DESIGN AND ANALYSIS OF ALGORITHMS

MCA

&

MSC COMPUTERS

2ND SEMESTER

PREPARED BY,

I VENKATESH, MCA

LECTURER IN COMPTER SCIENCE.

7989815990.

Semester	Course Code	Course Title	Hours/Week	Hours	Credits
II	MCA20202 & MSC20201	DESIGN & ANALYSIS OF ALGORITHMS	4	60	4

UNIT-1

Introduction- Algorithm, Pseudo Code for expressing algorithms, Performance Analysis- Space Complexity, Time Complexity, Asymptotic Notations. Disjoint Sets- disjoint set operations, union and find algorithms.

UNIT-2

Divide and Conquer- General method, Applications-Binary Search, Merge Sort and Quick Sort.
Greedy Method- General Method, Applications- Job Sequencing with deadlines, 0/1 Knapsack Problem, Minimum cost spanning trees

UNIT-3

Dynamic Programming- General Method, Applications-Optimal Binary Search Tree, 0/1 Knapsack Problem, All pair shortest path problem, Travelling Salesman Problem.
Backtracking:N-queen problem, Graph Coloring Problem and Hamiltonian Cycle.

UNIT-4

Branch and Bound: Travelling Salesman Problem, 0/1 Knapsack Problem, FIFO Branch and Bound Search and LIFO Branch and Bound.
Complexity Theory: Introduction to NP Hard and NP Complete Problems.

I.VENKATESH, MCA
LECTURER IN COMPUTER SCIENCE,
MOBLE:-7989815990.
RAO'S INSTITUTE OF COMPUTER SCIENCE,
NELLORE-524001.

UNIT-1

DAA Algorithm

The word algorithm has been derived from the Persian author's name, Abu Ja 'far Mohammed ibn Musa al Khowarizmi (c. 825 A.D.), who has written a textbook on Mathematics. The word is taken based on providing a special significance in computer science. The algorithm is understood as a method that can be utilized by the computer as when required to provide solutions to a particular problem.

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

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

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

Characteristics of Algorithms

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

Advantages of an Algorithm

- **Effective Communication:** Since it is written in a natural language like English, it becomes easy to understand the step-by-step delineation of a solution to any particular problem.

- **Easy Debugging:** A well-designed algorithm facilitates easy debugging to detect the logical errors that occurred inside the program.
- **Easy and Efficient Coding:** An algorithm is nothing but a blueprint of a program that helps develop a program.
- **Independent of Programming Language:** Since it is a language-independent, it can be easily coded by incorporating any high-level language.

Disadvantages of an Algorithm

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

Pseudo code for expressing algorithms

Pseudo code

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

Advantages of Pseudo code

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

Disadvantages of Pseudo code

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

Difference between Algorithm and Pseudo code:

An **algorithm** is a well defined sequence of instructions that provide a solution to the given problem.
A **pseudo code** is a method which is used to represent an algorithm.

An **algorithm** has some specific characteristics that describe the process.

A **pseudo code** on the other hand is not restricted to something. It's only objective is to represent an algorithm in a realistic manner.

An **algorithm** is written in plain English or general language.
A **pseudo code** is written with a hint of programming concepts such as control structures.

To understand the difference between an algorithm and pseudo code let's have a look into the following example:
Consider an example of calculating the area of a circle.

Algorithm:

1. Start.
2. Read r: radius value as the input given by the user.
3. Calculate the Area: 3.14 * r * r.
4. Display the Area.
5. End.

Pseudo code:

```
AreaofCircle()
{
BEGIN
Read: Number radius, Area;
Input r;
Area = 3.14 * r * r;
Output Area;
END
}
```

How to write a pseudo code?

As mentioned, a pseudo code is a method used to implement an algorithm. However there are a few points to be noted while writing a pseudo code. They are:

1. First, arrange the tasks in a sequence so that the pseudo code can be written by following the same sequence. This will make the process more clear and simple.

2. A pseudo code is something that can be understood by any basic programmer. However, in case of complex problems it could be difficult to understand the main goal of the problem if it is not specified in the pseudo code. So, do include a statement that established the main goal.

3. Follow the indentation and whitespaces as you would do while writing an actual program. Indentation helps greatly with readability.
   Example 1
   ```
   if  "1" print response
   "I am case 1"
   ```

Example 2

```
if  "2"
print response
"I am case 2"
```

Here, both the examples have the same meaning. However, the 2nd example is more clear and understandable compared to the 1st. This is the magic of Indentation and white spaces.

4. Following a naming convention is important. Sometimes, naming in the wrong way can lead to confusion. So, naming must be done in a simple and distinct way.

5. Sentence casings will always help you differentiate between constants, variables, methods, and others. This will avoid any confusion that might be present in the algorithm.
   For methods use CamelCase
   For constants use UPPERCASE
   For variables use lowercase

6. A pseudo code is supposed to explain the code in detail. Do not keep the pseudo code abstract.

7. Use of control structures such as 'for', 'while', 'if-then' and 'cases' makes it easy while you develop a program using the pseudo code as reference.

8. A pseudo code must be perfect in order for the code to be perfect. So do check if the pseudo code consists of any infinite loops or any gaps.

9. A pseudo code must be must enough to be understood by a layman so do not write it in a programmatic way.

Do's and Don'ts while writing a Pseudo code:

Do's:

- Make use of control structures.
- Naming conventions must be followed properly.
- Use indentation and white spaces wherever required as these are the key points for a pseudo code.
- Keep the pseudo code simple and concise.

Don'ts:

- Do not generalize the pseudo code.
- A pseudo code mustn't be abstract.

PERFORMANCE ANALYSIS

If we want to go from city "A" to city "B", there can be many ways of doing this. We can go by flight, by bus, by train and also by bicycle. Depending on the availability and convenience, we choose the one which suits us. Similarly, in computer science, there are multiple algorithms to solve a problem. When we have more than one algorithm to solve a problem, we need to select the best one. Performance analysis helps us to select the best algorithm from multiple algorithms to solve a problem.

When there are multiple alternative algorithms to solve a problem, we analyze them and pick the one which is best suitable for our requirements. The formal definition is as follows...

"Performance of an algorithm is a process of making evaluative judgment about algorithms."

It can also be defined as follows...

"Performance of an algorithm means predicting the resources which are required to an algorithm to perform its task."

That means when we have multiple algorithms to solve a problem, we need to select a suitable algorithm to solve that problem.

We compare algorithms with each other which are solving the same problem, to select the best algorithm. To compare algorithms, we use a set of parameters or set of elements like memory required by that algorithm, the execution speed of that algorithm, easy to understand, easy to implement, etc.,

Generally, the performance of an algorithm depends on the following elements...

1. Whether that algorithm is providing the exact solution for the problem?
2. Whether it is easy to understand?
3. Whether it is easy to implement?
4. How much space (memory) it requires to solve the problem?
5. How much time it takes to solve the problem? Etc.,

When we want to analyze an algorithm, we consider only the space and time required by that particular algorithm and we ignore all the remaining elements. Based on this information, performance analysis of an algorithm can also be defined as follows...

"Performance analysis of an algorithm is the process of calculating space and time required by that algorithm."

Performance analysis of an algorithm is performed by using the following measures...

1. Space required to complete the task of that algorithm (**Space Complexity**). It includes program space and data space
2. Time required to complete the task of that algorithm (**Time Complexity**)

Space complexity:

Space complexity is a combination of auxiliary space and input space. Where auxiliary space is the extra space or buffer space that will be used by an algorithm during execution. Also, we know that space complexity is all about memory. Below are a few points on how the memory is used during execution.

- Instruction space: the compiled version of instructions are stored in memory. This memory is called instruction space.
- Environmental stack: We use environmental stacks in cases where a function is called inside another function.

For example, a function cat() is called inside the function animals().
Then the variables of function animals() will be stored temporarily in a system stack when function cat() is being executed inside the function animals().

- Data Space: The variables and constants that we use in the algorithm will also require some space which is referred to as data space.

In most of the cases, we neglect environmental stack and instruction space. Whereas, data space is always considered.

How to calculate space complexity?

To calculate the space complexity we need to have an idea about the value of the memory of each data type. This value will vary from one operating system to another. However, the method used to calculate the space complexity remains the same.
Let’s have a look into a few examples to understand how to calculate the space complexity.

Example 1:

```
{
   int a,b,c,sum;
   sum = a + b + c;
   return(sum);
}
```

Here, in this example, we have 4 variables which are a, b, c, and sum. All these variables are of type int hence, each of them require 4 bytes of memory.
Total requirement of memory would be : ((4*4)+4) => 20 bytes.
Here, we have considered 4 additional bytes of memory which is for the return value. Also, for this example as the space requirement is fixed it is known as constant space complexity.

Example 2:

```
// n is the size of array arr[]
int sum(int arr[], int n)
{
int k, i = 0;
for(k = 0; k < n; k++)
{
   i  = i + arr[k];
}
return(i);
}
```

In this example, we have 3 variables i, k, n each of which will be assigned 4 bytes of memory. Also, for the return value 4 bytes of memory will be assigned. Here, we also have an array of size n, hence the memory required for that would be (4*n).

Hence the total memory requirement would be: (4n+12)

This value will vary based on the value of n. As the value of n increases the memory would also increase linearly. Therefore it is known as Linear Space Complexity.

Depending on the complexity of the algorithm, the space complexity can range unto quadratic or complex too. We should always try to keep the space complexity as minimum as possible.

Time complexity:

Time complexity is most commonly evaluated by considering the number of elementary steps required to complete the execution of an algorithm. Also, many times we make use of the big O notation which is an asymptotic notation while representing the time complexity of an algorithm.

Constant time complexity:

Example:

Statement – printf("hello venky");

In the above example, as it is a single statement the complexity of it would be constant. Its complexity wouldn't change as it is not dependent on any variable.

Linear time complexity:

Example:

```
for(i=0; i < N; i++)
{
   statement;
}
```

Here, the complexity would vary as the number of times the loop iterates would be based on the value of n. Hence, the time complexity would be linear.

Quadratic time complexity:

Example:

```
for(i=0; i < N; i++)
{
   for(j=0; j < N;j++)
   {
   statement;
   }
}
```

Here, in this example we have a loop inside another loop. In such a case, the complexity would be proportional to n square. Hence, it would be of type quadratic and therefore the complexity is quadratic complexity.

Logarithmic time complexity:

Example:

```
while(low <= high)
{
   mid = (low + high) / 2;
   if (target < list[mid])
      high = mid - 1;
   else if (target > list[mid])
      low = mid + 1;
   else break;
}
```

In this example, we are working on dividing a set of numbers into two parts. Which means that the complexity would be proportional to the number of times n can be divided by 2, which is a logarithmic value. Hence we have a logarithmic time complexity.

Asymptotic Notations

Execution time of an algorithm depends on the instruction set, processor speed, disk I/O speed, etc. Hence, we estimate the efficiency of an algorithm asymptotically.

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

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

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

Big Oh ‘O’: Asymptotic Upper Bound

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

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

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

Example

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

Considering $g(n)=n^3$,

$f(n) \leq 5.g(n)$ for all the values of $n>2$

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

Big Ω: Asymptotic Lower Bound

We say that $f(n)=\Omega(g(n))$ when there exists constant **c** that **$f(n)\geq c.g(n))$** for all sufficiently large value of **n**. Here **n** is a positive integer. It means function ***g*** is a lower bound for function ***f***; after a certain value of ***n, f*** will never go below ***g***.

Example

Let us consider a given function, $f(n)=4.n^3+10.n^2+5.n+1$
Considering $g(n)=n^3$, $f(n)\geq 4.g(n)$for all the values of $n>0$.
Hence, the complexity of ***f(n)*** can be represented as $\Omega(g(n))$, i.e. $\Omega(n^3)$

Big θ: Asymptotic Tight Bound

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

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

Example

Let us consider a given function, $f(n)= 4.n^3+10.n^2+5.n+1$
Considering $g(n)=n^3$, $4.g(n)\leq f(n)\leq 5.g(n)$ for all the large values of **n**.
Hence, the complexity of ***f(n)*** can be represented as $\theta(g(n))$, i.e. $\theta(n^3)$.

Small oh - Notation

The asymptotic upper bound provided by **Big O-notation** may or may not be asymptotically tight. The bound $2.n^2=O(n^2)$ is asymptotically tight, but the bound $2.n=O(n^2)$ is not.

We use **o-notation** to denote an upper bound that is not asymptotically tight.

We formally define ***o(g(n))*** (little-oh of g of n) as the set ***f(n) = o(g(n))*** for any positive constant c>0c>0 and there exists a value n0>0n0>0, such that $0\leq f(n)\leq c.g(n)$.

Intuitively, in the **o-notation**, the function ***f(n)*** becomes insignificant relative to ***g(n)*** as **n** approaches infinity; that is,

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

Example

Let us consider the same function, $f(n)=4.n^3+10.n^2+5.n+1$
Considering $g(n)=n^4$,

$$\lim_{n\to\infty}\left(\frac{4.n^3+10.n^2+5.n+1}{n^4}\right)=0$$

Hence, the complexity of ***f(n)*** can be represented as $o(g(n))$, i.e. $o(n^4)$.

Small omega (ω) – Notation

We use ω-notation to denote a lower bound that is not asymptotically tight.
Formally, however, we define $\omega(g(n))$ (little-omega of g of n) as the set $f(n) = \omega(g(n))$ for any positive constant $C > 0$ and there exists a value $n0>0$, such that $0\leq c.g(n)<f(n)$.

For example,

$n^2/2=\omega(n)$, but $n^2/2\neq\omega(n^2)$. The relation $f(n)=\omega(g(n))$ implies that the following limit exists

$$\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right)=\infty$$

That is, ***f(n)*** becomes arbitrarily large relative to ***g(n)*** as **n** approaches infinity.

Example

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

$$\lim_{n\to\infty}\left(\frac{4.n^3+10.n^2+5.n+1}{n^2}\right)=\infty$$

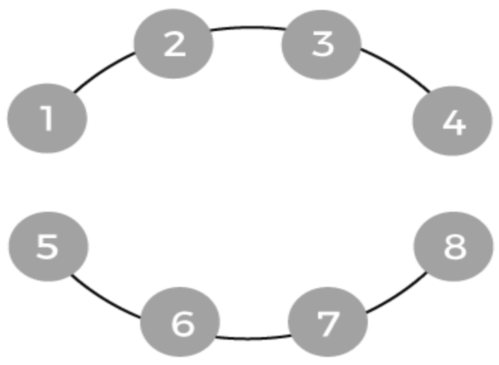
Hence, the complexity of f(n) can be represented as o(g(n)), i.e. $\omega(n^2)$.

Disjoint sets

Disjoint set

The disjoint set data structure is also known as union-find data structure and merge-find set. It is a data structure that contains a collection of disjoint or non-overlapping sets. The disjoint set means that when the set is partitioned into the disjoint subsets. The various operations can be performed on the disjoint subsets. In this case, we can add new sets, we can merge the sets, and we can also find the representative member of a set. It also allows to find out whether the two elements are in the same set or not efficiently.

The disjoint set can be defined as the subsets where there is no common element between the two sets. Let's understand the disjoint sets through an example.



s1 = {1, 2, 3, 4}

s2 = {5, 6, 7, 8}

We have two subsets named s1 and s2. The s1 subset contains the elements 1, 2, 3, 4, while s2 contains the elements 5, 6, 7, 8. Since there is no common element between these two sets, we will not get anything if we consider the intersection between these two sets. This is also known as a disjoint set where no elements are common.

Disjoint set operations

We can perform only two operations on disjoint sets, i.e.,
1. Find
2. Union.

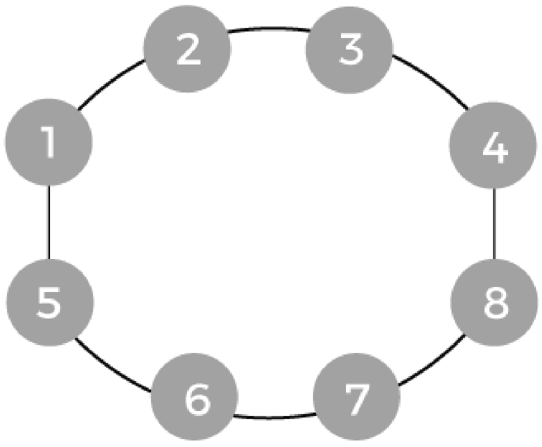
find operation: we have to check that the element is present in which set. There are two sets named s1 and s2 shown below:

s1 = {1, 2, 3, 4}

s2 = {5, 6, 7, 8}

Union operation:

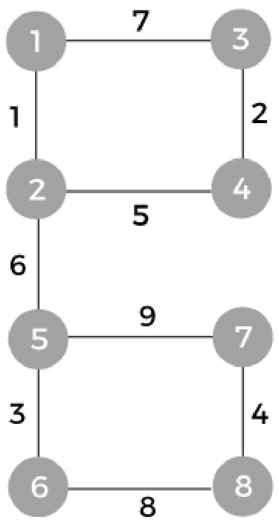
Suppose we want to perform the union operation on these two sets. First, we have to check whether the elements on which we are performing the union operation belong to different or same sets. If they belong to the different sets, then we can perform the union operation; otherwise, not. For example, we want to perform the union operation between 4 and 8. Since 4 and 8 belong to different sets, so we apply the union operation. Once the union operation is performed, the edge will be added between the 4 and 8 shown as below:
When the union operation is applied, the set would be represented as:



s1Us2 = {1, 2, 3, 4, 5, 6, 7, 8}
Suppose we add one more edge between 1 and 5. Now the final set can be represented as:
s3 = {1, 2, 3, 4, 5, 6, 7, 8}
If we consider any element from the above set, then all the elements belong to the same set; it means that the cycle exists in a graph.

How can we detect a cycle in a graph? (Using disjoint sets)

Consider the below example to detect a cycle with the help of using disjoint sets.



U = {1, 2, 3, 4, 5, 6, 7, 8}
Each vertex is labeled with some weight. There is a universal set with 8 vertices. We will consider each edge one by one and form the sets.

First, we consider vertices 1 and 2. Both belong to the universal set; we perform the union operation between elements 1 and 2. We will add the elements 1 and 2 in a set s1 and remove these two elements from the universal set shown below:

s1 = {1, 2}

The vertices that we consider now are 3 and 4. Both the vertices belong to the universal set; we perform the union operation between elements 3 and 4. We will form the set s3 having elements 3 and 4 and remove the elements from the universal set shown as below:

s2 = {3, 4}

The vertices that we consider now are 5 and 6. Both the vertices belong to the universal set, so we perform the union operation between elements 5 and 6. We will form the set s3 having elements 5 and 6 and will remove these elements from the universal set shown as below:

s3 = {5, 6}

The vertices that we consider now are 7 and 8. Both the vertices belong to the universal set, so we perform the union operation between elements 7 and 8. We will form the set s4 having elements 7 and 8 and will remove these elements from the universal set shown as below:

s4 = {7, 8}

The next edge that we take is (2, 4). The vertex 2 is in set 1, and vertex 4 is in set 2, so both the vertices are in different sets. When we apply the union operation, then it will form the new set shown as below:

s5 = {1, 2, 3, 4}

The next edge that we consider is (2, 5). The vertex 2 is in set 5, and the vertex 5 is in set s3, so both the vertices are in different sets. When we apply the union operation, then it will form the new set shown as below:

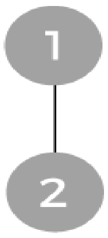
s6 = {1, 2, 3, 4, 5, 6}

Now, two sets are left which are given below:
s4 = {7, 8}
s6 = {1, 2, 3, 4, 5, 6}
The next edge is (1, 3). Since both the vertices, i.e.,1 and 3 belong to the same set, so it forms a cycle. We will not consider this vertex.
The next edge is (6, 8). Since both vertices 6 and 8 belong to the different vertices s4 and s6, we will perform the union operation. The union operation will form the new set shown as below:
s7 = {1, 2, 3, 4, 5, 6, 7, 8}
The last edge is left, which is (5, 7). Since both the vertices belong to the same set named s7, a cycle is formed.

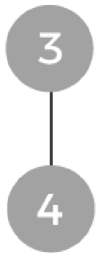
Representation of disjoint sets

1. Graphical representation:-

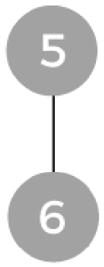
We have a universal set which is given below:
U = {1, 2, 3, 4, 5, 6, 7, 8}
We will consider each edge one by one to represent graphically.
First, we consider the vertices 1 and 2, i.e., (1, 2) and represent them through graphically shown as below:



In the above figure, vertex 1 is the parent of vertex 2.
Now we consider the vertices 3 and 4, i.e., (3, 4) and represent them graphically shown as below:

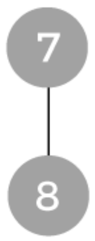


In the above figure, vertex 3 is the parent of vertex 4.
Consider the vertices 5 and 6, i.e., (5, 6) and represent them graphically shown as below:



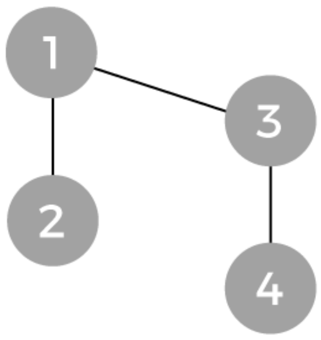
In the above figure, vertex 5 is the parent of vertex 6.

Now, we consider the vertices 7 and 8, i.e., (7, 8) and represent them through graphically shown as below:



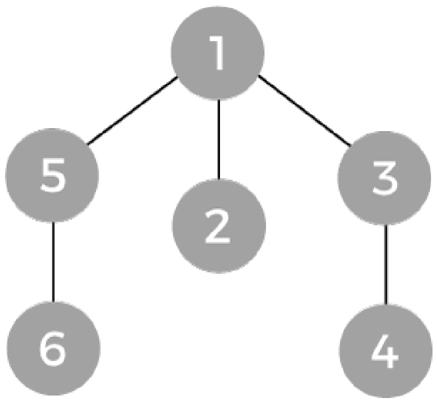
In the above figure, vertex 7 is the parent of vertex 8.

Now we consider the edge (2, 4). Since 2 and 4 belong to different sets, so we need to perform the union operation. In the above case, we observe that 1 is the parent of vertex 2 whereas vertex 3 is the parent of vertex 4. When we perform the union operation on the two sets, i.e., s1 and s2, then 1 vertex would be the parent of vertex 3 shown as below:



The next edge is (2, 5) having weight 6. Since 2 and 5 are in two different sets so we will perform the union operation. We make vertex 5 as a child of the vertex 1 shown as below:

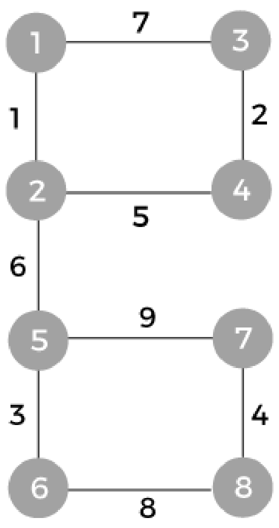
We have chosen vertex 5 as a child of vertex 1 because the vertex of the graph having parent 1 is more than the graph having parent 5.



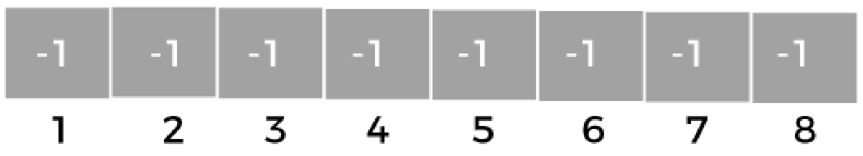
The next edge is (1, 3) having weight 7. Both vertices 1 and 3 are in the same set, so there is no need to perform any union operation. Since both the vertices belong to the same set; therefore, there is a cycle. We have detected a cycle, so we will consider the edges further.

2. Array representation:

The below graph contains the 8 vertices.

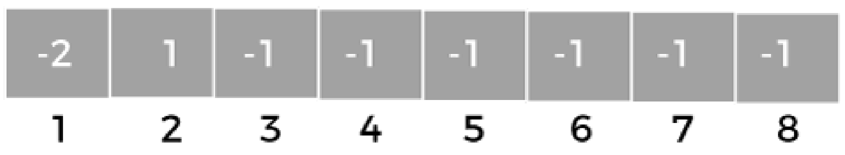


So, we represent all these 8 vertices in a single array. Here, indices represent the 8 vertices. Each index contains a -1 value. The -1 value means the vertex is the parent of itself.

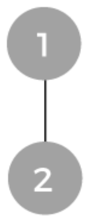


-1	-1	-1	-1	-1	-1	-1	-1
1	2	3	4	5	6	7	8

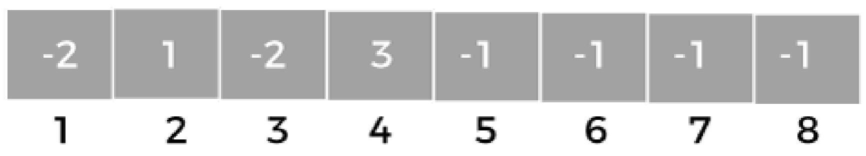
First, we consider the edge (1, 2). When we find 1 in an array, we observe that 1 is the parent of itself. Similarly, vertex 2 is the parent of itself, so we make vertex 2 as the child of vertex 1. We add 1 at the index 2 as 2 is the child of 1. We add -2 at the index 1 where '-' sign that the vertex 1 is the parent of itself and 2 represents the number of vertices in a set.



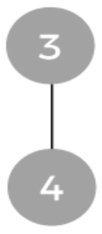
-2	1	-1	-1	-1	-1	-1	-1
1	2	3	4	5	6	7	8



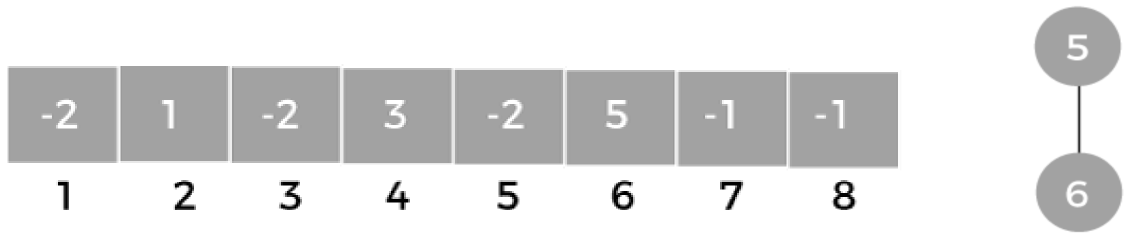
The next edge is (3, 4). When we find 3 and 4 in array; we observe that both the vertices are parent of itself. We make vertex 4 as the child of the vertex 3 so we add 3 at the index 4 in an array. We add -2 at the index 3 shown as below:



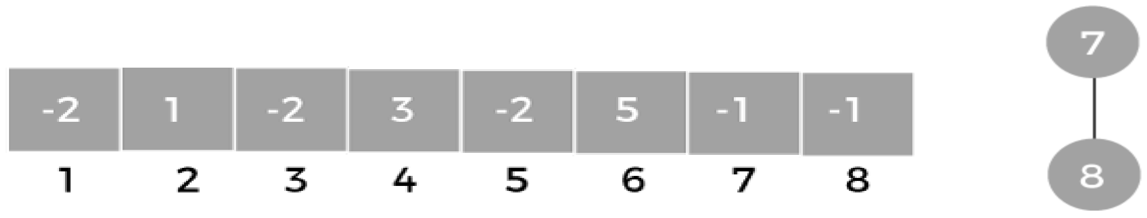
-2	1	-2	3	-1	-1	-1	-1
1	2	3	4	5	6	7	8



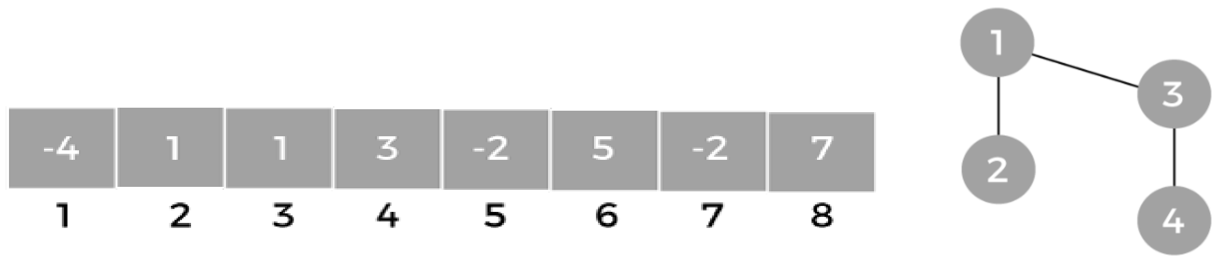
The next edge is (5, 6). When we find 5 and 6 in an array; we observe that both the vertices are parent of itself. We make 6 as the child of the vertex 5 so we add 5 at the index 6 in an array. We add -2 at the index 5 shown as below:



The next edge is (7, 8). Since both the vertices are parent of itself, so we make vertex 8 as the child of the vertex 7. We add 7 at the index 8 and -2 at the index 7 in an array shown as below:

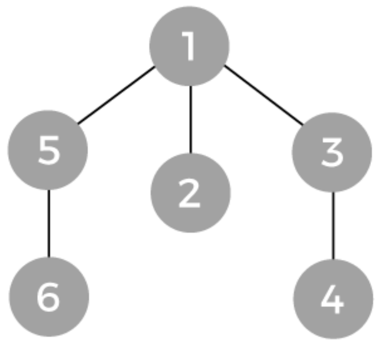


The next edge is (2, 4). The parent of the vertex 2 is 1 and the parent of the vertex is 3. Since both the vertices have different parent, so we make the vertex 3 as the child of vertex 1. We add 1 at the index 3. We add -4 at the index 1 as it contains 4 vertices.
Graphically, it can be represented as



The next edge is (2,5). When we find vertex 2 in an array, we observe that 1 is the parent of the vertex 2 and the vertex 1 is the parent of itself. When we find 5 in an array, we find -2 value which means vertex 5 is the parent of itself. Now we have to decide whether the vertex 1 or vertex 5 would become a parent. Since the weight of vertex 1, i.e., -4 is greater than the vertex of 5, i.e., -2, so when we apply the union operation then the vertex 5 would become a child of the vertex 1 shown as below:

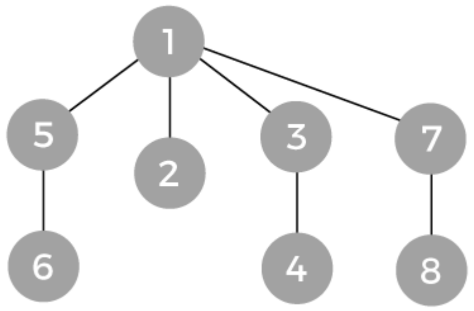
-6	1	1	3	1	5	-2	7
1	2	3	4	5	6	7	8



The next edge is (1,3). When we find vertex 1 in an array, we observe that 1 is the parent of itself. When we find 3 in an array, we observe that 1 is the parent of vertex 3. Therefore, the parent of both the vertices are same; so, we can say that there is a formation of cycle if we include the edge (1,3).

The next edge is (6,8). When we find vertex 6 in an array, we observe that vertex 5 is the parent of vertex 6 and vertex 1 is the parent of vertex 5. When we find 8 in an array, we observe that vertex 7 is the parent of the vertex 8 and 7 is the parent of itself. Since the weight of vertex 1, i.e., -6 is greater than the vertex 7, i.e., -2, so we make the vertex 7 as the child of the vertex and can be represented graphically as shown as below:
We add 1 at the index 7 because 7 becomes a child of the vertex 1. We add -8 at the index 1 as the weight of the graph now becomes 8.

