INSTITUTE OF AERONAUTICAL ENGINEERING
(Autonomous)
Dundigal - 500 043, Hyderabad, Telangana

COURSE CONTENT

DESIGN AND ANALYSIS OF ALGORITHMS LABORATORY

IV Semester: CSE / IT / CSE (CS)								
Course Code	**Category**	**Hours / Week**			**Credits**	**Maximum Marks**		
ACSD16	**Core**	**L**	**T**	**P**	**C**	**CIA**	**SEE**	**Total**
		-	-	2	1	40	60	100
Contact Classes: Nil	**Tutorial Classes: Nil**	**Practical Classes: 45**				**Total Classes: 45**		
Prerequisites: There are no prerequisites to take this course.								

I. COURSE OVERVIEW:

Design and analysis of algorithm lab provides hands on experience in implementing different algorithmic paradigms and develops competence in choosing appropriate data structure to improve efficiency of technique used. This laboratory implements sorting techniques using divide and conquer strategy, shortest distance algorithms based on Greedy, Dynamic programming techniques, Minimum spanning tree construction and applications of Backtracking, Branch and Bound. This is essential for developing software in areas Information storage and retrieval, Transportation through networks, Graph theory and Optimization problems.

II. COURSES OBJECTIVES:

The students will try to learn

I. The selection of Algorithmic technique and Data structures required for efficient development of technical and engineering applications.
II. The algorithmic design paradigms and methods for identifying solutions of optimization problems.
III. Implementation of different algorithms for the similar problems to compare their performance.

III. COURSE OUTCOMES:

At the end of the course students should be able to:

CO 1	Apply Divide and conquer strategy to organize the data in ascending or descending order
CO 2	Make use of Algorithmic Design paradigms to determine shortest distance and transitive closure of Directed or Undirected Graphs
CO 3	Utilize Greedy Technique for generating minimum cost spanning tree of a Graph.
CO 4	Compare the efficiencies of traversal problems using different Tree and Graph traversal algorithms.
CO 5	Utilize Backtracking method for solving Puzzles involving building solutions incrementally.
CO 6	Examine Branch and Bound Approach for solving Combinatorial optimization problems.

IV. COURSE CONTENT:

EXERCISES FOR DEDIGN AND ANALYSIS OF ALGORITHMS LABORATORY

Note: Students are encouraged to bring their own laptops for laboratory practice sessions.

1. Getting Started Exercises

1.1 Implicit Recursion

A specific type of recursion called **implicit recursion** occurs when a function calls itself without making an explicit recursive call. This can occur when a function calls another function, which then calls the original code once again and starts a recursive execution of the original function.

Using implicit recursion find the second-largest elements from the array.

In this case, the **find second largest** method calls the **find_largest()** function via implicit recursion to locate the second-largest number in a provided list of numbers. Implicit recursion can be used in this way to get the second-largest integer without having to write any more code

Input: nums = [1, 2, 3, 4, 5]

Output: 4

```
class IR {

  public static int findLargest(List<Integer> numbers)
  {
     // Write code here
     …
  }

  public static int
    findSecondLargest(List<Integer> numbers)
  {
     // Write code here
     …
  }

  public static void main(String[] args)
  {
    List<Integer> numbers
      = Arrays.asList(1, 2, 3, 4, 5);

    // Function call
    int secondLargest = findSecondLargest(numbers);
    System.out.println(secondLargest);
  }
}
```

1.2 Towers of Hanoi

Tower of Hanoi is a mathematical puzzle where we have three rods (A, B, and C) and N disks. Initially, all the disks are stacked in decreasing value of diameter i.e., the smallest disk is placed on the top and they are on rod A. The objective of the puzzle is to move the entire stack to another rod (here considered C), obeying the following simple rules:

- Only one disk can be moved at a time.
- Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
- No disk may be placed on top of a smaller disk.

Input: 2

Output: Disk 1 moved from A to B

Disk 2 moved from A to C

Disk 1 moved from B to C

Input: 3

Output: Disk 1 moved from A to C

Disk 2 moved from A to B

Disk 1 moved from C to B

Disk 3 moved from A to C

Disk 1 moved from B to A

Disk 2 moved from B to C

Disk 1 moved from A to C

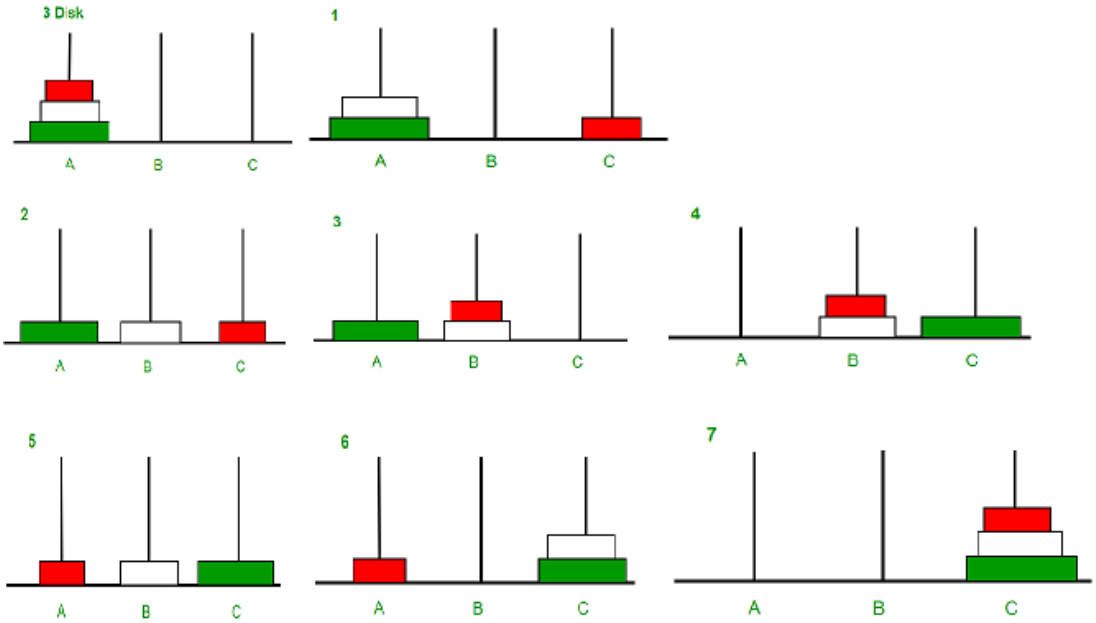
Tower of Hanoi using Recursion:

The idea is to use the helper node to reach the destination using recursion. Below is the pattern for this problem:

- Shift 'N-1' disks from 'A' to 'B', using C.
- Shift last disk from 'A' to 'C'.
- Shift 'N-1' disks from 'B' to 'C', using A.

Follow the steps below to solve the problem:

- Create a function towerOfHanoi where pass the N (current number of disk), from_rod, to_rod, aux_rod.
- Make a function call for N – 1 th disk.
- Then print the current the disk along with from_rod and to_rod
- Again make a function call for N – 1 th disk.




```
// Recursive java function to solve Tower of Hanoi
class TOH {
    static void towerOfHanoi(int n, char from_rod,
                             char to_rod, char aux_rod)
    {
        if (n == 0) {
            return;
        }
        // Write code here
        …
    }

    //Driver code

    public static void main(String args[])
    {
        int N = 3;

        // A, B and C are names of rods
        towerOfHanoi(N, 'A', 'C', 'B');
    }
}
```

1.3 Recursively Remove all Adjacent Duplicates

Given a string, recursively remove adjacent duplicate characters from the string. The output string should not have any adjacent duplicates.

Input: s = "azxxzy"

Output: "ay"

Explanation:

- First "azxxzy" is reduced to "azzy".

- The string “azzy” contains duplicates
- So it is further reduced to “ay”

Input: “caaabbbaacdddd”

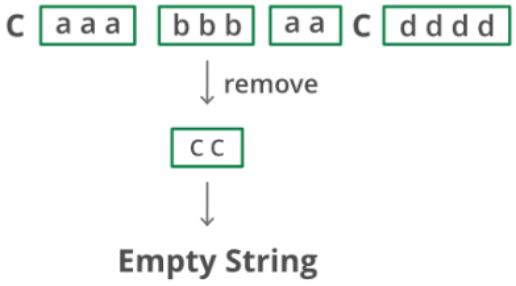
Output: Empty String

Input: “acaaabbbacdddd”

Output: “acac”

Procedure to remove duplicates:

- Start from the leftmost character and remove duplicates at left corner if there are any.
- The first character must be different from its adjacent now. Recur for string of length n-1 (string without first character).
- Let the string obtained after reducing right substring of length n-1 be rem_str. There are three possible cases
  - If first character of rem_str matches with the first character of original string, remove the first character from rem_str.
  - If remaining string becomes empty and last removed character is same as first character of original string. Return empty string.
  - Else, append the first character of the original string at the beginning of rem_str.
- Return rem_str.




```
// Java program to remove all adjacent duplicates from a string

class RD {

    static char last_removed; //will store the last char
                              // removed during recursion

    // Recursively removes adjacent duplicates from str and
    // returns new string. last_removed is a pointer to
    // last_removed character
    static String removeUtil(String str)
    {

        // If length of string is 1 or 0
        if (str.length() == 0 || str.length() == 1)
            return str;

        // Write code here
        …
```

```
    }

    static String remove(String str)
    {
        last_removed = '\0';
        return removeUtil(str);
    }

    // Driver code
    public static void main(String args[])
    {

        String str1 = "azxxxzy";
        System.out.println(remove(str1));

        String str2 = "caaabbbaac";
        System.out.println(remove(str2));

        String str3 = " gghhg ";
        System.out.println(remove(str3));

       String str4 = " aaaacddddcappp ";
        System.out.println(remove(str4));

       String str5 = " aaaaaaaaaa ";
        System.out.println(remove(str5));



    }
}
```

1.4 Randomized Algorithm - Shuffle a deck of cards

An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm

Given a deck of cards, the task is to shuffle them

Output:

29 27 20 23 26 21 35 51 15 18 46 32 33 19

24 30 3 45 40 34 16 11 36 50 17 10 7 5 4

39 6 47 38 28 13 44 49 1 8 42 43 48 0 12

37 41 25 2 31 14 22

Output will be different each time because of the random function used in the program

Procedure

1. First, fill the array with the values in order.
2. Go through the array and exchange each element with the randomly chosen element in the range

from itself to the end.
// It is possible that an element will be swap
// with itself, but there is no problem with that. If any of them become zero, return 0

```
import java.util.Random;

class SDC {

    // Function which shuffle and print the array
    public static void shuffle(int card[], int n)
    {

        Random rand = new Random();

        // Write code here
        …

    // Driver code
    public static void main(String[] args)
    {
        // Array from 0 to 51
        int a[] = {0, 1, 2, 3, 4, 5, 6, 7, 8,
                   9, 10, 11, 12, 13, 14, 15,
                   16, 17, 18, 19, 20, 21, 22,
                   23, 24, 25, 26, 27, 28, 29,
                   30, 31, 32, 33, 34, 35, 36,
                   37, 38, 39, 40, 41, 42, 43,
                   44, 45, 46, 47, 48, 49, 50,
                   51};

        shuffle(a, 52);

        // Printing all shuffled elements of cards
        for (int i = 0; i < 52; i ++)
           System.out.print(a[i]+" ");

    }
}
```

1.5 Strong Password Suggester using Randomized algorithm

Given a password entered by the user, check its strength and suggest some password if it is not strong.

Criteria for strong password is as follows :
A password is strong if it has :

1. At least 8 characters
2. At least one special char
3. At least one number
4. At least one upper and one lower case char.

Input : keshav123

Output : Suggested Password

k(eshav12G3

keshav1@A23

keGshav1$23

kesXhav@123

keAshav12$3

kesKhav@123

kes$hav12N3

$kesRhav123

Input: rakesh@1995kumar

Output: Your Password is Strong

Procedure:

Check the input password for its strength if it fulfills all the criteria, then it is strong password else check for the absent characters in it, and return the randomly generated password to the user.

```
// Java code to suggest strong password

public class SSP {

    // adding more characters to suggest strong password
    static StringBuilder add_more_char(
                        StringBuilder str, int need)
    {
        // Write code here
       …
    }

    // make powerful String
    static StringBuilder suggester(int l, int u, int d,
                                int s, StringBuilder str)
    {
        // Write code here
       …
    }

    /* make_password function :This function is used
    to check strongness and if input String is not
```

```
    strong, it will suggest*/
    static void generate_password(int n, StringBuilder p)
    {

        // Write code here
        …
    }

    // Driver code
    public static void main(String[] args)
    {
        StringBuilder input_String = new StringBuilder("iare@2024");
        generate_password(input_String.length(), input_String);
    }
}
```

1.6 Reservoir Sampling using Randomized Algorithm

Reservoir Sampling is a family of randomized algorithms for randomly choosing *k* samples from a list of *n* items, where *n* is either a very large or unknown number. Typically, *n* is large enough that the list doesn't fit into main memory.
Example: A list of search queries in Google and Facebook.
So we are given a big array (or stream) of numbers (to simplify), and we need to write an efficient function to randomly select *k* numbers where *1 <= k <= n*.
Let the input array be stream [].
A **simple solution** is to create an array reservoir [] of maximum size *k*. One by one randomly select an item from stream [0..n-1]. If the selected item is not previously selected, then put it in reservoir []. To check if an item is previously selected or not, we need to search the item in reservoir []. The time complexity of this algorithm will be *O(k^2)*. This can be costly if *k* is big. Also, this is not efficient if the input is in the form of a stream.
It **can be solved in O(n) time**. The solution also suits well for input in the form of stream.
Steps to get best solution:
1) Create an array reservoir r[0..k-1] and copy first *k* items of stream[] to it.
2) Now one by one consider all items from *(k+1)* th item to *n*th item
a) Generate a random number from 0 to *i* where *i* is the index of the current item in stream[]. Let the generated random number is *j*.
b**)** If *j* is in range 0 to *k-1*, *replace* reservoir[j] *with stream*[i]

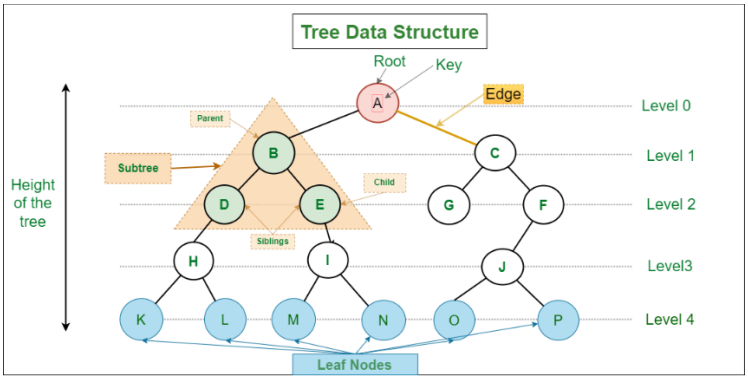
```
// An efficient Java program to randomly
// select k items from a stream of items
public class ReservoirSampling {

    // A function to randomly select k items from
    // stream[0..n-1].
    static void selectKItems(int stream[], int n, int k)
    {
        // Write code here
```

```
        …

        System.out.println(
            "Following are k randomly selected items");
        System.out.println(Arrays.toString(reservoir));
    }

    // Driver Program to test above method
    public static void main(String[] args)
    {
        int stream[]
            = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
        int n = stream.length;
        int k = 5;
        selectKItems(stream, n, k);
    }
}
```

2. Trees and Graphs

2.1 Tree Creation and Traversal Techniques (Recursive)

A tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.



