Nqueens problem

1) Start in the leftmost column

2) If all queens are placed

return true

3) Try all rows in the current column.

Do following for every tried row.

a) If the queen can be placed safely in this row

then mark this [row, column] as part of the

solution and recursively check if placing

queen here leads to a solution.

b) If placing the queen in [row, column] leads to

a solution then return true.

c) If placing queen doesn't lead to a solution then

unmark this [row, column] (Backtrack) and go to

step (a) to try other rows.

4) If all rows have been tried and nothing worked,

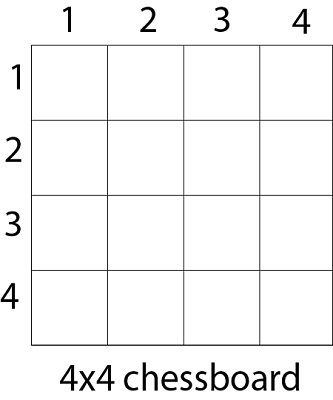
return false to trigger backtracking.

N-Queens Problem

N - Queens problem is to place n - queens in such a manner on an n x n chessboard that no queens attack each other by being in the same row, column or diagonal.

It can be seen that for n =1, the problem has a trivial solution, and no solution exists for n =2 and n =3. So first we will consider the 4 queens problem and then generate it to n - queens problem.

Given a 4 x 4 chessboard and number the rows and column of the chessboard 1 through 4.



Since, we have to place 4 queens such as q1 q2 q3 and q4 on the chessboard, such that no two queens attack each other. In such a conditional each queen must be placed on a different row, i.e., we put queen "i" on row "i."

34.9M

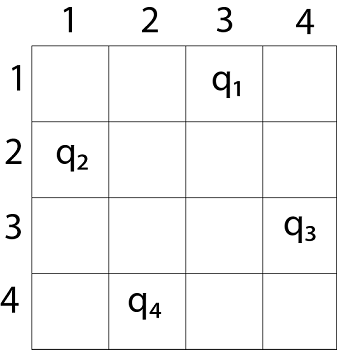
632

Prime Ministers of India | List of Prime Minister of India (1947-2020)

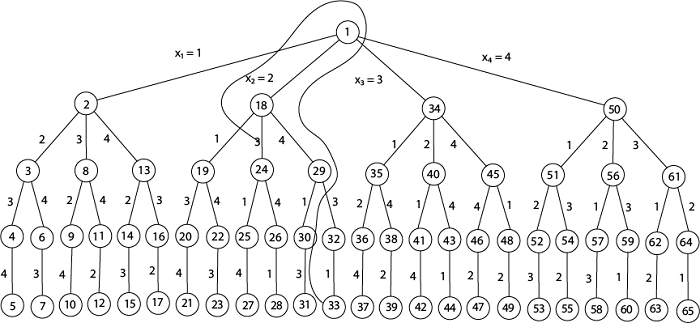
**Next**

**Stay**

Now, we place queen q1 in the very first acceptable position (1, 1). Next, we put queen q2 so that both these queens do not attack each other. We find that if we place q2 in column 1 and 2, then the dead end is encountered. Thus the first acceptable position for q2 in column 3, i.e. (2, 3) but then no position is left for placing queen 'q3' safely. So we backtrack one step and place the queen 'q2' in (2, 4), the next best possible solution. Then we obtain the position for placing 'q3' which is (3, 2). But later this position also leads to a dead end, and no place is found where 'q4' can be placed safely. Then we have to backtrack till 'q1' and place it to (1, 2) and then all other queens are placed safely by moving q2 to (2, 4), q3 to (3, 1) and q4 to (4, 3). That is, we get the solution (2, 4, 1, 3). This is one possible solution for the 4-queens problem. For another possible solution, the whole method is repeated for all partial solutions. The other solutions for 4 - queens problems is (3, 1, 4, 2) i.e.



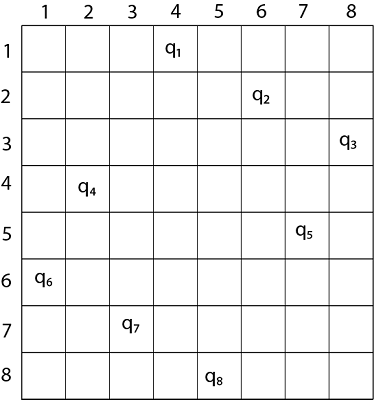
The implicit tree for 4 - queen problem for a solution (2, 4, 1, 3) is as follows:



**4 - Queens solution space with nodes numbered in DFS**

It can be seen that all the solutions to the 4 queens problem can be represented as 4 - tuples (x1, x2, x3, x4) where xi represents the column on which queen "qi" is placed.

One possible solution for 8 queens problem is shown in fig:



1. Thus, the solution **for** 8 -queen problem **for** (4, 6, 8, 2, 7, 1, 3, 5).
2. If two queens are placed at position (i, j) and (k, l).
3. Then they are on same diagonal only **if** (i - j) = k - l or i + j = k + l.
4. The first equation implies that j - l = i - k.
5. The second equation implies that j - l = k - i.
6. Therefore, two queens lie on the duplicate diagonal **if** and only **if** |j-l|=|i-k|

Place (k, i) returns a Boolean value that is true if the kth queen can be placed in column i. It tests both whether i is distinct from all previous costs x1, x2,....xk-1 and whether there is no other queen on the same diagonal.

