Module I

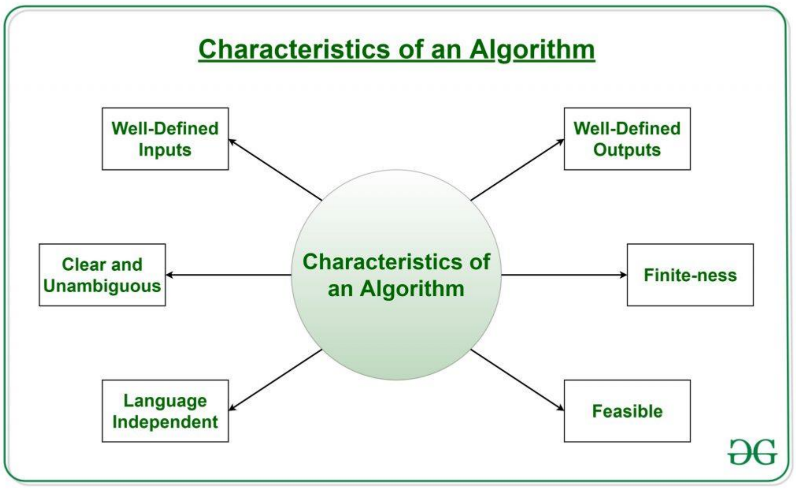
Algorithm Analysis

Algorithm

What is the need for algorithms?

1. Algorithms are necessary for solving complex problems efficiently and effectively.
2. They help to automate processes and make them more reliable, faster, and easier to perform.
3. Algorithms also enable computers to perform tasks that would be difficult or impossible for humans to do manually.
4. They are used in various fields such as mathematics, computer science, engineering, finance, and many others to optimize processes, analyze data, make predictions, and provide solutions to problems.

Properties of good Algorithm



- **Clear and Unambiguous**: The algorithm should be unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs**: If an algorithm says to take inputs, it should be well-defined inputs. It may or may not take input.
- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well. It should produce at least 1 output.
- **Finite-ness:** The algorithm must be finite, i.e. it should terminate after a finite time.
- **Feasible:** The algorithm must be simple, generic, and practical, such that it can be executed with the available resources. It must not contain some future technology or anything.
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.
- **Input**: An algorithm has zero or more inputs. Each that contains a fundamental operator must accept zero or more inputs.

- **Output**: An algorithm produces at least one output. Every instruction that contains a fundamental operator must accept zero or more inputs.
- **Definiteness:** All instructions in an algorithm must be unambiguous, precise, and easy to interpret. By referring to any of the instructions in an algorithm one can clearly understand what is to be done. Every fundamental operator in instruction must be defined without any ambiguity.
- **Finiteness:** An algorithm must terminate after a finite number of steps in all test cases. Every instruction which contains a fundamental operator must be terminated within a finite amount of time. Infinite loops or recursive functions without base conditions do not possess finiteness.
- **Effectiveness:** An algorithm must be developed by using very basic, simple, and feasible operations so that one can trace it out by using just paper and pencil.

Properties of Algorithm:

- It should terminate after a finite time.
- It should produce at least one output.
- It should take zero or more input.
- It should be deterministic means giving the same output for the same input case.
- Every step in the algorithm must be effective i.e. every step should do some work.

Types of Algorithms:

There are several types of algorithms . Some important algorithms are:

1. Brute Force Algorithm: It is the simplest approach to a problem. A brute force algorithm is the first approach that comes to finding when we see a problem.

2. Recursive Algorithm: A recursive algorithm is based on recursion. In this case, a problem is broken into several sub-parts and called the same function again and again.

3. Backtracking Algorithm: The backtracking algorithm builds the solution by searching among all possible solutions. Using this algorithm, we keep on building the solution following criteria. Whenever a solution fails we trace back to the failure point build on the next solution and continue this process till we find the solution or all possible solutions are looked after.

4. Searching Algorithm: Searching algorithms are the ones that are used for searching elements or groups of elements from a particular data structure. They can be of different types based on their approach or the data structure in which the element should be found.

5. Sorting Algorithm: Sorting is arranging a group of data in a particular manner according to the requirement. The algorithms which help in performing this function are called sorting algorithms. Generally sorting algorithms are used to sort groups of data in an increasing or decreasing manner.

6. Hashing Algorithm: Hashing algorithms work similarly to the searching algorithm. But they contain an index with a key ID. In hashing, a key is assigned to specific data.

7. Divide and Conquer Algorithm: This algorithm breaks a problem into sub-problems, solves a single sub-problem, and merges the solutions to get the final solution. It consists of the following three steps:

- Divide
- Solve

- Combine

8. Greedy Algorithm: In this type of algorithm, the solution is built part by part. The solution for the next part is built based on the immediate benefit of the next part. The one solution that gives the most benefit will be chosen as the solution for the next part.

9. Dynamic Programming Algorithm: This algorithm uses the concept of using the already found solution to avoid repetitive calculation of the same part of the problem. It divides the problem into smaller overlapping subproblems and solves them.

10. Randomized Algorithm: In the randomized algorithm, we use a random number so it gives immediate benefit. The random number helps in deciding the expected outcome.

Efficiency considerations of Algorithm

An algorithm is considered efficient if its resource consumption, also known as computational cost, is at or below some acceptable level. Acceptable means: it will run in a reasonable amount of time or space on an available computer, typically as a function of the size of the input.

Performance/Complexity

Two areas are important for performance:

1. *Space efficiency* - the memory required, also called, **space complexity**
2. *Time efficiency* - the time required, also called **time complexity**

Space Complexity

There are some circumstances where the space/memory used must be analyzed. For example, for large quantities of data or for embedded systems programming.

Components of space/memory use:

1. instruction space	Affected by: the compiler, compiler options, target computer (cpu)
2. data space	Affected by: the data size/dynamically allocated memory, static program variables,
3. run-time stack space	Affected by: the compiler, run-time function calls and recursion, local variables, parameters

The space requirement has fixed/static/compile time and a variable/dynamic/runtime components. Fixed components are the machine language instructions and static variables.

Variable components are the runtime stack usage and dynamically allocated memory usage.

Space efficiency is something we will try to maintain a general awareness of.

Time Compleexity

The actual running time depends on many factors:

- The speed of the computer: cpu (not just clock speed), I/O, etc.
- The compiler, compiler options .
- The quantity of data - ex. search a long list or short.
- The actual data - ex. in the sequential search if the name is first or last.

Time Efficiency - Approaches

When analyzing for time complexity we can take two approaches:

1. Order of magnitude/asymptotic categorization - This gives a general idea of performance. If algorithms fall into the same category, if data size is small, or if performance is critical, then the next approach can be looked at more closely.
2. Estimation of running time -
   By analysis of the code we can do:
    1. *operation counts* - select operation(s) that are executed most frequently and determine how many times each is done.
    2. *step counts* - determine the total number of steps, possibly lines of code, executed by the program.

Algorithm analysis

Algorithm analysis is the process of evaluating the performance of an algorithm, usually in terms of its time and space complexity. There are several ways to analyze the performance of an algorithm, including asymptotic analysis, which analyzes the behavior of an algorithm as the size of the input grows indefinitely.

Two approaches:

1. Experimental analysis
2. Asymptotic analysis

Experimental analysis

- It involves running the algorithm on a set of inputs and measuring its performance.
- This can provide a more accurate picture of an algorithm's performance, but it can be time-consuming and may only sometimes be feasible.
- Experimental analysis helps identify bottlenecks in the code.
- It may not be feasible for large input sizes, as it could take a long time to complete the experiments.

Asymptotic Analysis

- It analyses the behaviour of algorithm when the input size is grows indefinitely.
- Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time).
- We calculate, **order of growth** of time taken (or space) by an algorithm in terms of input size.
- For example linear search grows linearly and Binary Search grows logarithmically in terms of input size.

Asymptotic Notations:

Asymptotic notations are mathematical tools to represent the time complexity of algorithms for asymptotic analysis.

- Asymptotic Notations are mathematical tools used to analyze the performance of algorithms by understanding how their efficiency changes as the input size grows.
- These notations provide a concise way to express the behavior of an algorithm's time or space complexity as the input size approaches infinity.

- Rather than comparing algorithms directly, asymptotic analysis focuses on understanding the relative growth rates of algorithms' complexities.
- Asymptotic analysis allows for the comparison of algorithms' space and time complexities by examining their performance characteristics as the input size varies.
- By using asymptotic notations, such as Big O, Big Omega, and Big Theta, we can categorize algorithms based on their worst-case, best-case, or average-case time or space complexities.

There are mainly three asymptotic notations:

1. ***Big-O Notation (O-notation)***
2. ***Omega Notation (Ω-notation)***
3. ***Theta Notation (Θ-notation) Big-O -***

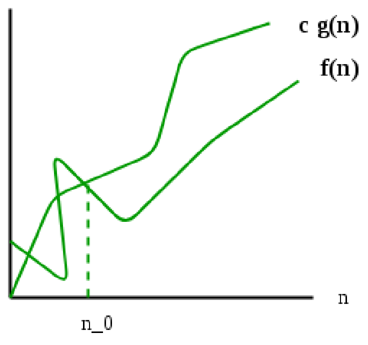
1. <u>Big-O Notation</u>

- ➢ The most frequently used notation.
- ➢ Always represent the **worst case** time complexity.
- ➢ Here we consider the least upperbound of an algorithm. Ie, the maximum time the algorithm will take to complete its execution.

<u>Formal definition of Big-O notation</u>

Assume f(n) and g(n) are two non-negative functions.

f(n) = O(g(n)), read as f of n is big oh of g of n, iff (if and only if) positive constants c and n_0 exist such that f(n) <= c.g(n) for all values of n, where n >= n_0.



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

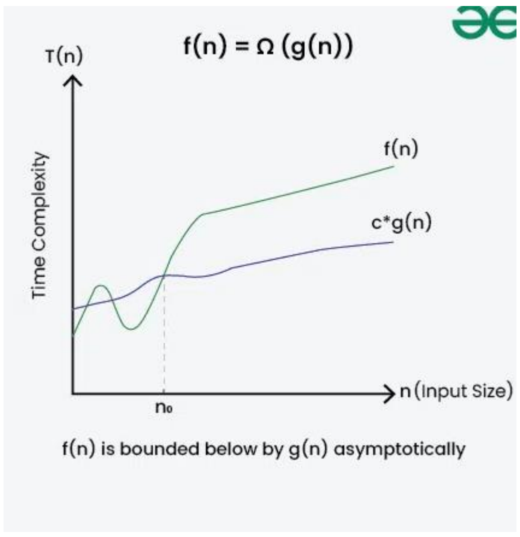
(Refer Lecture notes)

Qn: f(n)=5n+50 and g(n)=n. Is f(n)=O(g(n)).

2. Omega Notation

- ➢ Considering minimum time
- ➢ **Best case** time complexity
- ➢ F(n)=Ω(g(n))

***f(n) >= c.g(n)* for all *n, n >= n_0*.**

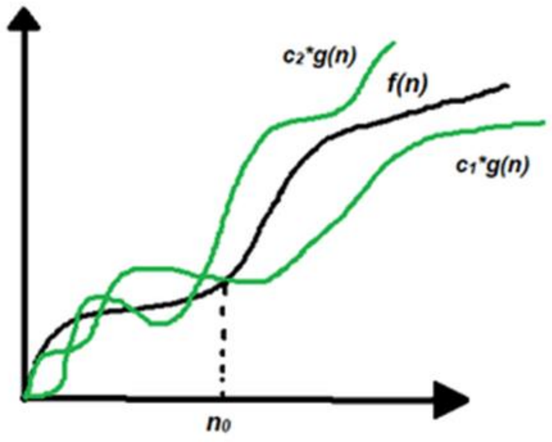


3. **Theta Notation**

Theta (θ) –

- Here we are considering the exact time.
- Represents the **average case time** complexity
- It is the combination of the above two.

***f(n) = θ(g(n))*, read as *f* of *n* is theta of *g* of *n*, iff (if and only if) positive constants c_1, c_2, and n_0 exist such that $c_1g(n) <= f(n) <= c_2g(n)$ for all *n*, *n* >= n_0.**



4. **Little o**
   - Same as Big-O.
   - Instead of least upperbound, we consider only upperbound

f(n) = o(g(n))*, read as *f* of *n* is little oh of *g* of *n*, *f(n) = O(g(n))* and *f(n) != omega(g(n))

5. **Little Omega** (ɯ)
   - Consider only lowerbound
   - In omega we consider greater lowerbound
     f(n)>ɯ. g(n)

Best case,Worst case,average case time complexity.

(Refer Module IV and refer lecture notes)

Recursion and its elimination

What is Recursion?
The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called a recursive function.