Creation of Binary Tree:

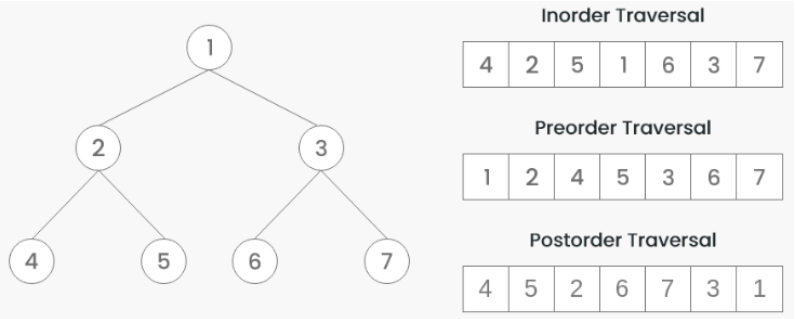
```
// Demonstration of Tree Basic Terminologies

class BT {

    // Function to print the parent of each node
    public static void
    printParents(int node, Vector<Vector<Integer> > adj,
                 int parent)
    {

        // Write code here
```

```
        …
    }

    // Function to print the children of each node
    public static void
    printChildren(int Root, Vector<Vector<Integer> > adj)
    {
        // Write code here
        …
    }

    // Function to print the leaf nodes
    public static void
    printLeafNodes(int Root, Vector<Vector<Integer> > adj)
    {
        // Write code here
        …
    }

    // Function to print the degrees of each node
    public static void
    printDegrees(int Root, Vector<Vector<Integer> > adj)
    {
        // Write code here
        …
    }

    // Driver code
    public static void main(String[] args)
    {

        // Number of nodes
        int N = 7, Root = 1;

        // Adjacency list to store the tree
        Vector<Vector<Integer> > adj
            = new Vector<Vector<Integer> >();
        for (int i = 0; i < N + 1; i++) {
            adj.add(new Vector<Integer>());
        }

        // Creating the tree
        adj.get(1).add(2);
        adj.get(2).add(1);

        adj.get(1).add(3);
        adj.get(3).add(1);

        adj.get(1).add(4);
        adj.get(4).add(1);

        adj.get(2).add(5);
        adj.get(5).add(2);
```

```
        adj.get(2).add(6);
        adj.get(6).add(2);

        adj.get(4).add(7);
        adj.get(7).add(4);

        // Printing the parents of each node
        System.out.println("The parents of each node are:");
        printParents(Root, adj, 0);

        // Printing the children of each node
        System.out.println(
            "The children of each node are:");
        printChildren(Root, adj);

        // Printing the leaf nodes in the tree
        System.out.println(
            "The leaf nodes of the tree are:");
        printLeafNodes(Root, adj);

        // Printing the degrees of each node
        System.out.println("The degrees of each node are:");
        printDegrees(Root, adj);
    }
}
```

A binary tree data structure can be traversed in following ways:

1. Inorder Traversal
2. Preorder Traversal
3. Postorder Traversal
4. Level Order Traversal



Algorithm Inorder (tree)

1. Traverse the left subtree, i.e., call Inorder(left->subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right->subtree)

Algorithm Preorder (tree)

1. Visit the root.

2. Traverse the left subtree, i.e., call Preorder(left->subtree)
3. Traverse the right subtree, i.e., call Preorder(right->subtree)

Algorithm Postorder (tree)

1. Traverse the left subtree, i.e., call Postorder(left->subtree)
2. Traverse the right subtree, i.e., call Postorder(right->subtree)
3. Visit the root.

```
// Java program for different tree traversals

// Class containing left and right child of current
// node and key value
class Node {
    int key;
    Node left, right;
    public Node(int item)
    {
        key = item;
        left = right = null;
    }
}

class BinaryTree {

    // Root of Binary Tree
    Node root;

    BinaryTree() { root = null; }

    // Given a binary tree, print its nodes in inorder
    void printInorder(Node node)
    {
        // Write code here
        …
    }
    // Given a binary tree, print its nodes in preorder
    void printPreorder(Node node)
    {
        // Write code here
        …
    }
    void printPostorder(Node node)
    {
        // Write code here
        …    }


    // Driver code
    public static void main(String[] args)
    {
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(1);
        tree.root.left = new Node(2);
```

```
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);

        // Function call
        System.out.println(
            "Inorder traversal of binary tree is ");
        tree.printInorder(tree.root);
        System.out.println(
            "Preorder traversal of binary tree is ");
        tree.printPreorder(tree.root);
        System.out.println(
            "Postorder traversal of binary tree is ");
        tree.printPostorder(tree.root);


    }
}
```

2.2 Tree Creation and Traversal Techniques (Iterative)

Inorder Traversal using Stack:

As we already know, recursion can also be implemented using stack. Here also we can use a stack to perform inorder traversal of a Binary Tree. Below is the algorithm for traversing a binary tree using stack.

1. Create an empty stack (say S).
2. Initialize the current node as root.
3. Push the current node to S and set current = current->left until current is NULL
4. If current is NULL and the stack is not empty then:
    - Pop the top item from the stack.
    - Print the popped item and set current = popped_item->right
    - Go to step 3.
5. If current is NULL and the stack is empty then we are done.

preorder Traversal using Stack:

Following is a simple stack based iterative process to print Preorder traversal.

1. Create an empty stack nodeStack and push root node to stack.
2. Do the following while nodeStack is not empty.
    1. Pop an item from the stack and print it.
    2. Push right child of a popped item to stack
    3. Push left child of a popped item to stack

The right child is pushed before the left child to make sure that the left subtree is processed first.

postorder Traversal using Stack:

1.1 Create an empty stack
2.1 Do following while root is not NULL
  a) Push root's right child and then root to stack.
  b) Set root as root's left child.
2.2 Pop an item from stack and set it as root.
  a) If the popped item has a right child and the right child

is at top of stack, then remove the right child from stack,
push the root back and set root as root's right child.

b) Else print root's data and set root as NULL.

2.3 Repeat steps 2.1 and 2.2 while stack is not empty.

```
//Inorder Traversal
class Node
{
    int data;
    Node left, right;

    public Node(int item)
    {
        data = item;
        left = right = null;
    }
}

// Class to print the inorder traversal
class BinaryTree
{
    Node root;
    void inorder()
    {
        if (root == null)
            return;
        Stack<Node> s = new Stack<Node>();
        Node curr = root;
        // Traverse the tree
        // Write code here
            ……
    }
     public static void main(String args[])
    {
        // Creating a binary tree and entering
        // the nodes
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);
        tree.inorder();
    }
}
```

```
// Preorder traversal
// A binary tree node
class Node {
    int data;
    Node left, right;
    Node(int item)
    {
        data = item;
        left = right = null;
    }
}
class BinaryTree {
    Node root;
    void iterativePreorder()    {
        iterativePreorder(root);
    }

    // An iterative process to print preorder traversal of Binary tree
    void iterativePreorder(Node node)
    {
        // Base Case
        if (node == null) {
            return;
        }
        // Write code here
            ……
    }

    // driver program to test above functions
    public static void main(String args[])
    {
        BinaryTree tree = new BinaryTree();
        tree.root = new Node(10);
        tree.root.left = new Node(8);
        tree.root.right = new Node(2);
        tree.root.left.left = new Node(3);
        tree.root.left.right = new Node(5);
        tree.root.right.left = new Node(2);
        tree.iterativePreorder();
    }
}
```

```
//Postorder Traversal
// A binary tree node
```

```
class Node {
    int data;
    Node left, right;
    Node(int item)
    {
        data = item;
        left = right;
    }
}
class BinaryTree {
    Node root;
    ArrayList<Integer> list = new ArrayList<Integer>();
     // Write code here
       ……..
    }
     // Driver program to test above functions
    public static void main(String args[])
    {
        BinaryTree tree = new BinaryTree();

        tree.root = new Node(1);
        tree.root.left = new Node(2);
        tree.root.right = new Node(3);
        tree.root.left.left = new Node(4);
        tree.root.left.right = new Node(5);
        tree.root.right.left = new Node(6);
        tree.root.right.right = new Node(7);

        ArrayList<Integer> mylist = tree.postOrderIterative(tree.root);

        System.out.println(
            "Post order traversal of binary tree is :");
        System.out.println(mylist);
    }
}
```

2.3 Hashing

Hashing is a technique or process of mapping keys, and values into the hash table by using a hash function. It is done for faster access to elements. The efficiency of mapping depends on the efficiency of the hash function used.In this case, the **find_second_largest** method calls the **find_largest()** function via implicit

recursion to locate the second-largest number in a provided list of numbers. Implicit recursion can be used in this way to get the second-largest integer without having to write any more code

Let a hash function H(x) maps the value **x** at the index **x%10** in an Array. For example if the list of values is [11,12,13,14,15] it will be stored at positions {1,2,3,4,5} in the array or Hash table respectively.

Given four sorted arrays each of size **n** of distinct elements. Given a value **x**. The problem is to count all **quadruples**(group of four numbers) from all the four arrays whose sum is equal to **x**.
Note: The quadruple has an element from each of the four arrays.
Examples:
Input : arr1 = {1, 4, 5, 6},
arr2 = {2, 3, 7, 8},
arr3 = {1, 4, 6, 10},
arr4 = {2, 4, 7, 8}
n = 4, x = 30
Output : 4
The quadruples are:
(4, 8, 10, 8), (5, 7, 10, 8),
(5, 8, 10, 7), (6, 7, 10, 7)
Input : For the same above given fours arrays
x = 25
Output : 14

Procedure using Hashing
Create a hash table where **(key, value)** tuples are represented as **(sum, frequency)** tuples. Here the sum are obtained from the pairs of 1st and 2nd array and their frequency count is maintained in the hash table. Now, generate all pairs from the 3rd and 4th array. For each pair so generated, find the sum of elements in the pair. Let it be **p_sum**. For each **p_sum**, check whether **(x – p_sum)** exists in the hash table or not. If it exists, then add the frequency of **(x – p_sum)** to the **count** of quadruples

```
// Java implementation to count quadruples from four sorted arrays
// whose sum is equal to a given value x
class HT {

    static int countQuadruples(int arr1[], int arr2[],
            int arr3[], int arr4[], int n, int x) {
        int count = 0;
        // Write code here
        ……..
    }

// Driver program to test above
    public static void main(String[] args) {

        // four sorted arrays each of size 'n'
        int arr1[] = {1, 4, 5, 6};
        int arr2[] = {2, 3, 7, 8};
```

```
        int arr3[] = {1, 4, 6, 10};
        int arr4[] = {2, 4, 7, 8};

        int n = arr1.length;
        int x = 30;
        System.out.println("Count = "
                + countQuadruples(arr1, arr2, arr3,
                        arr4, n, x));

    }
}
```

2.4 Graph Traversal Techniques

Graph Traversal Techniques:

a. **Breadth First Traversal (BFT)**
b. **Depth First Traversal (BFT)**

The **Breadth First Traversal (BFT)** algorithm is used to search a graph data structure for a node that meets a set of criteria. It starts at the root of the graph and visits all nodes at the current depth level before moving on to the nodes at the next depth level.

For a given graph G, print BFS traversal from a given source vertex.

```
class Graph {
    int vertices;
    LinkedList<Integer>[] adjList;

    @SuppressWarnings("unchecked") Graph(int vertices)
    {
        this.vertices = vertices;
        adjList = new LinkedList[vertices];
        for (int i = 0; i < vertices; ++i)
            adjList[i] = new LinkedList<>();
    }

    // Function to add an edge to the graph
    void addEdge(int u, int v) { adjList[u].add(v); }

    // Function to perform Breadth First Search on a graph
    // represented using adjacency list
    void bfs(int startNode)
    {
        // Write code here
        ……
    }
}

public class Main {
```

```
    public static void main(String[] args)
    {
        // Number of vertices in the graph
        int vertices = 5;

        // Create a graph
        Graph graph = new Graph(vertices);

        // Add edges to the graph
        graph.addEdge(0, 1);
        graph.addEdge(0, 2);
        graph.addEdge(1, 3);
        graph.addEdge(1, 4);
        graph.addEdge(2, 4);

        // Perform BFS traversal starting from vertex 0
        System.out.print(
            "Breadth First Traversal starting from vertex 0: ");
        graph.bfs(0);
    }
}


# BFT traversal from a given source vertex.

from collections import defaultdict

# This class represents a directed graph using adjacency list representation
class Graph:

    # Constructor
    def __init__(self):
        # Default dictionary to store graph
        self.graph = defaultdict(list)

    # Function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # Function to print a BFS of graph
    def BFT(self, s):
      # Write code here
      …
 # Create a graph given in the above diagram
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print("Following is Breadth First Traversal" " (starting from vertex 2)")
g.BFT(2)
```

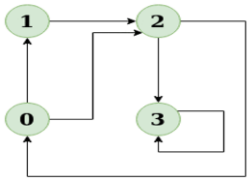
Output: Breadth First Traversal starting from vertex 0: 0 1 2 3 4

Output: Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1

Depth First Traversal (DFT) for a graph is similar to Depth First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a boolean visited array. A graph can have more than one DFS traversal.

For a given graph G, print DFS traversal from a given source vertex.
Input: n = 4, e = 6
0 -> 1, 0 -> 2, 1 -> 2, 2 -> 0, 2 -> 3, 3 -> 3
Output: DFS from vertex 1: 1 2 0 3

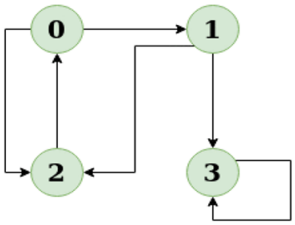
Explanation:
DFS Diagram:



Input: n = 4, e = 6
2 -> 0, 0 -> 2, 1 -> 2, 0 -> 1, 3 -> 3, 1 -> 3

Output: DFS from vertex 2: 2 0 1 3

Explanation:
DFS Diagram:




```
// Java program to print DFS traversal
// from a given graph
class Graph {
    private int V;

    // Array of lists for Adjacency List Representation
    private LinkedList<Integer> adj[];

    // Constructor
    @SuppressWarnings("unchecked") Graph(int v)
    {
        V = v;
```

```
        adj = new LinkedList[v];
        for (int i = 0; i < v; ++i)
            adj[i] = new LinkedList();
    }

    // Function to add an edge into the graph
    void addEdge(int v, int w)
    {
        // Add w to v's list.
        adj[v].add(w);
    }

    // A function used by DFS
    void DFSUtil(int v, boolean visited[])
    {
        // Write code here
        ……
    }

    // The function to do DFS traversal. It uses recursive DFSUtil()
    void DFS(int v)
    {
        // Write code here
        ……
    }

    // Driver Code
    public static void main(String args[])
    {
        Graph g = new Graph(4);

        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);

        System.out.println(
            "Following is Depth First Traversal "
            + "(starting from vertex 2)");

        // Function call
        g.DFS(2);
    }
}
```

2.5 Heap Sort

Heap sort is a comparison-based sorting technique based on Binary Heap data structure. It is similar to the selection sort where we first find the minimum element and place the minimum element at the beginning. Repeat the same process for the remaining elements.

Heap Sort Procedure:
First convert the array into heap data structure using heapify, then one by one delete the root node of the Max-heap and replace it with the last node in the heap and then heapify the root of the heap. Repeat this process until size of heap is greater than 1.

- Build a heap from the given input array.
- Repeat the following steps until the heap contains only one element:
  - Swap the root element of the heap (which is the largest element) with the last element of the heap.
  - Remove the last element of the heap (which is now in the correct position).
  - Heapify the remaining elements of the heap.
- The sorted array is obtained by reversing the order of the elements in the input array.

Input: arr = [12, 11, 13, 5, 6, 7]
Output: Sorted array is 5 6 7 11 12 13

```
// Java program for implementation of Heap Sort

public class HeapSort {
    public void sort(int arr[])
    {
        int N = arr.length;

        // Build heap (rearrange array)
        for (int i = N / 2 - 1; i >= 0; i--)
            heapify(arr, N, i);

        // One by one extract an element from heap
        for (int i = N - 1; i > 0; i--) {
            // Move current root to end
            // Write code here
               ……

            // call max heapify on the reduced heap
            heapify(arr, i, 0);
        }
    }

    // To heapify a subtree rooted with node i which is
    // an index in arr[]. n is size of heap
    void heapify(int arr[], int N, int i)
    {
        // Write code here
        ……    }

    /* A utility function to print array of size n */
    static void printArray(int arr[])
    {
        // Write code here
        ……
    }

    // Driver's code
```

```
    public static void main(String args[])
    {
        int arr[] = { 12, 11, 13, 5, 6, 7 };
        int N = arr.length;

        // Function call
        HeapSort ob = new HeapSort();
        ob.sort(arr);

        System.out.println("Sorted array is");
        printArray(arr);
    }
}
```

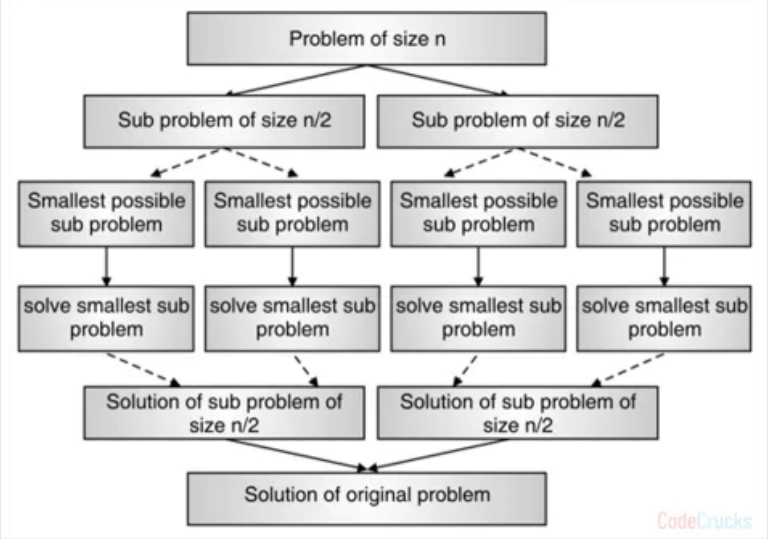
3. Divide and Conquer

General strategy of Divide and Conquer:
Divide and conquer algorithm operates in three stages:

- **Divide:** Divide the problem recursively into smaller subproblems.
- **Solve:** Subproblems are solved independently.
- **Combine:** Combine subproblem solutions in order to deduce the answer to the original large problem.

