
            Test(n-1);          - T(n-1)
      }
}
```

$$T(n) = \begin{cases} 1 & if\ n = 0 \\ T(n-1) + 1 & if\ n > 0 \end{cases}$$

Backward substation method:

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

Now, we find T(n-1) as follows,

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

Put T(n-1) in equation (1)

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

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

Now, find T(n-2) as follows,

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

Put T(n-2) in equation (2)

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

T(n) = T(n-3) + 3 - (3)

If we repeat this k times,

T(n) = T(n-k) + k - (4)

Assume n-k=0 so, n=k

T(n) = T(n-n) + n - (5)

T(n) = T(0) + n - (6)

T(n) = 1 + n - (7)

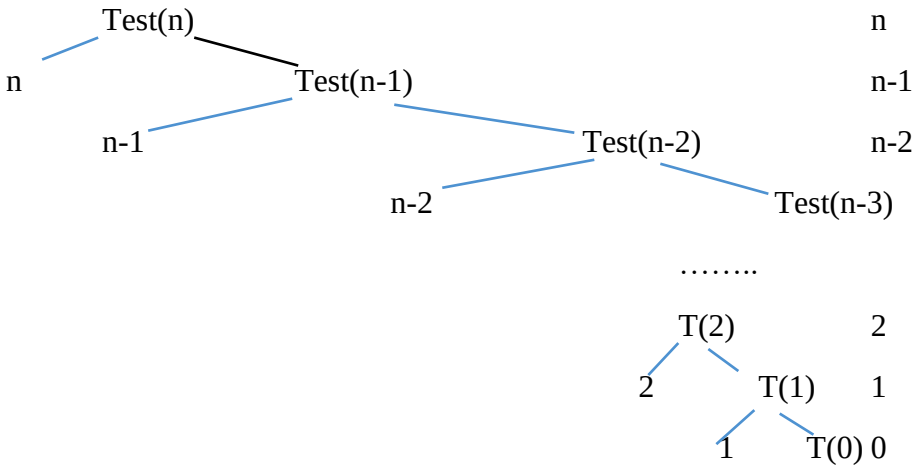
T(n) = O(n)

Another function

```
void Test(int n)                    - T(n)
{
        if(n>0)                     - 1
        {
                for(i=0;i<n;i++)    - n + 1
                {
                printf("%d",n);     - n
                }
                Test(n-1);          - T(n-1)
```

```
    }
}
```

Recursion tree approach



$0 + 1 + 2 + \ldots. + n- 1 + n = n (n + 1) / 2 = O(n^2)$

Another way of recursion tree approach:

Substitution method

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

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

Take the asymptotic notation of 2n + 1 which is O(n).

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

$$T(n) = \begin{cases} 1 & if\ n = 0 \\ T(n-1) + n & if\ n > 0 \end{cases}$$

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

Now we find T(n-1) as follows,

T(n-1) = T(n-2) + n – 1

Put T(n-1) in equation (1)

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

T(n) = T(n-2) + (n – 1) + n (2)

Now we find T(n-2) as follows,

T(n-2) = T(n-3) + n – 2

Put T(n-2) in equation (2)

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

$T(n) = T(n-3) + (n-2) + (n-1) + n$ (3)

Repeat this k times.

$T(n) = T(n-k) + (n-(k-1)) + (n-(k-2)) .. + n-1 + n$ (4)

Assume that n – k = 0 so, n = k

$T(n) = T(n-n) + (n-(n-1)) + (n-(n-2)) .. + n-1 + n$ (5)

$T(n) = T(0) + 1 + 2 + \ldots. + n-1 + n$ (6)

$T(n) = 1 + 1 + 2 + \ldots + n-1 + n$

$T(n) = 1 + n\ (n+1) / 2$

$T(n) = n^2$

$T(n) = n^2$

Which is $O(n^2)$

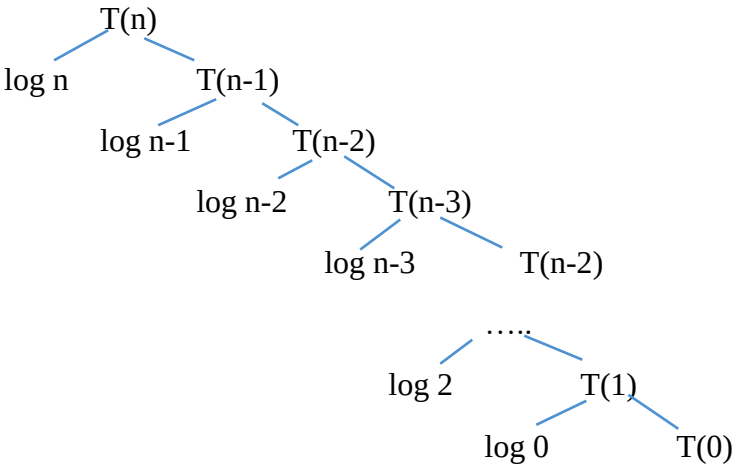
Another function

```
void Test(int)                          - T(n)
{
        if(n>0)                         - 1
        {
                for(i=1;i<n;i=i*2)      // this is repeated in power of two
                {
                printf("%d",n);         - log2 n
                }
                Test(n-1);              - T(n-1)
        }
}
```

$T(n) = T(n-1) + \log n$

$$T(n) = \begin{cases} 1 & if\ n = 0 \\ T(n-1) + \log n & if\ n > 0 \end{cases}$$

Recursion tree method:



= log n + log n-1 + log n-2 + log 1 + log 0

= log (n * n-1 * n-2 * 1 * 1)

= log n!

There is not a tight bound for n! but upper bound is n^n

= log n^n

= O(n log n)

Substitution method:

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

= [T(n-2) + log n-1] + log n

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

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

= T(n-3) + log n-2 + log n-1 + log n (3)

Repeat this k times…

= T(n-k) + log n-(k-1) + log n-(k-2) + log n-(k-3) … + log n

Assume n – k = 0 so, n = k

= T(n-n) + log 1 + log 2 + log 3 … + log n-1 + log n

= 1 + log 1 + log 2 + log 3 … + log n-1 + log n

= log n!

= log n^n

= O(n log n)

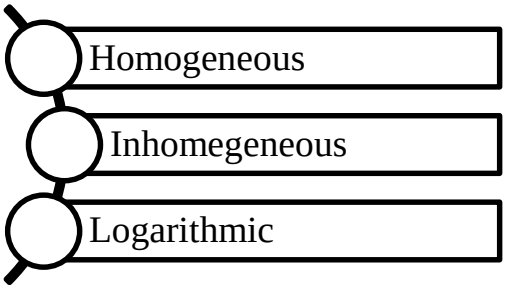
General observation for recurrence relation of the above types:

Recurrence	**Order**
T(n) = T(n-1) + 1	O(n)
T(n) = T(n-1) + n	O(n^2)
T(n) = T(n-1) + log n	O(n log n)
T(n) = T(n-1) + n^2	O(n^3)
T(n) = T(n-2) + 1	O(n)
T(n) = T(n-100) + n	O(n^2)
Whatever it is T(n – 200) or T(n – 1000) etc…	
T(n) = 2 T(n-1) + 1	?? not same as above

Important note:

In the above table, second term is multiplied with n to get the order.

Types of Recurrences:



Homogeneous recurrence:

If the recurrence is equated to zero and if it contains all terms in homogeneous form, then it is called homogeneous recurrence.

For example:

$$t_n + a\, t_{n-1} + b\, t_{n-2} + \cdots . + n\, t_{n-k} = 0$$

Inhomogeneous recurrence:

If the sum of the linear terms of the equation is not equal to zero, then it is called inhomogeneous recurrence.

The general format is as follows:

$$a_0 t_n + a_1 t_{n-1} + a_2 t_{n-2} + a_n t_{n-k} = b^n p(n)$$

Example:

$$t_{n+3} + 6\, t_{n+2} + 8\, t_{n+1} + 5 t_n = 2^n$$

Logarithmic recurrence:

Divide and conquer techniques uses logarithmic recurrence.

The general format is:

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

Where a and b are constant a > =1 and b > 1. f(n) is some function.

Examples:

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

Iteration method:

Recurrence is expanded as summation of terms, then summation provides the result.

This method is known as try back substituting until you know what is going on.

Substitution method:

We start the method by a guess of the solution and then prove it by induction.

Example 1:

Suppose a recurrence relation given as follows,

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

Solution:

Given recurrence is

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

But we assume that T(n) = O(log n) (2)

We have to prove that T(n) <= c * log n where c is constant. (3)

Compute T(n/2) from equation (2) as follow

$$T(\frac{n}{2}) \geq c\ (\log\frac{n}{2})$$

Now, substitute the above value of T(n/2) in equation (1)

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

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

$$T(n) \leq\ c * (\ log_2 n - log_2 2)\ + 1$$

$$T(n) \leq\ c * (\ log_2 n - 1)\ + 1$$

$$T(n) \leq \ c \ log_2 n - c \ + 1$$

For c = 1

$$T(n) \leq \ 1 \ log_2 n - 1 \ + 1$$

$$T(n) \leq \ 1 \ log_2 n$$

$$T(n) \leq \ log_2 n$$

Hence it is proved, T(n) = O (log n)

Example: 2

Suppose a recurrence relation given as follows,

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

Then show that it is asymptotically bounded by **Ω** (n logn).

Solution:

Given recurrence is

$$T(n) = 2\,T\left(\frac{n}{2}\right) + n \qquad (1)$$

But we assume that T(n) = **Ω** (n logn) (2)

We have to prove that T(n) >= c * n logn where c is constant. (3)

Note: We can also assume T(n) = O(n logn) in that case 0 <= T(n) <= c * n logn

Compute T(n/2) from equation (2) as follow

$$T(\frac{n}{2}) \geq c\ (\frac{n}{2}\log\frac{n}{2})$$

Now, substitute the above value of T(n/2) in equation (1)

$$T(n) \geq \ 2\ (\ c * \ (\frac{n}{2}\log\frac{n}{2}))\ + n$$

$$T(n) \geq \cancel{2}\ \ c * (\frac{n}{\cancel{2}} \log \frac{n}{2}) + n$$

$$T(n) \geq \ \ c * n * \log \frac{n}{2} + n$$

$$T(n) \geq \ \ c * n * (\ log_2 n - \ log_2 2) \ + n$$

$$T(n) \geq \ \ c * n * (\ log_2 n - 1) \ + n$$

$$T(n) \geq \ \ c\ n\ log_2 n - c\ n \ + n$$

For c = 1

$$T(n) \geq \ \ 1\ n\ log_2 n - 1 \ \ n \ + n$$

$$T(n) \geq \ \ 1\ n\ log_2 n - 1 \ \cancel{n} \ + \cancel{n}$$

$$T(n) \geq \ \ \ n\ log_2 n$$

Hence it is proved, T(n) = Ω (n logn)

Times it takes.

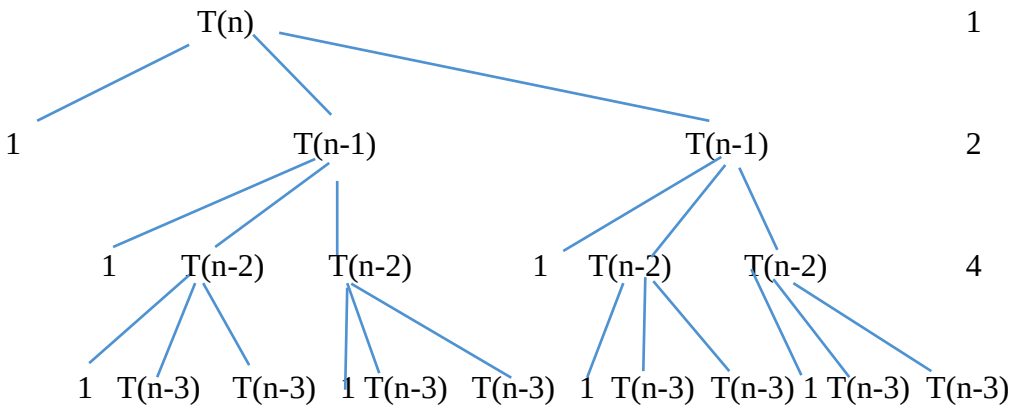
```
void Test(int)                          T(n)
{
    if(n>0)
    {
        printf("%d",,n);                1
        Test(n-1);                      T(n-1)
        Test(n-1);                      T(n-1)
    }
}
```

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

Recurrence relation is

$$T(n) = \begin{cases} 1 & if\ n = 0 \\ 2\,T(n-1) + 1 & if\ n > 0 \end{cases}$$

Recursion tree approach :



$1 + 2 + 2^2 + 2^3 + 2^4 \ldots + 2^k = 2^{k+1} - 1$ (GP Series)

Assume n-k=0 so, n=k

$= 2^{n+1} - 1$

$= O(2^n)$

For reference of GP series

$a + ar + ar^2 + ar^3 + \ldots + ar^k = a(r^{k+1} - 1) / r - 1$

in above series

a = 1, r = 2

Substitution method:

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

Now find T(n-1) as follows,

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

Put T(n-1) in equation (1)

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

$= 2^2\ T(n-2) + 2 + 1$ (2)

Now find T(n-2) as follows,

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

$T(n-2) = 2\ T(n-3) + 1$

Put T(n-2) in equation (2)

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

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

If we repeat k times..

$= 2^k\ T(n-k) + .. + 2^{k-1} + 2^{k-2} + 2^2 + 2 + 1$ (4)

Assume n – k = 0 so, n = k

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

$= 2^n\ 1 + .. + 2^{n-1} + 2^{n-2} + 2^2 + 2 + 1$

$= 2^{n+1} + 1$

$= O(2^n)$

3.2 Master Theorem

Master's Theorem is the best method to quickly find the algorithm's time complexity from its recurrence relation. This theorem can be applied to decreasing as well as dividing functions.

Recursive functions call themselves in their body. It might get complex if we start calculating its time complexity function by other commonly used simpler methods. Master's method is the most useful and easy method to compute the time complexity function of recurrence relations.

We can apply Master's Theorem only for:

1. Dividing Functions
2. Decreasing Functions

Master theorem for decreasing functions:

Recurrence	**Order**
T(n) = T(n-1) + 1	O(n)
T(n) = T(n-1) + n	$O(n^2)$
T(n) = T(n-1) + log n	O(n log n)
T(n) = T(n-1) + n^2	$O(n^3)$
T(n) = T(n-2) + 1	O(n)
T(n) = T(n-100) + n	$O(n^2)$
T(n) = 2 T(n-1) + 1	$O(2^n)$
T(n) = 3 T(n-1) + 1	$O(3^n)$
T(n) = 3 T(n-1) + n	$O(n3^n)$

T(n) = a T(n-b) + f(n)

a is number of sub problems in recursion and it is >= 1

n-b − size of the sub problems based on the assumption that all sub-problems are of the same size so, b > 0.

f(n) represents the cost of work done outside of recursion and it is f(n) = $O(n^k)$ and 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:

- Case:1 if a = 1, T(n) = O (n^{k+1})
- Case 2 if a > 1, T(n) = O ($a^{n/b} * n^k$)
- Case 3 if a < 1, T(n) = O (n^k)

Examples of different cases:

Case:1 if a = 1, T(n) = O (n^{k+1})

Example: 1)

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

In this problem, a = 1, b = 1 and $f(n) = O(n^k) = n^2$, giving us k = 2.
Since a = 1, case 1 must be applied for this equation.
To calculate, $T(n) = O(n^{k+1})$
$= n^{2+1}$
$= n^3$
Therefore, $T(n) = O(n^3)$ is the tight bound for this equation.

Case 2: if $a > 1$, $T(n) = O(a^{n/b} * n^k)$

Example: 1)

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

In this problem, a = 2, b = 1 and $f(n) = O(n^k) = n$, giving us k = 1.
Since a > 1, case 2 must be applied for this equation.
To calculate, $T(n) = O(a^{n/b} * n^k)$
$= O(2^{n/1} * n^1)$
$= O(n2^n)$
Therefore, $T(n) = O(n2^n)$ is the tight bound for this equation.

Case 3: if $a < 1$, $T(n) = O(n^k)$

Example: 1)

$T(n) = n^4$

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

More examples:

Examples 1:

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

a = 1, b = 1, k =2

Therefore, $T(n) = O(n^{k+1}) = O(n^3)$ is the tight bound for this equation

Example 2:

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

a = 3, b = 1, k =0

To calculate, $T(n) = O(a^{n/b} * n^k)$

$= O(3^{n/1} * n^0)$
$= O(3^n)$
Therefore, $T(n) = O(3^n)$ is the tight bound for this equation

Example 3:

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

This recurrence can't be solved using above method
since function is not of form **T(n) = aT(n-b) + f(n).**

Example 4:

Fibonacci series:

$$T(n) = \begin{cases} n & if\ n = 0\ or\ n = 1 \\ T(n-1) + T(n-2) & if\ n \geq 2 \end{cases}$$

Let $T(n-1) \approx T(n-2)$
$T(n) = 2T(n-1) + c$ // c is constant cost for addition operation
where, $f(n) = O(1)$
$\therefore k=0, a=2, b=1;$
$T(n) = O(n^0 2^{n/1})$
$= O(2^n)$

Example 5:

Factorial of number:

```
factorial(n):
   if n is 0                      - 1
      return 1
   return n * factorial(n-1)    - 1 + 1 + T(n-1)
```

1 unit cost for comparison, 1 multiplication and 1 for subtraction.

$T(n) = T(n-1) + 3$

Solve this equation using subtraction, recursion tree and mater theorem method.

$T(n) = O(n)$.

3.3 Recurrence of dividing functions

```
void Test(int)                          - T(n)
{
    if(n>0)                             - 1  // we may add or not. It will not make any
                                              difference.
    {
        printf("%d",n);                 - 1
        Test(n/2);                      - T(n/2)
    }
}
```

Different decreasing value – n , n-1, n / 2 , square root of n

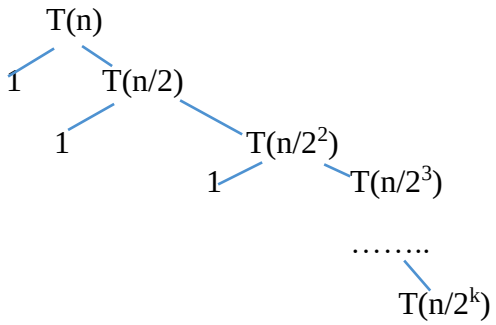
There are different functions for each.

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

$$T(n) = \begin{cases} 1 & if\ n = 1 \\ T\left(\frac{n}{2}\right) + 1 & if\ n > 1 \end{cases}$$

Note: In dividing function, we use n = 1 as base case not n = 0 as we took in decreasing function.

Recursion tree method:



$n/2^k = 1$

$n = 2^k$

$k = O(\log_2 n)$ - ($a^b = c \rightarrow b = \log_a c$)

Using substitution method

$T(n) = T(n/2) + 1$ (1)

Now find T(n/2) as follows:

$T(n/2) = T(n/2^2) + 1$

Replace T(n/2) in to (1)

$= T(n/2^2) + 1 + 1$

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

Now find $T(n/2^2)$ as follows:

$T(n/2^2) = T(n/2* 2^2) + 1$

Replace $T(n/2^2)$ in equation (2)

$= T(n/2^3) + 1 + 2$ (3)

Similarly, we get the next terms as follows,

$= T(n/2^4) + 4$ (4)

Repeat this k number of times

$= T(n/2^k) + k$ (5)

Assume $n / 2^k = 1$ so, $k = \log n$

$= T(1) + \log n$ (6)

$= 1 + \log n$

= **O(log n)**

Another dividing function

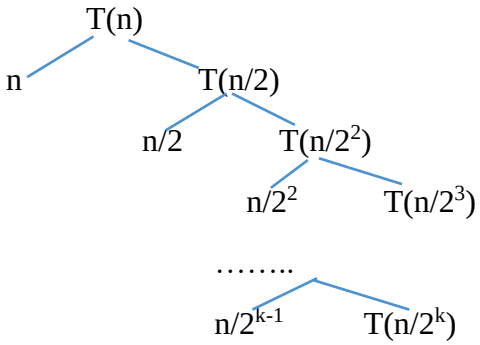
```
void Test(int)                  - T(n)
{
        if(n>0)                 - 1
        {
                for(i=0;i<n;i++)
                {
                printf("%d",n);         - n
                }
                Test(n/2);              - T(n/2)
```

}

}

$$T(n) = \begin{cases} 1 & if\ n = 1 \\ T\left(\frac{n}{2}\right) + n & if\ n > 1 \end{cases}$$

Recursion tree method

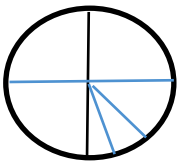


Add the all the above so, we get the following:

= n + n/2 + n/2^2 + n/2^3 + … + n/2^{k-1} + n/2^k

= n (1 + 1/2 + 1 / 2^2 + 1 / 2^3 + 1 / 2^k)

= n $(1 + \sum_{i=0}^{k} \frac{1}{2^i})$, $\sum_{i=0}^{k} \frac{1}{2^i} = 1$



Divide the circle in to ½, ¼, 1/8, 1/16, 1/∞ but it is approximate as 1.

= n * (1 + 1)

= O(n)

Substitution method:

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

Now find T(n/2) as follows,

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

Replace the above value of T(n/2) in (1)

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

Now find T(n/2^2) as follows,

$T(n/2^2) = T(n/2^3) + n/2^2$

Replace the above value of $T(n/2^2)$ in (2)

$= T(n/2^3) + n/2^2 + n / 2 + n$

Repeat this k number of times

$= T(n/2^k) + n/2^{k-1} + \ldots + n / 2 + n$

Assume $n/2^k = 1$

$= T(1) + n (1/2^{k-1} + 1/2^{k-2} + \ldots + ½^2 + ½ + 1)$

Same as previous - $\sum_{i=0}^{k} \frac{1}{2^i} = 1$

$= 1 + n (1 + 1)$

$= 1 + 2n$

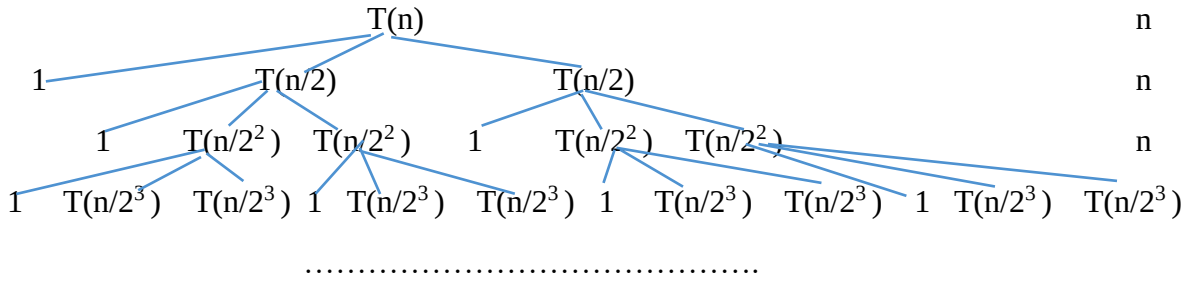
= **O(n)**

Another dividing function

```
void Test(int)                          T(n)
{
        if(n>0)                         1
        {
                for(i=0;i<n;i++)
                {
                printf("%d",n);         n
                }
                Test(n/2);              T(n/2)
                Test(n/2)               T(n/2)
        }
}
```

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

Recursion tree method :



……

$T(n/2^k)$ - k times

So,

k number of steps and in each step, it is called n number of times (n/4 + n/4 + n/4 + n/4 = n)

= n k (1)

Assume

$n / 2^k = 1$

k = log n

so, replace k in (1)

= O (n log n)

Substitution method:

$T(n) = 2\ T(n/2) + n$ (1)

Now find T(n/2) as follows,

$T(n/2) = 2\ T(n/2^2) + n/2$

Replace T(n/2) in equation (1)

$= 2\ [2\ T(n/2^2) + n/2] + n$

$= 2^2\ T(n/2^2) + n + n$ (2)

Now find $T(n/2^2)$ as follows,

$T(n/2^2) = 2\ T(n/2^3) + n/2^2$

Replace $T(n/2^2)$ in equation (2)

$= 2^2\ [\ 2\ T(n/2^3) + n/2^2\ ] + n + n$

$= 2^3\ T(n/2^3) + n + n + n$

$= 2^3\ T(n/2^3) + 3n$ (3)

Repeat this k times

$= 2^k T(n/2^k) + k n$ (4)

Assume that $n/2^k = 1$

k = log n

$= 2^k T(1) + k n$

= n 1 + log n * n

= n + n log n

We consider the dominating term only as per asymptotic notation.

= **O(n log n)**

3.4 Master Theorem for Dividing function

T(n) = a T(n/b) + f(n)

a >= 1 , b > 1 and $f(n) = O(n^k \log^p n)$

$\log_b a$ = ?

k = ?

Case 1:

If $\log_b a > k$ then $(n^{\log_b a})$

Case 2:

If $\log_b a = k$

p > -1 then $O(n^k \log^{p+1} n)$

p = -1then O(n^k loglog n)

$p < -1$ then $O(n^k)$

Case 3:

If $\log_b a < k$

p >= 0 then $O(n^k \log^p n)$

$p < 0$ then $O(n^k)$

Examples:

Case 1:

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

a = 2 , b = 2, k=0

$\log_2 2 = 1$

$1 > 0$ so, $O(n^{\log_b a}) – O(n)$

$T(n) = 4\ T(n/2) + n$

$a = 4, b = 2, k = 1$

$\log_2 4 = 2$

$2 > 1$ so, $O(n^{\log_b a}) – O(n^2)$

$T(n) = 8\ T(n/2) + n$

$a = 8, b = 2, k = 1$

$\log_2 8 = 3$

$3 > 1$ so, $O(n^{\log_b a}) – O(n^3)$

$T(n) = 9\ T(n/2) + n$

$a = 9, b = 2, k = 1$

$\log_2 9 = 2$

$2 > 1$ so, $O(n^{\log_b a}) – O(n^2)$

Case 2:

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

$a = 2, b = 2, k = 1, p = 0$

$\log_2 2 = 1$

$1 = 1$ so, $O(n^k \log^{p+1} n)$ - $O(n \log n)$

$T(n) = 4\ T(n/2) + n^2 \log n$

$a = 4, b = 2, k = 2, p = 0$

$2 = 2$ so, $O(n^k \log^{p+1} n)$ - $O(n^2 \log n)$

$T(n) = 4\ T(n/2) + n^2 \log^3 n$

Then simply multiply, f(n) with log n – so, $O(n^2 \log^4 n)$

$T(n) = 8\ T(n/2) + n^3$

$a = 8, b = 2$ and $k = 3$

$3 = 3$ so, $O(n^3 \log n)$

Case 2.2

$T(n) = 2\ T(n/2) + n / \log n$

$a = 2, b = 2, k = 1, p = -1$

$1 = 1$ and $p = -1$ so, $O(n^k \log\log n)$

Case 2.3

$T(n) = 2\ T(n/2) + n / \log^2 n$

$a = 2, b = 2, k = 1, p = -2$

$1 = 1$ and $p = -2$ so, $O(n)$

Case 3.2:

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

$a = 1, b = 2$ and $k = 2$

$0 < 2$ so, $O(n^2)$

Case 3.1

$T(n) = 2\ T(n/2) + n^2 \log^2 n$

$a = 2, b = 2, k = 2$ and $p = 2$

$1 < 2$ so, $O(n^2 \log^2 n)$

Recurrence relation for Root function

```
void Test(int)                          T(n)
{
    if(n>0)                             1
    {
        printf("%d",n);                 1
        Test(√n);                       T(√n)
    }
}
```

$T(n) = T(\sqrt{n}) + 1$

$$T(n) = \begin{cases} 1 & if\ n = 2 \\ T(\sqrt{n}) + 1 & if\ n > 2 \end{cases}$$

For root function, value of n should be 2 or more.

Using substitution method

$T(n) = T(\sqrt{n}) + 1$ (1)

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

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

$= T(n^{1/2^3}) + 1 + 1 + 1$ (3)

Repeat k times…

$= T(n^{1/2^k}) + k$ (4)

Assume that n is in power of 2 so, $n = 2^m$

$T(2^m) = T(2^{m/2^k}) + k$ (5)

Now, $T(2^{m/2^k})$ is reduced to T(2) so, $T(2^{m/2^k}) = T(2^1)$

$\frac{m}{2^k} = 1$

$m = 2^k$

$k = \log m$

we want answer in n and $n = 2^m$

so, $m = \log n$

k = loglog n

$= T(2^{m/2^k}) + k$ (6)

Replace value of k in (6)

= 1 + loglog n

= O(loglog n)

CHAPTER – 4 Sorting Algorithms

4.1 Category of sorting algorithms

Comparison based sorting –

In comparison-based sorting, elements of an array are compared with each other to find the sorted array.

Examples: Bubble sort, insertion sort, selection sort, quick sort, heap sort and merge sort.

Non-comparison-based sorting –

In non-comparison-based sorting, elements of array are not compared with each other to find the sorted array.

Examples: Radix sort, counting sort and bucket sort

In-place/Outplace technique –

A sorting technique is in place if it does not use any extra memory to sort the array. Among the comparison-based techniques discussed, only merge sort is outplaced technique as it requires an extra array to merge the sorted subarrays. Among the non-comparison-based techniques discussed, all are outplaced techniques. Counting sort uses a counting array and bucket sort uses a hash table for sorting the array.

Online/Offline technique –

A sorting technique is considered online if it can accept new data while the procedure is ongoing i.e. complete data is not required to start the sorting operation. Among the comparison-based techniques discussed, only Insertion Sort qualifies for this because of the underlying algorithm it uses i.e. it processes the array (not just elements) from left to right and if new elements are added to the right, it doesn't impact the ongoing operation.

Stable/Unstable technique –

A sorting technique is stable if it does not change the order of elements with the same value.
Out of comparison-based techniques, bubble sort, insertion sort and merge sort are stable techniques. Selection sort is unstable as it may change the order of elements with the same value. For example, consider the array 4, 4, 1, 3 and sort them using selection sort.
In the first iteration, the minimum element found is 1 and it is swapped with 4 at 0^{th} position. Therefore, the order of 4 with respect to 4 at the 1st position will change. Similarly, quick sort and heap sort are also unstable.

Out of non-comparison-based techniques, Counting sort and Bucket sort are stable sorting techniques whereas radix sort stability depends on the underlying algorithm used for sorting.

4.2 Bubble Sort

It compares the first and second elements of the array; if the first element is greater than the second element, it will swap both elements, and then compare the second and third elements, and so on.

The idea is that neighboring elements are compared with each other and swapped.

Why is it called bubble sort? – If we throw the stone in water then the stone which is heavier so, it goes down and bubbles which are lighter comes up. Similarly, largest element is placed in its proper position at the end of each iteration/pass.

8	5	7	3	2

1st Pass:

8	5	5	5	5
5	**8**	7	7	7
7	**7**	**8**	3	3
3	3	**3**	**8**	2
2	2	2	**2**	**8**

At each pass, the largest element is placed in its proper position. Element 8 is placed in the last position.

Number of comparisons - 4

2nd Pass:

5	5	5	5
7	**7**	3	3
3	**3**	**7**	2
2	2	**2**	**7**
8	8	8	**8**

Element 7 from unsorted elements, is placed in its position.

Number of comparisons - 3

3rd Pass:

5	3	3
3	**5**	2
2	**2**	**5**
7	7	**7**

8	8	**8**

Element 5 is placed in its proper position.

Number of comparisons - 2

4th Pass:

3	2
2	**3**
5	**5**
7	**7**
8	**8**

Element 2 is placed in its proper position.

Number of comparisons – 1

Time complexity:

Worst case:

Number of comparisons are = 1 + 2 + 3 + 4 … + n-1 = n (n-1) /2 = **$O(n^2)$** which is maximum time taken.

Note: We mostly see the number of comparisons for finding time complexity.

```
Algorithm bubble(A,n)
{
for(i=0;i<n-1;i++)
  for(j=0;j<n-1-i;j++)
      {
        if(A[j] > A[j+1])
            Swap element A[j] with A[j+1]
        }
}
```

Best case:

If the list of elements is already sorted. Then there is not any swap in the first pass, which shows that elements are sorted and that can be done by the following algorithm using flag variable.

```
Algorithm bubble(A,n)
{
```

```
for(i=0;i<n-1;i++)
{
flag=0;
  for(j=0;j<n-1-i;j++)
       {
        if(A[j] > A[j+1])
            Swap element A[j] with A[j+1]
           flag=1;
        }
if(flag==0)
     break;
}
}
```

Number of comparison in first pass – n-1 = **O(n)** which is minimum time taken by the bubble sort.

We can say that Bubble sort is adaptive by putting the flag variable.

Summary of time complexity:

	Best	Average	Worst
Without flag	$O(n^2)$	$O(n^2)$	$O(n^2)$
With flag	O(n)	$O(n^2)$	$O(n^2)$

Space complexity:

It is constant. No additional memory space is required. O(1). It is in place sort algorithm.

Stable or not?

Whether it is stable or not? A stable sorting algorithm maintains the relative order of the items with equal sort keys.

We can check it by sorting the following elements:

8 8 3 5 4

In the above list, two elements are having same value so, their relative order in the sorted list will remain the same means second 8 will be placed after first 8.

4.3 Insertion Sort

Insertion sort works similar to the sorting of playing cards in hands. It is assumed that the first card is already sorted in the card game, and then we select an unsorted card. If the selected unsorted card is greater than the first card, it will be placed at the right side; otherwise, it will be placed at the left side. Similarly, all unsorted cards are taken and put in their exact place.

We are given the list of sorted elements as follows:

2	6	10	15	20	25	30

Now, we have to insert element 12 in the above list.

Start comparing element from the last index, if element is larger than newly inserted then shift that element to right.

2	6	10	15	20	25		30
							12

2	6	10	15	20		25	30
					12		

2	6	10	15		20	25	30
				12			

2	6	10		15	20	25	30
			12				

Now, element 10 is smaller than 12 so, put element 12 in the empty slot.

2	6	10	**12**	15	20	25	30

Consider the following list of elements which is unsorted.

8	5	7	3	2

List with one element is already sorted as follows. Dark background shows sorted portions of the array.

8	5	7	3	2

<u>Insert 5: pass 1</u>

8	5	7	3	2

5	8	7	3	2

Number of comparisons – 1

Insert 7: pass 2

5	8	7	3	2

5	7	8	3	2

Number of comparisons – 2

Insert 3: pass 3

5	7	8	3	2

3	5	7	8	2

Number of comparisons - 3

Insert 2: pass 4

3	5	7	8	2

3	5	7		8

2

3	5		7	8

2

3		5	7	8

2

2	3	5	7	8

Number of comparisons – 4

Number of passes required = 4 means n – 1 pass

Time complexity:

Best Case: - It occurs when there is no sorting required, i.e. the array is already sorted and outer loop is iterated only n times and it does not enter into inner loop for swapping of the elements. The best-case time complexity of insertion sort is **O(n)**.

Average Case: - It occurs when the array elements are in jumbled order that is not properly in ascending and not properly in descending order. The average case time complexity of insertion sort is **$O(n^2)$** which as bad as worst case.

Worst Case: - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of insertion sort is **$O(n^2)$**.

Number of comparisons - $1 + 2 + 3 + 4 + \ldots + n - 1 = n (n-1) / 2 = O(n^2)$

Number of swapping are same as comparisons.

Summary of time complexity:

Best case	Average case	Worst case
O(n)	$O(n^2)$	$O(n^2)$

Space complexity:

It is constant. No additional memory space required so, O(1). It is in place sort algorithm

Algorithm InsertionSort(A[],n)

```
{
for(i=1;i<n;i++)
        {
          j=i-1;
          x=A[j];
         while( j >= -1 && A[j] > x)
           {
               A[j+1] = A[j];
                j--;
            }
       A[j+1]=x;
      }
```

}

Stable or not?

Insertion sort is also *stable sort*. It maintains the relative position of elements having same value.

Insertion sort has various advantages such as -

- Simple implementation
- Efficient for small data sets
- Adaptive, i.e., it is appropriate for data sets that are already substantially sorted.

(GATE CSE 2003)

The usual $\Theta(n^2)$ implementation of Insertion Sort to sort an array uses linear search to identify the position where an element is to be inserted into the already sorted part of the array. If instead, we use binary search to identify the position, the worst-case running time will _____.

a. Remain $\Theta(n^2)$
b. Become $\Theta(n(\log n)^2)$
c. Become $\Theta(n\log n)$
d. Become $\Theta(n)$

Answer (a)

4.4 Selection Sort

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. It is also the simplest algorithm. In this algorithm, the array is divided into two parts, first is sorted part, and another one is the unsorted part. Initially, the sorted part of the array is empty, and unsorted part is the given array. Sorted part is placed at the left, while the unsorted part is placed at the right.

The array is divided into sorted and unsorted array as the process progresses. In each pass/iteration, one minimum (ascending order)/maximum (descending order) element is selected and placed in its proper position.

0	1	2	3	4	5
7	4	10	8	3	1

Pass 1: (start searching the element for position 0). Wherever we get minimum element that is swapped with element with position 0.

0	1	2	3	4	5
1	4	10	8	3	7

Number 1 is swapped with 7.

Number of comparisons – 5

Pass 2:

0	1	2	3	4	5
1	**4**	10	8	**3**	7

Number 4 and 3 are swapped.

Number of comparisons – 4

0	1	2	3	4	5
1	**3**	10	8	**4**	7

Pass 3:

0	1	2	3	4	5
1	3	10	8	4	7

Number of comparisons – 3

0	1	2	3	4	5
1	3	**10**	8	**4**	7

Numbers 10 and 4 are swapped.

0	1	2	3	4	5
1	3	4	8	10	7

Pass 4:

0	1	2	3	4	5
1	3	4	8	10	7

Number of comparisons – 2

0	1	2	3	4	5
1	3	4	**8**	10	**7**

Number 8 and 7 are swapped.

0	1	2	3	4	5
1	3	4	7	10	8

Pass 5:

0	1	2	3	4	5
1	3	4	7	10	8

Number of comparisons – 1

0	1	2	3	4	5
1	3	4	7	**10**	**8**

Number 10 and 8 are swapped.

0	1	2	3	4	5
1	3	4	7	8	10

Number of passes are 5 which is n – 1.

Time complexity:

Best Case Complexity - It occurs when there is no sorting required, i.e. the array is already sorted. As per the mechanism of this sorting, it is difficult to decide that array is sorted or not and we need to do all the comparisons as we do in worst case scenario. The best-case time complexity of selection sort is $\mathbf{O(n^2)}$.

Average Case Complexity - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of selection sort is $\mathbf{O(n^2)}$.

Worst Case Complexity - It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but

its elements are in descending order. The worst-case time complexity of selection sort is **$O(n^2)$**.

Number of comparisons = 1 + 2 + 3 + 4 + 5 + … + (n-1) + n = n (n-1) /2 = $O(n^2)$

Summary of time complexity:

Best case	Average case	Worst case
$O(n^2)$	$O(n^2)$	$O(n^2)$

Space complexity:

It is constant. No additional memory space is required so, O(1). It is in place sort algorithm.

Algorithm SelectionSort(A[],n)