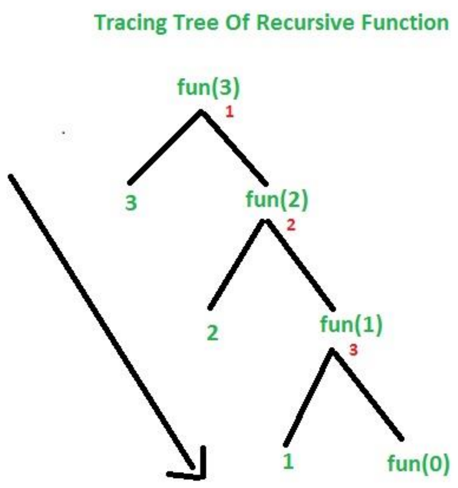
Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc. A recursive function solves a particular problem by calling itself with different parameters to solve smaller subproblems of the original problem.

1. Direct Recursion: These can be further categorized into **four types**:

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

```
#include <stdio.h>

// Recursion function

void fun(int n)

{

   if (n > 0) {

      printf("%d ", n);

      // Last statement in the function

      fun(n - 1);

   }

}

int main()

{

   int x = 3;

   fun(x);

   return 0;

}
```

Output: 3 2 1



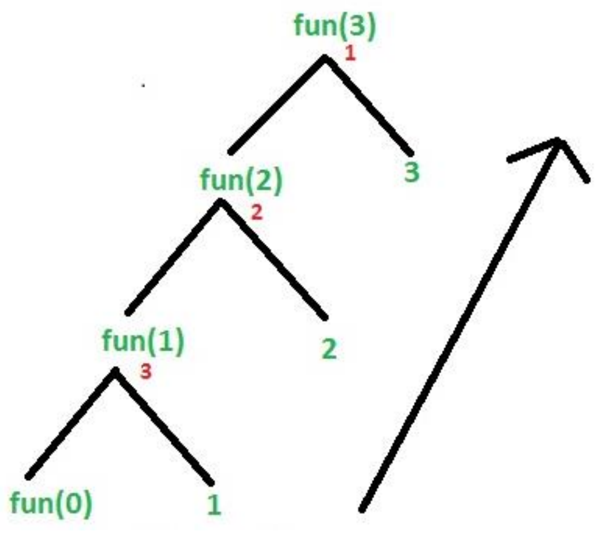
[Tail Recursion]

Time Complexity For Tail Recursion : O(n)
Space Complexity For Tail Recursion : O(n)
Note: Time & Space Complexity is given for this specific example. It may vary for another example.

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

```
#include <stdio.h>

 // Recursive function

void fun(int n)

{

   if (n > 0) {

        // First statement in the function

       fun(n - 1);

         printf("%d ", n);

   }

}

 int main()

{

   int x = 3;

   fun(x);

   return 0;

}
```

Output: 1 2 3

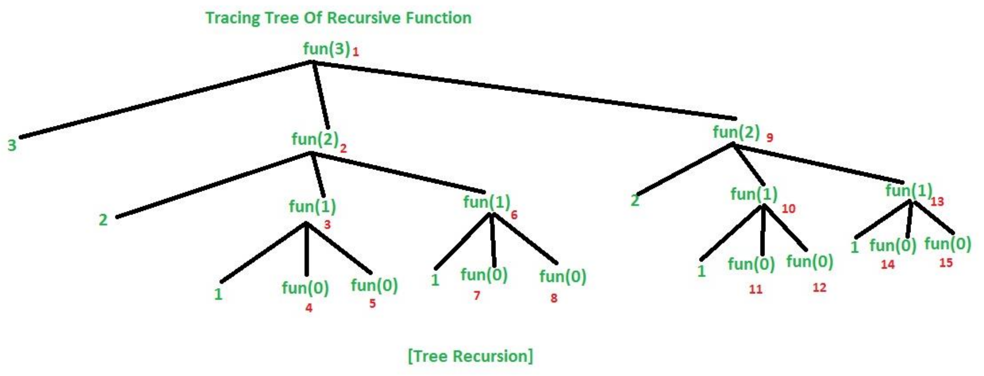


Time Complexity For Head Recursion: O(n)
Space Complexity For Head Recursion: O(n)

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

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

3 2 1 1 2 1 1



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

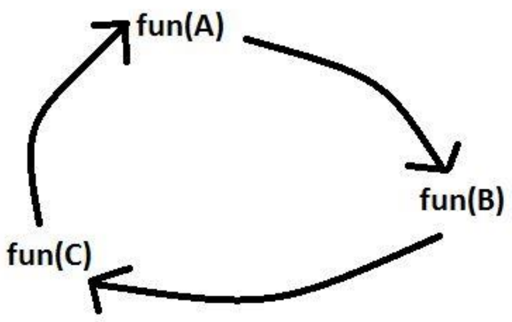
- Nested Recursion: In this recursion, a recursive function will pass the parameter as a recursive call. That means "recursion inside recursion". Let see the example to understand this recursion.

Example:

```
#include <stdio.h>

int fun(int n)

{

   if (n > 100)

       return n - 10;     // A recursive function passing parameter  as a recursive call or recursion inside
the recursion

   return fun(fun(n + 11));

}

 int main()

{    int r;

   r = fun(95);

   printf("%d\n", r);

   return 0;

}
```

Output :91   (Refer Lecture Notes)

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



Recursion VS Iteration

SR No.	Recursion	Iteration
1)	Terminates when the base case becomes true.	Terminates when the condition becomes false.
2)	Used with functions.	Used with loops.
3)	Every recursive call needs extra space in the stack memory.	Every iteration does not require any extra space.
4)	Smaller code size.	Larger code size.

Recursive and Non-Recursive algorithm for binary search(Refer lecture notes)

Module II

Algorithm Design Techniques

The following is a list of several popular design approaches:

1. Divide and Conquer Approach: It is a top-down approach. The algorithms which follow the divide & conquer techniques involve three steps:

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

2. Greedy Technique: Greedy method is used to solve the optimization problem. An optimization problem is one in which we are given a set of input values, which are required either to be maximized or minimized (known as objective), i.e. some constraints or conditions.

- Greedy Algorithm always makes the choice (greedy criteria) looks best at the moment, to optimize a given objective.
- The greedy algorithm doesn't always guarantee the optimal solution however it generally produces a solution that is very close in value to the optimal(approximate)

3. **Dynamic Programming:** Dynamic Programming is a bottom-up approach we solve all possible small problems and then combine them to obtain solutions for bigger problems.

This is helpful when the number of copying subproblems is exponentially large. Dynamic Programming is frequently related to Optimization Problems.

4. Branch and Bound: In Branch & Bound algorithm a given subproblem, which cannot be bounded, has to be divided into at least two new restricted subproblems. Branch and Bound algorithm are methods for global optimization in non-convex problems. Branch and Bound algorithms can be slow, however in the worst case they require effort that grows exponentially with problem size, but in some cases we are lucky, and the method coverage with much less effort.

5. Randomized Algorithms: A randomized algorithm is defined as an algorithm that is allowed to access a source of independent, unbiased random bits, and it is then allowed to use these random bits to influence its computation.

7. Backtracking Algorithm: Backtracking Algorithm tries each possibility until they find the right one. It is a depth-first search of the set of possible solution. During the search, if an alternative doesn’t work, then backtrack to the choice point, the place which presented different alternatives, and tries the next alternative.

Implementation of binary search using divide and conquer in C

The strategy behind divide and conquer method is to solve a problem where n inputs are split into many small subproblems and then each subproblem is solved and combined to find the solution of each subproblem. The solution of each subproblem then results in a solution to the original problem.

Why do we use divide and conquer method?

We use the divide and conquer method because it can be difficult to solve a problem with n inputs. So, we divide it into subproblems until it is easy to get a solution, and then we combine all solutions of the divided subproblem into one.

Solution :

Binary search is all about searching for an element – we can return the results as True/False or index value/-1(if not present).

here is an array of size 10 that contains integers in ascending order.

Can you find if 3 is there or not?

int a[10]={1,3,5,7,9,11,13,15,17,21};

First, grasp the logic of the divide and conquer technique visually for finding a target value (x=3).

Explanation

1. First take 3 variables in which the first integer index, last integer index, and middle integer index value of array/sub-array are stored and then use another variable to store the given integer to be found.

   int low=0,high=9,mid,x=3;

2. Consider a while-loop that will run until low variable is low/equal to high variable.
3. Now, within the while-loop, assign the mid variable to middle integer index –> mid=(low+high)/2
4. Now, check if x is greater than/less than/equals to mid

- If x>a[mid], that means x is on the right side of the middle index, so we will change low=mid.
- If x<a[mid], that means x is on the left side of the middle index, so we will change high=mid.
- Otherwise, x==a[mid] means x is found and return True.

[Code: Refer notes]

Example2: Try Yourself

{21, 17, 15, 13, 11, 9, 7, 5, 3, 1}

Complexity Analysis of Binary Search Algorithm

Time Complexity:

Best Case: O(1)

Average Case: O(log N)

Worst Case: O(log N)

Auxiliary Space: O(1), If the recursive call stack is considered then the auxiliary space will be O(logN).

Max-Min Problem

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

Solution

To find the maximum and minimum numbers in a given array ***numbers[]*** of size **n**, the following algorithm can be used. First we are representing the **naive method** and then we will present **divide and conquer approach**.

Naive Method

Naive method is a basic method to solve any problem. In this method, the maximum and minimum number can be found separately. To find the maximum and minimum numbers, the following straightforward algorithm can be used.

```
Algorithm: Max-Min-Element (numbers[])

max := numbers[1]

min := numbers[1]

for i = 2 to n do

  if numbers[i] > max then

    max := numbers[i]

  if numbers[i] < min then

    min := numbers[i]

return (max, min)
```

Analysis

The number of comparison in Naive method is **2n - 2**.

The number of comparisons can be reduced using the divide and conquer approach. Following is the technique.

Divide and Conquer Approach

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

```
    if (low == high) {
        result.max = arr[low];
        result.min = arr[low];
        return result;
    }
    // If there are two elements in the array
    if (high == low + 1) {
        if (arr[low] < arr[high]) {
            result.min = arr[low];
            result.max = arr[high];
        } else {
            result.min = arr[high];
            result.max = arr[low];
        }
        return result;
    }
    // If there are more than two elements in the array
    mid = (low + high) / 2;
    left = maxMin(arr, low, mid);//left subarray
    right = maxMin(arr, mid + 1, high);//right subarray
    // Compare and get the maximum of both parts
    result.max = (left.max > right.max) ? left.max : right.max;
    // Compare and get the minimum of both parts
    result.min = (left.min < right.min) ? left.min : right.min;
    return result;
}
```

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

$$T(n)=T(\lfloor n//2\rfloor)+T(\lceil n/2\rceil)+2$$

for n>2

for n=2

for n=1

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

So, $T(n)=2.T(n/2)+2=2.(2.T(n/4)+2)+2.....=[(3n)/2]-2$

Compared to Naïve method, in divide and conquer approach, the number of comparisons is less. However, using the asymptotic notation both of the approaches are represented by **O(n)**.

Strassens matrix multiplication

Strassen's algorithm, developed by Volker Strassen in 1969, is a fast algorithm for matrix multiplication. It is an efficient divide-and-conquer method that reduces the number of arithmetic operations required to multiply two matrices compared to the conventional matrix multiplication algorithm (the naive approach).

The traditional matrix multiplication algorithm has a time complexity of O(n^3) for multiplying two n x n matrices. However, Strassen's algorithm improves this to O(nlog7).The algorithm achieves this improvement by recursively breaking down the matrix multiplication into smaller subproblems and combining the results.

Here's how Strassen's Matrix Multiplication Algorithm works:

- **Divide**: Take the two matrices you want to multiply, let's call them A and B. Split them into four smaller matrices, each about half the size of the original matrices.
- **Calculate**: Use these smaller matrices to calculate seven special values, which we'll call P1, P2, P3, P4, P5, P6, and P7. You do this by doing some simple additions and subtractions of the smaller matrices.
- **Combine**: Take these seven values and use them to compute the final result matrix, which we'll call C. You calculate the values of C using the values of P1 to P7.

This method may sound a bit more complicated, but it's faster for really big matrices because it reduces the number of multiplications you need to do, even though it involves more additions and subtractions. For smaller matrices, the regular multiplication is faster, but for huge matrices, Strassen's method can save a lot of time.

Algorithm working As below:

Given two matrices A and B, of size n x n, we divide each matrix into four equal-sized submatrices, each of size n/2 x n/2.

A = | A11 A12 | B = | B11 B12 |

| A21 A22 | | B21 B22 |

We then define seven intermediate matrices:

P1 = A11 * (B12 - B22)

P2 = (A11 + A12) * B22

P3 = (A21 + A22) * B11

P4 = A22 * (B21 - B11)
P5 = (A11 + A22) * (B11 + B22)

P6 = (A12 - A22) * (B21 + B22)

P7 = (A11 - A21) * (B11 + B12)

Next, we recursively compute seven products of these submatrices, i.e., P1, P2, P3, P4, P5, P6, and P7.