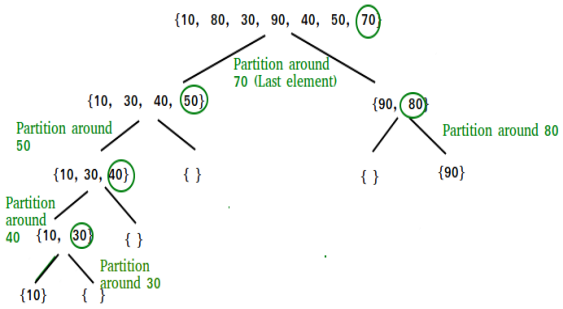
Because subproblems are identical to the main problem but have smaller parameters, they can be readily solved using recursion. When a subproblem is reduced to its lowest feasible size, it is solved, and the results are recursively integrated to produce a solution to the original larger problem.

Divide and conquer is a top-down, multi-branched recursive method. Each branch represents a subproblem and calls itself with a smaller argument. Understanding and developing divide and conquer algorithms requires expertise and sound reasoning. Divide and Conquer approach depicted graphically in the following figure. Sub Problems need not be exactly n/2 in size. Size of partition and number of partitions depends on application problem.



3.1 Quick Sort

QuickSort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array. The key process in quickSort is a partition(). The target of partitions is to place the pivot (any element can be chosen to be a pivot) at its correct position in the sorted array and put all smaller elements to the left of the pivot, and all greater elements to the right of the pivot. Partition is done recursively on each side of the pivot after the pivot is placed in its correct position and this finally sorts the array.



The quick sort method can be summarized in three steps:

1. **Pick:** Select a pivot element.
2. **Divide:** Split the problem set, move smaller parts to the left of the pivot and larger items to the right.
3. **Repeat and combine:** Repeat the steps and combine the arrays that have previously been sorted.

Algorithm for Quick Sort Function:

```
//start –> Starting index,  end --> Ending index
Quicksort(array, start, end)
{
        if (start < end)
        {
                pIndex = Partition(A, start, end)
                Quicksort(A,start,pIndex-1)
                Quicksort(A,pIndex+1, end)
        }
}
```

Algorithm for Partition Function:

```
partition (array, start, end)
{
        // Setting rightmost Index as pivot
        pivot = arr[end];

        i = (start - 1)  // Index of smaller element and indicates the
            // right position of pivot found so far
        for (j = start; j <= end- 1; j++)
```

```
        {
                // If current element is smaller than the pivot
                if (arr[j] < pivot)
                {
                        i++;    // increment index of smaller element
                        swap arr[i] and arr[j]
                }
        }
        swap arr[i + 1] and arr[end])
        return (i + 1)
}
```

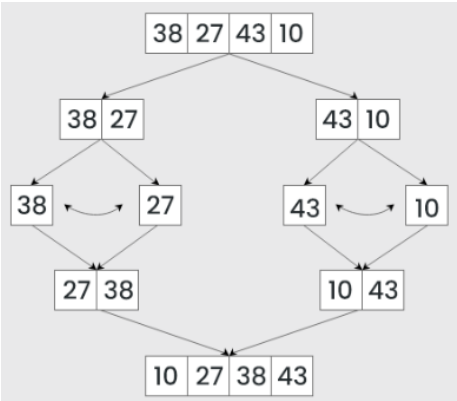
Input: arr = [10, 80, 30, 90, 40, 50, 70]
Output: arr = [10, 30, 40, 50, 70, 80, 90]

```
class QS {

    // A utility function to swap two elements
    static void swap(int[] arr, int i, int j)
    {
        // Write code here
         …
    }

    static int partition(int[] arr, int low, int high)
    {
        // Choosing the pivot
        int pivot = arr[high];

        // Write code here
         …
        swap(arr, i + 1, high);
        return (i + 1);
    }

    static void quickSort(int[] arr, int low, int high)
    {
        // Write code here
         …
    }
    // To print sorted array
    public static void printArr(int[] arr)
    {
        // Write code here
         …
    }

    // Driver Code
    public static void main(String[] args)
    {
        int[] arr = { 10, 7, 8, 9, 1, 5 };
        int N = arr.length;
```

```
        // Function call
        quickSort(arr, 0, N - 1);
        System.out.println("Sorted array:");
        printArr(arr);
    }
}
```

3.2 Merge Sort

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array. In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.



Input: arr = [12, 11, 13, 5, 6, 7]
Output: arr = [5, 6, 7, 11, 12, 13]

```
class MergeSort {
    void merge(int arr[], int l, int m, int r)
    {
        // Write code here
         …    }

    void sort(int arr[], int l, int r)
    {
        if (l < r) {
             // Find the middle point
            int m = l + (r - l) / 2;

            // Sort first and second halves
            sort(arr, l, m);
            sort(arr, m + 1, r);

            // Merge the sorted halves
            merge(arr, l, m, r);
        }
    }
```

```
    // A utility function to print array of size n
    static void printArray(int arr[])
    {
        // Write code here
         …
    }

    // Driver code
    public static void main(String args[])
    {
        int arr[] = { 12, 11, 13, 5, 6, 7 };

        System.out.println("Given array is");
        printArray(arr);

        MergeSort ob = new MergeSort();
        ob.sort(arr, 0, arr.length - 1);

        System.out.println("\nSorted array is");
        printArray(arr);
    }
}
```

3.3 Karatsuba Algorithm for fastest multiplication

Given two binary strings that represent value of two integers, find the product of two strings using **Karatsuba algorithm for fast multiplication**. For example, if the first bit string is "1100" and second bit string is "1010", output should be 120. For simplicity, let the length of two strings be same and be n.

or this algorithm, two n-digit numbers are taken as the input and the product of the two number is obtained as the output.

Step 1 − In this algorithm we assume that n is a power of 2.

Step 2 − If n = 1 then we use multiplication tables to find P = XY.

Step 3 − If n > 1, the n-digit numbers are split in half and represent the number using the formulae −

$$X = 10n/2X1 + X2$$

$$Y = 10n/2Y1 + Y2$$

where, X_1, X_2, Y_1, Y_2 each have n/2 digits.

Step 4 − Take a variable Z = W − (U + V),

where,

U = X1Y1, V = X2Y2
W = (X1 + X2) (Y1 + Y2), Z = X1Y2 + X2Y1.

Step 5 − Then, the product P is obtained after substituting the values in the formula −

P = 10n(U) + 10n/2(Z) + V

P = 10n (X1Y1) + 10n/2 (X1Y2 + X2Y1) + X2Y2.

Step 6 – Recursively call the algorithm by passing the sub problems (X_1, Y_1), (X_2, Y_2) and ($X_1 + X_2$, $Y_1 + Y_2$) separately. Store the returned values in variables U, V and W respectively.

```
public class Main {
   static long karatsuba(long X, long Y) {
      // Base Case
      if (X < 10 && Y < 10)
         return X * Y;
      // Write Code Here
         ……

   }
   static int get_size(long value) {
      // Write Code Here
         ……
   }
   public static void main(String args[]) {
      // two numbers
      long x = 145623;
      long y = 653324;
      System.out.print("The final product is: ");
      long product = karatsuba(x, y);
      System.out.println(product);
   }
}
```

3.4 Strassen's Matrix Multiplication

Strassen's Matrix Multiplication is the divide and conquer approach to solve the matrix multiplication problems. The usual matrix multiplication method multiplies each row with each column to achieve the product matrix. The time complexity taken by this approach is **$O(n^3)$**, since it takes two loops to multiply. Strassen's method was introduced to reduce the time complexity from **$O(n^3)$** to **$O(n^{\log 7})$**

Strassen's Matrix multiplication can be performed only on square matrices where n is a power of 2. Order of both of the matrices are n × n.
Divide X, Y and Z into four (n/2)×(n/2) matrices as represented below –

$$Z = \begin{bmatrix} I & J \\ K & L \end{bmatrix} \quad X = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \text{ and } \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

Using Strassen's Algorithm compute the following –

$$M_1 := (A + C) \times (E + F)$$
$$M_2 := (B + D) \times (G + H)$$
$$M_3 := (A - D) \times (E + H)$$
$$M_4 := A \times (F - H)$$
$$M_5 := (C + D) \times (E)$$
$$M_6 := (A + B) \times (H)$$
$$M_7 := D \times (G - E)$$

Then,

$$I := M_2 + M_3 - M_6 - M_7$$
$$J := M_4 + M_6$$
$$K := M_5 + M_7$$
$$L := M_1 - M_3 - M_4 - M_5$$

```
/** Java Program to Implement Strassen Algorithm**/
import java.util.Scanner;
/** Class Strassen **/
public class Strassen
{     /** Function to multiply matrices **/
      public int[][] multiply(int[][] A, int[][] B)
      {
            int n = A.length;
            int[][] R = new int[n][n];
            /** base case **/
            if (n == 1)
                  R[0][0] = A[0][0] * B[0][0];
            else
            {  ……..Write Code Here……..

            }
      }
      /** Function to sub two matrices **/
      public int[][] sub(int[][] A, int[][] B)
      {
            …….Write Code Here……..
            return C;
      }
      /** Function to add two matrices **/
      public int[][] add(int[][] A, int[][] B)
      {
            ……Write Code Here…….
            return C;
      }
      /** Function to split parent matrix into child matrices **/
      public void split(int[][] P, int[][] C, int iB, int jB) { ….Write
Code….}
      /** Function to join child matrices intp parent matrix **/
```

```
	public void join(int[][] C, int[][] P, int iB, int jB) { ….Write
Code….}
	/** Main function **/
	public static void main (String[] args)
	{
		….Write Code Here…..
	}
}
```

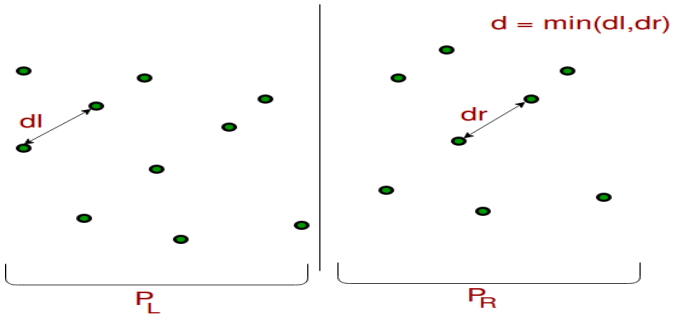
3.5 Closest Pair of Points

We are given an array of n points in the plane, and the problem is to find out the closest pair of points in the array. This problem arises in a number of applications. For example, in air-traffic control, you may want to monitor planes that come too close together, since this may indicate a possible collision. Recall the following formula for distance between two points p and q.

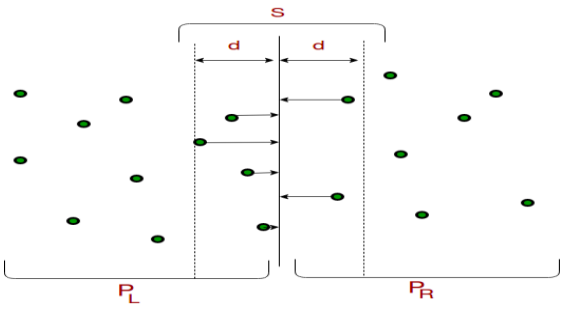
$$\|pq\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

The Brute force solution is O(n^2), compute the distance between each pair and return the smallest. We can calculate the smallest distance in O(nLogn) time using Divide and Conquer strategy. In this post, a O(n x (Logn)^2) approach is discussed. We will be discussing a O(nLogn) approach in a separate post.

Algorithm
Following are the detailed steps of a O (n (Logn)^2) algorithm.
Input: An array of n points P []
Output: The smallest distance between two points in the given array.
As a pre-processing step, the input array is sorted according to x coordinates.
1) Find the middle point in the sorted array, we can take P[n/2] as middle point.
2) Divide the given array in two halves. The first subarray contains points from P[0] to P[n/2]. The second subarray contains points from P[n/2+1] to P[n-1].
3) Recursively find the smallest distances in both subarrays. Let the distances be dl and dr. Find the minimum of dl and dr. Let the minimum be d.



4) From the above 3 steps, we have an upper bound d of minimum distance. Now we need to consider the pairs such that one point in pair is from the left half and the other is from the right half. Consider the vertical line passing through P[n/2] and find all points whose x coordinate is closer than d to the middle vertical line. Build an array strip [] of all such points.



5) Sort the array strip[] according to y coordinates. This step is O(nLogn). It can be optimized to O(n) by recursively sorting and merging.
6) Find the smallest distance in strip[]. This is tricky. From the first look, it seems to be a O(n^2) step, but it is actually O(n). It can be proved geometrically that for every point in the strip, we only need to check at most 7 points after it (note that strip is sorted according to Y coordinate).
7) Finally return the minimum of d and distance calculated in the above step (step 6)

```
import java.text.DecimalFormat;
import java.util.Arrays;
import java.util.Comparator;
// A divide and conquer program in Java to find the smallest distance from
// a given set of points A structure to represent a Point in 2D plane
class Point {
public int x;
public int y;
Point(int x, int y) {
      this.x = x;
      this.y = y;
}

// A utility function to find the distance between two points
public static float dist(Point p1, Point p2) {
      return (float) Math.sqrt((p1.x - p2.x) * (p1.x - p2.x) +
                                          (p1.y - p2.y) * (p1.y - p2.y)
                                          );
}
// A recursive function to find the smallest distance. The array P
// contains all points sorted according to x coordinate
public static float closestUtil(Point[] P, int startIndex, int endIndex)
{     // If there are 2 or 3 points, then use brute force
      if ((endIndex - startIndex) <= 3) {
      return bruteForce(P, endIndex);
      }
      ……..Write Code Here……
```

```
public class ClosestPoint {
// Driver code
public static void main(String[] args) { …….. Write Code Here……..
      DecimalFormat df = new DecimalFormat("#.######");
      System.out.println("The smallest distance is " +
                              df.format(Point.closest(P, P.length)));
}

}
```

4. Divide and Conquer

4.1 Tiling Problem using Divide and Conquer Algorithm

Given a n by n board where n is of form 2^k where k >= 1 (Basically n is a power of 2 with minimum value as 2). The board has one missing cell (of size 1 x 1). Fill the board using L shaped tiles. A L shaped tile is a 2 x 2 square with one cell of size 1×1 missing.

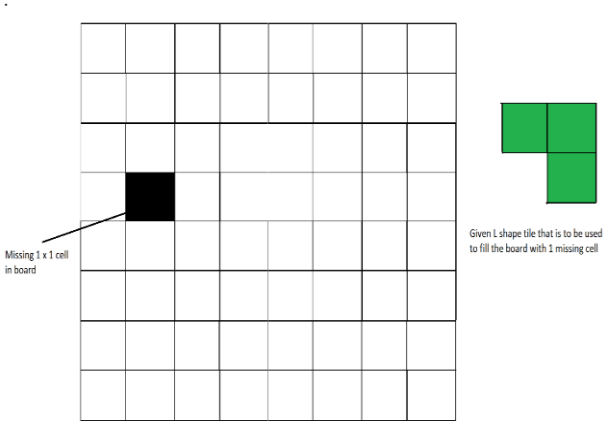


Figure 1: An example input

This problem can be solved using Divide and Conquer. Below is the recursive algorithm.
// n is size of given square, p is location of missing cell
Tile(int n, Point p)

1) Base case: n = 2, A 2 x 2 square with one cell missing is nothing but a tile and can be filled with a single tile.

2) Place a L shaped tile at the center such that it does not cover the n/2 * n/2 subsquare that has a missing square. **Now all four subsquares of size n/2 x n/2 have a missing cell** (a cell that doesn't need to be filled). See figure 2 below.