-8	1	1	3	1	5	1	7
1	2	3	4	5	6	7	8



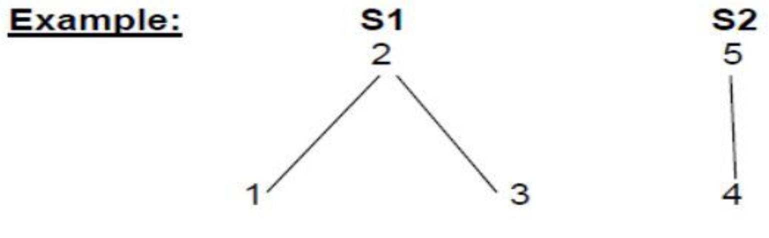
The last edge to be included is (5, 7). When we find vertex 5 in an array, we observe that vertex 1 is the parent of the vertex 5. Similarly, when we find vertex 7 in an array, we observe that vertex 1 is the parent of vertex 7. Therefore, we can say that the parent of both the vertices is same, i.e., 1. It means that the inclusion (5,7) edge would form a cycle.

Find and Union algorithms:

1. Simple Find: The find operation can be implemented by recursively traversing through the parent array until we reach a node that is the parent of itself.

Example:

Consider that we are given an element x and we have to find a set containing it.



By applying find(3), it returns the set which contains element 3 i.e., S1.
By applying find(5), it returns the set which contains element 5 i.e., S2.

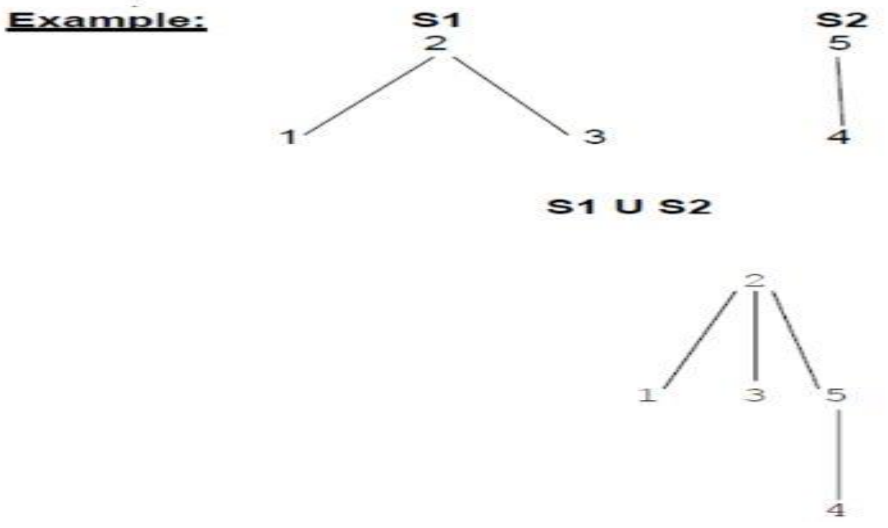
Following is the algorithm for find operation:

```
Algorithm SimpleFind(i)
{
while( P[i] >0) do
{
i := P[i];
}
Return i;
}
```

2. Simple Union: It inputs two elements and finds the representatives of their sets by using the find operation. It finally puts one of the trees (representing the set) under the root node of the other tree thereby effectively merging the trees and the sets.

Example:

Let us consider that S1 and S2 are two disjoint sets and their union is the set of all elements "x" such that "x" is either is S1 or S2. AS we require disjoint sets, S1 U S2 replaces S1 and S2 which no longer exist now. We can achieve the union by simply making one of the trees a subtree of the other.



Following is the algorithm for union operation

```
Algorithm SimpleUnion(i,j)
{
P[i] :=j;
}
```

3. Weighted Union:

Weighting rule for Union (i, j):

If the number of nodes in the tree with root 'i' is less than the tree with root 'j', then make 'j' the parent of 'i'; otherwise make 'i' the parent of 'j'.

Algorithm for weightedUnion(i, j)

```
Algorithm WeightedUnion(i,j)
//Union sets with roots i and j, i≠j
```

```
// The weighting rule, p[i]= -count[i] and p[j]= -count[j].
{
temp := p[i]+p[j];
if (p[i]>p[j]) then
{ // i has fewer nodes.
P[i]:=j;
P[j]:=temp;
}
else
{
 // j has fewer or equal
nodes. P[j] := i;
P[i] := temp;
}
}
```

For implementing the weighting rule, we need to know how many nodes there are in every tree. For this we maintain a count field in the root of every tree.
i root node
count[i] number of nodes in the tree.

Time complexity of Weighted Union Algorithm:
Time required for this above algorithm is O(1) + time for remaining unchanged is determined by using Lemma.

Lemma:-
Let T be a tree with m nodes created as a result of a sequence of unions each performed using Weighted Union. The height of T is no greater than $|\log_2 m|+1$.

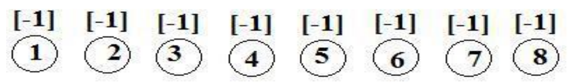
4. Collapsing Find:
Collapsing rule: If 'j' is a node on the path from 'i' to its root and p[i]≠root[i], then set p[j] to root[i].

Algorithm for Collapsing find:-

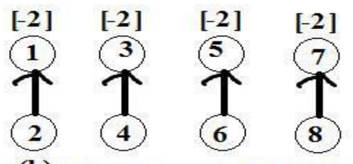
```
Algorithm CollapsingFind(i)
//Find the root of the tree containing element i.
//collapsing rule to collapse all nodes form i to the root.
{
r;=i;
while(p[r]>0) do r := p[r]; //Find the root.
While(i ≠ r) do // Collapse nodes from i to root r.
{
s:=p[i];
p[i]:=r;
i:=s;
}
return r;
}
```

Collapsing find algorithm is used to perform find operation on the tree created by WeightedUnion.

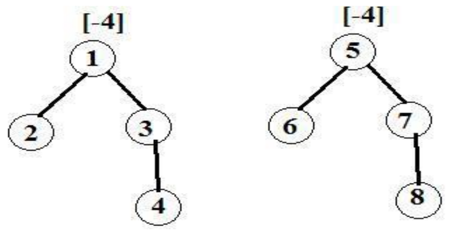
For example: Tree created by using WeightedUnion



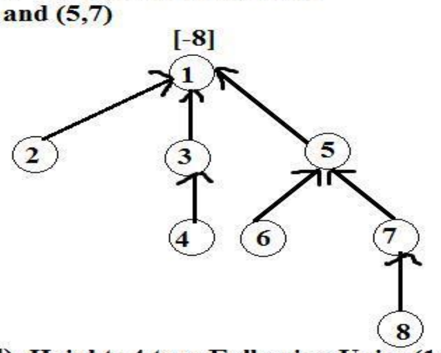
(a) initial height -1 tree



(b) Height -2 trees follwoing Union(1,2),(3,4),(5,6),(7,8)



(c) Height -3 trees following Union (1,3) and (5,7)



(d) Height -4 tree Following Union(1,5)

.

Now process the following eight finds: Find(8), Find(8),..........Find(8)

If **SimpleFind** is used, each Find(8) requires going up three parent link fields for a total of 24 moves to process all eight finds. i.e 3X8=24.

When **CollapsingFind** is uised the first Find(8) requires going up three links and then resetting two links. i.e 3X2(1st find) +1X7(remaining 7 finds)=13

Total 13 movies requies for process all eight finds.

UNIT-2
Divide and Conquer

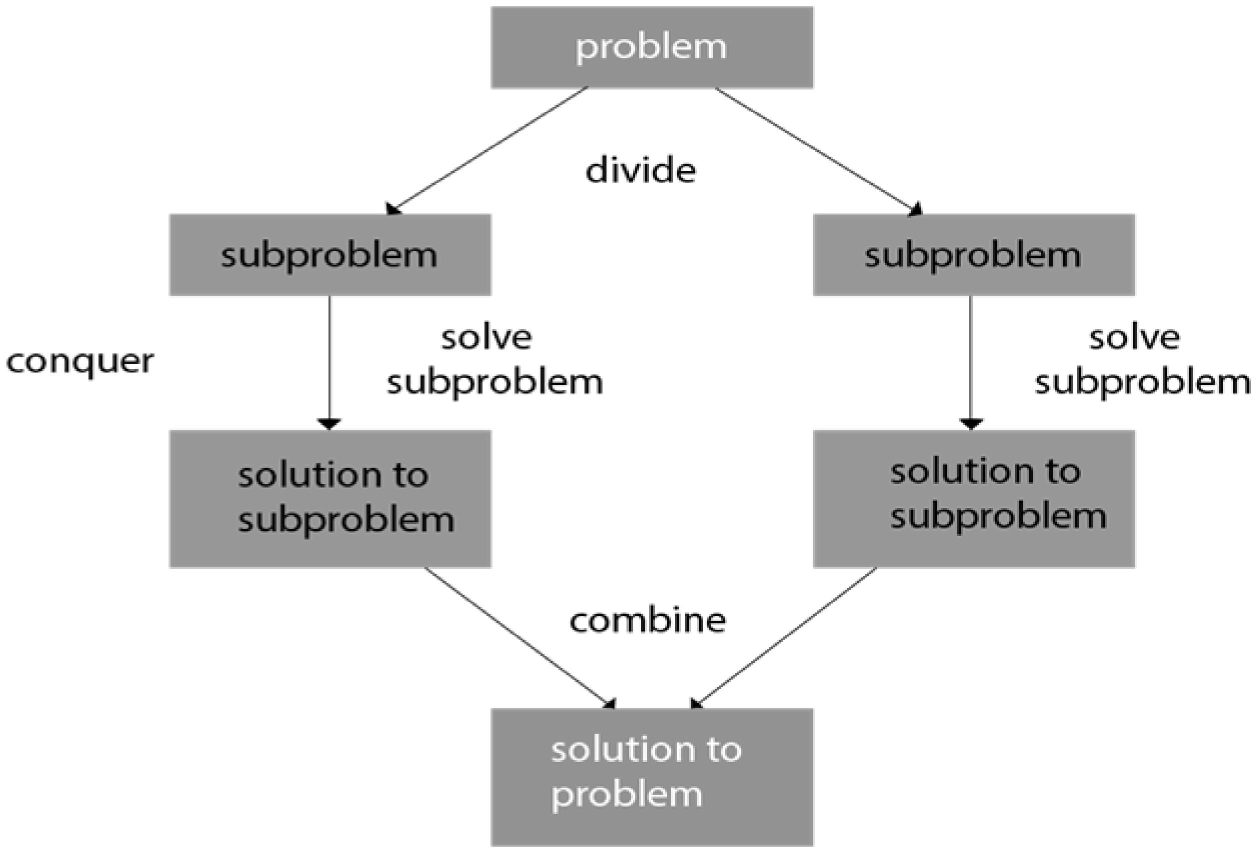
DIVIDE AND CONQUER:
General Method

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

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

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

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



Pseudo code Representation of Divide and conquer rule for problem "P"

```
Algorithm DAndC(P)
{
if small(P) then return S(P)
else
{
divide P into smaller instances P1,P2,P3...Pk;
apply DAndC to each of these subprograms; // means DAndC(P1), DAndC(P2).....DAndC(Pk)
return combine(DAndC(P1), DAndC(P2)..... DAndC(Pk));
}
}
```

Here small(P) Boolean value function. If it is true, then the function S is invoked

Time Complexity of DAndC algorithm:

$T(n) = T(1)$, if $n=1$

$aT(n/b)+f(n)$, if $n>1$

here a,b are constants.

This is called the general **divide and-conquer recurrence**.

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

- Maximum and Minimum Problem
- Binary Search

- Sorting (merge sort, quick sort)
- Tower of Hanoi.

Advantages of Divide and Conquer

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

Disadvantages of Divide and Conquer

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

Applications of divide and conquer

The Following algorithms are based on the concept of the Divide and Conquer Technique:
1. Binary Search
2. Quick Sort
3. Merge Sort

Binary search

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

Binary Search technique:-

1. In Binary Search technique, we search an element in a sorted array by recursively dividing the interval in half.
2. Firstly, we take the whole array as an interval.

3. If the Pivot Element (the item to be searched) is less than the item in the middle of the interval, We discard the second half of the list and recursively repeat the process for the first half of the list by calculating the new middle and last element.
4. If the Pivot Element (the item to be searched) is greater than the item in the middle of the interval, we discard the first half of the list and work recursively on the second half by calculating the new beginning and middle element.
5. Repeatedly, check until the value is found or interval is empty.

Analysis of Binary Search Algorithm

1. **Input:** an array A of size n, already sorted in the ascending or descending order.
2. **Output:** analyze to search an element item in the sorted array of size n.
3. **Logic:** Let T (n) = number of comparisons of an item with n elements in a sorted array.
   - Set BEG = 1 and END = n
   - Find mid = $\text{int}\left(\frac{beg + end}{2}\right)$
   - Compare the search item with the mid item.

Case 1: item = A[mid], then LOC = mid, but it the best case and T (n) = 1

Case 2: item ≠A [mid], then we will split the array into two equal parts of size $\frac{n}{2}$.
And again find the midpoint of the half-sorted array and compare with search element.
Repeat the same process until a search element is found.

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

{Time to compare the search element with mid element, then with half of the selected half part of array}

$T\left(\frac{n}{2}\right) = T\left(\frac{n}{2^2}\right) + 1$, putting $\frac{n}{2}$ in place of n.

Then we get: T (n) = $(T\left(\frac{n}{2^2}\right) + 1) + 1$...........By putting T $\frac{n}{2}$ in (1) equation

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

$T\left(\frac{n}{2^2}\right) = T\left(\frac{n}{2^3}\right) + 1$.................... Putting $\frac{n}{2}$ in place of n in eq 1.

T (n) = $T\left(\frac{n}{2^3}\right) + 1 + 2$

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

$T\left(\frac{n}{2^3}\right) = T\left(\frac{n}{2^4}\right) + 1$................. Putting $\frac{n}{3}$ in place of n in eq1

Put T $\left(\frac{n}{2^3}\right)$ in eq (3)

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

Repeat the same process ith times

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

Stopping Condition: T (1) =1

At least there will be only one term left that's why that term will compare out, and only one comparison be done that's why

$$\begin{tikzcd}
 & T(1)=1 \arrow[dl, dash] \arrow[dr, dash] & \\
\text{Item} & & \text{Comparison}
\end{tikzcd}$$

Is the last term of the equation and it will be equal to 1

$$\frac{n}{2^i} = 1$$

$\frac{n}{2^i}$ Is the last term of the equation and it will be equal to 1

$n = 2^i$

Applying log both sides

$\log n = \log_2 i$

Log n = i log 2

$\frac{\log n}{\log 2} = 1$

$\log_2 n = i$

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

$\frac{n}{2^i}$ = 1 as in eq 5

$= T(1) + i$

$= 1 + i$.................................. T (1) =1 by stopping condition

$= 1 + \log_2 n$

$= \log_2 n$ .... (1 is a constant that's why ignore it)

Therefore, binary search is of order o ($\log_2 n$)

Algorithm for binary search (recursive):-

```
Algorithm binary_search(A, key, imin, imax)
{
if (imax < imin) then
return "array is empty";
if(key<imin || key>imax) then
return "element not in array list"
else
{
imid = (imin +imax)/2;
if (A[imid] > key) then
return binary_search(A, key, imin, imid-1);
else if (A[imid] < key) then
return binary_search(A, key, imid+1, imax);
else
return imid;
}
}
```

Binary Search example program:-

```
// Binary Search in C
#include<stdio.h>
int binarySearch(int array[], int x, int low, int high) {
  if (high >= low) {
    int mid = low + (high - low) / 2;
    // If found at mid, then return it
    if (array[mid] == x)
      return mid;
    // Search the left half
    if (array[mid] > x)
      return binarySearch(array, x, low, mid - 1);
    // Search the right half
    return binarySearch(array, x, mid + 1, high);
  }
  return -1;
}
void main() {
  int array[] = {3, 4, 5, 6, 7, 8, 9};
  int n = sizeof(array) / sizeof(array[0]);
  int x = 4;
  int result = binarySearch(array, x, 0, n - 1);
  if (result == -1)
    printf("Not found");
  else
    printf("Element is found at index %d", result);
}
```

Binary Search Complexity

Time Complexities

Best case complexity: O(1)

Average case complexity: O(log n)

Worst case complexity: O(log n)

Space Complexity

The space complexity of the binary search is O(1).

MERGE SORT

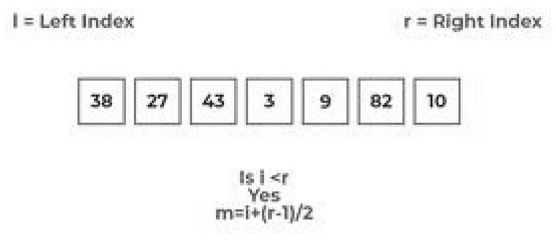
The **Merge Sort** algorithm is a sorting algorithm that is based on the **Divide and Conquer** paradigm. In this algorithm, the array is initially divided into two equal halves and then they are combined in a sorted manner.

Merge Sort Working Process:

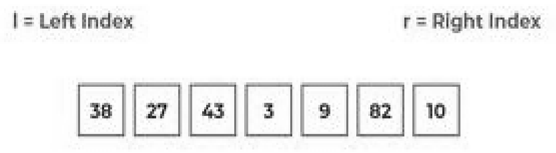
It as a recursive algorithm continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

To know the functioning of merge sort, lets consider an array arr[] = {38, 27, 43, 3, 9, 82, 10}

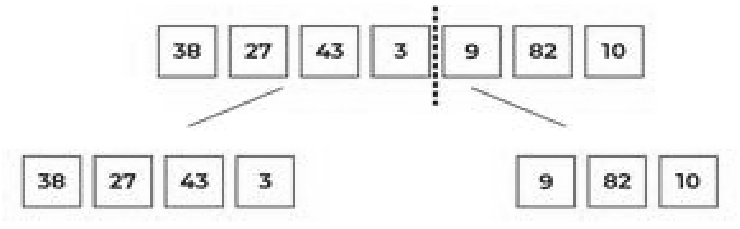
- *At first, check if the left index of array is less than the right index, if yes then calculate its mid point*



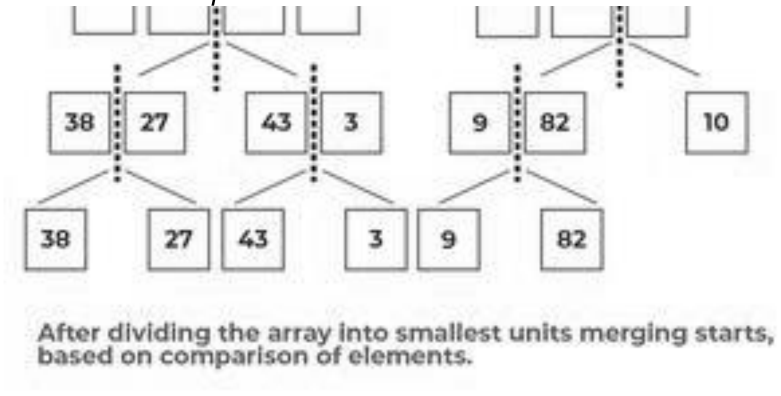
- *Now, as we already know that merge sort first divides the whole array iteratively into equal halves, unless the atomic values are achieved.*
- *Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.*



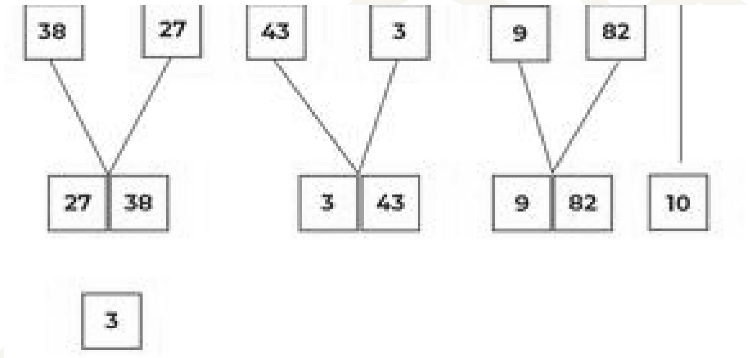
- *Now, again find that is left index is less than the right index for both arrays, if found yes, then again calculate mid points for both the arrays.*



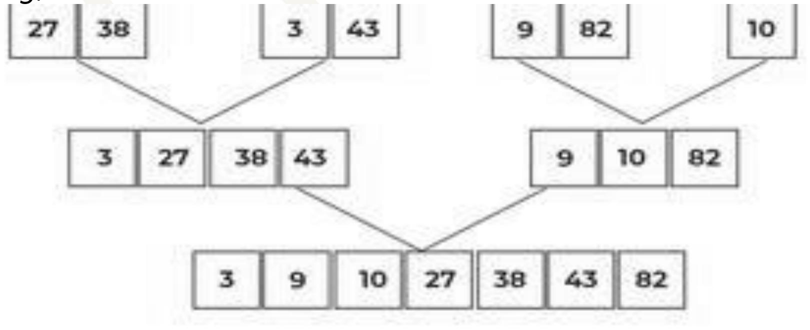
- *Now, further divide these two arrays into further halves, until the atomic units of the array is reached and further division is not possible.*


38
27
43
3
9
82
10
After dividing the array into smallest units merging starts, based on comparison of elements.

- *After dividing the array into smallest units, start merging the elements again based on comparison of size of elements*
- *Firstly, compare the element for each list and then combine them into another list in a sorted manner.*

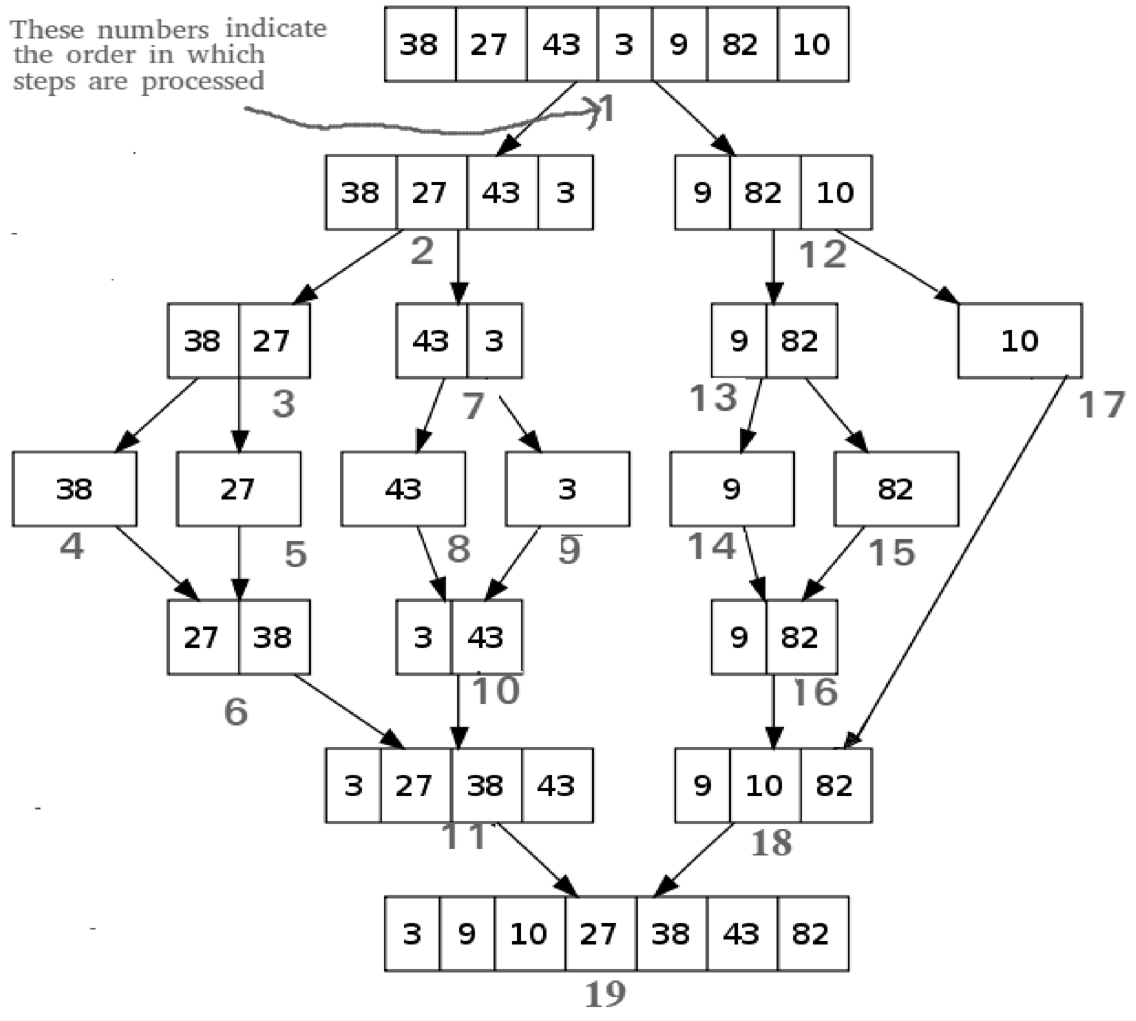


- *After the final merging, the list looks like this:*



The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}.

If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



Merge Sort Algorithm:

```
step 1: start
step 2: declare array and left, right, mid variable
step 3: perform merge function.
    if left > right
        return
    mid= (left+right)/2
    mergesort(array, left, mid)
    mergesort(array, mid+1, right)
    merge(array, left, mid, right)
step 4: Stop
```

Follow the steps below the solve the problem:

MergeSort(arr[], l, r)

If r > l

Find the middle point to divide the array into two halves:

middle m = l + (r – l)/2

Call mergeSort for first half:

```
        Call mergeSort(arr, l, m)
Call mergeSort for second half:
        Call mergeSort(arr, m + 1, r)
Merge the two halves sorted in steps 2 and 3:
        Call merge(arr, l, m, r)
```

Time Complexity: O(N log(N)), Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.
$T(n) = 2T(n/2) + \theta(n)$

The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of the Master Method and the solution of the recurrence is θ(Nlog(N)). The time complexity of Merge Sort isθ(Nlog(N)) in all 3 cases (worst, average, and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.

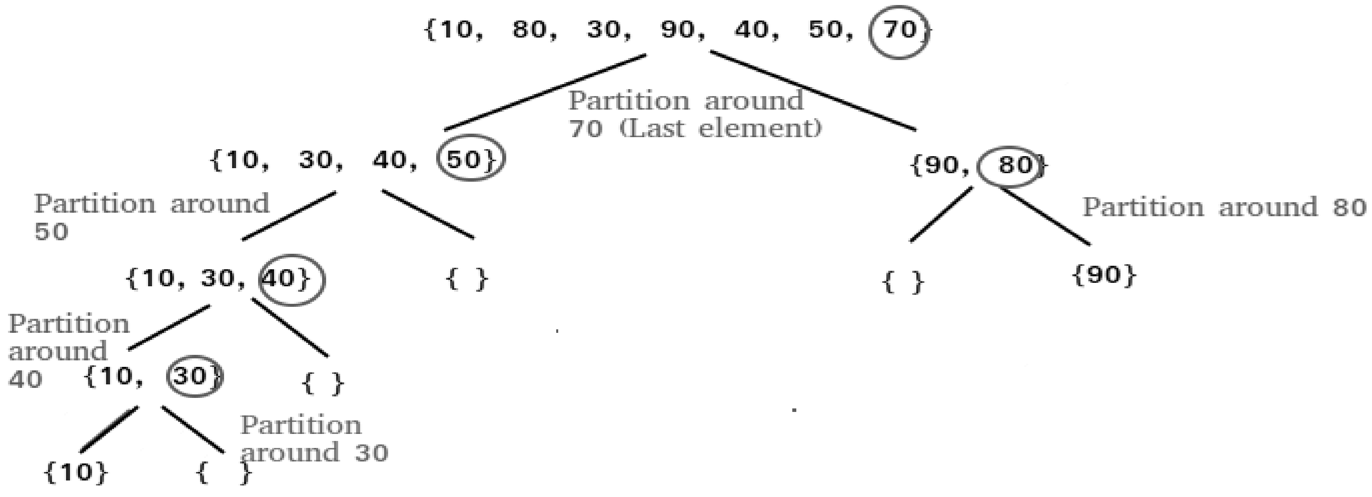
Auxiliary Space: O(n), In merge sort all elements are copied into an auxiliary array. So N auxiliary space is required for merge sort.

QUICK SORT (OR) PARTITION SORTING

Quick Sort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot (implemented below)
- Pick a random element as a pivot.
- Pick median as the pivot.

The key process in **quick Sort** is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.



Partition Algorithm:

The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap the current element with arr[i]. Otherwise, we ignore the current element.

Code for recursive QuickSort function:-

```
/* low  –> Starting index,  high  –> Ending index */
quickSort(arr[], low, high) {
   if (low < high) {
      /* pi is partitioning index, arr[pi] is now at right place */
      pi = partition(arr, low, high);
      quickSort(arr, low, pi – 1);  // Before pi
      quickSort(arr, pi + 1, high); // After pi
   }
}
```

Code for partition() :-

```
/* This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot */
partition (arr[], low, high)
{
   // pivot (Element to be placed at right position)
pivot = arr[high];
 i = (low – 1)  // Index of smaller element and indicates the
// right position of pivot found so far
for (j = low; j <= high- 1; j++){
 // If current element is smaller than the pivot
if (arr[j] < pivot){
i++;    // increment index of smaller element
 swap arr[i] and arr[j]
    }
 }
   swap arr[i + 1] and arr[high])
return (i + 1)
}
```

Analysis of Quick Sort:-

Time taken by Quick Sort, in general, can be written as follows.

$$T(n) = T(k) + T(n-k-1) + \theta(n)$$

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements that are smaller than the pivot.
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

Worst Case:

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

$$T(n) = T(0) + T(n-1) + \theta(n) \text{ which is equivalent to } T(n) = T(n-1) + \theta(n)$$

The solution to the above recurrence is (n^2).

Best Case:

The best case occurs when the partition process always picks the middle element as the pivot. The following is recurrence for the best case.

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

The solution for the above recurrence is (nLogn).

Greedy method

Greedy method

The greedy method is one of the strategies like Divide and conquer used to solve the problems. This method is used for solving optimization problems. An optimization problem is a problem that demands either maximum or minimum results. Let's understand through some terms.

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 the feasible, whereas the optimal solution is the best solution among all the solutions.

Characteristics of Greedy method

- To construct the solution in an optimal way, this algorithm creates two sets where one set contains all the chosen items, and another set contains the rejected items.
- A Greedy algorithm makes good local choices in the hope that the solution should be either feasible or optimal.

Components of Greedy Algorithm

- **Candidate set:** A solution that is created from the set is known as a candidate set.

- **Selection function:** This function is used to choose the candidate or subset which can be added in the solution.
- **Feasibility function**: A function that is used to determine whether the candidate or subset can be used to contribute to the solution or not.
- **Objective function**: A function is used to assign the value to the solution or the partial solution.
- **Solution function**: This function is used to intimate whether the complete function has been reached or not.

Applications of Greedy Algorithm

- It is used in finding the shortest path.
- It is used to find the minimum spanning tree using the prim's algorithm or the Kruskal's algorithm.
- It is used in a job sequencing with a deadline.
- This algorithm is also used to solve the fractional knapsack problem.