```
{
for(i=0;i<n-1;i++)
{
   min = i;
   for(j+i+1,j<n;j++)
    {
       if(A[j] < A[min])
            min = j;
     }
 if(min != i)
   swap(A[i],A[min])
}
```

Adaptive or not? Not adaptive, meaning it doesn't take advantage of the fact that the list may already be sorted or partially sorted.

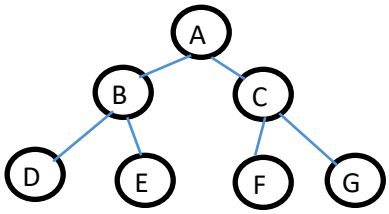
Stable or not?

By default, implementation is not stable, but it can be made stable.

4.5 Heap Sort

A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be filled completely.

First, we see representation of binary tree using array.



1	2	3	4	5	6	7
A	B	C	D	E	F	G

If index starts from 1 then:

If any element is at index i then its

Left child of i^{th} element is at = 2 * i

Right child of i^{th} element is at = 2 * i + 1

Parent of i^{th} element is at = $\left\lfloor \frac{i}{2} \right\rfloor$

If index starts from 0 then:

If any element is at index i then its

Left child of i^{th} element is at = 2 * i + 1

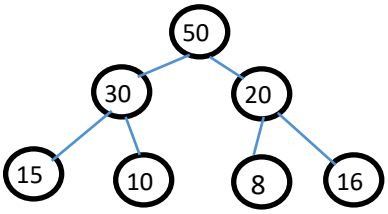
Right child of i^{th} element is at = 2 * i + 2

Parent of i^{th} element is at = $\left\lfloor \frac{i-1}{2} \right\rfloor$

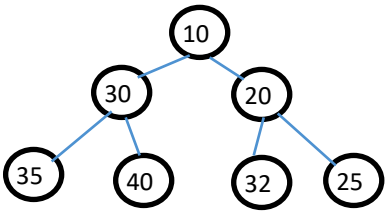
Complete binary tree (it is strictly tree of depth d, and all its leaves are located at level d) then the above representation is worth otherwise it keeps gap in the array. The elements are filled level by level in complete binary tree.

Height of the complete binary is $\mathbf{\log_2 n}$.

Max heap: All its descendants are less than its. Element which root – 50 having all its descendants are less than it.

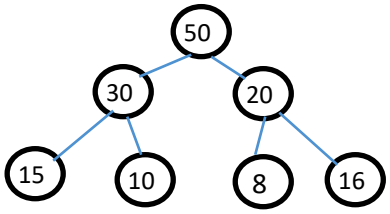


Min heap: All its descendants are greater than its. Element which root – 50 having all its descendants are less than it.



Let's take max heap and see the process of insertion and deletion.

Insertion process:



1	2	3	4	5	6	7	8
50	30	20	15	10	8	16	**60**

Now, insert element 60 in the above heap.

Element 60 will be inserted on index 8 but as per the property of max heap, we cannot insert it at 8th position.

So, it will go upward by checking $\lfloor 8/2 \rfloor = 4$ and compared with element at index 4 which is 15. Again, it goes upward by checking $\lfloor 4/2 \rfloor = 2$ and compared with element at index 2 which is 30. Again, it is compared with its parent 50 and interchanged.

1	2	3	4	5	6	7	8
50	30	20	**15**	10	8	16	**60**

1	2	3	4	5	6	7	8

50	30	20	**60**	10	8	16	**15**

1	2	3	4	5	6	7	8
50	**30**	20	**60**	10	8	16	15

1	2	3	4	5	6	7	8
50	**60**	20	**30**	10	8	16	15

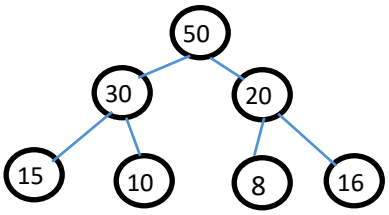
1	2	3	4	5	6	7	8
50	**60**	20	**30**	10	8	16	15

1	2	3	4	5	6	7	8
60	**50**	20	**30**	10	8	16	15

Number of comparison for insertion is equal to height of the tree which is $\lfloor \log_2 n \rfloor$.

Deletion process:

Delete 50 from the max heap. Mostly, we want the maximum or minimum element from the max heap and min heap respectively.



1	2	3	4	5	6	7
~~50~~	30	20	15	10	8	16

1	2	3	4	5	6	7
16	30	20	15	10	8	~~50~~

But it does not satisfy the property of max heap. So, we check its children and swap it with largest one.

1	2	3	4	5	6	7
30	16	20	15	10	8	~~50~~

Now, again check its children which are 15 and 10 but both are smaller than it.

1	2	3	4	5	6	7
30	16	20	15	10	8	~~50~~

No need to swap now.

Now, delete element 30 so,

1	2	3	4	5	6	7
8	16	20	15	10	~~30~~	~~50~~

Deleted element's (30) position will be taken by last element (8) but by putting it, property of max heap is not satisfied. So, we need to do it by changing the position of element 8 in the above.

1	2	3	4	5	6	7
8	16	**20**	15	10	~~30~~	~~50~~

Now, check the children of element 8 and it is swapped with which is larger. So, 8 is swapped with 20.

1	2	3	4	5	6	7
20	16	**8**	**15**	10	~~30~~	~~50~~

Still, element 8 has both of its children are greater than its so, swap element 8 with larger of its children which is 15.

1	2	3	4	5	6	7
20	16	**15**	**8**	10	~~30~~	~~50~~

In this way, we keep on deleting the elements and adding it in the list and finally we will get sorted list of elements.

Maximum time to delete the element is equal to height of binary tree which is $\log_2 n$.

Heap Sorting:

In heap sort, basically, there are two phases involved in the sorting of elements. By using the heap sort algorithm, they are as follows -

- The first step includes the creation of a heap by adjusting the elements of the array.

- After the creation of heap, remove the root element of the heap repeatedly by shifting it to the end of the array.

Creating heap for heap sort:

1	2	3	4	5
10	20	15	30	40

Take element one by one and compare with its parent by formula floor (i / 2) where i is the index of the element.

Take 1st element – 10:

1	2	3	4	5
10	20	15	30	40

So, it is max heap of one element, no need to do anything.

Take 2nd element – 20:

1	2	3	4	5
10	20	15	30	40

Check the parent of newly inserted element which should be greater than it. If not then interchange.

1	2	3	4	5
20	10	15	30	40

Take 3rd element – 15:

No need to do anything because 15's parent is 20 which is greater than it.

Take 4th element – 30:

1	2	3	4	5
20	**10**	15	**30**	40

Now, 30's parent – floor (4/2) = 2, which is 10 and less than 30 so, swap them.

1	2	3	4	5
20	**30**	15	**10**	40

In the above, element 30 is not its proper position so, it is swapped with its parent 20.

1	2	3	4	5
30	**20**	15	**10**	40

Take 5th element – 40:

1	2	3	4	5
30	**20**	15	10	40

40's parent (floor(5/2) – 2) which is 30 so, less than 40 hence swap them.

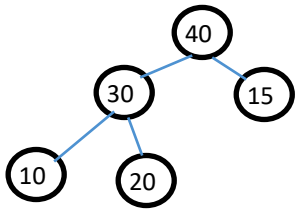
1	2	3	4	5
30	**40**	15	10	20

Still, 40's parent is smaller than it so, swap it.

1	2	3	4	5
40	**30**	15	10	20

The following is the final representation of elements using max heap in array and tree.

1	2	3	4	5
40	30	15	10	20



Deletion for heap sort:

Now, start deleting elements from the above max heap.

1	2	3	4	5
40	30	15	10	20

Delete 40:

1	2	3	4	5
20	30	15	10	~~40~~

First, element which is the largest element and swapped with last element as above.

But, this does not satisfy the property of max heap so, 20 will be sent down and swapped with one of its children which is larger so, it is swapped with 30.

1	2	3	4	5
30	20	15	10	~~40~~

Delete 30:

1	2	3	4	5
10	20	15	~~30~~	~~40~~

Now, element 10 is swapped with largest children.

1	2	3	4	5
20	10	15	~~30~~	~~40~~

Delete 20:

1	2	3	4	5
15	10	~~20~~	~~30~~	~~40~~

No, need to swap because no child is greater than it.

Delete 15:

1	2	3	4	5
~~10~~	~~15~~	~~20~~	~~30~~	~~40~~

Now, last element is already in its proper position.

Insertion process (creation of max heap): one element can be inserted in to max heap with log n comparison so, n elements can be inserted with n log n comparisons.

Deletion process: one element is deleted and in order to maintain the property of max heap it take log n comparisons so, for n number of elements we need n log n number of comparisons.

So, total time is =

max heap creation time + deletion time = $n \log n + n \log n = 2 n \log_2 n = O (n \log_2 n)$

Summary of time complexity:

Best case	Average case	Worst case
O(n log n)	O(n log n)	O(n log n)

There is no mechanism to detect whether the list is sorted or not. For any sequence of data, heap sort dose the same work. So, time complexity of heap sort is the same in all three cases. Hence heap sort is not adaptive.

Space complexity:

It is constant. So, no additional memory is required – O (1). It is in place sorting algorithm.

Stable or not?

It is not stable. It does not maintain the relative order of same element.

(GATE CSE 2015 Set 3)

- Consider the following array of elements.

⟨89,19,50,17,12,15,2,5,7,11,6,9,100⟩

The minimum number of interchanges needed to convert it into a max-heap is _____.

a. 4
b. 5
c. 2
d. 3

Answer (d)

Non – comparison-based sorting algorithms:

4.6 Radix Sort

It sorts the elements by sorting them on their radix. Radix for decimal number is from 0 to 9 (10 digits), radix for alphabets is a to z (26 alphabets)

The process of radix sorting works similar to the sorting of student's names, according to the alphabetical order. In this case, there are 26 radixes formed due to the 26 alphabets in English. In the first pass, the names of students are grouped according to the ascending order of the first letter of their names. After that, in the second pass, their names are grouped according to the ascending order of the second letter of their name. And the process continues until we find the sorted list.

Take the following list of elements:

237	146	259	348	152	163	235	48	36	62

Here, we have decimal number so, we take bin of size 9 (0 to 9).

Pass – 1:

Put the number in the specific bin based on the LSB or last digit of the number.

0	1	2	3	4	5	6	7	8	9
		62	163		235	146	237	348	259
						36		48	

Now, make each bin empty from 0 to 9 and list the number…

62 163 235 146 36 237 348 48 259

Pass – 2:

Now, put the number in the bins based on 2nd LSB of the number. For example, 62 and its LSB is 6 so, it is kept in bin number 6.

0	1	2	3	4	5	6	7	8	9
			235	146	259	62			
			36	348		163			
			237	48					

Now, make all the bins empty and list the numbers.

235 36 237 146 348 48 259 62 163

Pass-3:

Put the number into bins based on 3rd LSB. Largest number is of three digits and LSB for 235 number is 2 so, it is kept in bin number 2. For two-digit numbers, 0 is appended so, it is LSB is zero and kept in bin number 0.

0	1	2	3	4	5	6	7	8	9
36	146	235	348						
48	163	237							
62		259							

Now, make bins empty and list the numbers.

36 48 62 146 163 235 237 259 348

Time complexity:

Number of passes is equal to number of digits **d** in the largest number.

Number of elements we put into the bin is **n**

So, time complexity is – **O(d n)** where d is number of digits in the largest number. d cannot be constant. But if it is constant then the complexity is linear – O(n).

Summary of time complexity:

Best case	Average case	Worst case
O(nd)	O(nd)	O(nd)

Space complexity:

Radix sort also has a space complexity of **O (n + d),** where n is the number of elements and d is the base of the number system. This space complexity comes from the need to create buckets for each digit value and to copy the elements back to the original array after each sorting is done on digit.

Stable or not?

It is stable.

4.7 Bucket Sort

Sort a large set of floating-point numbers which are in range from 0.0 to 1.0 and are uniformly distributed across the range. How do we sort the numbers efficiently? A simple way is to apply a comparison-based sorting algorithm. The lower bound for Comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(n log n), i.e., they cannot do better than n log n.

Bucket sort is a sorting algorithm that separate the elements into multiple groups said to be buckets. Elements in bucket sort are first uniformly divided into groups called buckets, and then they are sorted by any other sorting algorithm. After that, elements are gathered in a sorted manner.

Bucket sort is commonly used -

- With floating-point values. It works on floating point number in range of 0.0 to 1.0.
- When input is distributed uniformly over a range.

The advantages of bucket sort are -

- Bucket sort reduces the no. of comparisons.
- It is asymptotically fast because of the uniform distribution of elements.

The limitations of bucket sort are -

- It may or may not be a stable sorting algorithm.
- It is not useful if we have a large array because it increases the cost.
- It is not an in-place sorting algorithm, because some extra space is required to sort the buckets.

Let's take the example:

0.79, 0.13, 0.64, 0.39, 0.20, 0.89, 0.53, 0.42, 0.06, 0.94

0	0.06				
1	0.13				
2	0.20				
3	0.39				
4	0.42				
5	0.53				
6	0.64				
7	0.79				
8	0.89				
9	0.94				

Put the elements into the appropriate bucket by applying - B$\lfloor n * A[i] \rfloor$ = B[10 * 0.79]=B[7] so, 0.79 is placed in bucket 7.

Algorithm Bucket Sort(A[],n)

1. Let B[0....n-1] be a **new** array

2. n=length[A]
3. **for** i=0 to n-1
4. make B[i] an empty list
5. **for** i=1 to n
6. **do** insert A[i] into list B$\lfloor n * A[i] \rfloor$ – (floor value of the index)
7. **for** i=0 to n-1
8. **do** sort list B[i] with insertion-sort
9. Concatenate lists B[0], B[1],....., B[n-1] together in order

End

Time complexity:

Best Case: - In Bucket sort, best case occurs when the elements are uniformly distributed in the buckets. The complexity will be better if the elements are already sorted in the buckets. Time complexity will be O (n + k), where O(n) is for making (insertion into the bucket) the buckets, and O(k) is for concatenating the bucket elements.

The best-case time complexity of bucket sort is **O(n + k)**.

Average Case: - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. Bucket sort runs in the linear time, even when the elements are uniformly distributed. The average case time complexity of bucket sort is **O (n + k)**.

Worst Case: - In bucket sort, worst case occurs when the elements are of the close range in the array and because of that, they have to be placed in the same bucket. So, some buckets have a greater number of elements than others.
The complexity will get worse when the elements are in the reverse order.
The worst-case time complexity of bucket sort is **O(n^2)**.

If the sequence is like – 0.79, 0.74, 0.72, 0.71 …

Insert – 0.79

0					
1					
2					
3					
4					
5					
6					
7	0.79				
8					
9					

Insert – 0.74

0					
1					
2					
3					
4					
5					
6					
7	0.74	0.79			
8					
9					

Insert – 0.72

0					
1					
2					
3					
4					
5					
6					
7	0.72	0.74	0.79		
8					
9					

If all the elements are stored in the same bucket as above, then they are stored using insertion sort and it takes time $O(n^2)$.

Worst case - $O(n^2)$.

Summary of time complexity:

Best case	Average case	Worst case
O(n+k)	O(n+k)	$O(n^2)$

Space complexity:

If k is the number of buckets required, then O(k) extra space is needed to store k empty buckets, and then we map each element to a bucket that requires O(n) extra space. So, the overall space complexity is O (n + k).

4.8 Counting Sort

It is non-comparison-based algorithm as bucket sort. It sorts data based on the frequency of the elements. it is an integer sorting algorithm.

We are given input size – n and range - k. Range means input must be in that range.

From the name, we have to count the frequency/occurrence of each input.

For example:

2 1 2 1 3 4 1 2

Create the array of 4 elements:

	2	**1**	**2**	**1**	**3**	**4**	**1**	**2**
1	0	1	1	2	2	2	3	**3**
2	1	1	2	2	2	2	2	**3**
3	0	0	0	0	1	1	1	**1**
4	0	0	0	0	0	1	1	**1**

Now, traverse the list and note down its occurrence in the array. If it appears again then its corresponding entry is incremented.

Order the list of elements based on their occurrence as follows. 1 appears thrice so, list it three times. 3 appears once so, list it one time.

So, the sorted order is - 1 1 1 2 2 2 3 4

Time complexity:

We traverse the list n times and array (to store the frequency of each element) is k times so, O(n+k).

Same time complexity in best, average and worst case.

But if we have the list of elements as follows:

2 23000 5 9 20

We need to take array of 23000 and remaining space will be wasted.

It does not work for floating points or negative values.

Summary of time complexity:

Best case	Average case	Worst case
O(n+k)	O(n+k)	O(n+k)

Space complexity:

O(n+k). where k is range of elements. It means dependent on the larger number in the list. The algorithm allocates two additional arrays: one for counts and one for the output.

Stable or not?
It is stable sort.

Counting sort is specifically useful in following scenarios:

- Linear complexity is needed
- Smaller integers with multiple count
- Can be used as a subroutine in Radix sort

4.9 Amortize Analysis

This analysis is used when the occasional operation is very slow, but most of the operations which are executing very frequently are faster. Data structures we need amortized analysis for Hash Tables, Disjoint Sets etc.

In the Hash-table, the most of the time the searching time complexity is O(1), but sometimes it executes O(n) operations. When we want to search or insert an element in a hash table for most of the cases it is constant time taking the task, but when a collision occurs, it needs O(n) times operations for collision resolution.

Aggregate Method

The aggregate method is used to find the total cost. If we want to add a bunch of data, then we need to find the amortized cost by this formula.

For a sequence of n operations, the cost is –

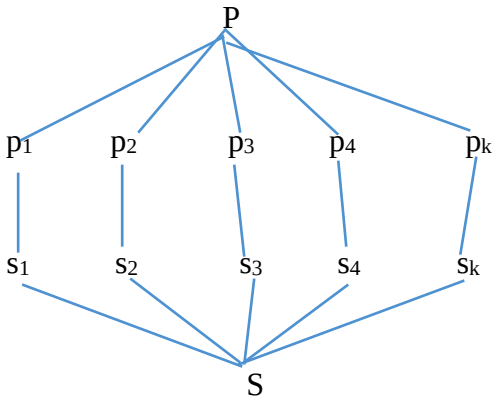
$$\frac{Cost(n\ operations)}{n} = \frac{Cost(normal\ operations) + Cost(Expensive\ operations)}{n}$$

CHAPTER – 5 Divide and Conquer Method

5.1 Introduction to Divide and Conquer approach

This is one of the approaches to solve the problem.

In this approach, bigger problem is divided into small (sub) problem. If sub problem is large, then again divide in to sub sub problem.



If sub problem is the same as large problem, then only we can apply divide and conquer method. If main problem is sorting, then sub problem must be also sorting.

For example, we have to plan some events and it has many sub task which are done independently. This can be solved by divide and conquer.

You must have some method to combine the solution otherwise it is not possible.

Problem solved using this approach can be written as recursive. We must know how to write recursive function for the problem.

```
Algorithm DAC(P)
{
if(small(P))
  S(P);
else
   {
     Divide P into P1, P2 .. Pk
     Apply DAC(P1), DAC(P2) … DAC(Pk)
     Combine (DAC(P1), DAC(P2) … DAC(Pk))
   }
}
```

Problems which are possible to solve using this approach:

1. Binary Search
2. Merge sort
3. Quick sort
4. Fibonacci series
5. Finding maximum and minimum
6. Strassem's matrix multiplication
7. Multiplication of two large integer numbers
8. Exponentiation

5.2 Quick Sort:

Process of quick sort is like following example.

If there are a group of students and teacher has to arrange them in order of their heights.

1) Teacher arranges them based on their height.
2) Teacher asks students to do themselves.

The quickest way is that students find their position themselves.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

The idea in quick sort is that any element is at sorted position then all elements on left hand side are smaller and on right hand side greater than it.

0	1	2	3	4	5	6	7	8	9
10	16	8	12	15	6	3	9	5	∞
i									j

Pivot – 10***:*** (first element, we select as pivot element, but other options are also there)

Increment i until we get element greater than 10 and decrement j until we get element less than or equal to 10.

0	1	2	3	4	5	6	7	8	9
10	16	8	12	15	6	3	9	5	
	i							j	

Now, interchange 16 and 5.

0	1	2	3	4	5	6	7	8	9
10	5	8	12	15	6	3	9	16	
	i							j	

0	1	2	3	4	5	6	7	8	9
10	5	8	12	15	6	3	9	16	
			i				j		

Now, interchange 12 and 9.

0	1	2	3	4	5	6	7	8	9
10	5	8	9	15	6	3	12	16	
			i				j		

0	1	2	3	4	5	6	7	8	9
10	5	8	9	15	6	3	12	16	
				i		j			

Now, interchange 15 and 3.

0	1	2	3	4	5	6	7	8	9
10	5	8	9	3	6	15	12	16	
				i		j			

0	1	2	3	4	5	6	7	8	9
10	5	8	9	3	6	15	12	16	
					j	i			

When j and i crosses each other then j is the position of pivot element 10 so, interchange 10 and 6.

0	1	2	3	4	5	6	7	8	9
6	5	8	9	3	10	15	12	16	
					j	i			

Elements on both sides are not sorted.

Now, perform the process recursively for both the partitions by selecting the first element as pivot element.

Algorithm **Partition**(l, h)

{

pivot = A[l];

i=l,j=h;

while(i < j)

{

do

{

i++; }

while(A[i] <= pivot);

do

{

j--;

```
}while(A[j] > pivot);
if(i < j)
  swap(A[i],A[j]);
 }
swap(A[l],A[j]);
return j;
}
```

```
Algorithm QuickSort(l,h)
{
if( l < h)
{
j = Partition(l,h);
QuickSort(l,j);
QuickSort(j+1,h);
}
}
```

Sorted element will work as infinity or highest element so, we have put j not j-1.

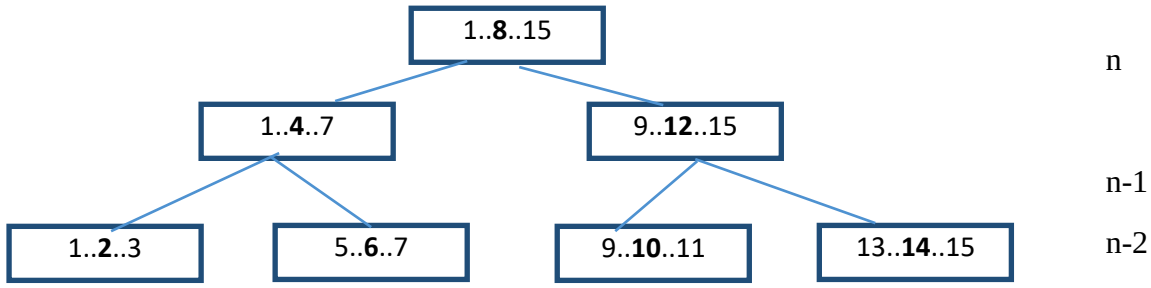
Choosing the pivot element

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

Analysis of the algorithm:

Best case:

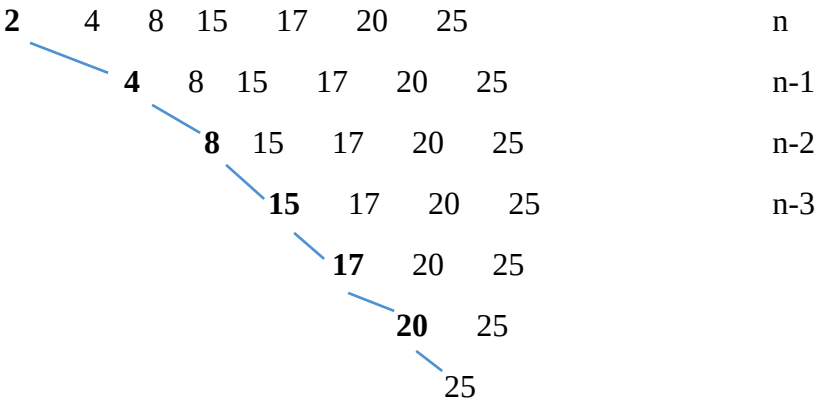


If the partitions is done as above with equal size then, height of the tree will be $\log_2 n$ and in each level number of comparisons are at maximum n (in lower level, it reduces by 1 if we go down) so, O ($n \log_2 n$).

Best case - **O($n \log_2 n$).** Pivot element must be as median all the time which is not always possible.

Worst case:

If all the elements are already sorted and first element is selected as pivot. Then partition is done as follows and height of the tree is not $\log_2 n$ as the best case.



Number of comparisons = 1 + 2 + 3 + … + n-1 +n

= n (n+1) /2

= **O(n^2)**

Summary of time complexity:

Best case	Average case	Worst case
O($n \log_2 n$)	O($n \log_2 n$)	O(n^2)

Space complexity:

We need additional memory space – O($\log_2 n$). It is because of stack frames of recursive calls for the partitions. In worst case, it needs n stack frames so, it is O(n).

Stable or not?

It is not stable.

Recurrence relation of quick sort:

Worst case:

$$T(n) = \begin{cases} 1 & if\ n = 1 \\ T(n-1) + n & if\ n > 1 \end{cases}$$

Best case and average case:

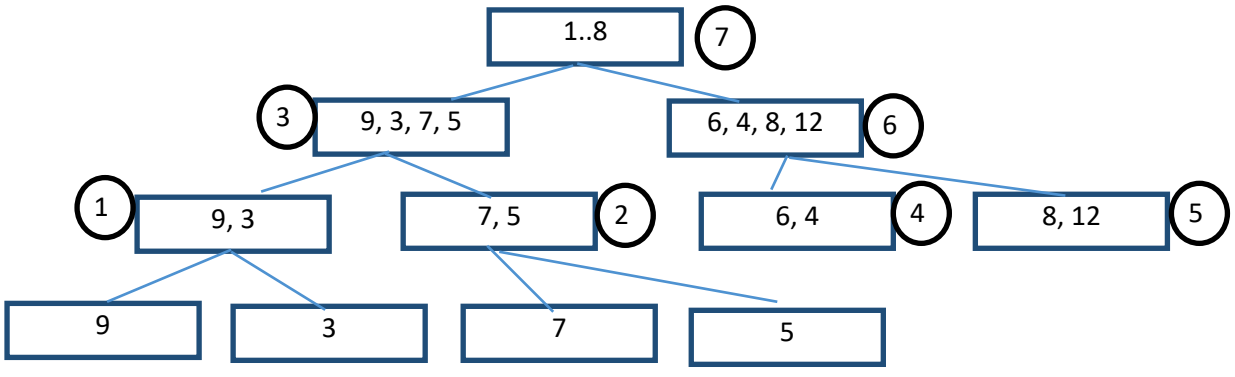
$$T(n) = \begin{cases} 1 & if\ n = 1 \\ 2\,T\left(\frac{n}{2}\right) + n & if\ n > 1 \end{cases}$$

5.3 Merge Sort:

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithms. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the **merge()** function to perform the merging.

This sorting is based on divide and conquer approach where the problem is divided into sub problem and solve it and then combine the sub problems.

9	3	7	5	6	4	8	2



Shows the merging operation.

In each level, n or a smaller number of elements are merged so, maximum n and height of the tree is $\log_2 n$. so, time complexity is $O(n \log_2 n)$.

Best case: $O(n \log_2 n)$.

Worst case: O(n $\log_2$ n). There is no change in the above tree so, height remains as $\log_2$ n unlike quick sort.

Summary of time complexity:

Best case	Average case	Worst case
O(n $\log_2$ n)	O(n $\log_2$ n)	(n $\log_2$ n)

Space complexity:

We need additional memory space: $\log_2$ n for stack and n for auxiliary array so, total n + $\log_2$ n – O(n).

Pros and Cons of Merge Sort:

Pros:

1. Large size list. No other sort algorithm supports large size list.
2. It is suitable using linked list. Two sorted linked list can be merged without creating third linked list.
3. Support external sorting. Two files having millions of numbers can be merged. Suppose we have the following two files in hard disk, and we have to sort them.

F1	**F2**	**F3**
2	3	
4	5	
6	8	
7	15	

4. Merge sort is stable.

Cons:

1. Extra space (on in place sort). Merging array is kept in separate array.
2. Not applied to small size problem. For small size problem, it is slower than other algorithm. It incurs time in recursion so, slower than insertion and other sorting algorithms.
3. Recursive algorithm. It uses a stack and height of the stack is as per the height of the tree.

```
Algorithm MergeSort(A[],l, h)          T(n)
{
if(l < h)
{
mid = (l+h)/2;                         1
```

MergeSort(l,mid); T(n/2)

MergeSort(mid+1,h); T(n/2)

Merge(A[],l, mid, h); n

}

}

Recurrence relation:

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

Example: We are given two sorted array and how to merge them.

A	B	C
2	5	
8	9	
15	12	
18	17	
22		

Compare the corresponding elements of A and B and if whichever is smaller, copy that element to C and move to the next element from whichever array it is copied.

If one list is larger than the other, then copy its remaining element directly to array C because they are sorted.

```
void Merge(int A[], int beg, int mid, int end)
{
   int i, j, k;
   int n1 = mid - beg + 1;
   int n2 = end - mid;

   int LeftArray[n1], RightArray[n2]; //temporary arrays

   /* copy data to temp arrays */
   for (int i = 0; i < n1; i++)
   LeftArray[i] = A[beg + i];
   for (int j = 0; j < n2; j++)
   RightArray[j] = A[mid + 1 + j];

   i = 0; /* initial index of first sub-array */
```

```
    j = 0; /* initial index of second sub-array */
    k = beg;  /* initial index of merged sub-array */

    while (i < n1 && j < n2)
    {
        if(LeftArray[i] <= RightArray[j])
        {
            A[k] = LeftArray[i];
            i++;
        }
        else
        {
            A[k] = RightArray[j];
            j++;
        }
        k++;
    }
    while (i<n1)
    {
        A[k] = LeftArray[i];
        i++;
        k++;
    }

    while (j<n2)
    {
        A[k] = RightArray[j];
        j++;
        k++;
    }
}
```

Recurrence relation:

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

5.4 2 Way Merge Sort:

It is iterative process than recursive process of merge sort.

9	3	7	5	6	4	8	2

2 lists are merged at a time.

Each element is considered as list of single elements. It is sorted by default.

1st Pass:

Take pair of elements means 2-2 elements and sort them. Divide the entire list into list of 2 elements.

9	3	7	5	6	4	8	2

Copy it into another array and after sorting copy them back to original array.

3	9	5	7	4	6	2	8

2nd Pass:

Consider the list of four elements so, we have two lists.

3	9	5	7	4	6	2	8

3	5	7	9	2	4	6	8

3rd Pass:

Now, merge these two lists into single list using the process of merging two sorted arrays.

2	3	4	5	6	7	8	9

In each pass, we do n number of comparison and total number of passes are $\log_2 n$ so, time complexity is – **$O(n \log_2 n)$.**

(GATE CSE 1995) • For merging two sorted lists of sizes m and n into a sorted list of size m+n, we require comparisons of _____.

a. O(m)
b. O(n)
c. O(m+n)
d. O(logm + logn)

Answer (c)

Time and Space Complexity Table

Sorting Algorithm	Time Complexity			Space Complexity
	Best Case	Average Case	Worst Case	Worst Case
Bubble Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	O(1)
Selection Sort	$\Omega(N^2)$	$\Theta(N^2)$	$O(N^2)$	O(1)
Insertion Sort	$\Omega(N)$	$\Theta(N^2)$	$O(N^2)$	O(1)
Merge Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$	O(N)
Heap Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N \log N)$	O(1)
Quick Sort	$\Omega(N \log N)$	$\Theta(N \log N)$	$O(N^2)$	O(log N)
Radix Sort	$\Omega(N k)$	$\Theta(N k)$	$O(N k)$	O(N + k)
Count Sort	$\Omega(N + k)$	$\Theta(N + k)$	$O(N + k)$	O(N + k)
Bucket Sort	$\Omega(N + k)$	$\Theta(N + k)$	$O(N^2)$	O(N + k)

5.5 Multiplication of large numbers:

We all know of the standard multiplication algorithm we were taught in primary school. For two n-digit numbers, it essentially requires the product of every digit of first number with every digit of second number. So, the time complexity is $O(n^2)$.

If we multiply large numbers, then it becomes tough and complex.

Multiplication of two two-digit numbers:

The two-digit number is splitted into two one-digit number as follows:

$a = a_1\ a_0$

$b = b_1\ b_0$

The number is represented as follows in the positional number system.

$a = 10^1\ a_1 + 10^0\ a_0$

$b = 10^1\ b_1 + 10^0\ b_0$

$a * b = (\ 10^1\ a_1 + 10^0\ a_0\ ) * (10^1\ b_1 + 10^0\ b_0\ )$

$= 10^2\ (a_1 * b_1) + 10^1\ (\ (a_0 * b_1) + (a_1\ * b_0)\ ) + 10^0\ (\ a_0 * b_0)$

$(a_1 + a_0\ ) * (b_1 + b_0) = a_1\ * b_1\ + a_1 * b_0 + a_0\ * b_1 + a_0 * b_0$

There are four multiplications in the above formula. It can be reduced to three using the following computations at the cost of some cost of addition, subtraction and shifting operations.

$(a_1 + a_0\ ) * (b_1 + b_0)\ =\ a_1\ * b_1\ (c_2) + (\ a_1 * b_0 + a_0\ * b_1\ )\ (c_1)\ + a_0 * b_0\ (c_0)$

$c_1\ =\ (a_1 + a_0\ ) * (b_1 + b_0) – (c_2\ + c_0\ )$

$c_2 = a_1\ * b_1$

$c_0 = a_0 * b_0$

$a * b = 10^2\ (a_1 * b_1) + 10^1\ (\ (a_0 * b_1) + (a_1\ * b_0)\ ) + 10^0\ (\ a_0 * b_0)$

$= 10^2\ c_2 + 10^1\ c_1\ + 10^0\ c_0$

$A = 42 :\ a_1 = 4,\ a_0 = 2$

$B = 34 : b_1 = 3\ ,\ b_0 = 4$

$A * B = 10^2\ c_2 + 10^1\ c_1\ + 10^0\ c_0$

$= 10^2\ a_1\ * b_1\ + 10^1\ (a_1 + a_0\ ) * (b_1 + b_0) – (c_2\ + c_0\ )\ + 10^0\ a_0 * b_0$

$= 10^2\ \ (12) + 10^1\ (6) * (7) – (12 + 8) + 10^0 * 8$

$=\ \ 1200 + 10^1\ \ (42 – 20) + 8$

$= 1200 + 220 + 8$

= **1428**

Multiplication of two four-digit numbers:

A = 981 = 0981 , a_1 = 09 , a_0 = 81

B = 1234 , b_1 = 12, b_0 = 34

A * B = 10^2 c_2 + 10^1 c_1 + 10^0 c_0

= 10^4 a_1 * b_1 + 10^2 (a_1 + a_0) * (b_1 + b_0) – (c_2 + c_0) + 10^0 a_0 * b_0

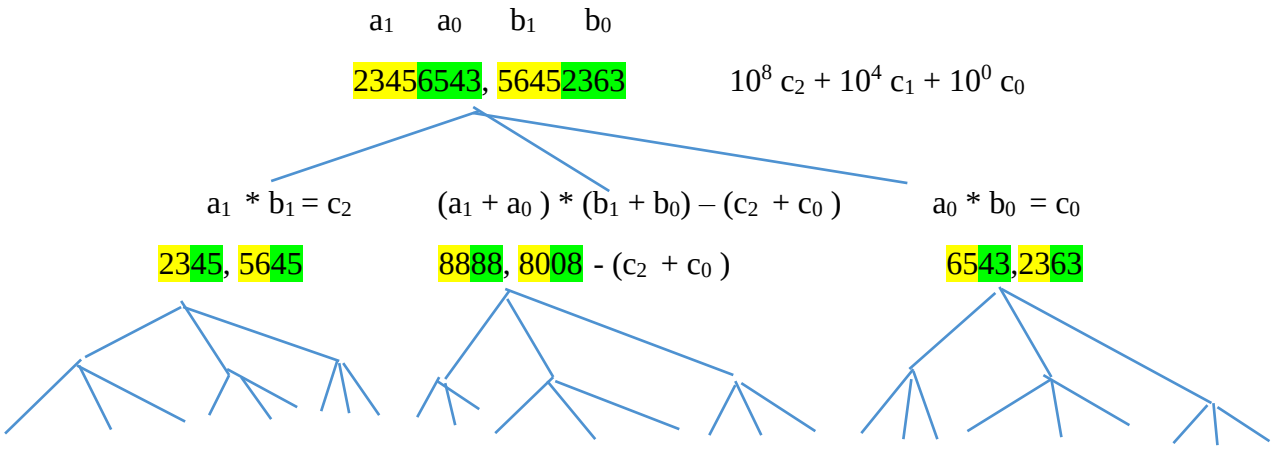
= 10^4 * (12 * 9) 108 + 10^2 (90) * (46) – (108 + 2754) + 10^0 * (81 * 34)2754

= 1080000 + 10^2 (4140 - 2862) + 2754

= 1080000 + 127800 + 2754

= **1210554**

Total 27 ($8^{1.585}$) multiplication are done with two eight-digit numbers as follows:



General formulate is:

10^n c_2 + $10^{n/2}$ c_1 + 10^0 c_0 where n is number of digits in the number and it is even.

The recurrence relation is:

$$T(n) = \begin{cases} 1 & if\ n = 1 \\ 3\,T\left(\frac{n}{2}\right) & if\ n > 1 \end{cases}$$

Time complexity is = O ($n^{\log 3}$) = O($n^{1.585}$) which is less than O(n^2)

5.6 Matrix multiplication:

Let's see how to multiply two matrices of equal dimensions…

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{21} \end{vmatrix} * \begin{vmatrix} b_{11} & b_{12} \\ b_{21} & b_{21} \end{vmatrix} = \begin{vmatrix} c_{11} & c_{12} \\ c_{21} & c_{21} \end{vmatrix}$$

$$C_{i,j} \sum_{k=1}^{n} A_{i,k} * B_{k,j}$$

We need three loops for making multiplications using naïve approach so, time complexity is – $O(n^3)$.

For divide and conquer, which size of matrix is possible to solve?

Before we divide the problem into sub problem, we need to define the solution of problem. So, we take matrix of size 2 * 2.

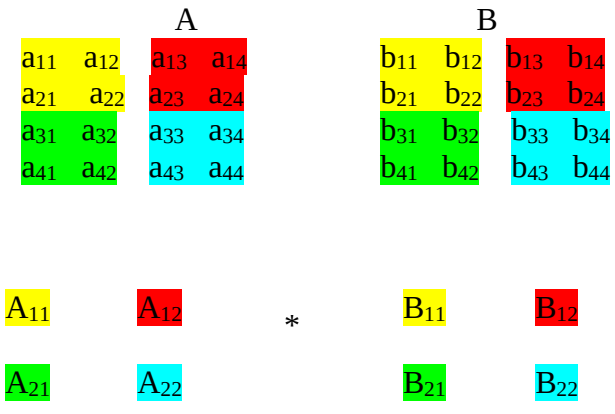
Matrix must be size of power of 2 and greater than 2 * 2.

$$\begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} * \begin{vmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{vmatrix} = \begin{vmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{vmatrix}$$

$$c_{11} = a_{11} * b_{11} + a_{12} * b_{21}$$
$$c_{12} = a_{11} * b_{12} + a_{12} * b_{22}$$
$$c_{21} = a_{21} * b_{11} + a_{22} * b_{21}$$
$$c_{22} = a_{21} * b_{12} + a_{22} * b_{22}$$

Here, we don't take three loops but four operations directly with eight multiplication or four statements which are constant.

$a_{11} * b_{11} = c_{11}$ if this is 1 * 1 matrix then it is called constant.



$$C_{11} = A_{11} * B_{11} + A_{12} * B_{21}$$

$$C_{12} = A_{11} * B_{12} + A_{12} * B_{22}$$

$$C_{21} = A_{21} * B_{11} + A_{22} * B_{21}$$

$$C_{22} = A_{21} * B_{12} + A_{22} * B_{22}$$

Algorithm DAA_MM(A, B, n)

{
if(n <= 2)

{

C = 4 formulas (means constant number of multiplications)

}
else

{

DAA_MM($A_{11}, B_{11}, n/2$) + DAA_MM($A_{12}, B_{21}, n/2$);

DAA_MM($A_{11}, B_{12}, n/2$) + DAA_MM($A_{12}, B_{22}, n/2$);

DAA_MM($A_{21}, B_{11}, n/2$) + DAA_MM($A_{22}, B_{21}, n/2$);

DAA_MM($A_{21}, B_{12}, n/2$) + DAA_MM($A_{22}, B_{22}, n/2$);

}

}

Time complexity using divide and conquer:

Calling of the functions is done 8 times and it is dividing functions so, 8 T(n/2). The addition of the matrix is not simple addition, but it takes n^2 time.

$$T(n) = \begin{cases} 1 & if\ n == 2 \\ 8\,T\left(\frac{n}{2}\right) + n^2 & if\ n > 2 \end{cases}$$

Appy the Master Theorem

a = 8, b= 2, f(n) = n^2

$\log_b a = \log_2 8 = 3$ and $n^k = n^2$ so, k = 2

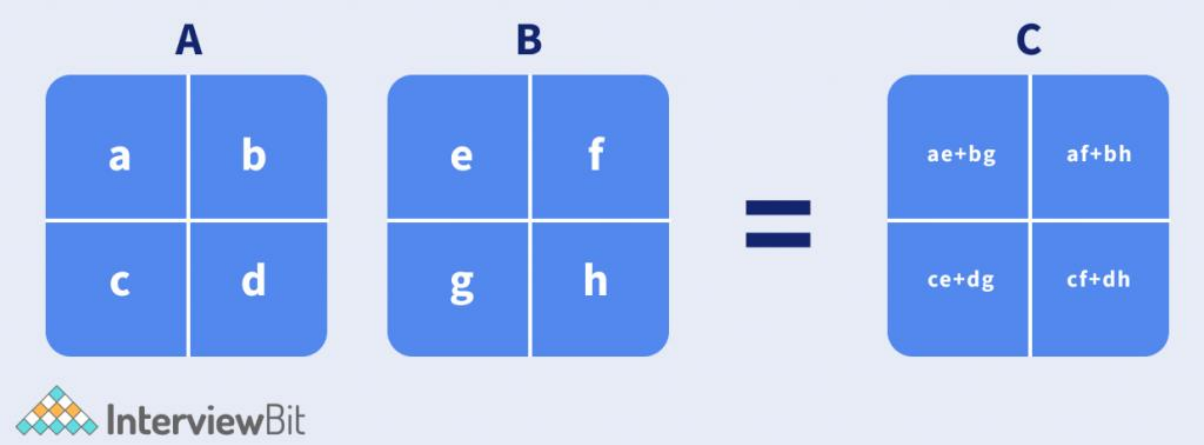
$\log_b a > k$ so, it comes under case -1

$O(n^3)$

If we apply divide and conquer, then it is also same as simple matrix multiplication. But it incurs more time in function calls due to recursive nature.

Strassen's Matrix Multiplication:

It has reduced the number of multiplications to seven instead of eight.



1. a * (f - h)
2. (a + b) * h
3. (c + d) * e
4. d * (g - e)
5. (a + d) * (e + h)
6. (b - d) * (g + h)
7. (a - c) * (e + f)

$$T(n) = \begin{cases} 1 & if\ n \geq 2 \\ 7\,T\left(\frac{n}{2}\right) + n^2 & if\ n > 2 \end{cases}$$

Appy the Master Theorem

a = 7, b= 2, f(n) = n^2

$\log_b a = \log_2 7 = 2.81$ and $n^k = n^2$ so, k = 2

$\log_b a > k$ so, it comes under case -1

$O(n^{2.81})$ which is less than $O(n^3)$

For n = 50, 000: $n^3 \approx 10^{17}$ and $n^{2.808} \approx 10^{13}$; −→ this algorithm is about 10,000 times faster than the naive algorithm.

5.7 Exponential problem using divide and conquer:

a^n - where a is called base and n is called exponent. The base is multiplied n number of times.