3) Solve the problem recursively for following four. Let p1, p2, p3 and p4 be positions of the 4 missing cells in 4 squares.
a) Tile(n/2, p1)

b) Tile(n/2, p2)
c) Tile(n/2, p3)
d) Tile(n/2, p3)

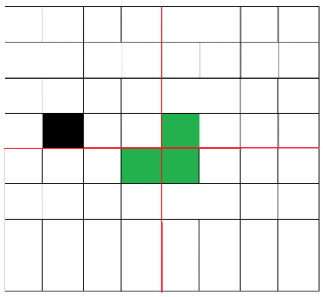


Figure 2: After placing the first tile

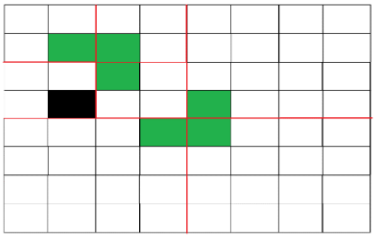


Figure 3: Recurring for the first subsquare.

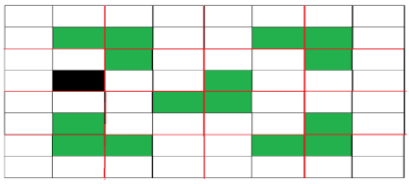


Figure 4: Shows the first step in all four subsquares.

Input : size = 2 and mark coordinates = (0, 0)
Output :
-1 1
1 1
Coordinate (0, 0) is marked. So, no tile is there. In the remaining three positions, a tile is placed with its number as 1.
Input : size = 4 and mark coordinates = (0, 0)
Output :
-1 3 2 2
3 3 1 2
4 1 1 5
4 4 5 5

```
public class TP
{
  static int size_of_grid, b, a, cnt = 0;
```

```
  static int[][] arr = new int[128][128];

  // Placing tile at the given coordinates
  static void place(int x1, int y1, int x2,
                    int y2, int x3, int y3)
  {
    // Write code here
       ……
  }
  // Quadrant names
  // 1   2
  // 3   4
  // Function based on divide and conquer
  static int tile(int n, int x, int y)
  {
    // Write code here
       ……
   }

  // Driver code
  public static void main(String[] args)
  {
    // size of box
    size_of_grid = 8;
    // Coordinates which will be marked
    a = 0; b = 0;
    // Here tile can not be placed
    arr[a][b] = -1;
    tile(size_of_grid, 0, 0);
    // The grid is
    for (int i = 0; i < size_of_grid; i++)
    {
      for (int j = 0; j < size_of_grid; j++)
        System.out.print(arr[i][j] + " ");
      System.out.println();;
    }
  }
}
```

4.2 The Skyline Problem

Given n rectangular buildings in a 2-dimensional city, computes the skyline of these buildings, eliminating hidden lines. The main task is to view buildings from a side and remove all sections that are not visible. All buildings share a common bottom and every **building** is represented by a triplet (left, ht, right)

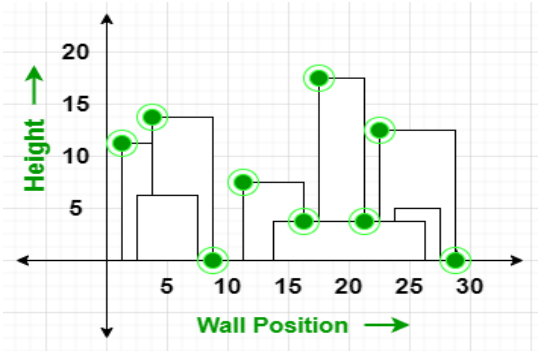
- 'left': is the x coordinate of the left side (or wall).
- 'right': is x coordinate of right side
- 'ht': is the height of the building.

A **skyline** is a collection of rectangular strips. A rectangular **strip** is represented as a pair (left, ht) where left is x coordinate of the left side of the strip and ht is the height of the strip.

Input: buildings[][] =
{ {1, 11, 5}, {2, 6, 7}, {3, 13, 9}, {12, 7, 16}, {14, 3, 25}, {19, 18, 22}, {23, 13, 29}, {24, 4, 28} }

Output: { {1, 11}, {3, 13}, {9, 0}, {12, 7}, {16, 3}, {19, 18}, {22, 3}, {23, 13}, {29, 0} }

Explanation:
The skyline is formed based on the key-points (representing by "green" dots) eliminating hidden walls of the buildings.
The Skyline Problem:
Input: buildings[ ][ ] = { {1, 11, 5} }
Output: { {1, 11}, {5, 0}



We can find Skyline using **Divide and Conquer**. The idea is similar to Merge Sort, divide the given set of buildings in two subsets. Recursively construct skyline for two halves and finally merge the two skylines. Start from first strips of two skylines, compare **x** coordinates. Pick the strip with smaller **x** coordinate and add it to result. The height of added strip is considered as maximum of current heights from skyline1 and skyline2.

```
// A class for building
class Building {
    int left, ht, right;

    public Building(int left, int ht, int right) {
        this.left = left;
        this.ht = ht;
        this.right = right;
    }
}
// A strip in skyline
class Strip {
    int left, ht;

    public Strip(int left, int ht) {
        this.left = left;
```

```
        this.ht = ht;
    }
}
// Skyline: To represent Output(An array of strips)
class SkyLine {
    List<Strip> arr;
    int capacity, n;

    public SkyLine(int cap) {
        this.arr = new ArrayList<>();
        this.capacity = cap;
        this.n = 0;
    }

    public int count() {
        return this.n;
    }

// A function to merge another skyline to this skyline
    public SkyLine merge(SkyLine other) {
        // Write code here
            ......
    }

// Function to add a strip 'st' to array
    public void append(Strip st) {
    // Write code here
            ......
    }

// A utility function to print all strips of skyline
    public void printSkyline() {
        // Write code here
            ......
    }
}
// This function returns skyline for a given array of buildings arr[l..h].
// This function is similar to mergeSort().
class SkylineProblem {
    public static SkyLine findSkyline(Building[] arr, int l, int h) {
          // Write code here
            ......
        return res;
    }
  // Driver Code
public static void main(String[] args) {
Building[] arr = {new Building(1, 11, 5), new Building(2, 6, 7), new Building(3, 13, 9),
new Building(12, 7, 16), new Building(14, 3, 25), new Building(19, 18, 22),
```

```
new Building(23, 13, 29), new Building(24, 4, 28)};

// Find skyline for given buildings and print the skyline
SkyLine res = findSkyline(arr, 0, arr.length-1);
res.printSkyline();
}
}
```

4.3 Allocate Minimum Number of Pages from N books to M students

Given that there are **N books** and **M students**. Also given are the **number of pages in each book in ascending order**. The task is to assign books in such a way that the **maximum number of pages assigned to a student is minimum**, with the condition that every student is assigned to read some consecutive books. Print that minimum number of pages.

Example :
Input: N = 4, pages[] = {12, 34, 67, 90}, M = 2
Output: 113
Explanation: There are 2 number of students. Books can be distributed in following combinations:

1. [12] and [34, 67, 90] -> Max number of pages is allocated to student '2' with 34 + 67 + 90 = 191 pages
2. [12, 34] and [67, 90] -> Max number of pages is allocated to student '2' with 67 + 90 = 157 pages
3. [12, 34, 67] and [90] -> Max number of pages is allocated to student '1' with 12 + 34 + 67 = 113 pages

Of the 3 cases, Option 3 has the minimum pages = 113.

Approach to solve this problem is to use Divide and Conquer, based on Binary Search idea:
Case 1: When no valid answer exists.

- If the number of students is greater than the number of books **(i.e, M > N)**, In this case at least 1 student will be left to which no book has been assigned.

Case 2: When a valid answer exists.

- The **maximum possible answer** could be when there is only one student. So, all the book will be assigned to him and the result would be the sum of pages of all the books.
- The **minimum possible answer** could be when number of student is equal to the number of book (i.e, M == N) , In this case all the students will get at most one book. So, the result would be the maximum number of pages among them **(i.e, maximum(pages[])).**
- Hence, we can apply binary search in this given range and each time we can consider the mid value as the maximum limit of pages one can get. And check for the limit if answer is valid then update the limit accordingly.

Below is the approach to solve this problem using Divide and Conquer:

- Calculate the mid and check if **mid** number of pages can be assigned to students such that all students will get at least one book.
- If yes, then update the result and check for the previous search space (end = mid-1)
- Otherwise, check for the next search space (start = mid+1)

```
public class NM {
   // Utility method to check if current minimum value is feasible or not.
   static boolean isPossible(int arr[], int n, int m,  int curr_min)
   {
      int studentsRequired = 1;
      int curr_sum = 0;
      // Write code here
        ……

      return studentsRequired <= m;
   }
   // method to find minimum pages
   static int findPages(int arr[], int n, int m)
   {
      int sum = 0;
      // return -1 if no. of books is less than no. of students
      if (n < m)
         return -1;
      int mx = arr[0];
      // Write code here
        ……
      return result;
   }
    // Driver Method
   public static void main(String[] args)
   {

      int arr[] = { 12, 34, 67,  90 };
 // Number of pages in books
      int m = 2; // No. of students
      System.out.println("Minimum number of pages = " + findPages(arr, arr.length, m));
   }
}
```

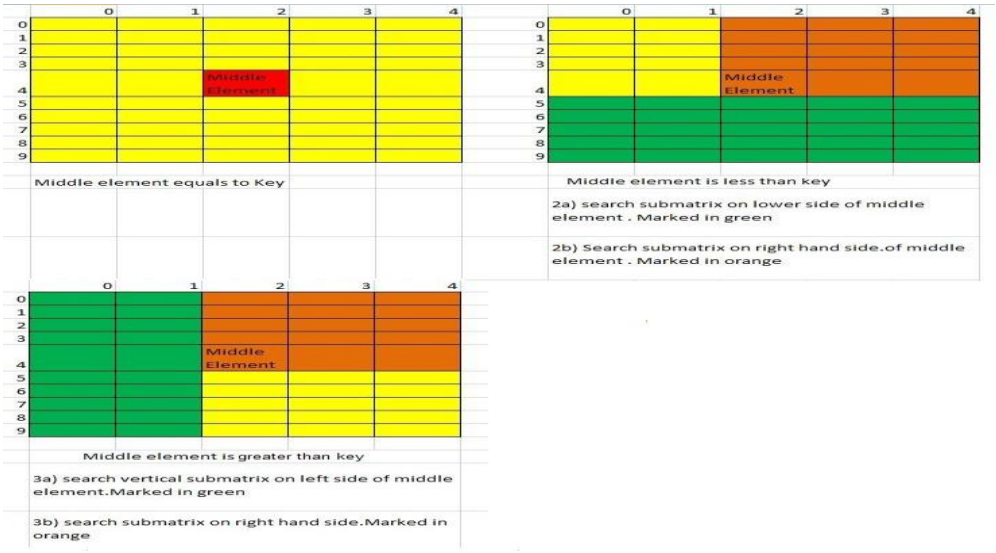
4.4 Efficiently merge `k` sorted linked lists
Minimum Number of Pages from N books to M students

Given k sorted linked lists, merge them into a single list in increasing order.
We already know that Two Linked Lists can be Merged in O(n) time and O(1) space (For arrays, O(n) space is required). The idea is to pair up k lists and merge each pair in linear time using the O(1) space. After the first cycle, K/2 lists are left each of size 2×N. After the second cycle, K/4 lists are left each of size 4×N and so on. Repeat the procedure until we have only one list left.

4.5 Search in a Row-wise and Column-wise Sorted 2D Array using Divide and Conquer algorithm

Given an n x n matrix, where every row and column is sorted in increasing order. Given a key, how to decide whether this key is in the matrix. This problem will be a very good example for divide and conquer algorithm.

1) Find the middle element.
2) If middle element is same as key return.
3) If middle element is lesser than key then
....3a) search submatrix on lower side of middle element
....3b) Search submatrix on right hand side.of middle element
4) If middle element is greater than key then
....4a) search vertical submatrix on left side of middle element
....4b) search submatrix on right hand side.**Shell Sort Procedure:**




```
class SearchInMatrix
{
   public static void main(String[] args)
   {
      int[][] mat = new int[][] { {10, 20, 30, 40},
                                  {15, 25, 35, 45},
                                  {27, 29, 37, 48},
                                  {32, 33, 39, 50}};
      int rowcount = 4,colCount=4,key=50;
      for (int i=0; i<rowcount; i++)
        for (int j=0; j<colCount; j++)
           search(mat, 0, rowcount-1, 0, colCount-1, mat[i][j]);
   }

   public static void search(int[][] mat, int fromRow, int toRow,
                             int fromCol, int toCol, int key)
   {
```

```
        // Write code here
          ......

    }
}
```

5. Greedy Technique

Greedy Algorithm is optimization method. When the problem has many feasible solutions with different cost or benefit, finding the best solution is known as an optimization problem and the best solution is known as the optimal solution.
The greedy algorithm derives the solution step by step, by looking at the information available at the current moment. It does not look at future prospects. Decisions are completely locally optimal. This method constructs the solution simply by looking at current benefit without exploring future possibilities and hence they are known as greedy. The choice made under greedy solution procedure are irrevocable, means once we have selected the local best solution, it cannot be backtracked. Thus, a choice made at each step in the greedy method should be:

- Feasible: choice should satisfy problem constraints.
- Locally optimal: Best solution from all feasible solution at the current stage should be selected.
- Irrevocable: Once the choice is made, it cannot be altered, i.e. if a feasible solution is selected (rejected) in step i, it cannot be rejected (selected) in subsequent stages.

5.1 Optimal Storage on Tapes

Optimal Storage on Tapes is one of the applications in the Greedy Method. The objective of this algorithm is to find the Optimal retrieval time for accessing programs that are stored on tape.

This algorithm works in steps. In each step, it selects the best available options until all options are finished.

EXPLANATION

- There are 'n' programs that are to be stored on a computer tape of length (L).
- Associated with each program (i) is a length (l).
- Let the programs are stored in the order (I = i1, i2, i3, ...)

$$\sum_{k=1}^{j} l_{ik}$$

If all the programs retrieved often the **Expected or Mean Retrieval Time (MRT)** is

$$MRT = \frac{1}{n}\sum_{j=1}^{n} t_j$$

- If all programs are retrieved equally often then the expected or Mean Retrieval Time (MRT) is,

$$d(I) = \sum_{j=1}^{n} \sum_{k=1}^{j} l_{ik}$$

```
// Java Program to find the order of programs for which MRT is minimized
import java.io.*;
import java .util.*;
class GFG
{
// This functions outputs the required order and Minimum Retrieval Time
static void findOrderMRT(int []L, int n)
{     // Here length of i'th program is L[i]
      Arrays.sort(L);
      System.out.print("Optimal order in which " +
                  "programs are to be stored is: ");
      // Write code Here
            ……….
      // MRT - Minimum Retrieval Time
      double MRT = 0;
      for (int i = 0; i < n; i++)
      {
            // Write code Here
            ……….
      }
      System.out.print( "Minimum Retrieval Time" +
                              " of this order is " + MRT);
}
// Driver Code
public static void main (String[] args)
{
      int []L = { 2, 5, 4 };
      int n = L.length;
      findOrderMRT(L, n);
}
}
```

5.2 Job sequencing with deadlines

Job scheduling algorithm is applied to schedule the jobs on a single processor to maximize the profits.

The greedy approach of the job scheduling algorithm states that, "Given 'n' number of jobs with a starting time and ending time, they need to be scheduled in such a way that maximum profit is received within the maximum deadline".

Job Scheduling Algorithm:

Set of jobs with deadlines and profits are taken as an input with the job scheduling algorithm and scheduled subset of jobs with maximum profit are obtained as the final output.

Step1: Find the maximum deadline value from the input set of jobs.
Step2: Once, the deadline is decided, arrange the jobs in descending order of their profits.
Step3: Selects the jobs with highest profits, their time periods not exceeding the maximum deadline.
Step4: The selected set of jobs are the output.

```
// Java code for the above problem
import java.util.*;
class Job {
      // Each job has a unique-id,profit and deadline
      char id;
      int deadline, profit;
      // Constructors
      public Job() {}
      public Job(char id, int deadline, int profit)
      {     this.id = id;
            this.deadline = deadline;
            this.profit = profit;
      }
      // Function to schedule the jobs take 2 arguments
      // arraylist and no of jobs to schedule
      void printJobScheduling(ArrayList<Job> arr, int t)
      {
            //  Write Code Here
      …….  …….     ……
            }
      }

      // Driver's code
      public static void main(String args[])
      {
            ArrayList<Job> arr = new ArrayList<Job>();
            arr.add(new Job('a', 2, 100));
            arr.add(new Job('b', 1, 19));
            arr.add(new Job('c', 2, 27));
            arr.add(new Job('d', 1, 25));
            arr.add(new Job('e', 3, 15));

      System.out.println("Following is maximum profit sequence of jobs");
            Job job = new Job();
            // Function call
            job.printJobScheduling(arr, 3);
      }
}
```

5.3 Single source shortest path - Dijkstra's Algorithm

Dijkstra's Algorithm is also known as Single Source Shortest Path (SSSP) problem. It is used to find the shortest path from source node to destination node in graph.