Greedy Algorithm

```
Algorithm Greedy (a, n)
{
   Solution : = 0;
  for i = 0 to n do
  {
      x: = select(a);
     if feasible(solution, x)
    {
        Solution: = union(solution , x)
    }
       return solution;
  } }
```

The above is the greedy algorithm. Initially, the solution is assigned with zero value. We pass the array and number of elements in the greedy algorithm. Inside the for loop, we select the element one by one and checks whether the solution is feasible or not. If the solution is feasible, then we perform the union.

Example:-

Suppose there is a problem 'P'. I want to travel from A to B shown as below:

P : A → B

The problem is that we have to travel this journey from A to B. There are various solutions to go from A to B. We can go from A to B by **walk, car, bike, train, aero plane**, etc. There is a constraint in the journey that we have to travel this journey within 12 hrs. If I go by train or aero plane then only, I can cover this distance within 12 hrs. There are many solutions to this problem but there are only two solutions that satisfy the constraint.

If we say that we have to cover the journey at the minimum cost. This means that we have to travel this distance as minimum as possible, so this problem is known as a minimization problem. Till now, we have two feasible solutions, i.e., one by train and another one by air. Since travelling by train will lead to the minimum cost so it is an optimal solution. An optimal solution is also the feasible solution, but providing the best result so that solution is the optimal solution with the minimum cost. There would be only one optimal solution.

The problem that requires either minimum or maximum result then that problem is known as an optimization problem. Greedy method is one of the strategies used for solving the optimization problems.

APPLICATIONS OF GREEDY METHOD

JOB SEQUENCING WITH DEADLINES

The sequencing of jobs on a single processor with deadline constraints is called as Job Sequencing with Deadlines.

Here-

- We are given a set of jobs.
- Each job has a defined deadline and some profit associated with it.
- The profit of a job is given only when that job is completed within its deadline.
- Only one processor is available for processing all the jobs.
- Processor takes one unit of time to complete a job.

Approach to Solution

- A feasible solution would be a subset of jobs where each job of the subset gets completed within its deadline.
- Value of the feasible solution would be the sum of profit of all the jobs contained in the subset.
- An optimal solution of the problem would be a feasible solution which gives the maximum profit.

Greedy Algorithm is adopted to determine how the next job is selected for an optimal solution.

The greedy algorithm described below always gives an optimal solution to the job sequencing problem.

Step-01: Sort all the given jobs in decreasing order of their profit.

Step-02: Check the value of maximum deadline. Draw a Gantt chart where maximum time on Gantt chart is the value of maximum deadline.

Step-03: Pick up the jobs one by one. Put the job on Gantt chart as far as possible from 0 ensuring that the job gets completed before its deadline.

Example:-

Given the jobs, their deadlines and associated profits as shown-

Jobs	J1	J2	J3	J4	J5	J6
Deadlines	5	3	3	2	4	2
Profits	200	180	190	300	120	100

Solution-

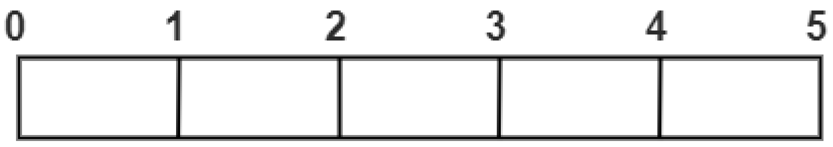
Step-01:

Sort all the given jobs in decreasing order of their profit-

Jobs	J4	J1	J3	J2	J5	J6
Deadlines	2	5	3	3	4	2
Profits	300	200	190	180	120	100

Step-02:

Value of maximum deadline = 5.
So, draw a Gantt chart with maximum time on Gantt chart = 5 units as shown-



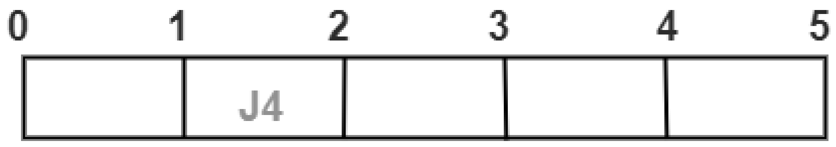
Gantt Chart

Now,

- We take each job one by one in the order they appear in Step-01.
- We place the job on Gantt chart as far as possible from 0.

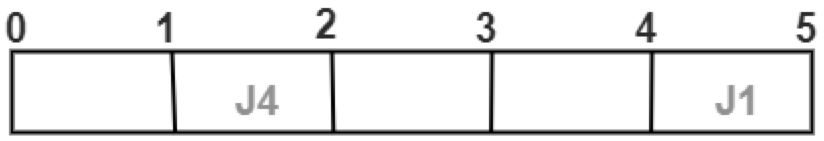
Step-03:

- We take job J4.
- Since its deadline is 2, so we place it in the first empty cell before deadline 2 as-



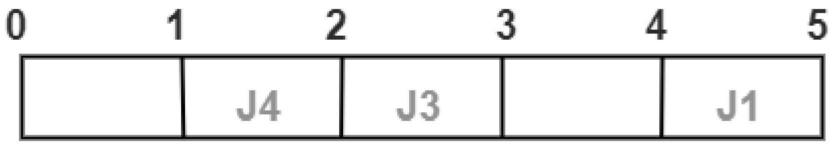
Step-04:

- We take job J1.
- Since its deadline is 5, so we place it in the first empty cell before deadline 5 as-



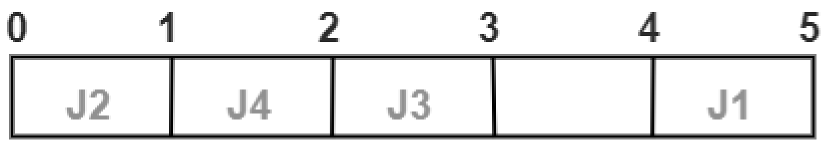
Step-05:

- We take job J3.
- Since its deadline is 3, so we place it in the first empty cell before deadline 3 as-



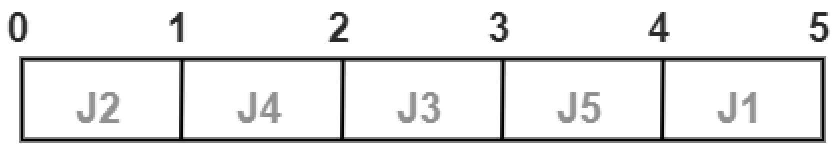
Step-06:

- We take job J2.
- Since its deadline is 3, so we place it in the first empty cell before deadline 3.
- Since the second and third cells are already filled, so we place job J2 in the first cell as-



Step-07:

- Now, we take job J5.
- Since its deadline is 4, so we place it in the first empty cell before deadline 4 as-



Now,

- The only job left is job J6 whose deadline is 2.
- All the slots before deadline 2 are already occupied.
- Thus, job J6 cannot be completed.

The optimal schedule is-

J2 , J4 , J3 , J5 , J1

This is the required order in which the jobs must be completed in order to obtain the maximum profit.

- All the jobs are not completed in optimal schedule.
- This is because job J6 could not be completed within its deadline.

Maximum earned profit
= 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

0/1 Knapsack Problem

Given a set of items, each with a weight and a value, determine a subset of items to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

The knapsack problem is in combinatorial optimization problem. It appears as a sub problem in many, more complex mathematical models of real-world problems. One general approach to difficult problems is to identify the most restrictive constraint, ignore the others, solve a knapsack problem, and somehow adjust the solution to satisfy the ignored constraints.

Problem Scenario

A thief is robbing a store and can carry a maximal weight of ***W*** into his knapsack. There are n items available in the store and weight of i^{th} item is w_i and its profit is p_i. What items should the thief take?
In this context, the items should be selected in such a way that the thief will carry those items for which he will gain maximum profit. Hence, the objective of the thief is to maximize the profit.
Based on the nature of the items, Knapsack problems are categorized as

- Fractional Knapsack
- Knapsack

Fractional Knapsack

In this case, items can be broken into smaller pieces, hence the thief can select fractions of items.
According to the problem statement,

- There are **n** items in the store
- Weight of **i**th item $w_i>0$
- Profit for **i**th item $p_i>0$ and
- Capacity of the Knapsack is **W**

In this version of Knapsack problem, items can be broken into smaller pieces. So, the thief may take only a fraction x_i of **i**th item.

$$0 \leqslant x_i \leqslant 1$$

The **i**th item contributes the weight **xi.wi** to the total weight in the knapsack and profit **xi.pi** to the total profit.

Hence, the objective of this algorithm is to $maximize \sum_{n=1}^{n} (x_i . pi)$

Subject to constraint, $$\sum_{n=1}^{n}(x_i . wi) \leqslant W$$

It is clear that an optimal solution must fill the knapsack exactly; otherwise we could add a fraction of one of the remaining items and increase the overall profit.

Thus, an optimal solution can be obtained by

$$\sum_{n=1}^{n}(x_i . wi) = W$$

In this context, first we need to sort those items according to the value of **pi/wi**, so that **pi+1/wi+1 ≤ pi/wi**. Here, ***x*** is an array to store the fraction of items.

Algorithm: Greedy-Fractional-Knapsack (w[1..n], p[1..n], W)

```
for i = 1 to n
  do x[i] = 0
weight = 0
for i = 1 to n
  if weight + w[i] ≤ W then
    x[i] = 1
    weight = weight + w[i]
  else
    x[i] = (W - weight) / w[i]
    weight = W
    break
return x
```

Analysis

If the provided items are already sorted into a decreasing order of piwipiwi, then the whileloop takes a time in ***O(n)***; Therefore, the total time including the sort is in ***O(n logn)***.

Example

Let us consider that the capacity of the knapsack ***W* = 60** and the list of provided items are shown in the following table −

Item	A	B	C	D
Profit	280	100	120	120
Weight	40	10	20	24
Ratio (Pi/Wi)	7	10	6	5

As the provided items are not sorted based on pi/wi. After sorting, the items are as shown in the following table.

Item	B	A	C	D
Profit	100	280	120	120
Weight	10	40	20	24
Ratio (Pi/Wi)	10	7	6	5

Solution

After sorting all the items according to pi/wi. First all of ***B*** is chosen as weight of ***B*** is less than the capacity of the knapsack. Next, item ***A*** is chosen, as the available capacity of the knapsack is greater than the weight of ***A***. Now, ***C*** is chosen as the next item. However, the whole item cannot be chosen as the remaining capacity of the knapsack is less than the weight of ***C***.

Hence, fraction of ***C*** (i.e. (60 − 50)/20) is chosen.

Now, the capacity of the Knapsack is equal to the selected items. Hence, no more items can be selected.

The total weight of the selected items is **10 + 40 + 20 * (10/20) = 60**

And the total profit is **100 + 280 + 120 * (10/20) = 380 + 60 = 440**

This is the optimal solution. We cannot gain more profit selecting any different combination of items.

Minimum Spanning Tree

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

A spanning tree of a connected undirected graph is a sub graph, i.e., a tree structure that binds all vertices with a minimum edge cost sum. If we have graph G with vertices V and edges E, then that graph can be represented as G(V, E). For this graph G(V, E), if we construct a tree structure G’(V’, E’) such that the formed tree structure follows constraints mentioned below, then that structure can be called a Spanning Tree.

V’ = V (number of Vertices in G’ must be equal to the number of vertices in G)

E’ = |V| - 1 (Edges of G’ must be equal to the number of vertices in graph G minus 1)

Prim’s algorithm

It is a greedy algorithm. It starts with an empty spanning tree. The idea is to maintain two sets of vertices:

- Contain vertices already included in MST.

- Contain vertices not yet included.

At every step, it considers all the edges and picks the minimum weight edge. After picking the edge, it moves the other endpoint of edge to set containing MST.

Steps for finding MST using Prim's Algorithm:

Step-01:

- Randomly choose any vertex.
- The vertex connecting to the edge having least weight is usually selected.

Step-02:

- Find all the edges that connect the tree to new vertices.
- Find the least weight edge among those edges and include it in the existing tree.
- If including that edge creates a cycle, then reject that edge and look for the next least weight edge.

Step-03:

- Keep repeating step-02 until all the vertices are included and Minimum Spanning Tree (MST) is obtained.

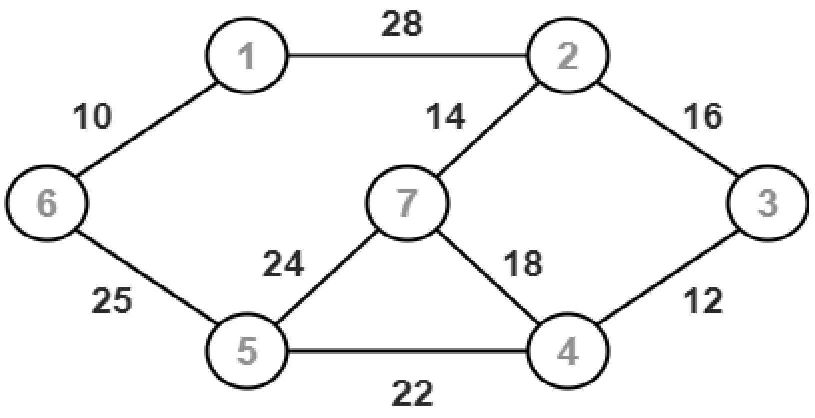
Prim's Algorithm Time Complexity-

Worst case time complexity of Prim's Algorithm is-

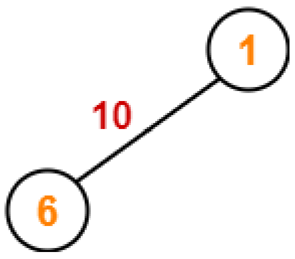
O(ElogV) using binary heap

O(E + VlogV) using Fibonacci heap

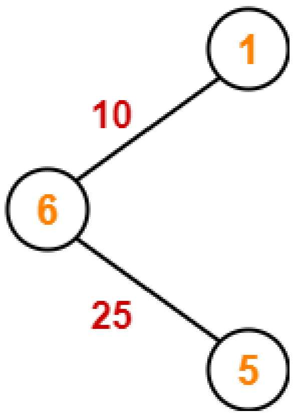
Example: Construct the minimum spanning tree (MST) for the given graph using Prim's Algorithm-



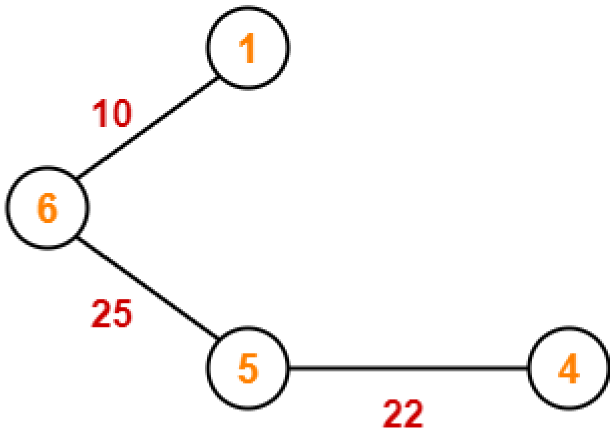
Step-01:



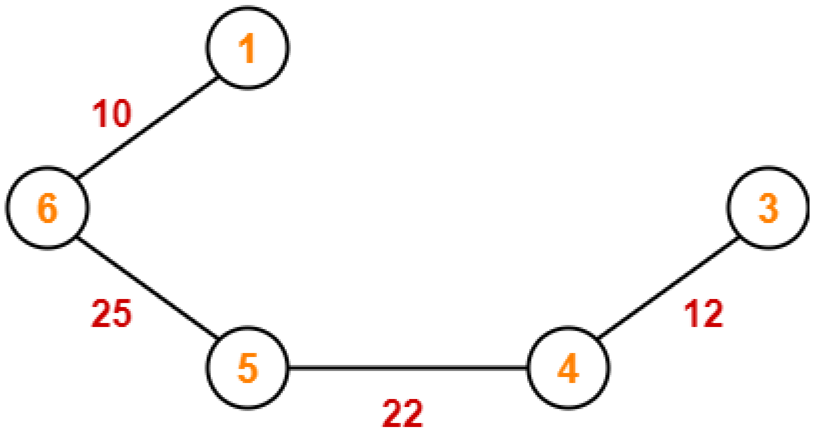
Step-02:



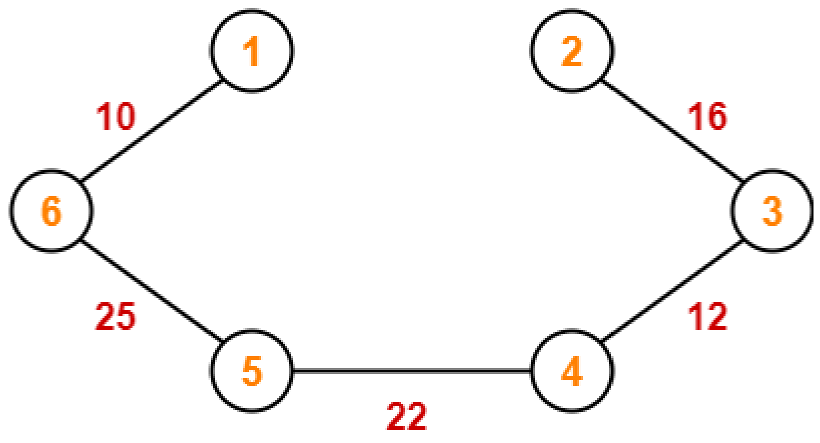
Step-03:



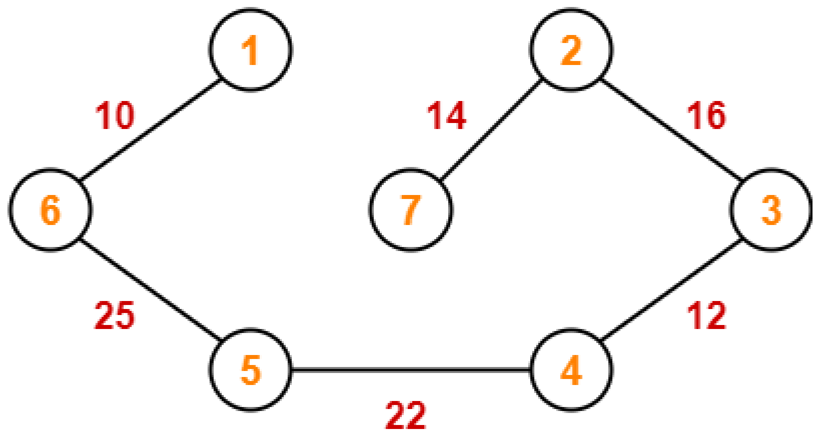
Step-04:



Step-05:



Step-06:



Since all the vertices have been included in the MST, so we stop.
Now, Cost of Minimum Spanning Tree
= Sum of all edge weights
= 10 + 25 + 22 + 12 + 16 + 14
= 99 units

KRUSKAL'S ALGORITHM-

- Kruskal's Algorithm is a famous greedy algorithm.
- It is used for finding the Minimum Spanning Tree (MST) of a given graph.
- To apply Kruskal's algorithm, the given graph must be weighted, connected and undirected.

Kruskal's Algorithm Implementation-

The implementation of Kruskal's Algorithm is explained in the following steps-

Step-01:

Sort all the edges from low weight to high weight.

Step-02:

- Take the edge with the lowest weight and use it to connect the vertices of graph.
- If adding an edge creates a cycle, then reject that edge and go for the next least weight edge.

Step-03:

- Keep adding edges until all the vertices are connected and a Minimum Spanning Tree (MST) is obtained.

Kruskal's Algorithm Time Complexity-

Worst case time complexity of Kruskal's Algorithm
= O(ElogV) or O(ElogE)

Analysis-

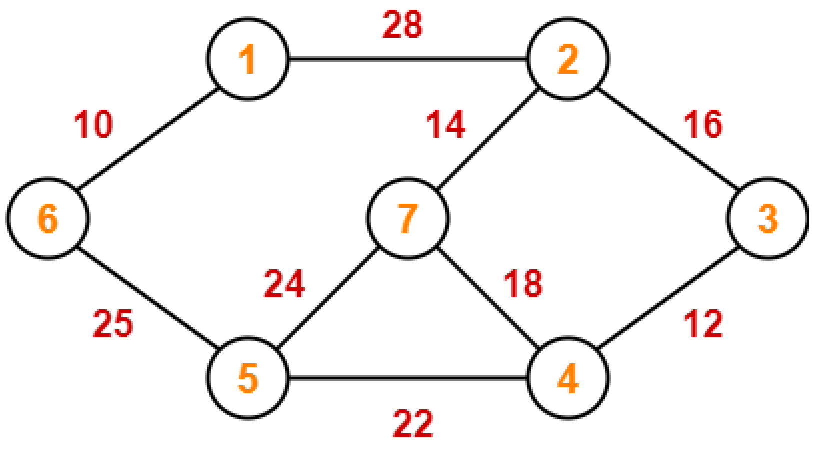
- The edges are maintained as min heap.
- The next edge can be obtained in O(logE) time if graph has E edges.
- Reconstruction of heap takes O(E) time.
- So, Kruskal's Algorithm takes O(ElogE) time.
- The value of E can be at most $O(V^2)$.
- So, O(logV) and O(logE) are same.

Special Case-

- If the edges are already sorted, then there is no need to construct min heap.
- So, deletion from min heap time is saved.
- In this case, time complexity of Kruskal's Algorithm = O(E + V)

Example:-

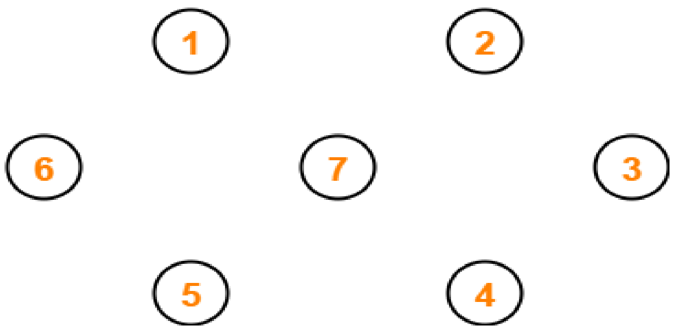
Construct the minimum spanning tree (MST) for the given graph using Kruskal's Algorithm-



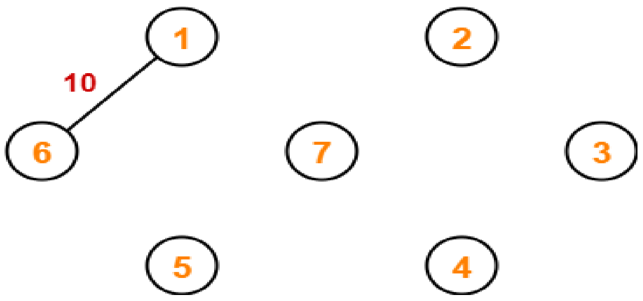
To construct MST using Kruskal's Algorithm,

- Simply draw all the vertices on the paper.
- Connect these vertices using edges with minimum weights such that no cycle gets formed.

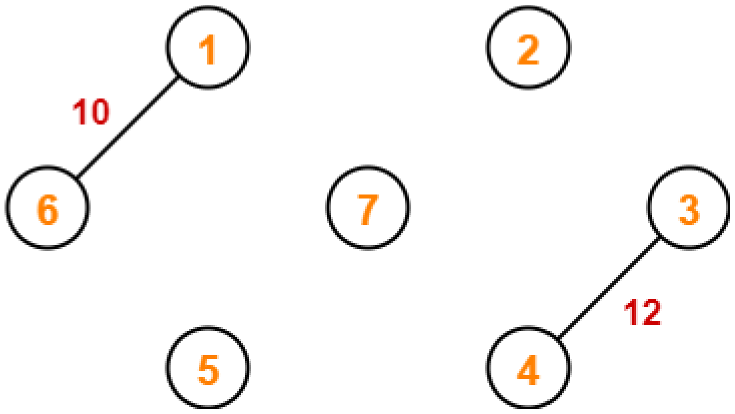
Step-01:



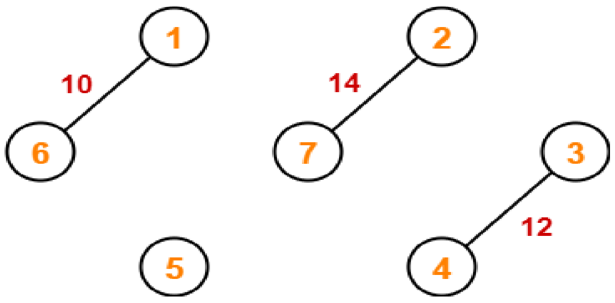
Step-02:



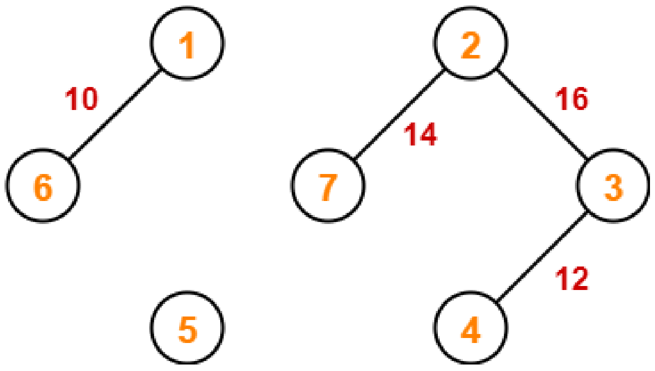
Step-03:



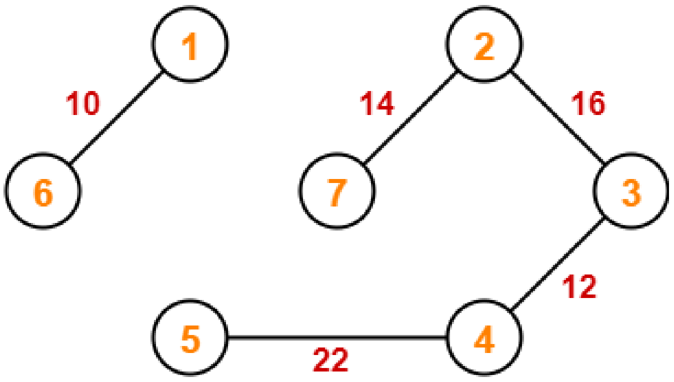
Step-04:



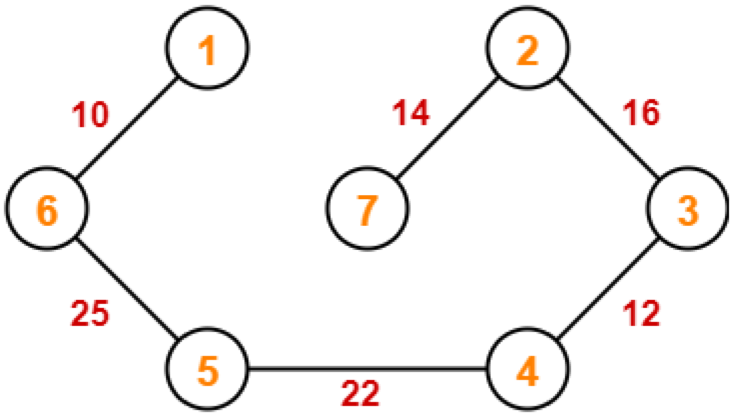
Step-05:



Step-06:



Step-07:



Since all the vertices have been connected / included in the MST, so we stop.
Weight of the MST
= Sum of all edge weights
= 10 + 25 + 22 + 12 + 16 + 14
= 99 units

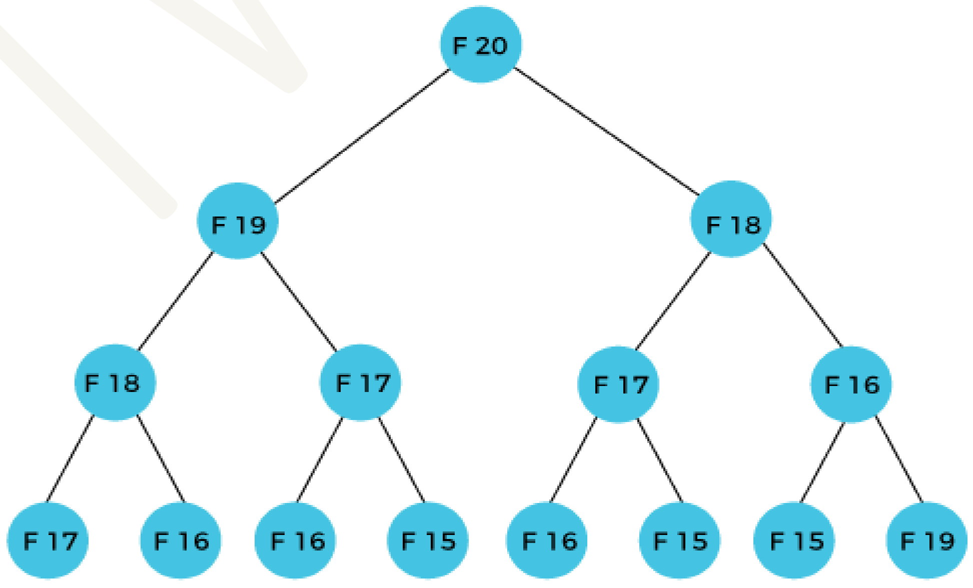
UNIT-3

Dynamic Programming: general method

Dynamic programming is a technique that breaks the problems into sub-problems, and saves the result for future purposes so that we do not need to compute the result again. The sub problems are optimized to optimize the overall solution is known as optimal substructure property. The main use of dynamic programming is to solve optimization problems. Here, optimization problems mean that when we are trying to find out the minimum or the maximum solution of a problem. The dynamic programming guarantees to find the optimal solution of a problem if the solution exists.

The definition of dynamic programming says that it is a technique for solving a complex problem by first breaking into a collection of simpler sub problems, solving each sub problem just once, and then storing their solutions to avoid repetitive computations.

Consider an example of the Fibonacci series:-
The following series is the Fibonacci series:
0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ,...
The numbers in the above series are not randomly calculated. Mathematically, we could write each of the terms using the below formula:
F(n) = F(n-1) + F(n-2),
With the base values F(0) = 0, and F(1) = 1. To calculate the other numbers, we follow the above relationship. For example, F(2) is the sum **f(0)** and **f(1),** which is equal to 1.
Calculate F(20):-
The F(20) term will be calculated using the nth formula of the Fibonacci series. The below figure shows that how F(20) is calculated.



As we can observe in the above figure that F(20) is calculated as the sum of F(19) and F(18). In the dynamic programming approach, we try to divide the problem into the similar subproblems. We are following this approach in the above case where F(20) into the similar subproblems, i.e., F(19) and F(18). If we recap the definition of dynamic programming that it says the similar subproblem should not be computed more than once. Still, in the above case, the subproblem is calculated twice. In the above example, F(18) is calculated two times; similarly, F(17) is also calculated twice. However, this technique is quite useful as it solves the similar subproblems, but we need to be cautious while storing the results because we are not particular about storing the result that we have computed once, then it can lead to a wastage of resources.

In the above example, if we calculate the F(18) in the right subtree, then it leads to the tremendous usage of resources and decreases the overall performance.

The solution to the above problem is to save the computed results in an array. First, we calculate F(16) and F(17) and save their values in an array. The F(18) is calculated by summing the values of F(17) and F(16), which are already saved in an array. The computed value of F(18) is saved in an array. The value of F(19) is calculated using the sum of F(18), and F(17), and their values are already saved in an array. The computed value of F(19) is stored in an array. The value of F(20) can be calculated by adding the values of F(19) and F(18), and the values of both F(19) and F(18) are stored in an array. The final computed value of F(20) is stored in an array.

working of dynamic programming approach :-

The following are the steps that the dynamic programming follows:

- It breaks down the complex problem into simpler sub problems.
- It finds the optimal solution to these sub-problems.
- It stores the results of sub problems (**memoization)**. The process of storing the results of sub problems is known as **memorization**.
- It reuses them so that same sub-problem is calculated more than once.
- Finally, calculate the result of the complex problem.