Finally, we combine the results to obtain the four submatrices of the resulting matrix C of size n x n:

C11 = P5 + P4 - P2 + P6

C12 = P1 + P2

C21 = P3 + P4

C22 = P5 + P1 - P3 - P7

Concatenate the four submatrices C11, C12, C21, and C22 to obtain the final result matrix C.

Greedy Algorithms

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

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

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

Five Components:

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

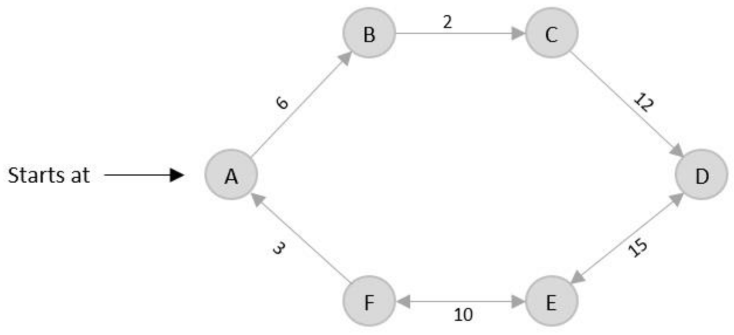
Application of Greedy Algorithm

Greedy approach is used to solve many problems, such as

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

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

The algorithm, similar to any shortest path algorithm, begins from a vertex that is set as a root and walks through all the vertices in the graph by determining the least cost adjacent edges.



Prim's Algorithm

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

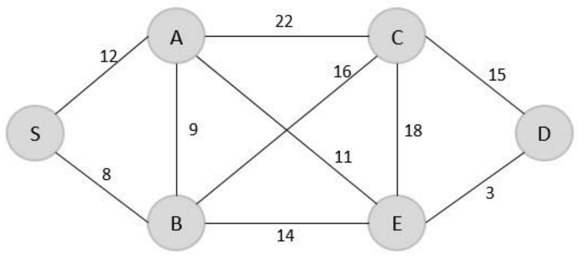
Algorithm

- Declare an array *visited*[] to store the visited vertices and firstly, add the arbitrary root, say S, to the visited array.
- Check whether the adjacent vertices of the last visited vertex are present in the *visited*[] array or not.

- If the vertices are not in the *visited*[] array, compare the cost of edges and add the least cost edge to the output spanning tree.
- The adjacent unvisited vertex with the least cost edge is added into the *visited*[] array and the least cost edge is added to the minimum spanning tree output.
- Steps 2 and 4 are repeated for all the unvisited vertices in the graph to obtain the full minimum spanning tree output for the given graph.
- Calculate the cost of the minimum spanning tree obtained.

Examples

- Find the minimum spanning tree using prim's method (greedy approach) for the graph given below with S as the arbitrary root.



Solution

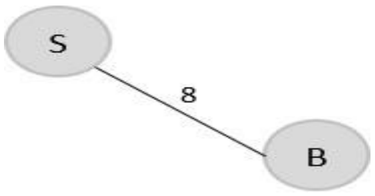
Step 1

Create a visited array to store all the visited vertices into it.

V = { }

The arbitrary root is mentioned to be S, so among all the edges that are connected to S we need to find the least cost edge. S → B = 8

V = {S, B}



Step 2

Since B is the last visited, check for the least cost edge that is connected to the vertex B.

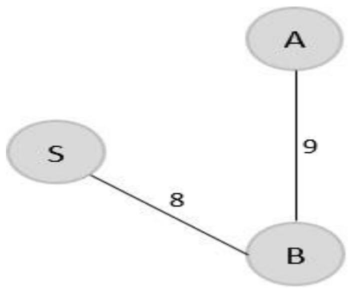
B → A = 9

B → C = 16

B → E = 14

Hence, B → A is the edge added to the spanning tree.

V = {S, B, A}



Step 3

Since A is the last visited, check for the least cost edge that is connected to the vertex A.

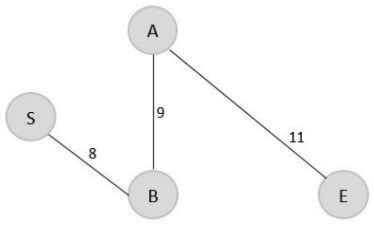
A → C = 22

A → B = 9

A → E = 11

But A → B is already in the spanning tree, check for the next least cost edge. Hence, A → E is added to the spanning tree.

V = {S, B, A, E}



Step 4

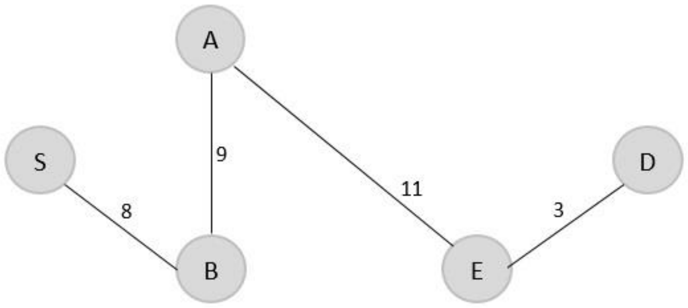
Since E is the last visited, check for the least cost edge that is connected to the vertex E.

E → C = 18

E → D = 3

Therefore, E → D is added to the spanning tree.

V = {S, B, A, E, D}



Step 5

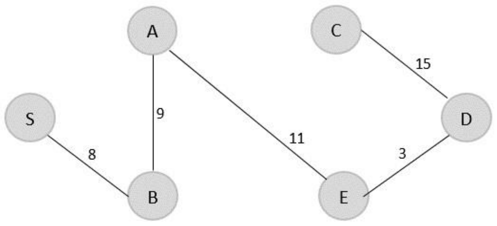
Since D is the last visited, check for the least cost edge that is connected to the vertex D.

D → C = 15

E → D = 3

Therefore, D → C is added to the spanning tree.

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



The minimum spanning tree is obtained with the minimum cost = 46

2. Kruskal’s Minimal Spanning Tree Algorithm

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

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

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

Kruskal's Algorithm

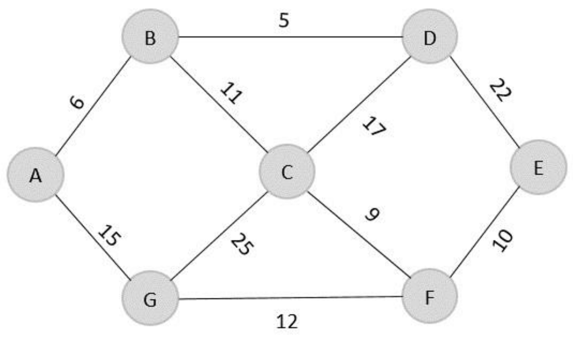
The inputs taken by the kruskal's algorithm are the graph G {V, E}, where V is the set of vertices and E is the set of edges, and the source vertex S and the minimum spanning tree of graph G is obtained as an output.

Algorithm

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

Examples

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

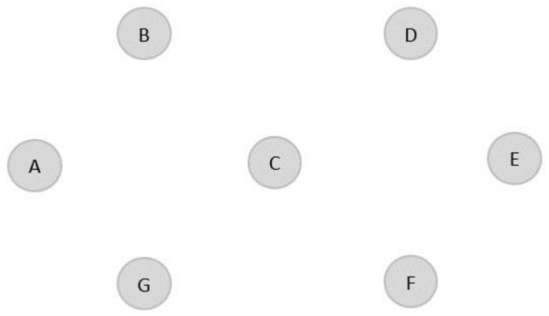


Solution

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

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

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

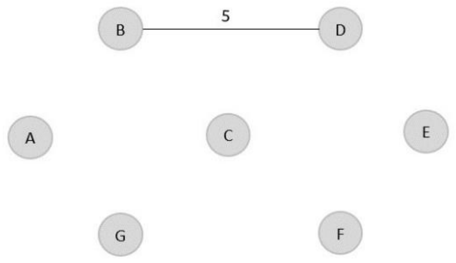


From the list of sorted edge costs, select the least cost edge and add it onto the forest in output graph.

B → D = 5

Minimum cost = 5

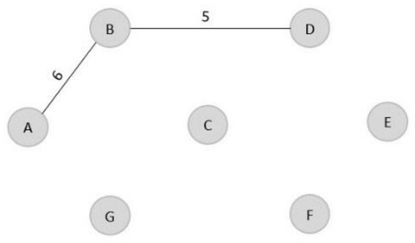
Visited array, v = {B, D}



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

Minimum cost = 5 + 6 = 11

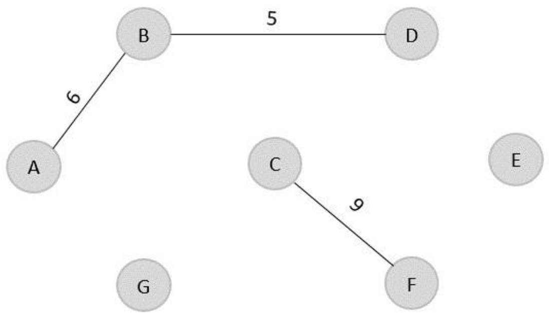
Visited array, v = {B, D, A}



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

Minimum Cost = 5 + 6 + 9 = 20

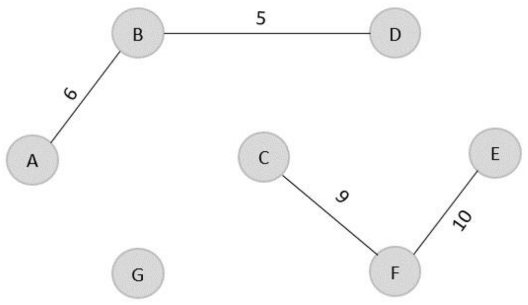
Visited array, v = {B, D, A, C, F}



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

Minimum Cost = 5 + 6 + 9 + 10 = 30

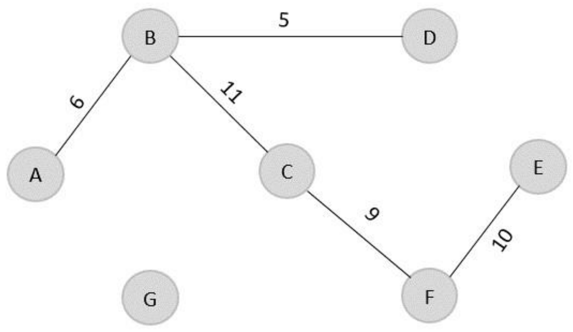
Visited array, v = {B, D, A, C, F, E}



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

Minimum cost = 5 + 6 + 9 + 10 + 11 = 41

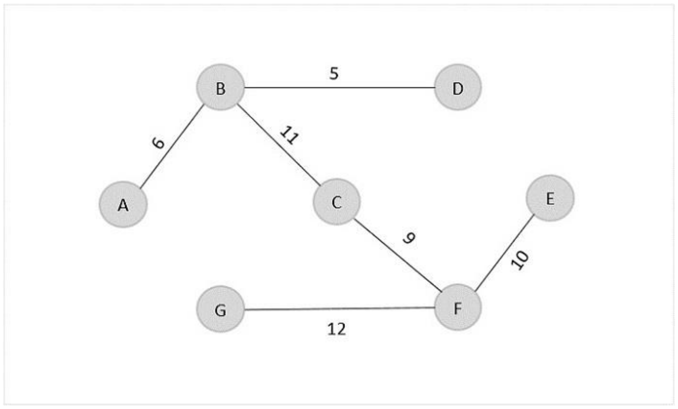
Visited array, v = {B, D, A, C, F, E}



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

Minimum cost = 5 + 6 + 9 + 10 + 11 + 12 = 53

Visited array, v = {B, D, A, C, F, E, G}



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

3. 0/1 Knapsack Problem

(Knapsack=container)

Given **N** items where each item has some weight and profit associated with it and also given a bag with capacity **W**, [i.e., the bag can hold at most **W** weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible.

Note: The constraint here is we can either put an item completely into the bag or cannot put it at all [It is not possible to put a part of an item into the bag].