The graph is widely accepted data structure to represent distance map. The distance between cities effectively represented using graph.

- Dijkstra proposed an efficient way to find the single source shortest path from the weighted graph. For a given source vertex s, the algorithm finds the shortest path to every other vertex v in the graph.
- Assumption: Weight of all edges is non-negative.
- Steps of the Dijkstra's algorithm are explained here:

1. Initializes the distance of source vertex to zero and remaining all other vertices to infinity.
2. Set source node to current node and put remaining all nodes in the list of unvisited vertex list. Compute the tentative distance of all immediate neighbour vertex of the current node.
3. If the newly computed value is smaller than the old value, then update it.
4. Weight updating in Dijkstra's Algorithm: When all the neighbours of a current node are explored, mark it as visited. Remove it from unvisited vertex list. Mark the vertex from unvisited vertex list with minimum distance and repeat the procedure.
5. Stop when the destination node is tested or when unvisited vertex list becomes empty.

```
// A Java program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph
import java.io.*;
import java.lang.*;
import java.util.*;
class ShortestPath {
      static final int V = 9;
      int minDistance(int dist[], Boolean sptSet[])
      {     // Initialize min value
            // Write Code Here
      …….     ……..
      }
      // A utility function to print the constructed distance array
      void printSolution(int dist[])
      {     // Write Code Here
      }
      // Function that implements Dijkstra's single source shortest path
// algorithm for a graph represented using adjacency matrix representation
      void dijkstra(int graph[][], int src)
      {     int dist[] = new int[V];
// The output array. dist[i] will hold the shortest distance from src to i
      }
// Distance of source vertex from itself is always 0
      // Write Code Here
      ……..     ……
```

```
// Driver's code
      public static void main(String[] args)
      {  // Write Code Here
      ……..    ……
          // Function call
            t.dijkstra(graph, 0);
      }
}
```

5.4 Minimum spanning trees

A minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected, undirected graph is a spanning tree with a weight less than or equal to the weight of every other spanning tree.
For a given Infix expression, convert it into Postfix form.
The minimum spanning tree from a graph is found using the following algorithms:

1. Prim's Algorithm
2. Kruskal's Algorithm

Prim's Algorithm

Prim's Algorithm begins with a single Node and adds up adjacent nodes one by one by discovering all of the connected edges along the way. Edges with the lowest weights that don't generate cycles are chosen for inclusion in the MST structure. As a result, we can claim that Prim's algorithm finds the globally best answer by making locally optimal decisions.

Steps involved in Prim's algorithms are mentioned below:

Step 1: Choose any vertex as a starting vertex.
Step 2: Pick an edge connecting any tree vertex and fringe vertex (adjacent vertex to visited vertex) having the minimum edge weight.
Step 3: Add the selected edge to MST only if it doesn't form any closed cycle.
Step 4: Keep repeating steps 2 and 3 until the fringe vertices exist.
Step 5: End.

```
// A Java program for Prim's Minimum Spanning Tree (MST) algorithm.
import java.io.*;
import java.lang.*;
import java.util.*;
class MST {
      // Number of vertices in the graph
      private static final int V = 5;
      // Write Code Here
      …..   …..   ….
      void printMST(int parent[], int graph[][])
      {
            // Write Code Here
      …..   …..   ….
      }

      void primMST(int graph[][])
      {
            // Write Code Here
```

```
      …..   …..   ….          }
}
public static void main(String[] args)
      {     MST t = new MST();
            int graph[][] = new int[][] { { 0, 2, 0, 6, 0 },
            // Write Code Here
      …..   …..   ….
            t.primMST(graph);
      }
}
```

Kruskal's Algorithm

Kruskal's approach sorts all the edges in ascending order of edge weights and only adds nodes to the tree if the chosen edge does not form a cycle. It also selects the edge with the lowest cost first and the edge with the highest cost last. As a result, we can say that the Kruskal's Algorithm makes a locally optimum decision in the hopes of finding the global optimal solution. Hence, this algorithm can also be considered as a Greedy Approach.

The steps involved in Kruskal's algorithm to generate a minimum spanning tree are:

Step 1: Sort all edges in increasing order of their edge weights.
Step 2: Pick the smallest edge.
Step 3: Check if the new edge creates a cycle or loop in a spanning tree.
Step 4: If it doesn't form the cycle, then include that edge in MST. Otherwise, discard it.
Step 5: Repeat from step 2 until it includes |V| - 1 edges in MST.

```
// Java program for Kruskal's algorithm
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
public class KruskalsMST {
      // Defines edge structure
      static class Edge {
            // Write Code Here
      ….    …     ….
      }
      // Defines subset element structure
      static class Subset {
            // Write Code Here
      ….    …     ….
      }
      // Starting point of program execution
      public static void main(String[] args)
      { // Write Code Here
      ….    …     …. };
            kruskals(V, graphEdges);
      }

      // Function to find the MST
      private static void kruskals(int V, List<Edge> edges)
```

```
      { // Write Code Here
      ….    …     ….
            }
system.out.println( "Following are the edges of the constructed MST:");
            int minCost = 0;
            // Write Code Here
      ….    …     ….

      }
      // Function to unite two disjoint sets
      private static void union(Subset[] subsets, int x, int y)
      { // Write Code Here
      ….    …     ….
      }
      // Function to find parent of a set
      private static int findRoot(Subset[] subsets, int i)
      {
            if (subsets[i].parent == i)
                  return subsets[i].parent;

            subsets[i].parent
                  = findRoot(subsets, subsets[i].parent);
            return subsets[i].parent;
      }
}
```

5.5 Huffman coding

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.
The variable-length codes assigned to input characters are Prefix coding, means the codes (bit sequences) are assigned in such a way that the code assigned to one character is not the prefix of code assigned to any other character. This is how Huffman Coding makes sure that there is no ambiguity when decoding the generated bitstream.
Let us understand prefix codes with a counter example. Let there be four characters a, b, c and d, and their corresponding variable length codes be 00, 01, 0 and 1. This coding leads to ambiguity because code assigned to c is the prefix of codes assigned to a and b. If the compressed bit stream is 0001, the de-compressed output may be “cccd” or “ccb” or “acd” or “ab”.

There are mainly two major parts in Huffman Coding

1. Build a Huffman Tree from input characters.
2. Traverse the Huffman Tree and assign codes to characters.

Algorithm:
The method which is used to construct optimal prefix code is called **Huffman coding**.

This algorithm builds a tree in bottom up manner. We can denote this tree by **T**

Let, |c| be number of leaves |c| -1 are number of operations required to merge the nodes. Q be the priority queue which can be used while constructing binary heap.

```
Algorithm Huffman (c)
{
  n= |c|
  Q = c
  for i<-1 to n-1
  do
  {
     temp <- get node ()

    left (temp] Get_min (Q) right [temp] Get Min (Q)
    a = left [templ b = right [temp]
    F [temp]<- f[a] + [b]
    insert (Q, temp)
  }
return Get_min (0)
}
```

Steps to build Huffman Tree

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)
2. Extract two nodes with the minimum frequency from the min heap.

3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.
4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete.
   Let us understand the algorithm with an example:

```
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Scanner;
class Huffman {
// recursive function to print the huffman-code through the tree raversal.
// Here s is the huffman - code generated.
      public static void printCode(HuffmanNode root, String s)
      { // Write Code Here
            …     …     …
      }
// main function
      public static void main(String[] args)
      {     Scanner s = new Scanner(System.in);
            int n = 6;
            char[] charArray = { 'a', 'b', 'c', 'd', 'e', 'f' };
```

```
            int[] charfreq = { 5, 9, 12, 13, 16, 45 };
            PriorityQueue<HuffmanNode>
            Q = new PriorityQueue<HuffmanNode>(n, new MyComparator());
            for (int i = 0; i < n; i++) {
                  // Write Code Here
                        …     ….    …
            }
            // create a root node
            HuffmanNode root = null;

            // Write Code Here
                        …     ….    …}
}
//node class is the basic structure of each node present in the Huffman -
class HuffmanNode {
      int data;
      char c;
      HuffmanNode left;
      HuffmanNode right;
}
class MyComparator implements Comparator<HuffmanNode> {
      public int compare(HuffmanNode x, HuffmanNode y)
      { // Write Code Here
                        …     ….    …     }
}
```

6. Greedy Technique

6.1 Assign Mice to Holes

Problem Statement and Example

Given the positions of mice and holes along a line, and the speed of each mouse, the task is to find the arrangement of mice in the holes that minimizes the maximum time taken for any mouse to reach its hole. Each mouse should be assigned to a unique hole, and the order of mice and holes should be preserved.

For example, consider the following scenario:

- Mice Positions: -3, 7, 5, 9, 16
- Hole Positions: 1, 9, 15, 4, 14

The goal is to arrange the mice in the holes in a way that minimizes the maximum time taken for any mouse to reach its hole.

Idea to Solve the Problem

To solve this problem, we can follow these steps:

1. Sort the positions of mice and holes in ascending order.

2. Iterate through the sorted arrays and calculate the absolute difference between each mouse's position and its corresponding hole's position.
3. Track the maximum absolute difference during the iteration.
4. The maximum absolute difference represents the time it takes for the mouse to reach its hole in the optimal arrangement.

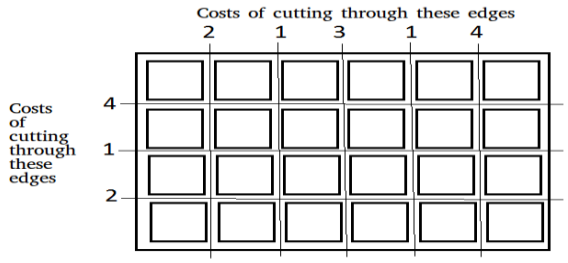
Algorithm Explanation

1. Sort both the mices and holes arrays in ascending order.
2. Initialize a variable result to store the maximum absolute difference (time taken).
3. Iterate through the arrays from 0 to n-1 (where n is the length of the arrays): a. Calculate the absolute difference between mices[i] and holes[i]. b. Update result if the calculated difference is greater than the current result.
4. Print the result, which represents the minimum time taken for the mice to reach their holes in an optimal arrangement.

```
// Java program to find the minimum time to place all mice in all holes.
import java.util.*;
public class GFG
{
      // Returns minimum time required to place mice in holes.
public int assignHole(ArrayList<Integer> mice,ArrayList <Integer> holes)
      {
            if (mice.size() != holes.size())
            return -1;
            /* Sort the lists */
            // Write Code Here
                  …     ….    …
            /* finding max difference between ith mice and hole */
      }

      /* Driver Function to test other functions */
      public static void main(String[] args)
      {
            GFG gfg = new GFG();
            ArrayList<Integer> mice = new ArrayList<Integer>();
            mice.add(4);
            mice.add(-4);
            mice.add(2);
            ArrayList<Integer> holes= new ArrayList<Integer>();
            holes.add(4);
            holes.add(0);
            holes.add(5);
            System.out.println("The last mouse gets into "+
            "the hole in time: "+gfg.assignHole(mice, holes));
      }
}
```

6.2 Minimum Cost to cut a board into squares

A board of length m and width n is given, we need to break this board into m*n squares such that cost of breaking is minimum. cutting cost for each edge will be given for the board. In short, we need to choose such a sequence of cutting such that cost is minimized.
Examples:



For above board optimal way to cut into square is:

Total minimum cost in above case is 42. It is

evaluated using following steps.

Initial Value : Total_cost = 0

Total_cost = Total_cost + edge_cost * total_pieces

Cost 4 Horizontal cut Cost = 0 + 4*1 = 4

Cost 4 Vertical cut Cost = 4 + 4*2 = 12

Cost 3 Vertical cut Cost = 12 + 3*2 = 18

Cost 2 Horizontal cut Cost = 18 + 2*3 = 24

Cost 2 Vertical cut Cost = 24 + 2*3 = 30

Cost 1 Horizontal cut Cost = 30 + 1*4 = 34

Cost 1 Vertical cut Cost = 34 + 1*4 = 38

Cost 1 Vertical cut Cost = 38 + 1*4 = 42

Minimum Cost to cut a board into squares using a greedy algorithm:

This problem can be solved by greedy approach . To get minimum cost , the idea is to cut the edge with highest cost first because we have less number of pieces and after every cut the number of pieces increase . As the question stated T**otal_cost = Total_cost + edge_cost * total_pieces .**

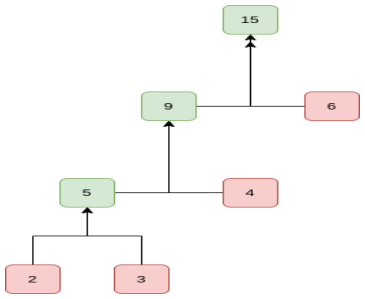
- At first sort both the array in non-ascending order
- We keep count of two variables vert(keeps track of vertical pieces) and **hzntl**(keeps track of horizontal pieces). We will initialize both of them with 1.
- We will keep track of two pointers starting from **0th** index of both the array
- Now we will take the highest cost edge from those pointers and multiply them with the corresponding variable. That is if we cut a horizontal cut we will **add (edge_cost*hzntl)** and increase vert by **1** and if we cut a vertical cut we will **add(edge_cost*vert)** and increase **hzntl** bt **1** .
- After cutting all the edges we will get the minimum cost

```
// Java program to divide a board into m*n squares
import java.util.Arrays;
import java.util.Collections;
class GFG
{
      // method returns minimum cost to break board into m*n squares
static int minimumCostOfBreaking(Integer X[], Integer Y[], int m, int n)
      {     int res = 0;
            // sort the horizontal cost in reverse order
            //Write Code here
            …..   …..   …..
            while (i < m && j < n)
            {//Write Code here
            …..   …..   …..

            }

            // loop for horizontal array, if remains
            //Write Code here
            …..   …..   …..

            // loop for vertical array,if remains
            //Write Code here
            …..   …..   …..
      }

      // Driver program
      public static void main(String arg[])
      {
            int m = 6, n = 4;
            Integer X[] = {2, 1, 3, 1, 4};
            Integer Y[] = {4, 1, 2};
            System.out.print(minimumCostOfBreaking(X, Y, m-1, n-1));
      }
}
```

6.3 Connect n ropes with minimum cost

Given are N ropes of different lengths, the task is to connect these ropes into one rope with minimum cost, such that the cost to connect two ropes is equal to the sum of their lengths.
Examples:
Input: arr[] = {4,3,2,6} , N = 4
Output: 29
Explanation:

1. First, connect ropes of lengths 2 and 3. Now we have three ropes of lengths 4, 6, and 5.
2. Now connect ropes of lengths 4 and 5. Now we have two ropes of lengths 6 and 9.
3. Finally connect the two ropes and all ropes have connected.



Input: arr[] = {1, 2, 3} , N = 3
Output: 9
Explanation:

1. First, connect ropes of lengths 1 and 2. Now we have two ropes of lengths 3 and 3.
2. Finally connect the two ropes and all ropes have connected.

Approach: If we observe the above problem closely, we can notice that the lengths of the ropes which are picked first are included more than once in the total cost. Therefore, the idea is to connect the smallest two ropes first and recur for the remaining ropes. This approach is similar to Huffman Coding. We put the smallest ropes down the tree so they can be repeated multiple times rather than the longer ones.

Algorithm: Follow the steps mentioned below to implement the idea:

- Create a min-heap and insert all lengths into the min-heap.
- Do following while the number of elements in min-heap is greater than one.
  - Extract the minimum and second minimum from min-heap
  - Add the above two extracted values and insert the added value to the min-heap.
  - Maintain a variable for total cost and keep incrementing it by the sum of extracted values.
- Return the value of total cost.

```
// Java program to connect n ropes with minimum cost
import java.util.*;
class ConnectRopes {
      static int minCost(int arr[], int n)
      {// Create a priority queue
            PriorityQueue<Integer> pq = new PriorityQueue<Integer>();
      // Write Code Here
      …     …..   ……
            return res;
      }

      // Driver program to test above function
      public static void main(String args[])
      {
            int len[] = { 4, 3, 2, 6 };
            int size = len.length;
            System.out.println("Total cost for connecting"
```

```
                                    + " ropes is "
                                    + minCost(len, size));
        }
}
```

6.4 Rearrange a string so that all same characters become d distance away

Given a string and a positive integer d. Some characters may be repeated in the given string. Rearrange characters of the given string such that the same characters become d distance away from each other. Note that there can be many possible rearrangements, the output should be one of the possible rearrangements. If no such arrangement is possible, that should also be reported.
The expected time complexity is O(n + m log(MAX)) Here n is the length of string, m is the count of distinct characters in a string and MAX is the maximum possible different characters.