The above five steps are the basic steps for dynamic programming. The dynamic programming is applicable that are having properties such as:

Those problems that are having overlapping sub problems and optimal substructures. Here, optimal substructure means that the solution of optimization problems can be obtained by simply combining the optimal solution of all the sub problems.

In the case of dynamic programming, the space complexity would be increased as we are storing the intermediate results, but the time complexity would be decreased.

Approaches of dynamic programming

There are two approaches to dynamic programming:

- Top-down approach (**memorization technique**)
- Bottom-up approach (**tabulation technique**)

Top-down approach (memorization technique):-

The top-down approach follows the memorization technique, while bottom-up approach follows the tabulation method. Here memorization is equal to the sum of recursion and caching. Recursion means calling the function itself, while caching means storing the intermediate results.

Advantages

- It is very easy to understand and implement.

- It solves the sub problems only when it is required.
- It is easy to debug.

Disadvantages

- It uses the recursion technique that occupies more memory in the call stack. Sometimes when the recursion is too deep, the stack overflow condition will occur.
- It occupies more memory that degrades the overall performance.

Example:-

```
 int fib(int n)
 {
   if(n<0)
   error;
  if(n==0)
  return 0;
  if(n==1)
 return 1;
 sum = fib(n-1) + fib(n-2);
 }
```

In the above code, we have used the recursive approach to find out the Fibonacci series. When the value of 'n' increases, the function calls will also increase, and computations will also increase. In this case, the time complexity increases exponentially, and it becomes 2^n.

One solution to this problem is to use the dynamic programming approach. Rather than generating the recursive tree again and again, we can reuse the previously calculated value. If we use the dynamic programming approach, then the time complexity would be O(n).

When we apply the dynamic programming approach in the implementation of the Fibonacci series, then the code would look like:

```
static int count = 0;
int fib(int n)
{
if(memo[n]!= NULL)
return memo[n];
count++;
  if(n<0)
  error;
 if(n==0)
 return 0;
 if(n==1)
return 1;
sum = fib(n-1) + fib(n-2);
memo[n] = sum;
}
```

In the above code, we have used the memorization technique in which we store the results in an array to reuse the values. This is also known as a top-down approach in which we move from the top and break the problem into sub-problems.

Bottom-Up approach (tabulation technique):-

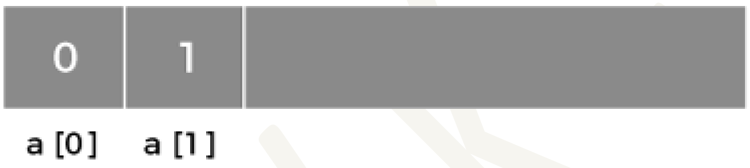
The bottom-up approach is also one of the techniques which can be used to implement the dynamic programming. It uses the tabulation technique to implement the dynamic programming approach. It solves the same kind of problems but it removes the recursion. If we remove the recursion, there is no stack overflow issue and no overhead of the recursive functions. In this tabulation technique, we solve the problems and store the results in a matrix.

The bottom-up is the approach used to avoid the recursion, thus saving the memory space. The bottom-up is an algorithm that starts from the beginning, whereas the recursive algorithm starts from the end and works backward. In the bottom-up approach, we start from the base case to find the answer for the end. As we know, the base cases in the Fibonacci series are 0 and 1. Since the bottom approach starts from the base cases, so we will start from 0 and 1.

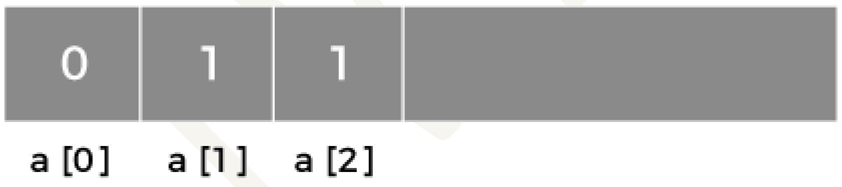
- We solve all the smaller sub-problems that will be needed to solve the larger sub-problems then move to the larger problems using smaller sub-problems.
- We use for loop to iterate over the sub-problems.
- The bottom-up approach is also known as the tabulation or table filling method.

Example:-

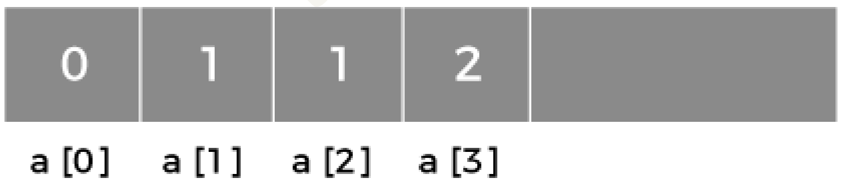
Suppose we have an array that has 0 and 1 values at a[0] and a[1] positions, respectively shown as below:



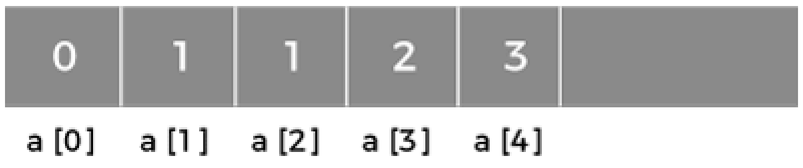
Since the bottom-up approach starts from the lower values, so the values at a[0] and a[1] are added to find the value of a[2] shown as below:



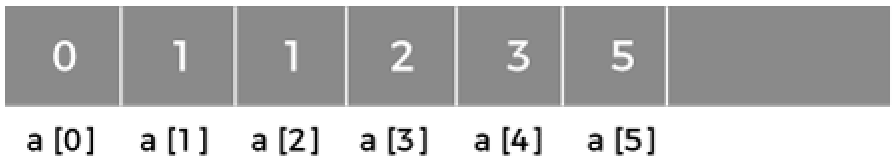
The value of a[3] will be calculated by adding a[1] and a[2], and it becomes 2 shown as below:



The value of a[4] will be calculated by adding a[2] and a[3], and it becomes 3 shown as below:



The value of a[5] will be calculated by adding the values of a[4] and a[3], and it becomes 5 shown as below:



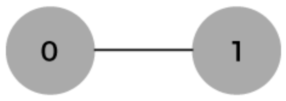
The code for implementing the Fibonacci series using the bottom-up approach is given below:

```
int fib(int n)
{
   int A[];
   A[0] = 0, A[1] = 1;
   for( i=2; i<=n; i++)
   {
       A[i] = A[i-1] + A[i-2]
   }
   return A[n];
}
```

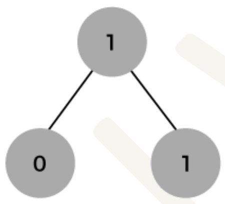
In the above code, base cases are 0 and 1 and then we have used for loop to find other values of Fibonacci series.

Diagrammatic representation:-

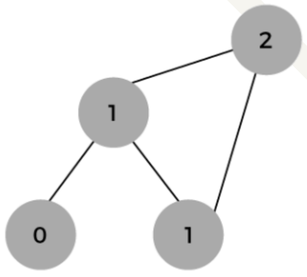
Initially, the first two values, i.e., 0 and 1 can be represented as:



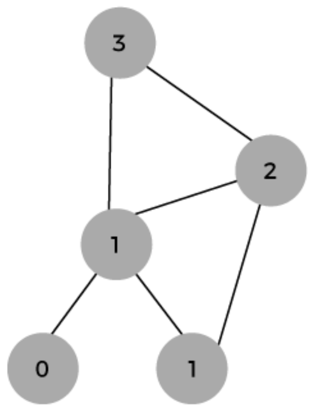
When i=2 then the values 0 and 1 are added shown as below:



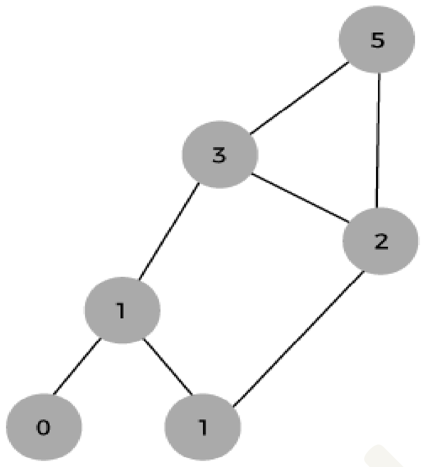
When i=3 then the values 1and 1 are added shown as below:



When i=4 then the values 2 and 1 are added shown as below:



When i=5, then the values 3 and 2 are added shown as below:



In the above case, we are starting from the bottom and reaching to the top.

Divide and Conquer Method v/s Dynamic Programming

Divide and Conquer Method	Dynamic Programming
1.It deals (involves) three steps at each level of recursion: **Divide** the problem into a number of subproblems. **Conquer** the subproblems by solving them recursively. **Combine** the solution to the subproblems into the solution for original subproblems.	**1.**It involves the sequence of four steps: ○ Characterize the structure of optimal solutions. ○ Recursively defines the values of optimal solutions. ○ Compute the value of optimal solutions in a Bottom-up minimum. ○ Construct an Optimal Solution from computed information.
2. It is Recursive.	**2.** It is non Recursive.

3. It does more work on subproblems and hence has more time consumption.	**3.** It solves subproblems only once and then stores in the table.
4. It is a top-down approach.	**4.** It is a Bottom-up approach.
5. In this subproblems are independent of each other.	**5.** In this subproblems are interdependent.
6. For example: Merge Sort & Binary Search etc.	**6. For example:** Matrix Multiplication.

Optimal Binary Search Tree

- Optimal Binary Search Tree extends the concept of Binary search tree. Binary Search Tree (BST) is a *nonlinear* data structure which is used in many scientific applications for reducing the search time. In BST, left child is smaller than root and right child is greater than root. This arrangement simplifies the search procedure.
- Optimal Binary Search Tree (OBST) is very useful in dictionary search. The probability of searching is different for different words. OBST has great application in translation. If we translate the book from English to German, equivalent words are searched from English to German dictionary and replaced in translation. Words are searched same as in binary search tree order.
- Binary search tree simply arranges the words in lexicographical order. Words like *'the'*, *'is'*, *'there'* are very frequent words, whereas words like *'xylophone'*, *'anthropology'* etc. appears rarely.
- It is not a wise idea to keep less frequent words near root in binary search tree. Instead of storing words in binary search tree in lexicographical order, we shall arrange them according to their probabilities. This arrangement facilitates few searches for frequent words as they would be near the root. Such tree is called **Optimal Binary Search Tree**.
- Consider the sequence of *n* keys K = < k_1, k_2, k_3, …, k_n> of distinct probability in sorted order such that
  k_1< k_2< … <k_n. Words between each pair of key lead to unsuccessful search, so for n keys, binary search tree contains n + 1 dummy keys d_i, representing unsuccessful searches.
- Two different representation of BST with same five keys {k_1, k_2, k_3, k_4, k_5} probability is shown in following figure
- With n nodes, there exist $\Omega(4^n / n^{3/2})$ different binary search trees. An exhaustive search for optimal binary search tree leads to huge amount of time.
- The goal is to construct a tree which minimizes the total search cost. Such tree is called optimal binary search tree. OBST does not claim minimum height. It is also not necessary that parent of sub tree has higher priority than its child.
- Dynamic_programming can help us to find such optima tree.

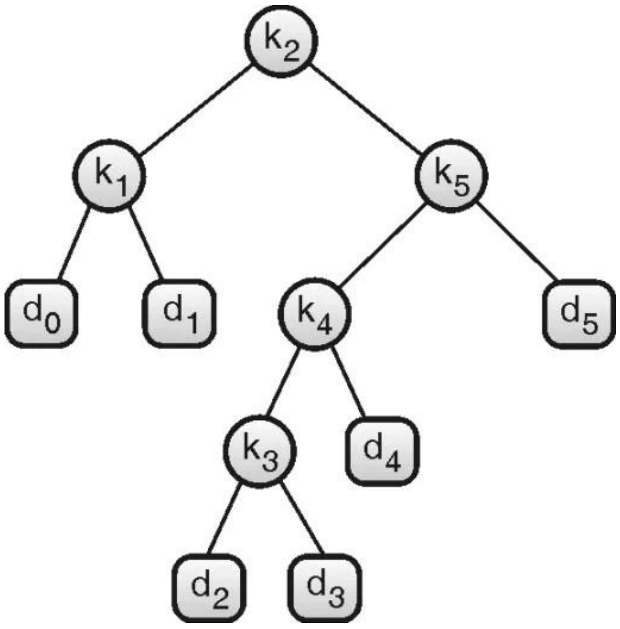


fig: Binary search trees with 5 keys

Mathematical formulation

- We formulate the OBST with following observations
- Any sub tree in OBST contains keys in sorted order $k_i \dots k_j$, where $1 \le i \le j \le n$.
- Sub tree containing keys $k_i \dots k_j$ has leaves with dummy keys $d_{i-1} \dots d_j$.
- Suppose k_r is the root of sub tree containing keys $k_i \dots k_j$. So, left sub tree of root k_r contains keys $k_i \dots k_{r-1}$ and right sub tree contain keys k_{r+1} to k_j. Recursively, optimal sub trees are constructed from the left and right sub trees of k_r.
- Let e[i, j] represents the expected cost of searching OBST. With n keys, our aim is to find and minimize e[1, n].
- Base case occurs when $j = i - 1$, because we just have the dummy key d_{i-1} for this case. Expected search cost for this case would be $e[i, j] = e[i, i - 1] = q_{i-1}$.
- For the case $j \ge i$, we have to select any key k_r from $k_i \dots k_j$ as a root of the tree.
- With k_r as a root key and sub tree $k_i \dots k_j$, sum of probability is defined as

$$w(i, j) = \sum_{m=i}^{j} p_m + \sum_{m=i-1}^{j} q_m$$

Thus, a recursive formula for forming the OBST is stated below:

$$e[i, j] = \begin{cases} q_{i-1} & \text{if } j = i - 1 \\ \min\limits_{i \le r \le j} \{e[i, r-1] + e[r+1, j] + w(i, j)\} & \text{if } i \le j \end{cases}$$

e[i, j] gives the expected cost in the optimal binary search tree.

Algorithm for Optimal Binary Search Tree

The algorithm for optimal binary search tree is specified below:

```
Algorithm OBST(p, q, n)
// e[1…n+1, 0…n ] : Optimal sub tree
// w[1…n+1,  0…n] : Sum of probability
// root[1…n, 1…n] : Used to construct OBST

for i ← 1 to n + 1 do
    e[i, i − 1] ← qi − 1
    w[i, i − 1] ← qi − 1
end

for m ← 1 to n do
    for i ← 1 to n − m + 1 do
        j ← i + m − 1
        e[i, j] ← ∞
        w[i, j] ← w[i, j − 1] + pj + qj
        for r ← i to j do
            t ← e[i, r − 1] + e[r + 1, j] + w[i, j]
            if t < e[i, j] then
                e[i, j] ← t
                root[i, j] ← r
            end
        end
    end
end
return (e, root)
```

Complexity Analysis of Optimal Binary Search Tree

It is very simple to derive the complexity of this approach from the above algorithm. It uses three nested loops. Statements in the innermost loop run in Q(1) time. The running time of the algorithm is computed as

$$T(n) = \sum_{m=1}^{n} \sum_{i=1}^{n-m+1} \sum_{j=i}^{n-1+1} \Theta(1)$$

$$= \sum_{m=1}^{n} \sum_{i=1}^{n-m+1} n = \sum_{m=1}^{n} n^2$$

$$= \Theta(n^3)$$

Thus, the OBST algorithm runs in cubic time.

0/1 Knapsack problem

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

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 the dynamic programming.
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.
Example of 0/1 knapsack problem:-
Consider the problem having weights and profits are:
Weights: {3, 4, 6, 5}
Profits: {2, 3, 1, 4}
The weight of the knapsack is 8 kg
The number of items is 4
The above problem can be solved by using the following method:
x_i = {1, 0, 0, 1}
= {0, 0, 0, 1}
= {0, 1, 0, 1}
The above are the possible combinations. 1 denotes that the item is completely picked and 0 means that no item is picked. Since there are 4 items so possible combinations will be:
2^4 = 16; So. There are 16 possible combinations that can be made by using the above problem. Once all the combinations are made, we have to select the combination that provides the maximum profit.

0/1 Knapsack Problem Using Dynamic Programming

In 0/1 Knapsack Problem,

- As the name suggests, items are indivisible here.
- We cannot take the fraction of any item.
- We have to either take an item completely or leave it completely.
- It is solved using dynamic programming approach.

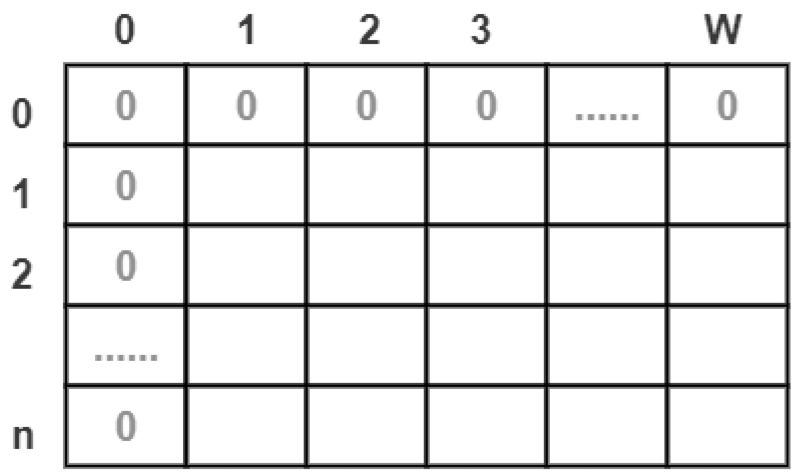
Consider-

- Knapsack weight capacity = w
- Number of items each having some weight and value = n

0/1 knapsack problem is solved using dynamic programming in the following steps-

Step-01:

- Draw a table say 'T' with (n+1) number of rows and (w+1) number of columns.
- Fill all the boxes of 0th row and 0th column with zeroes as shown-



	0	1	2	3		W
0	0	0	0	0	……	0
1	0					
2	0					
	……					
n	0					

T-Table

Step-02:

Start filling the table row wise top to bottom from left to right.
Use the following formula-

$$T(i, j) = \max \{ T(i-1, j), value_i + T(i-1, j - weight_i) \}$$

- Here, T(i , j) = maximum value of the selected items if we can take items 1 to i and have weight restrictions of j.
- This step leads to completely filling the table.
- Then, value of the last box represents the maximum possible value that can be put into the knapsack.

Step-03:

- To identify the items that must be put into the knapsack to obtain that maximum profit,
- Consider the last column of the table.
- Start scanning the entries from bottom to top.
- On encountering an entry whose value is not same as the value stored in the entry immediately above it, mark the row label of that entry.
- After all the entries are scanned, the marked labels represent the items that must be put into the knapsack.

Time Complexity-

- Each entry of the table requires constant time θ(1) for its computation.
- It takes θ(nw) time to fill (n+1)(w+1) table entries.
- It takes θ(n) time for tracing the solution since tracing process traces the n rows.

- Thus, overall θ(nw) time is taken to solve 0/1 knapsack problem using dynamic programming.

Example of 0/1 KNAPSACK PROBLEM:-

For the given set of items and knapsack capacity = 5 kg, find the optimal solution for the 0/1 knapsack problem making use of dynamic programming approach.

Item	Weight	Value
1	2	3
2	3	4
3	4	5
4	5	6

(OR)

Find the optimal solution for the 0/1 knapsack problem making use of dynamic programming approach. Consider-

n = 4

w = 5 kg

(w1, w2, w3, w4) = (2, 3, 4, 5)

(b1, b2, b3, b4) = (3, 4, 5, 6)

(OR)

A thief enters a house for robbing it. He can carry a maximal weight of 5 kg into his bag. There are 4 items in the house with the following weights and values. What items should thief take if he either takes the item completely or leaves it completely?

Item	Weight (kg)	Value ($)
Mirror	2	3
Silver nugget	3	4
Painting	4	5
Vase	5	6

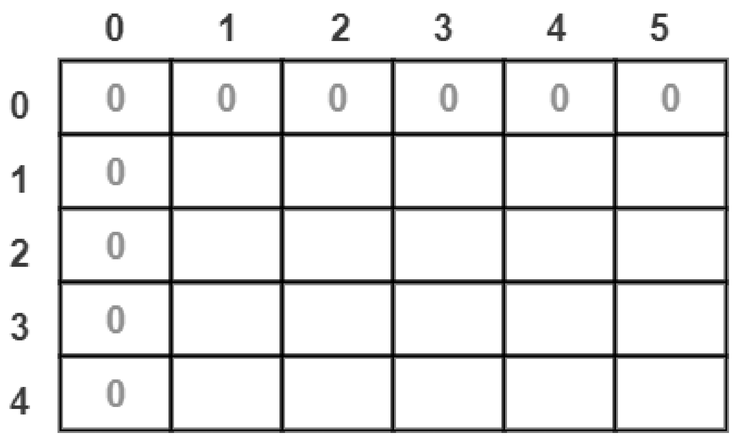
Solution-

Given-

- Knapsack capacity (w) = 5 kg
- Number of items (n) = 4

Step-01:

- Draw a table say 'T' with (n+1) = 4 + 1 = 5 number of rows and (w+1) = 5 + 1 = 6 number of columns.
- Fill all the boxes of 0th row and 0th column with 0.



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

T-Table

Step-02:

Start filling the table row wise top to bottom from left to right using the formula-

$$\mathbf{T(i, j) = \max\{ T(i-1, j), value_i + T(i-1, j - weight_i) \}}$$

Finding T(1,1)-

We have,

- i = 1
- j = 1
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

T(1,1) = max { T(1-1 , 1) , 3 + T(1-1 , 1-2) }

T(1,1) = max { T(0,1) , 3 + T(0,-1) }

T(1,1) = T(0,1) { Ignore T(0,-1) }

T(1,1) = 0

Finding T(1,2)-

We have,

- i = 1
- j = 2
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-

T(1,2) = max { T(1-1 , 2) , 3 + T(1-1 , 2-2) }

T(1,2) = max { T(0,2) , 3 + T(0,0) }
T(1,2) = max {0 , 3+0}
T(1,2) = 3

Finding T(1,3)-

We have,

- i = 1
- j = 3
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-
T(1,3) = max { T(1-1 , 3) , 3 + T(1-1 , 3-2) }
T(1,3) = max { T(0,3) , 3 + T(0,1) }
T(1,3) = max {0 , 3+0}
T(1,3) = 3

Finding T(1,4)-

We have,

- i = 1
- j = 4
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-
T(1,4) = max { T(1-1 , 4) , 3 + T(1-1 , 4-2) }
T(1,4) = max { T(0,4) , 3 + T(0,2) }
T(1,4) = max {0 , 3+0}
T(1,4) = 3

Finding T(1,5)-

We have,

- i = 1
- j = 5
- $(value)_i = (value)_1 = 3$
- $(weight)_i = (weight)_1 = 2$

Substituting the values, we get-
T(1,5) = max { T(1-1 , 5) , 3 + T(1-1 , 5-2) }
T(1,5) = max { T(0,5) , 3 + T(0,3) }
T(1,5) = max {0 , 3+0}
T(1,5) = 3

Finding T(2,1)-

We have,

- i = 2
- j = 1
- $(value)_i = (value)_2 = 4$

- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,1) = max { T(2-1 , 1) , 4 + T(2-1 , 1-3) }

T(2,1) = max { T(1,1) , 4 + T(1,-2) }

T(2,1) = T(1,1) { Ignore T(1,-2) }

T(2,1) = 0

Finding T(2,2)-

We have,

- i = 2
- j = 2
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,2) = max { T(2-1 , 2) , 4 + T(2-1 , 2-3) }

T(2,2) = max { T(1,2) , 4 + T(1,-1) }

T(2,2) = T(1,2) { Ignore T(1,-1) }

T(2,2) = 3

Finding T(2,3)-

We have,

- i = 2
- j = 3
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,3) = max { T(2-1 , 3) , 4 + T(2-1 , 3-3) }

T(2,3) = max { T(1,3) , 4 + T(1,0) }

T(2,3) = max { 3 , 4+0 }

T(2,3) = 4

Finding T(2,4)-

We have,

- i = 2
- j = 4
- $(value)_i = (value)_2 = 4$
- $(weight)_i = (weight)_2 = 3$

Substituting the values, we get-

T(2,4) = max { T(2-1 , 4) , 4 + T(2-1 , 4-3) }

T(2,4) = max { T(1,4) , 4 + T(1,1) }

T(2,4) = max { 3 , 4+0 }

T(2,4) = 4

Finding T(2,5)-

We have,

- i = 2
- j = 5
- $(\text{value})_i = (\text{value})_2 = 4$
- $(\text{weight})_i = (\text{weight})_2 = 3$

Substituting the values, we get-

T(2,5) = max { T(2-1 , 5) , 4 + T(2-1 , 5-3) }

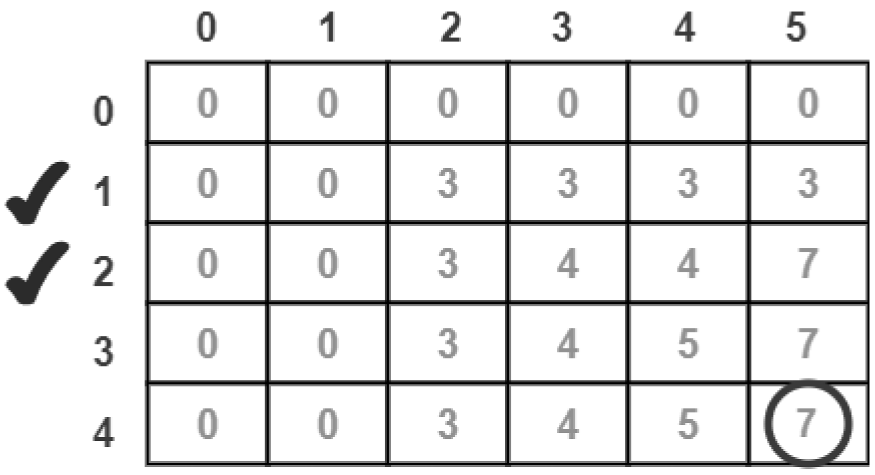
T(2,5) = max { T(1,5) , 4 + T(1,2) }

T(2,5) = max { 3 , 4+3 }

T(2,5) = 7

Similarly, compute all the entries.

After all the entries are computed and filled in the table, we get the following table-



		0	1	2	3	4	5
	0	0	0	0	0	0	0
✓	1	0	0	3	3	3	3
✓	2	0	0	3	4	4	7
	3	0	0	3	4	5	7
	4	0	0	3	4	5	7

T-Table

Step-03

- The last entry represents the maximum possible value that can be put into the knapsack.
- So, maximum possible value that can be put into the knapsack = 7.

Following **Step-04,**

- We mark the rows labeled “1” and “2”.
- Thus, items that must be put into the knapsack to obtain the maximum value 7 are-

Item-1 and Item-2

All Pairs Shortest Path algorithm (or) Floyd-Warshall Algorithm

All Pairs Shortest Path Algorithm is also known as the Floyd-Warshall algorithm. And this is an optimization problem that can be solved using dynamic programming.

Let G = <V, E> be a directed graph, where V is a set of vertices and E is a set of edges with nonnegative length. Find the shortest path between each pair of nodes.

L = Matrix, which gives the length of each edge

L[i, j] = 0, if i == j // Distance of node from itself is zero

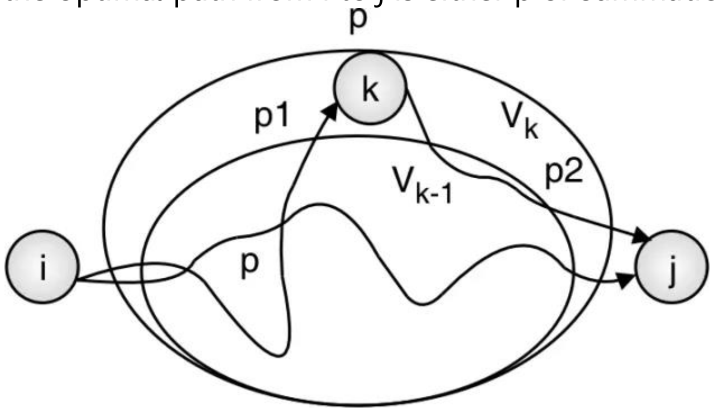
L[i, j] = ∞ , if $i \neq j$ and $(i, j) \notin E$

L[i, j] = w (i, j), if $i \neq j$ and $(i, j) \in E$ // w(i, j) is the weight of the edge (i, j)

Principle of optimality:

If k is the node on the shortest path from i to j, then the path from i to k and k to j, must also be shortest.

In the following figure, the optimal path from i to j is either p or summation of p_1 and p_2.



While considering kth vertex as intermediate vertex, there are two possibilities :

- If k is not part of shortest path from i to j, we keep the distance D[i, j] as it is.
- If k is part of shortest path from i to j, update distance D[i, j] as D[i, k] + D[k, j].

Optimal sub structure of the problem is given as :

$$D^k [i, j] = \min\{ D^{k-1} [i, j], D^{k-1} [i, k] + D^{k-1} [k, j] \}$$

D^k = Distance matrix after kth iteration

Algorithm for All Pairs Shortest Path

This approach is also known as the **Floyd-warshall** shortest path algorithm. The algorithm for all pair shortest path (APSP) problem is described below

```
Algorithm FLOYD_APSP ( L)
 n representing original graph// L is the matrix of size n
// D is the distance matrix
D ← L
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j]k ← min ( D[i, j]k-1, D[i, k]k-1 + D[k, j]k-1 )
        end
    end
end
return D
```

Complexity analysis of All Pairs Shortest Path Algorithm

It is very simple to derive the complexity of all pairs' shortest path problem from the above algorithm. It uses three nested loops. The innermost loop has only one statement. The complexity of that statement is O(1).

