UNIT-3

BRUTE FORCE AND EXHAUSTIVE SEARCH

INTRODUCTION TO BRUTE FORCE APPROACH:

Brute force approach

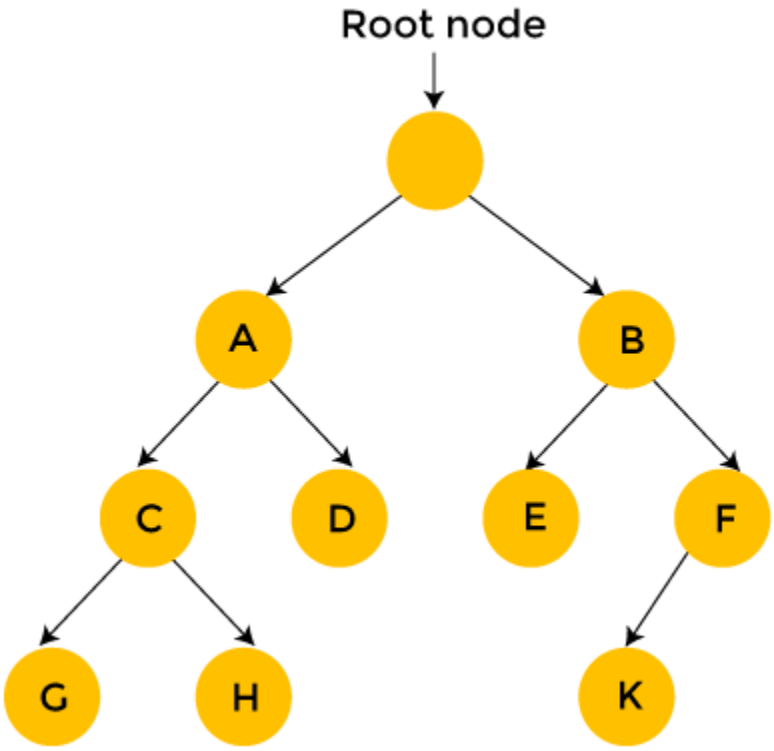
A brute force approach is an approach that finds all the possible solutions to find a satisfactory solution to a given problem. The brute force algorithm tries out all the possibilities till a satisfactory solution is not found.

Such an algorithm can be of two types:

- **Optimizing:** In this case, the best solution is found. To find the best solution, it may either find all the possible solutions to find the best solution or if the value of the best solution is known, it stops finding when the best solution is found. For example: Finding the best path for the travelling salesman problem. Here best path means that travelling all the cities and the cost of travelling should be minimum.
- **Satisficing:** It stops finding the solution as soon as the satisfactory solution is found. Or example, finding the travelling salesman path which is within 10% of optimal.
- Often Brute force algorithms require exponential time. Various heuristics and optimization can be used:
- **Heuristic:** A rule of thumb that helps you to decide which possibilities we should look at first.
- **Optimization:** A certain possibilities are eliminated without exploring all of them.

Let's understand the brute force search through an example.

Suppose we have converted the problem in the form of the tree shown as below:



Brute force search considers each and every state of a tree, and the state is represented in the form of a node. As far as the starting position is concerned, we have two choices, i.e., A state and B state. We can either generate state A or state B. In the case of B state, we have two states, i.e., state E and F.

In the case of brute force search, each state is considered one by one. As we can observe in the above tree that the brute force search takes 12 steps to find the solution.

On the other hand, backtracking, which uses Depth-First search, considers the below states only when the state provides a feasible solution. Consider the above tree, start from the root node, then move to node A and then node C. If node C does not provide the feasible solution, then there is no point in considering the states G and H. We backtrack from node C to node A. Then, we move from node A to node D. Since node D does not provide the feasible solution, we discard this state and backtrack from node D to node A.

We move to node B, then we move from node B to node E. We move from node E to node K; Since k is a solution, so it takes 10 steps to find the solution. In this way, we eliminate a greater number of states in a single iteration. Therefore, we can say that backtracking is faster and more efficient than the brute force approach.

Advantages of a brute-force algorithm

The following are the advantages of the brute-force algorithm:

- This algorithm finds all the possible solutions, and it also guarantees that it finds the correct solution to a problem.
- This type of algorithm is applicable to a wide range of domains.
- It is mainly used for solving simpler and small problems.
- It can be considered a comparison benchmark to solve a simple problem and does not require any particular domain knowledge.