Examples:
Input: "abb", d = 2
Output: "bab"
Input: "aacbbc", d = 3
Output: "abcabc"
Input: "aaa", d = 2
Output: Cannot be rearranged

The approach to solving this problem is to count frequencies of all characters and consider the most frequent character first and place all occurrences of it as close as possible. After the most frequent character is placed, repeat the same process for the remaining characters.

1. Let the given string be str and size of string be n
2. Traverse str, store all characters and their frequencies in a Max Heap MH(implemented using priority queue). The value of frequency decides the order in MH, i.e., the most frequent character is at the root of MH.
3. Make all characters of str as '\0'.
4. Do the following while MH is not empty.
    - Extract the Most frequent character. Let the extracted character be x and its frequency be f.
    - Find the first available position in str, i.e., find the first '\0' in str.
    - Let the first position be p. Fill x at p, p+d,.. p+(f-1)d

```
// Java program to rearrange a string so that all same
// characters become atleast d distance away
import java.util.*;
class GFG
{
static int MAX_CHAR = 256;
```

```
// The function returns next eligible character with maximum frequency
// (Greedy!!) and zero or negative distance
static int nextChar(int freq[], int dist[])
{
      // Write code Here
      ….    …..   …
}
// The main function that rearranges input string 'str'
static int rearrange(char str[], char out[], int d)
{
      int n = str.length;

// Create an array to store all characters and their frequencies in str[]
      int []freq = new int[MAX_CHAR];

      // Write code Here
      ….    …..   …

      return 1;
}

// Driver code
public static void main(String[] args)
{
      char str[] = "aaaabbbcc".toCharArray();
      int n = str.length;

      // To store output
      char []out = new char[n];

      if (rearrange(str, out, 2)==1)
                  System.out.println(String.valueOf(out));
      else
                  System.out.println("Cannot be rearranged");
}
}
```

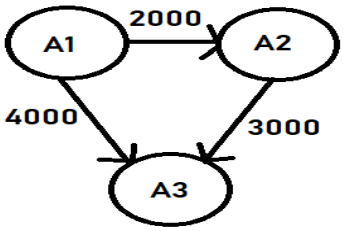
6.5 Minimization of Cash Flow among a given set of friends

Problem Statement

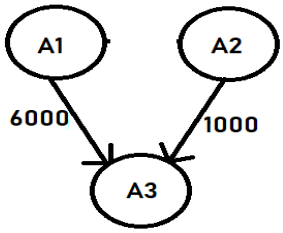
Suppose you are given that there are few friends. They have borrowed money from each other due to which there will be some cash flow on the network. Our main aim is to design an algorithm by which the total cash flow among all the friends is minimized.

For example:

There are three friends A1, A2, A3; you have to show the settlement of input debts between them.



A1 has to pay 2000 to A2 and 4000 to A3, A2 has to pay 3000 to A3



we can minimize the flow between A1 and A2 by directly paying to A3.

Solution using Greedy Approach

The greedy approach is used to build the solution in pieces, and this is what we want to minimize the cash flow. At every step, we will settle all the amounts of one person and recur for the remaining n-1 persons.

Calculate the net amount for every person, which can be calculated by subtracting all the debts, i.e., the amount to be paid from all credit, i.e., the amount to be paid to him. After this, we will find two persons with the maximum and the minimum net amounts. The person with a minimum of two is our first person to be settled and removed from the list.

Following algorithm will be done for every person varying 'i' from 0 to n-1.

1. The first step will be to calculate the net amount for every person and store it in an amount array. Net amount = sum(received money) - sum(sent money).
2. Find the two people that have the most credit and the most debt. Let the maximum amount to be credited from maximum creditor be max_credit and the maximum amount to be debited from maximum debtor be max_debit. Let the maximum debtor be debt and maximum creditor be cred.
3. Let the minimum of two amounts be 'y'.
4. If y equals max_credit, delete cred from the list and repeat for the remaining (n-1) people.
5. If y equals max_debit, delete debt from the group of people and repeat for recursion.
6. If the amount is 0, then the settlement is done.

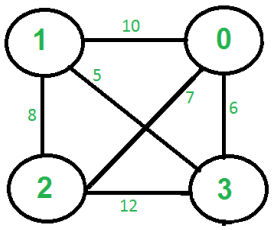
```
// Java program to find maximum cash  flow among a set of persons

class GFG
{
        // Number of persons (or vertices in the graph)
        static final int N = 3;

        // A utility function that returns  index of minimum value in arr[]
        static int getMin(int arr[])
        {
                int minInd = 0;
                for (int i = 1; i < N; i++)
                        if (arr[i] < arr[minInd])
                                minInd = i;
                return minInd;
        }
```

```
        // A utility function that returns index of maximum value in arr[]
        static int getMax(int arr[])
        {
                // Write Code Here
                …..     …….     ….
        }
        // A utility function to return minimum of 2 values
        static int minOf2(int x, int y)
        {
                return (x < y) ? x: y;
        }

        static void minCashFlowRec(int amount[])
        {
                // Write Code Here
                …..     …….     ….
        }

        static void minCashFlow(int graph[][])
        {// Write Code Here
                …..     …….     ….
        }
        // Driver code
        public static void main (String[] args)
        {
                // graph[i][j] indicates the amount  that person i needs to pay person j
                int graph[][] = { {0, 1000, 2000}, {0, 0, 5000}, {0, 0, 0},};
                minCashFlow(graph);
        }
}
```

6.6 Greedy Approximate Algorithm for K Centres Problem

Given n cities and distances between every pair of cities, select k cities to place warehouses (or ATMs or Cloud Server) such that the maximum distance of a city to a warehouse (or ATM or Cloud Server) is minimized.

For example consider the following four cities, 0, 1, 2, and 3, and the distances between them, how to place 2 ATMs among these 4 cities so that the maximum distance of a city to an ATM is minimized.



k = 2

The two ATMs should be placed in cities 2 and 3. The maximum distance of a city from an ATM becomes 6 in this optimal placement (We can not get the maximum distance less than 7)

There is no polynomial-time solution available for this problem as the problem is a known NP-Hard problem. There is a polynomial-time Greedy approximate algorithm, the greedy algorithm provides a solution that is never worse than twice the optimal solution. The greedy solution works only if the distances between cities follow Triangular Inequality (The distance between two points is always smaller than the sum of distances through a third point).

Approximate Greedy Algorithm:

1. Choose the first centre arbitrarily.
2. Choose remaining k-1 centres using the following criteria.
    - Let c1, c2, c3, … ci be the already chosen centres. Choose
    - (i+1)'th centre by picking the city which is farthest from already
    - selected centres, i.e, the point p which has following value as maximum
    - Min[dist(p, c1), dist(p, c2), dist(p, c3), …. dist(p, ci)]

```
// Java program for the above approach
import java.util.*;
class GFG{
static int maxindex(int[] dist, int n)
{
      // Write Code Here
      …..   …..   ….
}
static void selectKcities(int n, int weights[][], int k)
{
      // Write Code Here
      …..   …..   ….

      }


      System.out.println(dist[max]);

      // Write Code Here
      …..   …..   ….

}

// Driver Code
public static void main(String[] args)
{
      int n = 4;
```

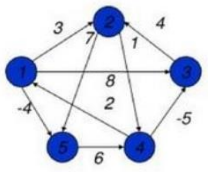
```
        int[][] weights = new int[][]{ { 0, 4, 8, 5 },  { 4, 0, 10, 7 },
                                       { 8, 10, 0, 9 },  { 5, 7, 9, 0 } };
        int k = 2;

        // Function Call
        selectKcities(n, weights, k);
}
}
```

7. Dynamic Programming

Dynamic Programming (DP) is defined as a technique that solves some particular type of problems in Polynomial Time. Dynamic Programming solutions are faster than the exponential brute method and can be easily proved their correctness. Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.
Irrevocable: Once the choice is made, it cannot be altered, i.e. if a feasible solution is selected (rejected) in step i, it cannot be rejected (selected) in subsequent stages.

7.1 All pairs Shortest Paths

The all pairs shortest path algorithm is also known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given Weighted Graph. As a result of this Algorithm , it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.



$$\begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

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.

The time complexity of this algorithm is O(V3), here V is the number of **vertices in the graph**.

Input − The cost matrix of the graph.

0 3 6 ∞ ∞ ∞ ∞

3 0 2 1 ∞ ∞ ∞

6 2 0 1 4 2 ∞

∞ 1 1 0 2 ∞ 4

∞ ∞ 4 2 0 2 1

∞ ∞ 2 ∞ 2 0 1

∞ ∞ ∞ 4 1 1 0

Output − Matrix of all pair shortest path.

0 3 4 5 6 7 7

3 0 2 1 3 4 4

4 2 0 1 3 2 3

5 1 1 0 2 3 3

6 3 3 2 0 2 1

7 4 2 3 2 0 1

7 4 3 3 1 1 0

Algorithm

floydWarshal(cost)

Input − The cost matrix of given Graph.

Output − Matrix to for shortest path between any vertex to any vertex.

```
Begin
  for k := 0 to n, do
    for i := 0 to n, do
      for j := 0 to n, do
        if cost[i,k] + cost[k,j] < cost[i,j], then
          cost[i,j] := cost[i,k] + cost[k,j]
        done
      done
    done
    display the current cost matrix
End
```

```
// Java program for Floyd Warshall All Pairs Shortest Path algorithm.
import java.io.*;
import java.lang.*;
import java.util.*;
class AllPairShortestPath {
      final static int INF = 99999, V = 4;
      void floydWarshall(int dist[][])
      {
            // Write Code Here
            …..   ……    …..
            printSolution(dist);
      }

      void printSolution(int dist[][])
      {
            System.out.println(
                  "The following matrix shows the shortest "
```

```
                + "distances between every pair of vertices");
        for (int i = 0; i < V; ++i) {
            for (int j = 0; j < V; ++j) {
                if (dist[i][j] == INF)
                    System.out.print("INF ");
                else
                    System.out.print(dist[i][j] + " ");
            }
            System.out.println();
        }
    }

    // Driver's code
    public static void main(String[] args)
    {
        // Provide Input
            ….          …..   ….
        AllPairShortestPath a = new AllPairShortestPath();
        a.floydWarshall(graph);
    }
}
```

7.2 Optimal Binary Search Tree

An Optimal Binary Search Tree (OBST), also known as a Weighted Binary Search Tree, is a binary search tree that minimizes the expected search cost. In a binary search tree, the search cost is the number of comparisons required to search for a given key.

In an OBST, each node is assigned a weight that represents the probability of the key being searched for. The sum of all the weights in the tree is 1.0. The expected search cost of a node is the sum of the product of its depth and weight, and the expected search cost of its children.

To construct an OBST, we start with a sorted list of keys and their probabilities. We then build a table that contains the expected search cost for all possible sub-trees of the original list. We can use dynamic programming to fill in this table efficiently. Finally, we use this table to construct the OBST.

The time complexity of constructing an OBST is O(n^3), where n is the number of keys. However, with some optimizations, we can reduce the time complexity to O(n^2). Once the OBST is constructed, the time complexity of searching for a key is O(log n), the same as for a regular binary search tree.

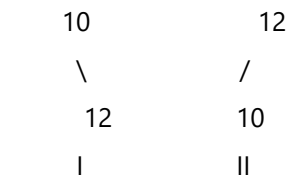
The OBST is a useful data structure in applications where the keys have different probabilities of being searched for. It can be used to improve the efficiency of searching and retrieval operations in databases, compilers, and other computer programs.

Given a sorted array key *[0.. n-1]* of search keys and an array *freq[0.. n-1]* of frequency counts,
where *freq[i]* is the number of searches for *keys[i]*. Construct a binary search tree of all keys such that the total cost of all the searches is as small as possible.
Let us first define the cost of a BST. The cost of a BST node is the level of that node multiplied by its frequency. The level of the root is 1.

Examples:

Input: keys[] = {10, 12}, freq[] = {34, 50}

There can be following two possible BSTs



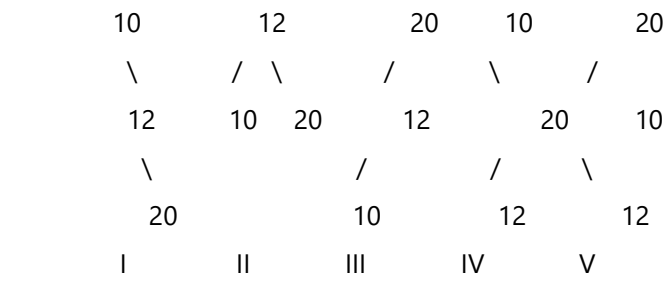
Frequency of searches of 10 and 12 are 34 and 50 respectively.

The cost of tree I is 34*1 + 50*2 = 134

The cost of tree II is 50*1 + 34*2 = 118

Input: keys[] = {10, 12, 20}, freq[] = {34, 8, 50}

There can be following possible BSTs



Among all possible BSTs, cost of the fifth BST is minimum.

Cost of the fifth BST is 1*50 + 2*34 + 3*8 = 142

We can use the recursive solution with a dynamic programming approach to have a more optimized code, reducing the complexity from O(n^3) from the pure dynamic programming to O(n). To do that, we have to store the subproblems calculations in a matrix of NxN and use that in the recursions, avoiding calculating all over again for every recursive call.

```
// Dynamic Programming Java code for Optimal Binary Search Tree Problem
public class Optimal_BST {

      /* A Dynamic Programming based function that calculates
            minimum cost of a Binary Search Tree. */
      static int optimalSearchTree(int keys[], int freq[], int n) {

      /* Create an auxiliary 2D matrix to store results of subproblems */
            int cost[][] = new int[n + 1][n + 1];
             //  Write Code Here
            …..   ….    …}

// A utility function to get sum of array elements freq[i] to freq[j]
      static int sum(int freq[], int i, int j) {
            int s = 0;]
            //  Write Code Here
```

```
                ….. ….     …  }
      public static void main(String[] args) {

            int keys[] = { 10, 12, 20 };
            int freq[] = { 34, 8, 50 };
            int n = keys.length;
            System.out.println("Cost of Optimal BST is "
                        + optimalSearchTree(keys, freq, n));
      }

}
```

7.3 0/1 Knapsack Problem

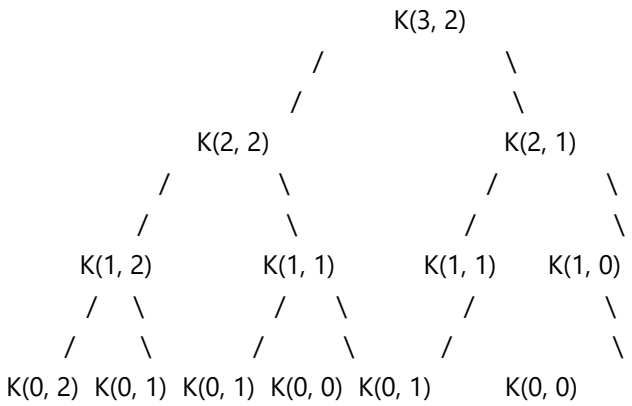
Given **N** items where each item has some weight and profit associated with it and also given a bag with capacity **W**, [i.e., the bag can hold at most **W** weight in it]. The task is to put the items into the bag such that the sum of profits associated with them is the maximum possible. Stop when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

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].

Memoization Approach for 0/1 Knapsack Problem:

Note: It should be noted that the above function using recursion computes the same subproblems again and again. See the following recursion tree, K(1, 1) is being evaluated twice.

In the following recursion tree, **K()** refers to knapSack(). The two parameters indicated in the following recursion tree are n and W.
The recursion tree is for following sample inputs.
weight[] = {1, 1, 1}, W = 2, profit[] = {10, 20, 30}



Recursion tree for Knapsack capacity 2 units and 3 items of 1 unit weight.

As there are repetitions of the same subproblem again and again we can implement the following idea to solve the problem.

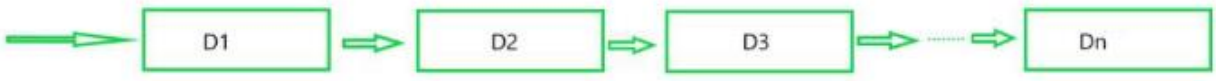
If we get a subproblem the first time, we can solve this problem by creating a 2-D array that can store a particular state (n, w). Now if we come across the same state (n, w) again instead of calculating it in exponential complexity we can directly return its result stored in the table in constant time.

```
// Dynamic Programming Java code for Optimal Binary Search Tree Problem
public class Optimal_BST {
      static int optimalSearchTree(int keys[], int freq[], int n) {
            int cost[][] = new int[n + 1][n + 1];
                  // Write Code Here
            ….    …..   ….

      }
      static int sum(int freq[], int i, int j) {
            int s = 0;
            for (int k = i; k <= j; k++) {
                  if (k >= freq.length)
                        continue;
                  s += freq[k];
            }
            return s;
      }

      public static void main(String[] args) {
            int keys[] = { 10, 12, 20 };
            int freq[] = { 34, 8, 50 };
            int n = keys.length;
            System.out.println("Cost of Optimal BST is "
                        + optimalSearchTree(keys, freq, n));
      }

}
```

7.4 Reliability Design

The reliability design problem is the designing of a system composed of several devices connected in series or parallel. **Reliability** means the probability to get the success of the device.