The running time of the algorithm is computed as :

$$T(n) = \sum_{k=1}^{n} \sum_{i=1}^{n} \sum_{j=1}^{n} \Theta(1) = \sum_{k=1}^{n} \sum_{i=1}^{n} n = \sum_{k=1}^{n} n^2 = O(n^3)$$

Example

Problem: Apply Floyd's method to find the shortest path for the below-mentioned all pairs.

$$\begin{array}{c|cccc|} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & \infty & 3 & \infty \\ 2 & 2 & 0 & \infty & \infty \\ 3 & \infty & 7 & 0 & 1 \\ 4 & 6 & \infty & \infty & 0 \end{array}$$

Solution:

Optimal substructure formula for Floyd's algorithm,

$D^k [i, j] = \min \{ D^{k-1} [i, j], D^{k-1} [i, k] + D^{k-1} [k, j] \}$

D^o =

	1	2	3	4
1	0	∞	3	∞
2	2	0	∞	∞
3	∞	7	0	1
4	6	∞	∞	0

Iteration 1 : k = 1 :

$D^1[1, 2] = \min \{ D^o [1, 2], D^o [1, 1] + D^o [1, 2] \}$
$= \min \{\infty, 0 + \infty\}$
$= \infty$
$D^1[1, 3] = \min \{ D^o [1, 3], D^o [1, 1] + D^o [1, 3] \}$
$= \min \{3, 0 + 3\}$
$= 3$
$D^1[1, 4] = \min \{ D^o [1, 4], D^o [1, 1] + D^o [1, 4] \}$
$= \min \{\infty, 0 + \infty\}$
$= \infty$
$D^1[2, 1] = \min \{ D^o [2, 1], D^o [2, 1] + D^o [1, 1] \}$
$= \min \{2, 2 + 0\}$
$= 2$
$D^1[2, 3] = \min \{ D^o [2, 3], D^o [2, 1] + D^o [1, 3] \}$
$= \min \{\infty, 2 + 3\}$
$= 5$
$D^1[2, 4] = \min \{ D^o [2, 4], D^o [2, 1] + D^o [1, 4] \}$
$= \min \{\infty, 2 + \infty\}$
$= \infty$
$D^1[3, 1] = \min \{ D^o [3, 1], D^o [3, 1] + D^o [1, 1] \}$
$= \min \{\infty, 0 + \infty\}$
$= \infty$
$D^1[3, 2] = \min \{ D^o [3, 2], D^o [3, 1] + D^o [1, 2] \}$
$= \min \{7, \infty + \infty\}$
$= 7$
$D^1[3, 4] = \min \{ D^o [3, 4], D^o [3, 1] + D^o [1, 4] \}$
$= \min \{1, \infty + \infty\}$

= 1

$D^1[4, 1] = \min \{ D^o [4, 1], D^o [4, 1] + D^o [1, 1] \}$

$= \min \{6, 6 + 0\}$

$= 6$

$D^1[4, 2] = \min \{ D^o [4, 2], D^o [4, 1] + D^o [1, 2] \}$

$= \min \{\infty, 6 + \infty\}$

$= \infty$

$D^1[4, 3] = \min \{ D^o [4, 3], D^o [4, 1] + D^o [1, 3] \}$

$= \min \{\infty, 6 + 3\}$

$= 9$

$$D^1 = \begin{array}{c|c|c|c|c|}
 & 1 & 2 & 3 & 4 \\ \hline
1 & 0 & \infty & 3 & \infty \\ \hline
2 & 2 & 0 & 5 & \infty \\ \hline
3 & \infty & 7 & 0 & 1 \\ \hline
4 & 6 & \infty & 9 & 0 \\ \hline
\end{array}$$

Note : Path distance for highlighted cell is improvement over original matrix.

Iteration 2 (k = 2) :

$D^2[1, 2] = D^1 [1, 2] = \infty$

$D^2[1, 3] = \min \{ D^1 [1, 3], D^1 [1, 2] + D^1 [2, 3] \}$

$= \min \{3, \infty + 5\}$

$= 3$

$D^2[1, 4] = \min \{ D^1 [1, 4], D^1 [1, 2] + D^1 [2, 4] \}$

$= \min \{\infty, \infty + \infty\}$

$= \infty$

$D^2[2, 1] = D^1 [2, 1] = 2$

$D^2[2, 3] = D^1 [2, 3] = 5$

$D^2[2, 4] = D^1 [2, 4] = \infty$

$D^2[3, 1] = \min \{ D^1 [3, 1], D^1 [3, 2] + D^1 [2, 1] \}$

$= \min \{\infty, 7 + 2\}$

$= 9$

$D^2[3, 2] = D^1 [3, 2] = 7$

$D^2[3, 4] = \min \{ D^1 [3, 4], D^1 [3, 2] + D^1 [2, 4] \}$

$= \min \{1, 7 + \infty\}$

$= 1$

$D^2[4, 1] = \min \{ D^1 [4, 1], D^1 [4, 2] + D^1 [2, 1] \}$

$= \min \{6, \infty + 2\}$

$= 6$

$D^2[4, 2] = D^1 [4, 2] = \infty$

$D^2[4, 3] = \min \{ D^1 [4, 3], D^1 [4, 2] + D^1 [2, 3] \}$

$= \min \{9, \infty + 5\}$

$= 9$

$$D^2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & \infty & 3 & \infty \\ 2 & 2 & 0 & 5 & \infty \\ 3 & 9 & 7 & 0 & 1 \\ 4 & 6 & \infty & 9 & 0 \end{array}$$

Iteration 3 (k = 3) :

$D^3[1, 2] = \min \{ D^2 [1, 2], D^2 [1, 3] + D^2 [3, 2] \}$
$= \min \{\infty, 3 + 7\}$
$= 10$
$D^3[1, 3] = D^2 [1, 3] = 3$
$D^3[1, 4] = \min \{ D^2 [1, 4], D^2 [1, 3] + D^2 [3, 4] \}$
$= \min \{\infty, 3 + 1\}$
$= 4$
$D^3[2, 1] = \min \{ D^2 [2, 1], D^2 [2, 3] + D^2 [3, 1] \}$
$= \min \{2, 5 + 9\}$
$= 2$
$D^3[2, 3] = D^2 [2, 3] = 5$
$D^3[2, 4] = \min \{ D^2 [2, 4], D^2 [2, 3] + D^2 [3, 4] \}$
$= \min \{\infty, 5 + 1\}$
$= 6$
$D^3[3, 1] = D^2 [3, 1] = 9$
$D^3[3, 2] = D^2 [3, 2] = 7$
$D^3[3, 4] = D^2 [3, 4] = 1$
$D^3[4, 1] = \min \{ D^2 [4, 1], D^2 [4, 3] + D^2 [3, 1] \}$
$= \min \{6, 9 + 9\}$
$= 6$
$D^3[4, 2] = \min \{ D^2 [4, 1], D^2 [4, 3] + D^2 [3, 2] \}$
$= \min \{\infty, 9 + 7\}$
$= 16$
$D^3[4, 3] = D^2 [4, 3] = 9$

$$D^3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 10 & 3 & 4 \\ 2 & 2 & 0 & 5 & 6 \\ 3 & 9 & 7 & 0 & 1 \\ 4 & 6 & 16 & 9 & 0 \end{array}$$

Iteration 4 (k = 4) :

$D^4[1, 2] = \min \{ D^3 [1, 2], D^3 [1, 4] + D^3 [4, 2] \}$
$= \min \{10, 4 + 16\}$
$= 10$

$D^4[1, 3] = \min \{ D^3 [1, 3], D^3 [1, 4] + D^3 [4, 1] \}$
$= \min \{3, 4 + 9\}$
$= 3$
$D^4[1, 4] = D^3 [1, 4] = 4$
$D^4[2, 1] = \min \{ D^3 [2, 1], D^3 [2, 4] + D^3 [4, 1] \}$
$= \min \{2, 6 + 6\}$
$= 2$
$D^4[2, 3] = \min \{ D^3 [2, 3], D^3 [2, 4] + D^3 [4, 3] \}$
$= \min \{5, 6 + 9\}$
$= 5$
$D^4[2, 4] = D^3 [2, 4] = 6$
$D^4[3, 1] = \min \{ D^3 [3, 1], D^3 [3, 4] + D^3 [4, 1] \}$
$= \min \{9, 1 + 6\}$
$= 7$
$D^4[3, 2] = \min \{ D^3 [3, 2], D^3 [3, 4] + D^3 [4, 2] \}$
$= \min \{7, 1 + 16\}$
$= 7$
$D^4[3, 4] = D^3 [3, 4] = 1$
$D^4[4, 1] = D^3 [4, 1] = 6$
$D^4[4, 2] = D^3 [4, 2] = 16$
$D^4[4, 3] = D^3 [4, 3] = 9$

Final distance matrix is,

$$D^4 = \begin{array}{|c|c|c|c|} \hline 0 & 10 & 3 & 4 \\ \hline 2 & 0 & 5 & 6 \\ \hline 7 & 7 & 0 & 1 \\ \hline 6 & 16 & 9 & 0 \\ \hline \end{array}$$

Traveling Salesman Problem

- Traveling salesman problem is stated as, “Given a set of *n* cities and distance between each pair of cities, find the minimum length path such that it covers each city exactly once and terminates the tour at starting city.”
- It is not difficult to show that this problem is NP complete problem. There exists *n!* paths, a search of the optimal path becomes very slow when *n* is considerably large.
- Each edge (u, v) in TSP graph is assigned some non-negative weight, which represents the distance between city u and v. This problem can be solved by finding the Hamiltonian cycle of the graph.
- The distance between cities is best described by the weighted graph, where edge (u, v) indicates the path from city u to v and w(u, v) represents the distance between cities u and v.
- Let us formulate the solution of TSP using dynamic programming.

Algorithm for Traveling salesman problem

Step 1:

Let d[i, j] indicates the distance between cities i and j. Function C[x, V – { x }]is the cost of the path starting from city x. V is the set of cities/vertices in given graph. The aim of TSP is to minimize the cost function.

Step 2:

Assume that graph contains n vertices V1, V2, ..., Vn. TSP finds a path covering all vertices exactly once, and the same time it tries to minimize the overall traveling distance.

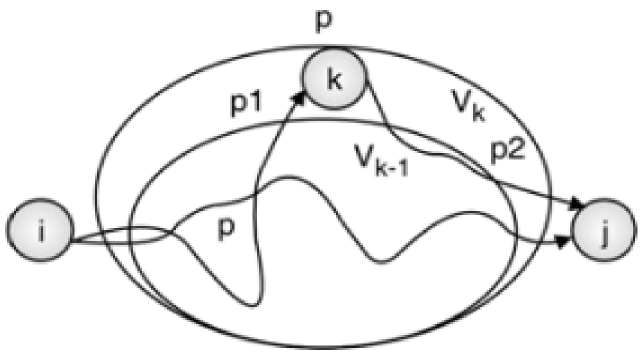
Step 3:

Mathematical formula to find minimum distance is stated below:

C(i, V) = min { d[i, j] + C(j, V – { j }) }, j ∈ V and i ∉ V.

TSP problem possesses the principle of optimality, i.e. for d[V1, Vn] to be minimum, any intermediate path (Vi, Vj) must be minimum.

- From following figure, d[i, j] = min(d[i, j], d[i, k] + d[k, j])
- Dynamic programming always selects the path which is minimum.



Complexity Analysis of Traveling salesman problem

Dynamic programming creates n.2^n sub problems for n cities. Each sub-problem can be solved in linear time. Thus the time complexity of TSP using dynamic programming would be O($n^2 2^n$). It is much less than n! but still, it is an exponent. Space complexity is also exponential.

Example

Problem: Solve the traveling salesman problem with the associated cost adjacency matrix using dynamic programming.

$$\begin{pmatrix} 0 & 24 & 11 & 10 & 9 \\ 8 & 0 & 2 & 5 & 11 \\ 26 & 12 & 0 & 8 & 7 \\ 11 & 23 & 24 & 0 & 6 \\ 5 & 4 & 8 & 11 & 0 \end{pmatrix}$$

Solution:

Let us start our tour from city 1.

Step 1: Initially, we will find the distance between city 1 and city {2, 3, 4, 5} without visiting any intermediate city.

Cost(x, y, z) represents the distance from x to z and y as an intermediate city.

Cost(2, Φ, 1) = d[2, 1] = 24

Cost(3, Φ, 1) = d[3, 1] = 11

Cost(4, Φ , 1) = d[4, 1] = 10

Cost(5, Φ , 1) = d[5, 1] = 9

Step 2: In this step, we will find the minimum distance by visiting 1 city as intermediate city.

Cost{2, {3}, 1} = d[2, 3] + Cost(3, f, 1)
= 2 + 11 = 13
Cost{2, {4}, 1} = d[2, 4] + Cost(4, f, 1)
= 5 + 10 = 15
Cost{2, {5}, 1} = d[2, 5] + Cost(5, f, 1)
= 11 + 9 = 20
Cost{3, {2}, 1} = d[3, 2] + Cost(2, f, 1)
= 12 + 24 = 36
Cost{3, {4}, 1} = d[3, 4] + Cost(4, f, 1)
= 8 + 10 = 18
Cost{3, {5}, 1} = d[3, 5] + Cost(5, f, 1)
= 7 + 9 = 16
Cost{4, {2}, 1} = d[4, 2] + Cost(2, f, 1)
= 23 + 24 = 47
Cost{4, {3}, 1} = d[4, 3] + Cost(3, f, 1)
= 24 + 11 = 35
Cost{4, {5}, 1} = d[4, 5] + Cost(5, f, 1)
= 6 + 9 = 15
Cost{5, {2}, 1} = d[5, 2] + Cost(2, f, 1)
= 4 + 24 = 28
Cost{5, {3}, 1} = d[5, 3] + Cost(3, f, 1)
= 8 + 11 = 19
Cost{5, {4}, 1} = d[5, 4] + Cost(4, f, 1)
= 11 + 10 = 21

Step 3: In this step, we will find the minimum distance by visiting 2 cities as intermediate city.

Cost(2, {3, 4}, 1) = min { d[2, 3] + Cost(3, {4}, 1), d[2, 4] + Cost(4, {3}, 1)]}
= min { [2 + 18], [5 + 35] }
= min{20, 40} = 20
Cost(2, {4, 5}, 1) = min { d[2, 4] + Cost(4, {5}, 1), d[2, 5] + Cost(5, {4}, 1)]}
= min { [5 + 15], [11 + 21] }
= min{20, 32} = 20
Cost(2, {3, 5}, 1) = min { d[2, 3] + Cost(3, {4}, 1), d[2, 4] + Cost(4, {3}, 1)]}
= min { [2 + 18], [5 + 35] }
= min{20, 40} = 20
Cost(3, {2, 4}, 1) = min { d[3, 2] + Cost(2, {4}, 1), d[3, 4] + Cost(4, {2}, 1)]}
= min { [12 + 15], [8 + 47] }
= min{27, 55} = 27
Cost(3, {4, 5}, 1) = min { d[3, 4] + Cost(4, {5}, 1), d[3, 5] + Cost(5, {4}, 1)]}
= min { [8 + 15], [7 + 21] }
= min{23, 28} = 23
Cost(3, {2, 5}, 1) = min { d[3, 2] + Cost(2, {5}, 1), d[3, 5] + Cost(5, {2}, 1)]}
= min { [12 + 20], [7 + 28] }
= min{32, 35} = 32

Cost(4, {2, 3}, 1) = min{ d[4, 2] + Cost(2, {3}, 1), d[4, 3] + Cost(3, {2}, 1)]}
= min { [23 + 13], [24 + 36] }
= min{36, 60} = 36
Cost(4, {3, 5}, 1) = min{ d[4, 3] + Cost(3, {5}, 1), d[4, 5] + Cost(5, {3}, 1)]}
= min { [24 + 16], [6 + 19] }
= min{40, 25} = 25
Cost(4, {2, 5}, 1) = min{ d[4, 2] + Cost(2, {5}, 1), d[4, 5] + Cost(5, {2}, 1)]}
= min { [23 + 20], [6 + 28] }
= min{43, 34} = 34
Cost(5, {2, 3}, 1) = min{ d[5, 2] + Cost(2, {3}, 1), d[5, 3] + Cost(3, {2}, 1)]}
= min { [4 + 13], [8 + 36] }
= min{17, 44} = 17
Cost(5, {3, 4}, 1) = min{ d[5, 3] + Cost(3, {4}, 1), d[5, 4] + Cost(4, {3}, 1)]}
= min { [8 + 18], [11 + 35] }
= min{26, 46} = 26
Cost(5, {2, 4}, 1) = min{ d[5, 2] + Cost(2, {4}, 1), d[5, 4] + Cost(4, {2}, 1)]}
= min { [4 + 15], [11 + 47] }
= min{19, 58} = 19

Step 4 : In this step, we will find the minimum distance by visiting 3 cities as intermediate city.
Cost(2, {3, 4, 5}, 1) = min { d[2, 3] + Cost(3, {4, 5}, 1), d[2, 4] + Cost(4, {3, 5}, 1), d[2, 5] + Cost(5, {3, 4}, 1)}
= min { 2 + 23, 5 + 25, 11 + 36}
= min{25, 30, 47} = 25
Cost(3, {2, 4, 5}, 1) = min { d[3, 2] + Cost(2, {4, 5}, 1), d[3, 4] + Cost(4, {2, 5}, 1), d[3, 5] + Cost(5, {2, 4}, 1)}
= min { 12 + 20, 8 + 34, 7 + 19}
= min{32, 42, 26} = 26
Cost(4, {2, 3, 5}, 1) = min { d[4, 2] + Cost(2, {3, 5}, 1), d[4, 3] + Cost(3, {2, 5}, 1), d[4, 5] + Cost(5, {2, 3}, 1)}
= min {23 + 30, 24 + 32, 6 + 17}
= min{53, 56, 23} = 23
Cost(5, {2, 3, 4}, 1) = min { d[5, 2] + Cost(2, {3, 4}, 1), d[5, 3] + Cost(3, {2, 4}, 1), d[5, 4] + Cost(4, {2, 3}, 1)}
= min {4 + 30, 8 + 27, 11 + 36}
= min{34, 35, 47} = 34

Step 5 : In this step, we will find the minimum distance by visiting 4 cities as an intermediate city.
Cost(1, {2, 3, 4, 5}, 1) = min { d[1, 2] + Cost(2, {3, 4, 5}, 1), d[1, 3] + Cost(3, {2, 4, 5}, 1), d[1, 4] + Cost(4, {2, 3, 5}, 1) , d[1, 5] + Cost(5, {2, 3, 4}, 1)}
= min { 24 + 25, 11 + 26, 10 + 23, 9 + 34 }
= min{49, 37, 33, 43} = 33
Thus, minimum length tour would be of 33.

Trace the path:

- Let us find the path that gives the distance of 33.
- Cost(1, {2, 3, 4, 5}, 1) is minimum due to d[1, 4], so move from 1 to 4. Path = {1, 4}.

- Cost(4, {2, 3, 5}, 1) is minimum due to d[4, 5], so move from 4 to 5. Path = {1, 4, 5}.
- Cost(5, {2, 3}, 1) is minimum due to d[5, 2], so move from 5 to 2. Path = {1, 4, 5, 2}.
- Cost(2, {3}, 1) is minimum due to d[2, 3], so move from 2 to 3. Path = {1, 4, 5, 2, 3}.

All cities are visited so come back to 1. Hence the optimum tour would be 1 – 4 – 5 – 2 – 3 – 1.

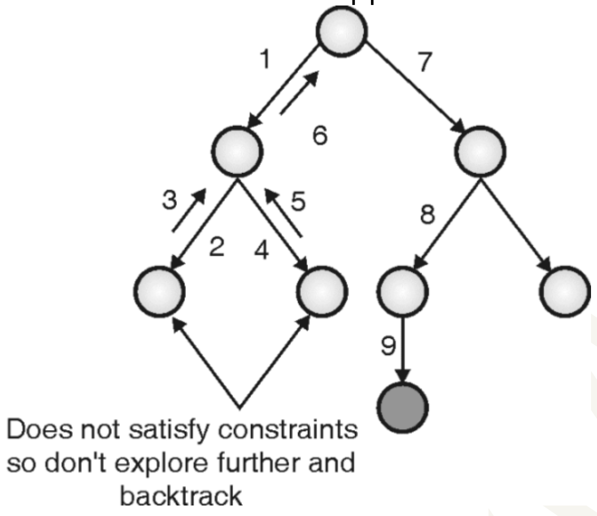
BACK TRACKING

Back tracking:-

Backtracking is an intelligent way of gradually building the solution. Typically, it is applied to constraint satisfaction problems like Sudoku, crossword, 8-queen puzzles, chess, and many other games. Dynamic programming and greedy algorithms are optimization techniques, whereas back tracing is s general problem-solving method. It does not guarantee an optimal solution.

- A solution too many problems can be viewed as the making of a sequence of decisions. For example, TSP can be solved by making the sequence of the decision of which city should be visited next.
- Similarly, a knapsack is also viewed as a sequence of decisions. At each step, the decision to select or reject the given item is made.
- Backtracking builds the solution incrementally. Partial solutions that do not satisfy the constraints are abandoned.
- Backtracking is a recursive approach, and it is a refinement of the brute force method. The brute force approach evaluates all possible candidates, whereas backtracking limits the search by eliminating the candidates that do not satisfy certain constraints. Hence, backtracking algorithms are often faster than brute force approaches.
- Backtracking algorithms are used when we have a set of choices, and we don't know which choice will lead to the correct solution. The algorithm generates all partial candidates that may generate a complete solution.
- The solution in backtracking is expressed as an n-tuple (x1, x2,… xn), where xi is chosen from the finite set of choices, Si. Elements in the solution tuple are chosen such that they maximize or minimize the given criterion function C(x1, x2,…, xn).
- Let C be the set of constraints on problem P, and let D is the set of all the solutions that satisfy the constraints C. Then
    - Finding if a given solution is feasible or not is the **decision problem**.
    - Finding the best solution is the optimization problem.
    - Listing all feasible solutions is an enumeration problem.
- Backtracking systematically searches the set of all feasible solutions, called solution space, to solve the given problem.
- Each choice leads to a new set of partial solutions. Partial solutions are explored in DFS (Depth First Search) order.
- If a partial solution satisfies a certain bounding function, then the partial solution is explored in depth-first order.
- If the partial solution does not satisfy the constraint, it will not be explored further. The algorithm backtracks from that point and explores the next possible candidate.
- Such processing is convenient to represent using a state space tree. In a state space tree, the root represents the initial state before the search begins.

- Figure (a) shows the workings of backtracking algorithms. It keeps exploring the partial solution by adding the next possible choice in the DFS order and building the new partial solution. This process continues till a partial solution satisfies the given constraints or a solution is not found. This is an incremental approach.


1
7
6
3
5
8
2
4
9
Does not satisfy constraints so don't explore further and backtrack

(a). Process of backtracking

- A node in the state-space tree is called **promising** if it represents a partially constructed solution which may lead to a complete solution. **Non-promising** node violates constraints and hence cut down from further computation.
- A leaf node is either a non-promising node or represents a complete solution.

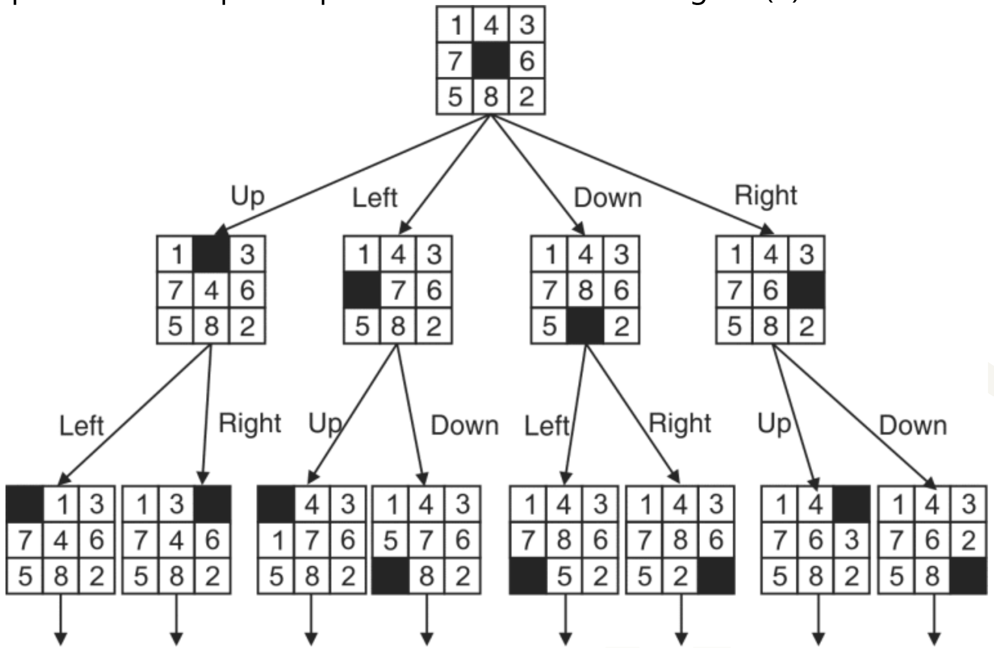
We can consider the backtracking process as finding a particular leaf in the tree. It works as follows:

- If node N is the goal node, then return success and exit.
    - If node is a is leaf node not a goal node then it hen returns failure.
- Otherwise, for each child C of node N,
    - Explore child node C
    - If C is the goal node, return “success”
    - Return failure
- In backtracking, the solution is expressed in form of tuple (x1, x2, …, xn), each xi ∈ Si, where Si is some finite set of choice. The backtracking g algorithm tries to maximize or minimize certain criterion function f(.) by selecting or not selecting item xi.
- While solving the problem, the backtracking method imposes two constraints:
    - **Explicit constraints**: it restricts the section of xi from Si. Explicit constraints are problem-dependent. All from these from solution space must satisfy explicit constraints.
    - **Implicit constraints:** Such constraints determine which tuple in solution space satisfies the criterion function.

Terminology of state space tree

- The **state space** of the system is the set of states in which the system can exist.
- Solution to the problem which is derived by making sequence of decisions can be represented using **state space tree** T.

- The root of the tree is the state before making any decision. Nodes at the first level in the tree corresponding to all possible choices for the first decision.
- Nodes at second level corresponding to all possible choices for the second decision and so forth.
- Usually, number of decision sequence is in exponential order of input size n.
- State space for the 8-puzzle problem is shown in the Figure (b).



(b). State space of the 8-puzzle game

- Backtracking guaranteed to find the solution to any problem modeled by a state space tree.
- Based on traversal order of the tree, two strategies are defined to solve the problem.
- Backtracking traverse the tree in depth-first order
- Branch and bound traverse the tree in breadth-first order
- As backtracking always explores the current node first, there is no need of explicitly implementing the state space tree.

Some terminologies related to state space tree are discussed here:

(1) Solution space : The collection of all feasible solutions is called solution space.

(2) State space : All the paths in the tree from root to other nodes form the state space of the given problem.

(3) Problem state : Each node in the state space tree represents the problem state.

(4) Answer state : Answer state is the leaf in the state space tree which satisfies the implicit constraints.

(5) Solution-state : State S for which the path from the root to S represents a tuple in the solution space is called solution state.

(6) Live node : In state space tree, a node that is generated but its children are yet to be generated is called a live node.

(7) E-Node : Live node whose children are currently being explored is called E (Expansion) node.

(8) Dead node : In state space tree, a node that is generated and either it's all children are generated or node will not be expanded further due to a violation of criterion function, is called a dead node.

(9) Bounding Functions : The bounding function kills the live node without exploring its children if the bound value of the live node crosses the bound limits.

(10) Static tree : If a tree is independent of an instance of the problem being solved, it is called a static tree

(11) Dynamic tree : If the tree depends on an instance of the problem being solved, it is called a dynamic tree.

APPLICATIONS OF BACKTRACKING

Backtracking is useful in solving the following problems:

- **N-Queen problem**
- Sum of subset problem
- **Graph coloring problem**
- Knapsack problem
- **Hamiltonian cycle problem**
- Games like chess, tic-tac-toe, Sudoku etc.
- Constraint satisfaction problems
- Artificial intelligence
- Network communication
- Robotics
- Optimization problems
- It is also a basis of a logic programming language called PROLOG.

N-QUEENS PROBLEM

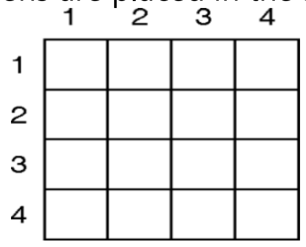
What is N Queen Problem?

N Queen Problem is the classical Example of backtracking. N-Queen problem is defined as, "given N x N chess board, arrange N queens in such a way that no two queens attack each other by being in same row, column or diagonal".

- For N = 1, this is trivial case. For N = 2 and N = 3, solution is not possible. So we start with N = 4 and we will generalize it for N queens.

4-Queen Problem

Problem: Given 4 x 4 chess board, arrange four queens in a way, such that no two queens attack each other. That is, no two queens are placed in the same row, column, or diagonal.



4 x 4 Chessboard

- We have to arrange four queens, Q1, Q2, Q3 and Q4 in 4 x 4 chess board. We will put ith queen in ith row. Let us start with position (1, 1). Q1 is the only queen, so there is no issue. partial solution is <1>
- We cannot place Q2 at positions (2, 1) or (2, 2). Position (2, 3) is acceptable. partial solution is <1, 3>.
- Next, Q3 cannot be placed in position (3, 1) as Q1 attacks her. And it cannot be placed at (3, 2), (3, 3) or (3, 4) as Q2 attacks her. There is no way to put Q3 in third row. Hence, the

algorithm backtracks and goes back to the previous solution and readjusts the position of queen Q2. Q2 is moved from positions (2, 3) to
(2, 4). Partial solution is <1, 4>

- Now, Q3 can be placed at position (3, 2). Partial solution is <1, 4, 3>.
- Queen Q4 cannot be placed anywhere in row four. So again, backtrack to the previous solution and readjust the position of Q3. Q3 cannot be placed on (3, 3) or(3, 4). So the algorithm backtracks even further.
- All possible choices for Q2 are already explored, hence the algorithm goes back to partial solution <1> and moves the queen Q1 from (1, 1) to (1, 2). And this process continues until a solution is found.
- All possible solutions for 4-queen are shown in fig (a) & fig. (b)

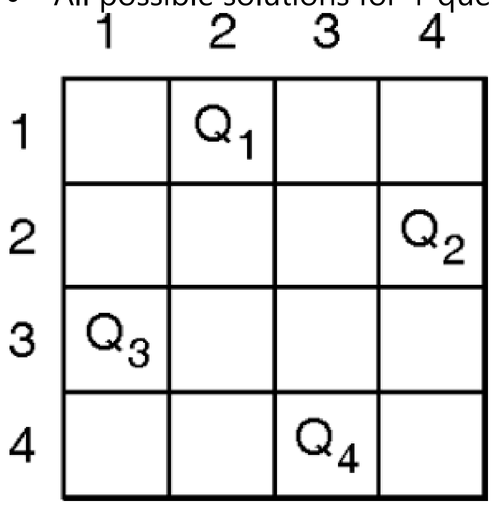


Fig. (a): Solution – 1

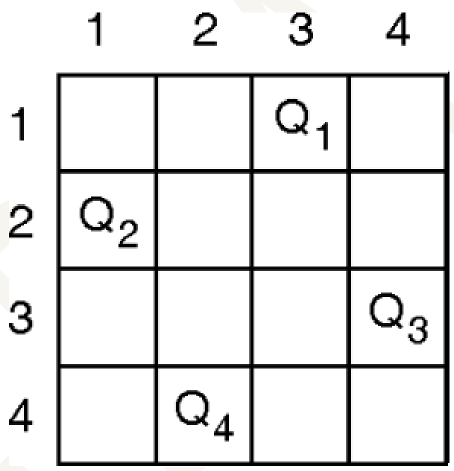


Fig. (b): Solution – 2

We can see that backtracking is a simple brute force approach, but it applies some intelligence to cut out unnecessary computation. The solution tree for the 4-queen problem is shown in Fig. (c).