Examples:

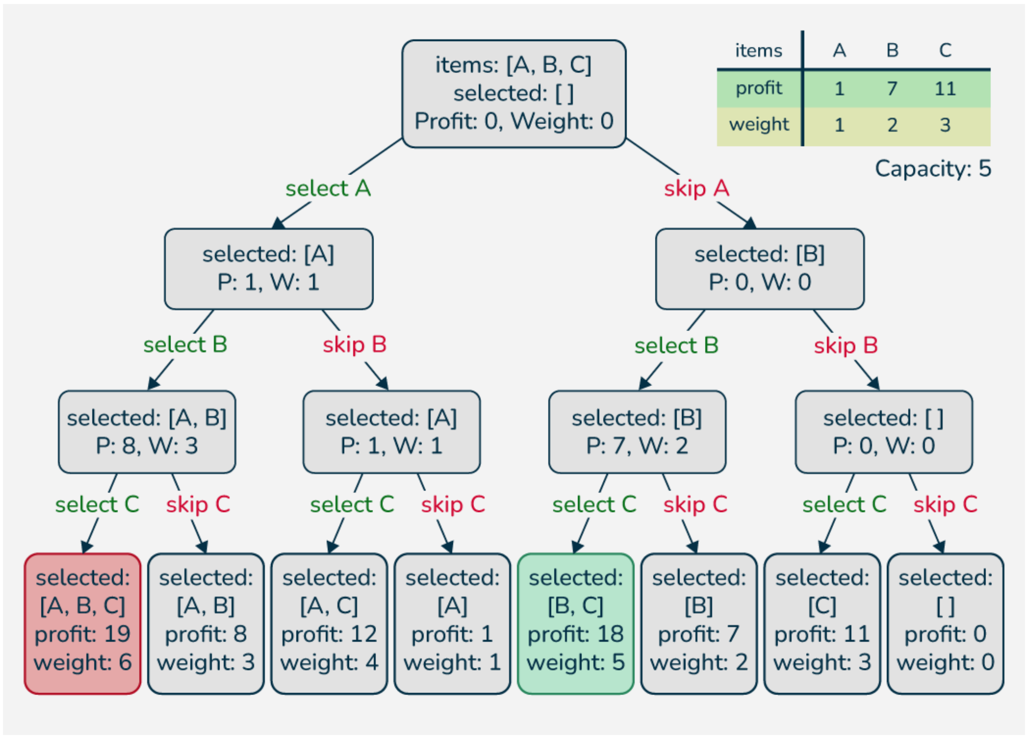
Input: *N = 3, W = 4, profit[] = {1, 2, 3}, weight[] = {4, 5, 1}*
Output: *3*
Explanation: *There are two items which have weight less than or equal to 4. If we select the item with weight 4, the possible profit is 1. And if we select the item with weight 1, the possible profit is 3. So the maximum possible profit is 3. Note that we cannot put both the items with weight 4 and 1 together as the capacity of the bag is 4.*

Input: *N = 3, W = 3, profit[] = {1, 2, 3}, weight[] = {4, 5, 6}*
Output: *0*

A simple solution is to consider all subsets of items and calculate the total weight and profit of all subsets. Consider the only subsets whose total weight is smaller than W. From all such subsets, pick the subset with maximum profit.

Optimal Substructure: To consider all subsets of items, there can be two cases for every item.

- ***Case 1: The item is included in the optimal subset.***
- ***Case 2: The item is not included in the optimal set.***



Eg:

Module III

Dynamic Programming

Dynamic programming is an Algorithm Deign 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.

It is used when the solution to a problem can be viewed as sequence of decisions.

Drawback of Greedy method is ,we will make one decision at a time. This can be overcome in Dynamic programming.

Here we can make more decisions at a time.

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

Principle of optimality

The principle of optimality which was developed by Richard Bellman,

is a fundamental aspect of dynamic programming, which states that the optimal solution to a dynamic optimization problem can be found by combining the optimal solutions to its sub-problems.

Key points:

- obtain the solution using principle of optimality.
- In an optimal sequence of decisions or choices,each subsequence must also be optimal.
- When it is not possible to obtain the principle of optimality it is almost impossible to obtain the solution using dynamic programming approach.
- Example:finding shortest path in a given graph uses the principle of optimality.

Consider an example of the Fibonacci series. The following series is the Fibonacci series:

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ,…

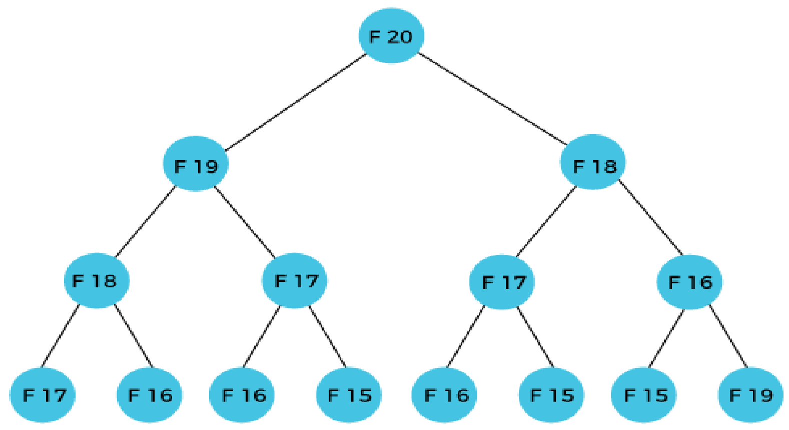
The numbers in the above series are not randomly calculated. Mathematically, we could write each of the terms using the below formula:

F(n) = F(n-1) + F(n-2),

With the base values F(0) = 0, and F(1) = 1. To calculate the other numbers, we follow the above relationship. For example, F(2) is the sum **f(0)** and **f(1),** which is equal to 1.

How can we calculate F(20)?

The F(20) term will be calculated using the nth formula of the Fibonacci series. The below figure shows that how F(20) is calculated.



In the dynamic programming approach, we try to divide the problem into the similar subproblems. In the above figure that F(20) is calculated as the sum of 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 , 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.]

The following are the steps that the dynamic programming follows:

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

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

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

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

Approaches of dynamic programming

There are two approaches to dynamic programming:

- Top-down approach
- Bottom-up approach

Top-down approach

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

Advantages

- It is very easy to understand and implement.
- It solves the subproblems 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 2n.

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

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.

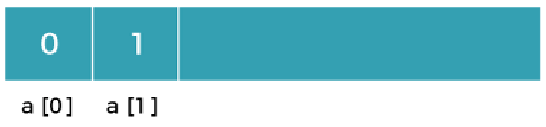
The bottom-up is the approach used to avoid the recursion, thus saving the memory space.

The bottom-up is an algorithm that starts from the beginning, whereas the recursive algorithm starts from the end and works backward.

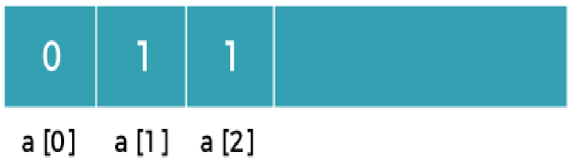
In the bottom-up approach, we start from the base case to find the answer for the end.

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.

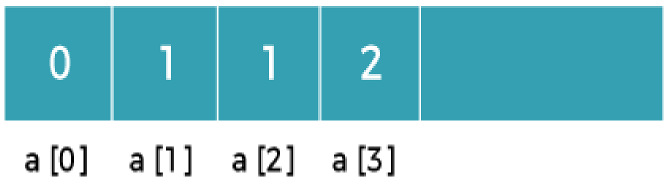
Suppose we have an array that has 0 and 1 values at a[0] and a[1] positions, respectively shown as below:



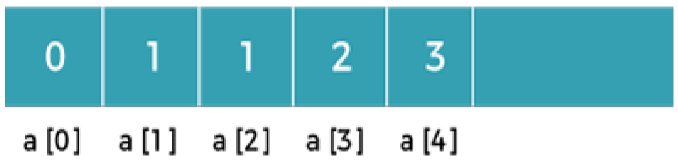
Since the bottom-up approach starts from the lower values, so the values at a[0] and a[1] are added to find the value



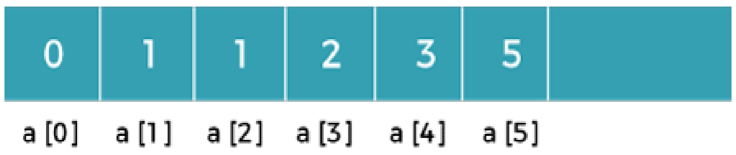
The value of a[3] will be calculated by adding a[1] and a[2], and it becomes 2 shown as below:



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



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



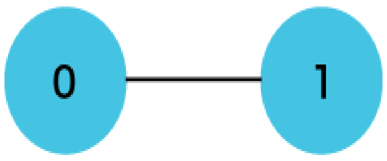
The code for implementing the Fibonacci series using the bottom-up approach is given below:

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

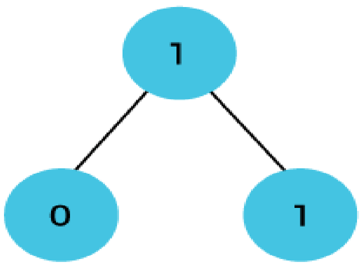
In the above code, base cases are 0 and 1 and then we have used for loop to find other values of Fibonacci series.

Let's understand through the diagrammatic representation.

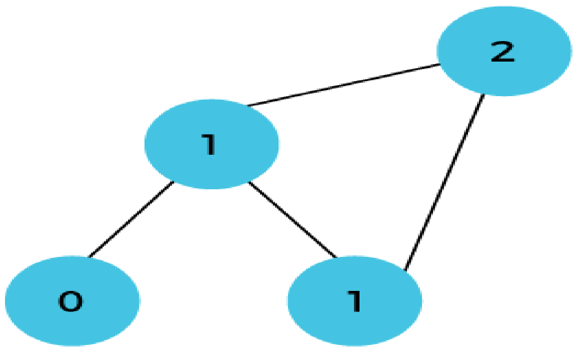
Initially, the first two values, i.e., 0 and 1 can be represented as:



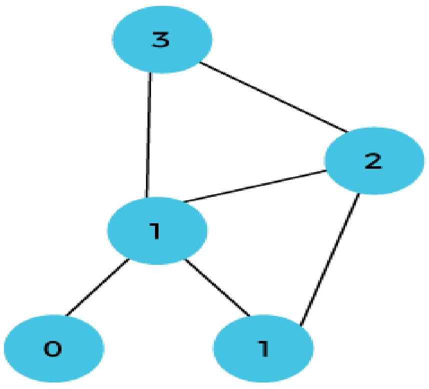
When i=2 then the values 0 and 1 are added shown as below:



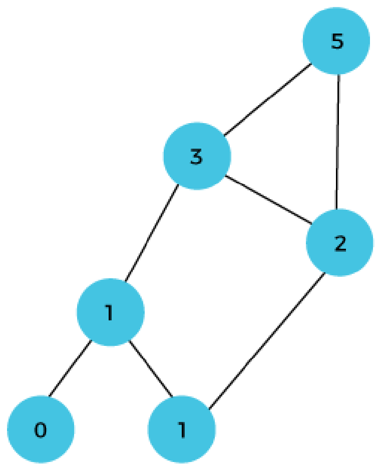
When i=3 then the values 1and 1 are added shown as below:



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.

All pairs shortest path

The **Floyd-Warshall algorithm**, named after its creators **Robert Floyd and Stephen Warshall**, is a fundamental algorithm in computer science and graph theory.

It is used to find the shortest paths between all pairs of nodes in a weighted graph.

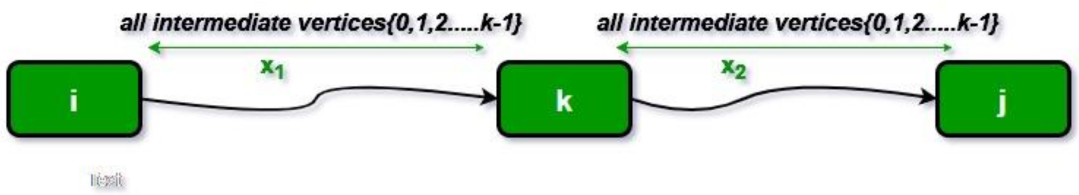
This algorithm is highly efficient and can handle graphs with both **positive** and n**egative edge weights**, making it a versatile tool for solving a wide range of network and connectivity problems.

The Floyd-Warshall algorithm is a dynamic programming algorithm used to discover the shortest paths in a weighted graph, which includes negative weight cycles.

The algorithm works with the aid of computing the shortest direction between every pair of vertices within the graph, it uses a matrix of intermediate vertices. At first the output matrix is same as given cost matrix of the graph. After that the output matrix will be updated with all vertices k as the intermediate vertex.