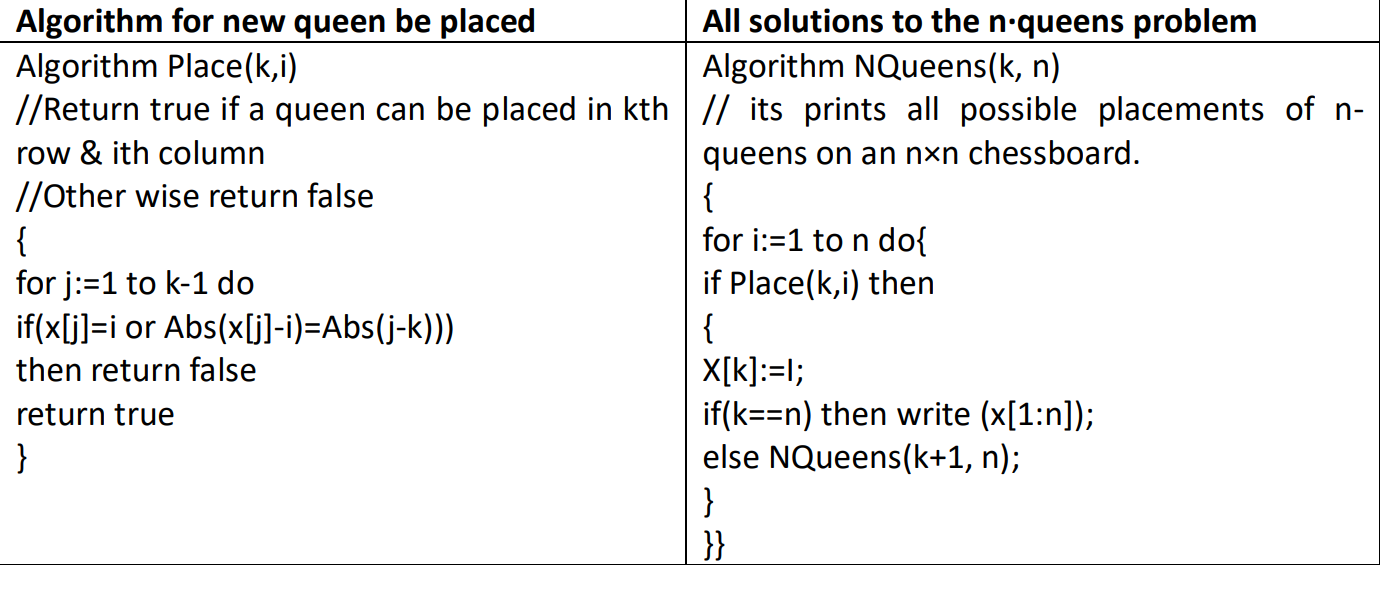
Using place, we give a precise solution to then n- queens problem.

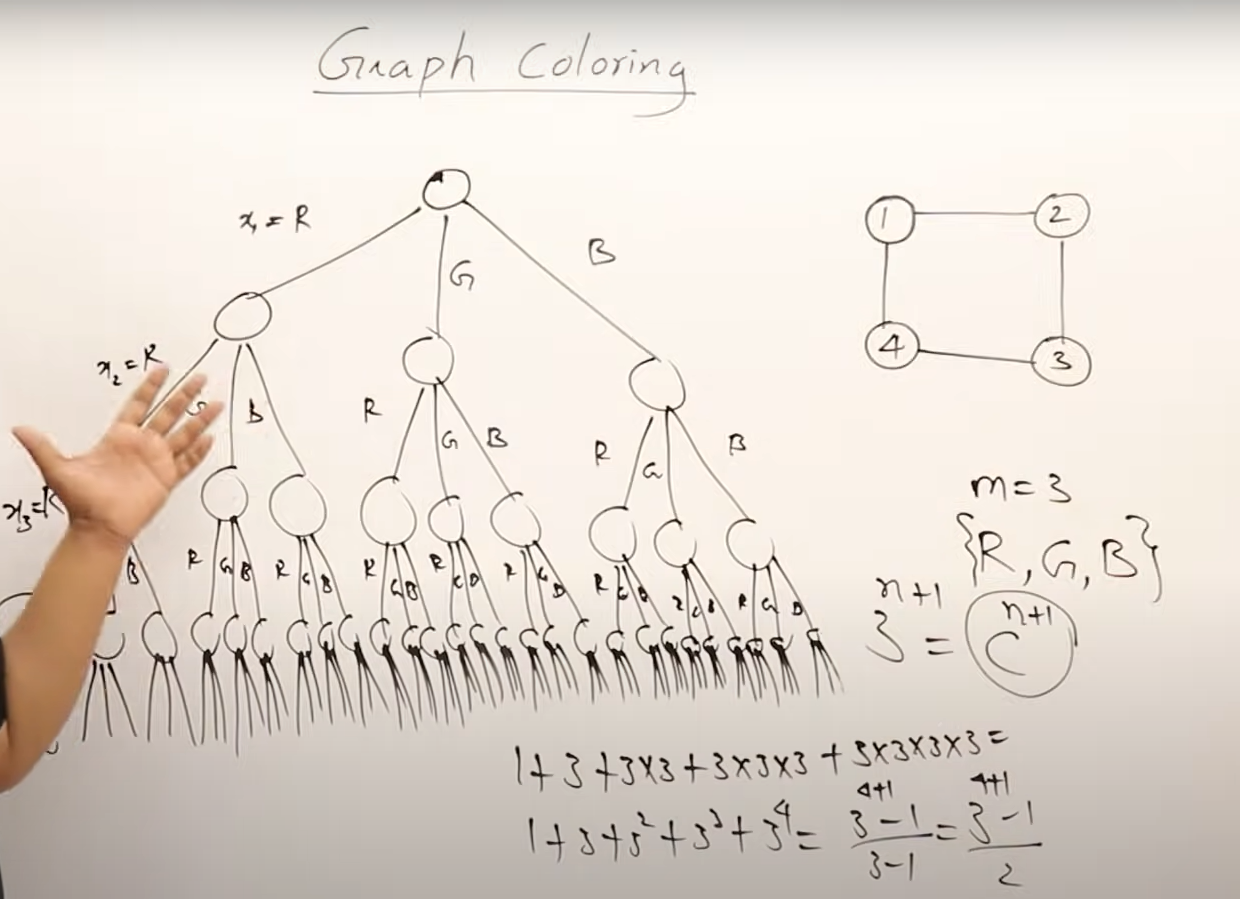
1. Place (k, i)
2. {
3. For j  ←  1 to k - 1
4. **do** **if** (x [j] = i)
5. or (Abs x [j]) - i) = (Abs (j - k))
6. then **return** **false**;
7. **return** **true**;
8. }

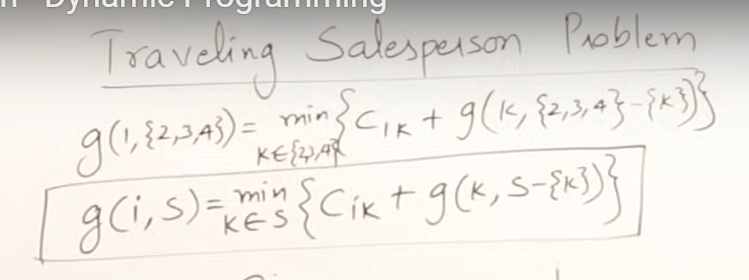
Place (k, i) return true if a queen can be placed in the kth row and ith column otherwise return is false.