Disadvantages of a brute-force algorithm

The following are the disadvantages of the brute-force algorithm:

- It is an inefficient algorithm as it requires solving each and every state.
- It is a very slow algorithm to find the correct solution as it solves each state without considering whether the solution is feasible or not.
- The brute force algorithm is neither constructive nor creative as compared to other algorithms.

SELECTION SORT ALGORITHM:

Selection Sort

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

ALGORITHM: SELECTION SORT (A)

1. 1. k ← length [A]
2. 2. **for** j ←1 to n-1
3. 3. smallest ← j
4. 4. **for** I ← j + 1 to k
5. 5. **if** A [i] < A [smallest]
6. 6. then smallest ← i
7. 7. exchange (A [j], A [smallest])

How Selection Sort works

1. In the selection sort, first of all, we set the initial element as a **minimum**.
2. Now we will compare the minimum with the second element. If the second element turns out to be smaller than the minimum, we will swap them, followed by assigning to a minimum to the third element.
3. Else if the second element is greater than the minimum, which is our first element, then we will do nothing and move on to the third element and then compare it with the minimum.
   We will repeat this process until we reach the last element.
4. After the completion of each iteration, we will notice that our minimum has reached the start of the unsorted list.
5. For each iteration, we will start the indexing from the first element of the unsorted list. We will repeat the Steps from 1 to 4 until the list gets sorted or all the elements get correctly positioned.
   Consider the following example of an unsorted array that we will sort with the help of the Selection Sort algorithm.

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

1st Iteration:

Set minimum = 7

- Compare a_0 and a_1

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

- Compare a_1 and a_2

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

- Compare a_2 and a_3

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

- Compare a_2 and a_4

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

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

2nd Iteration:

Set minimum = 4

- Compare a_1 and a_2

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

- Compare a_1 and a_3

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

- Compare a_1 and a_4

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

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

3rd Iteration:

Set minimum = 7

- Compare a_2 and a_3

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

- Compare a_3 and a_4

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

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

4th Iteration:

Set minimum = 6

- Compare a_3 and a_4

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

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

Complexity Analysis of Selection Sort

Input: Given **n** input elements.

Output: Number of steps incurred to sort a list.

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

Output;

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

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

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

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

Time Complexities:

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

In the selection sort algorithm, the time complexity is **$O(n^2)$** in all three cases. This is because, in each step, we are required to find **m**

BUBBLE SORT

DAA Bubble Sort

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

This is the easiest method among all sorting algorithms.

Algorithm

Step 1 ➤ Initialization

1. set 1 ← n, p ← 1

Step 2 ➤ loop,

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

Step 3 ➤ comparison, loop.

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

Step 4 ➤ Finish, or reduce the size.

1. **if** (E = 0) then
2. exit
3. **else**
4. set l ← l - 1.

How Bubble Sort Works

1. The bubble sort starts with the very first index and makes it a bubble element. Then it compares the bubble element, which is currently our first

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

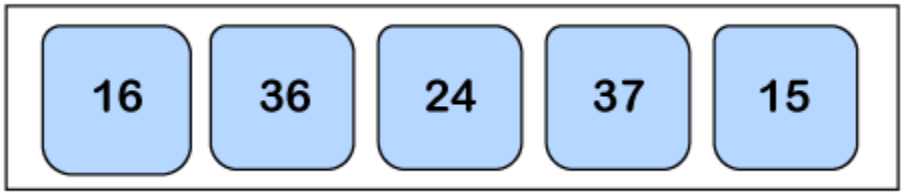
2. We will follow the same process for the rest of the iterations. After each of the iteration, we will notice that the largest element present in the unsorted array has reached the last index.