*Suppose we have a graph **G[][]** with **V** vertices from **1** to **N**. Now we have to evaluate a **shortestPathMatrix[][]** where s**hortestPathMatrix[i][j]** represents the shortest path between vertices **i** and **j**.*

*Obviously the shortest path between **i** to **j** will have some **k** number of intermediate nodes. The idea behind floyd warshall algorithm is to treat each and every vertex from **1** to **N** as an intermediate node one by one.*


all intermediate vertices{0,1,2.....k-1}
all intermediate vertices{0,1,2.....k-1}
x1
x2
i
k
j
text

Floyd Warshall Algorithm Algorithm:

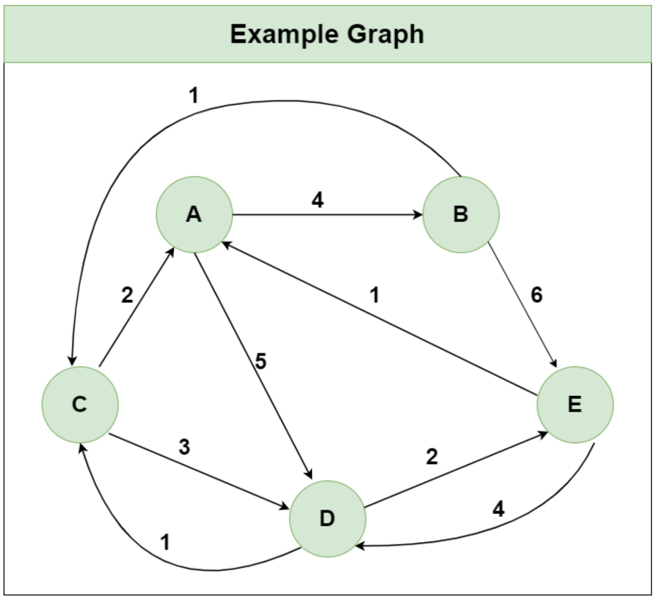
- Initialize the solution matrix same as the input graph matrix as a first step.

- Then update the solution matrix by considering all vertices as an intermediate vertex.
- The idea is to pick all vertices one by one and updates all shortest paths which include the picked vertex as an intermediate vertex in the shortest path.
- When we pick vertex number **k** as an intermediate vertex, we already have considered vertices **{0, 1, 2, .. k-1}** as intermediate vertices.
- For every pair **(i, j)** of the source and destination vertices respectively, there are two possible cases.
-
    - **k** is not an intermediate vertex in shortest path from **i** to **j**. We keep the value of **dist[i][j]** as it is.
    - **k** is an intermediate vertex in shortest path from **i** to **j**. We update the value of **dist[i][j]** as **dist[i][k] + dist[k][j],** if **dist[i][j] > dist[i][k] + dist[k][j]**

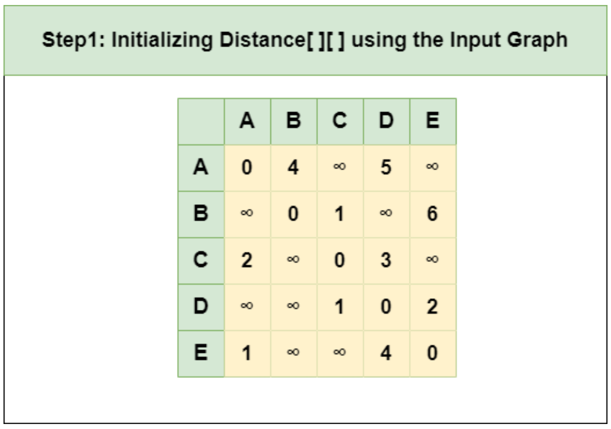
<u>Pseudo-Code of Floyd Warshall Algorithm :</u>

```
For k = 0 to n – 1
For i = 0 to n – 1
For j = 0 to n – 1
Distance[i, j] = min(Distance[i, j], Distance[i, k] + Distance[k, j])
```

where i = source Node, j = Destination Node, k = Intermediate Node



***<u>Step 1</u>:** Initialize the Distance[][] matrix using the input graph such that Distance[i][j]= weight of edge from **i** to **j**, also Distance[i][j] = Infinity if there is no edge from **i** to **j**.*

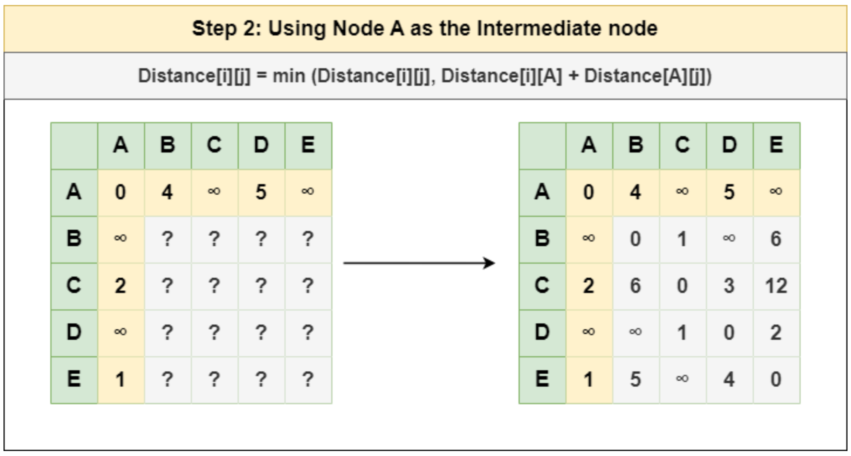


Step1: Initializing Distance[ ][ ] using the Input Graph

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	∞	0	3	∞
D	∞	∞	1	0	2
E	1	∞	∞	4	0

Step 2: *Treat node **A** as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:*

*= Distance[i][j] = minimum (Distance[i][j], (Distance from i to **A**) + (Distance from **A** to j))*
*= Distance[i][j] = minimum (Distance[i][j], Distance[i][**A**] + Distance[**A**][j])*



Step 2: Using Node A as the Intermediate node

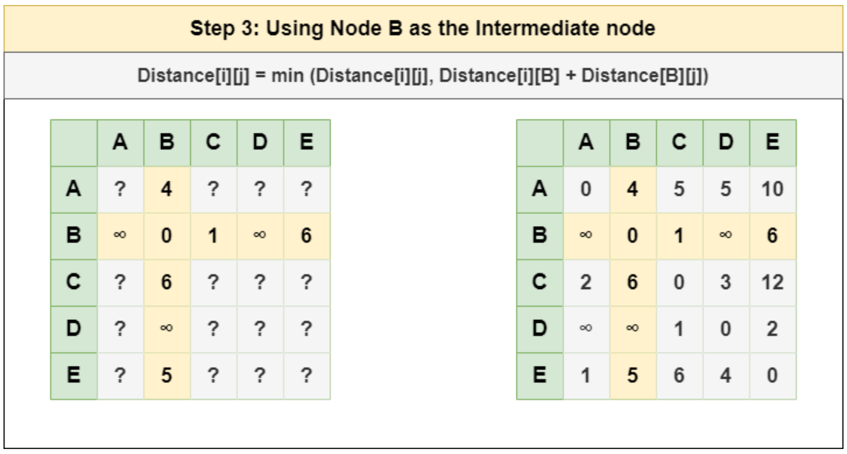
Distance[i][j] = min (Distance[i][j], Distance[i][A] + Distance[A][j])

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	?	?	?	?
C	2	?	?	?	?
D	∞	?	?	?	?
E	1	?	?	?	?

	A	B	C	D	E
A	0	4	∞	5	∞
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	∞	4	0

Step 3: *Treat node **B** as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:*

*= Distance[i][j] = minimum (Distance[i][j], (Distance from i to **B**) + (Distance from **B** to j))*
*= Distance[i][j] = minimum (Distance[i][j], Distance[i][**B**] + Distance[**B**][j])*



Step 3: Using Node B as the Intermediate node

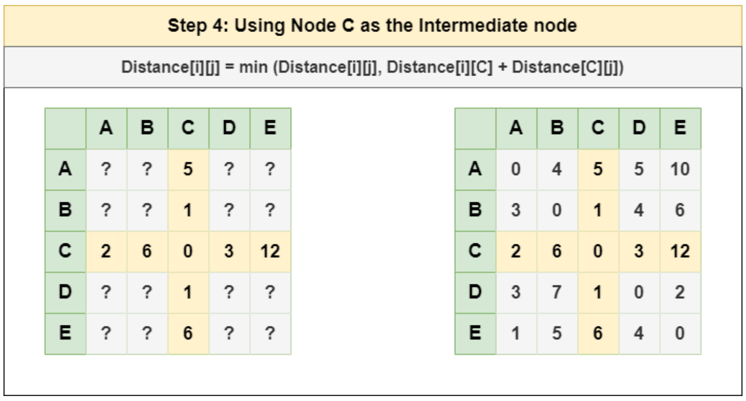
Distance[i][j] = min (Distance[i][j], Distance[i][B] + Distance[B][j])

	A	B	C	D	E
A	?	4	?	?	?
B	∞	0	1	∞	6
C	?	6	?	?	?
D	?	∞	?	?	?
E	?	5	?	?	?

	A	B	C	D	E
A	0	4	5	5	10
B	∞	0	1	∞	6
C	2	6	0	3	12
D	∞	∞	1	0	2
E	1	5	6	4	0

***<u>Step 4</u>**: Treat node **C** as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:*

*= Distance[i][j] = minimum (Distance[i][j], (Distance from i to **C**) + (Distance from **C** to j))*
*= Distance[i][j] = minimum (Distance[i][j], Distance[i][**C**] + Distance[**C**][j])*



Step 4: Using Node C as the Intermediate node

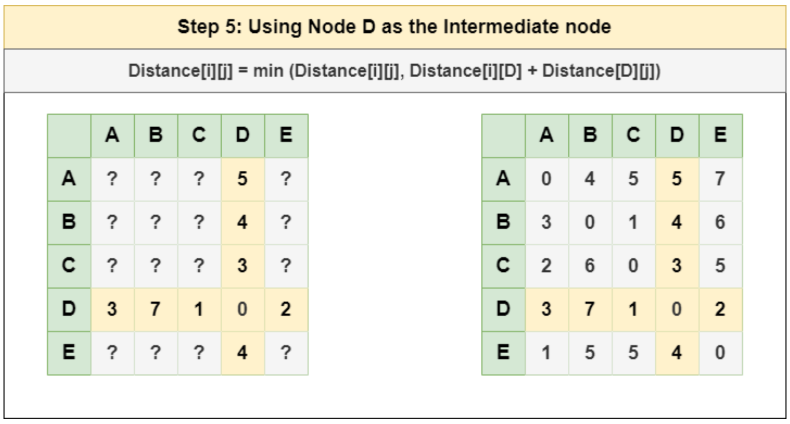
Distance[i][j] = min (Distance[i][j], Distance[i][C] + Distance[C][j])

	A	B	C	D	E
A	?	?	5	?	?
B	?	?	1	?	?
C	2	6	0	3	12
D	?	?	1	?	?
E	?	?	6	?	?

	A	B	C	D	E
A	0	4	5	5	10
B	3	0	1	4	6
C	2	6	0	3	12
D	3	7	1	0	2
E	1	5	6	4	0

***<u>Step 5</u>**: Treat node **D** as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:*

*= Distance[i][j] = minimum (Distance[i][j], (Distance from i to **D**) + (Distance from **D** to j))*
*= Distance[i][j] = minimum (Distance[i][j], Distance[i][**D**] + Distance[**D**][j])*



Step 5: Using Node D as the Intermediate node

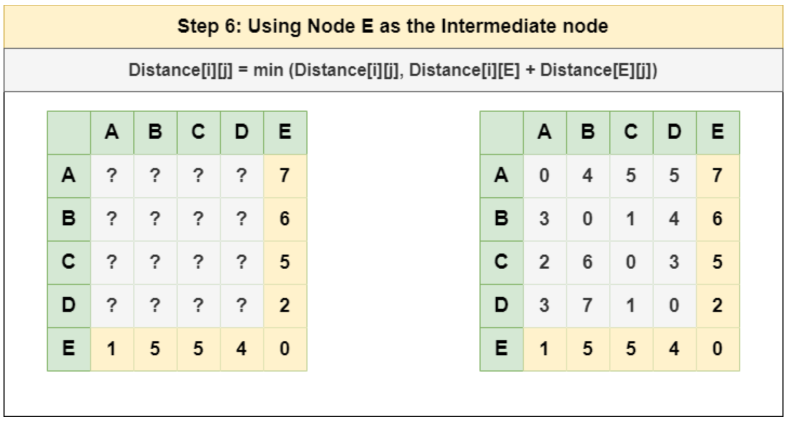
Distance[i][j] = min (Distance[i][j], Distance[i][D] + Distance[D][j])

	A	B	C	D	E
A	?	?	?	5	?
B	?	?	?	4	?
C	?	?	?	3	?
D	3	7	1	0	2
E	?	?	?	4	?