```
Pow(x,n)
{
if(n==0)
   return 1;
else
   return (x * Pow(x, n-1));
}
```

Recursive function/relation is as follows:

$$T(n) = \begin{cases} 1 & if\ n == 0 \\ T(n-1) + 1 & if\ n > 0 \end{cases}$$

If we solve the above equation using substitution method then,

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

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

Replace T(n-1) in equation (1)

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

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

T(n) = T(n-3) + 3 (3)

If we repeat k times then

T(n) = T(n-k) + k

Simplified form is T(n) = 1 if n == 0

We assume n – k = 0, n = k

T(n) = T(0) + n

T(n) = 1 + n

T(n) = **O(n)**

Another of writing the algorithm to reduce the complexity of exponential.

$$a^n = \begin{cases} 1 & if\ n == 0 \\ (a^{\frac{n}{2}})^2 & if\ n\ is\ even \\ a * a^{n-1} & if\ n\ is\ odd \end{cases}$$

$X = a^{27}$

$= a * a^{26}$

$= a * (a^{13})^2$

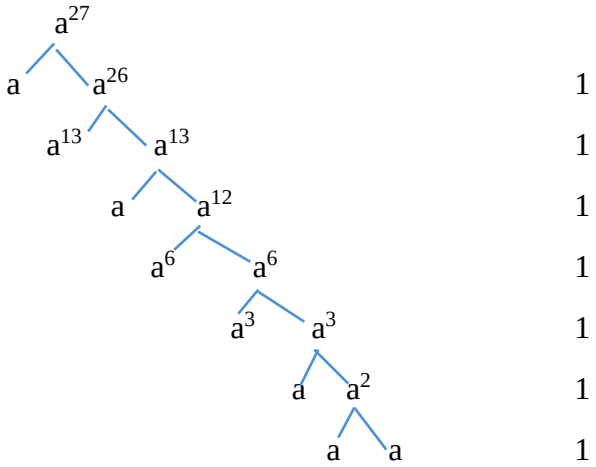
$= a * (a * a^{12})^2$

$= a * (a * (a * (a^6)^2)^2$

$= a * (a * (a * (a^3)^2)^2)^2$

$= a * (a * (a * (a * a^2)^2)^2)^2$

$= a * (a * (a * (a * (a^1)^2)^2)^2)^2$



A total of seven multiplications are done as shown above than total 27 multiplication of traditional approach.

Recursive function is written as follows:

$$T(n) = \begin{cases} 1 & if\ n == 0 \\ T\left(\frac{n}{2}\right) + 1 & if\ n\ is\ even \\ T(n-1) + 1 & if\ n\ is\ odd \end{cases}$$

T(n) = T(n/2) + 1 so, time complexity is O ($\log_2$ n).

5.8 Max-Min Problem using Divide and Conquer approach:

Traditional approach:

There are various ways to this problem, but the most traditional approach to solve this problem is the linear approach. In the linear approach, we traverse all elements once and find the minimum and maximum element.

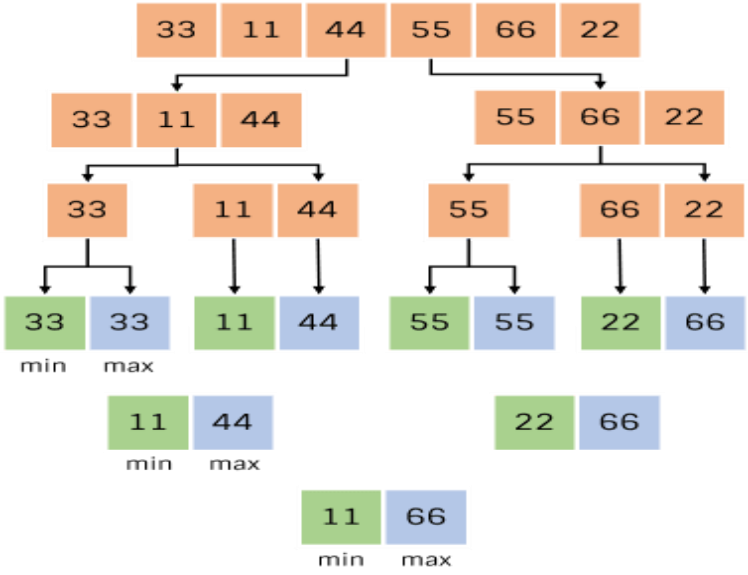
In this approach, the time complexity to solve this problem is O(n).

Let's see the algorithm for the linear approach,
suppose A is the array of integers and n is the number of elements in the array A.

```
MinMax(A, n){
   int min = A[0];
   int max = A[0];
   for(int i = 1; i < n; i++)
  {
      if(max < A[i])
         max = A[i];
      else if(min > A[i])
         min = A[i];
   }
   return (min, max);
}
```

Time Complexity for the above algorithm is T(n) = 2(n-1) + C ≈ O(n).

In the divide and conquer approach, we will divide the problem into sub-problems and find the max and min of each group, now max. of each group will compare with the only max of another group and min with min.



The **orange cell** represents the stage of stepwise division. In the conquer stage, the smallest element from the parent arrays is placed in the **green cell**, while the largest element is stored in the **blue cell**.

Recurrence relation for min-max problem is:

$$T(n) = \begin{cases} 0 & if\ n = 1 \\ 1 & if\ n == 2 \\ 2\,T\left(\frac{n}{2}\right) + 2 & if\ n > 2 \end{cases}$$

T(n) = 2T(n/2) + 2 **(1)**
By substituting n by (n / 2) in Equation (1)

T(n/2) = 2T(n/4) + 2

⇒ T(n) = 2(2T(n/4) + 2) + 2

= 4T(n/4) + 4 + 2 **(2)**
By substituting n by n/4 in Equation (1),

T(n/4) = 2T(n/8) + 2

Substitute it in Equation (1),

T(n) = 4[2T(n/8) + 2] + 4 + 2

= 8T(n/8) + 8 + 4 + 2

= 2^3 T(n/2^3) + 2^3 + 2^2 + 2^1 **(3)**

Repeat this k number of times.

= 2^k T(n/2^k) + 2^k + 2^{k-1} + 2^{k-2} + … + 2^2 + 2^1

Assume n/2^k = 2 so, n = 2^{k+1} and 2^k = n / 2

= n/ 2 T(2^{k+1} / 2^k) + $\sum_{i=0}^{k} 2^i$

= n / 2 T(2) + 2^k – 2 (GP Series)

= n /2 * 1 + (n – 2) (replace 2^k = n here)

= n / 2 + n – 2

= (3n / 2) - 2

Number of comparisons requires applying the divide and conquering algorithm on n elements/items = $\frac{3n}{2} - 2$ = O(n)

CHAPTER – 6

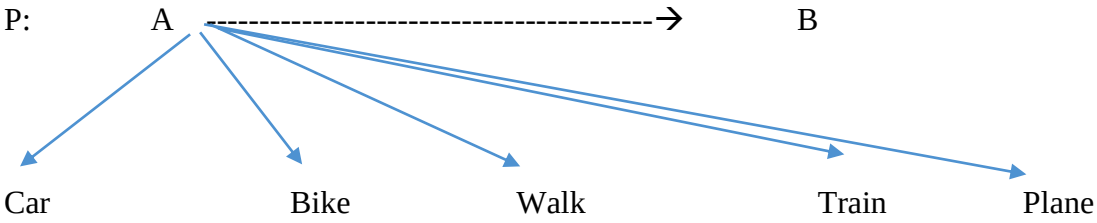
Greedy Method

6.1 Introduction to Greedy Method

This is one the methods to solve the optimization problem. The optimization problems are those they require minimum or maximum result.

For example:

P : A -> B , suppose you have to travel from A to B place .



But there is constraint in the problem that it must be completed in 10 hours. Then feasible solutions are only Train and plane means they are satisfying the constraints.

Minimum cost of travelling from A to B that is minimization problem.

If we go by train, then it will have minimum cost of travelling and it is called optimal solution (which is achieving the objective of the problem).

An optimal solution is a feasible solution that results in the largest possible objective function value when maximizing (or smallest when minimizing).

The following methods are used to solve the optimization problem:

1. Greedy
2. Branch and Bound
3. Dynamic Programming

Greedy Method:

The Greedy method is the simplest and straightforward approach. It is not an algorithm, but it is a technique. The main function of this approach is that the decision is taken on the basis of the currently available information. Whatever the current information is present, the decision is made without worrying about the effect of the current decision in future.

This technique is basically used to determine the feasible solution that may or may not be optimal. The feasible solution is a subset that satisfies the given criteria. The optimal solution is the solution which is the best and the most favorable solution in the subset. In the case of feasible, if more than one solution satisfies the given criteria then those solutions will be considered as feasible, whereas the optimal solution is the best solution among all the solutions.

The greedy method solves the problem into stages. In each stage it considers one input and checks whether it is feasible or not. If it is feasible then add it to the list of feasible solutions.

Suppose we are given any problem A having list of inputs: n = 5.

A1	A2	A3	A4	A5

In general, we can use following algorithm for solving any problem using greedy method.