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

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

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

Initially,



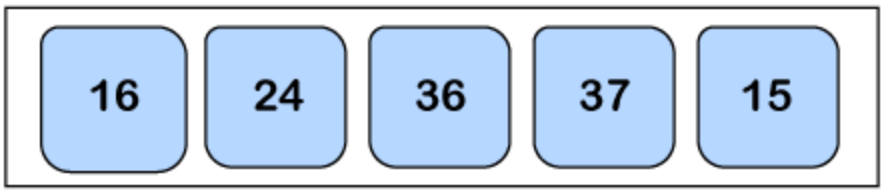
Pass 1:

- **Compare a_0 and a_1**

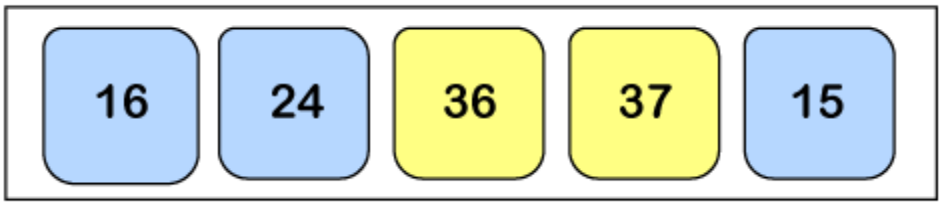
As $a_0 < a_1$ so the array will remain as it is.

- **Compare a_1 and a_2**

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



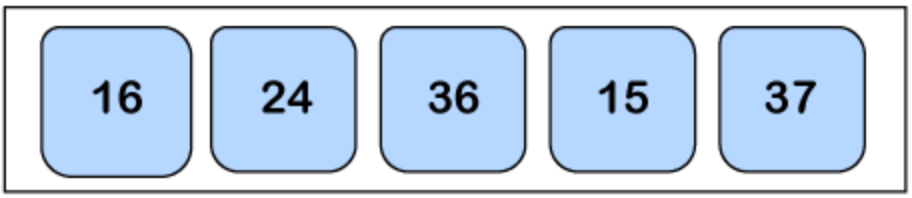
- **Compare a_2 and a_3**



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

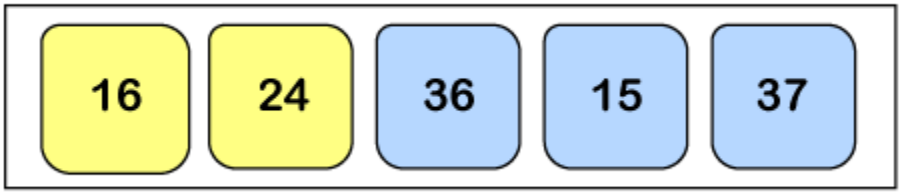
- **Compare a_3 and a_4**

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



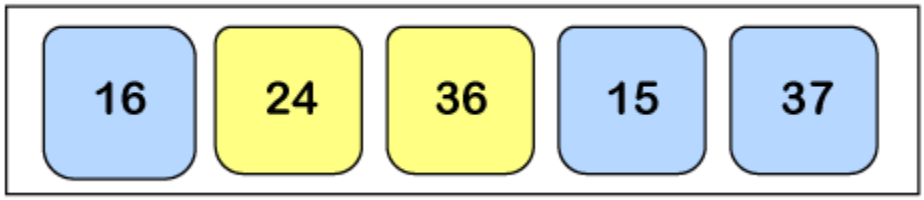
Pass 2:

- **Compare a_0 and a_1**



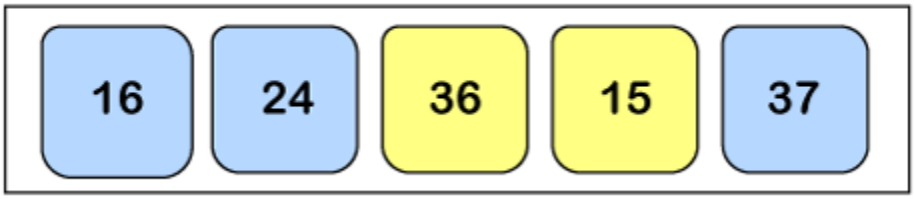
As $a_0 < a_1$ so the array will remain as it is.

- **Compare a_1 and a_2**

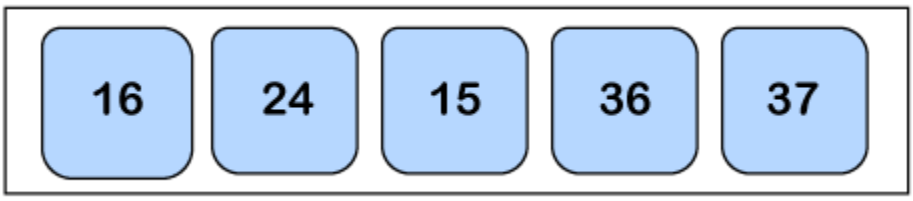


Here $a_1 < a_2$, so the array will remain as it is.