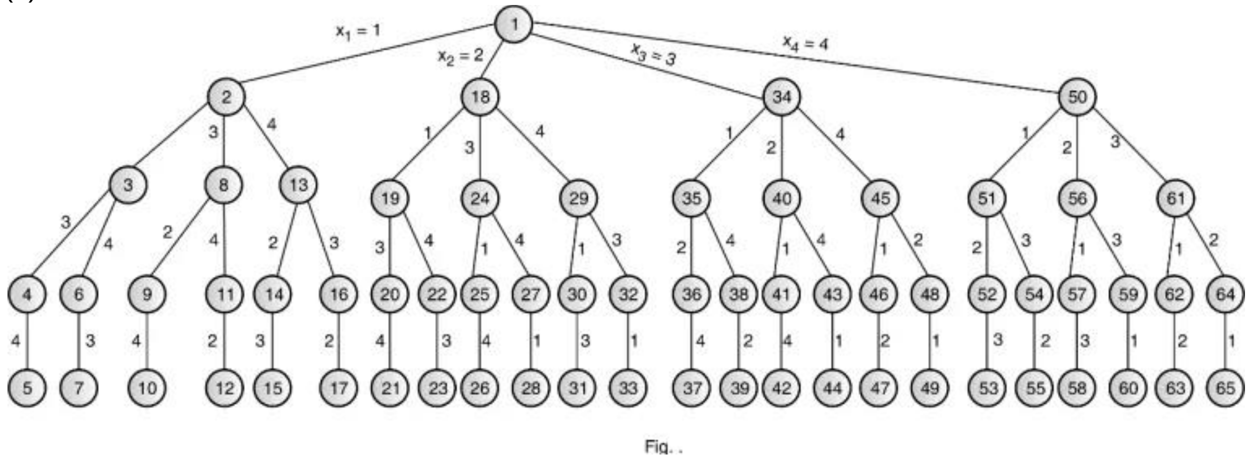


Fig. (c): State-space diagram for 4 – queen problem

Fig. (d) describes the backtracking sequence for the 4-queen problem.

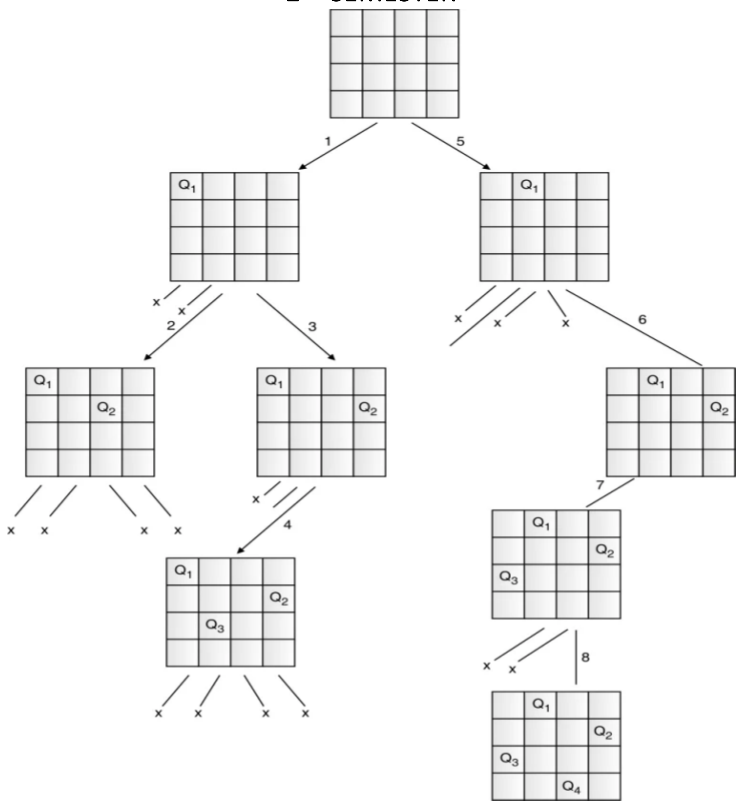


Fig. (d): Backtracking sequence for 4-queen

The solution of the 4-queen problem can be seen as four tuples (x_1, x_2, x_3, x_4), where x_i represents the column number of queen Q_i. Two possible solutions for the 4-queen problem are (2, 4, 1, 3) and (3, 1, 4, 2).

8-Queen Problem

Problem: Given an 8 x 8 chessboard, arrange 8 queens in a way such that no two queens attack each other.

Two queens are attacking each other if they are in the same row, column, or diagonal. Cells attacked by queen Q are shown in fig. (e).

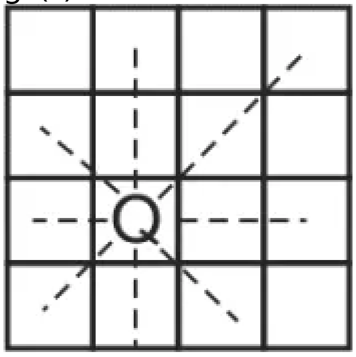


Fig. (e): Attacked cells by queen Q

- 8 queen problem has ${}^{64}C_8$= 4,42,61,65,368 different arrangements. Of these, only 92 arrangements are valid solutions. Out of which, only 12 are the fundamental solutions. The remaining 80 solutions can be generated using reflection and rotation.
- The 2-queen problem is not feasible. The minimum problem size for which a solution can be found is 4. Let us understand the workings of backtracking on the 4-queen problem.
- For simplicity, a partial state space tree is shown in fig. (f). Queen 1 is placed in the first column in the first row. All the positions are crossed in which Queen 1 is attacking. In the

next level, queen 2 is placed in a 3rd column in row 2 and all cells that are crossed are attacked by already placed queens 1 and 2. As can be seen from fig (f), no place is left to place the next queen in row 3, so queen 2 backtracks to the next possible position and the process continues. In a similar way, if (1, 1) position is not feasible for queen 1, then the algorithm backtracks and puts the first queen in cell (1, 2), and repeats the procedure. For simplicity, only a few nodes are shown in fig. (f).

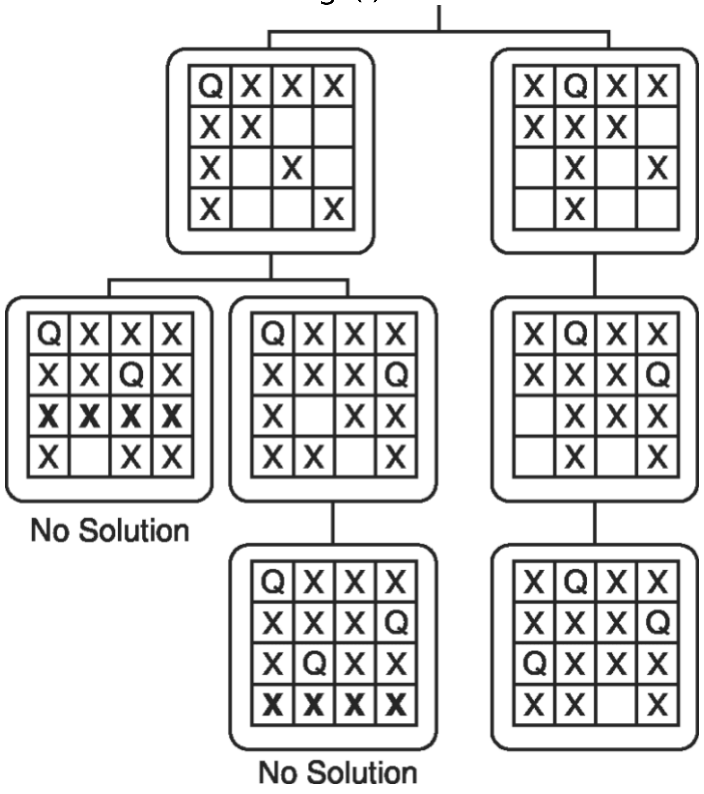


Fig. (f): Snapshot of backtracking procedure of 4-Queen problem

- A complete state space tree for the 4-queen problem is shown in fig. (g)
- The number within the circle indicates the order in which the node gets explored. The height of the e from the t indicates row and label, besides the arc indicating that the Q is placed in an ith column. Out of all the possible states, only a few are the answer states.

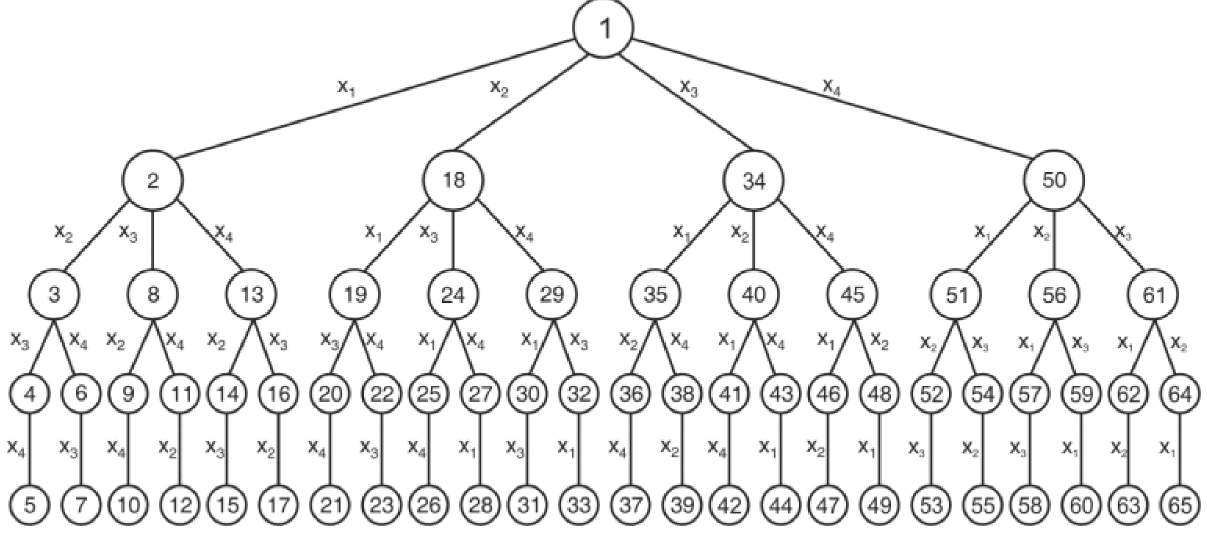


Fig. (g): Solution of 8-queen problem

- Solution tuple for the solution shown in fig (h) is defined as <4, 6, 8, 2, 7, 1, 3, 5>. From observations, two queens placed at (i, j) and (k, l) positions, can be in same diagonal only if,

(i – j) = (k – l) or
(i + j) = (k + l)

From first equality, j – l = i – k
From second equality, j – l = k – i
So queens can be in diagonal only if |j – l| = | i – k|.
The arrangement shown in fig. (i) leads to failure. As it can be seen from fig. (i), Queen Q6 cannot be placed anywhere in the 6th row. So the position of Q5 is backtracked and it is placed in another feasible cell. This process is repeated until the solution is found.

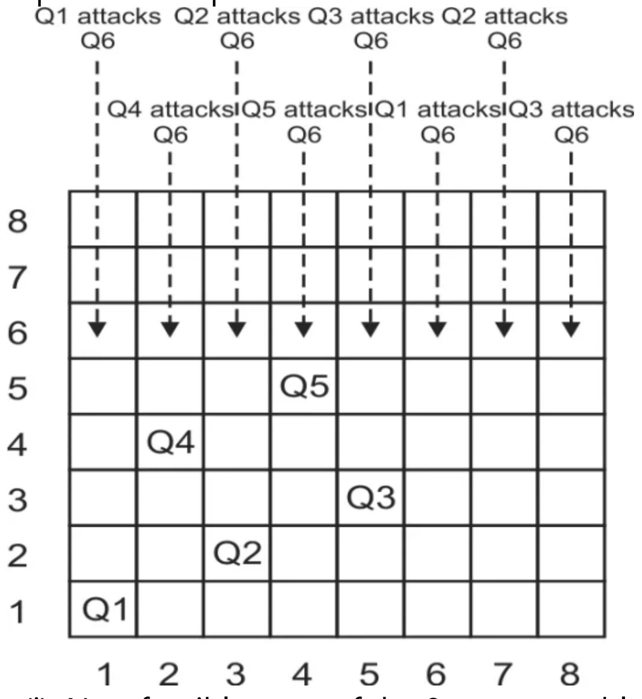


Fig. (i): Non-feasible state of the 8-queen problem

N-Queen Algorithm:-

The following algorithm arranges n queens on n x n board using a backtracking algorithm.

```
Place (k, i)
   {
     For j  ←  1 to k - 1
      do if (x [j] = i)
       or (Abs x [j]) - i) = (Abs (j - k))
    then return false;
     return true;
}
```

Place (k, i) return true if a queen can be placed in the kth row and ith column otherwise return is false.

x [ ] is a global array whose final k - 1 values have been set. Abs (r) returns the absolute value of r.

```
N - Queens (k, n)
{
   For i  ←  1 to n
        do if Place (k, i) then
   {
      x [k]  ←  i;
```

```
        if (k ==n) then
          write (x [1....n));
        else
        N - Queens (k + 1, n);
    }
}
```

Explicit Constraints:

- Explicit constraints are the rules that allow/disallow selection of x_i to take value from the given set. For example, $x_i = 0$ or 1.

  x_i = 1 if $LB_i \leq x_i \leq UB_i$

  x_i = 0 otherwise
- **Solution space** is formed by the collection of all tuple which satisfies the constraint.

Implicit constraints:

The implicit constraint is to determine which of the tuple of solution space satisfies the given criterion functions. The implicit constraint for n queen problem is that two queens must not appear in the same row, column or diagonal.

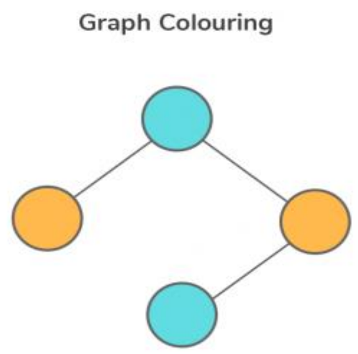
Complexity analysis: In backtracking, at each level branching factor decreases by 1 and it creates a new problem of size (n – 1) . With n choices, it creates n different problems of size (n – 1) at level 1.

- PLACE function determines the position of the queen in O(n) time. This function is called n times.
- Thus, the recurrence of n-Queen problem is defined as, $T(n) = n*T(n – 1) + n^2$. Solution to recurrence would be O(n!).

GRAPH COLORING PROBLEM

Graph coloring problem involves assigning colors to certain elements of a graph subject to certain restrictions and constraints. In other words, the process of assigning colors to the vertices such that no two adjacent vertexes have the same color is caller Graph Coloring. This is also known as **vertex coloring**.

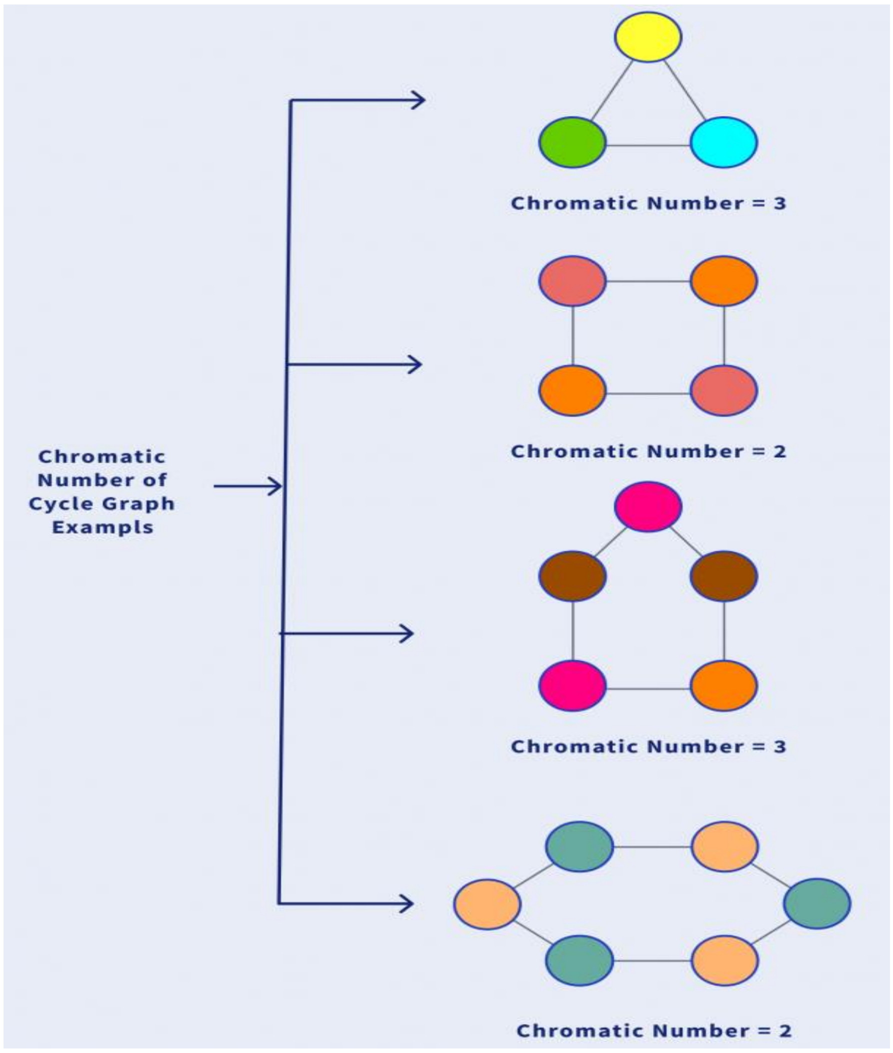
Example:



Chromatic Number: The smallest number of colours needed to colour a graph G is called its chromatic number.

For example, in the above image, vertices can be coloured using a minimum of 2 colours. Hence the **chromatic number** of the graph is 2.

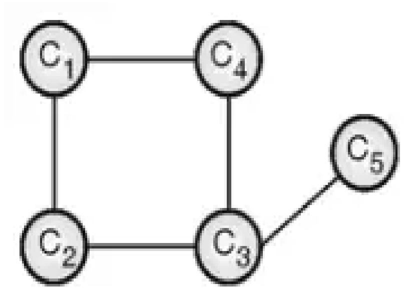
Further examples for a more clear understanding:

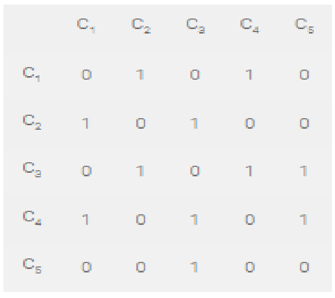


Applications of Graph Coloring Problem

- Design a timetable.
- Sudoku
- Register allocation in the compiler
- Map coloring
- Mobile radio frequency assignment:

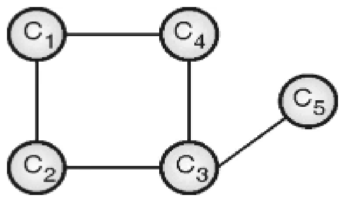
The input to the graph is an adjacency matrix representation of the graph. Value M(i, j) = 1 in the matrix represents there exists an edge between vertex i and j. A graph and its adjacency matrix representation are shown in Figure (a)



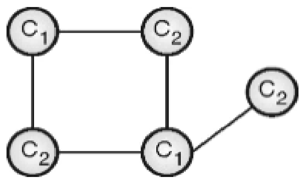


	C_1	C_2	C_3	C_4	C_5
C_1	0	1	0	1	0
C_2	1	0	1	0	0
C_3	0	1	0	1	1
C_4	1	0	1	0	1
C_5	0	0	1	0	0

The problem can be solved simply by assigning a unique color to each vertex, but this solution is not optimal. It may be possible to color the graph with colors less than |V|. Figure (b) and figure (c) demonstrate both such instances. Let Ci denote the ith color.



(b). Nonoptimal solution (uses 5 colors)



(c). Optimal solution (uses 2 colors)

This problem can be solved using **backtracking algorithms** as follows:

- List down all the vertices and colors in two lists
- Assign color 1 to vertex 1
- If vertex 2 is not adjacent to vertex 1 then assign the same color, otherwise assign color 2.
- Repeat the process until all vertices are colored.

Algorithm backtracks whenever color i is not possible to assign to any vertex k and it selects next color i + 1 and test is repeated. Consider the graph shown in Figure (d)

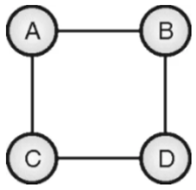


Figure (d)

If we assign color 1 to vertex A, the same color cannot be assigned to vertex B or C. In the next step, B is assigned some different colors 2. Vertex A is already colored and vertex D is a neighbor of B, so D cannot be assigned color 2. The process goes on. State-space tree is shown in Figure (e)

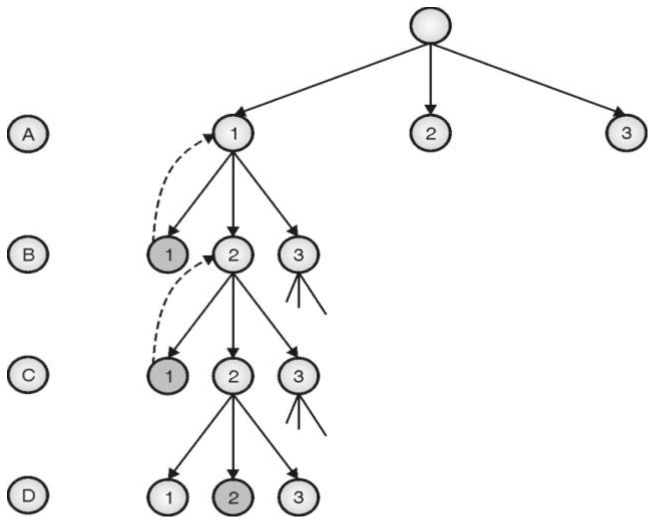


Figure (e): State-space tree of the graph of Figure (d)

Thus, vertices A and C will be colored with color 1, and vertices B and D will be colored with color 2.

Algorithm for graph coloring is described here:

```
 GRAPH_COLORING(G, COLOR, i)
// Description :  Solve the graph coloring problem using backtracking
// Input : Graph G with n vertices, list of colors, initial
vertex i
// COLOR[1…n] is the array of n different colors
// Output : Colored graph with minimum color
if
 CHECK_VERTEX(i) == 1
then
if
i == N
then
print
COLOR[1…n]
else
   j ← 1
while(j ≤ M)
do
      COLOR(i + 1) ← j
      j ← j + 1
  end
end
end
```

Function

```
CHECK_VERTEX(i)
for  j ← 1 to i – 1
do
if
Adjacent(i, j)
then
if
COLOR(i) == COLOR(j)
then
return 0
end
end
end
return 1
```

Complexity Analysis of graph coloring:-

The number of anode increases exponentially at every level in state space tree. With M colors and n vertices, total number of nodes in state space tree would be

$1 + M + M^2 + M^3 + ... + M^n$

Hence, $T(n) = 1 + M + M^2 + M^3 + ... + M^n$

= frac{M^{n+1} - 1}{M - 1}=*fracMn*+1−1*M*−1

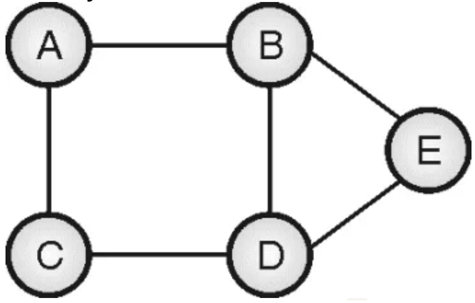
So, $T(n) = O(M^n)$.

Thus, the graph coloring algorithm runs in exponential time.

HAMILTONIAN CYCLE /CIRCUIT PROBLEMS

*The **Hamiltonian cycle** is the cycle in the graph which visits all the vertices in graph exactly once and terminates at the starting node. It may not include all the edges*

- **The** Hamiltonian cycle problem is the problem of finding a Hamiltonian cycle in a graph if there exists any such cycle.
- The input to the problem is an undirected, connected graph. For the graph shown in Figure (a), a path A – B – E – D – C – A forms a Hamiltonian cycle. It visits all the vertices exactly once, but does not visit the edges <B, D>.



- The Hamiltonian cycle problem is also both, decision problem and an optimization problem. A decision problem is stated as, "Given a path, is it a Hamiltonian cycle of the graph?".
- The optimization problem is stated as, "Given graph G, find the Hamiltonian cycle for the graph."
- We can define the constraint for the Hamiltonian cycle problem as follows:
  - In any path, vertex i and (i + 1) must be adjacent.
  - 1st and (n – 1)th vertex must be adjacent (nth of cycle is the initial vertex itself).
  - Vertex i must not appear in the first (i – 1) vertices of any path.
- With the adjacency matrix representation of the graph, the adjacency of two vertices can be verified in constant time.

Algorithm for Hamiltonian Cycle Problem using Backtracking

```
HAMILTONIAN (i)
// Description : Solve Hamiltonian cycle problem using backtracking.
// Input : Undirected, connected graph G = <V, E> and initial vertex i
// Output : Hamiltonian cycle
if
FEASIBLE(i)
then
if(i == n - 1)
then
   Print V[0… n – 1]
else
   j ← 2
while(j ≤ n)
do
     V[i] ← j
     HAMILTONIAN(i + 1)
```

```
    j ← j + 1
end
end
end
```

function

```
FEASIBLE(i)
flag ← 1
for  j ← 1 to i – 1
do
if
Adjacent(Vi, Vj)
then
   flag ← 0
end
end
if
Adjacent (Vi, Vi-1)
then
  flag ← 1
else
  flag ← 0
end
return flag
```

Complexity Analysis

Looking at the state space graph, in worst case, total number of nodes in tree would be,

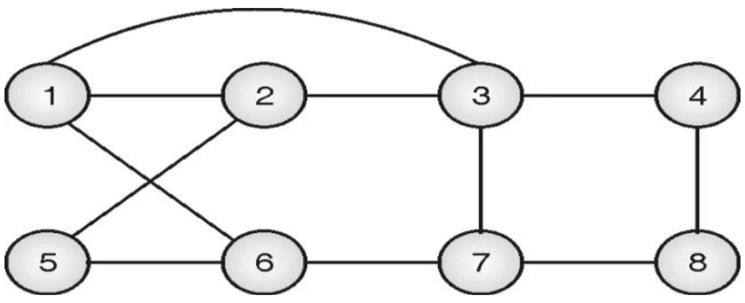
$T(n) = 1 + (n – 1) + (n – 1)^2 + (n – 1)^3 + … + (n – 1)^{n – 1}$

= frac{(n-1)^n - 1}{n - 2}=$frac(n-1)_n-1n-2$

$T(n) = O(n^n)$. Thus, the Hamiltonian cycle algorithm runs in exponential time.

Example

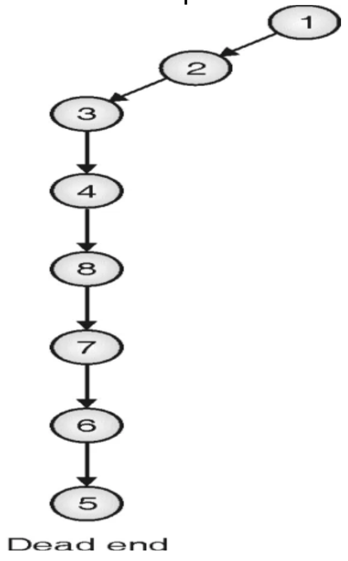
Example: Explain how to find Hamiltonian Cycle by using Backtracking in a given graph



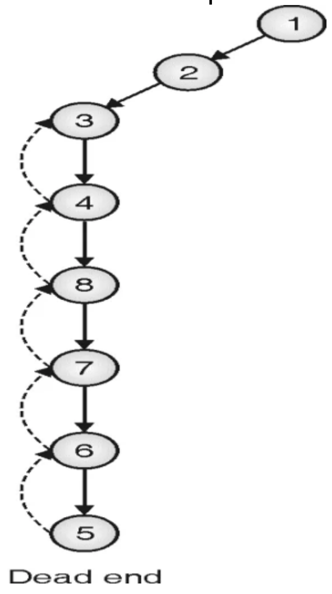
Solution:

The backtracking approach uses a state-space tree to check if there exists a Hamiltonian cycle in the graph. Figure (f) shows the simulation of the Hamiltonian cycle algorithm. For simplicity, we have not explored all possible paths, the concept is self-explanatory.

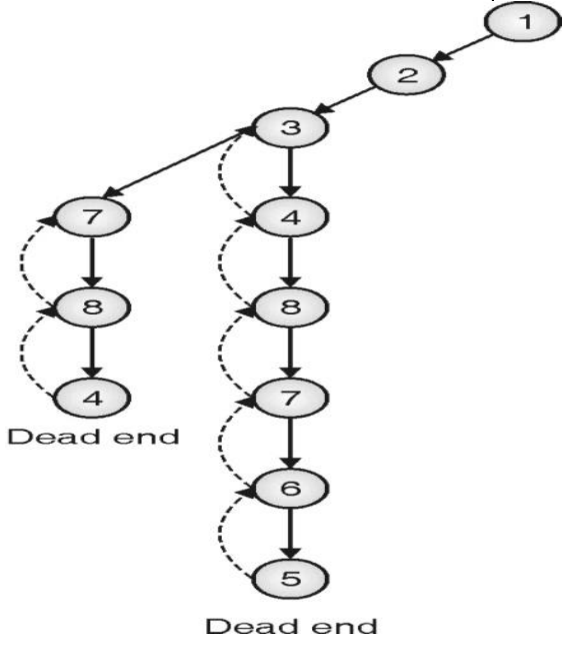
Step 1: Tour is started from vertex 1. There is no path from 5 to 1. So it's the dead-end state.