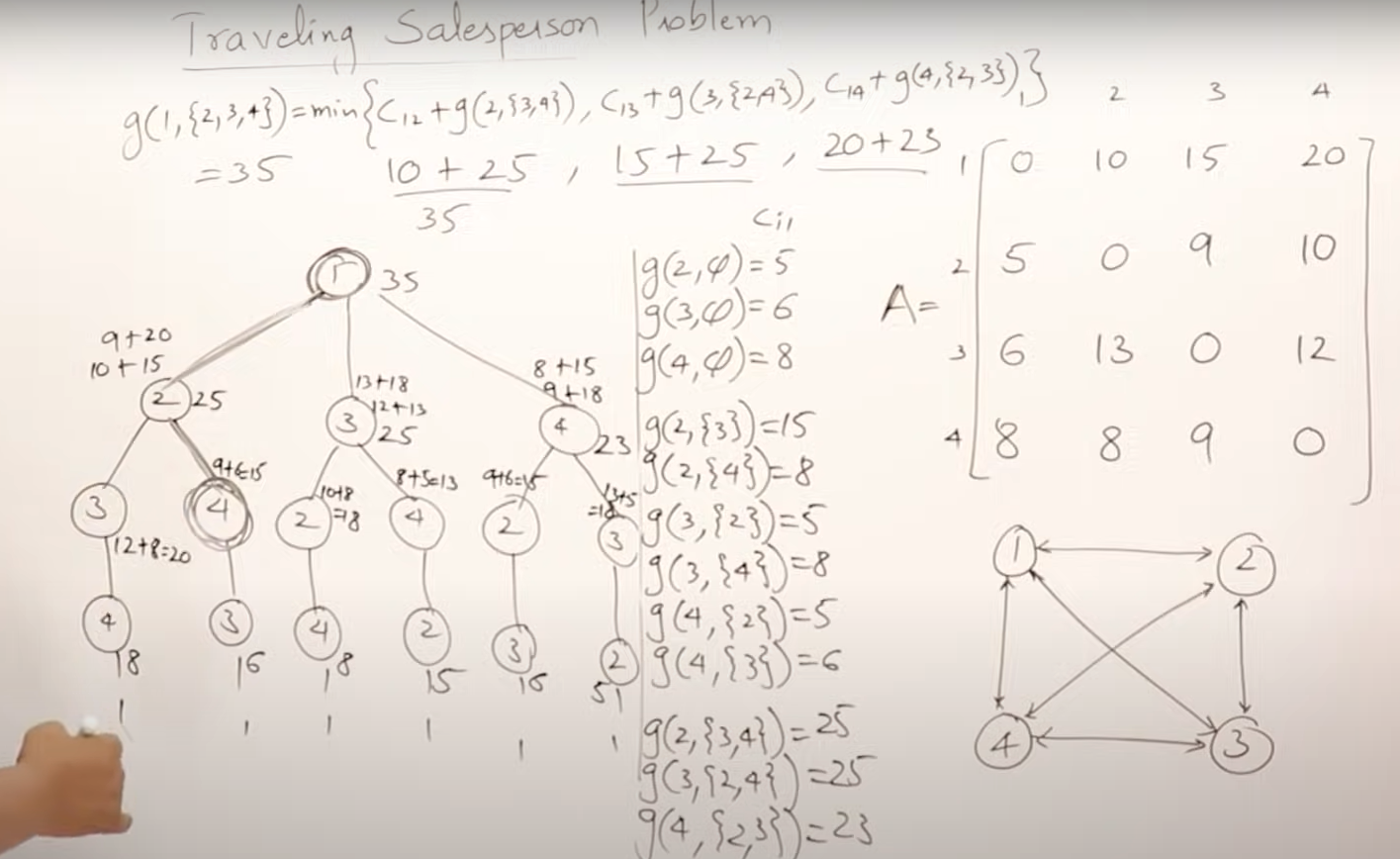
x [] is a global array whose final k - 1 values have been set. Abs (r) returns the absolute value of r.

1. N - Queens (k, n)
2. {
3. For i  ←  1 to n
4. **do** **if** Place (k, i) then
5. {
6. x [k]  ←  i;
7. **if** (k ==n) then
8. write (x [1....n));
9. **else**
10. N - Queens (k + 1, n);
11. }
12. }

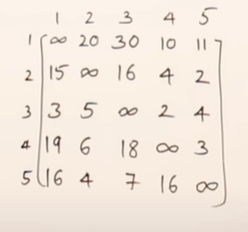








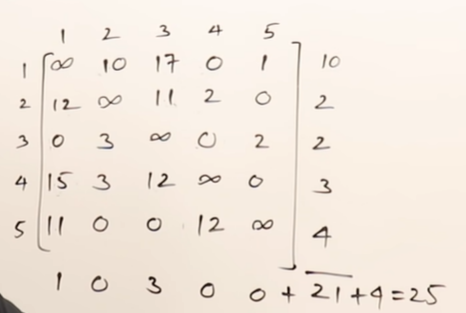
Travelling sales person using branch and bound