- **Compare a_2 and a_3**



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



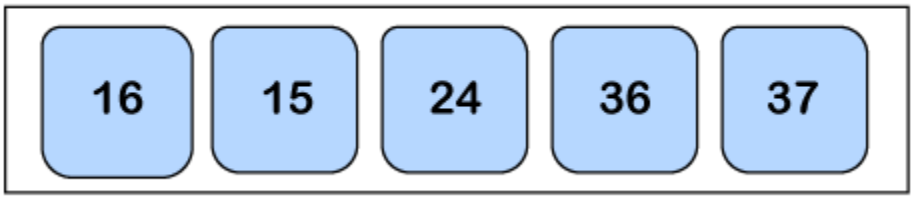
Pass 3:

- **Compare a_0 and a_1**

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

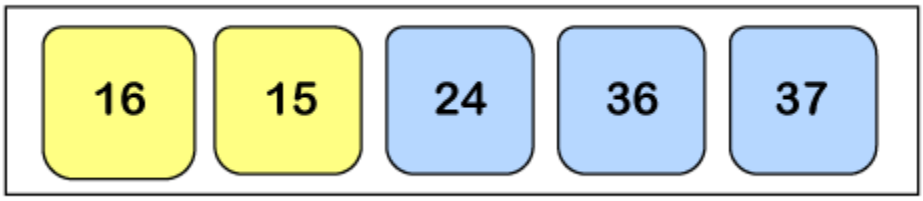
- **Compare a_1 and a_2**

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

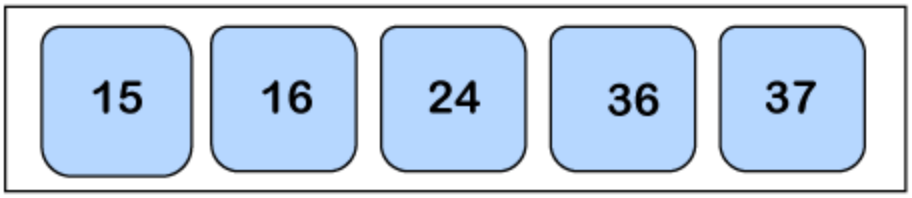


Pass 4:

- **Compare a_0 and a_1**



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



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

Complexity Analysis of Bubble Sort

Input: Given n input elements.

Output: Number of steps incurred to sort a list.

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

Output;

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

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

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

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

Time Complexities:

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

Advantages of Bubble Sort

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

Disadvantages of Bubble Sort

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

SEQUENTIAL SEARCH ALGORITHM

Sequential Search

Sequential Search, also called *linear search*, is the simplest of all searching algorithms. It is a brute-force approach to locating a single target value, *t*, in some collection, *C*. It finds *t* by starting at the first element of the collection and examining each subsequent element until either the matching element is found or each element of the collection has been examined.

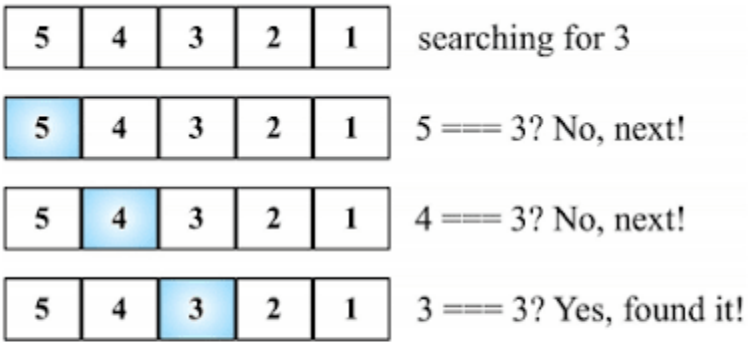
Example: Sequential search

ALGORITHM *SequentialSearch*$(A[0..n-1], K)$

```
//Searches for a given value in a given array by sequential search
//Input: An array A[0..n − 1] and a search key K
//Output: The index of the first element of A that matches K
//        or −1 if there are no matching elements
i ← 0
while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return −1
```

- What's the …
  - Worst case
  - Best case
  - Average case

12



Sequential Search Algorithm

www.programmingboss.com

Best Case Complexity - In Linear search, best case occurs when the element we are finding is at the first position of the array.

• Average Case Complexity - The average case time complexity of linear search is O(n).