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

***<u>Step 6</u>**: Treat node **E** as an intermediate node and calculate the Distance[][] for every {i,j} node pair using the formula:*

*= Distance[i][j] = minimum (Distance[i][j], (Distance from i to **E**) + (Distance from **E** to j))*
*= Distance[i][j] = minimum (Distance[i][j], Distance[i][**E**] + Distance[**E**][j])*



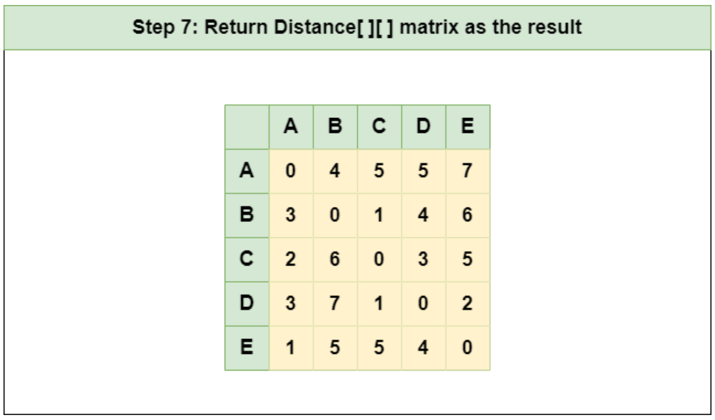
Step 6: Using Node E as the Intermediate node

Distance[i][j] = min (Distance[i][j], Distance[i][E] + Distance[E][j])

	A	B	C	D	E
A	?	?	?	?	7
B	?	?	?	?	6
C	?	?	?	?	5
D	?	?	?	?	2
E	1	5	5	4	0

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

Step 7*: Since all the nodes have been treated as an intermediate node, we can now return the updated Distance[][] matrix as our answer matrix.*



Step 7: Return Distance[ ][ ] matrix as the result

	A	B	C	D	E
A	0	4	5	5	7
B	3	0	1	4	6
C	2	6	0	3	5
D	3	7	1	0	2
E	1	5	5	4	0

- **Time Complexity:** O(V3), where V is the number of vertices in the graph and we run three nested loops each of size V
- **Auxiliary Space:** O(V2), to create a 2-D matrix in order to store the shortest distance for each pair of nodes.

Real World Applications of Floyd-Warshall Algorithm:

- In computer networking, the algorithm can be used to find the shortest path between all pairs of nodes in a network. This is termed as **network routing**.
- Flight Connectivity In the aviation industry to find the shortest path between the airports.
- **GIS**(**Geographic Information Systems**) applications often involve analyzing spatial data, such as road networks, to find the shortest paths between locations.

Single source shortest path algorithm

Find Shortest Paths from Source to all Vertices using Dijkstra's Algorithm.

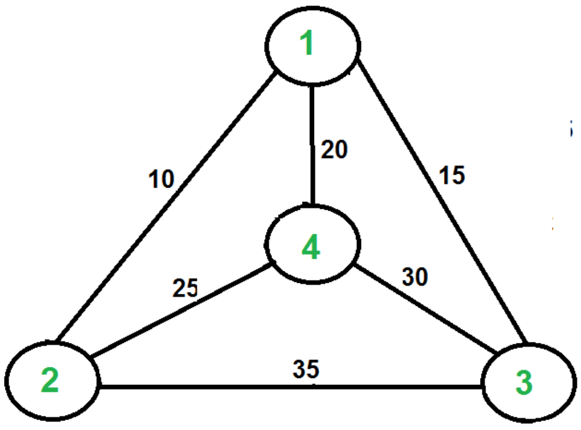
Travelling salesman problem using dynamic programming

Travelling Salesman Problem (TSP)

"Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?"

Given a set of cities and the distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point. Note the difference between Hamiltonian Cycle and TSP. The Hamiltonian cycle problem is to find if there exists a tour that visits every city exactly once. Here we know that Hamiltonian Tour exists (because the graph is complete) and in fact, many such tours exist, the problem is to find a minimum weight Hamiltonian Cycle.

(***Hamiltonian Cycle or Circuit*** *in a graph* ***G*** *is a cycle that visits every vertex of* ***G*** *exactly once and returns to the starting vertex.*)



For example, consider the graph shown in the figure on the right side.

A TSP tour in the graph is 1-2-4-3-1. The cost of the tour is 10+25+30+15 which is 80.

The problem is a famous NP hard problem.

The following are different solutions for the traveling salesman problem.

Naive Solution:

1) Consider city 1 as the starting and ending point.

2) Generate all (n-1)! Permutations of cities.

3) Calculate the cost of every permutation and keep track of the minimum cost permutation.

4) Return the permutation with minimum cost.

Dynamic Programming:

Let the given set of vertices be {1, 2, 3, 4,….n}. Let us consider 1 as starting and ending point of output.

For every other vertex I (other than 1), we find the minimum cost path with 1 as the starting point, I as the ending point, and all vertices appearing exactly once.

Let the cost of this path cost (i), and the cost of the corresponding Cycle would cost (i) + dist(i, 1) where dist(i, 1) is the distance from I to 1.

Finally, we return the minimum of all [cost(i) + dist(i, 1)] values. This looks simple so far.

To calculate the cost(i) using Dynamic Programming, we need to have some recursive relation in terms of sub-problems.

Let us define a term *C(S, i) be the cost of the minimum cost path visiting each vertex in set S exactly once, starting at 1 and ending at i*.

We start with all subsets of size 2 and calculate C(S, i) for all subsets where S is the subset, then we calculate C(S, i) for all subsets S of size 3 and so on. Note that 1 must be present in every subset.

If size of S is 2, then S must be {1, i},
C(S, i) = dist(1, i)
Else if size of S is greater than 2.
C(S, i) = min { C(S-{i}, j) + dis(j, i)} where j belongs to S, j != i and j != 1.

Qn: Find the greedy solution for following job sequencing with deadlines problem n = 7,

profit (p1,p2,p3,p4,p5,p6,p7) = (3,5,20,18,1,6,30),

deadline (d1,d2,d3,d4,d5,d6,d7) = (1,3,4,3,2,1,2)

The **Greedy Method** for scheduling or optimization problems typically involves making a sequence of choices that are locally optimal in the hope that they lead to a globally optimal solution. In this case, we are dealing with a **job scheduling problem** where we have seven tasks with associated profits and deadlines.

Here's the setup:

- **Profits (p)**: (3,5,20,18,1,6,30)(3,5,20,18,1,6,30)
- **Deadlines (d)**: (1,3,4,3,2,1,2)(1,3,4,3,2,1,2)

Each task i*i* has a profit pi*pi* and a deadline di*di*, where the deadline represents the latest time by which the task must be completed. The goal is to schedule these tasks within their deadlines to maximize the total profit.

Greedy Method for Job Scheduling

In this problem, the greedy method is typically used to **maximize profit** while ensuring that the tasks are completed within their respective deadlines. A common approach to solve this is:

1. **Sort the tasks by profit**: Start by sorting the tasks in **descending order** of their profits. This allows us to prioritize tasks that offer higher profits.
2. **Schedule each task**: For each task, find the latest available time slot (starting from its deadline and going backwards) and schedule it there. If a slot is already occupied, move to the next earliest time slot.
3. **Stop when all tasks are scheduled or no more slots are available**.

Step-by-Step Solution

Step 1: Sort tasks by profit

The first step is to sort the tasks based on their profit values in descending order.

- Tasks with profits p=(3,5,20,18,1,6,30) are sorted as:(30,20,18,6,5,3,1)
- Corresponding tasks (indices): (7,3,4,6,2,1,5)

Step 2: Initialize slots

Let's assume we have a number of slots available based on the maximum deadline. In this case, the highest deadline is 4, so we need 4 slots.

Step 3: Schedule the tasks

Now, let's schedule the tasks one by one:

- **Task 7 (profit = 30, deadline = 2)**: Try to schedule at time slot 2 (its deadline). Slot 2 is empty, so schedule Task 7 at time slot 2.
- **Task 3 (profit = 20, deadline = 4)**: Try to schedule at time slot 4 (its deadline). Slot 4 is empty, so schedule Task 3 at time slot 4.
- **Task 4 (profit = 18, deadline = 3)**: Try to schedule at time slot 3 (its deadline). Slot 3 is empty, so schedule Task 4 at time slot 3.
- **Task 6 (profit = 6, deadline = 1)**: Try to schedule at time slot 1 (its deadline). Slot 1 is empty, so schedule Task 6 at time slot 1.

Step 4: Tasks that couldn't be scheduled

- **Task 2 (profit = 5, deadline = 3)**: Deadline 3 is already filled by Task 4, and earlier slots are unavailable. Therefore, Task 2 cannot be scheduled.
- **Task 1 (profit = 3, deadline = 1)**: Deadline 1 is already filled by Task 6, so Task 1 cannot be scheduled.
- **Task 5 (profit = 1, deadline = 2)**: Deadline 2 is already filled by Task 7, so Task 5 cannot be scheduled.

Final Scheduled Tasks:

- Task 7 at time slot 2
- Task 3 at time slot 4
- Task 4 at time slot 3
- Task 6 at time slot 1

Total Profit:

The total profit is the sum of the profits of the scheduled tasks:

30+20+18+6=74

Thus, by using the **greedy method**, we have scheduled tasks to maximize the profit, resulting in a total profit of **74**.

Backtracking

It Is a problem solving strategy.

This uses Brute force approach. For any given problem we should try out all possible solutions and pick up desired solutions.

This uses in dynamic programming, but in DP we were solving for optimization pblms.

Backtracking is not used for optimization pblms.

Backtracking is a problem-solving algorithmic technique that involves finding a solution incrementally by trying **different options** and **undoing** them if they lead to a **dead end**.

It is commonly used in situations where you need to explore multiple possibilities to solve a problem, like searching for a path in a maze or solving puzzles like Sudoku.

When a dead end is reached, the algorithm backtracks to the previous decision point and explores a different path until a solution is found or all possibilities have been exhausted.

Backtracking can be defined as a general algorithmic technique that considers searching every possible combination in order to solve a computational problem.

Eg: if there are 3 students, 2 boys and 1 girl. And if there are 3 chairs, we have to arrange them in different ways. 3 students, so we can arrange them in 3! ,Ie. 6 ways.(n=3)

The possible arrangements are.

We can represent the solutions in the form of a tree.the tree is called **state space tree**.

We can arrange the chairs as, Students:B1,B2 and G1

B1	B2	G1

B1	G1	B2

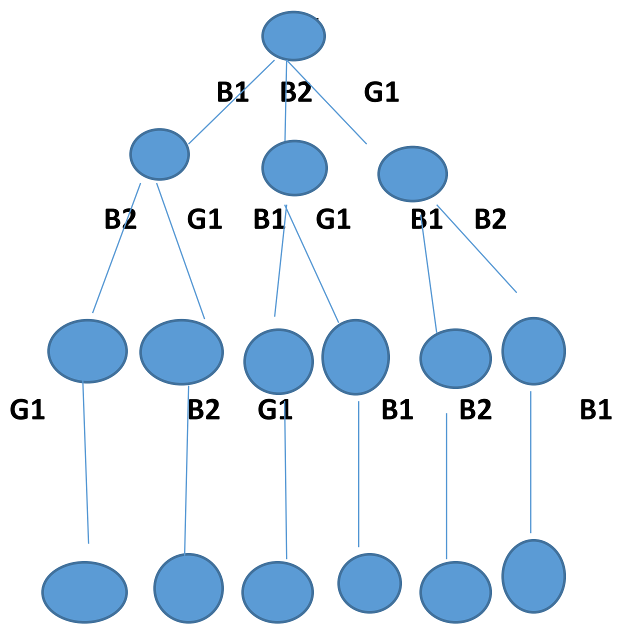
B2	B1	G1

B2	G1	B1

G1	B1	B2

G1	B2	B1

State space tree



In backtracking it uses some constraints (conditions) ,are called **Bounding Functions..**

Eg: girls should not sit in the middle of two boys.

Bounding function used to kill live nodes without generating all their children. Backtracking: Depth first generation with bounding function is known as Backtracking.

<u>Basic Terminologies</u>

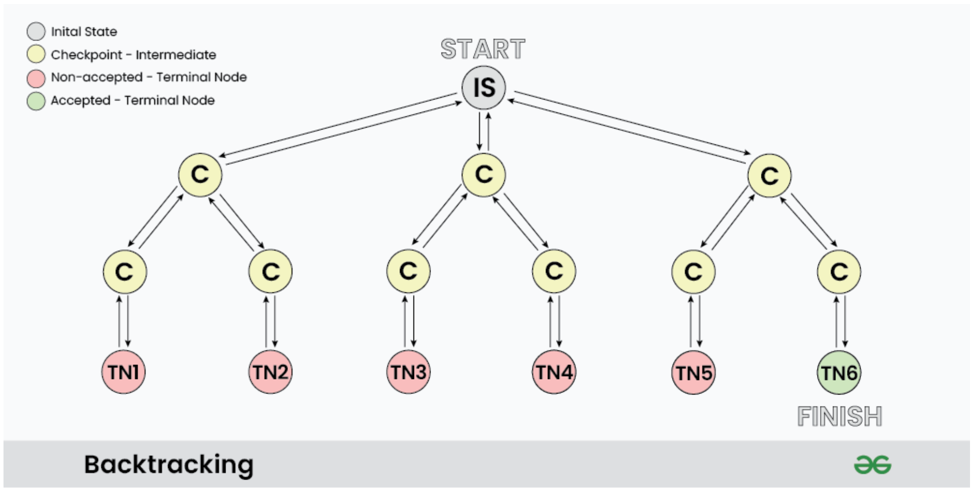
- **Candidate:** A candidate is a potential choice or element that can be added to the current solution.