```
Algorithm Greedy(A, n)
{
        for(i=1;i<=n;i++)
            {
                x = select(A);
                if (x is feasible)
                   Solution = Solution + x;
              }
         return Solution;
 }
```

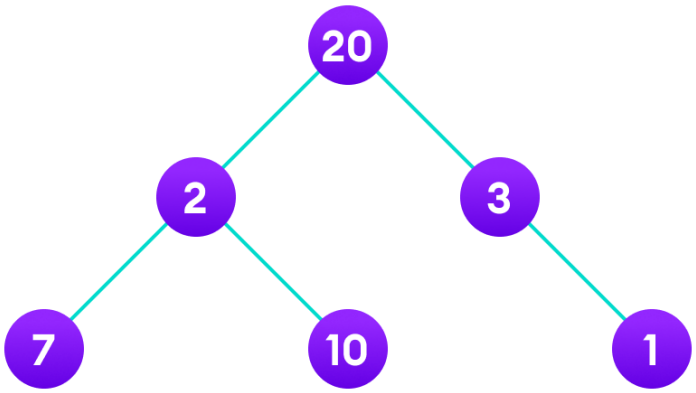
Advantages of Greedy Approach

- The algorithm is **easier to describe**.
- This algorithm can **perform better** than other algorithms (but, not in all cases).

Drawback of Greedy Approach

- As mentioned earlier, the greedy algorithm doesn't always produce the optimal solution. This is the major disadvantage of the algorithm.

For example, suppose we want to find the longest path in the graph below from root to leaf. Let's use the greedy algorithm here.



Greedy Approach

1. Let's start with the root node **20**. The weight of the right child is **3** and the weight of the left child is **2**.

2. Our problem is to find the largest path. And the optimal solution at the moment is **3**. So, the greedy algorithm will choose **3**.

3. Finally the weight of an only child of **3** is **1**. This gives us our final result `20 + 3 + 1 = 24`. However, it is not the optimal solution. There is another path that carries more weight (`20 + 2 + 10 = 32`)

We follow some known method of solving the problem that is Greedy method.

Examples:

1. If we have to purchase the best car in terms of features, then we follow the greedy approach for selecting best car.
2. If we have to select the best candidate for job out of 1000 then we do conduct written test, GD, PI and select the best candidate. It may not be the best. If we give chance to all candidates for GD, PI then we can select the best candidate.

6.2 Knapsack Problem: (fraction)

Here knapsack is like a container or a bag. Suppose we have given some items which have some weights or profits. We have to put some items in the knapsack in such a way total value produces a maximum profit.

For example, the weight of the container is 20 kg. We have to select the items in such a way that the sum of the weight of items should be either smaller than or equal to the weight of the container, and the profit should be maximum.

There are two types of knapsack problems:

- 0/1 knapsack problem
- Fractional knapsack problem

What is the 0/1 knapsack problem?

The 0/1 knapsack problem means that the items are either completely or no items are filled in a knapsack. For example, we have two items having weights 2kg and 3kg, respectively. If we pick the 2kg item, then we cannot pick 1kg item from the 2kg item (item is not divisible); we have to pick the 2kg item completely. This is a 0/1 knapsack problem in which either we pick the item completely or we will pick that item. The 0/1 knapsack problem is solved by dynamic programming.

What is the fractional knapsack problem?

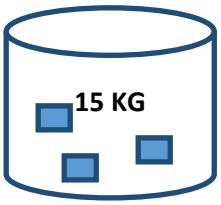
The fractional knapsack problem means that we can divide the item. For example, we have an item of 3 kg then we can pick the item of 2 kg and leave the item of 1 kg. The fractional knapsack problem is solved by the Greedy approach.

We are given n=7 objects and we have to put them in bag of m=15 kg which maximize the profit. The object must be divisible into piece then this can be applied. TV, washing machine etc. can't be put in fraction.

It is also called container loading problem.

We have to fill the objects in a bag in such a way that it can maximize the profit.

Object	1	2	3	4	5	6	7
Profit - P	10	5	15	7	6	18	3
Weight - W	2	3	5	7	1	4	1



Object	1	2	3	4	5	6	7
Values	0-1						

$0 <= x <= 1$

We can include the fraction of the object.

There are three possible solutions to fill the objects in the bag.

- The first approach is to select the item based on the maximum profit.
- The second approach is to select the item based on the minimum weight.
- The third approach is to calculate the ratio of profit/weight.

Greedy method:

We calculate profit/weight and based on it we can select the objects. This is the method we can follow for selecting the objects for putting in the bag.

Object	1	2	3	4	5	6	7
Profit – P	10	5	15	7	6	18	3
Weight - W	2	3	5	7	1	4	1
Profit/Weight	**5**	**1.3**	**3**	**1**	**6**	**4.5**	**3**

Object	1	2	3	4	5	6	7

We will put 1 in the respective column of the object if that object is included in the solution.

Step 1:

Object 5 is selected based on P/W ratio so, remaining weight in bag is 15 – 1 = 14

Object	1	2	3	4	5	6	7
					1		

Step 2:

Now, object 1 is selected so, remaining weight in the bag is 14 – 2 = 12

Object	1	2	3	4	5	6	7
	1				1		

Step 3:

Object 6 is selected so, remaining weight in the bag is 12 – 4 = 8

Object	1	2	3	4	5	6	7
	1				1	1	

Step 4:

Next, we can select either object 3 or 7 (no matter it is P/W so, we can select anyone), so, we select object 3 and remaining weight in the bag is 8 – 5 = 3.

Object	1	2	3	4	5	6	7
	1		1		1	1	

Step 5:

Object 7 is selected so, remaining weight in the bag is 3 -1 = 2

Object	1	2	3	4	5	6	7
	1	2/3	1	0	1	1	1

Step 6:

Now, we can include object 2 but by including it exceeds the bag capacity so, we include only 2/3 of it and remaining bag capacity is 2 - 2 = 0

The total weights included in the solution is:

Object	1	2	3	4	5	6	7
Xi	1	2/3	1	0	1	1	1
W_i-Weight	2	3	5	7	1	4	1
$X_i * W_i$	1 * 2	2/3 * 3	1 * 5	7 * 0	1 * 1	4 * 1	1 * 1

$$\sum_{i=1}^{n} X_i W_i$$

Now, total profit of including weight in the bag is:

Object	1	2	3	4	5	6	7
Xi	1	2/3	1	0	1	1	1
P_i-Profit	10	5	15	7	6	18	3
$X_i * P_i$	1 * 10	2/ 3 * 5	1 * 15	0 * 7	1 * 6	1 * 18	1 * 3

$\sum_{i=1}^{n} X_i P_i$ = 1 * 10 + 2/3 * 5 + 1 * 15 + 0 * 7 + 1 * 6 + 1 * 18 + 1 * 3 = **54.6**

The constraint given is:

$$\sum_{i=1}^{n} X_i W_i \leq m$$

Objective is:

Maximize - $\sum_{i=1}^{n} X_i P_i$

Time complexity:

Naïve approach: Try all possible subsets with all different fractions. – $O(2^n)$

Greedy approach:

The algorithm uses sorting to sort the items which takes **O(n×logn)** time complexity and then loops through each item which takes **O(n)**. Hence summing up to a time complexity of **O(n×logn+n)**=**O(n×logn)**.

Example:

Find the maximum profit for the following knapsack problem using greedy method.

n = 7 and m = 15

Objects	1	2	3	4	5	6	7
Profit	5	10	15	7	8	9	4
Weight	1	3	5	4	1	3	2

Solution:

Based on maximum profit – 47.25, based on minimum weight - 46

Total profit based on profit/weight would be equal to (8 + 5 + 10 + 15 + 9 + 4), i.e., 51.

6.3 Making change problem:

We are given the coins of different denominations and a value X. We need to find the minimum number of coins required to make value X. We have infinite supply of each coin.

Follow the steps below to implement the idea:

- Sort the array of coins in decreasing order.
- Initialize **ans** vector as empty.
- Find the largest denomination that is smaller than **remaining amount** and while it is smaller than the **remaining amount**:
    - Add found denomination to **ans**. Subtract value of found denomination from **amount**.
- If amount becomes **0**, then print **ans**.

Algorithm **ChangeCoin**(denom[], value)

{
S = Ø # it set to hold the solution

sum = 0 # sum of items in set S

while sum != value

x = select the largest item/coin from denom[] such that sum + x <= value ;

if there is no such item found then

return “no solution found”;

S = S U {coin value of x} ;

sum = sum + x;

return S;

}

Example 1:

denom[] = { 10, 15, 60} and value= 40

solution: S = {15, 15, 10}

Example 2:

denom[] = { 1, 5, 10, 20} and value= 50

solution: S = {20, 20, 10}

Note: The above approach may not work for all denominations.

For example, it doesn't work for denominations {9, 6, 5, 1} and V = 11. The above approach would print 9, 1 and 1. But we can use 2 denominations 5 and 6.

For general input, a dynamic programming approach can be used.

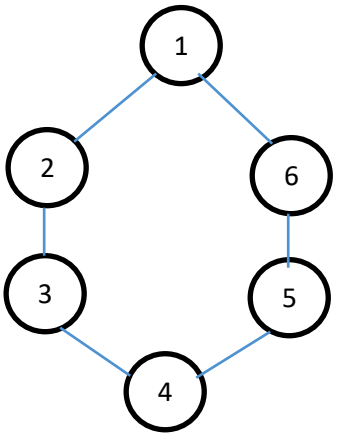
Time complexity:

1. For sorting n coins O(nlogn).
2. While loop, the worst case is O(total). If all we have is the coin with 1-denomination.
3. Complexity for coin change problem becomes O(n log n) + O(total).

6.4 Minimum Spanning Tree (MST):

G = { V, E}

V = { 1, 2, 3, 4, 5, 6 }
E = { (1,2), (2,3), (3,4), (4,5), (5,6), (6,1)}



Spanning Tree:

It is subgraph (tree) of the original graph which includes all the vertices and edges are one less than number of vertices. It does not have cycle.

Minimum Spanning Tree:

Minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree.

Possible number of spanning trees are = $|E|_{C_{|V|-1}}$ – no. of cycles # (n! / r! (n-r)!)

Prim's algorithm:

Prim's algorithm is a greedy algorithm that starts from one vertex and continue to add the edges with the smallest weight until the goal is reached. The steps to implement the prim's algorithm are given as follows -

- First, we have to initialize an MST with the randomly chosen vertex.
- Now, we have to find all the edges that connect the tree in the above step with the new vertices. From the edges found, select the minimum edge and add it to the tree.
- Repeat step 2 until the minimum spanning tree is formed.

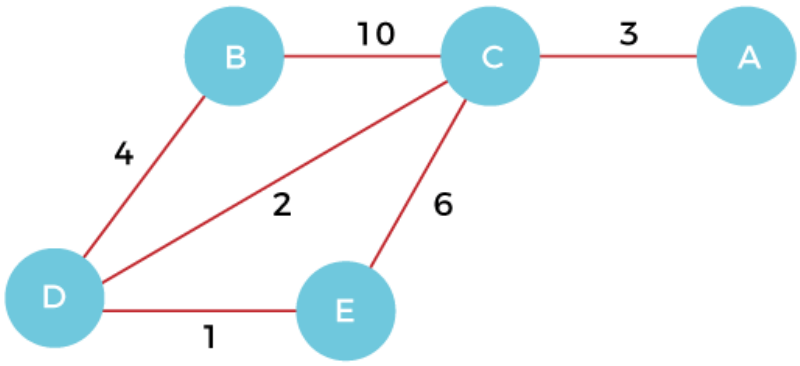
The applications of prim's algorithm are -

- Prim's algorithm can be used in network designing.
- It can be used to make network cycles.
- It can also be used to lay down electrical wiring cables.

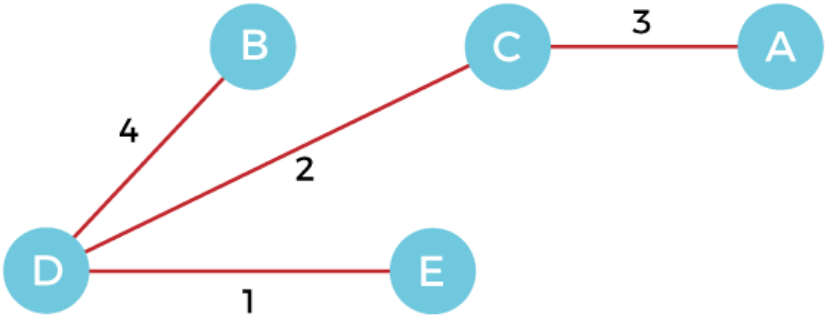
Time complexity:

Prim's algorithm has a time complexity of $O(V^2)$, V being the number of vertices and can be improved up to O(E log V) using Fibonacci heaps.

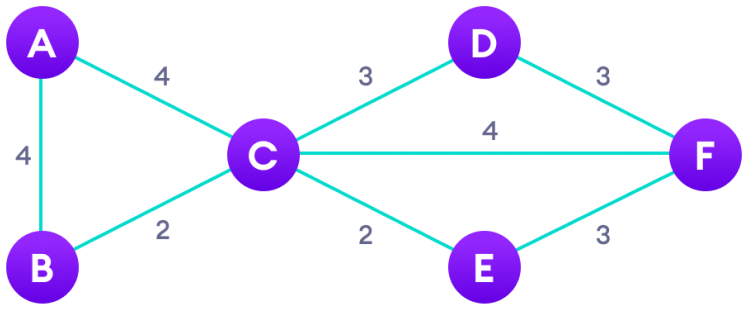
Example 1: Find the MST for the following weighted graph:



MST will be as follows:

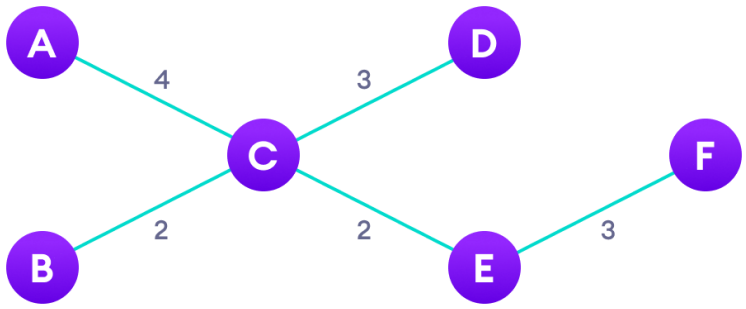


Example 2: Find the MST for the following graph:



Step: 1

MST will be as follows:



Step: 6

Kruskal's algorithm:

In Kruskal's algorithm, we start from edges with the lowest weight and keep adding the edges until the goal is reached. The steps to implement Kruskal's algorithm are listed as follows -

- First, sort all the edges from low weight to high.
- Now, take the edge with the lowest weight and add it to the spanning tree. If the edge to be added creates a cycle, then reject the edge.
- Continue to add the edges until we reach all vertices, and a minimum spanning tree is created.

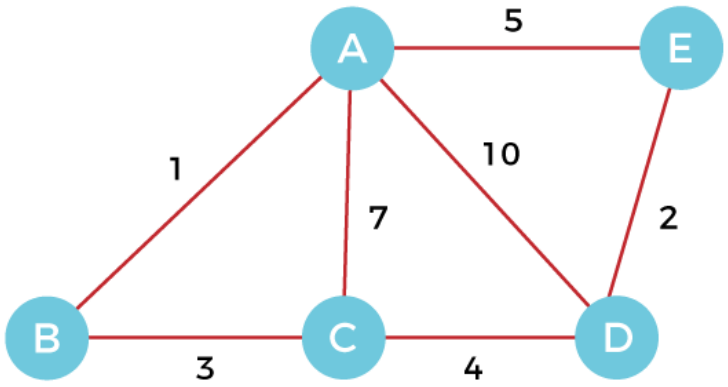
The applications of Kruskal's algorithm are -

- Kruskal's algorithm can be used to layout electrical wiring among cities.
- It can be used to lay down LAN connections.

Time complexity:

The Kruskal method has an O(E logE) or O(V logV) time complexity, where E is the number of edges and V is the number of vertices.

Example 1: Find the MST of the following graph:



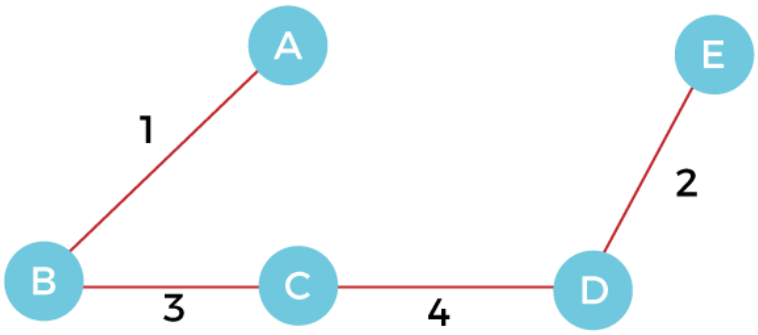
The weight of the edges of the above graph is given in the below table -

Edge	AB	AC	AD	AE	BC	CD	DE
Weight	1	7	10	5	3	4	2

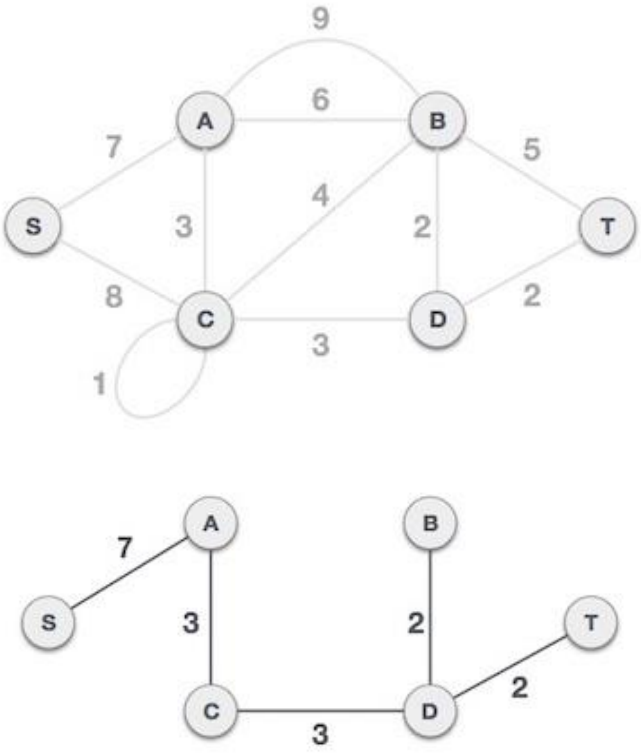
Now, sort the edges given above in the ascending order of their weights.

Edge	AB	DE	BC	CD	AE	AC	AD
Weight	1	2	3	4	5	7	10

MST will be as follows:



Example 2:



The total cost of MST = 17

6.5 Dijkstra algorithm:

Shortest path – Singe source shortest path (source to all vertices)

Given a graph and a source vertex in the graph, find the **shortest paths** from the source to all vertices in the given graph.

Follow the steps below to solve the problem:

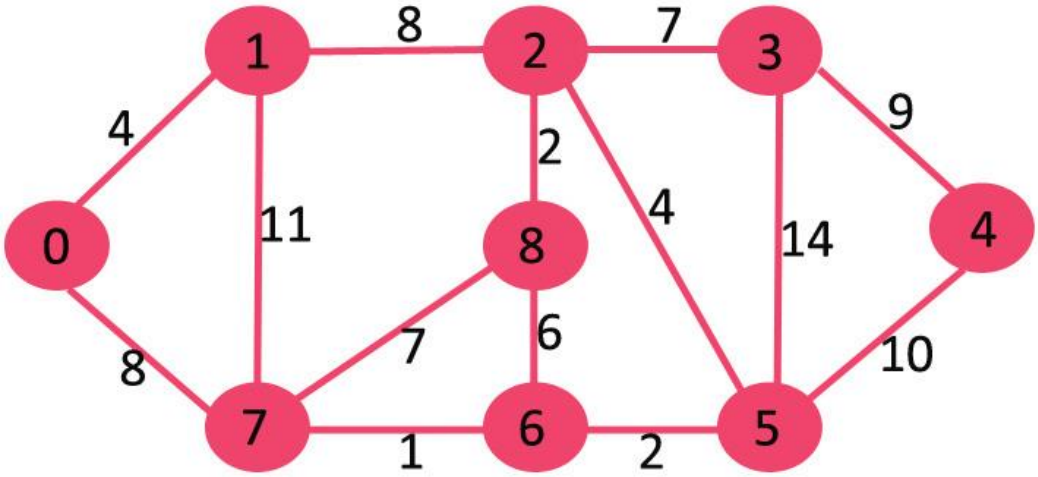
- Create a set **sptSet** (shortest path tree set) that keeps track of vertices included in the shortest path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
- Assign a distance value to all vertices in the input graph. Initialize all distance values as **INFINITE**. Assign the distance value as 0 for the source vertex so that it is picked first.
- While **sptSet** doesn't include all vertices
    - Pick a vertex **u** that is not there in **sptSet** and has a minimum distance value.
    - Include u to **sptSet**.
    - Then update the distance value of all adjacent vertices of u.
        - To update the distance values, iterate through all adjacent vertices.
        - For every adjacent vertex v, if the sum of the distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

Note: We use a boolean array sptSet[] to represent the set of vertices included in SPT. If a value sptSet[v] is true, then vertex v is included in SPT, otherwise not. Array dist[] is used to store the shortest distance values of all vertices.

Time complexity:

In worst case, time complexity is $O(|V|^2)$ or $O(n^2)$ where V is number of vertices and for each vertex V, we need to relax all remaining vertices (V-1) if we have complete graph.

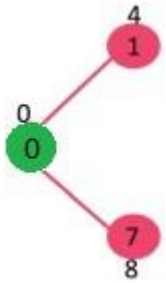
Example 1:



Step 1:

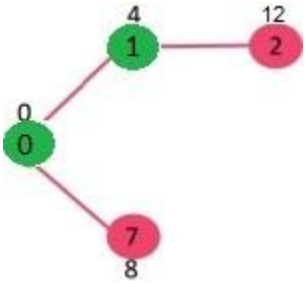
- The set **sptSet** is initially empty and distances assigned to vertices are {0, INF, INF, INF, INF, INF, INF, INF} where **INF** indicates infinite.
- Now pick the vertex with a minimum distance value. The vertex 0 is picked, include it in sptSet. So sptSet becomes {0}. After including 0 to sptSet, update distance values of its adjacent vertices.
- Adjacent vertices of 0 are 1 and 7. The distance values of 1 and 7 are updated as 4 and 8.

- The following subgraph shows vertices and their distance values, only the vertices with finite distance values are shown. The vertices included in SPT are shown in **green** colour.



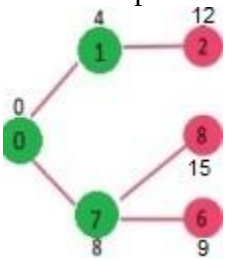
Step 2**:**

- Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). The vertex 1 is picked and added to sptSet.
- So sptSet now becomes {0, 1}. Update the distance values of adjacent vertices of 1.
- The distance value of vertex 2 becomes **12**.



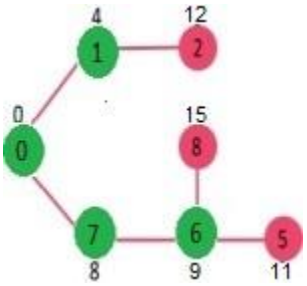
Step 3**:**

- Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 7 is picked. So sptSet now becomes **{0, 1, 7}.**
- Update the distance values of adjacent vertices of 7. The distance value of vertex 6 and 8 becomes finite (**15 and 9** respectively).

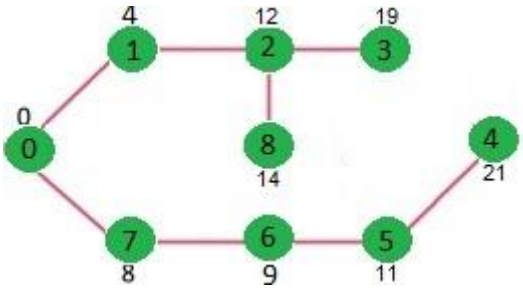


Step 4**:**

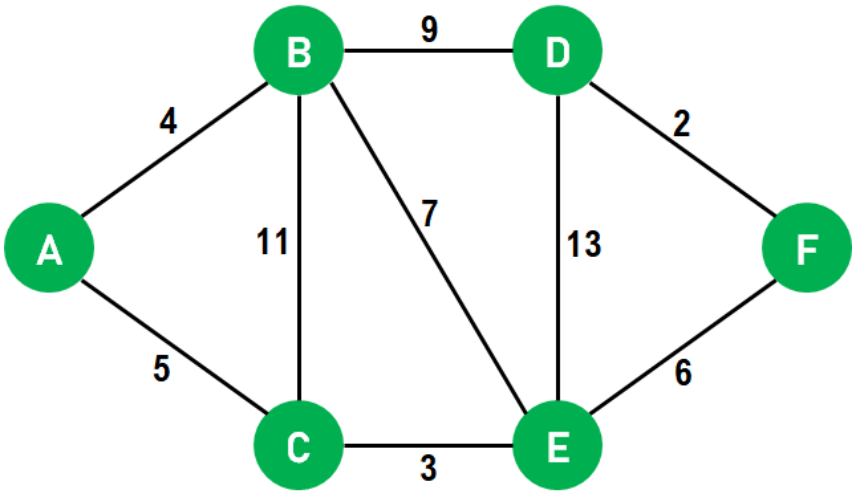
- Pick the vertex with minimum distance value and not already included in SPT (not in sptSET). Vertex 6 is picked. So sptSet now becomes **{0, 1, 7, 6}**.
- Update the distance values of adjacent vertices of 6. The distance value of vertex 5 and 8 are updated.



We repeat the above steps until sptSet **includes** all vertices of the given graph. Finally, we get the following Shortest Path Tree (SPT).



Example 2:



1. A = 0
2. B = 4 (A -> B)
3. C = 5 (A -> C)
4. D = 4 + 9 = 13 (A -> B -> D)
5. E = 5 + 3 = 8 (A -> C -> E)
6. F = 5 + 3 + 6 = 14 (A -> C -> E -> F)

6.6 Huffman Code:

Suppose one message is being sent from one system to another.

Message - **B C C A B B D D A E C C B B A E D D C C**

If ASCI coding is used, then each character takes 8 bit for encoding.

Total 20 characters are there so, message size would be 20 * 8 = 160 bytes

Symbol	ASCII Code	Binary – 8 bit	Frequency	Code
A	65	01000001	3	000
B	66	01000010	5	001
C	67	01000011	6	010
D	68	01000100	4	011
E	69		2	100

The number of symbols used are only 5 so, do we need 8 bits to encode them? No.

Fixed length coding:

Three bits are sufficient to encode five symbols.

Total number of bits required to send the message is 20 * 3 = 60 bits

Also, the table of codes need to be sent with the message for decoding purpose and its size would be = 5 * 8 (each character is of 8 bit) + 5 * 3 (each character's code is 3 bit) = 55 bits

Total size – message + table = 60 + 55 = 115 bits and it is less than 160 bits. 30=35% reduction in size.

Variable length coding – Huffman code

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.

The variable-length codes assigned to input characters are Prefix Codes, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.

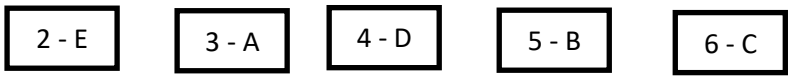
Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be "cccd" or "ccb" or "acd" or "ab".

Less size code to the more appearing character and more size to less appearing character so, it would reduce the size of message being sent.

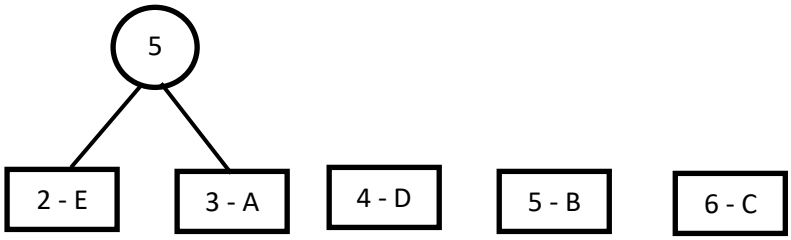
Symbol	Frequency	Code
A	3	001
B	5	10
C	6	11
D	4	01
E	2	000

The Huffman tree is constructed as follows:

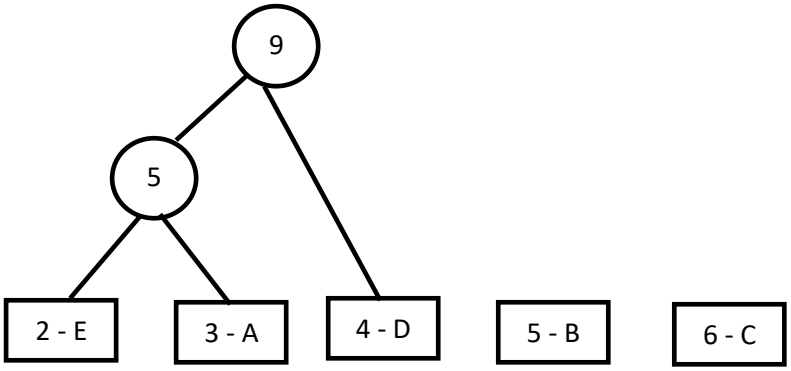
Arrange all the characters/alphabets in order of their frequency.



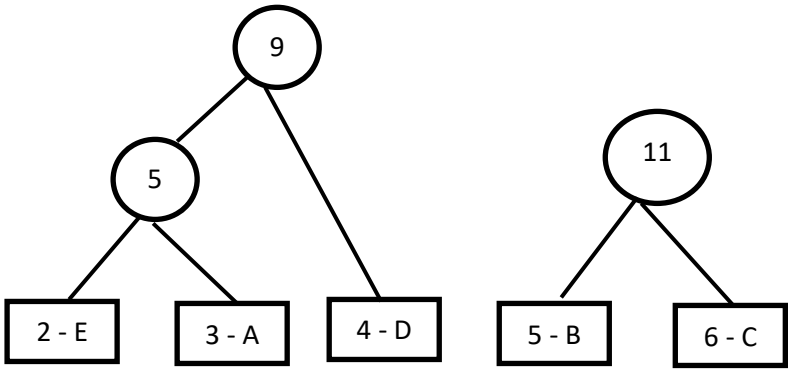
Step 1: Select two least numbers and merge them.



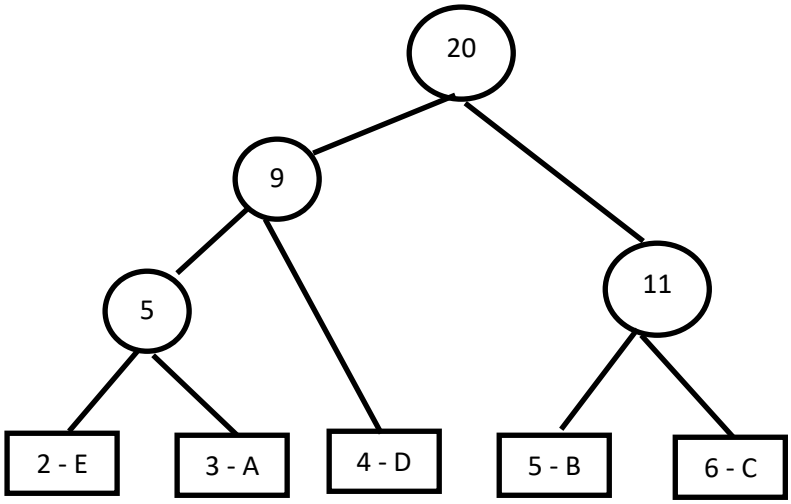
Step 2: Select two least numbers which are 4, 5 and 5 so, select any of the pair. We select as follows.



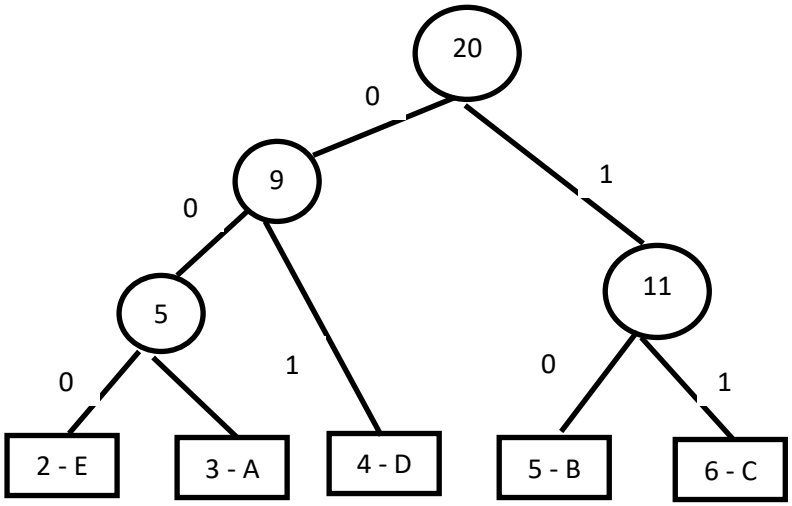
Step 3: Select two least numbers which are 5 and 6.



Step 4: Select the remaining two numbers which are 9 and 11.



Step 5: Assign 0 to left edges and 1 to right edges.



Step 6: Now assign the code to each character: start the from the root and travers towards the leaf.

For example: E is assigned the code 000 and A is assigned 001.

Symbol	**Frequency**	**Code**	**Size**
A	3	001	3 * 3 = 9
B	5	10	5 * 2 = 10
C	6	11	6 * 2 = 12
D	4	01	4 * 2 = 8
E	2	000	2 * 3 = 6
		Total = 12 bits	Total = 45 bits

Total 45 bits message size, size of the table is 5 * 8 + 12 bits= 52 bits.

So, total size = 45 + 52 = 97 and it is less 115 bits of fixed length code.

Message size will be = $\sum \boldsymbol{d_i f_i}$ where d_i is the distance from the root to leaf and f_i is the frequency of symbol/character.

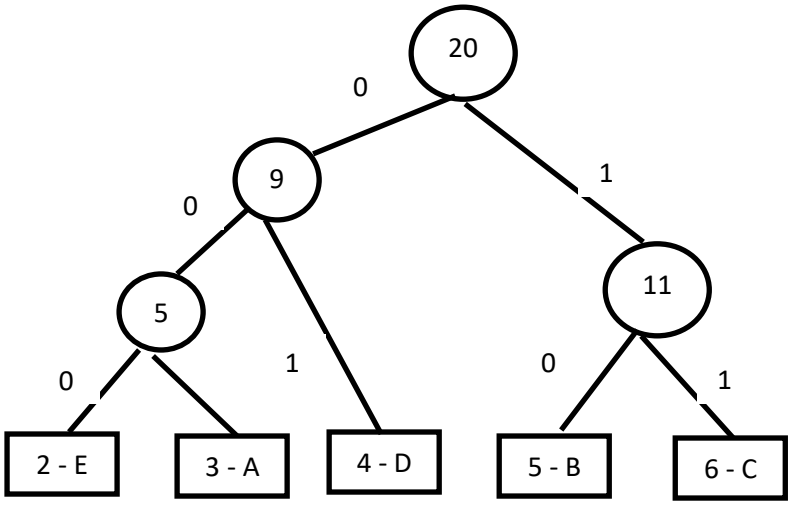
Encoding of the message:

B C C A B B D D A E C C B B A E D D C C

10111100110100101001000111110100010000101111 1

Decoding of the message:

Start the taking the bits from the incoming message and follow the branch of the tree (0 then left and 1 then right) from root to leaft. Once we reach to the leaf, we get the decoded character. Now resume from the next bit in the incoming message and do the same process.



10111100110100101001000111110100010000101111 1

B C C A B B D D …….

6.7 Job Scheduling (sequencing) Problem:

Given an array of jobs where every job has a deadline and associated profit if the job is finished before the deadline. It is also given that every job takes a single unit of time, so the minimum possible deadline for any job is 1. Maximize the total profit if only one job can be scheduled at a time.

The following five jobs are given with profit and deadline.

Job ID	**Profit**	**Deadline**
A	100	2
B	19	1
C	27	2
D	25	1
E	15	3

Naive Approach**:** To solve the problem follow the below idea:

Generate all subsets of a given set of jobs and check individual subsets for the feasibility of jobs in that subset. Keep track of maximum profit among all feasible subsets.

Greedy Approach:

Follow the given steps to solve the problem:

- Sort all jobs in decreasing order of profit.
- Iterate on jobs in decreasing order of profit. For each job , do the following :
    - Find a time slot i, such that slot is empty and i < deadline and i is greatest. Put the job in this slot and mark this slot filled.
    - If no such i exists, then ignore the job.

The jobs are sorted in decreasing order of their profit.

Job ID	**Profit**	**Deadline**
A	100	2
C	27	2
D	25	1
B	19	1
E	15	3

Step 1:

Take the first job and find the slot so we can put it in the slot of 1-2 because its deadline is 2.

	A - 100	
0	1	2 3

Step 2:

Take the second job C and its deadline is 2. But already another job A is scheduled in 1-2 so, we cannot put there. But slot 0-1 is free so, put it in the slot 0-1.

C-27	A - 100	
0	1	2 3

Step 3:

Take the next job D but it is not possible to schedule it as slots are not available as per its deadline. Same for the next job B. Then we take the next job E and its deadline is 3 and the slot is also available so, put it in the slot of 2-3.

C-27	A - 100	E-15
0	1	2 3

Maximum profit is = 27 + 100 + 15 = 142.

Example 1:

Solve the following instance of "job scheduling with deadlines" problem : n = 7, profits (p_1, p_2, p_3, p_4, p_5, p_6, p_7) = (3, 5, 20, 18, 1, 6, 30) and deadlines (d_1, d_2, d_3, d_4, d_5, d_6, d_7) = (1, 3, 4, 3, 2, 1, 2). Schedule the jobs in such a way to get maximum profit.

Solution:

Given that,

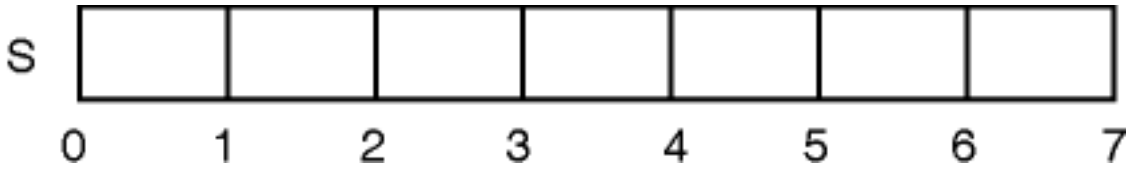
Jobs	J_1	J_2	J_3	J_4	J_5	J_6	J_7
Profit	3	5	20	18	1	6	30
Deadline	1	3	4	3	2	1	2

Sort all jobs in descending order of profit.

So, P = (30, 20, 18, 6, 5, 3, 1), J = (J_7, J_3, J_4, J_6, J_2, J_1, J_5) and D = (2, 4, 3, 1, 3, 1, 2). We shall select one by one job from the list of sorted jobs J, and check if it satisfies the deadline. If so,

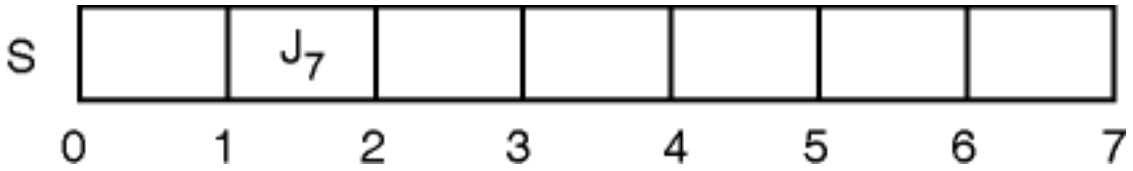
schedule the job in the latest free slot. If no such slot is found, skip the current job and process the next one.

Initially, Profit of scheduled jobs, SP = 0



Iteration 1:

Deadline for job J_7 is 2. Slot 2 (t = 1 to t = 2) is free, so schedule it in slot 2. Solution set S = {J_7}, and Profit SP = {30}



Iteration 2:

Deadline for job J_3 is 4. Slot 4 (t = 3 to t = 4) is free, so schedule it in slot 4. Solution set S = {J_7, J_3}, and Profit SP = {30, 20}

Iteration 3:

Deadline for job J_4 is 3. Slot 3 (t = 2 to t = 3) is free, so schedule it in slot 3.

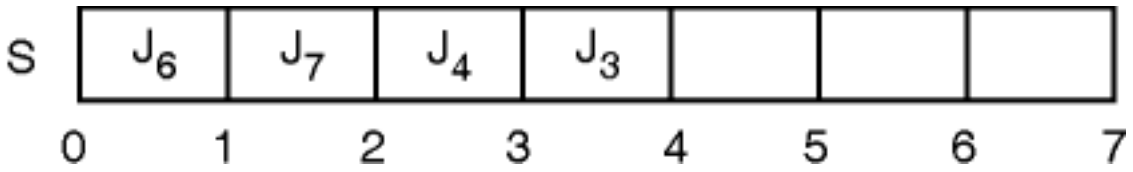
Solution set S = {J_7, J_3, J_4}, and Profit SP = {30, 20, 18}

Iteration 4:

Deadline for job J_6 is 1. Slot 1 (t = 0 to t = 1) is free, so schedule it in slot 1.

Solution set S = {J_7, J_3, J_4, J_6}, and Profit

SP = {30, 20, 18, 6}



First, all four slots are occupied and none of the remaining jobs has deadline lesser than 4. So none of the remaining jobs can be scheduled. Thus, with the greedy approach, we will be able to schedule four jobs {J_7, J_3, J_4, J_6}, which give a profit of (30 + 20 + 18 + 6) = 74 units.

Example 2:

Find the profit for the following jobs.

Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	10

Solution:

= Sum of profit of all the jobs in optimal schedule

= Profit of job J2 + Profit of job J4 + Profit of job J3 + Profit of job J5 + Profit of job J1

= 180 + 300 + 190 + 120 + 200 = 990 units

CHAPTER – 7 Dynamic Programming

7.1 Introduction to Dynamic Programming

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.

There are two main approaches to dynamic programming:

Top-down approach (Memorization):

In computer science, problems are resolved by recursively formulating solutions, employing the answers to the problems' subproblems. If the answers to the subproblems overlap, they may be memorized or kept in a table for later use. The top-down approach follows the strategy of memorization. The memorization process is equivalent to adding the recursion and caching steps. The difference between recursion and caching is that recursion requires calling the function directly, whereas caching requires preserving the intermediate results.

The top-down strategy has many benefits, including the following:

1. The top-down approach is easy to understand and implement. In this approach, problems are broken down into smaller parts, which help users identify what needs to be done. With each step, more significant, more complex problems become smaller, less complicated, and, therefore, easier to solve. Some parts may even be reusable for the same problem.
2. It allows for subproblems to be solved upon request. The top-down approach will enable problems to be broken down into smaller parts and their solutions stored for reuse. Users can then query solutions for each part.
3. It is also easier to debug. Segmenting problems into small parts allows users to follow the solution quickly and determine where an error might have occurred.

Disadvantages of the top-down approach include:

1. The top-down approach uses the recursion technique, which occupies more memory in the call stack. This leads to reduced overall performance. Additionally, when the recursion is too deep, a stack overflow occurs.

Bottom-up approach (Tabulation):

In the bottom-up method, once a solution to a problem is written in terms of its subproblems in a way that loops back on itself, users can rewrite the problem by solving the smaller subproblems first and then using their solutions to solve the larger subproblems.

Unlike the top-down approach, the bottom-up approach removes the recursion. Thus, there is neither stack overflow nor overhead from the recursive functions. It also allows for saving memory space. Removing recursion decreases the time complexity of recursion due to recalculating the same values.

The advantages of the bottom-up approach include the following:

1. It makes decisions about small reusable subproblems and then decides how they will be put together to create a large problem.
2. It removes recursion, thus promoting the efficient use of memory space. Additionally, this also leads to a reduction in timing complexity.

When one can apply dynamic programing to solve the problem?

Dynamic programming solves complex problems by breaking them up into smaller ones using recursion and storing the answers, so they don't have to be worked out again. It isn't practical when there aren't any problems that overlap because it doesn't make sense to store solutions to the issues that won't be needed again.

Two main signs are that one can solve a problem with dynamic programming: subproblems that overlap and the best possible substructure.

Overlapping subproblems

When the answers to the same subproblem are needed more than once to solve the main problem, we say that the subproblems overlap. In overlapping issues, solutions are put into a table so developers can use them repeatedly instead of recalculating them. The recursive program for the Fibonacci numbers has several subproblems that overlap, but a binary search doesn't have any subproblems that overlap.

A binary search is solved using the divide and conquer technique. Every time, the subproblems have a unique array to find the value. Thus, binary search lacks the overlapping property.

For example, when finding the nth Fibonacci number, the problem F(n) is broken down into finding F(n-1) and F. (n-2). You can break down F(n-1) even further into a subproblem that has to do with F. (n-2). In this scenario, F(n-2) is reused, and thus, the Fibonacci sequence can be said to exhibit overlapping properties.

Optimal substructure:

The optimal substructure property of a problem says that you can find the best answer to the problem by taking the best solutions to its subproblems and putting them together. **(**Optimal solution to a problem contains an optimal solution of the sub problems**.)**

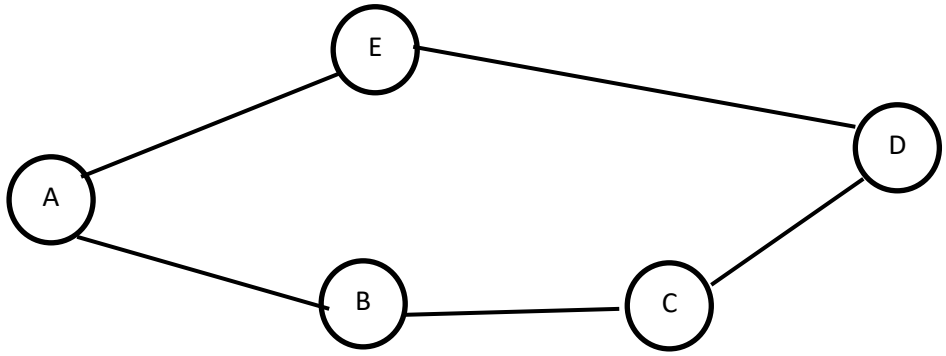
This property is not exclusive to dynamic programming alone, as several problems consist of optimal substructures. However, most of them lack overlapping issues. So, they can't be solved with dynamic programming.

What is the difference between Optimal Substructure and Overlapping Subproblems?

In optimal substructure, we use the optimal answer of the subproblems to find the optimal answer for a bigger problem. Whereas in Overlapping subproblems we store the answer for the Overlapping subproblems so that we don't have to re-compute the answer for the same subproblem again.

For example:

The following example demonstrates the optimal substructure property for the path finding problem.



- Shortest path problem exhibits optimal substructure property:

S(A,D) = S(A,E) + S(E,D)

The shortest path from A to E and E to D gives the shortest path A to D. So, sub problem's optimal solution gives optimal solution of the problem.

- Longest path problem does not exhibit optimal substructure property:

L(A, C) = L(A,D) + (D,C)

There are two possible path from A to C: one is (A, B), (B, C) and another is (A,E), (E,D), (D,C). The second one gives the longest path.

= (A, E), (E, D), (D,C) + (D,C)

If we add this path to the edge (D,C), there would be a repeating edge, i.e., D to C which is not allowed. So, in this case, the sub-solutions to a problem do not combine to form the overall optimal solution. Therefore, we can say that the Longest-path problem does not exhibit an Optimal substructure.

Problems which can be solved using dynamic programming:

The technique is widely used to solve problems in various domains, such as optimization problems, combinatorial problems, graph algorithms, and more. Classic examples of problems solved using dynamic programming include the Fibonacci sequence, the knapsack problem, shortest paths in graphs (e.g., Floyd-Warshall algorithm), and longest common subsequence, among others.

First, we need to clear the difference between Greedy method and Dynamic Programming. Both are used to solve optimization problems. Optimization problems are those which need minimum or maximum result.

Greedy Method	Dynamic Programming
We try to follow the predefined procedure. But the procedure to be known to the optimum.	We try to find all the possible solutions and then pick the best solution. So, it is time consuming as compared to Greedy method.
For ex. Krushkal or Prim's method to find MST (minimum spanning tree) Dijakstra shortes path algorithm.	For any problem, there are multiple solution so, we explore all and then pick the optimum.
	Dynamic programming is solved using recursive formula but mostly the problem is solved using iteration. Dynamic programming follows the principle of optimality. Means problem is solved by taking sequence of decisions.
We take the decision once and we follow that procedure only.	In every stage, we take the decision.
it not give always optimal solution (it give feasibale solution)	it always give optimal solution
not as reliable as dynamic	very reliable
fast result	comparativly slow result

7.2 Fibonacci Series

Algorithm Fibo(n) T(n)

{

if(n<=1)

return n;

else

return(Fibo(n-2) + Fibo(n-1)); T(n-2)+ T(n-1)

}

Based on the algorithm, we can write the following recursive function:

$$\text{Fibo(n)} = \begin{cases} 0 & if\ \ n == 0 \\ 1 & if\ \ n == 1 \\ Fibo(n-2) + Fibo(n-1) & if\ n > 1 \end{cases}$$

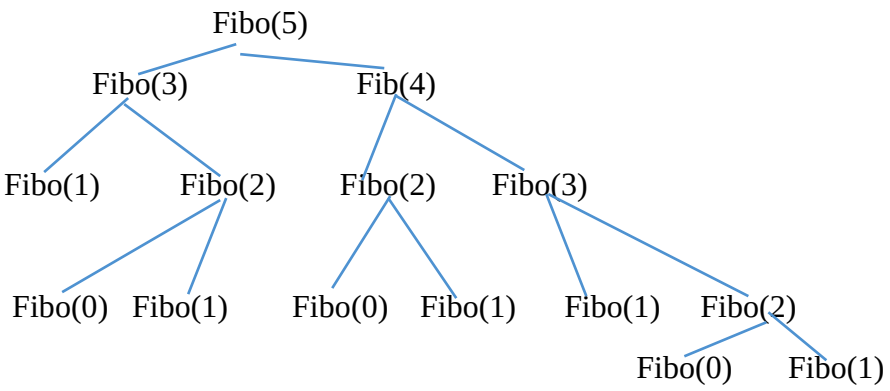
The recurrence relation is:

$$T(n) = 2\,T(n-1) + 1$$

Time complexity, using master theorem of decreasing function = $O(2^n)$

Is there any way to reduce time?

0, 1, 1, 2, 3, 5, 8, 13, 21 ….



Top-Down approach or Memorization technique*:*

We can observe that Fibo(0) and Fibo(1) is called many times…why calling and computing of same function again and again?

We can store the result of function call and use array to store it.

Traverse the above tree in post order and store the result.

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

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

When we have to call Fibo(2) then already Fibo(1) is found which we can use directly and Fibo(0) is 0 so, Fibo(2) is Fibo(0) + Fibo(1) = 0 + 1 = 1

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

In this way, result of calls is stored and used later. No unnecessary calling of the same functions is done.

Total six calls of the functions for finding Fib(5).

Fibo(n) = n + 1 calls

Fibo(n) = O(n) which is polynomial and less than $O(2^n)$.

Bottom-Up approach or Tabulation method or iteration method:

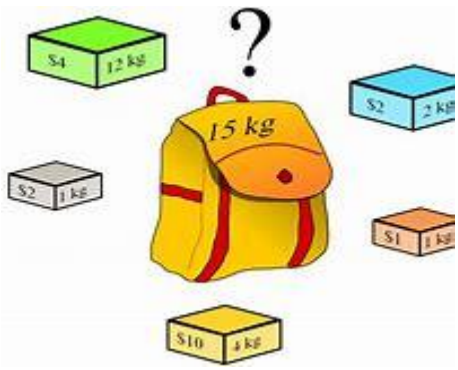
How to write the algorithm using iterative process?

```
int Fibo(n)
{
if(n <=1)
  return n;
F[0]=0,F[1]=1;
for(i=2;i<=n;i++)
{
  F[i]=F[i-2]+F[i-1];
}
return F[n];
}
```

7.3 0/1 Knapsack Problem:

The knapsack problem is an optimization problem used to illustrate both problem and solution. It derives its name from a scenario where one is constrained in the number of items that can be placed inside a fixed-size knapsack. Given a set of items with specific weights and values, the aim is to get as much value into the knapsack as possible given the weight constraint of the knapsack by maximizing the value/profit.

In the following figure, we are given bag of 15 kg and we have to put items in such a way that it maximizes the profit/value and weight should not be exceeded than 15 kg.



Example:

P = { 1, 2, 5, 6 }

W = { 2, 3, 4, 5 }

M = 8

n = 4

We want a solution like this whether item is included or not?

Like X = { 0, 1, 1, 0} – 0 means not included and 1 means included

We can try all possible solutions so, how many possible solutions?

0 1 1 0
1 1 0 0

0 0 1 1

….

Total = 2^4 = 16 for four items then how many for n items = 2^n

We use tabulation method to solve this problem. The column shows the bag capacity from 0 to 8 and row shows the objects.

Step 1:

When the first object is included. Its weight is 2 and profit is 1 so, put 1 in the column of 2. Also, for the remaining columns from 3rd columns onwards put profit 1.

P	W		0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0								
5	4	3	0								
6	5	4	0								

Step 2:

When the second object is considered with weight is 3, it can be kept in column 3 and put profit 2 in that column. For a capacity of 5, we need to consider the object 1 and 2 both so, it's profit becomes 3.

P	W		0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0								
6	5	4	0								

Step 3:

When the third object is selected with weight 4 then profit is 5 so, put 5 in column 4. In capacity 6, we can include object 1 and 3 and its combined profit is 6. In capacity 7 and 8, we can include object 2 and 3, its combined profit is 7.

P	W		0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0								

Step 4:

When the fourth object is selected with weight 5 then put its profit 6 in column 5. In capacity 7, we can include object 1 and 4 so, profit is 7. In capacity 8, we can include object 2 and 4 so, profit is 8.

P	W		0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0	**0**	**1**	**2**	**5**	**6**	**6**	**7**	**8**

Where i – first index is object and w – second index is weight in the following vector V.

V[i,w] = max{ V[i-1,w] , V[i-1,w-w[i]] + P[i]}

V[3,4] = max { V[2,4], V[2,4-3]+ 5} = max{2,0+5}=max(2,5}=5

V[4,1] = max{V[3,1], V[3,1-5] + 6} = max{0,V[3,-4]} = max{0,undefined} = 0

Upto 5th weight, we get the same value so, take the value from the previous row.

V[4,4] = max { V[3,4], V[3,4-4]+ 5} = max{5,0+5}=max(5,5}=5

V[4,5] = max { V[3,5], V[3,5-5]+ 6} = max{5,0+6}=max(5,6}=6

V[4,8] = max { V[3,8], V[3,8-5]+ 6} = max{7,2+6}=max(7,8}=8

We can also fill the table without using the above formula.

Apply the combination of weight and stick to the constraint of weights to fill the values in the table.

Now, how to get the objects which are included in the solution?

X = { x_1 , x_2, x_3, x_4 }

Total profit we got is 8 and it is generated in 4th row so, x_4 = 1,

Now, total profit – profit of x_4 = 8 – 6 = 2

Find 2 in the row but it is also there in previous row. Try to find that from where it has come. It is included from second row so, x_2 = 1 not from x_3 .

Now, remaining profit 2 – profit of x_2 = 2 – 2=0

So, final solution is X = { x_1=0 , x_2=1, x_3=0, x_4 = 1}

Algorithm:

int knapsack(int W, int wt[], int val[], int n){

```
  int K[n+1][W+1];
  for(int i = 0; i<=n; i++) {
    for(int w = 0; w<=W; w++) {
      if(i == 0 || w == 0) {
        K[i][w] = 0;
      } else if(wt[i-1] <= w) {
        K[i][w] = findMax(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];
}
```

Time complexity:

This algorithm takes O(n * w) times as table c has (n+1).(w+1) entries, where each entry requires O(1) time to compute.

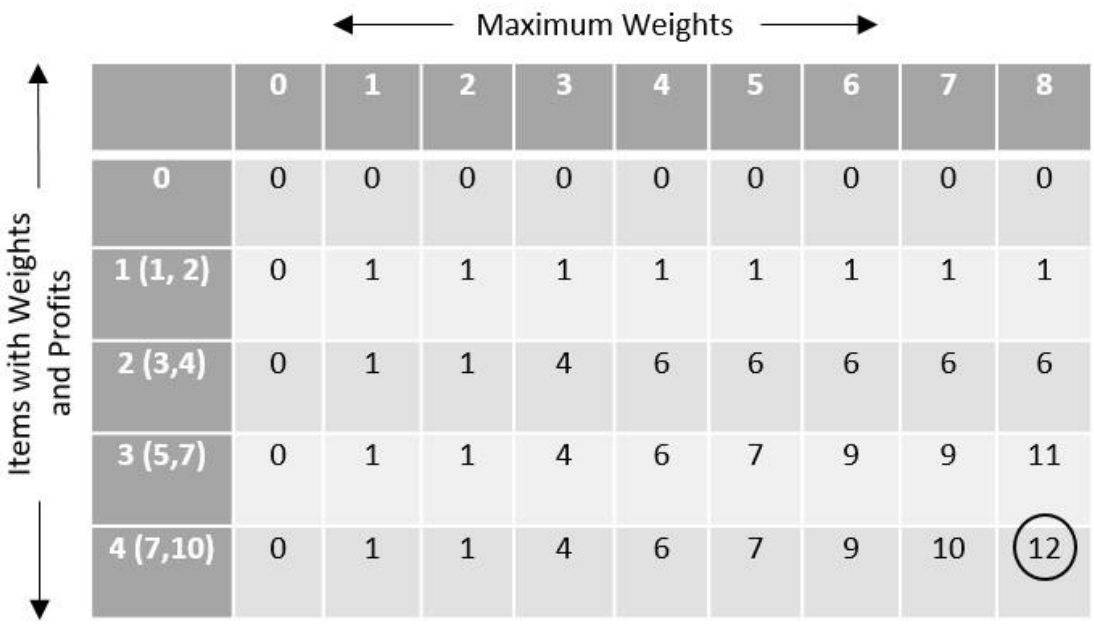
Example:

Solve the following knapsack problem for the capacity of 8.

Item	**A**	**B**	**C**	**D**
Profit	2	4	7	10
Weight	1	3	5	7

Solution:

The optimal solution is {1, 4} with the maximum profit is 12.



	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

7.4 Binomial Coefficient

In combinatorics, the binomial coefficient is used to denote the number of possible ways to choose a subset of objects of a given numerosity from a larger set.

$$\binom{n}{k} = \frac{n!}{(n-k)!k!}$$

The value of C(n, k) can be recursively calculated using the following standard formula for Binomial Coefficients.

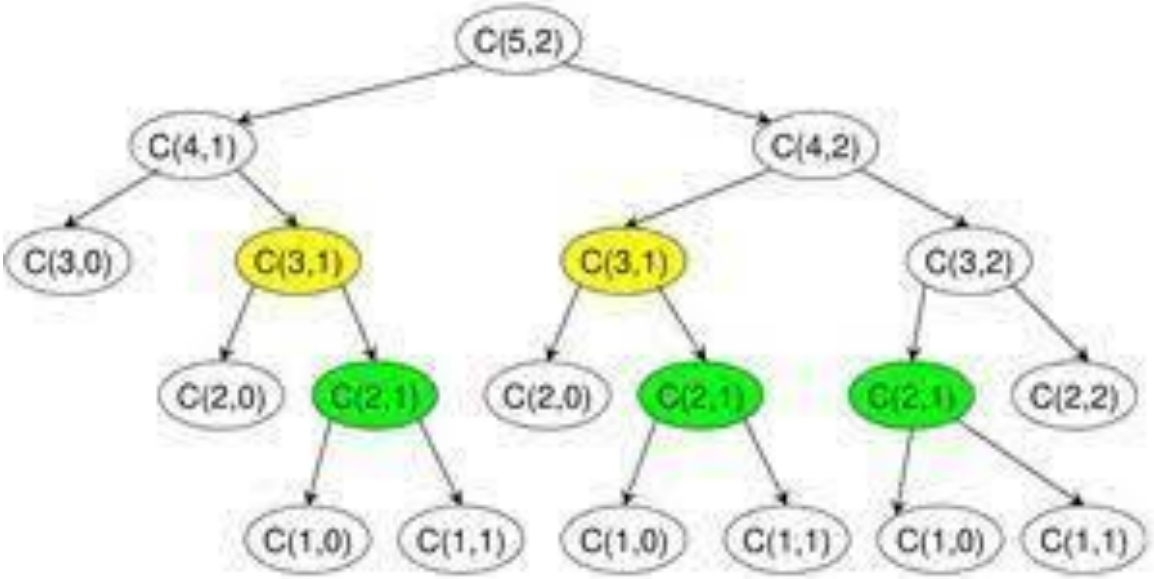
C(n, k) = C(n-1, k-1) + C(n-1, k)

C(n, 0) = C(n, n) = 1

Recursion:

```
int BC(int n, int k)
{
    if(k==0 || k==n)
        return 1;
    return BC(n-1,k-1) + BC(n-1,k);
}
```

Suppose we have to find C(5,2) then following tree will be constructed. We can see that C(3,1) and C(2,1) are calculated at more than one places.



We can solve it using tabulation method as follows:

K N	0	1	2	3	4	5
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
4	1	4	6	4	1	
5	1	5	**10**	10	5	1

Program using iterative approach:

```
// A Dynamic Programming based solution
// that uses table C[][] to
// calculate the Binomial Coefficient
#include <stdio.h>

// Prototype of a utility function that
// returns minimum of two integers
int min(int a, int b);

// Returns value of Binomial Coefficient C(n, k)
int binomialCoeff(int n, int k)
{
	int C[n + 1][k + 1];
	int i, j;

	// Calculate value of Binomial Coefficient
	// in bottom up manner
	for (i = 0; i <= n; i++) {
		for (j = 0; j <= min(i, k); j++) {
			// Base Cases
			if (j == 0 || j == i)
				C[i][j] = 1;

			// Calculate value using
			// previously stored values
			else
				C[i][j] = C[i - 1][j - 1] + C[i - 1][j];
		}
	}

	return C[n][k];
}

// A utility function to return
```

```
// minimum of two integers
int min(int a, int b) { return (a < b) ? a : b; }

/* Driver program to test above function*/
int main()
{
        int n = 5, k = 2;
        printf("Value of C(%d, %d) is %d ", n, k,
                binomialCoeff(n, k));
        return 0;
}
```

7.5 Matrix Multiplication

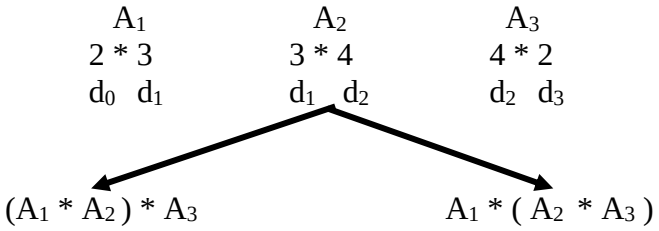
Matrix chain multiplication:

For the following two matrices of 2 * 3 and 3 * 2, there is one condition that number of columns of the first matrix and row of the second matrix must be the same.

A				B			C	
3	4	5		1	3		3 *1 + 4 * 2 + 5 * 4	3 * 3 + 4 * 2 + 5 * 1
2	3	1	*	2	2	=	2 * 1 + 3 * 2 + 1 * 4	2 * 3 + 3 * 2 + 1 * 1
				4	1			
2 * 3				3 * 2		=	2 * 2	

Total number of multiplications required are = 2 * 3 * 2 = 12
The dimension of resultant matrix is number of rows of the first matrix and column of the second matrix that is 2 * 2 in the above example.

Suppose we have following three matrices.



Is the result same if we do the above multiplications in two different ways?

Yes. Matrix multiplication has associativity property. But, what about number of multiplications required? Is the same or different?

(A1 * A2) * A3

2 * 3 3 * 4 4 * 2

2 * 3 * 4 = 24

2 * 4 0

4 * 2

2 * 4 * 2 = 16

So, total number of multiplications are 24 + 16 = 40

A1 * (A2 * A3)

2 * 3 3 * 4 4 * 2

0 3 * 4 * 2 = 24

2 * 3 3 * 2

2 * 3 * 2 = 12

So, total number of multiplications are 24 + 12 = 36

If we have a greater number of matrices then it becomes difficult to find all possible combinations in order to reduce the number of multiplications so, we need to devise formula for the same.

Let's try to formulate the general formula based on the above example.

Take the above problem of finding C [1,3] means multiplication of three matrices.

$(A_1 * A_2) * A_3$	$A_1 * (A_2 A_3)$
C[1,2] C[3,3]	C[1,1] C[2,3]
2 * 3 3 * 4 4 * 2	2 * 3 3 * 4 4 * 2
2 * 3 * 4 = 24 0	0 3 * 4 * 2 = 24
2 * 4 4 * 2	2 * 3 3 * 2
C[1,2] + C[3,3] + 2 * 4 * 2	C[1,1] + C[2,3] + 2 * 3 * 2
C[i, k] + C[k+1,j] + $d_0 * d_2 * d_3$	C[i, k] + C[k+1,j] + $d_0 * d_1 * d_3$
d_{i-1} d_k d_j	d_{i-1} d_k d_j

$$\boldsymbol{C[i,j] = \min_{i \le k < j} \{ C[i,k] + C[k+1,j] + d_{i-1} * d_k * d_j \}}$$

How many total possible combinations of multiplications of n matrices?

$A_1 * A_2 * A_3 * A_4$

Total number of combinations of matrices' multiplications are = $\frac{2\,n_{C_n}}{n+1} = \frac{2\,(n-1)_{C_{n-1}}}{n}$

For example: n = 4 so, $\frac{2\,(4-1)_{C_{4-1}}}{4} = \frac{2*3_{C_3}}{4} = \frac{6_{C_3}}{4}$ = 5 possible ways to multiply them

Lets apply equation of C[i,j] to find the optimal number of multiplications for four matrices.

$$C[1,4] = \min_{1 \le k < 4} \begin{cases} k = 1 & C[1,1] + C[2,4] + d_0 * d_1 * d_4, \\ k = 2 & C[1,2] + C[3,4] + d_0 * d_2 * d_4, \\ k = 3 & C[1,3] + C[4,4] + d_0 * d_3 * d_4 \end{cases}$$

In the above formula, C [1,1] and C[4,4] are 0 but we can find C[2,4] as follows.

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

In this way, it becomes difficult to find the optimal answer.

We can start with lower values like C[1,1], C[1,2] etc.. and then we can come to the answer for finding for higher value.

Let's take one example as follows:

A_1 * A_2 * A_3 * A_4
3 * 2 2 * 4 4 * 2 2 * 5
d_0 --------d_1-------- ----------d_2 --------- ------d_3------ d_4

C[] – Cost array

	1	2	3	4
1	0	24	48	58
2		0	16	36
3			0	40
4				0

Table 7.1

K – Table – This is used as shown in Table 7.2 to know which matrices are multiplied in which order means where to put the parenthesis.

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

Table 7.2

The calculation is done only in upper triangular values and the cost table (Table 7.1) is filled diagonally.

First, we find the cost for difference 1 (difference between i and j) entries as shown in the table (color), means 1,2; 2,3 etc..

$$C[1,2] = \min_{1 \le k < 2} \begin{cases} k = 1 \;\; C[1,1] + C[2,2] + d_0 * d_1 * d_2 \\ 0 + 0 + 3 * 2 * 4 = 24 \end{cases}$$

$$C[2,3] = \min_{2 \le k < 3} \begin{cases} k = 2 \;\; C[2,2] + C[3,3] + d_1 * d_2 * d_3 \\ 0 + 0 + 2 * 4 * 2 = 16 \end{cases}$$

$$C[3,4] = \min_{3 \le k < 4} \begin{cases} k = 3 \;\; C[3,3] + C[4,4] + d_2 * d_3 * d_4 \\ 0 + 0 + 4 * 2 * 5 = 40 \end{cases}$$

Now, we find the cost for the entries where i and j difference is 2 in the table as shown (color) means 1,3;2,4 etc.

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

$$= \min_{1\le k<3} \begin{matrix} \{ \\ k=1\ \ 0+16+3*2*2\ -28, \\ k=2\ \ 24+\ \ 0+3*4*2\ -48\ \} \end{matrix}$$

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

$$= \min_{2\le k<4} \begin{matrix} \{ \\ k=2\ \ 0+40+2*4*5-\ \ 80\ , \\ k=3\ \ \ 16+0+2*2*5-\ \ 36\ \} \end{matrix}$$

Finally, we calculate the cost of the last diagonal element as shown in the table (color) and it is the result of total number of multiplications.

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

$$= \min_{1\le k<4} \begin{matrix} \{\ \ k=1\ \ 0+36+3*2\ *5-66\ , \\ k=2\ \ 24\ +40+3*4*5-124\ , \\ k=3\ \ 28\ +0+3*2*5-\ \ 58\ \} \end{matrix}$$

Now, K matrix is used to know the order of matrix multiplications means where we have to put the parenthesis.

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

Here, entry K[1,4] shows 3 it means put the bracket after 3rd matrix.

(A_1 * A_2 * A_3) * A_4

The second entry K[1,3] is 1 so, it conveys that put bracket after A_1 .

((A_1) * A_2 * A_3) * A_4

So, the final order of multiplications is as above.

Time complexity of finding the place for the parenthesis:

The total number of entries in the table are:
1 + 2 + 3 + 4 … + = n (n + 1) /2 = O (n^2)

Now, we find the above values k times for each entry and k is from i to j, and if it can be assumed as n.

So, time complexity is $O(n^3)$.

Example:

Find the total number of multiplications and parenthesization for the following matrix chain multiplication.

$A_{5\times10}\times B_{10\times15}\times C_{15\times20}\times D_{20\times25}$

Solution:

Cost matrix:

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

Note: Instead of 2200 it 2250 and instead of 4700 it is 4750.

K matrix:

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

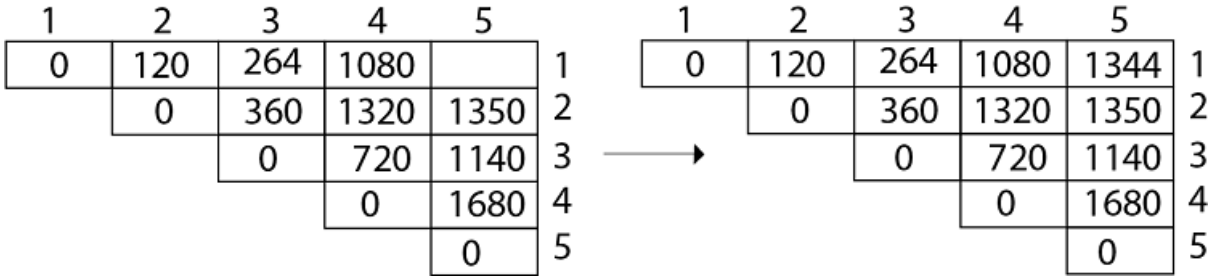
Parenthesization will be done as follows:

((AB)(C)) (D)

Example:

The matrices are $A_{4 \times 10}$, $A_{10 \times 3}$, $A_{3 \times 12}$, $A_{12 \times 20}$, $A_{20 \times 7}$.
Find the number of multiplications and parenthesization of it.

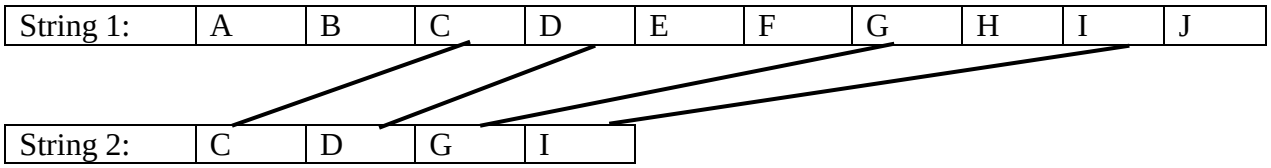
Solution:



7.6 Longest Common Subsequence

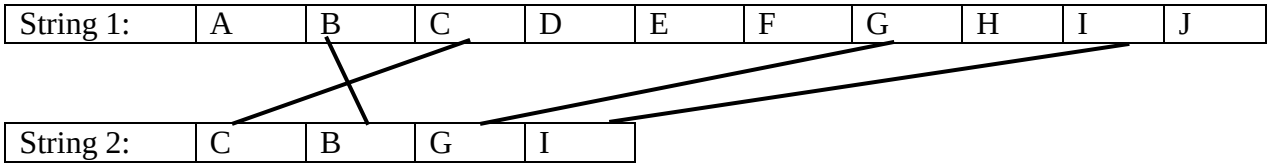
Given two strings, **S1** and **S2**, the task is to find the length of the Longest Common Subsequence, i.e. longest subsequence present in both of the strings. The matching of the characters need not to be continuous.

Take the following example:



In the above example, we can say that C D G I is the longest common subsequence between them. Also, D G I, D G, C D are common subsequence but they are not longest.

The following example, C B G I is not longest common subsequence because B is not following to C in another string. But B G I and G I are common subsequences, and B G I is longest common subsequence.



Take another example:

String 1:	A	B	D	A	C	E

String 1:	B	A	B	C	E

Here, B A C E and A B C E are the longest common subsequences between them. So, it is possible to have more than one LCS.

Applications of LCS:
The problem of computing longest common subsequences is a classic computer science problem, the basis of data comparison programs such as the diff utility and has applications in computational linguistics and bioinformatics (where strings represent DNA or protein sequences).

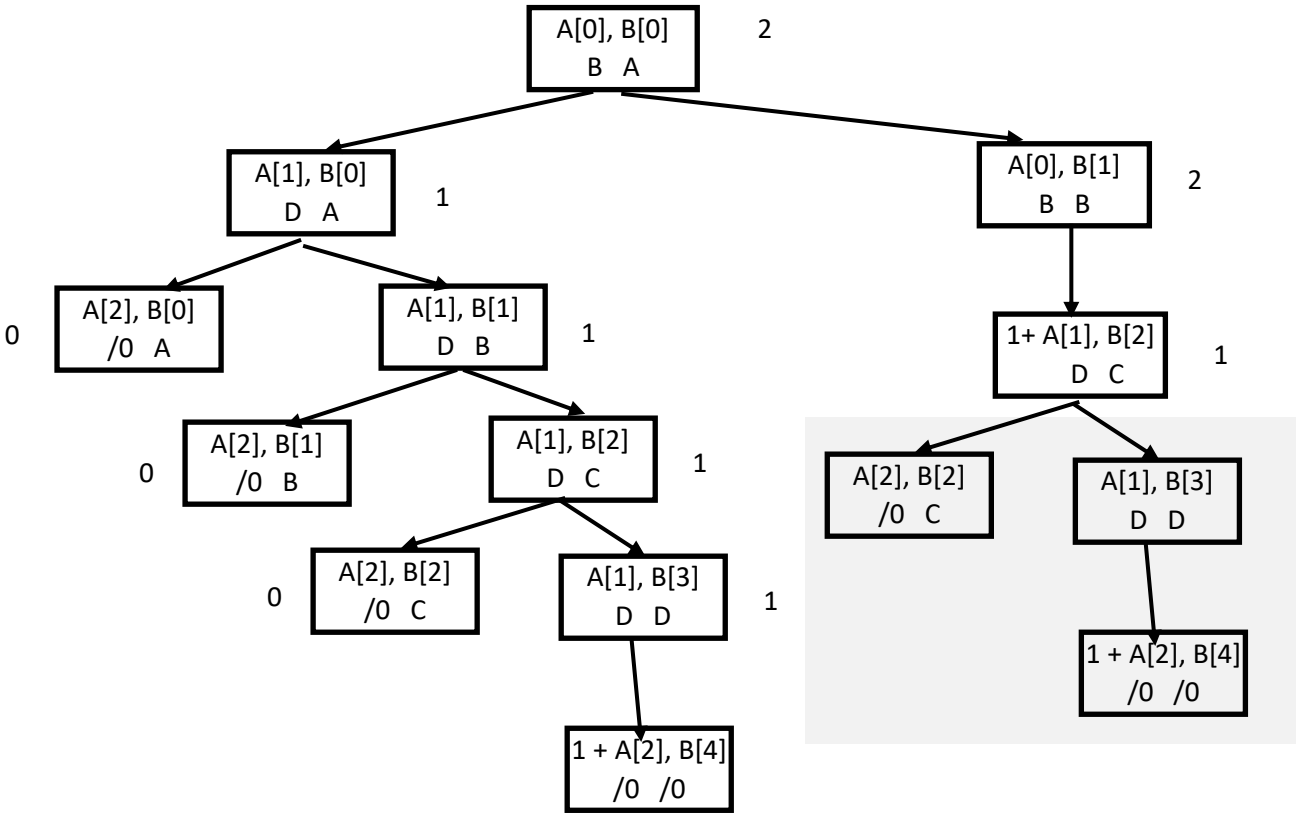
Recursive algorithm for finding LCS:

```
int lcs(string A, string B, int i, int j)
{
   if (A[i] !='\0'  || B[j] != '\0')
      return 0;
   else if (A[i] == B[j])
      return 1 + lcs(A, B, i + 1, j + 1);
   else
      return max(lcs(A, B, i+1, j),lcs(A, B, i, j + 1));
}
```

Let's trace the above algorithm by taking simple example.

String A:	B	D

String B:	A	B	C	D



The dark portion shows the overlapping of the call for same input so, we can apply memorization approach.

Memorization approach:

The table starts filling from the bottom.

	A	B	C	D	\0
B	2	2			
D	1	1	1	1	
\0	0	0	0	0	0

Time complexity:

Size of string 1 is m and size of string 2 is n.

Then it is O (m * n) for filling the table entries.

Implementation of top down (memorization) algorithm:

```
// A Top-Down DP implementation
// of LCS problem
#include <bits/stdc++.h>
using namespace std;

// Returns length of LCS for X[0..m-1],
// Y[0..n-1]
int lcs(char* X, char* Y, int m, int n,
                vector<vector<int> >& dp)
{
        if (m == 0 || n == 0)
                return 0;
        if (X[m - 1] == Y[n - 1])
                return dp[m][n] = 1 + lcs(X, Y, m - 1, n - 1, dp);

        if (dp[m][n] != -1) {
                return dp[m][n];
        }
        return dp[m][n] = max(lcs(X, Y, m, n - 1, dp),
                                                lcs(X, Y, m - 1, n, dp));
}

// Driver code
int main()
{
        char X[] = "AGGTAB";
        char Y[] = "GXTXAYB";

        int m = strlen(X);
        int n = strlen(Y);
```

```
        vector<vector<int> > dp(m + 1, vector<int>(n + 1, -1));
        cout << "Length of LCS is " << lcs(X, Y, m, n, dp);

        return 0;
    }
```

Tabulation approach:

```
int lcs(string X, string Y, int m, int n)
{

   // Initializing a matrix of size
   // (m+1)*(n+1)
   int L[m + 1][n + 1];

   // Following steps build L[m+1][n+1]
   // in bottom up fashion. Note that
   // L[i][j] contains length of LCS of
   // X[0..i-1] and Y[0..j-1]
   for (int i = 0; i <= m; i++) {
      for (int j = 0; j <= n; j++) {
         if (i == 0 || j == 0)
            L[i][j] = 0;

         else if (X[i] == Y[j])
            L[i][j] = L[i - 1][j - 1] + 1;

         else
            L[i][j] = max(L[i - 1][j], L[i][j - 1]);
      }
   }

   // L[m][n] contains length of LCS
   // for X[0..n-1] and Y[0..m-1]
   return L[m][n];
}
```

		0	1	2	3	4
			A	B	C	D
	0	0	0	0	0	0
B	1	0				
D	2	0				

The approach is bottom up but the table filling starts from top to bottom.

Now, take the table entries one by one. L[1,1] and compare the characters so, they are B and A and they don't match so, take the maximum of L[1,0] and L[0,1] which is 0 and shown as follows.

		0	1	2	3	4
			A	B	C	D
	0	0	0	0	0	0
B	1	0	0			
D	2	0				

Now, take another entry L[1,2] so, characters B and B, they match so, L[0,1] + 1 so, 0 + 1 is 1.

		0	1	2	3	4
			A	B	C	D
	0	0	0	0	0	0
B	1	0	0	1		
D	2	0				

Take L[1,3] and characters are B and C so, they don't match. Max of (L[0,3], L[1,2])= max (0,1) = 1.

		0	1	2	3	4
			A	B	C	D
	0	0	0	0	0	0
B	1	0	0	1	1	1
D	2	0				

Now L[2,1] and characters are same so, 1 + L[1,0] = 1 + 0 = 1.

		0	1	2	3	4
			A	B	C	D
	0	0	0	0	0	0
B	1	0	0	1	1	1
D	2	0	0	1	1	

Now, find L[2,4] and characters D and D match so, L[2,4] = 1 + L[1,3] = 1 + 1 = 2

		0	1	2	3	4
			A	B	C	D
	0	0	0	0	0	0
B	1	0	0	1	1	1
D	2	0	0	1	1	2

As above we got the length of the common subsequence that is 2 but which characters are in the subsequence?

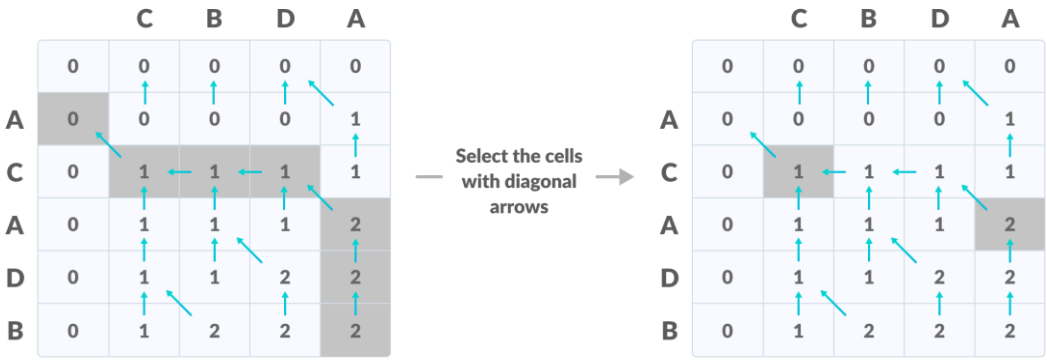
It is found as follows:

- The entry L[2,4] = 2 has come from L[1,3] so, put slant arrow as shown in the following table.
- L[1,3] has come from L[1,2] so, put left arrow as shown in the following table.
- L[1,2] has come from L[0,1] so, put slant arrow.
- If the L[i,j] value has come through the maximum of L[i-1,j] or L[i, j-1] then put horizontal left arrow or vertical upward arrow (shown using black arrow) and if the L[i, j] value has come through 1 + L[i-1,j-1] then put slant arrow (shown using yellow arrow).
- Consider only slanted arrows (yellow) and only those columns from where these arrows have started. In the following example, arrow is drawn from 2 to 1 and we consider one of the characters as D. Similarly, another slanted arrow is from 1 to 0 and we consider the character B.
- The longest common subsequence is B D.

		0	1	2	3	4
			A	B	C	D
	0	0	0	0	0	0
B	1	0	0	1	1	1
D	2	0	0	1	1	2

Example 1:

String 1: A C A D B
String 2: C B D A



Example 2:

String 1: S T O N E
String 2: L O N G E S T

			L	O	N	G	E	S	T
		0	1	2	3	4	5	6	7
	0	0	0	0	0	0	0	0	0
S	1	0	0	0	0	0	0	1	1
T	2	0	0	0	0	0	0	1	2
O	3	0	0	1	1	1	1	1	2
N	4	0	0	1	2	2	2	2	2
E	5	0	0	1	2	2	3	3	3

The longest common subsequence is O N E with length 3.

7.7 Making change problem

We are given the coins of different denominations and a value X. We need to find the minimum number of coins required to make value X. We have infinite supply of each coin.

Now, see the following making change problem using greedy approach:

Denom[] = { 1, 5, 6, 9 }

X = 11

Solution using greedy approach gives the answer = { 9, 1, 1 } which is not optimal.

Here, it shows that solution does not exist even if solution is found.

Denom[] ={ 4, 10, 25 }

X = 41

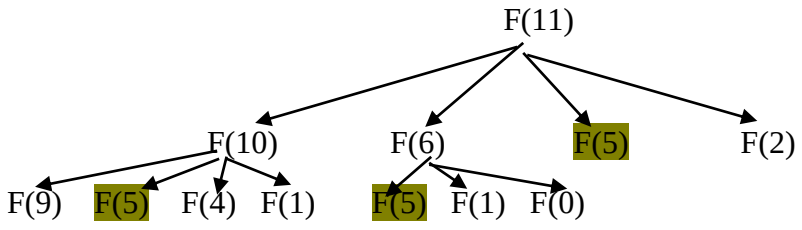
Solution using greedy does not give the solution – 41 – 25 = 16 – 10 = 6 – 4 = 2.

Hence, we can use dynamic programming where it gives solution to any problem of making change.

Top-down approach:

Denom[] = {1, 5, 6, 9}

X = 11



Here, in the above recursive tree we can see the overlapping of the sub problem.

Tabulation approach:

Denom[] = {1, 5, 6, 9}

X = 10

Prepare the table as follows: rows for denominations and column for the sum from 1 to X in increment of 1.

Optimal substructure:

$$C[i, j] = \begin{cases} 1 + C[1, j - d_1], & \text{, if } i = j \\ C[i - 1, j], & \text{if } j < d_i \\ \min(C[i - 1, j], 1 + C[i, j - d_i]), & \text{otherwise} \end{cases}$$

Apply the below formula to fill the table entries.

If denom[i] > M[j]

Copy the value from M[i-1][j]

Else

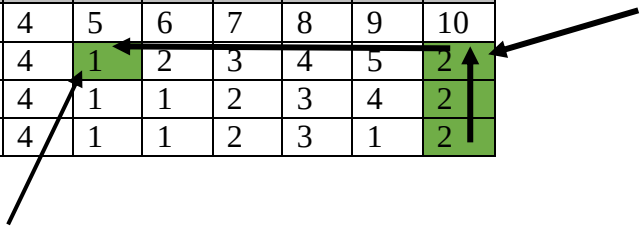
Min(M[i-1][j], 1 + M[i][j-denom[i]])

j i	0	1	2	3	4	5	6	7	8	9	10
1	0	1	2	3	4	5	6	7	8	9	10
5	0	1	2	3	4	1	2	3	4	5	2
6	0	1	2	3	4	1	1	2	3	4	2
9	0	1	2	3	4	1	1	2	3	1	2

From the above table, we come to know that we need minimum 2 coins to make value X=10.

Now, find the coins of denominations to make value X = 10.

j i	0	1	2	3	4	5	6	7	8	9	10
1	0	1	2	3	4	5	6	7	8	9	10
5	0	1	2	3	4	1	2	3	4	5	2
6	0	1	2	3	4	1	1	2	3	4	2
9	0	1	2	3	4	1	1	2	3	1	2



Value has changed here so,
sum – denom[i]
10 – 5 = 5 so, move to column 5 in same row

Value has changed here so,
sum – denom[i]
5 – 5 = 0 so, move to column 0 in same row

The last bottom-right entry shows the minimum number of coins which are 2 to make value 10.

- Move in the same column (column 10) to see that from where the value 2 has changed so, it is row 2 means denomination 5. So, add 5 in the solution S = {5}
- Now, subtract column number from the denomination of that row means 10 – 5 = 5 and move in the same row with column 5 and again check in that column that from where that value is changed. The value is 1 and it has changed from that row only so, another coin of denomination in the solution is 5. So, add 5 in the solution = {5, 5}
- Now, subtract 5 – 5 = 0. We got the solution.

Example:
Demom[] = { 1, 4, 6 }
X = 8

Solution:
Minimum number of coins are 2 and they are { 4, 4 }

		j								
		0	1	2	3	4	5	6	7	8
i=0		0	0	0	0	0	0	0	0	0
i=1	d_1=1	0	1	2	3	4	5	6	7	8
i=2	d_2=4	0	1	2	3	1	2	3	4	2
i=3	d_3=6	0	1	2	3	1	2	1	2	**2**

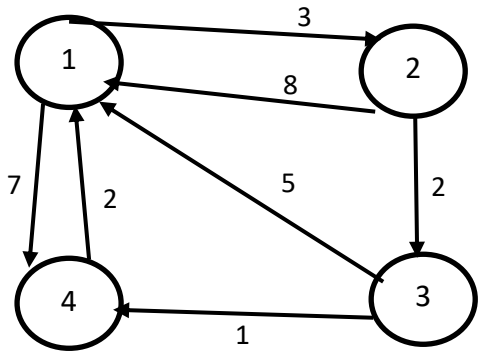
7.8 Floyd Warshall algorithm (all pair shortest path)

It is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. This algorithm follows the dynamic programming approach to find the shortest path.

This algorithm works for both the directed and undirected weighted graphs. It also works on the graph with negative edges. But it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative).

Can Dijkstra algorithm work for finding all pair shortest path?
Yes. We can. It takes time $O(n^2)$ from one source to all destination so, for all sources (n vertices) it takes $O(n^3)$.

Let's consider the following weighted graph:



A^0 =

	1	2	3	4
1	0	3	∞	7
2	8	0	2	∞
3	5	∞	0	1
4	2	∞	∞	0

In this algorithm, we find the path via all vertices one by one to check whether it has shortest path through it or not.

Step 1: First we calculate the shortest path via vertex 1 and update the path if there is any.

Keep all path through vertex 1 as it is that means row of 1 and column of 1 will remain as it is. Also, diagonal entries are path to itself will remain 0.

A^1 =

	1	2	3	4
1	0	3	∞	7
2	8	0	**2**	**15**
3	5	**8**	0	1
4	2	5	∞	0

Now find A^1 [2,3] first,

Min(A^0 [2,3], A^0 [2,1] + A^0 [1,3]) = Min(2, 8 + ∞) = 2

A^1[2,4] = Min(A^0 [2,4], A^0 [2,1] + A^0 [1,4]) = Min(∞, 8 + 7) = 15

A^1[3,2] = Min(A^0 [3,2], A^0 [3,1] + A^0 [1,2]) = Min(∞, 5 + 3) = 8

Step 2: Now, we take 2 as intermediate vertex and calculate the values in the matrix.

A^2 =

	1	2	3	4
1	0	3	5	7
2	8	0	2	15
3	5	8	0	1
4	2	5	7	0

A^2[1,3] = Min(A^1 [1,3], A^1 [1,2] + A^1 [2,3]) = Min(∞, 3 + 2) = 5

Step 3: Now. we take 3 as intermediate vertex and calculate the values in the matrix.

A^3 =

	1	2	3	4
1	0	3	5	6
2	7	0	2	3
3	5	8	0	1
4	2	5	7	0

A^3[1,2] = Min(A^2 [1,2], A^2 [1,3] + A^2 [3,2]) = Min(3, 5 + 8) = 3

Step 4: Now, we take 4 as intermediate vertex and calculate the values in the matrix.

A^4 =

	1	2	3	4
1	0	3	5	6
2	5	0	2	3
3	3	6	0	1
4	2	5	7	0

The entries in the table are updated using the following equation.

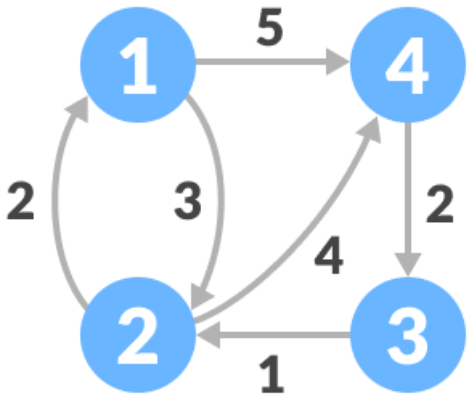
$A^k[i,j] = \min\{ A^{k-1}[i,j],\ A^{k-1}[i,k] + A^{k-1}[k,j]\}$

Code snippet for filling the table:

```
for(k=1;k<=n;k++)
  {
      for(i=1;i<=n;i++)
          {
              for(j=1;j<=n;j++)
                  A^k[i,j] = min{ A^{k-1}[i,j],  A^{k-1}[i,k] +  A^{k-1}[k,j]}
           }
  }
```

Time complexity is $O(n^3)$.

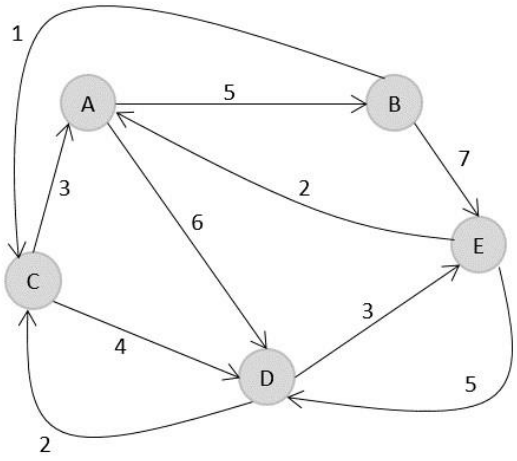
Example 1:



Solution:

	1	2	3	4
1	0	3	7	5
2	2	0	6	4
3	3	1	0	5
4	5	3	2	0

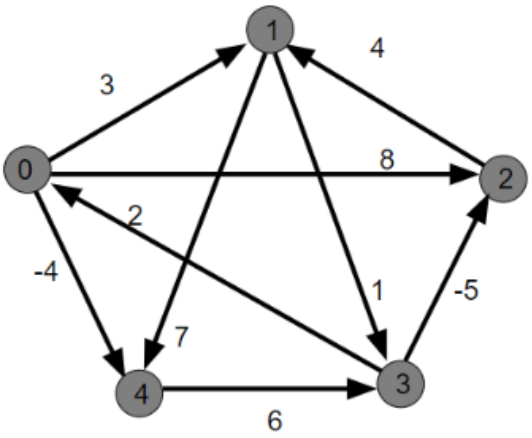
Example 2:



Solution:

$$A_5 = \begin{matrix} 0 & 5 & 6 & 6 & 9 \\ 4 & 0 & 1 & 5 & 7 \\ 3 & 8 & 0 & 4 & 7 \\ 5 & 10 & 2 & 0 & 3 \\ 2 & 7 & 7 & 5 & 0 \end{matrix}$$

Example 3:



Solution:

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

7.9 Bellman Ford algorithm (single source shortest path)

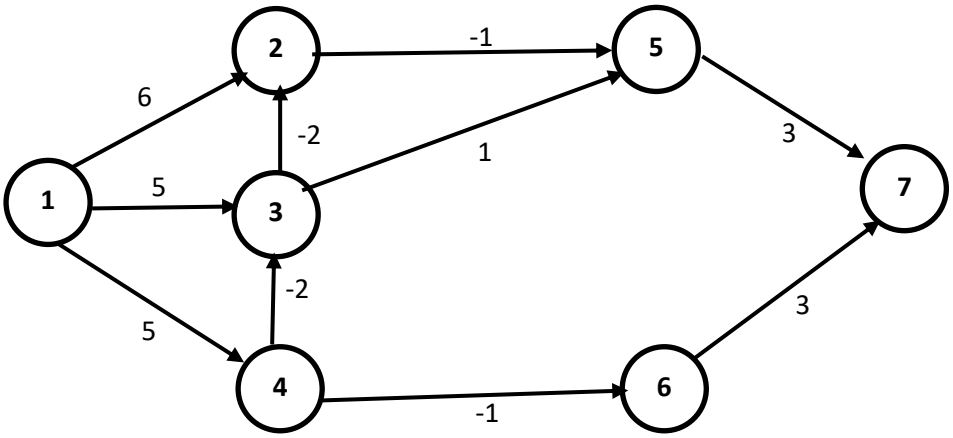
This is similar to Dijkstra's algorithm, but this can work with graphs in which edges can have negative weights. Dijkstra's algorithm may or may not give the correct result.

Why would one ever have edges with negative weights in real life?

Negative weight edges might seem useless at first, but they can explain a lot of phenomena like cashflow, the heat released/absorbed in a chemical reaction, etc.

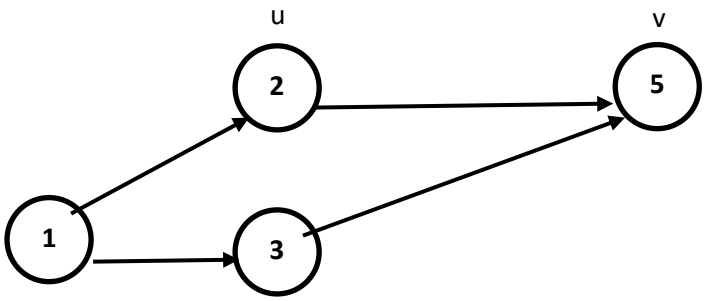
Why do we need to be careful with negative weights?

Negative weight edges can create negative weight cycles i.e. a cycle that will reduce the total path distance by coming back to the same point.



We need to do relaxation process for each edge and number of times equal to one less than the number of vertices.

What is the relaxation process? Let's take the following example.



In the above graph, two vertices u and v are such that we have shortest path to u and v. If we relax/update the vertex v then it is done as follows. d[u] – distance to u from some source, d[v] – distance to u from some source, C(u,v) – cost between u and v.

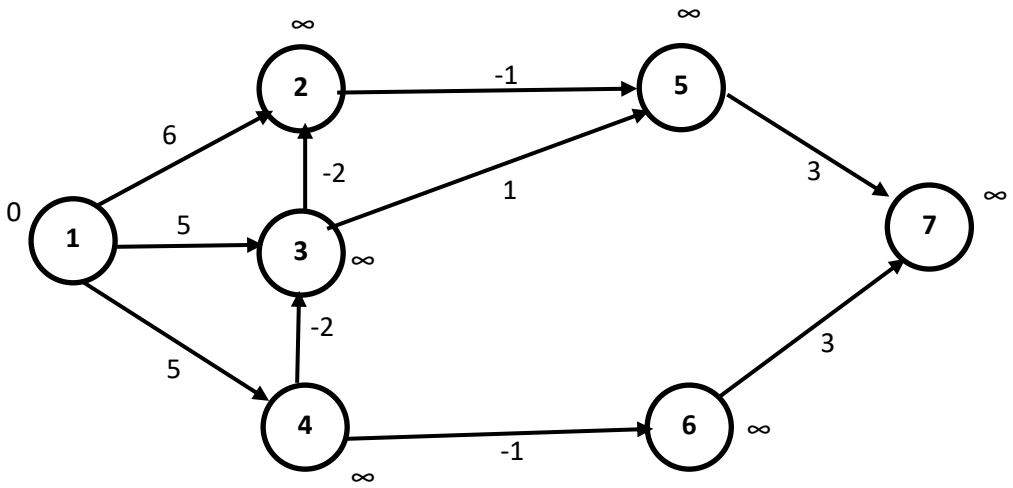
if(d[u] + C(u,v) < d[v])

then d[v] = d[u] + C(u,v)

List out all the edges:

(1,2), (1,3), (1,4), (3,2), (3,2), (2,5), (3,5), (4,6), (5,7), (6,7)

Select the source vertex as 1 and set its cost to zero and the remaining vertices distance to infinity from source vertex as shown in the following graph.



Now, relax all the edges n – 1 number of times. It is possible that after few steps (means before reaching n – 1 steps), there is not a single update in the distance which shows that no need to perform further steps.

Step 1:

Relax all the edges one by one.

Take edge (1,2):

d[2] = d[1] + C(1,2) = 0 + 6 < ∞ so, it is updated as 6.

Take edge (1,3):

d[3] = d[1] + C(1,3) = 0 + 5 < ∞ so, it is updated as 5.

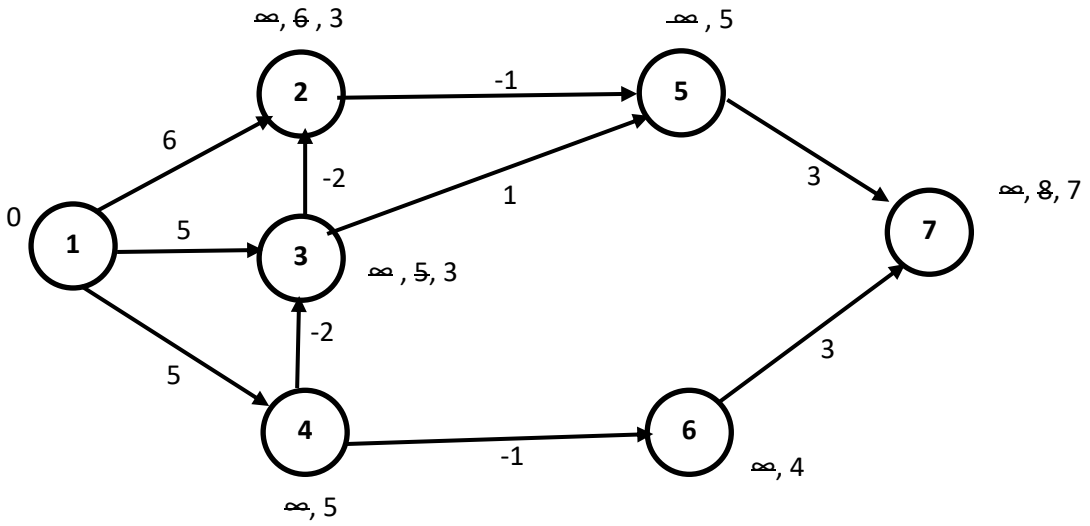
Take edge (1,4):

d[4] = d[1] + C(1,4) = 0 + 5 < ∞ so, it is updated as 5.

Take edge (3,2):

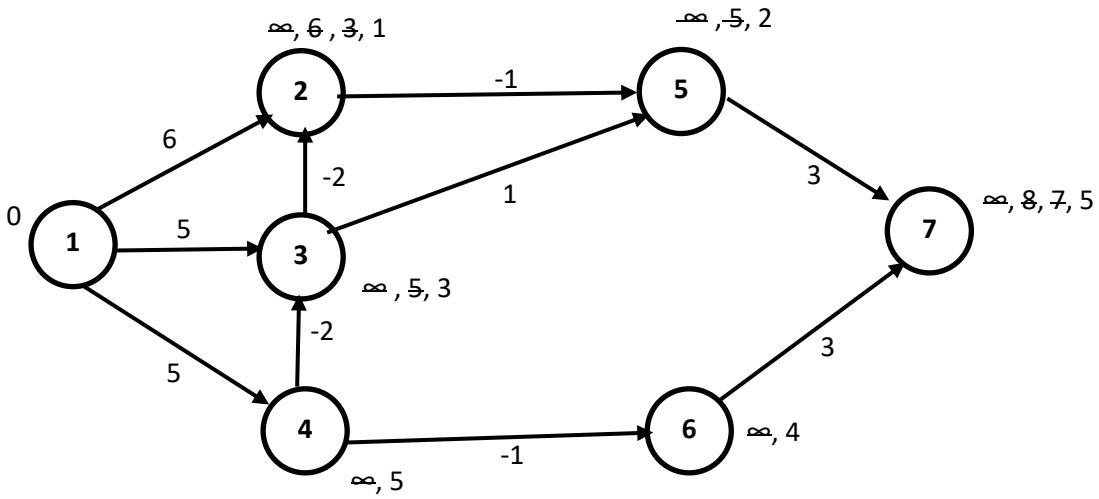
d[2] = d[3] + C(3,2) = 5 + (-2) < 6 so, it is updated as 3.

The following graph shows the shortest path from vertex 0 to all other vertices after relaxation of all the edges.



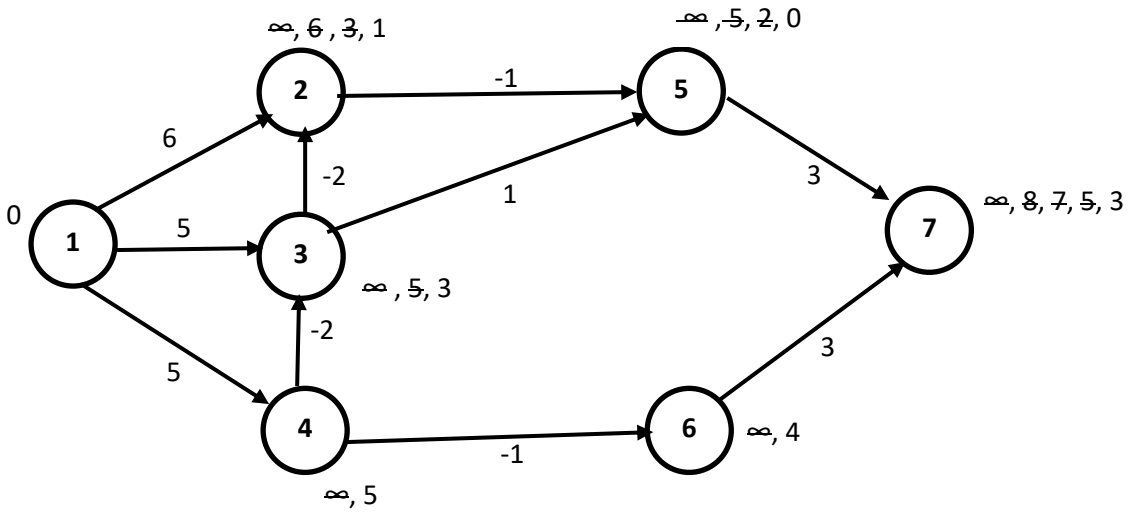
Step 2:

Relax all the vertices second time. The following graph shows the updated distance after relaxing all the vertices.



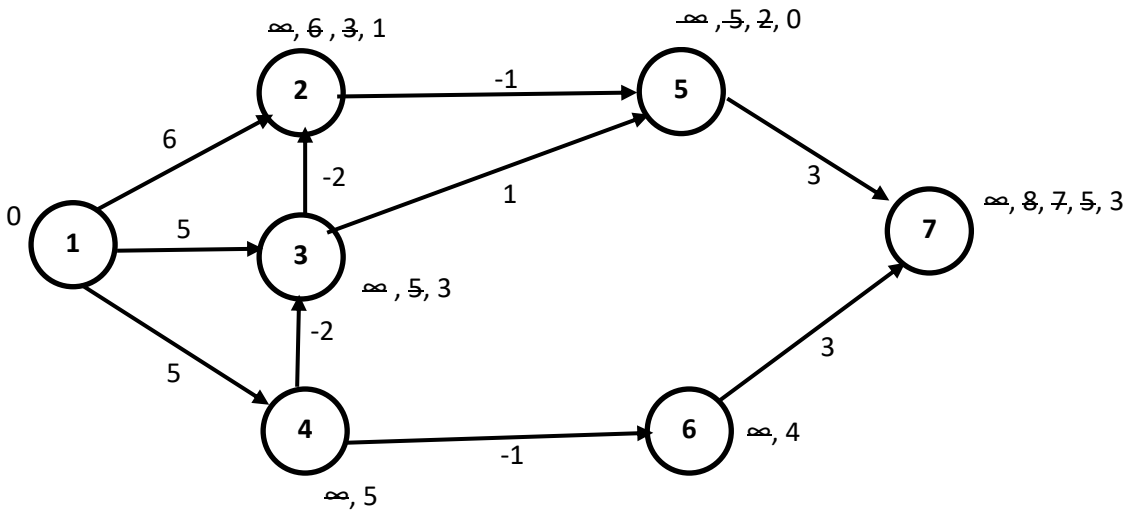
Step 3:

Relax all the vertices third time. The following graph shows the updated distance after relaxing all the vertices.



Step 4:

Relax all the vertices fourth time. The following graph shows the updated distance after relaxing all the vertices.



There is not change in the distance at end of relaxation of vertices in fourth time. So, no need to do the further process.

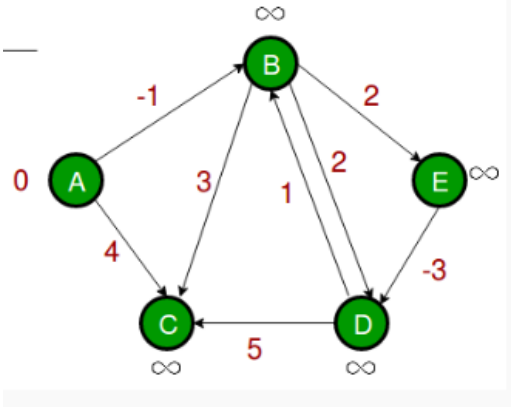
Shortest distances from vertex 0 to other vertices.

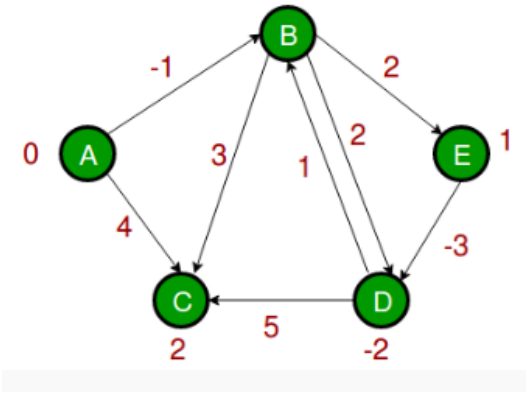
source-destination	Distance
0-1	0
0-2	1
0-3	3
0-4	5
0-5	0
0-6	4
0-7	3

Time complexity:

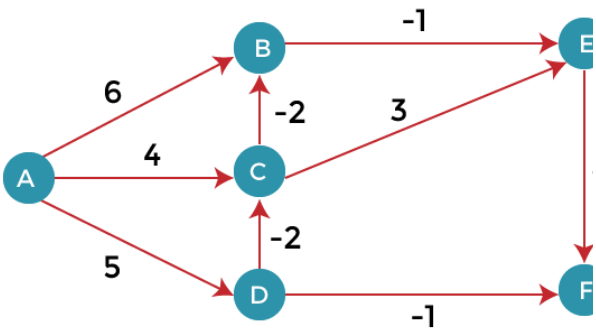
We relax |E| number of edges n-1 (n is number of vertices) number of times so, it is O(|E| * |V|-1). If E and V are equal to n then O(n^2).

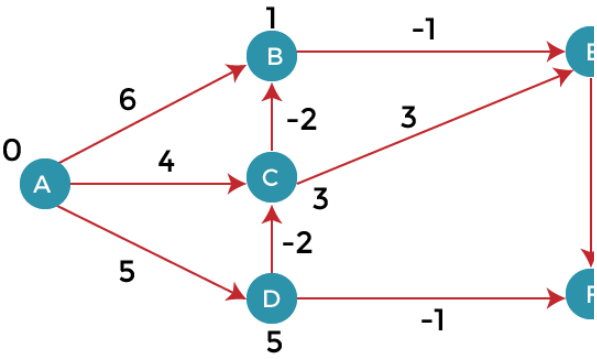
Example 1:





Example 2:





CHAPTER – 8

Graphs, Backtracking and Branch & Bound

8.1 What is graph?

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices (V) and a set of edges(E). The graph is denoted by G (E, V).

Components of a Graph:

Vertices: Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabeled.

Edges: Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labeled/unlabeled.

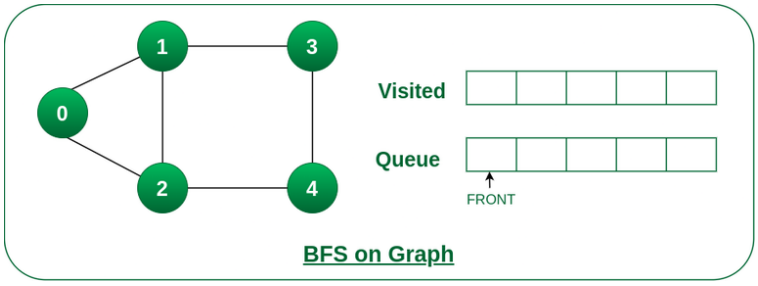
Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex (or node). Each node is a structure and contains information like person id, name, gender, locale etc.

8.2 Breadth First Search (BFS)

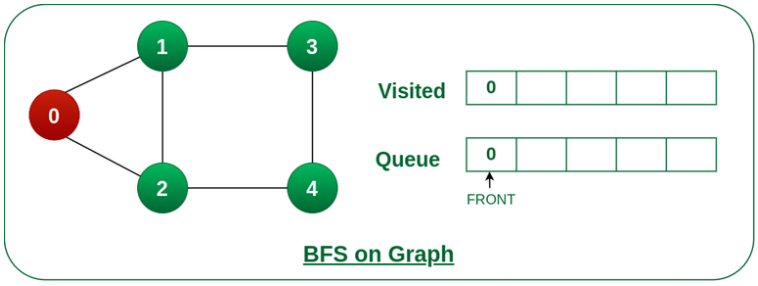
Starting from the root, all the nodes at a particular level are visited first and then the nodes of the next level are traversed till all the nodes are visited.

To do this a queue is used. All the adjacent unvisited nodes of the current level are pushed into the queue and the nodes of the current level are marked visited and popped from the queue.

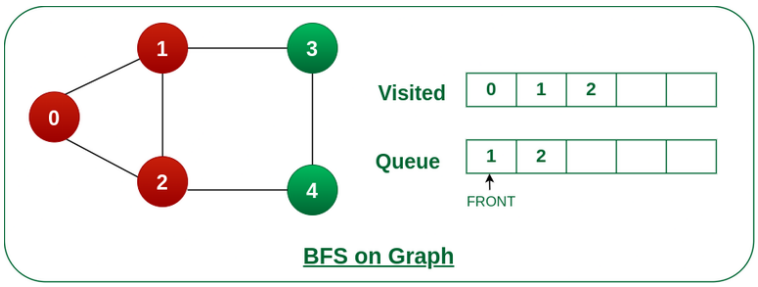
Step 1: Initially visited array and Queue are empty.



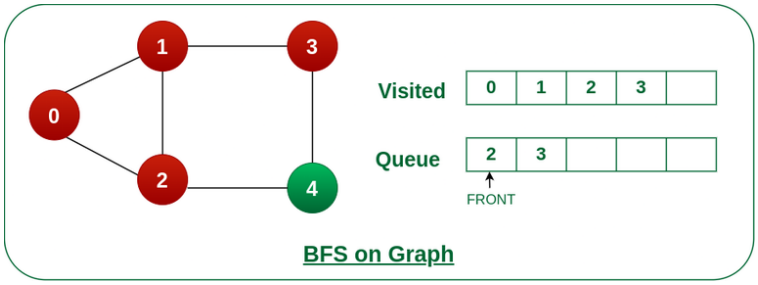
Step 2: Push node 0 in the queue and mark it as visited.



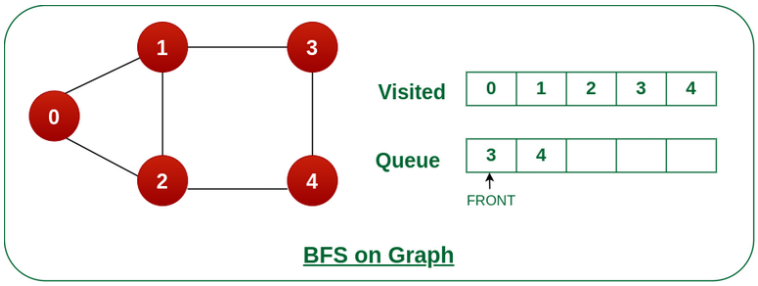
Step 3: Remove node 0 from the front of queue and visit the unvisited neighbors (0 and 1) and push them into queue. Also, add 1 and 2 in the list of visited vertices.



Step 4: Remove node 1 from the front of queue and visit the unvisited neighbors (only 3) and push them into queue. Also, add 3 in the list of visited vertex.

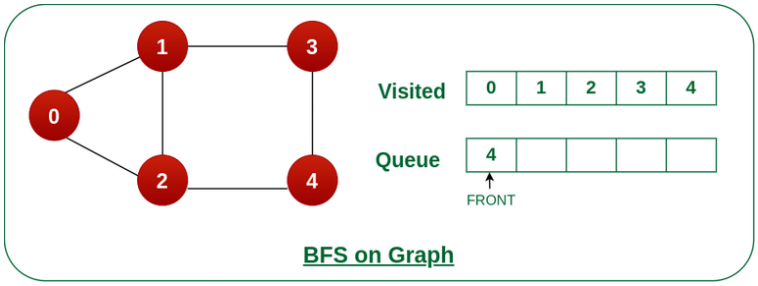


Step 5: Remove node 2 from the front of queue and visit the unvisited neighbors (only 4) and push them into queue. Also, add 4 in the list of visited vertex.



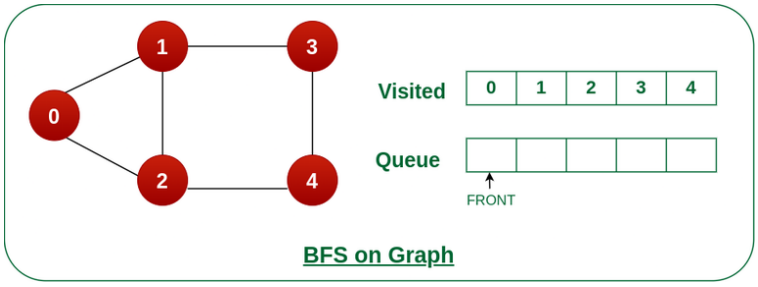
Step 6: Remove node 3 from the front of queue and visit the unvisited neighbors and push them into queue.

As we can see that every neighbor of node 3 is visited, so move to the next node that are in the front of the queue.



Step 7: Remove node 4 from the front of queue and visit the unvisited neighbors and push them into queue.

As we can see that every neighbor of node 4 are visited, so move to the next node that is in the front of the queue but queue has become empty.

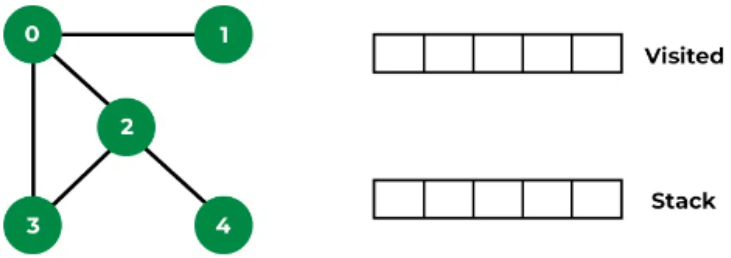


8.3 Depth First Search (DFS)

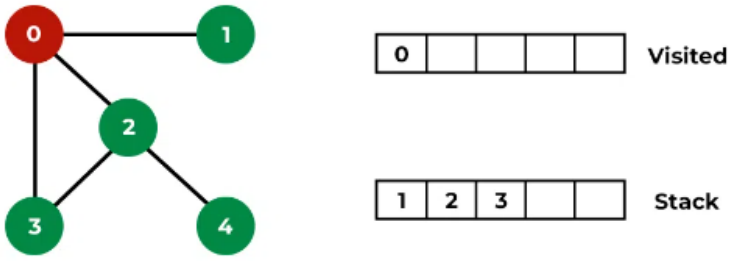
Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.

Let us understand the working of Depth First Search with the help of the following illustration:

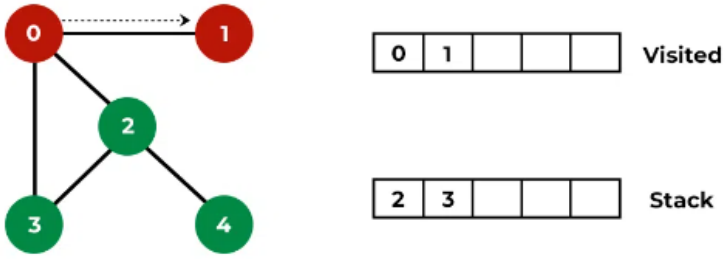
Step 1: Initially stack and visited arrays are empty.



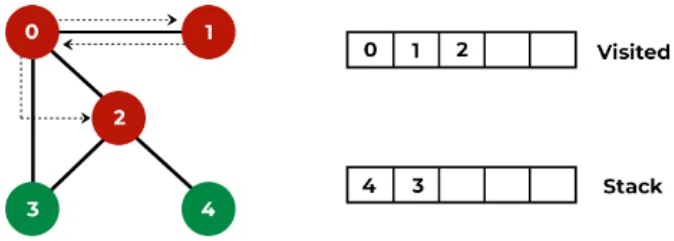
Step 2: Visit 0 and put its adjacent nodes which are not visited yet into the stack.



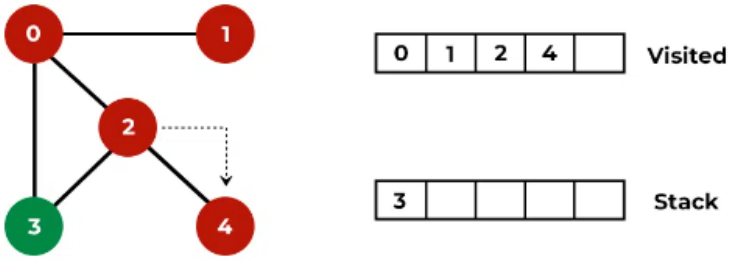
Step 3: Now, Node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



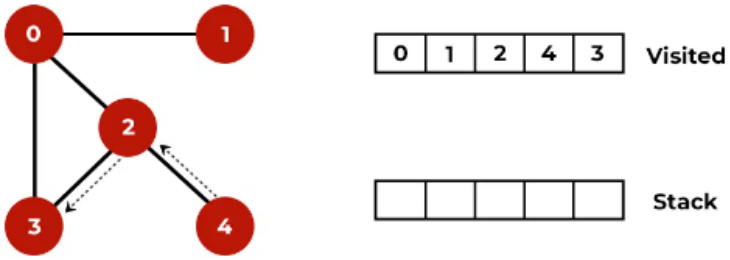
Step 4: Now, Node 2 at the top of the stack, so visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited (i.e, 3, 4) in the stack.



Step 5: Now, Node 4 at the top of the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



Step 6: Now, Node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.



Now, Stack becomes empty, which means we have visited all the nodes and our DFS traversal ends.

8.4 Backtracking

Backtracking is one of the techniques that can be used to solve the problem. We can write the algorithm using this strategy. It uses the Brute force search to solve the problem, and the brute force search says that for the given problem, we try to make all the possible solutions and pick out the best solution from all the desired solutions. This rule is also followed in dynamic programming, but dynamic programming is used for solving optimization problems. In contrast, backtracking is not used in solving optimization problems. Backtracking is used when we have multiple solutions, and we require all those solutions.

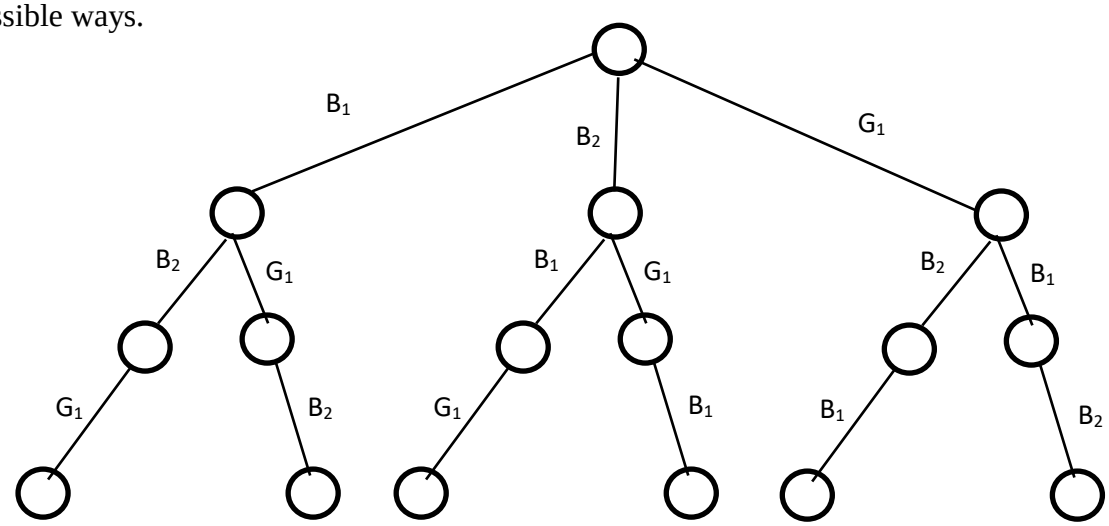
Backtracking name itself suggests that we are going back and coming forward; if it satisfies the condition, then return success, else we go back again. It is used to solve a problem in which a sequence of objects is chosen from a specified set so that the sequence satisfies some criteria.

Backtracking uses the approach of Depth First Search in creating state space tree.

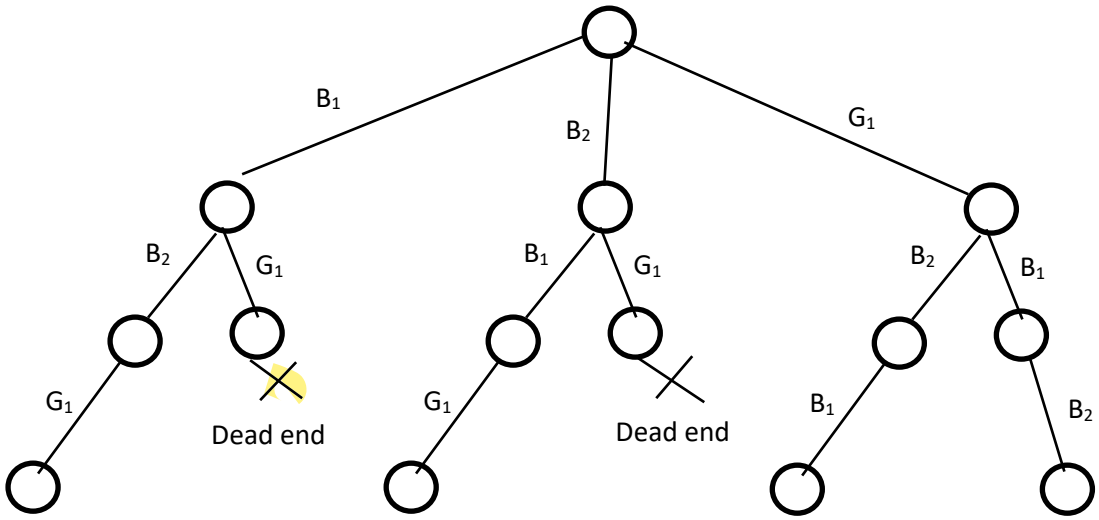
For example:

Let's see the construction of state space tree or solution tree.

We want to arrange two boys and one girl in different order. B_1 B_2 G_1 then we have total 3! possible ways.



Now, we put the constraint in the problem that the girl must not be there between two boys. Then the number of possibilities is 4. We don't further explore (kill the branch) the branch if it does not fulfill the constraint.



8.4.1 N-queens Problem

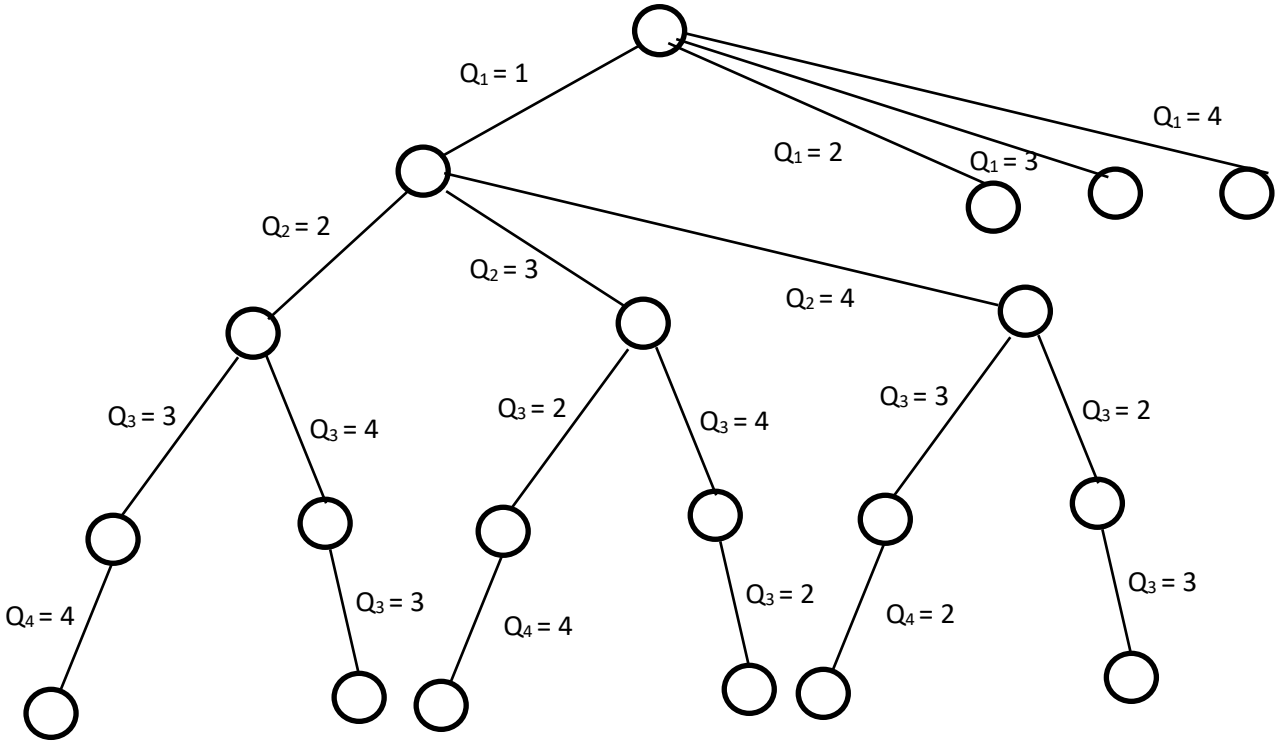
N - Queens problem is to place n – queens on n x n chessboard in such a manner that no queens attack each other by being in the same row, column or diagonal.

It can be seen that for n =1, the problem has a trivial solution, and no solution exists for n =2 and n =3. So first we will consider the 4 queens' problem and then generate it to n - queens problem.

	1	2	3	4
1	Q_1			
2		Q_2		
3				Q_4
4			Q_3	

We can place them in 16_C4 different ways.

Let's first explore the possibilities of putting queens in different rows and column without checking the diagonal constraint.



Now count the number of possible ways with two queens not in same row and column constraint:

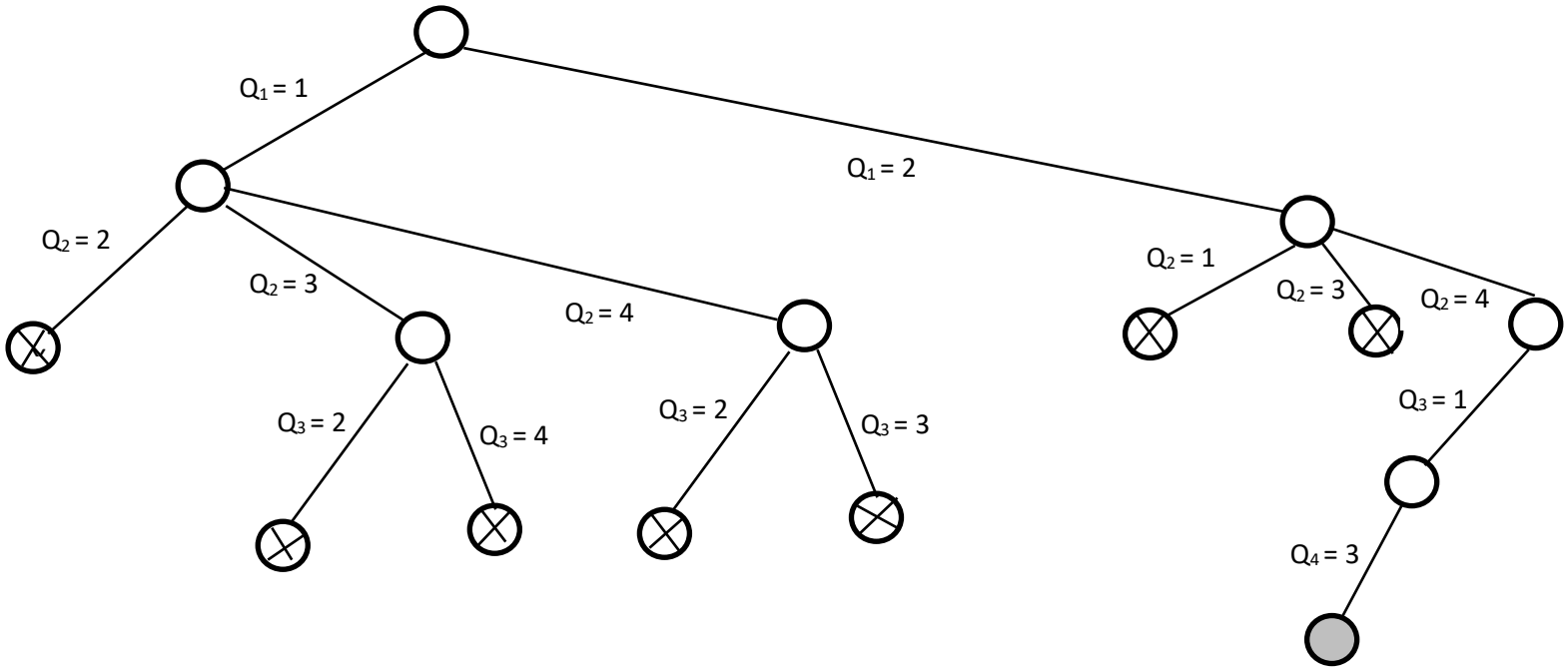
1 + 4 + 4 * 3 + 4 * 3 * 2 + 4 * 3 * 2 * 1 = 65 ways

$$1 + \sum_{i=0}^{3}[\Lambda_{j=0}^{i}(4 - j)]$$

If N * N chessboard is there,

$$\mathbf{1} + \sum_{\boldsymbol{i=0}}^{\mathbf{3}}[\Lambda_{\boldsymbol{j=0}}^{\boldsymbol{i}}(\boldsymbol{N} - \boldsymbol{j})]$$

Now, we generate the state space tree by imposing bounding condition that no two queens in same row, column, or diagonal.



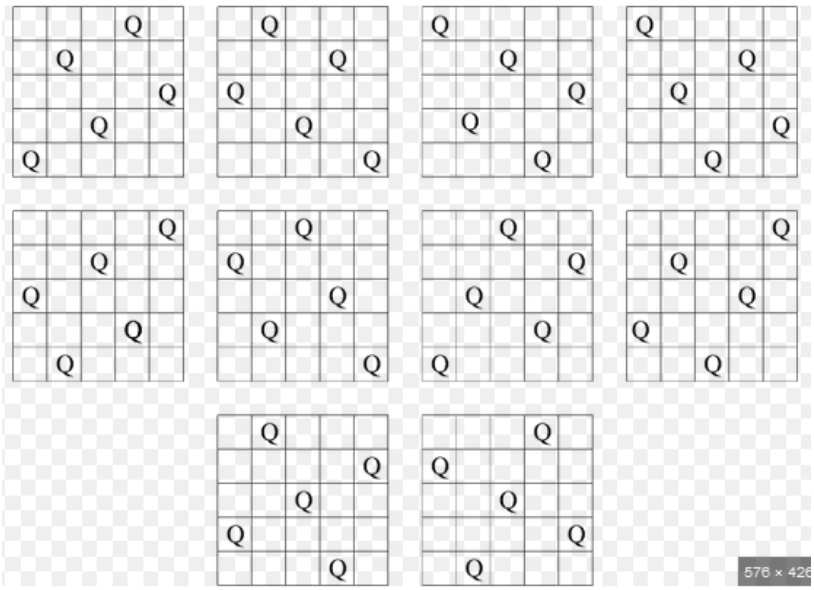
The answer is 2 4 1 3 and its mirror image is also another solution – 3 1 4 2

	1	2	3	4
1		Q_1		
2				Q_2
3	Q_3			
4			Q_4	

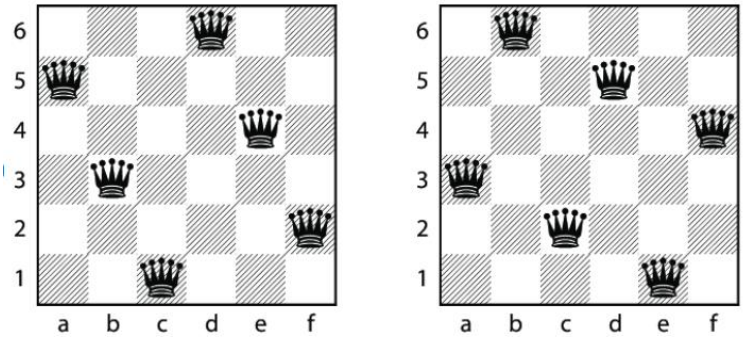
2413

42531

5 Queens problem solution:



6 Queens problem solution:



8 Queens problem's solution:

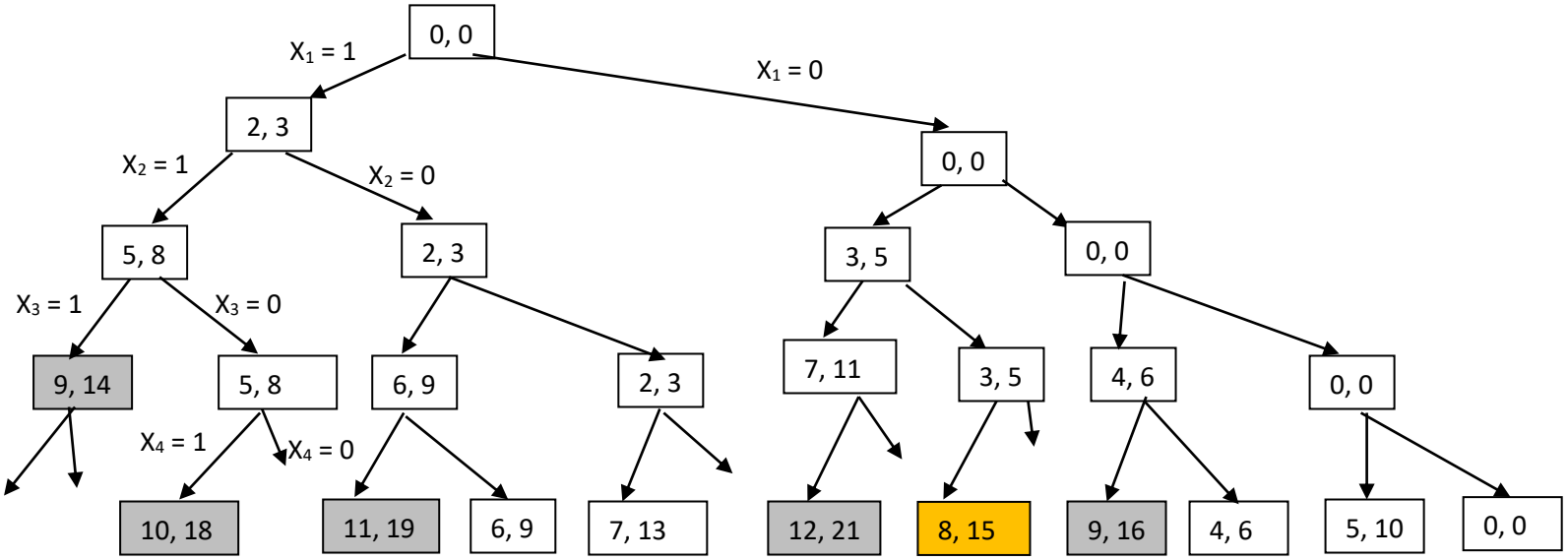
	1	2	3	4	5	6	7	8
1				q_1				
2						q_2		
3								q_3
4		q_4						
5							q_5	
6	q_6							
7			q_7					
8					q_8			

8.4.2 0/1 Knapsack problem using Backtracking

Find the solution for the following 0/1 knapsack problem with weight capacity of 8 using backtracking. Total 2^4 possibilities

Objects	1	2	3	4
Profit	3	5	6	10
Weight	2	3	4	5

a, b - a value shows the weight and b value shows the profit.



8.5 Branch and Bound

Branch and bound is one of the techniques used for problem solving. It is similar to the backtracking since it also uses the state space tree. Branch and Bound method follow breadth first search approach and Backtracking follows depth first search**. Branch and bound is used for solving the optimization problems and minimization problems**. If we have given a maximization problem, then we can convert it using the Branch and bound technique by simply converting the problem into a minimization problem.

Job Scheduling Problem:

Jobs = {j1, j2, j3, j4}

P = {10, 5, 8, 3}

d = {1, 2, 1, 2}

The above are jobs, profit and deadline. We can write the solutions in two ways which are given below:

Suppose we want to perform the jobs j1 and j2 then the solution can be represented in two ways:

The first way of representing the solutions is the subsets of jobs.

S1 = {j1, j4}

The second way of representing the solution is that first job is done, second and third jobs are not done, and fourth job is done.

S2 = {1, 0, 0, 1}

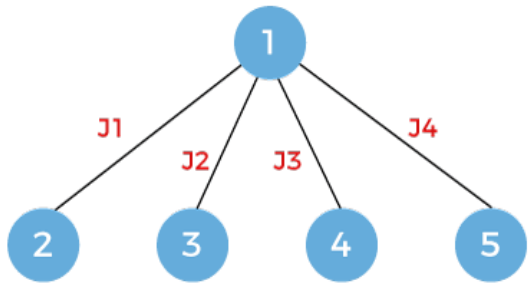
The solution S1 is the variable-size solution while the solution S2 is the fixed-size solution.

There three methods to explore the state space tree in branch and bound approach.

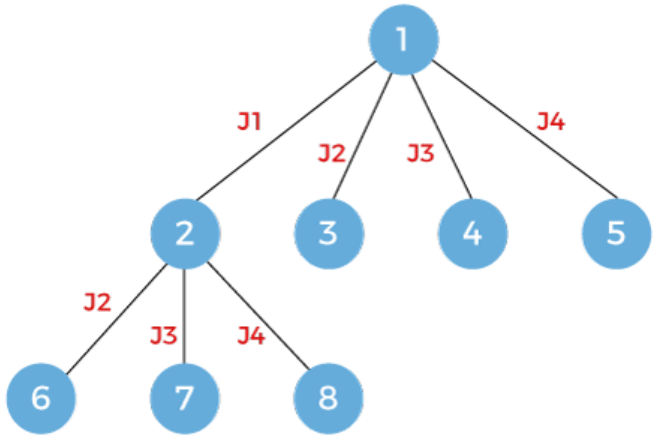
First method:

This method uses the concept of queue. FIFO Branch and Bound. We will use variable size solution.

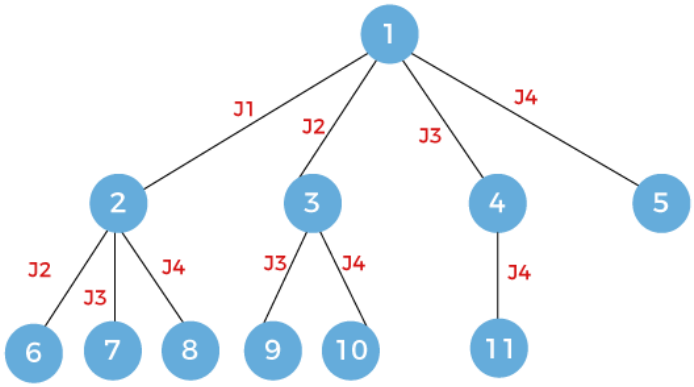
First explore all the four jobs using breadth first search as follows:



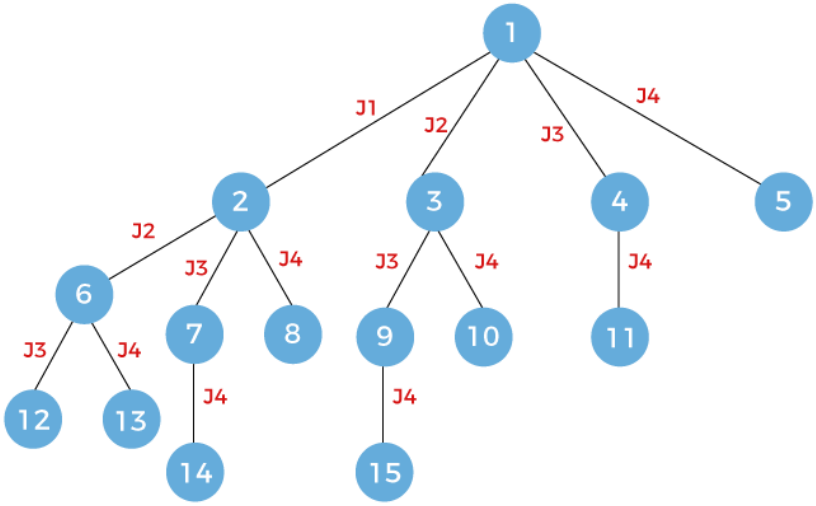
Then explore node 2 where we can schedule J_2, J_3 and J_4.



The following figure shows the exploration of level 2.



Final state space tree is as follows:



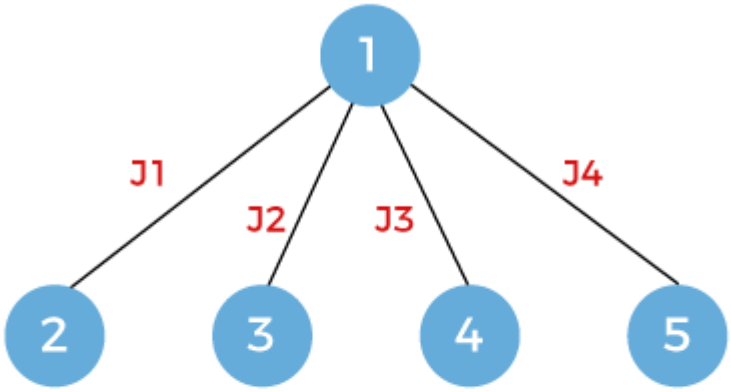
The last node, i.e., node 12 which is left to be expanded. Here, we consider job j4.

Second method:

This method uses concept of stack. It is called LIFO Branch and Bound.

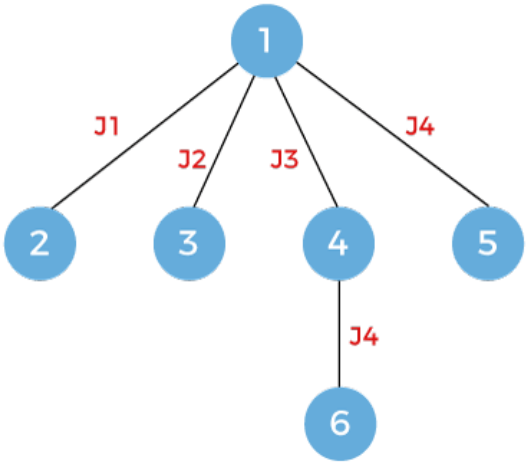
Step 1:

First, expand node 1 and put the nodes on the stack.



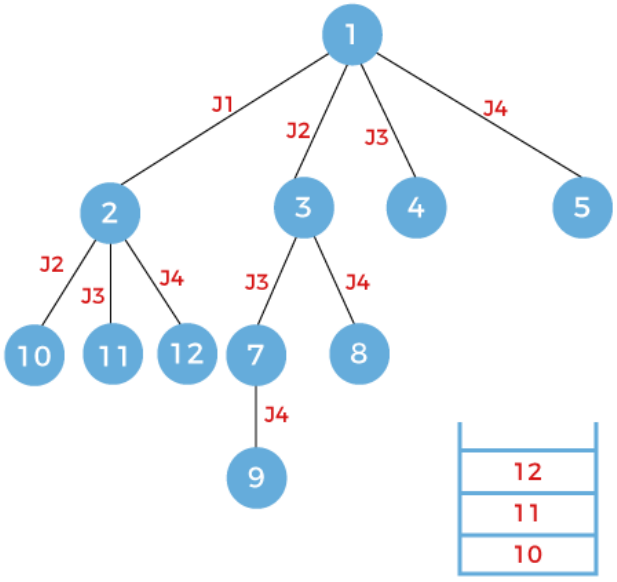
Step 2:

Remove node 5 from the stack and expand it but not possible to expand it further. So, remove another node 4 from the stack and expand it as follows. Also, push the expanded nodes on the stack.



Step 3:

The final state space tree would be as follows:



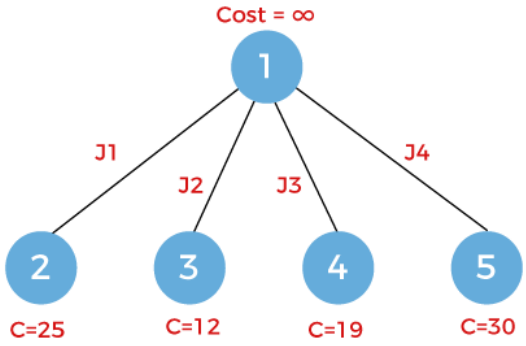
Third method:

There is one more method that can be used to find the solution and that method is Least Cost Branch and Bound (LCBB). In this technique, nodes are explored based on the cost of the node. The cost of the node can be defined using the problem and with the help of the given problem, we can define the cost function. Once the cost function is defined, we can define the cost of the node.

Step 1:

Let's first consider the node 1 having cost infinity shown as below:

Now we will expand the node 1. The node 1 will be expanded into four nodes named as 2, 3, 4 and 5 shown as below:



Important:

In the above, state space tree the following nodes will be explored based on type of method.

FIFO – BB: node 2 will be explored.

LIFO – BB: node 5 will be explored.

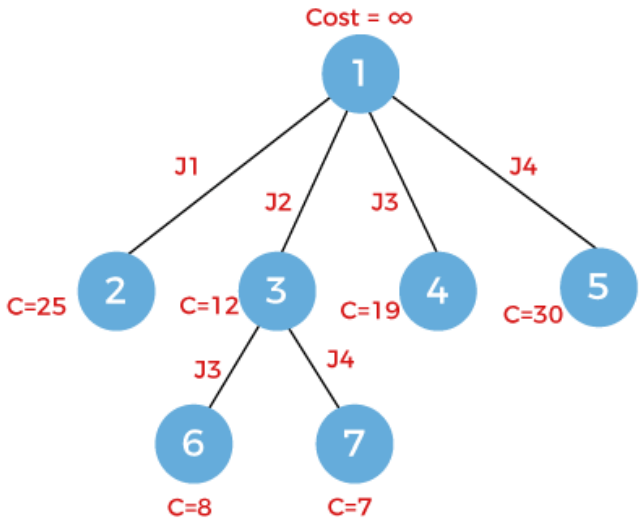
LC – BB: node 3 will be explored.

Step 2:

Let's assume that cost of the nodes 2, 3, 4, and 5 are 25, 12, 19 and 30 respectively.

Since it is the least cost branch and bound, so we will explore the node which is having the least cost. In the above figure, we can observe that the node with a minimum cost is node 3. So, we will explore the node 3 having cost 12.

Since the node 3 works on the job j2 so it will be expanded into two nodes named as 6 and 7 shown as below:


Cost = ∞
1
J1
J2
J3
J4
2
3
4
5
C=25
C=12
C=19
C=30
J3
J4
6
7
C=8
C=7

8.5.1. 0/1 Knapsack problem using Branch & Bound method

For the following 0/1 knapsack problem, find the maximum profit.

Capacity of knapsack – M = 15, Number of objects -N = 4

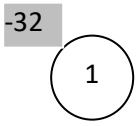
Objects	1	2	3	4
Profit - P	10	10	12	18
Weight	2	4	6	9

We can either go for subset finding method (variable size) or fixed size solution.

We will use Least Cost Branch and Bound with fixed size solution.

Step 1:

Find the initial cost by including objects = 10 (I_1) + 10 (I_2) + 12 (I_3) = -32 and their combined weight is 12 and less than the capacity.



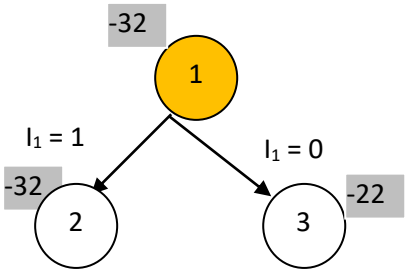
Step 2:

Now, we find the profit by considering I_1 and by not considering I_1.

Profit of Node 2 = -32

Profit of Node 3 = 10 (I_2) + 12 (I_3) = -22

`



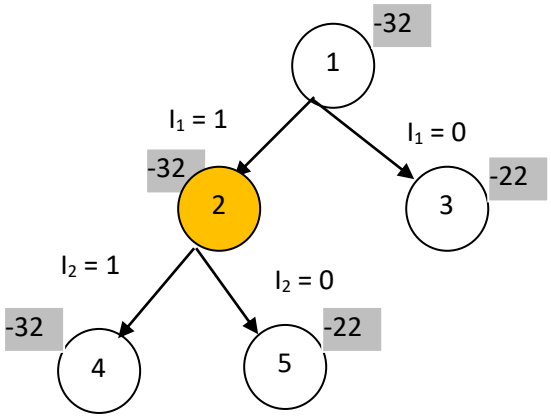
Step 3:

We choose the node 2 for exploration as it gives minimum profit.

Profit of Node 4 = profit will remain -32

Profit of Node 5 = 10 (I_1) + 12 (I_3) = - 22

`



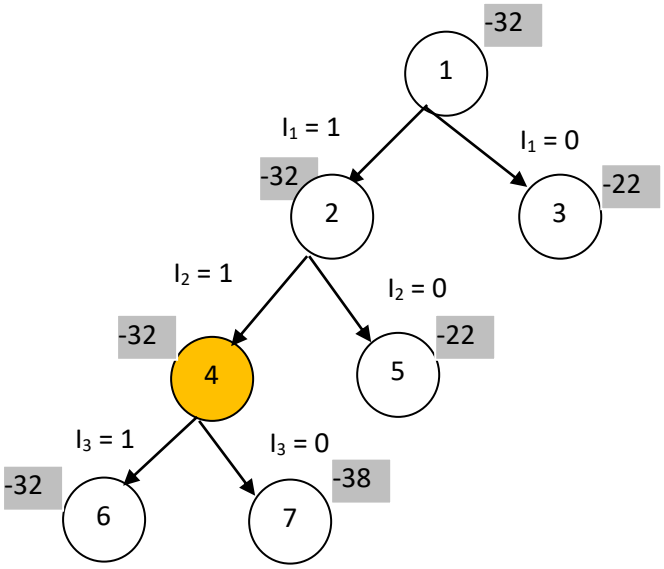
Step 4:

We choose the node 4 for exploration as it gives minimum profit.

Profit of Node 6 = 10 (I_1) + 10 (I_2) + 12 (I_3) = -32

Profit of Node 7 = 10 (I_1) + 10 (I_2) + 18 (I_4) = -38

`



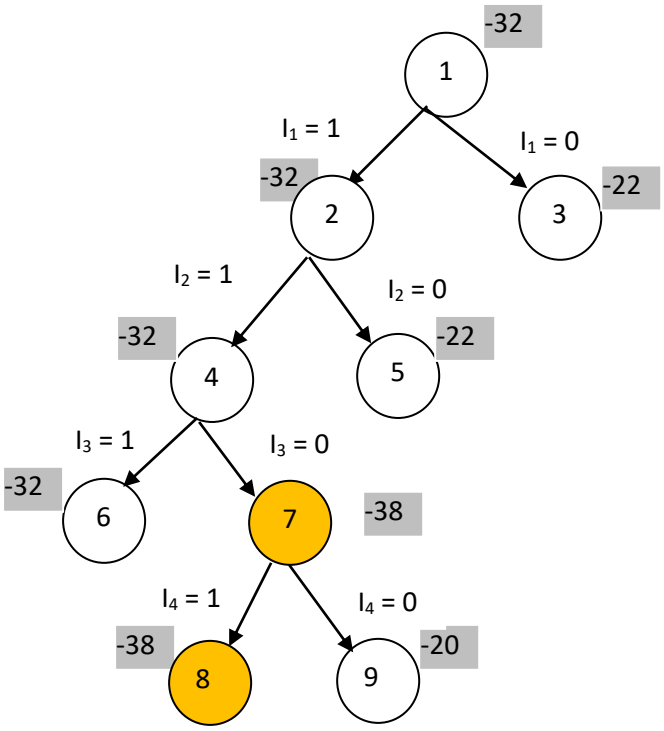
Step 5:

We choose the node 7 for exploration as it gives minimum profit.

Profit of Node 8 = 10 (I_1) + 10 (I_2) + 18 (I_4) = -38

Profit of Node 9 = 10 (I_1) + 10 (I_2) = -20

`



The solution is = {1, 1, 0, 1 } = { 10, 10, 0, 18 } = 38

Example 1:

Find the maximum profit for the following 01/ knapsack problem with capacity (M) of 12.

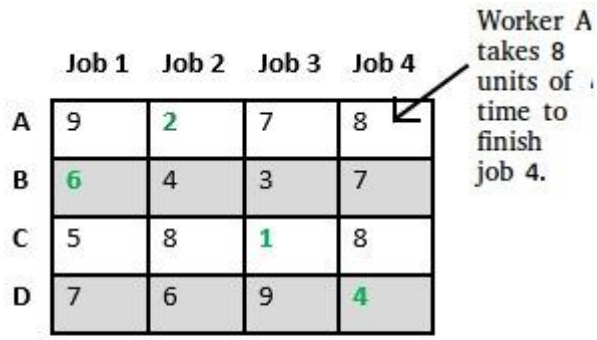
Object	1	2	3	4
Profit – P	30	28	20	24
Weight	5	7	4	2

Solution: {1, 0, 1, 1}, Profit = 74

8.5.2. Assignment problem using Branch and Bound method

Let there be N workers and N jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized.

See the following example:



An example job assignment problem. Green values show optimal job assignment that is A-Job**2**, B-Job1 C-Job3 and D-Job4

Find the total minimum cost of assignment for the following data.

Jobs Persons	1	2	3	4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	2	4

Step 1:

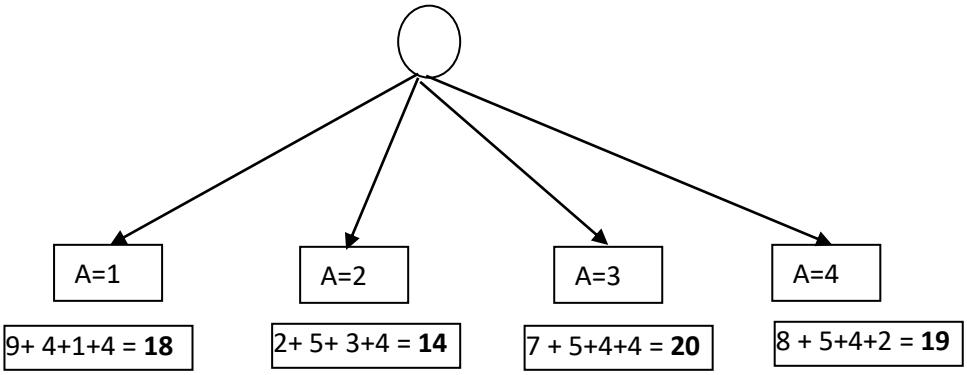
Assign the job to person A with all possible jobs as follows and find the cost.

For example, if we assign Job 1 to person A then = 9 + 4 + 1 + 4 = 18. First, assign job 1 to person A and select the minimum value from each column except row of person A.

Jobs Persons	1	2	3	4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	2	4

If we assign Job 2 to person A then = 5 + 2 + 1 + 4 = 12.

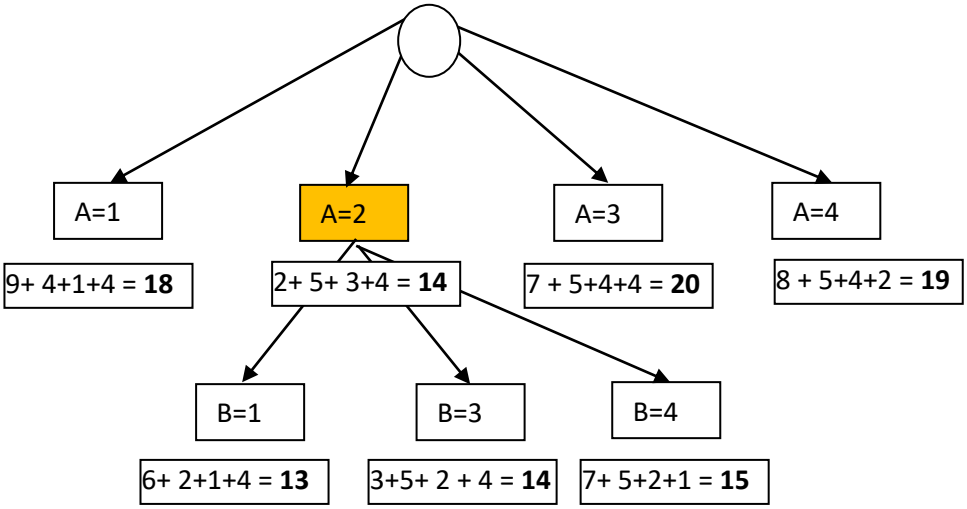
Jobs Persons	1	2	3	4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	2	4



Step 2: Explore the node A=2 with minimum cost by assigning Job 2 to Person A. Already, Job 2 is assigned to Person A so, now start assigning jobs to Person B and it can be assigned Jobs 1, 3 or 4.

For example, if we assign Job 1 to person B (B=1) then = 6 + 2 + 1 + 4 = 13.

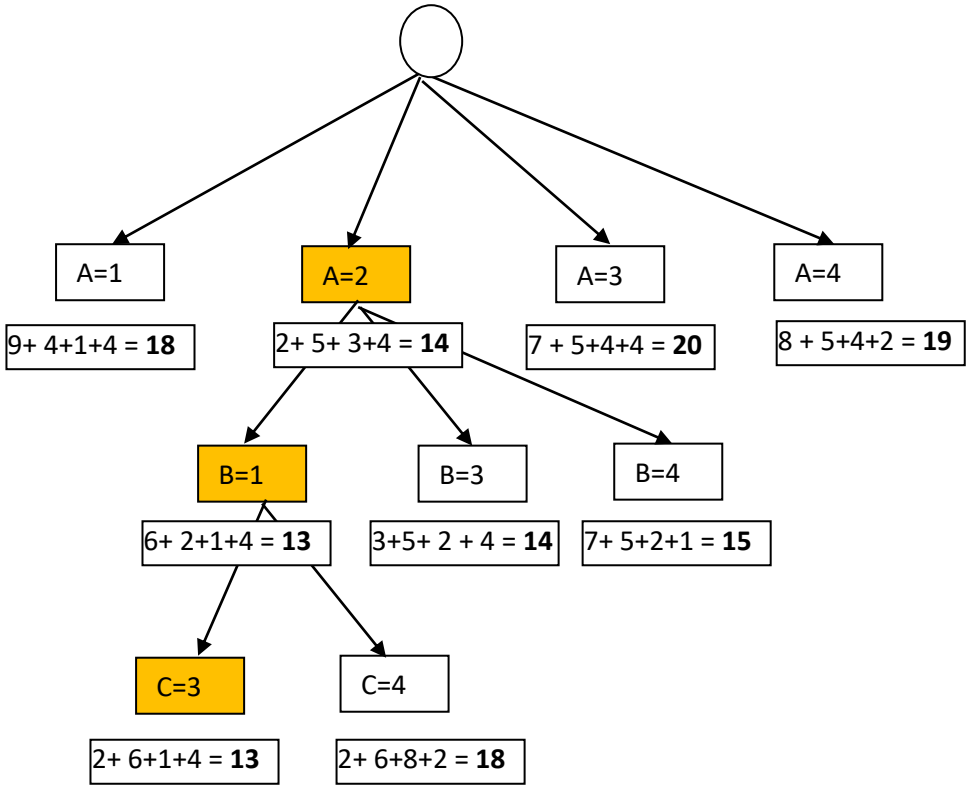
Jobs Persons	1	2	3	4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	2	4



Step 3: Explore the node B=1 with minimum cost by assigning Job 1 to Person B. Already, Job 1 is assigned to Person B so, now start assigning jobs to Person C and it can be assigned Jobs 3 or 4 and remaining person D is also assigned Job 3 or 4.

For example, if we assign Job 3 to person C (C=3) then = 6 + 2 + 1 + 4 = 13 and Person D is assigned Job 4.

Jobs Persons	1	2	3	4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	2	4



The minimum cost is = 6 (Person A is assigned Job 1) + 2 (Person B is assigned Job 2) + 1 (Person C is assigned Job 3) + 4 (Person C is assigned Job 4) = **13**

Example 1:

Find the machines assignment for the following jobs with minimum cost.

		machines			
		I	II	III	IV
jobs	A	10	12	19	11
	B	5	10	7	8
	C	12	14	13	11
	D	8	15	11	9

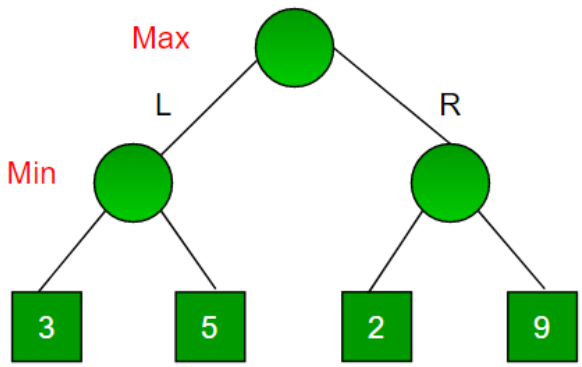
8.5.3. Minimax principle

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.

In Minimax the two players are called maximizer and minimizer. The **maximizer** tries to get the highest score possible while the **minimizer** tries to do the opposite and get the lowest score possible.

Every board state has a value associated with it. In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

Example:
Consider a game which has 4 final states and paths to reach final state are from root to 4 leaves of a perfect binary tree as shown below. Assume you are the maximizing player and you get the first chance to move, i.e., you are at the root and your opponent at next level. Which move you would make as a maximizing player considering that your opponent also plays optimally?