• Worst Case Complexity - In Linear search, the worst case occurs when the element we are looking is present at the end of the array.

The worst-case in linear search could be when the target element is not present in the given array, and we have to traverse the entire array.

CASE TIME COMPLEXITY

Best Case O(1)

Average Case O(n)

Worst Case O(n)

EXHAUSTIVE SEARCH

Exhaustive Search Algorithm:
Exhaustive Search is a brute-force algorithm that systematically enumerates all possible solutions to a problem and checks each one to see if it is a valid solution. This algorithm is typically used for problems that have a small and well-defined search space, where it is feasible to check all possible solutions.

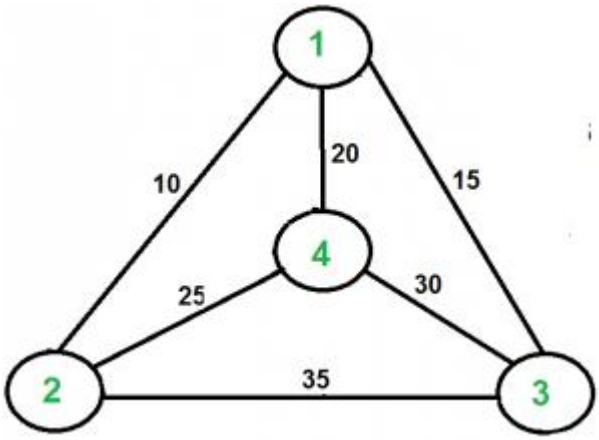
Examples:
Input: Items[] = {1, 2, 3, 4, 5, 6};
List of sets = {1, 2, 3}, {4, 5}, {5, 6}, {1, 4}
Output: Maximum number of sets that can be packed: 3
Input: Items[] = {1, 2};
List of sets = {1}, {4, }, {5}, {1}
Output: Maximum number of sets that can be packed: 1
Approach:
Loop through all the sets and check if the current item is in the current set If the item is in the set, increment the number of sets that can be packed and Update the maximum number of sets that can be packed

TRAVELLING SALESMAN PROBLEM

Traveling Salesman Problem (TSP) Implementation

Given a set of cities and the distan
ce 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

Travelling Salesman Problem (TSP) : Given a set of cities and distances 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.
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. There is no polynomial-time known solution for this problem.



Examples:
Output of Given Graph:

minimum weight Hamiltonian Cycle :

10 + 25 + 30 + 15 := 80

1. Consider city 1 as the starting and ending point. Since the route is cyclic, we can consider any point as a starting 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.

Time complexity: O(n!) where n is the number of vertices in the graph. This is because the algorithm uses the next_permutation function which generates all the possible permutations of the vertex set.

KNAPSACK PROBLEM:

0/1 Knapsack Problem

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*

Recursion Approach for 0/1 Knapsack Problem:
To solve the problem follow the below idea:
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.*

Follow the below steps to solve the problem:
The maximum value obtained from ‘N’ items is the max of the following two values.

- Case 1 (include the Nth item): Value of the Nth item plus maximum value obtained by remaining N-1 items and remaining weight i.e. (W-weight of the Nth item).
- Case 2 (exclude the Nth item): Maximum value obtained by N-1 items and W weight.
- If the weight of the 'Nth' item is greater than 'W', then the Nth item cannot be included and **Case 2** is the only possibility.

Time Complexity: O(2N)
Auxiliary Space: O(N), Stack space required for recursion

DEPTH FIRST SEARCH ALGORITHM

Depth-First Search or DFS algorithm is a recursive algorithm that uses the backtracking principle. It entails conducting exhaustive searches of all nodes by moving forward if possible and backtracking, if necessary. To visit the next node, pop the top node from the stack and push all of its nearby nodes into a stack

DFS (Depth First Search) algorithm

In this article, we will discuss the DFS algorithm in the data structure. It is a recursive algorithm to search all the vertices of a tree data structure or a graph. The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

Because of the recursive nature, stack data structure can be used to implement the DFS algorithm. The process of implementing the DFS is similar to the BFS algorithm.

The step by step process to implement the DFS traversal is given as follows -

1. First, create a stack with the total number of vertices in the graph.
2. Now, choose any vertex as the starting point of traversal, and push that vertex into the stack.
3. After that, push a non-visited vertex (adjacent to the vertex on the top of the stack) to the top of the stack.
4. Now, repeat steps 3 and 4 until no vertices are left to visit from the vertex on the stack's top.
5. If no vertex is left, go back and pop a vertex from the stack.