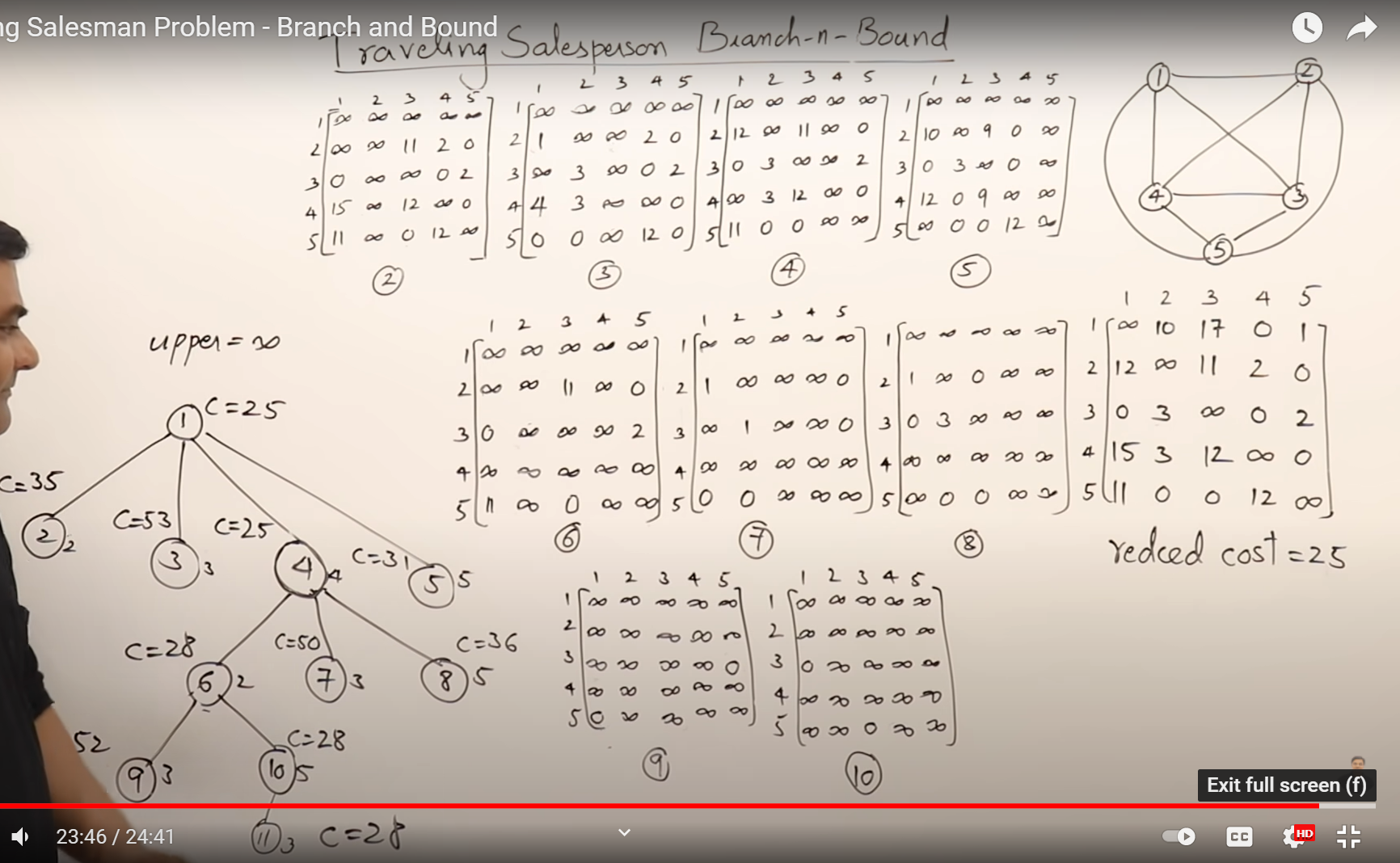
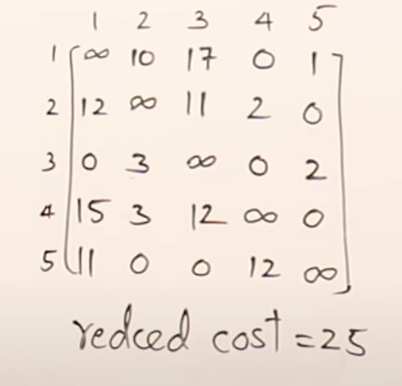
Reducing the given matrix

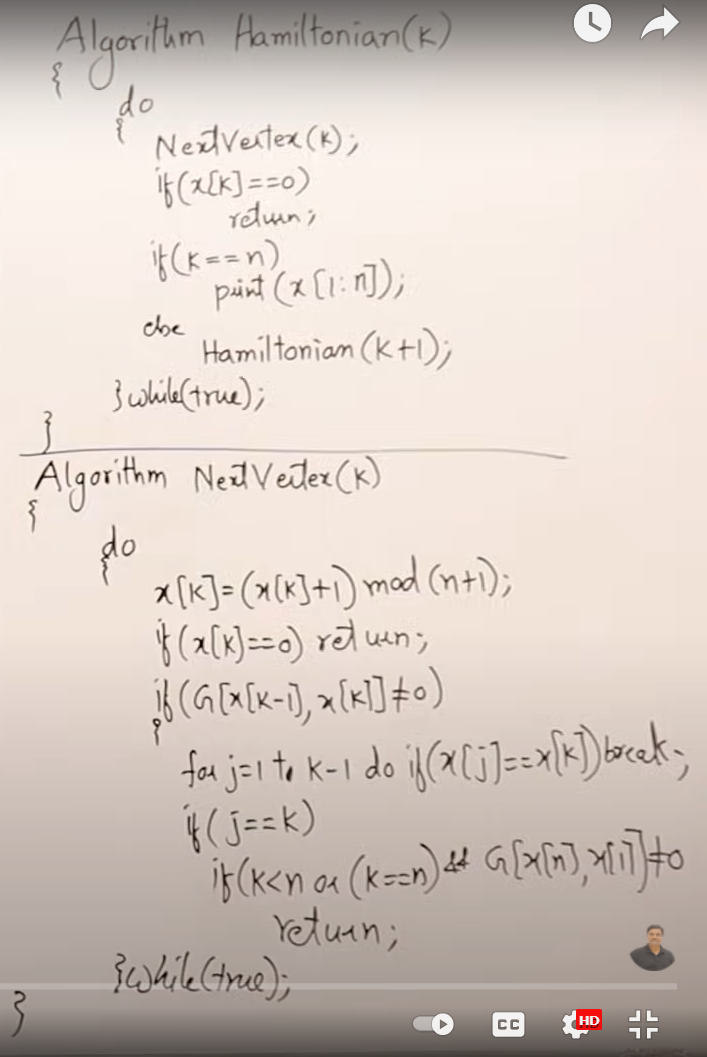
FIRSTLY SUBTRACT THE ROW THEN COLUMN

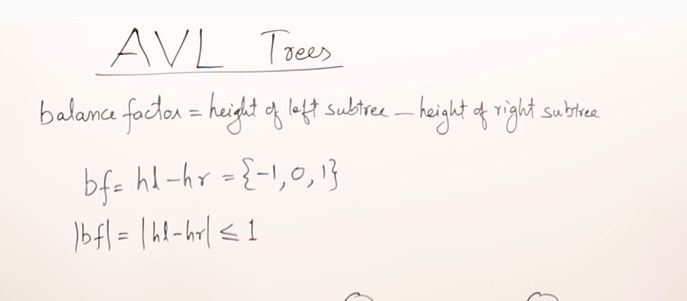
FIND MIN COST (BY TAKING MIN VALUE FROM ROW AND COLUMN)