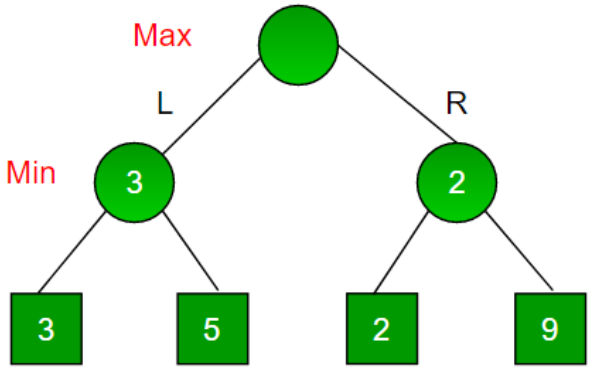


Since this is a backtracking-based algorithm, it tries all possible moves, then backtracks and makes a decision.

- Maximizer goes LEFT: It is now the minimizers turn. The minimizer now has a choice between 3 and 5. Being the minimizer it will definitely choose the least among both, that is 3
- Maximizer goes RIGHT: It is now the minimizers turn. The minimizer now has a choice between 2 and 9. He will choose 2 as it is the least among the two values.

Being the maximizer you would choose the larger value that is 3. Hence the optimal move for the maximizer is to go LEFT and the optimal value is 3.

Now the game tree looks like below:



The above tree shows two possible scores when maximizer makes left and right moves.

Note: Even though there is a value of 9 on the right subtree, the minimizer will never pick that. We must always assume that our opponent plays optimally.

CHAPTER – 9

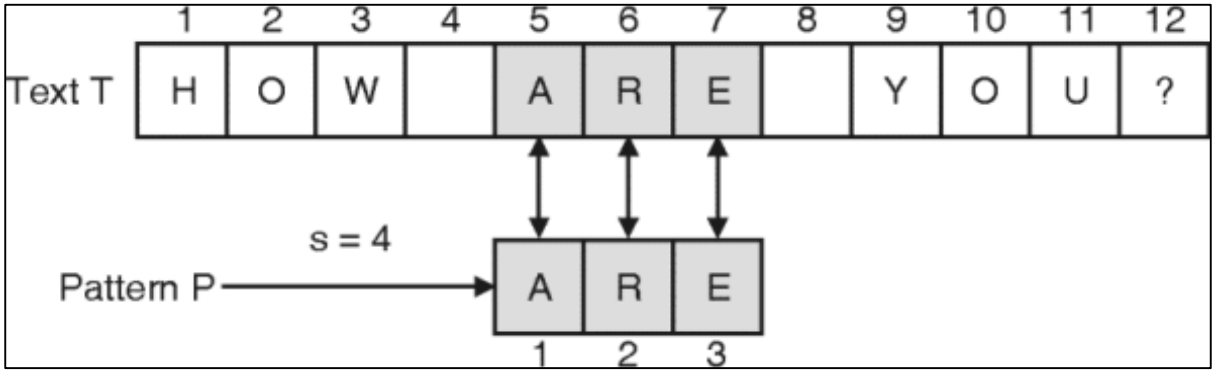
String Matching Algorithms

String matching operation is a core part in many text processing applications. The objective of this algorithm is to find pattern P from given text T. Typically |P|<< |T|. In the design of compilers and text editors, string matching operation is crucial. So, locating P in T efficiently is very important.

The problem is defined as follows: “Given some text string T[1….n] of size n, find all occurrences of pattern P[1…m] of size m in T.”

We say that P occurs in text T with number of shifts s, if $0 \leq s \leq n - m$ and $T[(s + 1) \ldots (s + m)] = P[1 \ldots m]$.

Consider the following example



- In this example, pattern P = ARE is found in text T after four shifts.
- The classical application of such algorithms is to find particular protein pattern in DNA sequence.

Applications of String maching:

Plagiarism Detection: The documents to be compared are decomposed into string tokens and compared using string matching algorithms. Thus, these algorithms are used to detect similarities between them and declare if the work is plagiarized or original.

Bioinformatics and DNA Sequencing: Bioinformatics involves applying information technology and computer science to problems involving genetic sequences to find DNA patterns. String matching algorithms and DNA analysis are both collectively used for finding the occurrence of the pattern set.

Digital Forensics: String matching algorithms are used to locate specific text strings of interest in the digital forensic text, which are useful for the investigation.

Spelling Checker: Trie is built based on a predefined set of patterns. Then, this trie is used for string matching. The text is taken as input, and if any such pattern occurs, it is shown by reaching the acceptance state.

Spam filters: Spam filters use string matching to discard the spam. For example, to categorize an email as spam or not, suspected spam keywords are searched in the content of the email by string matching algorithms. Hence, the content is classified as spam or not.

Search engines or content search in large databases: To categorize and organize data efficiently, string matching algorithms are used. Categorization is done based on the search keywords. Thus, string matching algorithms make it easier for one to find the information they are searching for.

9.1 Naïve String-Matching Algorithm

This is simple and efficient brute force approach. It compares the first character of pattern with searchable text. If a match is found, pointers in both strings are advanced. If a match is not found, the pointer to text is incremented and pointer of the pattern is reset. This process is repeated till the end of the text.

The naïve approach does not require any pre-processing. Given text T and pattern P, it directly starts comparing both strings character by character.

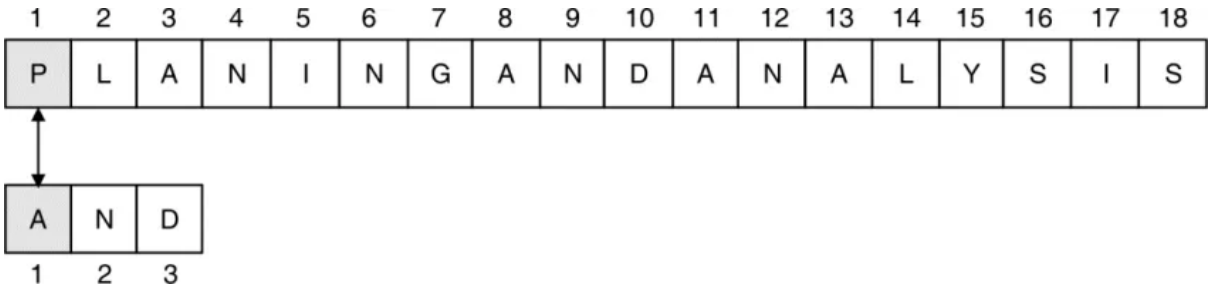
After each comparison, it shifts pattern string *one position* to the right.

Following example illustrates the working of naïve string-matching algorithm. Here,
T = PLANINGANDANALYASIS and P = AND

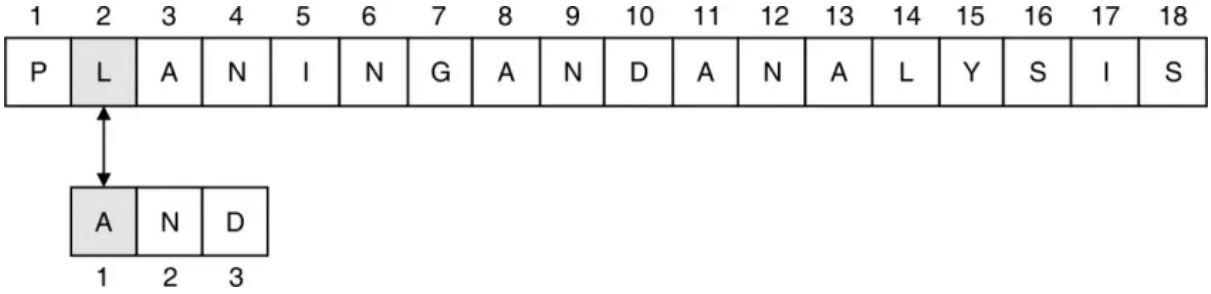
Here, i and j are indices of text and pattern respectively.

Step 1:

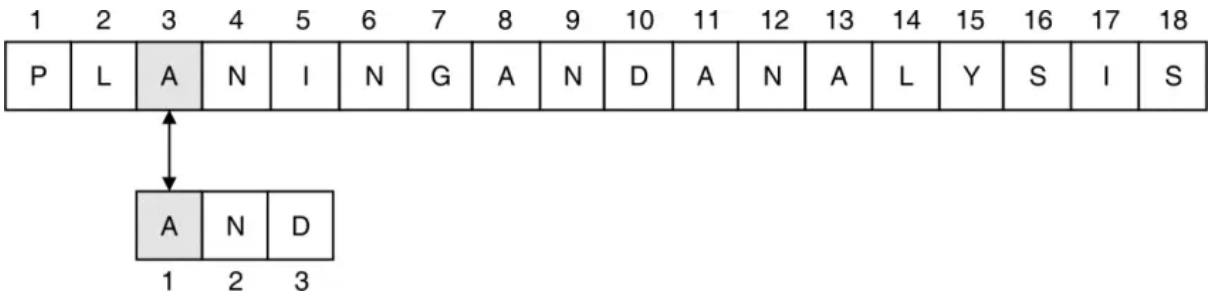
T[1] ≠ P[1], so advance text pointer, i.e. i++.



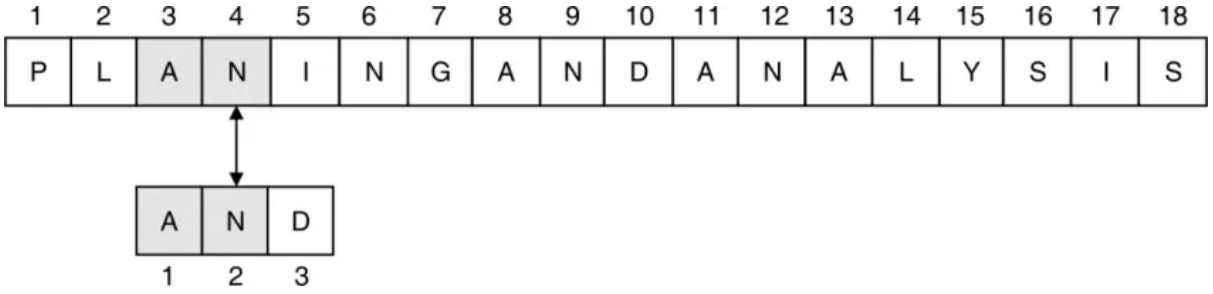
Step 2: T[2] ≠ P[1], so advance text pointers i.e. i++



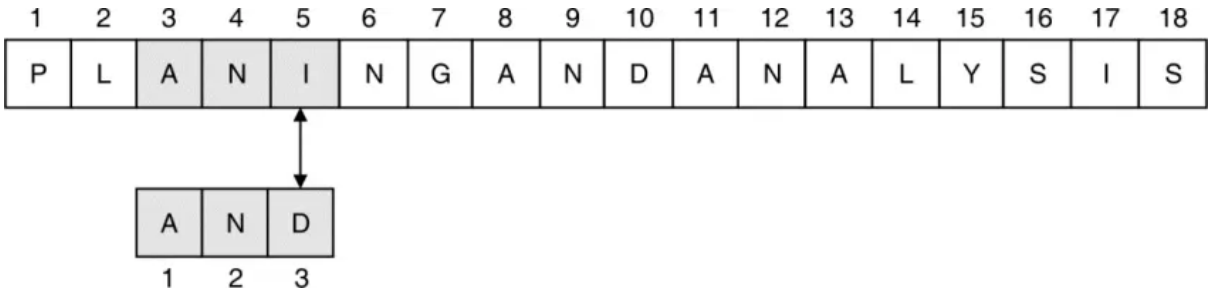
Step 3: T[3] = P[1], so advance both pointers i.e. i++, j++



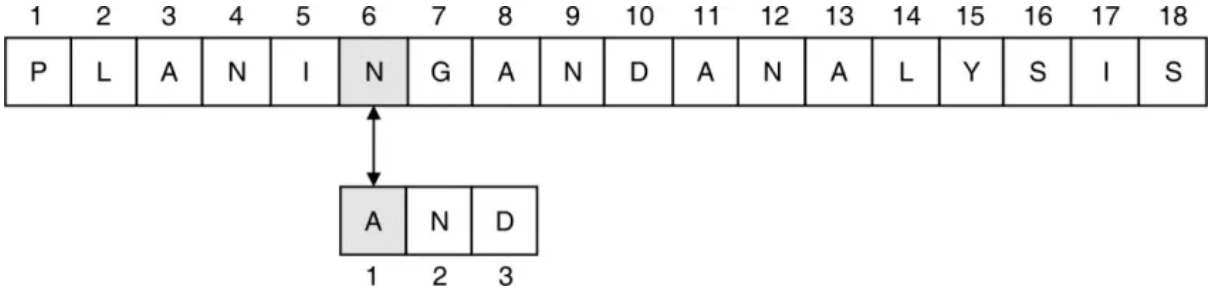
Step 4: T[4] = P[2], so advance both pointers, i.e. i++, j++



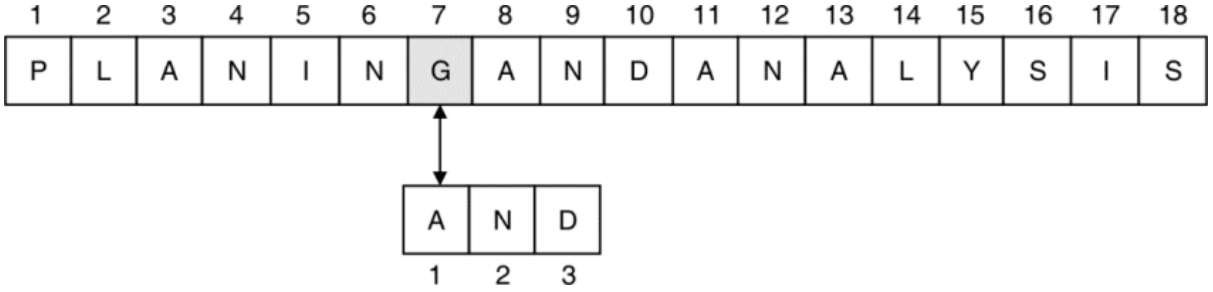
Step 5: T[5] ≠ P[3], so advance text pointer and reset pattern pointer, i.e. i++, j=1



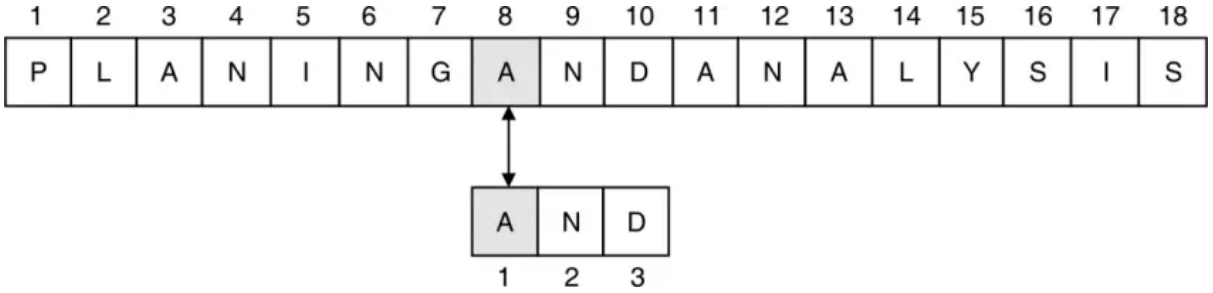
Step 6: T[6] ≠ P[1], so advance text pointer, i.e. i++



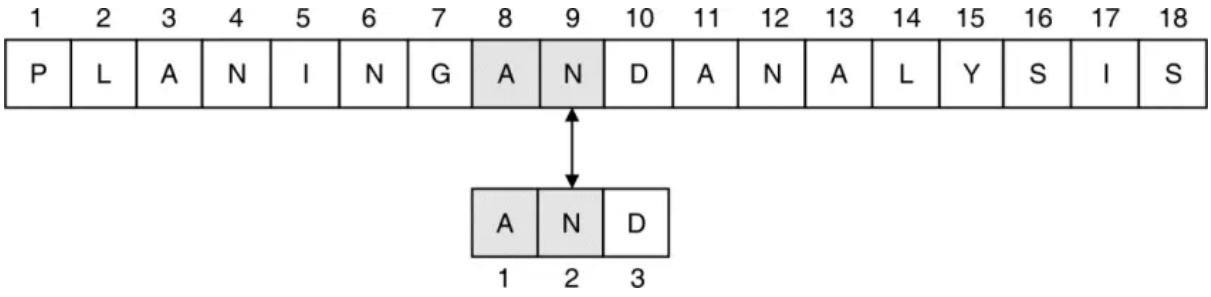
Step 7: T[7] ≠ P[1], so advance text pointer i.e. i++.



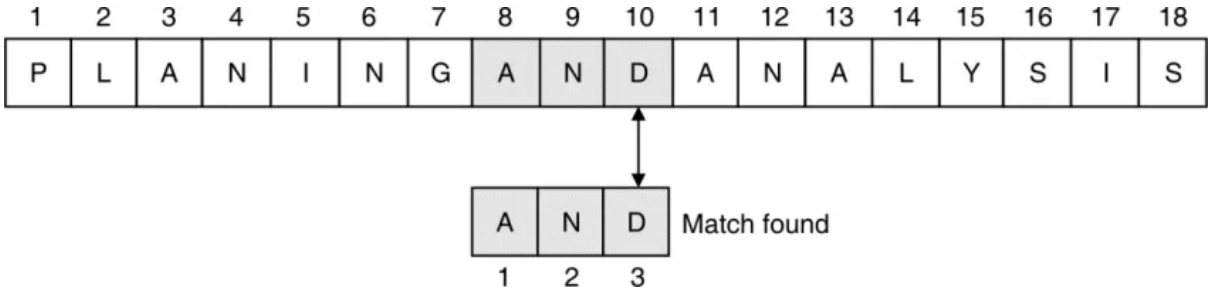
Step 8: T[8] = P[1], so advance both pointers, i.e. i++, j++



Step 9: T[9] = P[2], so advance both pointers, i.e. i++, j++



Step 10: T[10] = P[3], so advance both pointers, i.e. i++, j++



Time complexity analysis:

There are two cases of consideration:

(i) Pattern found

- The worst case occurs when the pattern is at last position and there are spurious hits all the way. Example, T = AAAAAAAAAAB, P = AAAB.
- To move pattern one position right, m comparisons are made. Searchable text in T has a length (n – m). Hence, in worst case algorithm runs in O(m*(n – m)) time.

(ii) Pattern not found

- In the best case, the searchable text does not contain any of the prefixes of the pattern. Only one comparison requires moving pattern one position right.

Example**,** T = ABABCDBCAC, P = XYXZ. The algorithm does O(n – m) comparisons.

- In the worst case, first (m – 1) characters of pattern and text are matched and only last character does not match.

Example**,** T = AAAAAAAAAAAAC, P = AAAAB. Algorithm takes O(m*(n – m)) time.

9.2 Rabin Karp String Matching Algorithm

The Rabin-Karp string matching algorithm calculates a hash value for the pattern, as well as for each M-character subsequences of text to be compared. If the hash values are unequal, the algorithm will determine the hash value for next M-character sequence. If the hash values are equal, the algorithm will analyze the pattern and the M-character sequence. In this way, there is only one comparison per text subsequence, and character matching is only required when the hash values match.

	1	2	3	4	5	6
Text	A	A	A	A	A	B

	1	2	3
Pattern	A	A	B

a-1
b-2
c-3
d-4
...

aab= 1+1+2=4
compare it with string charater sum
(if length of pattern is 3 the compare 3-3 character)

How to convert alphabet into numeric value?

We can take ASCII value of the character but here for explanation purpose let's take the following numeric values.

A – 1, B- 2, C- 3, D- 4, E- 5, F-6 etc..

First find the hash code for the pattern based on their numeric value. The roll the hash code in the text and compare with hash code of the pattern.

A A B = 1 + 1 +2 = 4

Step 1: Take first three characters in the text and find their hash code which is 3 so, there is not match.

	1	2	3	4	5	6
Text	A	A	A	A	A	B
	1	1	1	1	1	2

Step 2: Take next three characters in the text and find their hash code which is 3 so, there is not match.

	1	2	3	4	5	6
Text	A	A	A	A	A	B
	1	1	1	1	1	2

Step 3: Take next three characters in the text and find their hash code which is 3 so, there is not match.

	1	2	3	4	5	6
Text	A	A	A	A	A	B
	1	1	1	1	1	2

Step 4: Take next three characters in the text and find their hash code which is 4 so, the match is found and corresponding characters are matched.

	1	2	3	4	5	6
Text	A	A	A	A	A	B
	1	1	1	1	1	2

Let's take another example:

	1	2	3	4	5	6	7	8	9	10	11
Text	C	C	A	C	C	A	A	E	D	B	A

	1	2	3
Pattern	D	B	A
	4	2	1

Hash code of the pattern is = 4 + 2 + 1 = 7

Step 1: Hash code of the first three characters is 7 but they don't match with the pattern.

	1	2	3	4	5	6	7	8	9	10	11
Text	C	C	A	C	C	A	A	E	D	B	A
	3	3	1								

Step 2: Hash code of the next three characters is also 7 but they don't match with the pattern.

	1	2	3	4	5	6	7	8	9	10	11
Text	C	C	A	C	C	A	A	E	D	B	A
		3	1	3							

Step 3: Hash code of the next three characters is also 7 but they don't match with the pattern.

	1	2	3	4	5	6	7	8	9	10	11
Text	C	C	A	C	C	A	A	E	D	B	A
			1	3	3						

This situation is called spurious hit where hash code of pattern and substring in text is similar, but the corresponding characters don't match exactly.

Time taken by the algorithm is O(m * n) where m is the length of pattern and n is the length of text.

Now, how to avoid the spurious hits?

	1	2	3	4	5	6	7	8	9	10	11
Text	C	C	A	C	C	A	A	E	D	B	A
	1	2	3								
Pattern	D	B	A								
	$4 * 10^2$	$2 * 10^1$	$1 * 10^1$								
	400	20	1	=421							

Here, we have taken power to the 10 because it is assumed that in the text, we have only 10 characters. But if we have all the characters in the text then we can take power to the 127 also.

Step 1: Hash code of the first three characters is 331 which does not match with hash code of pattern 421.

	1	2	3	4	5	6	7	8	9	10	11
Text	C	C	A	C	C	A	A	E	D	B	A
	3	3	1								
	$3 * 10^2$	$3 * 10^1$	$1 * 10^0$								
	300	30	1	=331							

How to do the rolling of the hash function?

	1	2	3	4	5	6	7	8	9	10	11
Text	C	C	A	C	C	A	A	E	D	B	A
	3	3	1								
	$3 * 10^2$	$3 * 10^1$	$1 * 10^0$	=331							

$3 * 10^2 + 3 * 10^1 + 1 * 10^0 = 331$

$3 * 10^2 + 3 * 10^1 + 1 * 10^0 - 3 * 10^2 = 331 – 300 = 31 * 10 = 310 + 3 * 10^0 = 313$

Still there is one problem here. If the value of hash code is increased than size of the data type, then we can take mod function to avoid the overflow of the data. The mod value depends upon the size of data type.

Time complexity:

Best case: O(n-m+1)

Worst case: O(m * n)

Example:

Text - A B C C D D A E F G

Pattern – C D D

9.3 Knuth-Morris-Pratt (KMP) String Matching Algorithm

Knuth-Morris and Pratt introduce a linear time algorithm for the string-matching problem. A matching time of O (n) is achieved by avoiding comparison with an element of 'S' that have previously been involved in comparison with some element of the pattern 'p' to be matched. i.e., backtracking on the string 'S' never occurs.

Text - A B C D A B C A B C D F

Pattern - A B C D F

As per the naïve approach, we start matching the corresponding characters in text and pattern until there is mismatch. When the mismatch occurs, we shift one character in text and reset the pattern string from the beginning.

The running time of the naïve approach is O(m * n).

Let see the concepts of prefix and suffix before we procced for the mechanism of KMP algorithm.

A B C D A B C

Prefix: Any valid subset of the string from left to right. A, AB, ABC, ABCD, ABCDA etc.

Suffix: Any valid subset of the string from right to left. C, BC, ABC, DABC etc.

Is there any suffix same as prefix in the pattern? Yes, ABC is there in both. Simply, we are checking that beginning part of the pattern is appearing again in the text or not so, we have a smaller number of comparisons unlike naïve approach. This observation is done by the KMP algorithm in finding the sub string.

Let's see the process of calculating π table or LPS (Longest Prefix which is also Suffix) table.

Index	1	2	3	4	5	6	7	8	9	10
Character	A	B	C	D	A	B	E	A	B	F
Index of previous appearance - LPS	0	0	0	0	1	2	0	1	2	0

When we scan the pattern and if same character appears again then we write its index number. A and B appears again so, we put 1 for A and 2 for B as above.

Examples of finding LPS value:

Index	1	2	3	4	5	6	7	8	9	10	11
Character	A	B	C	D	E	A	B	F	A	B	C
Index of previous appearance - LPS	0	0	0	0	0	1	2	0	1	2	3

Index	1	2	3	4	5	6	7	8	9
Character	A	A	A	A	B	A	A	C	D
Index of previous appearance - LPS	0	1	2	3	0	1	2	0	0

Find the sub string in the following example using KMP algorithm:

Text

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	A	B	C	A	B	C	A	B	A	B	A	B	D

Pattern

A	B	A	B	D

Solution:

i is the index in the Text.

j is index in the Pattern.

Text

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	A	B	C	A	B	C	A	B	A	B	A	B	D

Pattern

	0	1	2	3	4
	A	B	A	B	D
LPS	0	0	1	2	0

Following the below process to increment the value of i and j to iterate the process.

- If there is a match, increment both i and j.
- If there is a mismatch after a match, place j at LPS[j - 1]] and compare again.
- If j = 0, and there is a mismatch, increment i.

Step 1:

i = 0 and j = 0

Text –

i														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	A	B	C	A	B	C	A	B	A	B	A	B	D

Pattern –

	j				
index	0	1	2	3	4
	A	B	A	B	D
LPS	0	0	1	2	0

Check T[0] and P[0], they match so, increase i and j both.

Step 2:

Text –

	i													
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	A	B	C	A	B	C	A	B	A	B	A	B	D

Pattern –

		j			
	0	1	2	3	4
	A	B	A	B	D
LPS	0	0	1	2	0

Check T[1] and P[1], they match so, increase i and j both.

Step 3:

Text –

		i												
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	A	B	C	A	B	C	A	B	A	B	A	B	D

Pattern –

			j		
	0	1	2	3	4
	A	B	A	B	D
LPS	0	0	1	2	0

Check T[2] and P[2], they match so, increase i and j both.

Step 4:

Text –

			i											
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	A	B	C	A	B	C	A	B	A	B	A	B	D

Pattern –

				j	
	0	1	2	3	4
	A	B	A	B	D
LPS	0	0	1	2	0

Check T[3] and P[3], they match so, increase i and j both.

Step 5:

Text –

				i										
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	A	B	C	A	B	C	A	B	A	B	A	B	D

Pattern –

					j
	0	1	2	3	4
	A	B	A	B	D
LPS	0	0	1	2	0

Check T[4] and P[4], they don’t match so, move j to index on 2 (LPS[j-1], it means LPS[4-1] = LPS[3] = 2) based on the LPS array.

			j		
	0	1	2	3	4
	A	B	A	B	D
LPS	0	0	1	2	0

Again compare T[4] and T[2], they don’t match so, move j to index 0 (LPS[j-1] = LPS[2-1]=LPS[0]=0). Now, j has become zero so, increment i in the next step (step no. 6).

	j				
	0	1	2	3	4

	A	B	A	B	D
LPS	0	0	1	2	0

Step 6:

Text –

					i									
	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	A	B	C	A	B	C	A	B	A	B	A	B	D

	j				
	0	1	2	3	4
	A	B	A	B	D
LPS	0	0	1	2	0

Check T[5] and P[0], they match so, increment both i and j.

Step 7:

Text –

						i								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	A	B	C	A	B	C	A	B	A	B	A	B	D

		j			
	0	1	2	3	4
	A	B	A	B	D
LPS	0	0	1	2	0

Check T[6] and P[1], they match so, increment i and j both.

Step 8:

Text –

							i							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	A	B	C	A	B	C	A	B	A	B	A	B	D

			j		
	0	1	2	3	4
	A	B	A	B	D
LPS	0	0	1	2	0

Check T[7] and P[2], they don’t match so, move j to index 0 (LPS[j-1]=LPS[2-1]=LPS[1]=0)

	j				
	0	1	2	3	4
	A	B	A	B	D
LPS	0	0	1	2	0

Compare T[7] and P[0], they don’t match and j is zero so, increment i only in next step (step no. 9).

Step 9:

Text –

								i						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	A	B	C	A	B	C	A	B	A	B	A	B	D

	j				
	0	1	2	3	4
	A	B	A	B	D
LPS	0	0	1	2	0

Check T[8] and P[0], they match so, increment i and j both.

Step 10:

Text –

									i					
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	A	B	C	A	B	C	A	B	A	B	A	B	D

		j			
	0	1	2	3	4
	A	B	A	B	D
LPS	0	0	1	2	0

Check T[9] and P[1], they match so, increment i and j both.

Step 11:

Text –

										i				
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	A	B	C	A	B	C	A	B	A	B	A	B	D

			j		
	0	1	2	3	4
	A	B	A	B	D
LPS	0	0	1	2	0

Check T[10] and P[2], they match so, increment i and j both.

Step 12:

Text –

											i			
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	A	B	C	A	B	C	A	B	A	B	A	B	D

				j	
	0	1	2	3	4
	A	B	A	B	D
LPS	0	0	1	2	0

Check T[11] and P[3], they match so, increment i and j both.

Step 13:

Text –

												i		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	A	B	C	A	B	C	A	B	A	B	A	B	D

					j
	0	1	2	3	4
	A	B	A	B	D
LPS	0	0	1	2	0

Check T[12] and P[4], they don’t match so, move j to LPS[j-1]=LPS[4-1]=LPS[3]=2 as follows.

			j		
	0	1	2	3	4
	A	B	A	B	D
LPS	0	0	1	2	0

Compare T[12] and P[2], they match so, increment i and j both.

Step 14:

Text –

													i	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	A	B	C	A	B	C	A	B	A	B	A	B	D

				j	
	0	1	2	3	4
	A	B	A	B	D
LPS	0	0	1	2	0

Check T[13] and P[3], they match and increment i and j both.

Step 15:

Text –

														i
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	B	A	B	C	A	B	C	A	B	A	B	A	B	D

					j
	0	1	2	3	4
	A	B	A	B	D
LPS	0	0	1	2	0

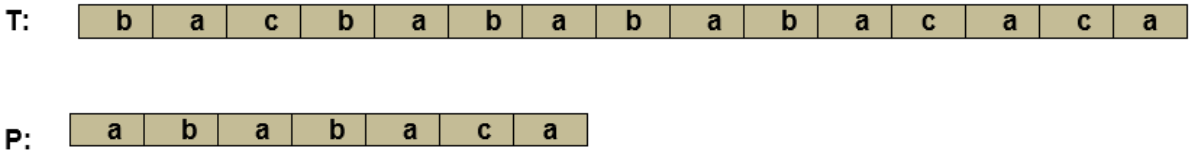
Check T[14] and P[4], they match and increment i and j both. There is end of text and pattern, and string matching is found for the pattern.

Time Complexity: O(n + m) where n is the length of the text and m is the length of the pattern.

Auxiliary Space: O(m) for storing LPS values of pattern of length m.

Example:

Find the following pattern in the given text using KMP algorithm.



CHAPTER – 10

P, NP, NP Hard and NP Complete

Look at the following problems and they are categorized based on their time complexity.

Polynomial time	**Exponential time**
Linear search - n	0/1 knapsack – 2^n
Binary search – log n	Travelling salesperson - 2^n
Insertion sort – n^2	Graph coloring - 2^n
Merge sort – n log n	Hamiltonian cycle - 2^n
Matrix multiplication – n^3	

Time complexity in exponential is much larger than polynomial time so, we want polynomial time algorithm for solving the above problems and it is area of research.

Till date, no solution is found to solve the problem which are taking exponential time to reduce their time complexity to polynomial time.

The framework is prepared to make the research on these types of problems which are taking exponential time.

NP-Hard

NP-Complete

There are two points based on the entire things will be clear:

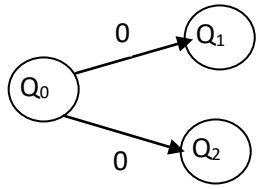
1) Try to show the similarities between them so, if one is solved then another will also be solved.
2) If we are not able to write polynomial time deterministic algorithm, then write polynomial time non-deterministic algorithm.

What are deterministic and non-deterministic algorithms?

Deterministic algorithm: Every step of the algorithm is known, and it shows the clear operation to perform or what is to be done.

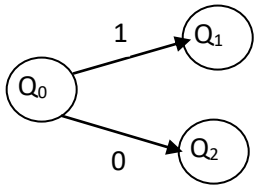
Non-deterministic algorithms: Output can't be predicted properly even if we know the input. The same input to the algorithm will produce different output on different rounds of the execution.

Non-deterministic algorithm:



For the same input, it will produce different output.

Deterministic algorithm:



For different input, it will produce different output.

Suppose we write the non-deterministic algorithm for 0/1 knapsack problem then some of the steps are deterministic, and some may be non-deterministic. We don't know how these non-deterministic statements are figured out and may leave them blank. But most of the statements are deterministic. We can write non-deterministic algorithms for some problem, and we can preserve the work so, someone can propose deterministic solution of the problem in future.

<u>If we are not able to write polynomial time deterministic algorithm, then write polynomial time non-deterministic algorithm.</u>

Lets write the non-deterministic algorithm for searching which takes O (1) time and taking polynomial time.