6. Repeat steps 2, 3, and 4 until the stack is empty.

Applications of DFS algorithm

The applications of using the DFS algorithm are given as follows -

- DFS algorithm can be used to implement the topological sorting.
- It can be used to find the paths between two vertices.
- It can also be used to detect cycles in the graph.
- DFS algorithm is also used for one solution puzzles.
- DFS is used to determine if a graph is bipartite or not.

Algorithm

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

Step 6: EXIT

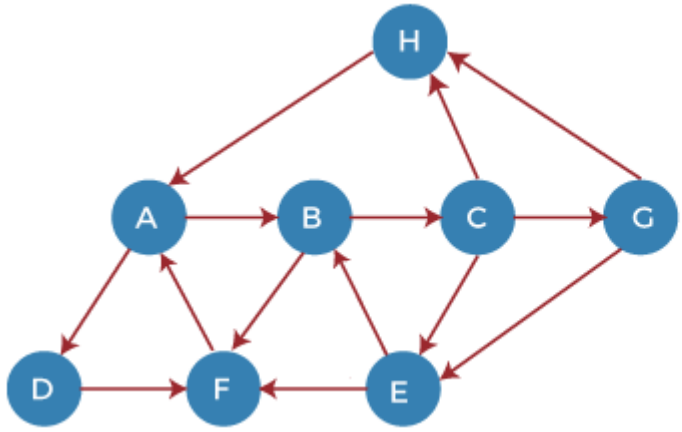
Pseudocode

```
1. DFS(G,v)   ( v is the vertex where the search starts )
2. Stack S := {};   ( start with an empty stack )
3. for each vertex u, set visited[u] := false;
4. push S, v;
5. while (S is not empty) do
6. u := pop S;
7. if (not visited[u]) then
8. visited[u] := true;
9. for each unvisited neighbour w of uu
```

10. push S, w;
11. end **if**
12. end **while**
13. END DFS()

Example of DFS algorithm

Now, let's understand the working of the DFS algorithm by using an example. In the example given below, there is a directed graph having 7 vertices.



Adjacency Lists

A : B, D
B : C, F
C : E, G, H
G : E, H
E : B, F
F : A
D : F
H : A

Now, let's start examining the graph starting from Node H.

Step 1 - First, push H onto the stack.

1. STACK: H

Step 2 - POP the top element from the stack, i.e., H, and print it. Now, PUSH all the neighbors of H onto the stack that are in ready state.

1. Print: H]STACK: A

Step 3 - POP the top element from the stack, i.e., A, and print it. Now, PUSH all the neighbors of A onto the stack that are in ready state.

1. Print: A
2. STACK: B, D

Step 4 - POP the top element from the stack, i.e., D, and print it. Now, PUSH all the neighbors of D onto the stack that are in ready state.

1. Print: D

2. STACK: B, F

Step 5 - POP the top element from the stack, i.e., F, and print it. Now, PUSH all the neighbors of F onto the stack that are in ready state.

1. Print: F
2. STACK: B

Step 6 - POP the top element from the stack, i.e., B, and print it. Now, PUSH all the neighbors of B onto the stack that are in ready state.

1. Print: B
2. STACK: C

Step 7 - POP the top element from the stack, i.e., C, and print it. Now, PUSH all the neighbors of C onto the stack that are in ready state.

1. Print: C
2. STACK: E, G

Step 8 - POP the top element from the stack, i.e., G and PUSH all the neighbors of G onto the stack that are in ready state.

1. Print: G
2. STACK: E

Step 9 - POP the top element from the stack, i.e., E and PUSH all the neighbors of E onto the stack that are in ready state.

1. Print: E
2. STACK:

Now, all the graph nodes have been traversed, and the stack is empty.

Complexity of Depth-first search algorithm

The time complexity of the DFS algorithm is **O(V+E)**, where V is the number of vertices and E is the number of edges in the graph.

The space complexity of the DFS algorithm is O(V).

BFS Algorithm

What is BFS?

Breadth-First Search (BFS) is based on traversing nodes by adding the neighbors of each node to the traversal queue starting from the root node. The BFS for a graph is similar to that of a tree, with the exception that graphs may have cycles. In contrast to depth-first search, all neighbor nodes at a given depth are investigated before proceeding to the next level.