Let's say, we have to set up a system consisting of $\mathbf{D_1, D_2, D_3, \ldots\ldots\ldots, and\ D_n}$ devices, each device has some costs $\mathbf{C_1, C_2, C_3, \ldots\ldots\ldots, C_n}$. Each device has a reliability of **0.9** then the entire system has reliability which is equal to the **product** of the reliabilities of all devices i.e., $\mathbf{\pi r_i = (0.9)^4}$.



Serial Combination of one copy of each device

It means that **35%** of the system has a chance to fail, due to the failure of any one device. the problem is that we want to construct a system whose reliability is maximum. How it can be done so? we can think

that we can take more than one copy of each device so that if one device fails we can use the copy of that device, which means we can connect the devices **parallel**.

When the same type of **3** devices is connected parallelly in **stage 1** having a reliability 0.9 each then:

```
public class SeriesSystemReliability {
    public static void main(String[] args) {
        double[] componentReliabilities = {0.99, 0.98, 0.97, 0.96, 0.95};
        double systemReliability =
calculateSeriesSystemReliability(componentReliabilities);
        System.out.println("The reliability of the series system is: " +
systemReliability);
    }
    public static double calculateSeriesSystemReliability(double[]
reliabilities) {
        double systemReliability = 1.0; // Start with a system that's
initially 100% reliable
        for (double reliability : reliabilities) {
            systemReliability *= reliability;
        }
        return systemReliability;
    }
}
```

7.5 Flow Shop Scheduling

Flow shop scheduling problem:

- In flow shop, m different machines should process n jobs. Each job contains exactly n operations. The i^{th} operation of the job must be executed on the i^{th} machine. Operations within one job must be performed in the specified order.
- The first operation gets executed on the first machine, then the second operation on the second machine, and so on. Jobs can be executed in any order. The problem is to determine the optimal such arrangement, i.e. the one with the shortest possible total job execution makespan.
- With two machines, problem can be solved in O(nlogn) time using Johnson's algorithm. For more than 2 machines, the problem is NP hard. The goal is to minimize the sum of completion time of all jobs.
- The flow shop contains n jobs simultaneously available at time zero and to be processed by two machines arranged in series with unlimited storage in between them. The processing time of all jobs are known with certainty.
- It is required to schedule n jobs on machines so as to minimize makespan. The Johnson's rule for scheduling jobs in two machine flow shop is given below: In an optimal schedule, job i precedes job j if $\min\{p_{i1},p_{j2}\} < \min\{p_{j1},p_{i2}\}$. Whereas, p_{i1} is the processing time of job i on machine 1 and p_{i2} is the processing time of job i on machine 2. Similarly, p_{j1} and p_{j2} are processing times of job j on machine 1 and machine 2 respectively.
- We can find the optimal scheduling using Dynamic Programming.

Algorithm for Flow Shop Scheduling

Johnson's algorithm for flow shop scheduling is described below:

Algorithm JOHNSON_FLOWSHOP(T, Q)
// T is array of time of jobs, each column indicating time on machine Mi
// Q is queue of jobs

Q = Φ

for j = 1 to n **do**
  t = minimum machine time scanning in booth columns
  if t occurs in column 1 **then**
    Add Job j to the first empty slot of Q
  else
    Add Job j to last empty slot of Q
  end
  Remove processed job from consideration
end
return Q

```
public class FlowshopScheduling {
    static class Job {
        int jobID;
        int timeMachine1;
        int timeMachine2;
        public Job(int jobID, int timeMachine1, int timeMachine2) {
            this.jobID = jobID;
            this.timeMachine1 = timeMachine1;
            this.timeMachine2 = timeMachine2;
        }
    }

    public static void main(String[] args) {
        Job[] jobs = {
            new Job(1, 3, 2),
            new Job(2, 2, 1),
            new Job(3, 4, 3)
        };

        // Write Code Here
      ..    …     ..    ….    ….

        System.out.println("Minimum makespan: " + makespan);
    }
}
```

8. Dynamic Programming

8.1 Sherlock and Cost

Problem Statement and Example

In this challenge, you will be given an array B and must determine an array A. There is a special rule: For all i, A[i]<=B[i]. That is, A[i] can be any number you choose such that 1<=A[i]<=B[i]. Your task is to select a series of A[i] given B[i] such that the sum of the absolute difference of consecutive pairs of A is maximized. This will be the array's cost and will be represented by the variable S below.

The equation can be written:

$$S = \sum_{i=2}^{n} |A[i] - A[i-1]|$$

For example, if the array B=[1,2,3], we know that 1<=A[1]<=1, 1<=A[2]<=2, and 1<=A[3]<=3. Arrays meeting those guidelines are:
[1,1,1], [1,1,2], [1,1,3]
[1,2,1], [1,2,2], [1,2,3]

Our calculations for the arrays are as follows:
|1-1| + |1-1| = 0 |1-1| + |2-1| = 1 |1-1| + |3-1| = 2
|2-1| + |1-2| = 2 |2-1| + |2-2| = 1 |2-1| + |3-2| = 2
The maximum value obtained is 2.

Function Description:
Complete the cost function in the editor below. It should return the maximum value that can be obtained.
cost has the following parameter(s):
B: an array of integers

Input Format
The first line contains the integer t, the number of test cases.
Each of the next t pairs of lines is a test case where:
- The first line contains an integer n, the length of B
- The next line contains n space-separated integers B[i]
Constraints
1<=t<=20
1<n<=10^5
1<=B[i]<=100

Output Format
For each test case, print the maximum sum on a separate line.

Sample Input
1
5
10 1 10 1 10

Sample Output
36

Explanation
The maximum sum occurs when A[1]=A[3]=A[5]=10 and A[2]=A[4]=1. That is 36.

```
#include<iostream>
#include<algorithm>
#include<cstring>
using namespace std;

int b[100000];
int dp[100000][2];

int main()
{
      int t;
      cin >> t;
      while (t--)
      {
      memset(dp, 0, sizeof(dp));
      int n;
      cin >> n;
      for (int i = 0; i < n; i++)
      cin >> b[i];
      for (int i = 1; i < n; i++)
      {
       dp[i][0] = max(dp[i - 1][0], dp[i - 1][1] + abs(b[i - 1] - 1));
       dp[i][1] = max(dp[i - 1][0] + abs(b[i] - 1), dp[i - 1][1]);
      }
      cout << max(dp[n - 1][0], dp[n - 1][1]) << endl;
      }
      return 0;
}
```

8.2 Summing Pieces

Consider an array, A, of length n. We can split A into contiguous segments called pieces and store them as another array, B. For example, if A=[1,2,3], we have the following arrays of pieces:

B=[(1),(2),(3)] contains three 1-element pieces.
B=[(1,2),(3)] contains two pieces, one having 2 elements and the other having 1 element.
B=[(1),(2,3)] contains two pieces, one having 1 element and the other having 2 elements.
B=[(1,2,3)] contains one 3-element piece.
We consider the value of a piece in some array B to be
(sum of all numbers in the piece) X (length of piece), and we consider the total value of some array B to be the sum of the values for all pieces in that B. For example, the total value of B=[(1,2,4), (5,1), (2)] is (1+2+4) X3+(5+1)X2+(2)X1=35.

Given A, find the total values for all possible B's, sum them together, and print this sum modulo (10^9 + 7) on a new line.

Input Format

The first line contains a single integer, n, denoting the size of array A.
The second line contains n space-separated integers describing the respective values in A.

Output Format

Print a single integer denoting the sum of the total values for all piece arrays (B's) of A, modulo (10^9 + 7).

Sample Input 1

3
1 3 6

Sample Output 1

73

Sample Input 2

5
4 2 9 10 1

Sample Output 2

971

```
public class SummingPieces {

    static final int MOD = 1000000007;

    public static void main(String[] args) {
        int[] arr = {1, 3, 6}; // Example array
        System.out.println("Total sum is: " + summingPieces(arr));
    }

    public static long summingPieces(int[] arr) {
        int n = arr.length;
        long totalSum = 0;
        long[] pow2 = new long[n];
        pow2[0] = 1; // 2^0
      // Write Code Here
      …     …..   ….
        }

        return totalSum;
    }
}
```

8.3 Stock Maximize

Given are N ropes of different lengths, the task is to connect these
Your algorithms have become so good at predicting the market that you now know what the share price of Wooden Orange Toothpicks Inc. (WOT) will be for the next number of days.

Each day, you can either buy one share of WOT, sell any number of shares of WOT that you own, or not make any transaction at all. What is the maximum profit you can obtain with an optimum trading strategy?

Example
Prices=[1,2]
Buy one share day one, and sell it day two for a profit of 1. Return 1.
Prices[2,1]
No profit can be made so you do not buy or sell stock those days. Return 0.

Function Description
Complete the stockmax function in the editor below.
stockmax has the following parameter(s):

- prices: an array of integers that represent predicted daily stock prices

Returns

- int: the maximum profit achievable

Input Format
The first line contains the number of test cases t.
Each of the next t pairs of lines contain:
- The first line contains an integer n, the number of predicted prices for WOT.
- The next line contains n space-separated integers prices[i], each a predicted stock price for day i.

Constraints

- 1<=t<=10
- 1<=n<=50000
- 1<=prices[i]<=100000

Output Format
Output i lines, each containing the maximum profit that can be obtained for the corresponding test case.
Sample Input

STDIN	Function
3	q = 3
3	prices[] size n = 3
5 3 2	prices = [5, 3, 2]
3	prices[] size n = 3
1 2 100	prices = [1, 2, 100]
4	prices[] size n = 4
1 3 1 2	prices =[1, 3, 1, 2]

Sample Output
0
197
3
Explanation
For the first case, there is no profit because the share price never rises, return 0.
For the second case, buy one share on the first two days and sell both of them on the third day for a profit of 197.
For the third case, buy one share on day 1, sell one on day 2, buy one share on day 3, and sell one share on day 4. The overall profit is 3.

```
public class StockMaximize {

    public static void main(String[] args) {
        int[] prices = {1, 2, 100};
```

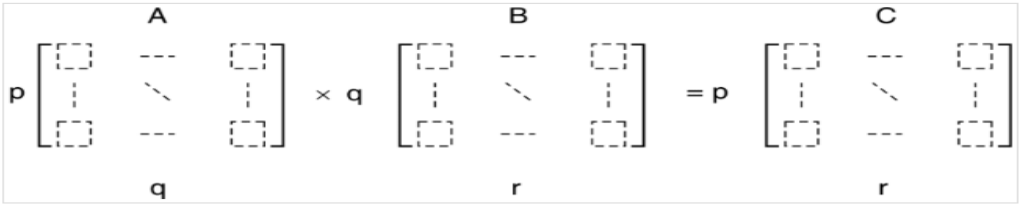
```
        System.out.println("Maximum Profit: " + maxProfit(prices));
    }

    public static long maxProfit(int[] prices) {
        // Write Code Here
            ….    ….    …..
        }

        return maxProfit;
    }
```

8.4 Matrix Chain Multiplication Problem

Problem: In what order, n matrices A_1, A_2, A_3, …. A_n should be multiplied so that it would take a minimum number of computations to derive the result.

Cost of matrix multiplication: Two matrices are called compatible only if the number of columns in the first matrix and the number of rows in the second matrix are the same. Matrix multiplication is possible only if they are compatible. Let A and B be two compatible matrices of dimensions p x q and q x r

Each element of each row of the first matrix is multiplied with corresponding elements of the appropriate column in the second matrix. The total numbers of multiplications required to multiply matrix A and B are p x q x r.



Suppose dimension of three matrices are :
A_1 = 5 x 4
A_2 = 4 x 6
A_3 = 6 x 2
Matrix multiplication is associative. So
(A_1 A_2) A_3 = {(5 x 4) x (4 x 6) } x (6 x 2)
= (5 x 4 x 6) + (5 x 6 x 2)
= 180
A_1 (A_2 A_3) = (5 x 4) x {(4 x 6) x (6 x 2) }
= (5 x 4 x 2) + (4 x 6 x 2)
= 88
The answer of both multiplication sequences would be the same, but the numbers of multiplications are different. This leads to the question, what order should be selected for a chain of matrices to minimize the number of multiplications?

Counting the number of paranthesization

Let us denote the number of alternative parenthesizations of a sequence of n matrices by p(n). When n = 1, there is only one matrix and therefore only one way to parenthesize the matrix. When n ≥ 2, a fully parenthesized matrix product is the product of two fully parenthesized matrix sub-products, and the split between the two subproducts may occur between the k and $(k + 1)^{st}$ matrices for any k = 1, 2, 3…, n – 1. Thus we obtain the recurrence.

$$P(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{k=1}^{n-1} p(k)\, p(n-k) & \text{if } n \geq 2 \end{cases}$$

The solution to the recurrence is the sequence of ***Catalan numbers***, which grows as $\Omega(4^n / n^{3/2})$, roughly equal to $\Omega(2^n)$. Thus, the numbers of solutions are exponential in n. A brute force attempt is infeasible to find the solution.

Any parenthesizations of the product $A_i A_{i+1} \ldots A_j$ must split the product between A_k and A_{k+1} for some integer k in the range $i \leq k < j$. That is for some value of k, we first compute the matrices $A_{i\ldots k}$ and $A_{k+1\ldots j}$ and then multiply them together to produce the final product $A_{i\ldots j}$ The cost of computing these parenthesizations is the cost of computing $A_{i\ldots k}$, plus the cost of computing $A_{k+1\ldots j}$ plus the cost of multiplying them together.

We can define m[i, j] recursively as follows. If i == j, the problem is trivial; the chain consists of only one matrix $A_{i\ldots i} = A$. No scalar multiplications are required. Thus m[i, i] = 0 for i = 1, 2 …n.

To compute m[i, j] when i < j, we take advantage of the structure of an optimal solution of the first step. Let us assume that the optimal parenthesizations split the product $A_i A_{i+1} \ldots A_j$ between A_k and A_{k+1}, where $i \leq k < j$. Then m[i, j] is equal to the minimum cost for computing the subproducts $A_{i\ldots k}$ and $A_{k+1\ldots j}$ plus the cost of multiplying these two matrices together.

The optimal substructure is defined as,

$$m[i, j] = \begin{cases} 0 & , \text{if } i = j \\ \min_{i \leq k < j} \{m[i, k] + m[k + 1, j] + d_{i-1} \times d_k \times d_j\} & , \text{if } i < j \end{cases}$$

Where d = $\{d_0, d_1, d_2, \ldots, d_n\}$ is the vector of matrix dimensions.

m[i, j] = Least number of multiplications required to multiply matrix sequence $A_i \ldots A_j$.

```
// Java code to implement the matrix chain multiplication using recursion
import java.io.*;
import java.util.*;
class MatrixChainMultiplication {
```

```
      static int MatrixChainOrder(int p[], int i, int j)
      {
            // Write Code Here
      …….. ……. ……..
      }
      // Driver code
      public static void main(String args[])
      {
            int arr[] = new int[] { 1, 2, 3, 4, 3 };
            int N = arr.length;

            // Function call
            System.out.println(
                  "Minimum number of multiplications is "
                  + MatrixChainOrder(arr, 1, N - 1));
      }
}
```

8.5 Bridge and Torch Problem

Problem Statement

Given an array of positive distinct integer denoting the crossing time of ‘n’ people. These ‘n’ people are standing at one side of bridge. Bridge can hold at max two people at a time. When two people cross the bridge, they must move at the slower person’s pace. Find the minimum total time in which all persons can cross the bridge.

Note: Slower person space is given by larger time.

Input: Crossing Times = {10, 20, 30}

Output: 60

Explanation

1. Firstly person '1' and '2' cross the bridge
   with total time about 20 min(maximum of 10, 20)
2. Now the person '1' will come back with total
   time of '10' minutes.
3. Lastly the person '1' and '3' cross the bridge
   with total time about 30 minutes

Hence total time incurred in whole journey will be
20 + 10 + 30 = 60

Input: Crossing Times = [1, 2, 5, 8}

Output: 15

Explanation

The approach is to use Dynamic programming. Before getting dive into dynamic programming let’s see the following observation that will be required in solving the problem.