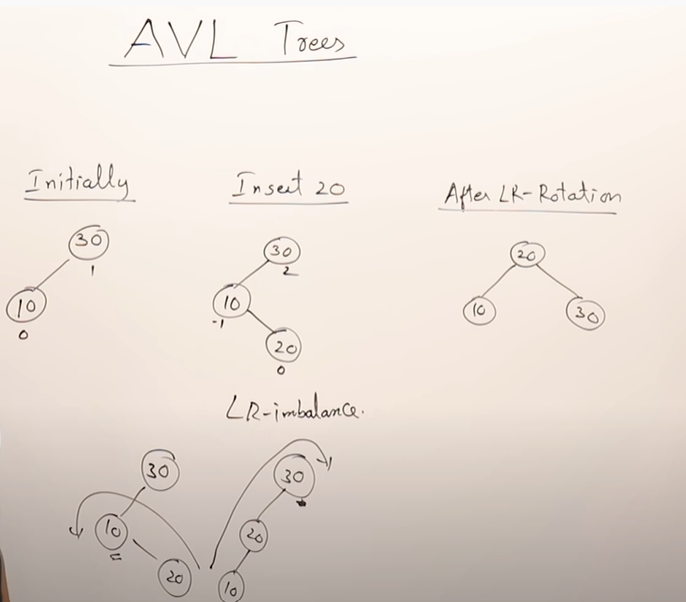
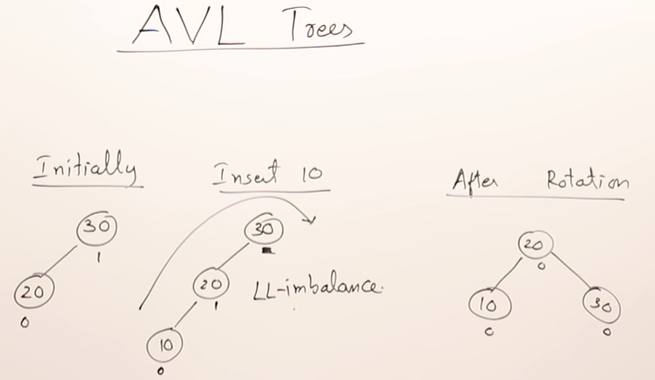


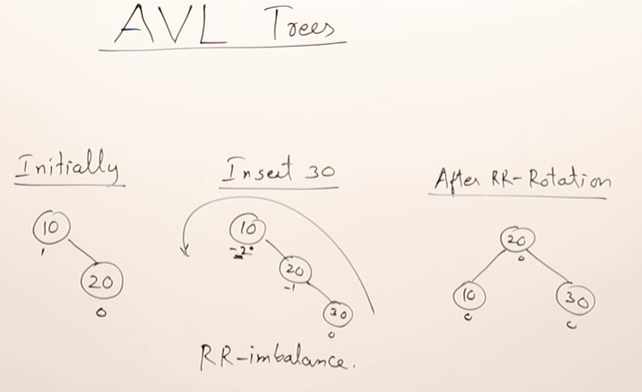
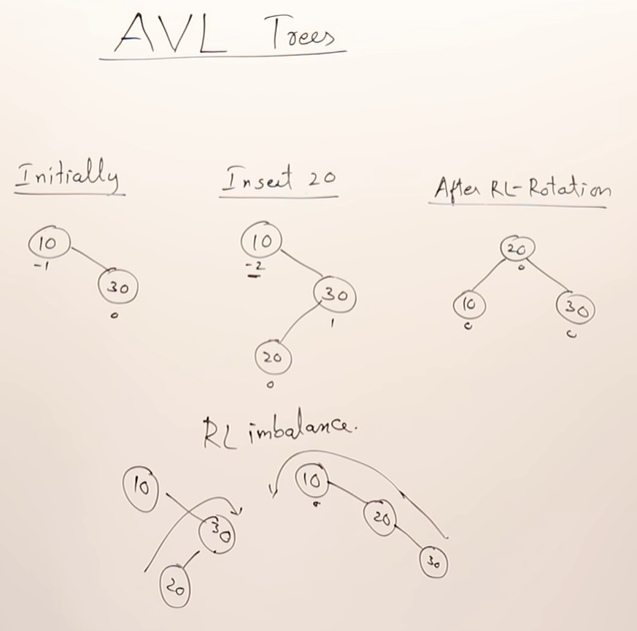
1)

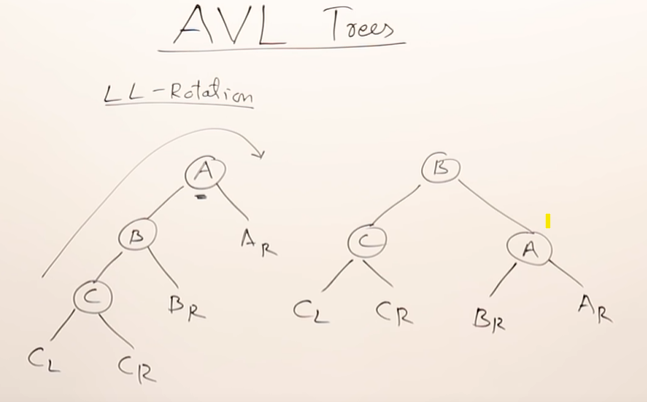


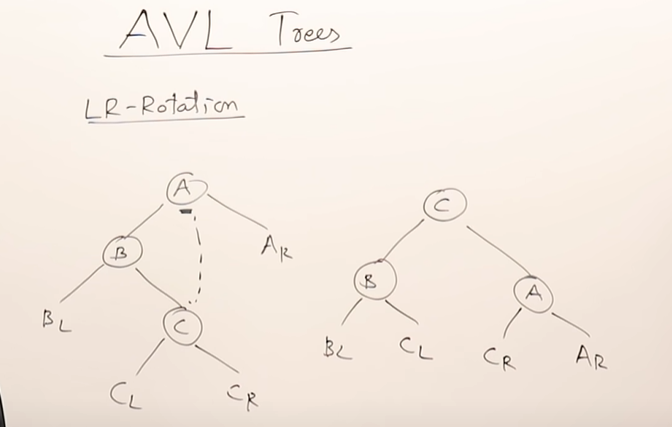












Algorithm for optimal merge pattern

**Algorithm: TREE (n)**

for i := 1 to n – 1 do

declare new node

node.leftchild := least (list)

node.rightchild := least (list)

node.weight) := ((node.leftchild).weight) + ((node.rightchild).weight)

insert (list, node);

return least (list);

algorithm for merge short

**Algorithm: Merge-Sort (numbers[], p, r)**

if p < r then

q = ⌊(p + r) / 2⌋

Merge-Sort (numbers[], p, q)

Merge-Sort (numbers[], q + 1, r)

Merge (numbers[], p, q, r)

**Function: Merge (numbers[], p, q, r)**

n1 = q – p + 1

n2 = r – q

declare leftnums[1…n1 + 1] and rightnums[1…n2 + 1] temporary arrays

for i = 1 to n1

leftnums[i] = numbers[p + i - 1]

for j = 1 to n2

rightnums[j] = numbers[q+ j]

leftnums[n1 + 1] = ∞

rightnums[n2 + 1] = ∞

i = 1

j = 1

for k = p to r

if leftnums[i] ≤ rightnums[j]

numbers[k] = leftnums[i]

i = i + 1

else

numbers[k] = rightnums[j]

j = j + 1

algorithm for travelling sales person

**Algorithm: Traveling-Salesman-Problem**

C ({1}, 1) = 0

for s = 2 to n do

for all subsets S Є {1, 2, 3, … , n} of size s and containing 1

C (S, 1) = ∞

for all j Є S and j ≠ 1

C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}

Return minj C ({1, 2, 3, …, n}, j) + d(j, i)