- **Solution:** The solution is a valid and complete configuration that satisfies all problem constraints.
- **Partial Solution:** A partial solution is an intermediate or incomplete configuration being constructed during the backtracking process.
- **Decision Space:** The decision space is the set of all possible candidates or choices at each decision point.
- **Decision Point:** A decision point is a specific step in the algorithm where a candidate is chosen and added to the partial solution.
- **Feasible Solution:** A feasible solution is a partial or complete solution that adheres to all constraints.
- **Dead End:** A dead end occurs when a partial solution cannot be extended without violating constraints.
- **Backtrack:** Backtracking involves undoing previous decisions and returning to a prior decision point.
- **Search Space:** The search space includes all possible combinations of candidates and choices.
- **Optimal Solution:** In optimization problems, the optimal solution is the best possible solution.

Types of Backtracking Problems

Problems associated with backtracking can be categorized into 3 categories:

- **Decision Problems:** Here, we search for a feasible solution.
- **Optimization Problems:** For this type, we search for the best solution.
- **Enumeration Problems:** We find set of all possible feasible solutions to the problems of this type.



*Here ,"**IS**" represents the Here,***IS*** represents the **Initial State** where the recursion call starts to find a valid solution.*
C : *it represents different **Checkpoints** for recursive calls*

***TN**: it represents the **Terminal Nodes** where no further recursive calls can be made, these nodes act as base case of recursion and we determine whether the current solution is valid or not at this state.*

At each Checkpoint, our program makes some decisions and move to other checkpoints untill it reaches a terminal Node, after determining whether a solution is valid or not, the program starts to revert back to the checkpoints and try to explore other paths.

*For example in the above image **TN1…TN5** are the terminal node where the solution is not acceptable, while **TN6** is the state where we found a valid solution.*

The back arrows in the images shows backtracking in actions, where we revert the changes made by some checkpoint.

Let us take a simple problem:
***<u>Problem:</u>** Imagine you have 3 closed boxes, among which 2 are empty and 1 has a gold coin. Your task is to get the gold coin.*

***<u>Why dynamic programming fails to solve this question:</u> D**oes opening or closing one box has any effect on the other box? Turns out NO, each and every box is independent of each other and opening/closing state of one box can not determine the transition for other boxes. Hence DP fails.*

***<u>Why greedy fails to solve this question:</u>** Greedy algorithm chooses a local maxima in order to get global maxima, but in this problem each and every box has equal probability of having a gold coin i.e 1/3 hence there is no criteria to make a greedy choice.*

***<u>Why Backtracking works:</u>** As discussed already, backtracking algorithm is simply brute forcing each and every choice, hence we can one by one choose every box to find the gold coin, If a box is found empty we can close it back which acts as a Backtracking step. for backtracking problems:*

- *The algorithm builds a solution by exploring all possible paths created by the choices in the problem, this solution begins with an empty set **S={}***
- *Each choice creates a new sub-tree **'s'** which we add into are set.*
- *Now there exist two cases:*
    - ***S+s is valid set***
    - ***S+s is not valid set***
- *In case the set is valid then we further make choices and repeat the process until a solution is found, otherwise we backtrack our decision of including **'s'** and explore other paths until a solution is found or all the possible paths are exhausted.*

<u>Difference between Backtracking and Recursion</u>

Recursion	***Backtracking***
Recursion does not always need backtracking	*Backtracking always uses recursion to solve problems*
Solving problems by breaking them into smaller, similar subproblems and solving them recursively.	*Solving problems with multiple choices and exploring options systematically, backtracking when needed.*
Controlled by function calls and call stack.	*Managed explicitly with loops and state.*
***Applications of Recursion:** Tree and Graph Traversal, Towers of Hanoi, Divide and Conquer Algorithms, Merge Sort, Quick Sort, and Binary Search.*	***Application of Backtracking:** N Queen problem, Rat in a Maze problem, Knight's Tour Problem, Sudoku solver, and Graph coloring problems.*

Examples of Problems Solved by Backtracking:

1. N-Queens Problem:

- The goal is to place **N queens** on an **N x N** chessboard such that no two queens can attack each other (i.e., no two queens can share the same row, column, or diagonal).
- Backtracking explores different placements of queens row by row, and whenever a conflict arises, it backtracks to try another placement.

2. Sudoku Solver:

- In Sudoku, the goal is to fill a 9x9 grid with digits such that every row, every column, and each of the nine 3x3 subgrids contains all the digits from 1 to 9.
- Backtracking can be used to place digits in empty cells, ensuring no row, column, or subgrid contains duplicates.

3. Subset Sum Problem:

- Given a set of integers, the goal is to determine whether there is a subset that sums to a specific target.
- Backtracking explores different combinations of elements, pruning the search whenever the sum exceeds the target.

4. Graph Coloring:

- The problem is to assign colors to the vertices of a graph such that no two adjacent vertices have the same color, using a limited number of colors.
- Backtracking is used to assign colors to each vertex and backtrack when a vertex cannot be colored without violating the constraint.

Steps Involved in Backtracking:

1. **Choose:** Select the first available option.
2. **Constraints Check:** If this option violates any constraints, backtrack.
3. **Exploration:** If valid, recursively explore further options.
4. **Prune (if needed):** If at any point the solution becomes invalid, discard the current state (i.e., backtrack).
5. **Base Case:** If a complete solution is found, return it; otherwise, backtrack.

Time Complexity of Backtracking:

The time complexity of backtracking depends on the problem being solved.

In general, backtracking has an exponential time complexity because it explores all possible solutions in the worst case:

- **Worst-case time complexity** can be expressed as O(bd)O(b^d)O(bd), where:
    - bbb is the branching factor (the number of choices available at each step),
    - ddd is the depth of the search (i.e., the number of steps to a solution).

However, the use of **pruning** (i.e., backtracking) helps reduce the number of states to explore and can significantly improve performance, especially for problems with large solution spaces.

Advantages of Backtracking:

- **General-purpose approach:** Can be applied to a wide variety of problems.
- **Memory efficient:** In many cases, backtracking doesn't require storing all the solutions, only the current state of the solution.
- **Exhaustive:** Guarantees that all possible solutions are considered.

Disadvantages of Backtracking:

- **Inefficiency:** Backtracking can be inefficient for large search spaces if there is no pruning.
- **Exponential time complexity:** In the worst case, it might take too long to find a solution for large problems.

<u>Explicit and explicit constraints in backtracking</u>

Explicit constraints

Explicit constraints constitute well-defined imposed set of principles or conditions which must be met before a solution becomes acceptable. They are either given directly with the problem's text or extracted from its context.

Examples :

- **N-Queens Problem:** For example, placing two queens on any row, column or diagonal is prohibited according to the specifications of N-queens' problem.
- **Graph Coloring:** Also, coloring two adjacent vertices with same color is not allowed according to graph coloring problem specification.
- **Sudoku Puzzle:** This explicitly includes that every row, column and 3×3 sub-grid should contain digits from 1 through n in no particular order except for no repetition within each individual cell of itself.

Implicit Constraints

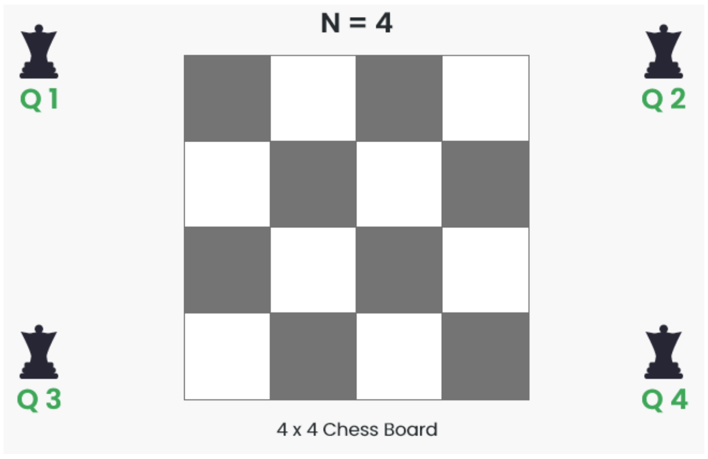
On the other hand, implicit constraints are less visible than explicit ones because they are only implicitly mentioned in the problem statement. Implicit constraints may be based on the nature of a problem.

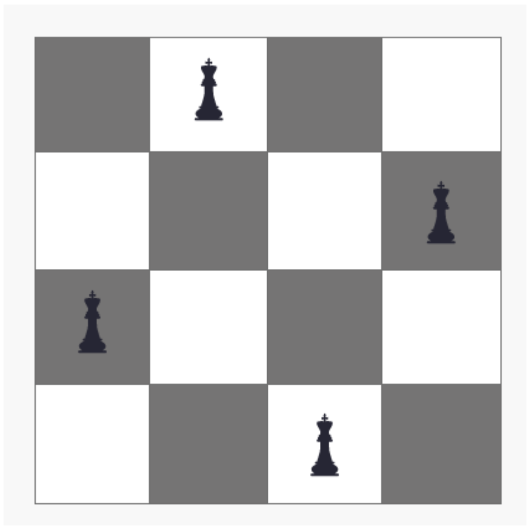
The main difference between implicit and explicit constraints concerns their visibility and definiteness.

Example for implicit constraints: consider the Traveling Salesman Problem that requires visiting all cities exactly once. However, one could find a solution here by minimizing the total distance traveled on the shortest possible route.

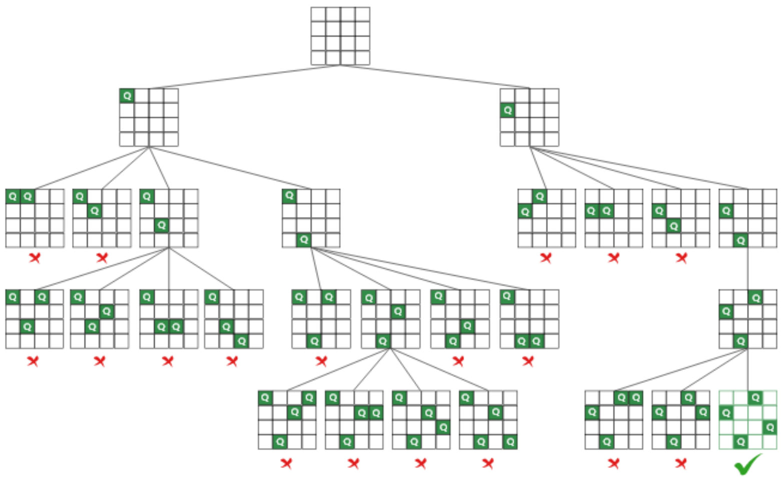
N-Queen problem

The **N** Queen is the problem of placing **N** chess queens on an **N×N** chessboard so that no two queens attack each other.





Solution1= 2,4,1,3, Solution2 = 3,1,4,2



<ins>**Branch and Bound Algorithm**</ins>

The **Branch and Bound Algorithm** is a method used in **combinatorial** optimization problems to systematically search for the best solution. It works by dividing the problem into smaller subproblems, or branches, and then eliminating certain branches based on bounds on the optimal solution. This process continues until the best solution is found or all branches have been explored.
Branch and Bound is commonly used in problems like the **traveling salesman** and **job scheduling**.
It also uses the state space tree.

Traveling Salesman Problem (TSP), 0/1 Knapsack Problem, Integer Linear Programming, and others are solve using **BB**. The main idea of B&B is to break down a problem into subproblems (branching), solve them in an efficient way, and use bounding techniques to eliminate subproblems that cannot lead to an optimal solution.

Key Concepts

1. **Branching**:
    - Involves dividing the problem into smaller subproblems (branches).
    - Each subproblem represents a partial solution or a "candidate" for the final solution.
    - This is typically done by recursively breaking the problem into two or more subproblems.
2. **Bounding**:
    - Bounding is used to evaluate the upper or lower bounds of a partial solution (depending on whether you are solving a maximization or minimization problem).
    - The bound gives an estimate of the best possible solution that could be obtained from a particular subproblem.

  - If the bound indicates that the current subproblem cannot lead to an optimal solution (because it's worse than the best known solution), that subproblem is discarded (pruned).

3. **Pruning**:
   - Pruning refers to discarding or ignoring certain subproblems that do not need to be explored further, based on the bounding condition.
   - The goal of pruning is to avoid unnecessary computation and reduce the size of the search space.

4. **Solution Space Tree**:
   - B&B explores the solution space as a tree structure, where each node represents a partial solution and edges represent decisions made to build the solution.
   - The algorithm proceeds by exploring nodes in the tree, branching out and bounding as necessary.

Steps in Branch and Bound

1. **Initialization**:
   - Start with an initial solution (e.g., the best known feasible solution).
   - Initialize the best solution bound (lower bound for minimization or upper bound for maximization).

2. **Branching**:
   - Choose a subproblem (a node in the search tree) and branch it into sub-subproblems. This means you generate child nodes for that node.
   - For example, in the 0/1 Knapsack problem, branching might correspond to deciding whether an item is included in the knapsack or not.