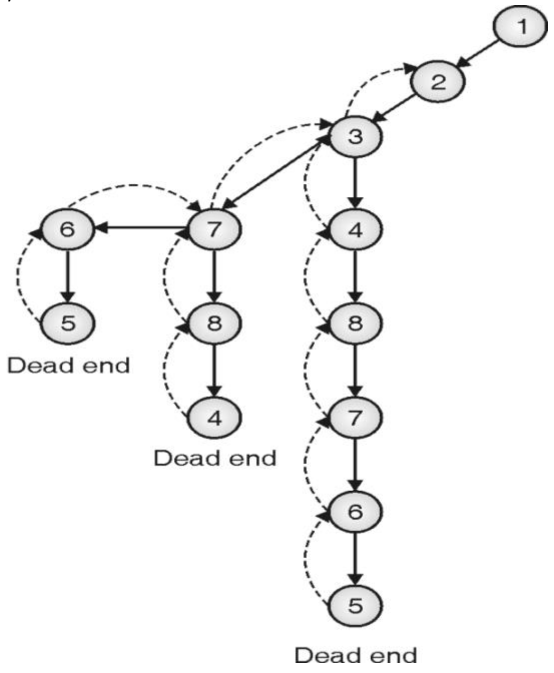
Step 2: Backtrack to the node from where the new path can be explored, that is 3 here



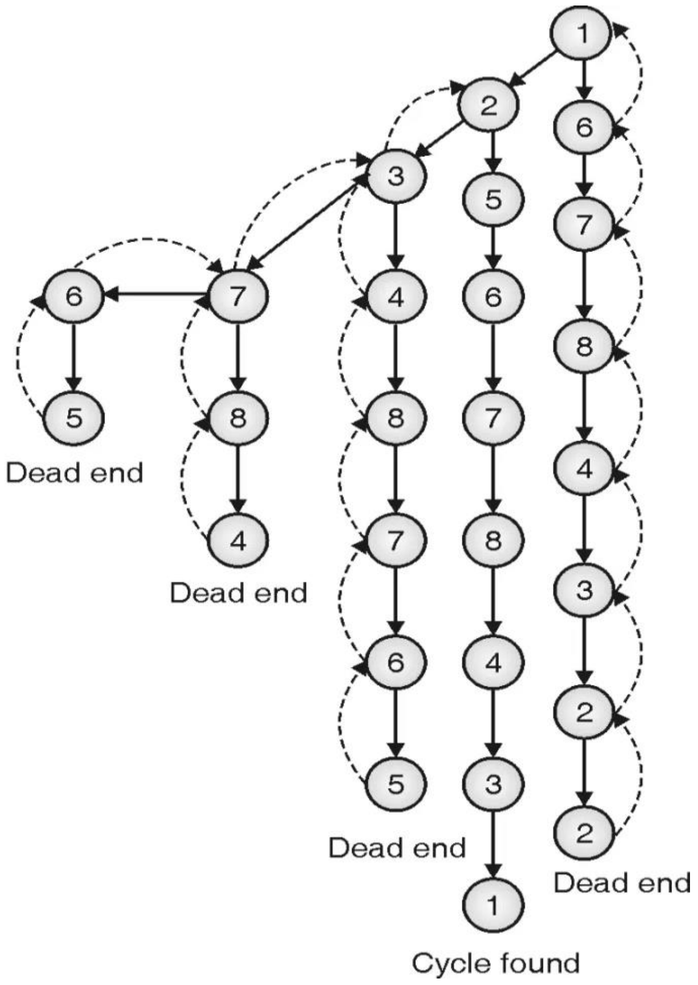
Step 3: New path also leads to a dead end so backtrack and explore all possible paths



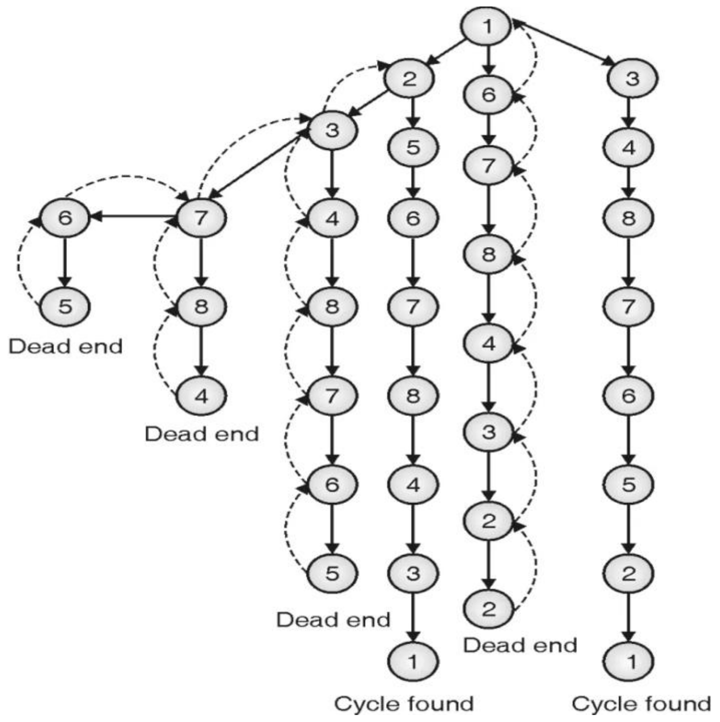
Step 4: Next path is also leading to a dead-end so keep backtracking until we get some node that can generate a new path, i.e .vertex 2 here



Step 5: One path leads to Hamiltonian cycle, next leads to a dead end so backtrack and explore all possible paths at each vertex



Step 6: Total two Hamiltonian cycles are detected in a given graph



Unit-4
Branch and bound

Branch and bound method

The branch and bound technique is used to solve optimization problems, whereas the backtracking method is used to solve decision problems.

- Branch and bound build the state space tree, and find the optimal solution quickly by pruning a few of the branches of the tree that do not satisfy the bound.
- Branch and bound build the state space tree, one can find the optimal solution quickly by pruning a few of the tree branches that do not satisfy the bound.
- Branch and bound can be useful where some other optimization techniques like greedy or dynamic programming fail. Such algorithms are typically slower than their counterparts. In the worst case, it may run in exponential time, but careful selection of bounds and branches makes an algorithm run reasonably faster.

Definition:-

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

Example

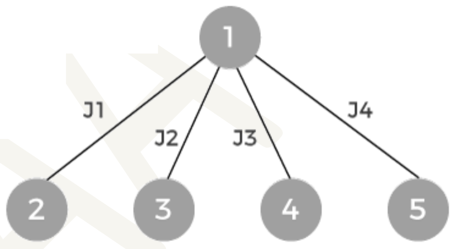
Jobs = {j1, j2, j3, j4}

P = {10, 5, 8, 3}

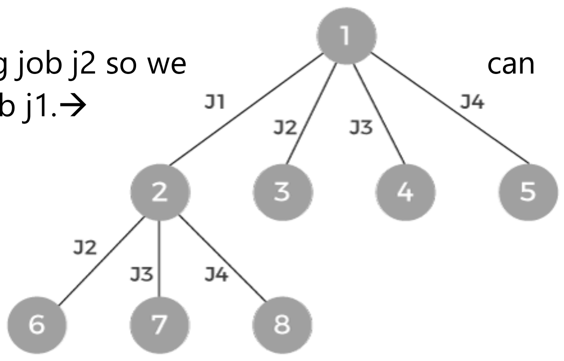
d = {1, 2, 1, 2}

The above are jobs, problems and problems given. 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.

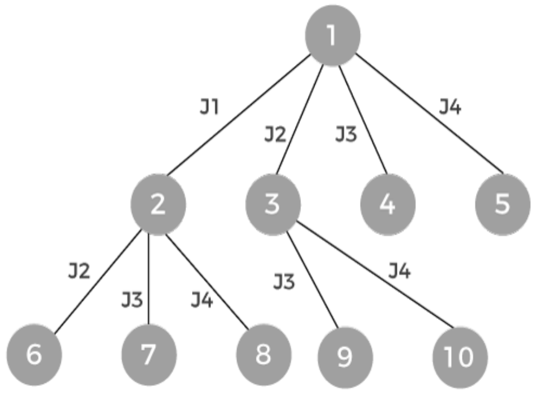
FIFO BRANCH AND BOUND METHOD:-



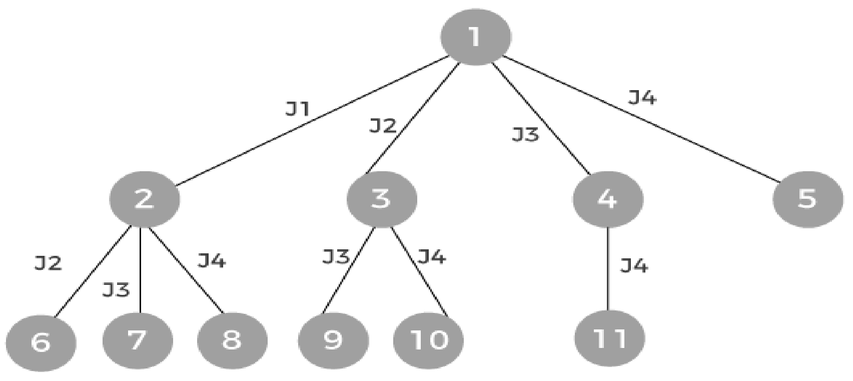
In this case, we first consider the first job, then second job, then third job and finally we consider the last job.
As we can observe in the above figure that the breadth first search is performed but not the depth first search.
Here we move breadth wise for exploring the solutions.
In backtracking, we go depth-wise whereas in branch and bound, we go breadth wise.
Now one level is completed. Once I take first job, then we can consider j2, j3 or j4. If we follow the route then it says that we are doing jobs j1 and j4 so we will not consider jobs j2 and j3.



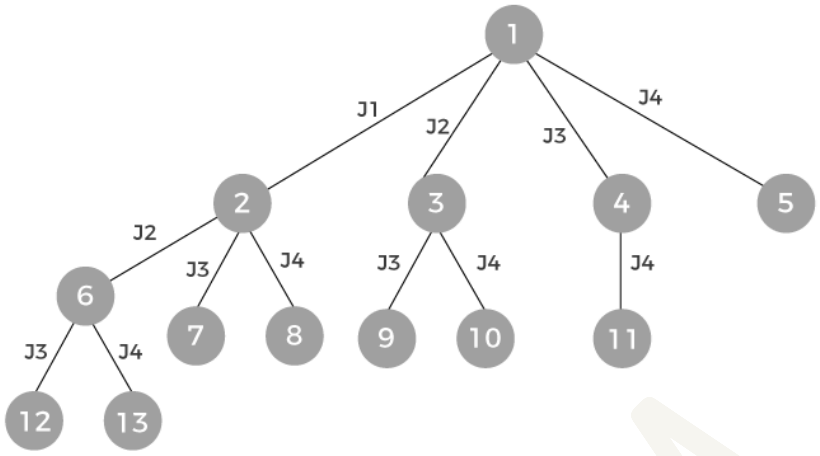
Now we will consider the node 3. In this case, we are doing job j2 so we can consider either job j3 or j4. Here, we have discarded the job j1.→



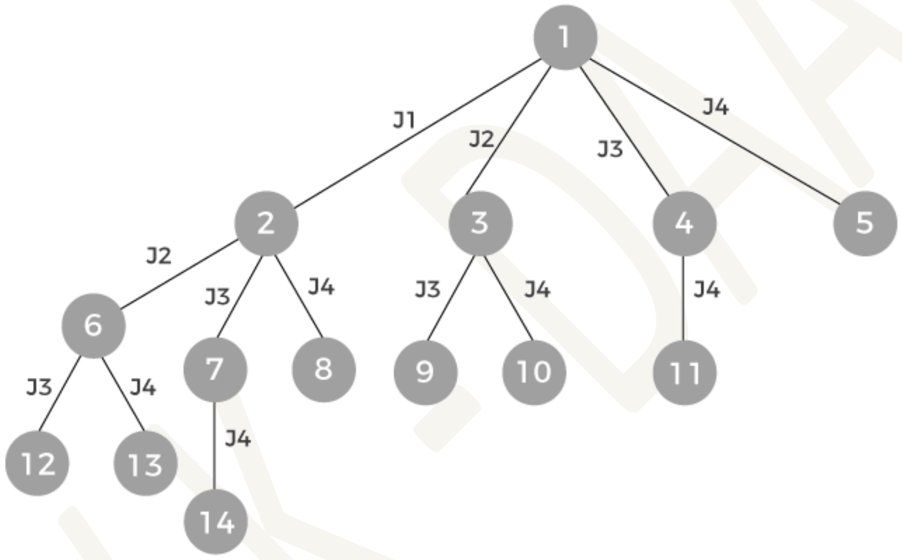
Now we will expand the node 4. Since here we are doing job j3 so we will consider only job j4.



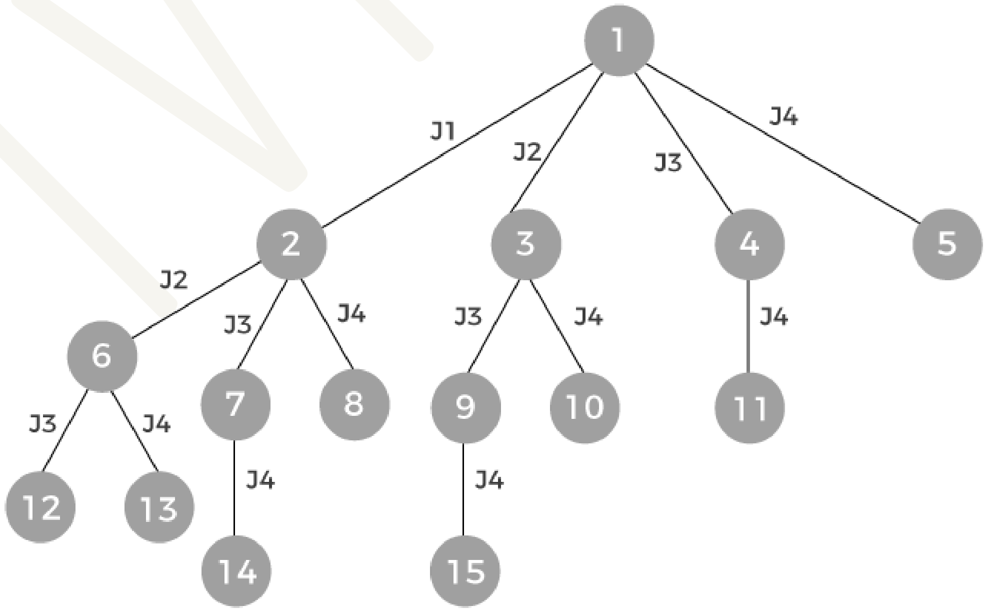
Now we will expand node 6, and here we will consider the jobs j3 and j4.



Now we will expand node 7 and here we will consider job j4.



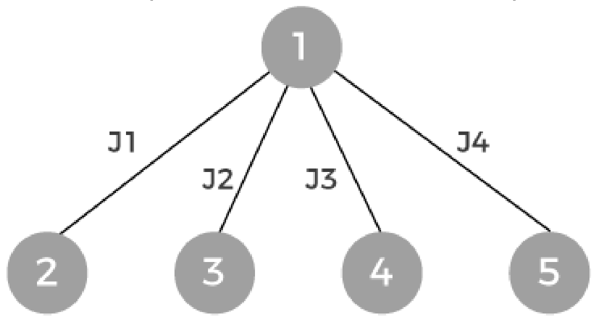
Now we will expand node 9, and here we will consider job j4.

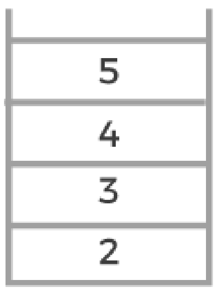


The last node, i.e., node 12 which is left to be expanded. Here, we consider job j4.
The above is the state space tree for the solution s1 = {j1, j4}.

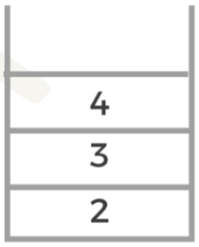
LIFO BRANCH AND BOUND METHOD:-

We will see another way to solve the problem to achieve the solution s1.
First, we consider the node 1 shown as below:
Now, we will expand the node 1. After expansion, the state space tree would be appeared as:
On each expansion, the node will be pushed into the stack shown as below:

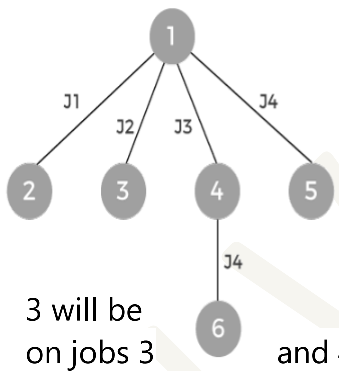


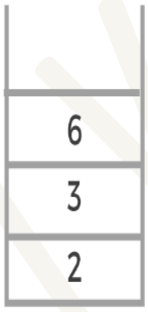


Now the expansion would be based on the node that appears on the top of the stack. Since the node 5 appears on the top of the stack, so we will expand the node 5. We will pop out the node 5 from the stack. Since the node 5 is in the last job, i.e., j4 so there is no further scope of expansion.



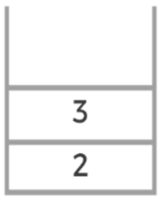
The next node that appears on the top of the stack is node 4. Pop out the node 4 and expand. On expansion, job j4 will be considered and node 6 will be added into the stack shown as below:

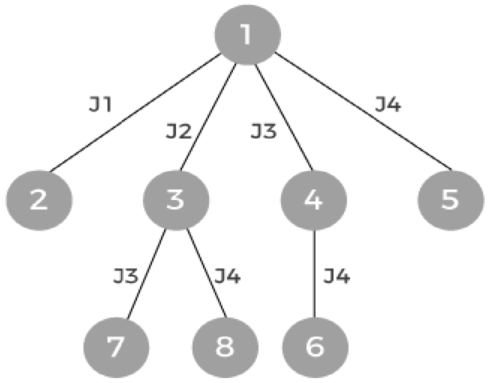


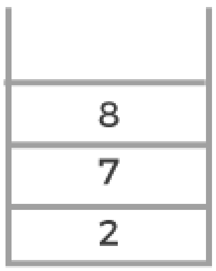


The next node is 6 which is to be expanded. Pop out the node 6 and expand. Since the node 6 is in the last job, i.e., j4 so there is no further scope of expansion.

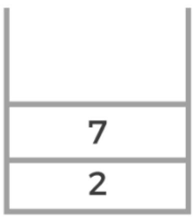
The next node to be expanded is node 3. Since the node 3 works on the job j2 so node 3 will be expanded to two nodes, i.e., 7 and 8 working on jobs 3 and 4 respectively. The nodes 7 and 8 will be pushed into the stack shown as below:

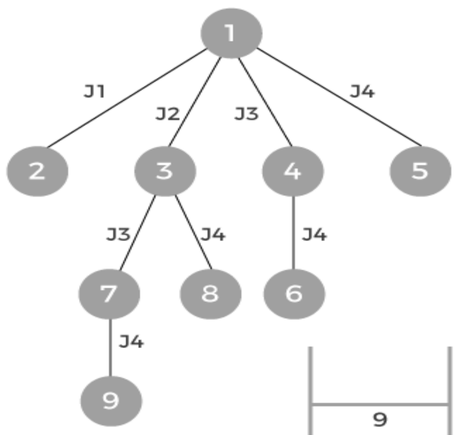


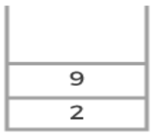




The next node that appears on the top of the stack is node 8. Pop out the node 8 and expand. Since the node 8 works on the job j4 so there is no further scope for the expansion.
The next node that appears on the top of the stack is node 7. Pop out the node 7 and expand. Since the node 7 works on the job j3 so node 7 will be further

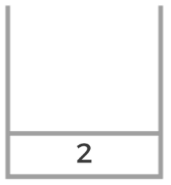




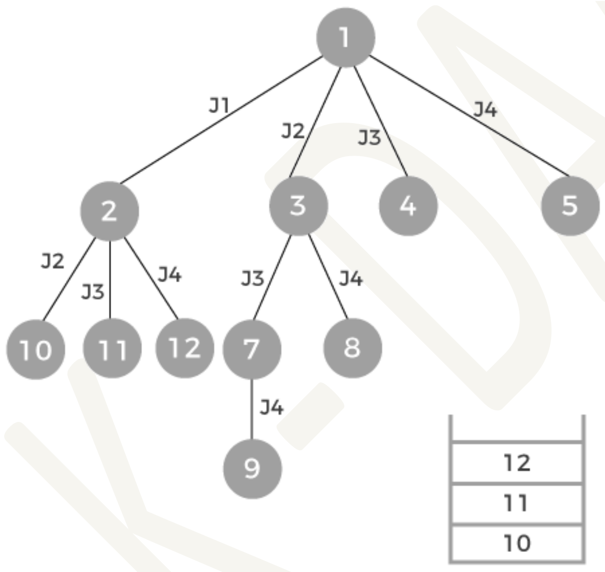


expanded to node 9 that works on the job j4 as shown as below and the node 9 will be pushed into the stack.

<--The next node that appears on the top of the stack is node 9. Since the node 9 works on the job 4 so there is no further scope for the expansion.



The next node that appears on the top of the stack is node 2. Since the node 2 works on the job j1 so it means that the node 2 can be further expanded. It can be expanded upto three nodes named as 10, 11, 12 working on jobs j2, j3, and j4 respectively. There newly nodes will be pushed into the stack shown as below:



In the above method, we explored all the nodes using the stack that follows the LIFO principle.

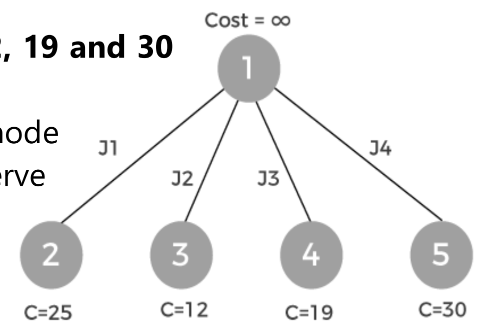
Least cost branch and bound method:-

There is one more method that can be used to find the solution and that method is Least cost branch and bound. 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.

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:

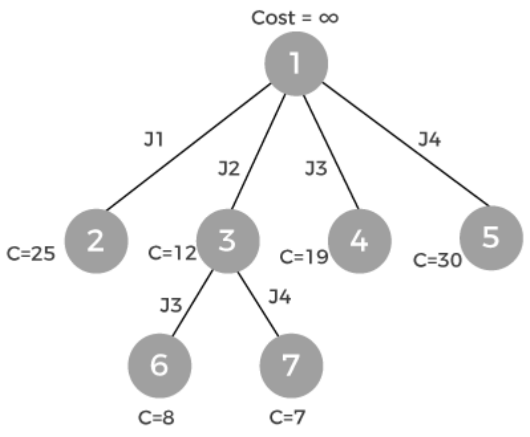
Let's assume that cost of the nodes 2, 3, 4, and 5 are 25, 12, 19 and 30 respectively.



Since it is the least cost branch n 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:

The node 6 works on job j3 while the node 7 works on job j4. The cost of the node 6 is 8 and the cost of the node 7 is 7. Now we have to select the node which is having the minimum cost. The node 7 has the minimum cost so we will explore the node 7. Since the node 7 already works on the job j4 so there is no further scope for the expansion.



Backtracking Vs Branch and Bound

Backtracking Vs Branch and Bound are discussed in this blog post. Backtracking is used to solve decision problems, whereas branch and bound is used to solve optimization problems. Decision problems are normally answered with yes or no. Optimization problems are those that seek to get the best possible solution from a given pool of solutions. In that way, the branch and bound methods sit in the category of dynamic programming and greedy algorithms.

Backtracking is a generic technique for solving various computational problems, most notably constraint satisfaction problems, that progressively constructs candidates to the solutions and abandons a candidate whenever it judges that it cannot potentially be completed to a legitimate solution.

Branch and Bound is a method for solving discrete and combinatorial optimization issues as well as mathematical optimization problems. The collection of candidate solutions is viewed as forming a rooted tree with the entire set at the root in a branch-and-bound method, which uses state-space search to systematically enumerate them. The method investigates the tree's branches, which are subsets of the solution set. Before enumerating a branch's candidate solutions, it's tested against upper and lower estimated bounds on the optimal solution, and it's eliminated if it can't generate a better solution than the algorithm's best so far.

Backtracking	**Branch and Bound**
Backtracking is normally used to solve decision problems	Branch and bound is used to solve optimization problems
Nodes in the state-space tree are explored in depth-first order in the backtracking method	Nodes in the tree may be explored in depth-first or breadth-first order in branch and bound method
It realizes that it has made a bad choice & undoes the last choice by backing up.	It realizes that it already has a better optimal solution that the pre-solution leads to so it abandons that pre-solution.

The feasibility function is used in backtracking.	Branch-and-Bound involves a bounding function.
The next move from the current state can lead to a bad choice	The next move is always towards a better solution
On successful search of a solution in state-space tree, the search stops	The entire state space tree is searched in order to find the optimal solution
Backtracking is more efficient.	Branch-and-Bound is less efficient.
Applications: N Queen Problem Knapsack Problem Sum of subsets problem Hamiltonian cycle problem, Graph coloring problem	**Applications**: Traveling salesman problem Knapsack problem Job sequencing problem

0/1 Knapsack Problem using Branch and Bound

We are a given a set of *n* objects which have each have a value v_i and a weight w_i. The objective of the 0/1 Knapsack problem is to find a subset of objects such that the total value is maximized, and the sum of weights of the objects does not exceed a given threshold *W*. An important condition here is that one can either take the entire object or leave it. It is not possible to take a fraction of the object.
Consider an example where *n* = 4, and the values are given by {10, 12, 12, 18} and the weights given by {2, 4, 6, 9}. The maximum weight is given by *W = 15*. Here, the solution to the problem will be including the first, third and the fourth objects.
In solving this problem, we shall use the Least Cost- Branch and Bound method, since this shall help us eliminate exploring certain branches of the tree. We shall also be using the fixed-size solution here. Another thing to be noted here is that this problem is a maximization problem, whereas the Branch and Bound method is for minimization problems. Hence, the values will be multiplied by -1 so that this problem gets converted into a minimization problem.
Now, consider the 0/1 knapsack problem with *n* objects and total weight W. We define the upper bound(U) to be the summation of v_ix_i (where v_i denotes the value of that objects, and x_i is a binary value, which indicates whether the object is to be included or not), such that the total weights of the included objects is less than *W*. The initial value of U is calculated at the initial position, where objects are added in order until the initial position is filled.
We define the cost function to be the summation of v_if_i, such that the total value is the maximum that can be obtained which is less than or equal to *W*. Here f_i indicates the fraction of the object that is to be included. Although we use fractions here, it is not included in the final solution.

Here, the procedure to solve the problem is as follows are:

- Calculate the cost function and the Upper bound for the two children of each node. Here, the (i + 1)th level indicates whether the ith object is to be included or not.
- If the cost function for a given node is greater than the upper bound, then the node need not be explored further. Hence, we can kill this node. Otherwise, calculate the upper bound for this node. If this value is less than *U*, then replace the value of *U* with this value. Then, kill all unexplored nodes which have cost function greater than this value.
- The next node to be checked after reaching all nodes in a particular level will be the one with the least cost function value among the unexplored nodes.
- While including an object, one needs to check whether the adding the object crossed the threshold. If it does, one has reached the terminal point in that branch, and all the succeeding objects will not be included.

In this manner, we shall find a value of U at the end which eliminates all other possibilities. The path to this node will determine the solution to this problem.

Algorithm for Knapsack Problem using Branch and Bound

```
Algorithm BB_KNAPSACK(cp, cw, k)
// Description : Solve knapsack problem using branch and bound
// Input:
cp: Current profit, initially 0
cw: Current weight, initially 0
k: Index of item being processed, initially 1
// Output:
fp: Final profit
Fw: Final weight
X: Solution tuple
cp ← 0
cw ← 0
k ← 1
if (cw + w[k] ≤ M) then
  Y[k]  ← 1
  if (k < n) then
    BB_KNAPSACK(cp + p[k], cw + w[k], k + 1)
  end
  if (cp + p[k] > fp) && (k == n) then
    fp ← cp + c[k]
    fw ← cw + w[k]
    X ← Y
  end
end
if BOUND(cp, cw, k) ≥ fp then
  Y[k] ← 0
  if (k < n) then
```

```
    BB_KNAPSACK(cp, cw, k + 1)
  end
  if( cp>fp ) && ( k == n ) then
    fp ← cp
    fw ← cw
    X ← Y
  end
end
```

Upper bound is computed as follow :

```
Function BOUND(cp, cw, k)
b ← cp
c ← cw
for i ← k + 1 to n do
  if (c + w[i] ≤ M) then
    c ← c + w[i]
    b ← b – p[i]
  end
end
return b
```

LCBB for Knapsack

LC branch and bound solution for knapsack problem is derived as follows :

1. Derive state space tree.
2. Compute lower bound *hatc()* and upper bound *u()* for each node in state space tree.
3. If lower bound is greater than upper bound then kill that node.
4. Else select node with minimum lower bound as E-node.
5. Repeat step 3 and 4 until all nodes are examined.
6. The node with minimum lower bound value *hatc()* is the answer node. Trace the path from leaf to root in the backward direction to find the solution tuple

The bounding function is a heuristic computation. For the same problem, there may be different bounding functions. Apart from the above-discussed bounding function, another very popular bounding function for knapsack is,

$$\mathbf{ub = v + (W - w) * (v_{i+1} / w_{i+1})}$$

Where,

v is value/profit associated with selected items from the first i items.

W is the capacity of the knapsack.

w is the weight of selected items from first i items

Example:

Solve the following instance of knapsack using LCBB for knapsack capacity M = 15.

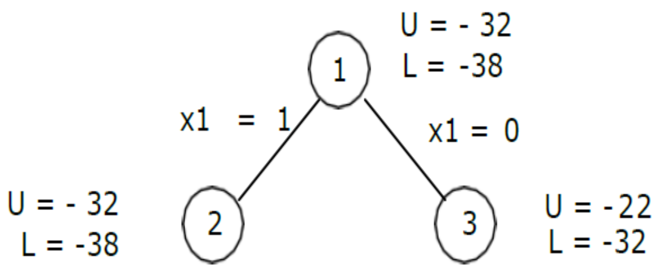
i	**P_i**	**W_i**

1	10	2
2	10	4
3	12	6
4	18	9

Solution:
Consider the instance: M = 15, n = 4, (P1, P2, P3, P4) = (10, 10, 12, 18) and (w1, w2, w3, w4) = (2, 4, 6, 9).
0/1 knapsack problem can be solved by using branch and bound technique. In this problem we will calculate lower bound and upper bound for each node.
Place first item in knapsack. Remaining weight of knapsack is 15 – 2 = 13. Place next item w2 in knapsack and the remaining weight of knapsack is 13 – 4 = 9. Place next item w3 in knapsack then the remaining weight of knapsack is 9 – 6 = 3. No fractions are allowed in calculation of upper bound so w4 cannot be placed in knapsack.
Profit = P1 + P2 + P3 = 10 + 10 + 12
So, Upper bound = 32
To calculate lower bound we can place w4 in knapsack since fractions are allowed in calculation of lower bound.

Lower bound = $10 + 10 + 12 + (\frac{3}{9} X\ 18) = 32 + 6 = 38$

Knapsack problem is maximization problem but branch and bound technique is applicable for only minimization problems. In order to convert maximization problem into minimization problem we have to take negative sign for upper bound and lower bound.
Therefore, Upper bound (U) = -32
Lower bound (L) = -38
We choose the path, which has minimum difference of upper bound and lower bound. If the difference is equal then we choose the path by comparing upper bounds and we discard node with maximum upper bound.
Now we will calculate upper bound and lower bound for nodes 2, 3.
For node 2, x1= 1, means we should place first item in the knapsack.
U = 10 + 10 + 12 = 32, make it as -32
L = $10 + 10 + 12 + \frac{3}{9}$ x 18 = 32 + 6 = 38, make it as -38
For node 3, x1 = 0, means we should not place first item in the knapsack.
U = 10 + 12 = 22, make it as -22



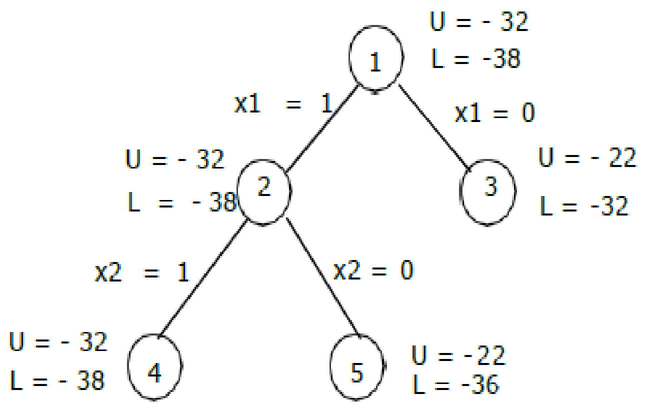
$L = 10 + 12 + \frac{5}{9} x\ 18 = 10 + 12 + 10 = 32$, make it as -32

Next, we will calculate difference of upper bound and lower bound for nodes 2, 3

For node 2, U – L = -32 + 38 = 6

For node 3, U – L = -22 + 32 = 10

Choose node 2, since it has minimum difference value of 6.



Now we will calculate lower bound and upper bound of node 4 and 5. Calculate difference of lower and upper bound of nodes 4 and 5.

For node 4, U – L = -32 + 38 = 6

For node 5, U – L = -22 + 36 = 14

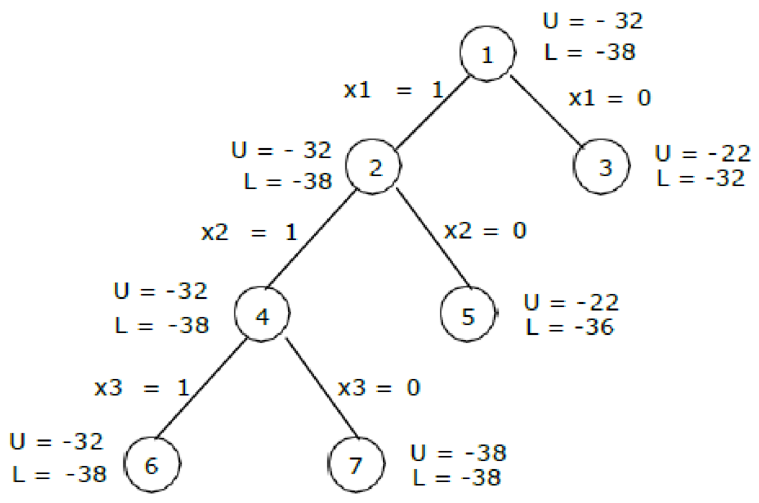
Choose node 4, since it has minimum difference value of 6.