Algorithm for heap sort

**Algorithm: Max-Heapify(numbers[], i)**

leftchild := numbers[2i]

rightchild := numbers [2i + 1]

if leftchild ≤ numbers[].size and numbers[leftchild] > numbers[i]

largest := leftchild

else

largest := i

if rightchild ≤ numbers[].size and numbers[rightchild] > numbers[largest]

largest := rightchild

if largest ≠ i

swap numbers[i] with numbers[largest]

Max-Heapify(numbers, largest)

**Algorithm: Build-Max-Heap(numbers[])**

numbers[].size := numbers[].length

fori = ⌊ numbers[].length/2 ⌋ to 1 by -1

Max-Heapify (numbers[], i)

Dijkstras algorithm

**Algorithm: Dijkstra’s-Algorithm (G, w, s)**

for each vertex v Є G.V

v.d := ∞

v.∏ := NIL

s.d := 0

S := Ф

Q := G.V

while Q ≠ Ф

u := Extract-Min (Q)

S := S U {u}

for each vertex v Є G.adj[u]

if v.d > u.d + w(u, v)

v.d := u.d + w(u, v)

v.∏ := u =

### binary search tree

### Pseudocode for Inserting a Node in BST:

insert (element, root)

Node x = root

Node y = NULL

while x:

y = x

if x.value < element.value

x = x.right

else

x = x.left

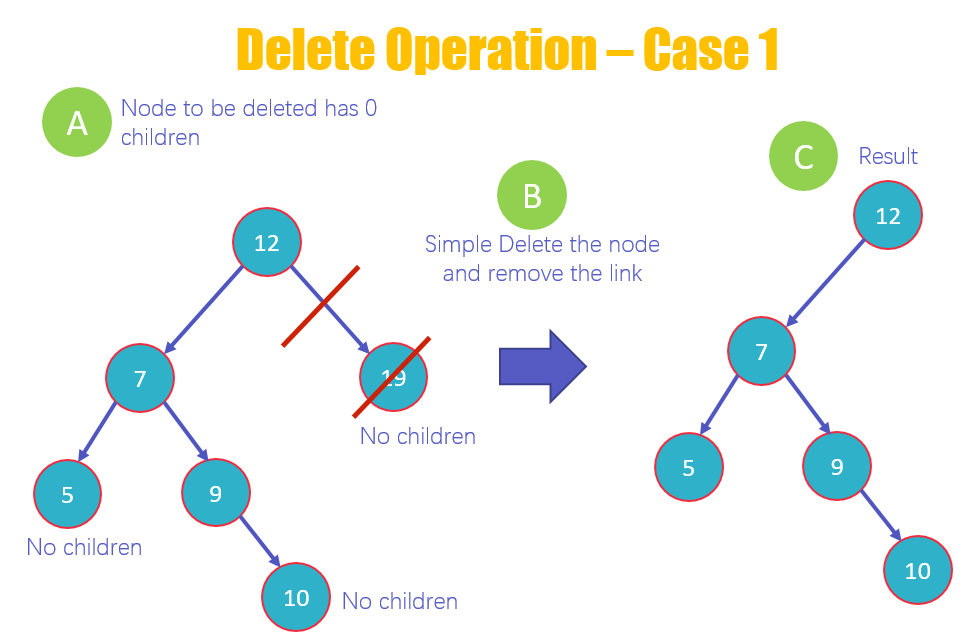
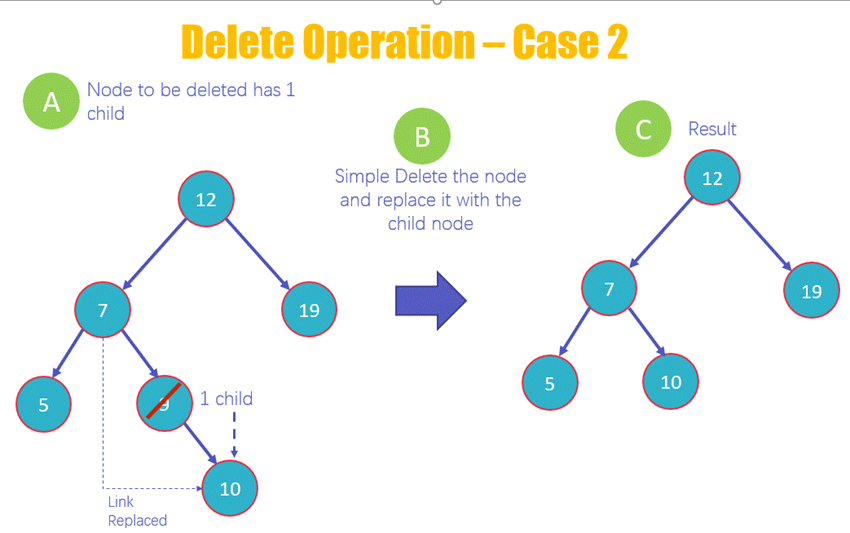
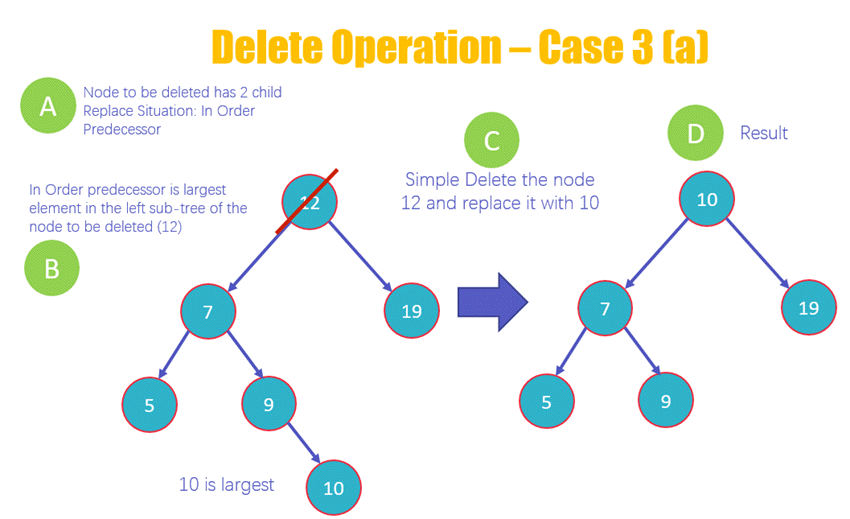
if y.value < element