BFS Algorithm

The following are the steps involved in employing breadth-first search to explore a graph:

1. Take the data for the graph's adjacency matrix or adjacency list.
2. Create a queue and fill it with items.
3. Activate the root node (meaning that get the root node at the beginning of the queue).
4. Dequeue the queue's head (or initial element), then enqueue all of the queue's nearby nodes from left to right. Simply dequeue the head and resume the operation if a node has no nearby nodes that need to be investigated. (Note: If a neighbor emerges that has previously been investigated or is in the queue, don't enqueue it; instead, skip it.)
5. Continue in this manner until the queue is empty.

BFS Applications

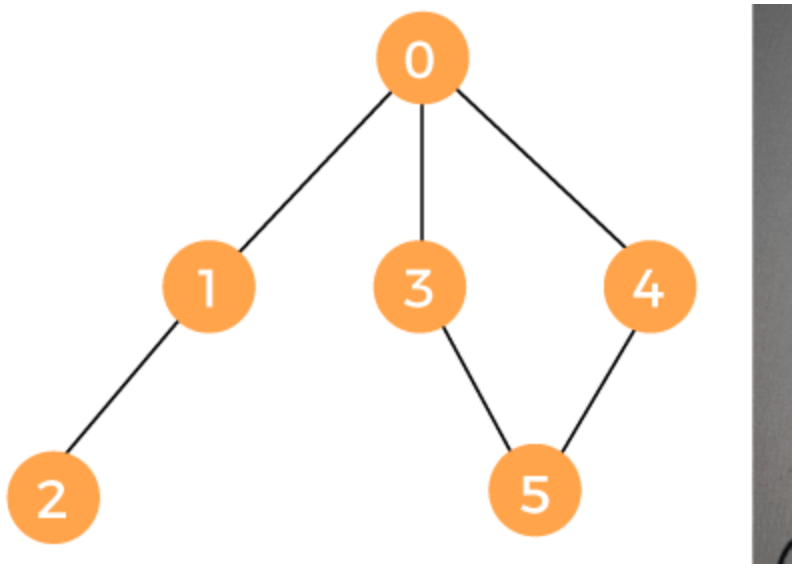
Because of the algorithm's flexibility, Breadth-First Search is quite useful in the real world. These are some of them:

1. In a peer-to-peer network, peer nodes are discovered. Most torrent clients, such as BitTorrent, uTorrent, and qBittorent, employ this process to find "seeds" and "peers" in the network.
2. The index is built using graph traversal techniques in web crawling. The procedure starts with the source page as the root node and works its way down to all secondary pages that are linked to the source page (and this process continues). Because of the reduced depth of the recursion tree, Breadth-First Search has an inherent advantage here.
3. The use of GPS navigation systems using the GPS, conduct a breadth-first search to locate nearby sites.

4. Cheney's technique, which employs the concept of breadth-first search, is used to collect garbage.

Example BFS Traversal

To get started, let's look at a simple example. We'll start with 0 as the root node and work our way down the graph.



Step 1: Enqueue(0)

Queue

0

Step 2: Dequeue(0), Enqueue(1), Enqueue(3), Enqueue(4)

Queue

1	3

Step 3: Dequeue(1), Enqueue(2)

Queue

3	4

Step 4: Dequeue(3), Enqueue(5). We won't add 1 to the queue again because 0 has already been explored.

Queue

4	2

Step 5: Dequeue(4)

Queue

2	5

Step 6: Dequeue(2)

Queue

5

Step 7: Dequeue(5)

Queue

The queue is empty now so we'll stop the process.

Breadth-First Search Java Program

There are several approaches of dealing with the code. We'll mostly discuss the steps involved in implementing a breadth first search in Java. An adjacency list or an adjacency matrix can be used to store graphs; either method is acceptable. The adjacency list will be used to represent our graph in our code. When implementing the Breadth-First Search algorithm in Java, it is much easier to deal with the adjacency list since we only have to travel through the list of nodes attached to each node once the node is dequeued from the head (or start) of the queue.

Complexity of Breadth-first search algorithm

The time complexity of the BFS algorithm is **O(V+E)**, where V is the number of vertices and E is the number of edges in the graph.

The space complexity of the BFS algorithm is O(V).