Now we will calculate lower bound and upper bound of node 8 and 9. Calculate difference of lower and upper bound of nodes 8 and 9.

For node 6, U – L = -32 + 38 = 6

For node 7, U – L = -38 + 38 = 0

Choose node 7, since it is minimum difference value of 0.



Now we will calculate lower bound and upper bound of node 4 and 5. Calculate difference of lower and upper bound of nodes 4 and 5.

For node 8, U – L = -38 + 38 = 0

For node 9, U – L = -20 + 20 = 0

Here the difference is same, so compare upper bounds of nodes 8 and 9. Discard the node, which has maximum upper bound. Choose node 8, discard node 9 since, it has maximum upper bound.

Consider the path from

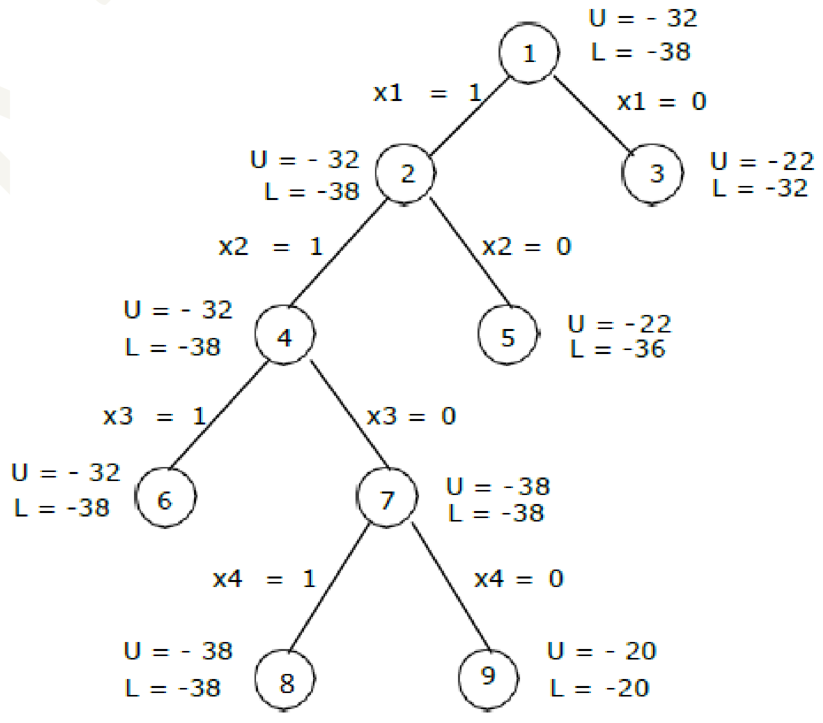
1 -> 2 -> 4 -> 7 -> 8

X1 = 1

X2 = 1

X3 = 0

X4 = 1



The solution for 0/1 Knapsack problem is (x1, x2, x3, x4) = (1, 1, 0, 1)

Maximum profit is: $\Sigma P_i\, x_i$ = 10 x 1 + 10 x 1 + 12 x 0 + 18 x 1
= 10 + 10 + 18 = 38.

Traveling Sale Person Problem:

By using dynamic programming algorithm we can solve the problem with time complexity of $O(n^2 2^n)$ for worst case. This can be solved by branch and bound technique using efficient bounding function. The time complexity of traveling sale person problem using LC branch and bound is $O(n^2 2^n)$ which shows that there is no change or reduction of complexity than previous method.

We start at a particular node and visit all nodes exactly once and come back to initial node with minimum cost.

Let G = (V, E) is a connected graph.

Let C(i, J) be the cost of edge <i, j>. $C_{ij} = \infty$• if <i, j> E and let |V| = n, the number of vertices.

Every tour starts at vertex 1 and ends at the same vertex.

So, the solution space is given by S = {1,π,1 | π is a permutation of (2,3,...n)} and |S|=(n-1)!

The size of S can be reduced by restricting S so that $(1, i_1, i_2, \ldots i_{n-1}, 1) \in S$ iff $<i_j, i_{j+1}> \in E$, $0 \le j \le n - 1$ and $i_0 = i_n = 1$.

Procedure for solving traveling sale person problem:

1. Reduce the given cost matrix. A matrix is reduced if every row and column is reduced. A row (column) is said to be reduced if it contain at least one zero and all-remaining entries are non-negative. This can be done as follows:

a) ***Row reduction***: Take the minimum element from first row, subtract it from all elements of first row, next take minimum element from the second row and subtract it from second row. Similarly apply the same procedure for all rows.

b) Find the sum of elements, which were subtracted from rows.

c) Apply column reductions for the matrix obtained after row reduction.

Column reduction: Take the minimum element from first column, subtract it from all elements of first column, next take minimum element from the second column and subtract it from second column. Similarly apply the same procedure for all columns.

d) Find the sum of elements, which were subtracted from columns.

e) Obtain the cumulative sum of row wise reduction and column wise reduction.

Cumulative reduced sum = Row wise reduction sum + column wise reduction sum.

Associate the cumulative reduced sum to the starting state as lower bound and • as upper bound.

2. Calculate the reduced cost matrix for every node R. Let A is the reduced cost matrix for node R. Let S be a child of R such that the tree edge (R, S) corresponds to including edge <i, j> in the tour. If S is not a leaf node, then the reduced cost matrix for S may be obtained as follows:

a) Change all entries in row i and column j of A to ∞.

b) Set A (j, 1) to ∞.

c) Reduce all rows and columns in the resulting matrix except for rows and column containing only ∞. Let r is the total amount subtracted to reduce the matrix.

c) Find $c(S)= c(R)+ A(i, j)+ r$, where 'r' is the total amount subtracted to reduce the matrix, *c(R)* indicates the lower bound of the ith node in (i, j) path and *c (S)* is called the cost function.

3. Repeat step 2 until all nodes are visited.

Example:

Find the LC branch and bound solution for the traveling sale person problem whose cost matrix is as follows:

The cost matrix is = $\begin{bmatrix} \infty & 20 & 30 & 10 & 11 \\ 15 & \infty & 16 & 4 & 2 \\ 3 & 5 & \infty & 2 & 4 \\ 19 & 6 & 18 & \infty & 3 \\ 16 & 4 & 7 & 16 & \infty \end{bmatrix}$

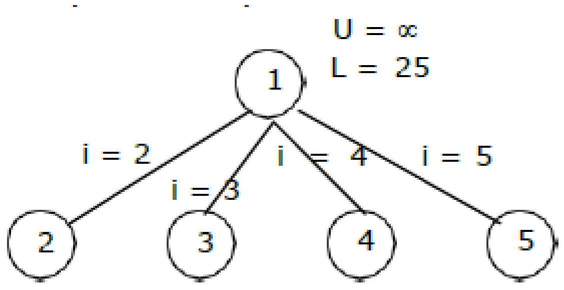
Step 1: Find the reduced cost matrix. Apply row reduction method:
Deduct 10 (which is the minimum) from all values in the 1st row.
Deduct 2 (which is the minimum) from all values in the 2nd row.
Deduct 2 (which is the minimum) from all values in the 3rd row.
Deduct 3 (which is the minimum) from all values in the 4th row.
Deduct 4 (which is the minimum) from all values in the 5th row.

The resulting row wise reduced cost matrix= $\begin{bmatrix} \infty & 10 & 20 & 0 & 1 \\ 13 & \infty & 14 & 2 & 0 \\ 1 & 3 & \infty & 0 & 2 \\ 16 & 3 & 15 & \infty & 0 \\ 12 & 0 & 3 & 12 & \infty \end{bmatrix}$

Row wise reduction sum = 10 + 2 + 2 + 3 + 4 = 21
Now apply column reduction for the above matrix:
Deduct 1 (which is the minimum) from all values in the 1st column.
Deduct 3 (which is the minimum) from all values in the 3rd column.

The resulting column wise reduced cost matrix (A) = $\begin{bmatrix} \infty & 10 & 17 & 0 & 1 \\ 12 & \infty & 11 & 2 & 0 \\ 0 & 3 & \infty & 0 & 2 \\ 15 & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & 12 & \infty \end{bmatrix}$

Column wise reduction sum = 1 + 0 + 3 + 0 + 0 = 4
Cumulative reduced sum = row wise reduction + column wise reduction sum.
= 21 + 4 = 25.
This is the cost of a root i.e., node 1, because this is the initially reduced cost matrix. The lower bound for node is 25 and upper bound is ∞.
Starting from node 1, we can next visit 2, 3, 4 and 5 vertices. So, consider to explore the paths (1, 2), (1, 3), (1, 4) and (1, 5).
The tree organization up to this point is as follows:
Variable "i" indicates the next node to visit.



Step 2:
Consider the path (1, 2):
Change all entries of row 1 and column 2 of A to ∞ and also set A(2, 1) to ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is = $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 2 & 0 \\ 0 & \infty & \infty & 0 & 2 \\ 15 & \infty & 12 & \infty & 0 \\ 11 & \infty & 0 & 12 & \infty \end{bmatrix}$

Row reduction sum = 0 + 0 + 0 + 0 = 0

Column reduction sum = 0 + 0 + 0 + 0 = 0

Cumulative reduction (r) = 0 + 0 = 0

Therefore, as *c*(*S) = c (R)* + *A (*1, 2) + *r*

c (S) = 25 + 10 + 0 = 35

Consider the path (1, 3):

Change all entries of row 1 and column 3 of A to ∞ and also set A (3, 1) to ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 15 & 3 & \infty & \infty & 0 \\ 11 & 0 & \infty & 12 & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & 2 & 0 \\ \infty & 3 & \infty & 0 & 2 \\ 4 & 3 & \infty & \infty & 0 \\ 0 & 0 & \infty & 12 & \infty \end{bmatrix}$

Row reduction sum = 0

Column reduction sum = 11

Cumulative reduction (r) = 0 + 11 = 11

Therefore, as c(*S) = c (R)* + *A (*1, 3) + *r*

c (*S*) = 25 + 17 + 11 = 53

Consider the path (1, 4):

Change all entries of row 1 and column 4 of A to ∞ and also set A(4, 1) to ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 0

Column reduction sum = 0

Cumulative reduction (r) = 0 + 0 = 0

Therefore, as c(*S) = c (R)* + *A (*1, 4) + *r*

c(S)= 25+0+0=25.

Consider the path (1, 5):

Change all entries of row 1 and column 5 of A to ∞ and also set A(5, 1) to ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & 2 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 15 & 3 & 12 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 10 & \infty & 9 & 0 & \infty \\ 0 & 3 & \infty & 0 & \infty \\ 12 & 0 & 9 & \infty & \infty \\ \infty & 0 & 0 & 12 & \infty \end{bmatrix}$

Row reduction sum = 5

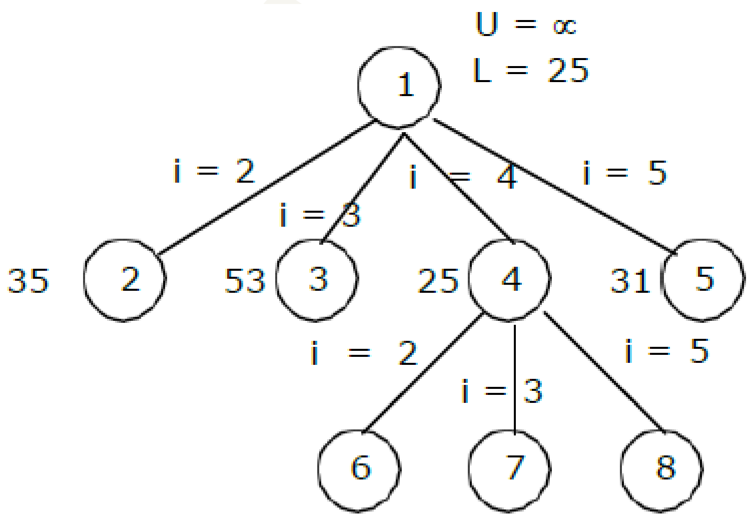
Column reduction sum = 0

Cumulative reduction (r) = 5 + 0 = 5

Therefore, as c(*S*) = *c*(*R*) + *A* (1, 5) + *r*

c(*S*)= 25+1+5 = 31.

The tree organization up to this point is as follows:



The cost of the paths between (1, 2) = 35, (1, 3) = 53, (1, 4) = 25 and (1, 5) = 31. The cost of the path between (1, 4) is minimum. Hence the matrix obtained for path (1, 4) is considered as reduced cost matrix.

$$A= \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & 0 \\ 0 & 3 & \infty & \infty & 2 \\ \infty & 3 & 12 & \infty & 0 \\ 11 & 0 & 0 & \infty & \infty \end{bmatrix}$$

The new possible paths are (4, 2), (4, 3) and (4, 5).

Consider the path (4, 2):

Change all entries of row 4 and column 2 of A to ∞ and also set A(2, 1) to ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is= $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$

Row reduction sum = 0

Column reduction sum = 0

Cumulative reduction (r) = 0 + 0 = 0

Therefore, as c(*S*) = *c*(*R*) + *A* (4, 2) + *r*

c(*S*)= 25+3+0=28.

Consider the path (4, 3):

Change all entries of row 4 and column 3 of A to ∞ and also set A(3, 1) to ∞

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & \infty & \infty & 0 \\ \infty & 3 & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & 0 & \infty & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is= $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & \infty & \infty & 0 \\ \infty & 1 & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & \infty & \infty & \infty \end{bmatrix}$

Row reduction sum = 2, Column reduction sum = 11

Cumulative reduction (r) = 2 + 11 = 13

Therefore, as c(*S) = c(R) + A* (4, 3) + *r*

c(S)=25+12+13=50

Consider the path (4, 5):

Change all entries of row 4 and column 5 of A to ∞ and also set A(5, 1) to ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 12 & \infty & 11 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ 1 & \infty & 0 & \infty & \infty \\ 0 & 3 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & 0 & 0 & \infty & \infty \end{bmatrix}$

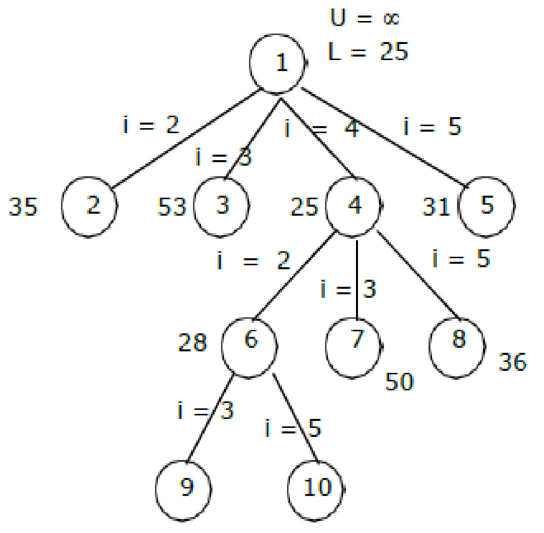
Row reduction sum = 11, Column reduction sum = 0

Cumulative reduction (r) = 11+0 = 11

Therefore, as c(*S) = c(R) + A* (4, 5) + *r*

c(S)=25+0+11=36

The tree organization up to this point is as follows:



The cost of the paths between (4, 2) = 28, (4, 3) = 50 and (4, 5) = 36. The cost of the path between (4, 2) is minimum. Hence the matrix obtained for path (4, 2) is considered as reduced cost matrix.

A= $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & \infty & 0 \\ 0 & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & 0 & \infty & \infty \end{bmatrix}$

The new possible paths are (2, 3) and (2, 5).

Consider the path (2, 3):

Change all entries of row 2 and column 3 of A to ∞ and also set A (3, 1) to ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 2 \\ \infty & \infty & \infty & \infty & \infty \\ 11 & \infty & \infty & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 0 \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \end{bmatrix}$

Row reduction sum = 2, Column reduction sum = 11

Cumulative reduction (r) = 2 + 11 = 13

Therefore, as c(*S*) = *c*(*R*) + *A* (2,3) + *r*

c(*S*)= *28+11+13 = 52*

Consider the path (2, 5):

Change all entries of row 2 and column 5 of A to ∞ and also set A(5, 1) to ∞.

$$\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$

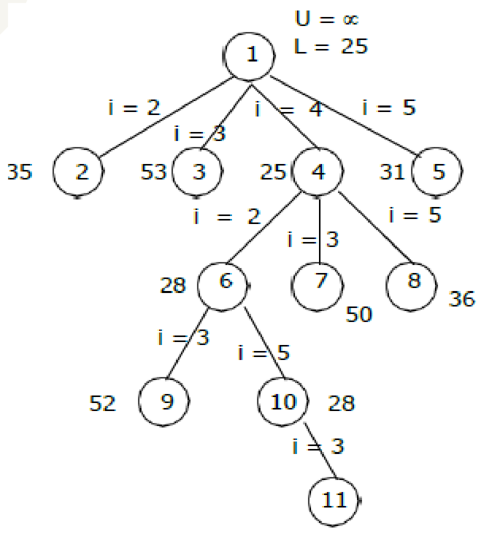
Row reduction sum = 0 Column reduction sum = 0

Cumulative reduction (r) = 0 + 0 = 0

Therefore, as c(*S*) = *c*(*R*) + *A* (2,5) + *r*

c(*S*)= *28+0+0=28*

The tree organization up to this point is as follows:



The cost of the paths between (2, 3) = 52 and (2, 5) = 28. The cost of the path between (2, 5) is minimum. Hence the matrix obtained for path (2, 5) is considered as reduced cost matrix.

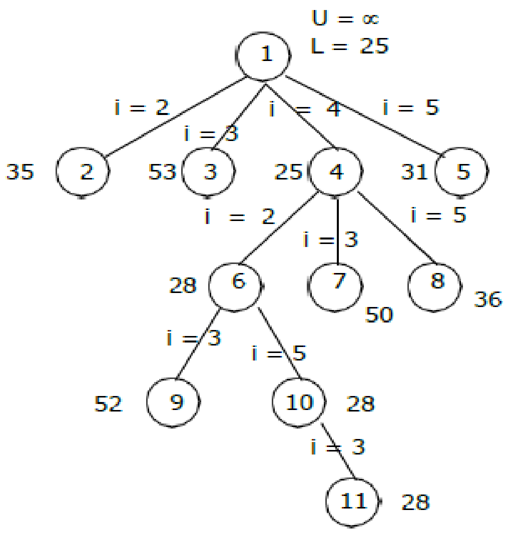
$$A = \begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 0 & \infty & \infty \end{bmatrix}$$

The new possible path is (5, 3).

Consider the path (5, 3):

Change all entries of row 5 and column 3 of A to ∞ and also set A(3, 1) to ∞. Apply row and column reduction for the rows and columns whose rows and columns are not completely ∞.

Then the resultant matrix is= $\begin{bmatrix} \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty \end{bmatrix}$



Row reduction sum = 0 Column reduction sum = 0

Cumulative reduction (r) = 0 + 0 = 0

Therefore, as $c(S) = c(R) + A(5, 3) + r$

$c(S)= 28+0+0= 28$

The overall tree organization is as follows:→

The path of traveling sale person problem is:

1→ 4→ 2→ 5→ 3→ 1

The minimum cost of the path is: 10 + 6 +2+ 7 + 3 = 28.

COMPLEXITY THEORY

In computer science, there exist some problems whose solutions are not yet found, the problems are divided into classes known as **Complexity Classes**. In complexity theory, a Complexity Class is a set of problems with related complexity. These classes help scientists to groups problems based on how much time and space they require to solve problems and verify the solutions. It is the branch of the theory of computation that deals with the resources required to solve a problem.

The common resources are time and space, meaning how much time the algorithm takes to solve a problem and the corresponding memory usage.

The time complexity of an algorithm is used to describe the number of steps required to solve a problem, but it can also be used to describe how long it takes to verify the answer.

The space complexity of an algorithm describes how much memory is required for the algorithm to operate.

Complexity classes are useful in organizing similar types of problems.

Types of Complexity Classes

This article discusses the following complexity classes:

1. P Class
2. NP Class
3. CoNP Class
4. NP hard
5. NP complete

P Class

The P in the P class stands for **Polynomial Time.** It is the collection of decision problems (problems with a “yes” or “no” answer) that can be solved by a deterministic machine in polynomial time.

Features:

1. The solution to P problems is easy to find.
2. P is often a class of computational problems that are solvable and tractable. Tractable means that the problems can be solved in theory as well as in practice. But the problems that can be solved in theory but not in practice are known as intractable.

This class contains many natural problems like:

1. Calculating the greatest common divisor.
2. Finding a maximum matching.
3. Decision versions of linear programming.

NP Class

The NP in NP class stands for **Non-deterministic Polynomial Time**. It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.

Features:

1. The solutions of the NP class are hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify.
2. Problems of NP can be verified by a Turing machine in polynomial time.

Example:

Let us consider an example to better understand the NP class. Suppose there is a company having a total of 1000 employees having unique employee IDs. Assume that there are 200 rooms available for them. A selection of 200 employees must be paired together, but the CEO of the company has the data of some employees who can't work in the same room due to some personal reasons.

This is an example of an NP problem. Since it is easy to check if the given choice of 200 employees proposed by a coworker is satisfactory or not i.e. no pair taken from the coworker list appears on the list given by the CEO. But generating such a list from scratch seems to be so hard as to be completely impractical.

It indicates that if someone can provide us with the solution to the problem, we can find the correct and incorrect pair in polynomial time. Thus for the NP class problem, the answer is possible, which can be calculated in polynomial time.

This class contains many problems that one would like to be able to solve effectively:

1. Boolean Satisfiability Problem (SAT).
2. Hamiltonian Path Problem.
3. Graph coloring.

Co-NP Class

Co-NP stands for the complement of NP Class. It means if the answer to a problem in Co-NP is No, then there is proof that can be checked in polynomial time.

Features:

1. If a problem X is in NP, then its complement X' is also is in CoNP.
2. For an NP and CoNP problem, there is no need to verify all the answers at once in polynomial time, there is a need to verify only one particular answer "yes" or "no" in polynomial time for a problem to be in NP or CoNP.

Some example problems for C0-NP are:

1. To check prime number.
2. Integer Factorization.

NP-hard class

An NP-hard problem is at least as hard as the hardest problem in NP and it is the class of the problems such that every problem in NP reduces to NP-hard.

Features:

1. All NP-hard problems are not in NP.
2. It takes a long time to check them. This means if a solution for an NP-hard problem is given then it takes a long time to check whether it is right or not.

3. A problem A is in NP-hard if, for every problem L in NP, there exists a polynomial-time reduction from L to A.

Some of the examples of problems in Np-hard are:

1. Halting problem.
2. Qualified Boolean formulas.
3. No Hamiltonian cycle.

NP-complete class

A problem is NP-complete if it is both NP and NP-hard. NP-complete problems are the hard problems in NP.

Features:

1. NP-complete problems are special as any problem in NP class can be transformed or reduced into NP-complete problems in polynomial time.
2. If one could solve an NP-complete problem in polynomial time, then one could also solve any NP problem in polynomial time.

Some example problems include:

1. Decision version of 0/1 Knapsack.
2. Hamiltonian Cycle.
3. Satisfiability.
4. Vertex cover**.**

Complexity Class	Characteristic feature
P	Easily solvable in polynomial time.
NP	Yes, answers can be checked in polynomial time.
Co-NP	No, answers can be checked in polynomial time.
NP-hard	All NP-hard problems are not in NP and it takes a long time to check them.
NP-complete	A problem that is NP and NP-hard is NP-complete.

Deterministic and Non-deterministic Algorithms

In the context of programming, an Algorithm is a set of well-defined instructions in sequence to perform a particular task and achieve the desired output. Here we say set of defined instructions which means that somewhere user knows the outcome of those instructions if they get executed in the expected manner.

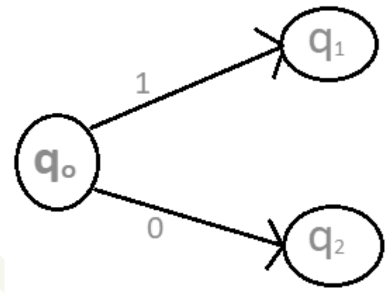
On the basis of the knowledge about outcome of the instructions, there are two types of algorithms namely – Deterministic and Non-deterministic Algorithms. Following are the main differences between both of the algorithms –

Sr. No.	Key	Deterministic Algorithm	Non-deterministic Algorithm
1	Definition	The algorithms in which the result of every algorithm is uniquely defined are known as the Deterministic Algorithm. In other words, we can say that the deterministic algorithm is the algorithm that performs fixed number of steps and always get finished with an accept or reject state with the same result.	On other hand, the algorithms in which the result of every algorithm is not uniquely defined and result could be random are known as the Non-Deterministic Algorithm.
2	Execution	In Deterministic Algorithms execution, the target machine executes the same instruction and results same outcome which is not dependent on the way or process in which instruction get executed.	On other hand in case of Non-Deterministic Algorithms, the machine executing each operation is allowed to choose any one of these outcomes subjects to a determination condition to be defined later.
3	Type	On the basis of execution and outcome in case of Deterministic algorithm, they are also classified as reliable algorithms as for a particular input instructions the machine will give always the same output.	On other hand Non deterministic algorithm are classified as non-reliable algorithms for a particular input the machine will give different output on different executions.
4	Execution Time	As outcome is known and is consistent on different executions so Deterministic algorithm takes polynomial time for their execution.	On other hand as outcome is not known and is non-consistent on different executions so Non-Deterministic algorithm could not get executed in polynomial time.

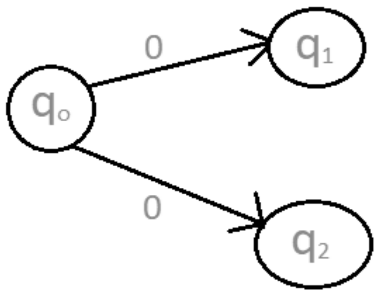
Sr. No.	Key	Deterministic Algorithm	Non-deterministic Algorithm
5	Execution path	In deterministic algorithm the path of execution for algorithm is same in every execution.	On other hand in case of Non-Deterministic algorithm the path of execution is not same for algorithm in every execution and could take any random path for its execution.

NON DETERMINISTIC ALGORITHM:

In a **deterministic algorithm**, for a given particular input, the computer will always produce the same output going through the same states but in the case of the **non-deterministic algorithm**, for the same input, the compiler may produce different output in different runs. In fact, non-deterministic algorithms can't solve the problem in polynomial time and can't determine what the next step is. The non-deterministic algorithms can show different behaviors for the same input on different execution and there is a degree of randomness to it.



Deterministic Algorithm



Non-Deterministic Algorithm

To implement a non-deterministic algorithm, we have a couple of languages like Prolog but these don't have standard programming language operators and these operators are not a part of any standard programming languages. **Some of the terms related to the non-deterministic algorithm are defined below**:

- **choice(X):** chooses any value randomly from the set X.
- **failure():** denotes the unsuccessful solution.
- **success():** The solution is successful and the current thread terminates.

Example :

Problem Statement: Search an element x on A[1:n] where n>=1, on successful search return j if a[j] is equals to x otherwise return 0. **Non-deterministic Algorithm for this problem :**

1.j= choice(a, n)

```
2.if(A[j]==x) then
   {
      write(j);
      success();
   }
3.write(0); failure();
```

Satisfiability Problem-NP complete

A *literal* is a either a propositional variable or the negation of a propositional variable.
Variables, such as *x* and *y*, are called *positive literals*.
Negated variables, such as ¬*x* and ¬*y*, are called *negative literals*.
A *clause* is a disjunction (*or*) of one or more literals. For example,

$$(x \lor \neg y \lor z)$$

is a clause.
A *clausal formula*, which we will just call a formula, is a conjunction (*and*) of one or more clauses. For example,

$$(\neg x \lor y) \land$$
$$(\neg y \lor z) \land$$
$$(x \lor \neg z \lor y)$$

is a (clausal) formula.

The satisfiability problem

Definition. A formula φ is *satisfiable* if there exists an assignment of values to its variables that makes φ true.
For example, formula

$$(\neg x \lor y) \land$$
$$(\neg y \lor z) \land$$
$$(x \lor \neg z \lor y)$$

is satisfiable. Choose

x = false

y = true

z = true

The satisfiability problem, usually called SAT, is the following language.

SAT = {φ | φ is a satisfiable clausal formula}.

Thought of as a computational problem, the input to SAT is a clausal formula φ and the problem is to determine whether φ is satisfiable.

SAT is NP-complete

Cook and Levin independently proved that SAT is NP-complete.

(Both did so before the term *NP-complete* was even in use. Their work led to the study of NP.)
Cook/Levin Theorem. SAT is NP-complete.
Proof. There are two parts to the proof because there are two parts to the definition of NP-completeness.
First, you must show that SAT is in NP.
Then you must show that, for every problem X in NP, $X \le_p$ SAT.
The first part is by far the easiest. The satisfiablity problem can be expressed as a test for existence.
φ is satisfiable $\Leftrightarrow$ **there exists** an assignment A of truth values to the variables in φ **so that** assignment A makes φ true.
So a nondeterministic algorithm for SAT goes like this.
Evidence for φ. An assignment of truth values to the variables that occur in φ.
The evidence is accepted if: the assignment makes φ true.
The second part ($X \le_p$ SAT for every X in NP) is considerably more difficult.
The proof is by a quite long *generic reduction*. Sipser describes the entire proof.
I do not duplicate the proof here, but here is a very rough sketch of how it goes.
Given an arbitrary language X in NP, select a polynomial-time nondeterministic Turing machine M for X.
Since M runs in polynomial time, there must be a k so that M runs in time n^k.
To show that $X \le_m$ SAT, we need a polynomial-time computable function f so that, for every y,

$$y \in X \Leftrightarrow f(y) \in \text{SAT}$$

Suppose that y has length n.
Notice that f(y) produces a clausal formula.
$f(y)$ begins by creating propositional variables $e_1 \ldots e_z$ where $z = n^k$.
Since M only runs for n^k steps on input y, M cannot look at more bits of evidence than that.
f(y) writes down a logical (clausal) formula φ that expresses, in logic,
"M accepts input (y, e) where $e = e_1 \ldots e_z$."
Formula φ does not say anything directly about the evidence variables. But once they are chosen, the remaining variables are constrained to describe a computation of M on input (y, e). So φ is satisfiable if and only if there exist values of the evidence variables so that M accepts input (y, e). But, by the choice of M, that is equivalent to saying that $y \in X$. ◊
It should not come as a great surprise that you can use logic to describe computation. After all, computers are built from logic gates, such as and-gates, or-gates and not-gates (inverters).
Consequences of the Cook/Levin theorem
First, the Cook/Levin theorem tells us that, if the conjecture that P ≠ NP is correct, then SAT is not in P.
It also makes it unnecessary to do more generic reductions in proofs of NP-completeness.
Suppose that you want to show that some language, W, is NP-complete.
First, of course, you show that W is in NP. So you find a polynomial-time nondeterministic algorithm for W.
Then you must show that $X <_p W$ for every X in NP.
But that can be done by showing that SAT $\le_p W$.
Then, since $X \le_p$ SAT and SAT $\le_p W$, $X \le_p W$, for every X in NP.