```
Algorithm NSearch(A,n,key)
{
        j = choice();                   // 1 this takes O(1)
        if( key = A[j])
        {
        write(j);
        Success();                      //1 this takes O(1)
        }
        else
        {
        write(0);
        Failure();                      //1 this takes O(1)
        }
}
```

choice() function directly gives the index of the key element from the array but how it does we don't know. At this point of time, we don't know how it locates the key from the array and return index j.

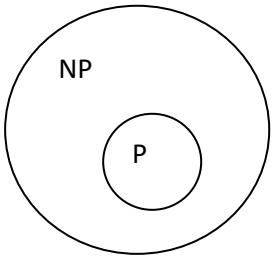
0	1	2	3	4
5	17	9	12	44

For example, in the above array key is 9 and the choice () function returns element index 2.

Based on the above discussion, we can define two classes of the problems.

P – problems which takes polynomial time and deterministic

NP – problems for which we can write non-deterministic algorithms, but it is taking polynomial time.



Try to show the similarities between them so, if one is solved then another will also be solved.

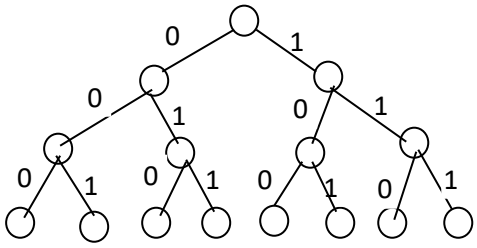
We take satisfiability problem as base problem.

$X_i = \{ x_1 , x_2 , x_3 \}$

$$CNF = \left(x_1 \bigvee x_2 \bigvee \ddot{x_3} \right) \bigwedge (x_2 \bigvee \ddot{x_1})$$

For which values of X_i the CNF (Conjunctive Normal Form) will be true. So, we have to total 2^3 possibilities for checking i.t

Let's represent these possibilities in the state space tree.



Now, we take 0/1 knapsack problem as follows:

P = { 1, 2, 3 } , W={ 3, 4, 5 }

X = { 0/1, 0/1, 0/1 }

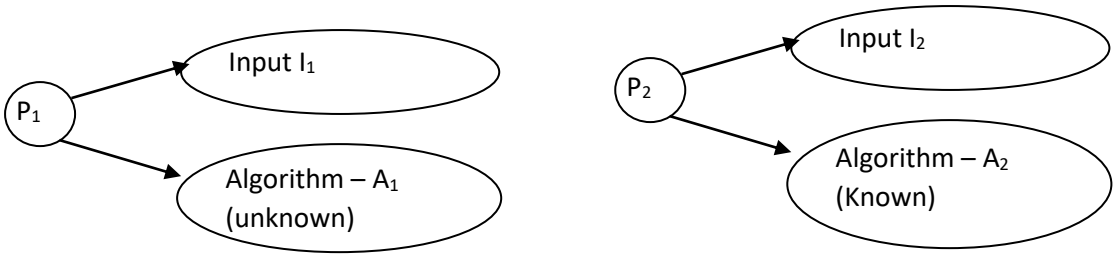
Similarly, we can represent 0/1 knapsack problem as satisfiability problem using state space tree.

A similar type of state space tree is constructed for all the problems which are taking exponential time means NP problems. If we can solve this state space tree using polynomial time then we can solve any problem in polynomial time.

NP-Hard and NP-Complete

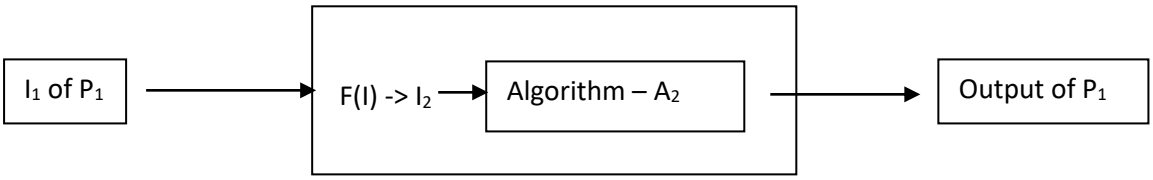
Reduction:

Consider we have two decision problems (whose answers are in YES or NO) – P_1 and P_2



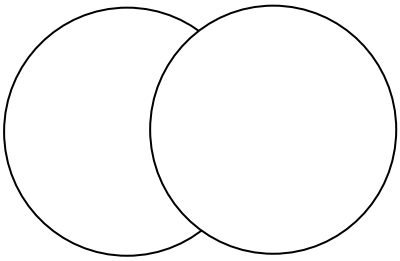
If problem P_1 can be solved with help of algorithm A_2 by converting Input I_2 into Input I_1 for finding the solution of problem.

Then we can say that P_1 is reducible to P_2.



For example,

Satisfiability problem is NP-Hard problem and if we reduce Satisfiability problem into 0/1 knapsack problem then we can say that 0/1 knapsack problem is also NP-Hard.



NP-Hard Problem:
A Problem X is NP-Hard if there is an NP-Complete problem Y, such that Y is reducible to X in polynomial time. NP-Hard problems are as hard as NP-Complete problems. NP-Hard Problem need not be in NP class.

NP-Complete Problem:

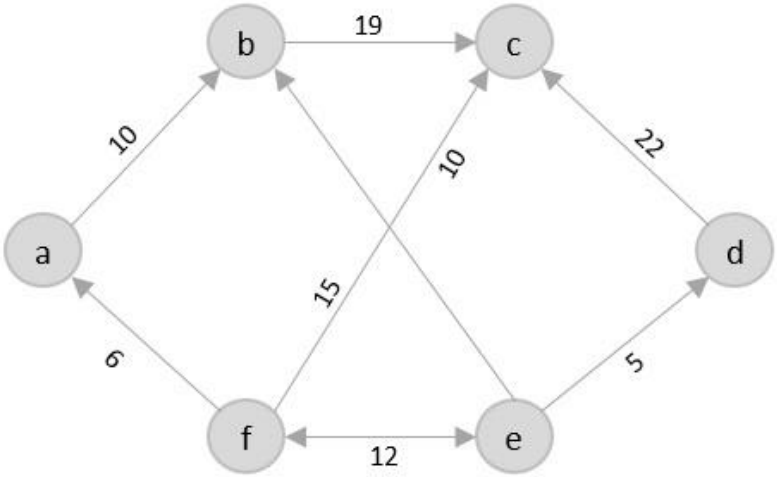
A problem X is NP-Complete if there is an NP problem Y, such that Y is reducible to X in polynomial time. NP-Complete problems are as hard as NP problems. A problem is NP-Complete if it is a part of both NP and NP-Hard Problem. A non-deterministic Turing machine can solve NP-Complete problem in polynomial time.

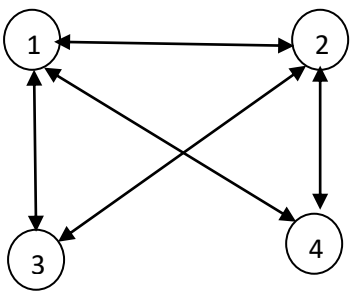
A problem is np-complete when it is both np and np hard combine together.

Travelling Salesman Problem (TSP):

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.

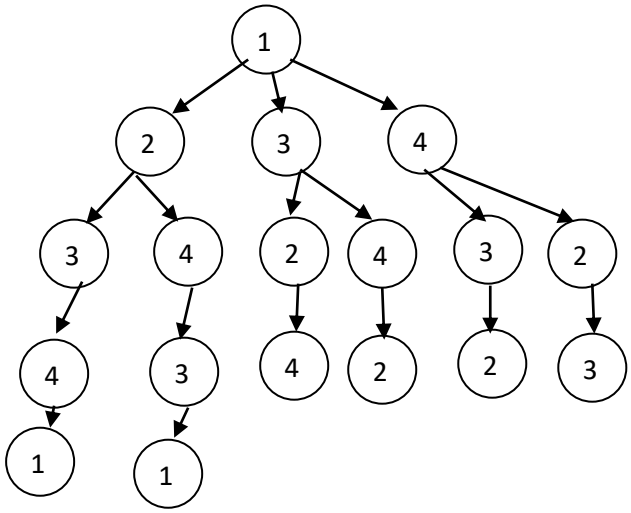




	1	**2**	**3**	**4**
1	0	10	15	20
2	5	0	25	10
3	15	30	0	5
4	15	10	20	0

Greedy approach:

1 -> 2 -> 4 -> 3 -> 1 – **55**



1 -> 3 -> 4 -> 2 - > 1 – **35**

So, possible number of tours are (n-1)!. – O(n!) which is approximated as $O(n^n)$

Hamiltonian Cycle: NP hard problem

The Hamiltonian cycle of undirected graph G <= V, E> is the cycle containing each vertex in V. -If graph contains a Hamiltonian cycle, it is called Hamiltonian graph otherwise it is non-Hamiltonian.

Traveling salesperson problem looks similar to this but in TSP, we are concerned with finding the tour with minimum cost.

Finding a Hamiltonian cycle in a graph is a well-known problem with many real-world applications, such as in network routing and scheduling.

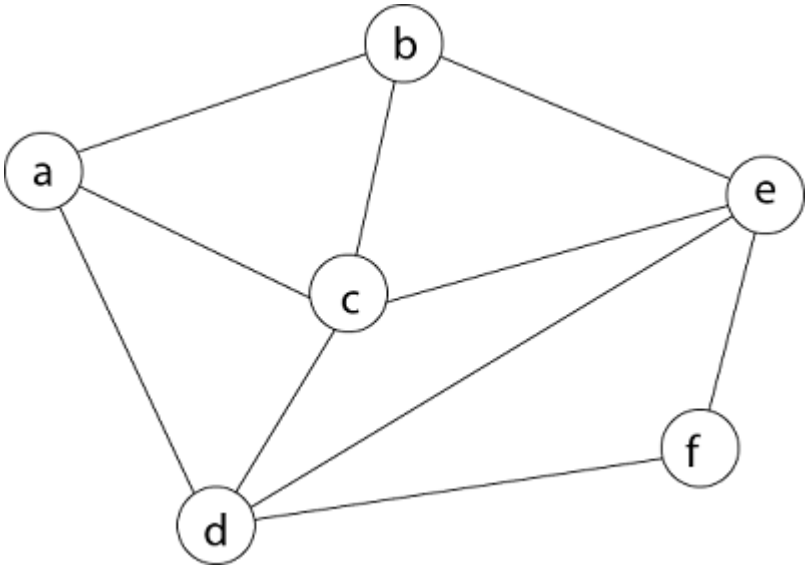
Hamiltonian Path in an undirected graph is a path that visits each vertex exactly once. A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian Path such that there is an edge (in the graph) from the last vertex to the first vertex of the Hamiltonian Path. Determine

whether a given graph contains Hamiltonian Cycle or not. If it contains, then prints the path. Following are the input and output of the required function.

Given a graph G = (V, E) we have to find the Hamiltonian Circuit using Backtracking approach. We start our search from any arbitrary vertex say 'a.' This vertex 'a' becomes the root of our implicit tree. The first element of our partial solution is the first intermediate vertex of the Hamiltonian Cycle that is to be constructed. The next adjacent vertex is selected by alphabetical order. If at any stage any arbitrary vertex makes a cycle with any vertex other than vertex 'a' then we say that **dead end** is reached. In this case, we backtrack one step, and again the search begins by selecting another vertex and backtrack the element from the partial; solution must be removed. The search using backtracking is successful if a Hamiltonian Cycle is obtained.

Time Complexity : O(N!), where N is number of vertices.

Example: Consider a graph G = (V, E) shown in fig. we have to find a Hamiltonian circuit using Backtracking method.



102045601 –Design and Analysis of Algorithms

Master Theorem for Dividing Function

Master Theorem

The Master Theorem applies to recurrences of the following form:

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

where $a \geq 1$ and $b > 1$ are constants and $f(n)$ is an asymptotically positive function.

There are 3 cases:

1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a} \log^k n)$ with[1] $k \geq 0$, then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
3. If $f(n) = \Omega(n^{\log_b a + \epsilon})$ with $\epsilon > 0$, and $f(n)$ satisfies the regularity condition, then $T(n) = \Theta(f(n))$.
   Regularity condition: $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

Examples and Solutions:

1. T (n) = 2T (n/2) + nlog n

   Ans: n $\log^2$ n (Case 2)
2. T (n) = 2T (n/4)+ $n^{0.51}$

   Ans: $n^{0.51}$ (Case 3)
3. T (n) = 0.5T (n/2)+ 1/n

   Ans: Does not apply (a < 1)
4. T (n) = $\sqrt{2}$T (n/2) + log n

   Ans: $\sqrt{n}$ (Case 1)
5. T (n) = 64T (n/8)− n^2 log n

   Ans: Does not apply (f(n) is not positive)
6. T (n) = 7T (n/3)+ n^2

   Ans: n^2 (Case 3)

7. $T(n) = T(n/2) + n(2 - \cos n)$

   Ans: Does not apply. We are in Case 3, but the regularity condition is violated. (Consider $n = 2\pi k$, where k is odd and arbitrarily large. For any such choice of n, you can show that $c \geq 3/2$, thereby violating the regularity condition.)

Master Theorem for Decreasing Function

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 $a < 1$, $T(n) = O(n^k)$

Examples and Solutions:

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

   Ans: n^3 (Case 1)

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

   Ans: $n * 2^n$ (Case 2)

3. $T(n) = n^4$

   Ans: n^4 (Case 3)

4. $T(n) = 3T(n-1)$

   Ans: 3^n

5. $T(n) = T(n-1) + n(n-1)$

   Ans: n^3

6. $T(n) = 2T(n-1) – 1$

   Ans: Sol: This recurrence can't be solved using above method as f(n) is not a positive function

   This can be solved with forward **substitution** method.