3. **Bounding**:
   - For each subproblem, calculate an upper or lower bound.
   - For minimization problems, the bound can be the least value achievable from the current partial solution.
   - For maximization problems, the bound can be the highest value achievable.

4. **Pruning**:
   - If a subproblem's bound is worse than the current best solution, discard it (prune the branch).
   - If a subproblem's bound is better, explore further.

5. **Repeat**:
   - Continue the process until the search tree is exhausted or an optimal solution is found.

6. **Return Solution**:
    - Once the search is complete, the best solution found so far is returned as the optimal solution.

Example: 0/1 Knapsack Problem

In the **0/1 Knapsack problem**, the objective is to select a subset of items such that the total weight does not exceed the capacity of the knapsack, and the total value of the selected items is maximized.

Branch and Bound for 0/1 Knapsack:

1. **Branching**:
    - At each step, decide whether to include an item in the knapsack or exclude it. This creates two branches from each node: one where the item is included, and one where it is not.
2. **Bounding**:
    - For each subproblem, calculate the maximum possible value that can be obtained, given the remaining items and the remaining capacity.
    - This is typically done using a greedy heuristic like fractional knapsack to get a quick upper bound (maximal possible value given the remaining capacity).
3. **Pruning**:
    - If a subproblem's bound is worse than the current best known solution (either the remaining capacity can't accommodate the items left or the total value is too low), prune that branch.
4. **Optimal Solution**:
    - Once all the branches are explored, the best solution (maximum value with weight <= capacity) will be the optimal solution.

Example of Branch and Bound in Action

Let's consider a small problem with three items and a knapsack of capacity 5:

- Item 1: weight = 2, value = 3
- Item 2: weight = 3, value = 4
- Item 3: weight = 4, value = 5

The total capacity of the knapsack is 5, so you need to select the best combination of items such that the weight does not exceed the capacity.

Step-by-Step Execution:

1. **Root Node (Start)**:
    - Initial capacity: 5, no items selected.
    - Branching: Two subproblems — include Item 1 or exclude Item 1.
2. **Branch 1 (Include Item 1)**:

    - Remaining capacity = 3 (5 - 2).
    - Branch further: Include Item 2 or exclude Item 2.
3. **Branch 1.1 (Include Item 2)**:
    - Remaining capacity = 0 (3 - 3).
    - Cannot include Item 3, because the remaining capacity is 0.
    - Current value = 3 + 4 = 7 (best solution so far).
4. **Branch 1.2 (Exclude Item 2)**:
    - Remaining capacity = 3.
    - Include Item 3 is not possible due to weight > remaining capacity.
    - Current value = 3 (best solution so far).
5. **Branch 2 (Exclude Item 1)**:
    - Remaining capacity = 5.
    - Branch further: Include Item 2 or exclude Item 2.

... and so on.

By pruning branches where it is clear that no better solution can be found (e.g., when remaining capacity is insufficient), the algorithm will eventually find the optimal selection of items.

Applications of Branch and Bound

- **Traveling Salesman Problem (TSP)**: Find the shortest possible route that visits each city exactly once and returns to the starting point.
- **0/1 Knapsack Problem**: Maximize the total value of items placed in a knapsack with a fixed capacity.
- **Integer Linear Programming (ILP)**: Solve problems with linear constraints and integer variables.
- **Job Scheduling**: Minimize the completion time or cost in job scheduling problems.

Advantages and Disadvantages

Advantages:

- Can provide an exact optimal solution.
- Efficient pruning can reduce the search space significantly.

Disadvantages:

- For large problem spaces, the algorithm can be slow if the branching factor is too high.
- The effectiveness of the algorithm depends heavily on the bounding function and how well it can prune suboptimal solutions.

LC Search

Least Cost Search (LCS) is a search algorithm used in the context of **Decision Analysis and Algorithms** (DAA), as well as in fields such as Artificial Intelligence and Operations Research. It focuses on finding a path through a problem space that minimizes the total cost of traversing that path. In general terms, LCS refers to a search method that explores a problem's state space by always choosing the path that appears to have the least cost at each step.

In the context of **Designn and Analysis (DAA)**, Least Cost Search might be used to explore decision spaces where you are trying to find the optimal sequence of decisions that minimizes a particular cost or maximizes utility. The basic steps in an LCS in this context would be:

1. **Define the State Space**: Identify all possible states and actions that can be taken in your problem domain.
2. **Define the Cost Function**: Assign a cost to each state or transition based on your decision criteria. For example, in routing problems, the cost could represent travel distance or time.
3. **Apply Least Cost Search Algorithm**:
    - Start from the initial state.
    - Explore the state space, moving to adjacent states in a manner that minimizes the cost at each step.
    - Use either an informed or uninformed search strategy to decide which state to explore next.
4. **Check for Goal State**: Once the algorithm reaches a state where the goal condition is satisfied, return the path that leads to that state with the least cost.

Algorithm Structure:

The structure typically follows these steps:

1. **Initialize**: Start with a priority queue (min-heap) to keep track of the least cost paths.
2. **Explore**: Pop the node with the lowest cost from the priority queue and explore its neighbors.
3. **Update Costs**: If a neighbor offers a cheaper cost path, update its cost and insert it back into the priority queue.
4. **Repeat**: Continue this process until the goal is found or all states have been explored.
5. **Path Construction**: Once the goal is reached, reconstruct the path taken to reach it.

Example of Least Cost Search (Uniform Cost Search):

Finding the shortest path between two cities in a map where each road between cities has a specific travel cost. You can apply a Least Cost Search (such as UCS) where:

- Each city is a node.
- Each road between cities has a cost (e.g., travel time or distance).
- The goal is to reach a target city with the least total travel cost.

The algorithm will:

- Start at the source city.
- Explore neighboring cities, adding the travel cost to each neighboring city.
- Keep track of the minimum cost path to each city in a priority queue.
- Eventually, the algorithm will reach the destination city with the minimum possible cost.

Comparison with Other Search Algorithms:

- **Depth-First Search (DFS)**: DFS doesn't consider the cost and may not find the least cost path.
- **Breadth-First Search (BFS)**: BFS finds the shortest path in terms of the number of steps (or uniform cost), but it does not handle non-uniform costs.
- **A* Search**: A* improves on Least Cost Search by using both the path cost and a heuristic estimate of the remaining cost to the goal, often leading to more efficient search.

Application Areas:

- **Routing and Navigation**: Finding the cheapest or shortest path in transportation networks, like GPS systems.
- **Planning**: In decision-making processes, minimizing cost can be used to evaluate various strategies.
- **Resource Allocation**: Minimizing resource usage or costs in scheduling or manufacturing.

Module IV

Time Complexities of all Sorting Algorithms

The efficiency of an algorithm depends on two parameters:

1. Time Complexity
2. Space Complexity

Time Complexity: Time Complexity is defined as the number of times a particular instruction set is executed rather than the total time taken. It is because the total time taken also depends on some external factors like the compiler used, the processor's speed, etc.

Space Complexity: Space Complexity is the total memory space required by the program for its execution.

Both are calculated as the function of input size(n). One important thing here is that despite these parameters, the efficiency of an algorithm also depends upon the **nature** and **size of** the **input.**

Types of Time Complexity :

1. **Best Time Complexity:** Define the input for which the algorithm takes less time or minimum time. In the best case calculate the lower bound of an algorithm. Example: In the linear search when search data is present at the first location of large data then the best case occurs.
2. **Average Time Complexity:** In the average case take all random inputs and calculate the computation time for all inputs.
   And then we divide it by the total number of inputs.
3. **Worst Time Complexity:** Define the input for which algorithm takes a long time or maximum time. In the worst calculate the upper bound of an algorithm. Example: In the linear search when search data is present at the last location of large data then the worst case occurs.

Algorithm	**Time Complexity**			**Space Complexity**
	Best	**Average**	**Worst**	**Worst**
Selection Sort	O(n2)	O(n2)	O(n2)	O(1)
Bubble Sort	O(n)	O(n2)	O(n2)	O(1)
Insertion Sort	O(n)	O(n2)	O(n2)	O(1)

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Heap Sort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Quick Sort	O(n log(n))	O(n log(n))	O(n2)	O(n)
Merge Sort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)

Deterministic and Non deterministic algorithms

A deterministic algorithm is an algorithm that, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states.

A deterministic algorithm computes a mathematical function; a function has a unique value for any input in its domain, and the algorithm is a process that produces this particular value as output.

Deterministic algorithms can be defined in terms of a state machine:

A *state* describes what a machine is doing at a particular instant in time.

State machines pass in a discrete manner from one state to another.

Just after we enter the input, the machine is in its *initial state* or *start state*.

If the machine is deterministic, this means that from this point onwards, its current state determines what its next state will be; its course through the set of states is predetermined.

Non-deterministic algorithms

A variety of factors can cause an algorithm to behave in a way which is not deterministic, or non-deterministic:

- If it uses an external state other than the input, such as user input, a global variable, a hardware timer value, a random value, or stored disk data.
- If it operates in a way that is timing-sensitive, for example, if it has multiple processors writing to the same data at the same time. In this case, the precise order in which each processor writes its data will affect the result.
- If a hardware error causes its state to change in an unexpected way.

Although real programs are rarely purely deterministic,

Disadvantages of determinism

It is advantageous, in some cases, for a program to exhibit nondeterministic behavior.

The behavior of a card shuffling program used in a game of blackjack, for example, should not be predictable by players — even if the source code of the program is visible.

The use of a pseudorandom number generator is often not sufficient to ensure that players are unable to predict the outcome of a shuffle. A clever gambler might guess precisely the numbers the generator will choose and so determine the entire contents of the deck ahead of time, allowing him to cheat;

For example, the Software Security Group at Reliable Software Technologies was able to do this for an implementation of Texas Hold 'em Poker that is distributed by ASF Software, Inc, allowing them to consistently predict the outcome of hands ahead of time.[7] These problems can be avoided, in through the use of a cryptographically secure pseudo-random number generator. Hence non deterministic algorithm is needed.

Difference between Deterministic and Non Deterministic Algorithm

Key Aspect	Deterministic Algorithm	Non-deterministic Algorithm
Definition	Deterministic algorithms produce uniquely defined results. They perform a fixed number of steps and always finish with an accept or reject state with the same result.	Non-deterministic algorithms do not produce uniquely defined results. The outcome may be random or not uniquely determined.
Execution	In deterministic algorithms, the target machine executes the same instructions and yields the same outcome, independent of the way or process of execution.	In non-deterministic algorithms, the machine executing each operation can choose any of several outcomes, subject to a determination condition defined later.
Type	Deterministic algorithms are classified as reliable algorithms. They consistently produce the same output for the same input instructions.	Non-deterministic algorithms are considered non-reliable algorithms because they may produce different outputs for the same input.
Execution Time	Deterministic algorithms typically execute in polynomial time since the	Non-deterministic algorithms often cannot be executed in polynomial time

Key Aspect	Deterministic Algorithm	Non-deterministic Algorithm
	outcome is known and consistent across executions.	due to their unpredictable nature.
Execution Path	In deterministic algorithms, the execution path remains the same in every execution.	In non-deterministic algorithms, the execution path varies between executions and may take different, random paths.

Q. **Can you provide an example of a deterministic algorithm?**

A. An example of a deterministic algorithm is binary search. Given a sorted list, binary search follows a fixed process to find a specific element, and it always produces the same result for the same input.

Q. **What are some characteristics of deterministic algorithms?**

A. Deterministic algorithms are predictable, repeatable, and produce the same result for the same input. They are typically used when consistency and reliability of results are essential.

Q. **When are non-deterministic algorithms used?**

A. Non-deterministic algorithms are often used in scenarios where randomness, choices, or exploring multiple possibilities are required. Examples include randomized algorithms and some optimization problems.

P, NP, CoNP, NP hard and NP complete | Complexity Classes

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 group problems based on how much time and space they require to solve problems and verify the solutions.

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 organising similar types of problems.

Types of Complexity Classes

The classes:

- P Class
- NP Class
- CoNP Class
- NP-hard
- 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:

- The solution to P problems is easy to find.
- 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.

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.

Feature:

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.

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 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 co-worker is satisfactory or not

i.e. no pair taken from the co-worker 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:

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.

If a problem X is in NP, then its complement X' is also in CoNP.

Example problems for CoNP are:

- To check prime number.
- Integer Factorization.

NP-hard class

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

Features:

- All NP-hard problems are not in NP.
- 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.
- A problem A is in NP-hard if, for every problem L in NP, there exists a polynomial-time reduction from L to A.

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:

NP-complete problems are special as any problem in NP class can be transformed or reduced into NP-complete problems in polynomial time.

If one could solve an NP-complete problem in polynomial time, then one could also solve any NP problem in polynomial time.

Example:Hamiltonian Cycle.

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.

****