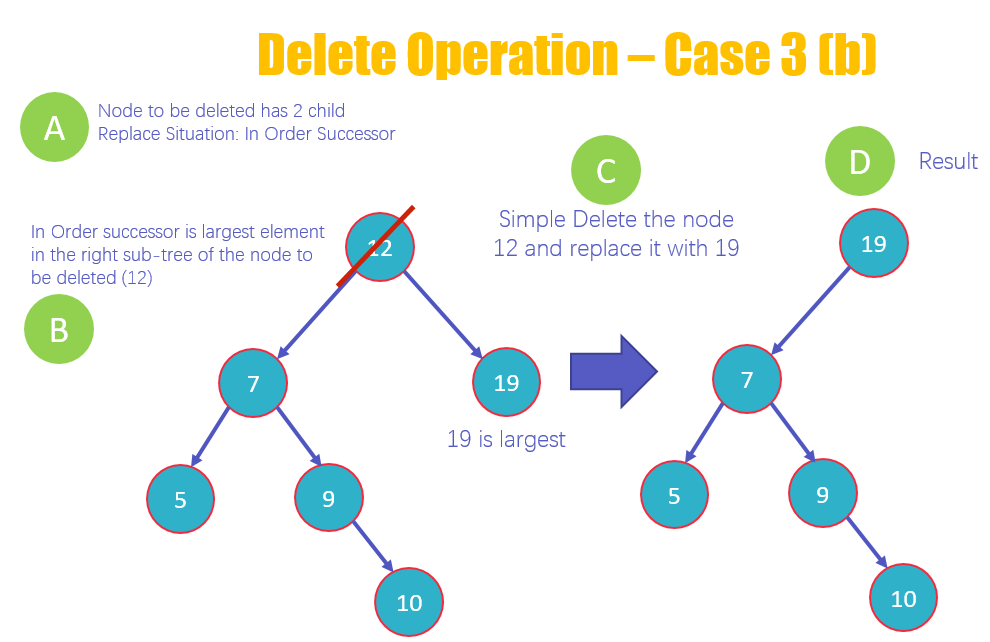
y.right = element

else

y.left = element

### Pseudo Code for Deleting a Node:

* Case 1- Node with zero children: this is the easiest situation, you just need to delete the node which has no further children on the right or left.
* 
* This is the first case of deletion in which you delete a node that has no children. As you can see in the diagram that 19, 10 and 5 have no children. But we will delete 19.
* Delete the value 19 and remove the link from the node.
* View the new structure of the BST without 19
* Case 2 – Node with one child: once you delete the node, simply connect its child node with the parent node of the deleted value. 
* This is the second case of deletion in which you delete a node that has 1 child, as you can see in the diagram that 9 has one child.
* Delete the node 9 and replace it with its child 10 and add a link from 7 to 10
* View the new structure of the BST without 9
* Case 3 Node with two children: this is the most difficult situation, and it works on the following two rules
* 3a – In Order Predecessor: you need to delete the node with two children and replace it with the largest value on the left-subtree of the deleted node
* 
* Here you will be deleting the node 12 that has two children
* The deletion of the node will occur based upon the in order predecessor rule, which means that the largest element on the left subtree of 12 will replace it.
* Delete the node 12 and replace it with 10 as it is the largest value on the left subtree
* View the new structure of the BST after deleting 12
* 3b – In Order Successor: you need to delete the node with two children and replace it with the largest value on the right-subtree of the deleted node



1. 1 Delete a node 12 that has two children
2. 2 The deletion of the node will occur based upon the In Order Successor rule, which means that the largest element on the right subtree of 12 will replace it
3. 3 Delete the node 12 and replace it with 19 as it is the largest value on the right subtree
4. 4 View the new structure of the BST after deleting 12

PSEUDOCODE FOR DELETION

delete (value, root):

Node x = root

Node y = NULL

# searching the node

while x:

y = x

if x.value < value

x = x.right

else if x.value > value

x = x.left

else if value == x

break

# if the node is not null, then replace it with successor

if y.left or y.right:

newNode = GetInOrderSuccessor(y)

root.value = newNode.value

# After copying the value of successor to the root #we're deleting the successor

free(newNode)

else

free(y)

# Tree traversal (Inorder, Preorder an Postorder)

.

The term 'tree traversal' means traversing or visiting each node of a tree. There is a single way to traverse the linear data structure such as linked list, queue, and stack. Whereas, there are multiple ways to traverse a tree that are listed as follows -

* Preorder traversal
* Inorder traversal
* Postorder traversal

### **Preorder traversal**

This technique follows the 'root left right' policy. It means that, first root node is visited after that the left subtree is traversed recursively, and finally, right subtree is recursively traversed. As the root node is traversed before (or pre) the left and right subtree, it is called preorder traversal.

The applications of preorder traversal include -

* It is used to create a copy of the tree.
* It can also be used to get the prefix expression of an expression tree.
* Step 1 - Visit the root node
* Step 2 - Traverse the left subtree recursively.
* Step 3 - Traverse the right subtree recursively.



### **Postorder traversal**

This technique follows the 'left-right root' policy. It means that the first left subtree of the root node is traversed, after that recursively traverses the right subtree, and finally, the root node is traversed

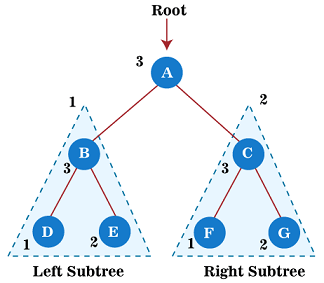
As the root node is traversed after (or post) the left and right subtree, it is called postorder traversal.

So, in a postorder traversal, each node is visited after both of its subtrees.

The applications of postorder traversal include -

* It is used to delete the tree.
* It can also be used to get the postfix expression of an expression tree.
* Step 1 - Traverse the left subtree recursively.
* Step 2 - Traverse the right subtree recursively.
* Step 3 - Visit the root node.

**Example**



### **Inorder traversal**

This technique follows the 'left root right' policy. It means that first left subtree is visited after that root node is traversed, and finally, the right subtree is traversed. As the root node is traversed between the left and right subtree, it is named inorder traversal.

So, in the inorder traversal, each node is visited in between of its subtrees.

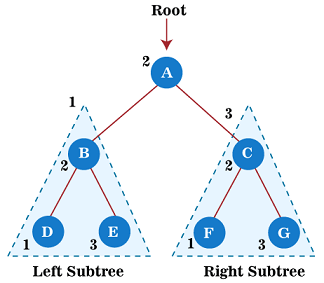
The applications of Inorder traversal includes -

* It is used to get the BST nodes in increasing order.
* It can also be used to get the prefix expression of an expression tree.

**Algorithm**

1. Until all nodes of the tree are not visited
3. Step 1 - Traverse the left subtree recursively.
4. Step 2 - Visit the root node.
5. Step 3 - Traverse the right subtree recursively.

**Example**



# Difference between BFS and DFS

**Breadth First Search:**  
**BFS**stands for [**Breadth First Search**](https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/) is a vertex-based technique for finding the shortest path in the graph. It uses a [Queue data structure](https://www.geeksforgeeks.org/queue-data-structure/) that follows first in first out. In BFS, one vertex is selected at a time when it is visited and marked then its adjacent are visited and stored in the queue. It is slower than DFS.   
**Example**:

**Input:**

A

/ \

B C

/ / \

D E F

**Output:**

A, B, C, D, E, F

**Depth First Search:**  
**DFS**stands for [**Depth First Search**](https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/) is an edge-based technique. It uses the [Stack data structure](https://www.geeksforgeeks.org/stack-data-structure/) and performs two stages, first visited vertices are pushed into the stack, and second if there are no vertices then visited vertices are popped.   
**Example:**

**Input:**

A

/ \

B C

/ / \

D E F

**Output:**

A, B, D, C, E, F

**BFS vs DFS**

|  |  |  |
| --- | --- | --- |
| S.No | BFS | DFS |
| 1. | BFS stands for Breadth First Search. | DFS stands for Depth First Search. |
| 2. | BFS(Breadth First Search) uses Queue data structure for finding the shortest path. | DFS(Depth First Search) uses Stack data structure. |
| 3. | BFS can be used to find single source shortest path in an unweighted graph, because in BFS, we reach a vertex with minimum number of edges from a source vertex. | In DFS, we might traverse through more edges to reach a destination vertex from a source. |
| 4. | BFS is more suitable for searching vertices which are closer to the given source. | DFS is more suitable when there are solutions away from source. |
| 5. | BFS considers all neighbors first and therefore not suitable for decision making trees used in games or puzzles. | DFS is more suitable for game or puzzle problems. We make a decision, then explore all paths through this decision. And if this decision leads to win situation, we stop. |
| 6. | The Time complexity of BFS is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges. | The Time complexity of DFS is also O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges. |
| 7. | Here, siblings are visited before the children | Here, children are visited before the siblings |
| 8. | In BFS there is no concept of backtracking. | DFS algorithm is a recursive algorithm that uses the idea of backtracking |
| 9. | BFS is used in various application such as  bipartite graph, and shortest path etc. | DFS is used in various application such as acyclic graph and topological order etc. |
| 10. | BFS requires more memory. | DFS requires less memory. |

**Types of Complexity Classes**

This article discusses the following complexity classes:

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

**P Class**

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

**Features:**

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

This class contains many natural problems like:

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

**NP Class**

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

**Features:**

1. The solutions of the NP class are hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify.
2. Problems of NP can be verified by a Turing machine in polynomial time. This class contains many problems that one would like to be able to solve effectively:
3. **Boolean Satisfiability Problem (SAT).**
4. **Hamiltonian Path Problem.**
5. **Graph coloring.**

**Co-NP Class**

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

**Features:**

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

Some example problems for C0-NP are:

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

**NP-hard class**

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

**Features:**

1. All NP-hard problems are not in NP.
2. It takes a long time to check them. This means if a solution for an NP-hard problem is given then it takes a long time to check whether it is right or not.
3. A problem A is in NP-hard if, for every problem L in NP, there exists a polynomial-time reduction from L to A.

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

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

**NP-complete class**

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

**Features:**

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

Some example problems include:

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

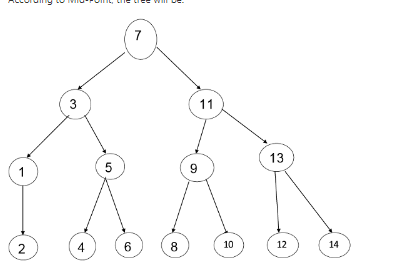
|  |  |
| --- | --- |
| **Complexity Class** | **Characteristic feature** |
| **P** | Easily solvable in polynomial time. |
| **NP** | Yes, answers can be checked in polynomial time. |
| **Co-NP** | No, answers can be checked in polynomial time. |
| **NP-hard** | All NP-hard problems are not in NP and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

**The lower bound theory**

**The lower bound theory** is the technique that has been used to establish the given algorithm in the most efficient way which is possible. This is done by discovering a function **g(n)** that is a lower bound on the time that any algorithm must take to solve the given problem

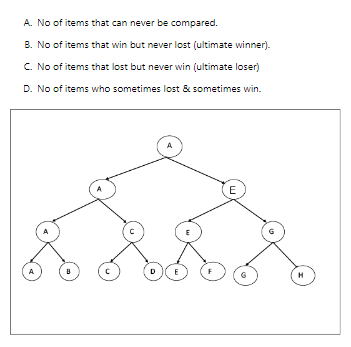
**The proofing techniques that are useful for obtaining lower bounds are:**

1. **Comparison trees:**  
   Comparison trees are the computational model useful for determining lower bounds for sorting and searching problems.



1. **Oracles and adversary arguments:**  
   One of the techniques that are important for obtaining lower bounds consists of making the use of an oracle
2. **Lower bounds through reduction:**  
   This is a very important technique of lower bound, This technique calls for reducing the given problem for which a lower bound is already known.
3. **Techniques for the algebraic problem:**  
   Substitution and linear independence are two methods used for deriving lower bounds on algebraic and arithmetic problems. The algebraic problems are operation on integers, polynomials, and rational functions.
4. State Space Method:

. State Space Method is a set of rules that show the possible states (n-tuples) that an algorithm can assume from a given state of a single comparison.



**parallel algorithm**

. A **parallel algorithm** is an algorithm that can execute several instructions simultaneously on different processing devices and then combine all the individual outputs to produce the final result.

* **Parallel Algorithm** − The problem is divided into sub-problems and are executed in parallel to get individual outputs. Later on, these individual outputs are combined together to get the final desired output.

It is not easy to divide a large problem into **sub-problems**. Sub-problems may have data dependency among them. Therefore, the processors have to communicate with each other to solve the problem.

It has been found that the time needed by the processors in communicating with each other is more than the actual processing time. So, while designing a parallel algorithm, proper CPU utilization should be considered to get an efficient algorithm.