1. When any two people cross the bridge, then the fastest person crossing time will not be contributed in answer as both of them move with slowest person speed.
2. When some of the people will cross the river and reached the right side then only the fastest people(smallest integer) will come back to the left side.
3. Person can only be present either left side or right side of the bridge. Thus, if we maintain the left mask, then right mask can easily be calculated by setting the bits '1' which is not present in the left mask. For instance, Right_mask = $((2^n) - 1)$ XOR (left_mask).
4. Any person can easily be represented by bitmask(usually called as 'mask'). When i^{th} bit of 'mask' is set, that means that person is present at left side of the bridge otherwise it would be present at right side of bridge. For instance, let the mask of **6 people** is 100101, which represents the person 1, 4, 6 are present at left side of bridge and the person 2, 3 and 5 are present at the right side of the bridge.

```
//To find minimum time required to send people on other side of bridge
import java.io.*;
class GFG {
      static int dp[][] = new int[1 << 20][2];
      public static int findMinTime(int leftmask, boolean turn, int arr[],
                                                      int n)
      {     // If all people has been transferred
            if (leftmask == 0)
                  return 0;
            // Write Code Here
            ….    …..   …..

            else {      if (Integer.bitCount(leftmask) == 1) {
                        for (int i = 0; i < n; ++i) {
                       // Write Code Here
                        ….    …..   …..
                  }
            // Write Code Here
            ….    …..   …..}

      // Utility function to find minimum time
      public static int findTime(int arr[], int n)
      {
            // Write Code Here
            ….    …..   …..   }

      // Driver Code
      public static void main(String[] args)
      {
            int arr[] = { 10, 20, 30 };
            int n = 3;
            System.out.print(findTime(arr, n));
      }
}
```

9. BackTracking

Backtracking is an algorithmic technique for solving problems recursively by trying to build a solution incrementally, one piece at a time, removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to the time elapsed till reaching any level of the search tree).

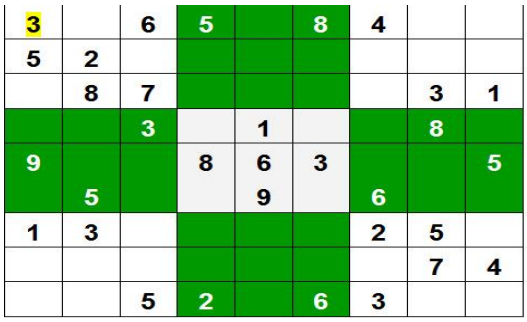
Types of Backtracking Algorithm

There are three types of problems in backtracking

1. Decision Problem: In this, we search for a feasible solution.
2. Optimization Problem: In this, we search for the best solution.
3. Enumeration Problem: In this, we find all feasible solutions.

9.1 Sudoku Puzzle

Given a partially filled 9×9 2D array 'grid[9][9]', the goal is to assign digits (from 1 to 9) to the empty cells so that every row, column, and subgrid of size 3×3 contains exactly one instance of the digits from 1 to 9.



Sudoku using Backtracking:

Like all other BackTracking problem, Sudoku can be solved by assigning numbers one by one to empty cells. Before assigning a number, check whether it is safe to assign.

Check that the same number is not present in the current row, current column and current 3X3 subgrid. After checking for safety, assign the number, and recursively check whether this assignment leads to a solution or not. If the assignment doesn't lead to a solution, then try the next number for the current empty cell. And if none of the number (1 to 9) leads to a solution, return false and print no solution exists.

Follow the steps below to solve the problem:

- Create a function that checks after assigning the current index the grid becomes unsafe or not. Keep Hashmap for a row, column and boxes. If any number has a frequency greater than 1 in the hashMap return false else return true; hashMap can be avoided by using loops.

- Create a recursive function that takes a grid.
- Check for any unassigned location.
    - If present then assigns a number from 1 to 9.
    - Check if assigning the number to current index makes the grid unsafe or not.
    - If safe then recursively call the function for all safe cases from 0 to 9.
    - If any recursive call returns true, end the loop and return true. If no recursive call returns true then return false.
- If there is no unassigned location then return true.

```
/* A Backtracking program in Java to solve Sudoku problem */
class GFG
{
public static boolean isSafe(int[][] board, int row, int col, int num)
      {
            // Row has the unique (row-clash)
            for (int d = 0; d < board.length; d++)
            {     // Write Code Here
            ….    …..   …..}
            for (int r = 0; r < board.length; r++)
            {     // Write Code Here
            ….    …..   …..
            }
            // Write Code Here
            ….    …..   …..
      }

      public static boolean solveSudoku(int[][] board, int n)
      {
            // Write Code Here
            ….    ….    …
      }

      public static void print(int[][] board, int N)
      {
            for (int r = 0; r < N; r++)
            {  for (int d = 0; d < N; d++)
                  {
                        System.out.print(board[r][d]);
                        System.out.print(" ");
                  }
                  System.out.print("\n");

                  if ((r + 1) % (int)Math.sqrt(N) == 0)
                  {
                        System.out.print("");
                  }
            }
      }
```

```
      // Driver Code
      public static void main(String args[])
      {     // Write Code Here
            …..   …..   ….
            };
            int N = board.length;

            if (solveSudoku(board, N))
            {  print(board, N);           }
            else {
                  System.out.println("No solution");
            }
```

9.2 8-Queen Problem

The eight queens problem is the problem of placing eight queens on an 8×8 chessboard such that none of them attack one another (no two are in the same row, column, or diagonal). More generally, the n queens problem places n queens on an n×n chessboard. There are different solutions for the problem.

Explanation:

- This **pseudocode uses a backtracking algorithm** to find a solution to the 8 Queen problem, which consists of placing 8 queens on a chessboard in such a way that no two queens threaten each other.
- The algorithm starts by placing a queen on the first column, then it proceeds to the next column and places a queen in the **first safe** row of that column.
- If the algorithm reaches the 8th column and al**l queens are placed in a safe position**, it prints the board and returns true.
  If the algorithm is unable to place a queen in a safe position in a certain column, it backtracks to the previous column and tries a different row.
- The "isSafe" function **checks** if it is safe to place a queen on a certain row and column by checking if there are any queens in the same row, diagonal or anti-diagonal.
- It's worth to notice that this is just a high-level **pseudocode** and it might need to be adapted depending on the specific implementation and language you are using.

```
N = 8 # (size of the chessboard)

def solveNQueens(board, col):
      if col == N:
            print(board)
            return True
      for i in range(N):
            // Write code here
            …     ….    …

def isSafe(board, row, col):
      // Write code here
            …     ….    …
      return True

board = [[0 for x in range(N)] for y in range(N)]
```

```
if not solveNQueens(board, 0):
      print("No solution found")
```

9.3 Sum of Subsets

Given a **set[]** of non-negative integers and a value **sum**, the task is to print the subset of the given set whose sum is equal to the given **sum**.

Examples:

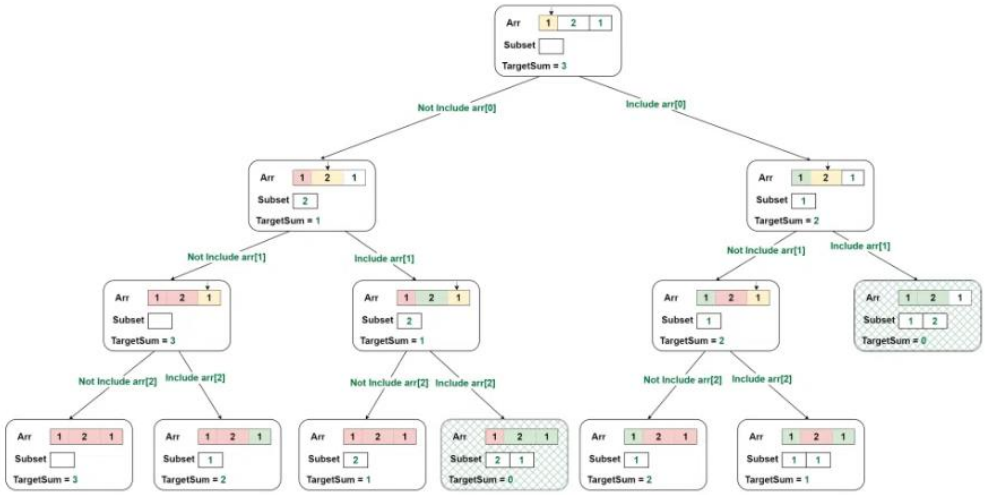
Input: set[] = {1,2,1}, sum = 3
Output: [1,2],[2,1]
Explanation: There are subsets [1,2],[2,1] with sum 3.

Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 30
Output: []
Explanation: There is no subset that add up to 30.

ubset sum can also be thought of as a special case of the 0/1 knapsack problem. For each item, there are two possibilities:

- **Include** the current element in the subset and recur for the remaining elements with the remaining **Sum**.
- **Exclude** the current element from the subset and recur for the remaining elements.

Finally, if **Sum** becomes **0** then print the elements of current subset. The recursion's **base case** would be when **no items are left**, or the **sum** becomes **negative**, then simply return.



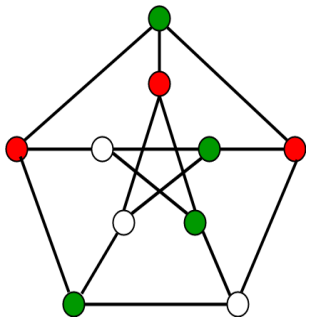
Subset Sum Problem

```
import java.util.ArrayList;
import java.util.List;
public class SubsetSum {
      static boolean flag = false;
      static void printSubsetSum(int i, int n, int[] set,int targetSum,
                                          List<Integer> subset)
      {// Write Code Here
      ….    ….    ….
            }

      // Driver code
      public static void main(String[] args)
      {
            // Test case 1
            int[] set1 = { 1, 2, 1 };
            int sum1 = 3;
            int n1 = set1.length;
            List<Integer> subset1 = new ArrayList<>();
            System.out.println("Output 1:");
            printSubsetSum(0, n1, set1, sum1, subset1);
            System.out.println();
            flag = false;

            // Test case 2
            int[] set2 = { 3, 34, 4, 12, 5, 2 };
            int sum2 = 30;
            int n2 = set2.length;
            List<Integer> subset2 = new ArrayList<>();
            System.out.println("Output 2:");
            printSubsetSum(0, n2, set2, sum2, subset2);
            if (!flag) {
                  System.out.println("There is no such subset");
            }
      }
}
```

9.4 Graph Coloring

Given an undirected graph and a number **m**, the task is to color the given graph with at most **m** colors such that no two adjacent vertices of the graph are colored with the same color

Note: Here coloring of a graph means the assignment of colors to all verticesBelow is an example of a graph that can be colored with 3 different colors:



Assign colors one by one to different vertices, starting from vertex 0. Before assigning a color, check for safety by considering already assigned colors to the adjacent vertices i.e check if the adjacent vertices have the same color or not. If there is any color assignment that does not violate the conditions, mark the color assignment as part of the solution. If no assignment of color is possible then backtrack and return false
Follow the given steps to solve the problem:

- Create a recursive function that takes the graph, current index, number of vertices, and color array.
- If the current index is equal to the number of vertices. Print the color configuration in the color array.
- Assign a color to a vertex from the range (1 to m).
- For every assigned color, check if the configuration is safe, (i.e. check if the adjacent vertices do not have the same color) and recursively call the function with the next index and number of vertices otherwise, return **false**
- If any recursive function returns **true** then return **true**
- If no recursive function returns **true** then return **false**

```
/* Java program for solution of M Coloring problem using backtracking */

public class mColoringProblem {
        final int V = 4;
        int color[];

        /* A utility function to check if the current color assignment  is safe for vertex v */
        boolean isSafe(int v, int graph[][], int color[], int c)
        {
                //  Write Code Here
        ....    ....    ...
        }
        boolean graphColoringUtil(int graph[][], int m, int color[], int v)
        {
                /* base case: If all vertices are
                //  Write Code Here
        ....    ....    ...                                     }
                }
```

```
            return false;
        }

        boolean graphColoring(int graph[][], int m)
        {//  Write Code Here
        ....    ....    ...             }

        void printSolution(int color[])
        {//  Write Code Here
        ....    ....    ...             }

        // Driver code
        public static void main(String args[])
        { //   Write Code Here to input Graph
        ....    ....    ...
            Coloring.graphColoring(graph, m);
        }
}
```

9.5 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.

If graph contains a Hamiltonian cycle, it is called **Hamiltonian graph** otherwise it is **non-Hamiltonian**.

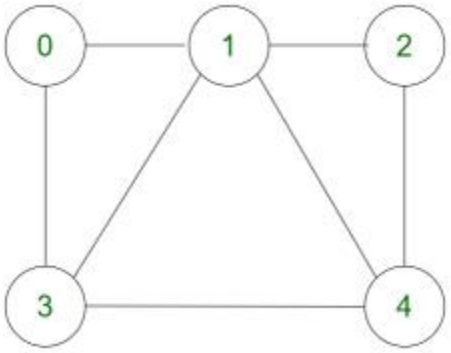
Finding a Hamiltonian Cycle in a graph is a well-known NP Complete Problem, which means that there's no known efficient algorithm to solve it for all types of graphs. However, it can be solved for small or specific types of graphs.

The Hamiltonian Cycle problem has practical applications in various fields, such as **logistics, network design, and computer science**.

What is Hamiltonian Path?

> **Hamiltonian Path** in a graph **G** is a path that visits every vertex of G exactly once and **Hamiltonian Path** doesn't have to return to the starting vertex. It's an open path.

Input: graph[][] = {{0, 1, 0, 1, 0},{1, 0, 1, 1, 1},{0, 1, 0, 0, 1},{1, 1, 0, 0, 1},{0, 1, 1, 1, 0}}



Input graph[][]

Output: {0, 1, 2, 4, 3, 0}.

Create an empty path array and add vertex **0** to it. Add other vertices, starting from the vertex **1**. Before adding a vertex, check for whether it is adjacent to the previously added vertex and not already added. If we find such a vertex, we add the vertex as part of the solution. If we do not find a vertex then we return **false**.

```
/* Java program for solution of Hamiltonian Cycle problem using backtracking */
class HamiltonianCycle
{       final int V = 5;
        int path[];
        boolean isSafe(int v, int graph[][], int path[], int pos)
        {
                //  Write Code Here
        .....   ......      .......
        }
        boolean hamCycleUtil(int graph[][], int path[], int pos)
        {       //  Write Code Here
        .....   ......      .......
                }


        /* A utility function to print solution */
        void printSolution(int path[])
        {
                System.out.println("Solution Exists: Following" + " is one Hamiltonian Cycle");
                for (int i = 0; i < V; i++)
                        System.out.print(" " + path[i] + " ");
                System.out.println(" " + path[0] + " ");
        }
        // driver program to test above function
        public static void main(String args[])
        {       //  Write Code Here
        .....   ......      .......
                hamiltonian.hamCycle(graph2);
        }
}
```

10. Backtracking

10.1 Partition of a set into K subsets with equal sum

Problem Statement and Example

Given an integer array of N elements, the task is to divide this array into K non-empty subsets such that the sum of elements in every subset is same. All elements of this array should be part of exactly one partition.

Input : arr = [2, 1, 4, 5, 6], K = 3

Output : Yes

we can divide above array into 3 parts with equal

sum as [[2, 4], [1, 5], [6]]

Input : arr = [2, 1, 5, 5, 6], K = 3

Output : No

It is not possible to divide above array into 3 parts with equal sum

Idea to Solve the Problem

We can solve this problem recursively, we keep an array for sum of each partition and a boolean array to check whether an element is already taken into some partition or not.

First we need to check some base cases,

If K is 1, then we already have our answer, complete array is only subset with same sum.

If N < K, then it is not possible to divide array into subsets with equal sum, because we can't divide the array into more than N parts.

If sum of array is not divisible by K, then it is not possible to divide the array. We will proceed only if k divides sum. Our goal reduces to divide array into K parts where sum of each part should be array_sum/K

```
// Java program to check whether an array can be partitioned into K subsets of equal sum
class GFG
{
   Static boolean isKPartitionPossibleRec(int arr[], int subsetSum[], boolean taken[],
                                          int subset, int K, int N, int curIdx, int limitIdx)
     {
         // Write Code Here
         ....        ....        ...
         }
         }
         return false;
```

```
}

// Method returns true if arr can be partitioned into K subsets with equal sum
static boolean isKPartitionPossible(int arr[], int N, int K)
{       // Write Code Here
        ....    ....    ...
}

// Driver code
public static void main(String[] args)
{
        int arr[] = {2, 1, 4, 5, 3, 3};
        int N = arr.length;
        int K = 3;

        if (isKPartitionPossible(arr, N, K))
                System.out.println("Partitions into equal sum is possible.");
        else
                System.out.println("Partitions into equal sum is not possible.");
}
}
```

10.2 Print all longest common sub-sequences in lexicographical order

Given two strings, the task is to print all the longest common sub-sequences in lexicographical order.

Examples:

Input : str1 = "abcabcaa", str2 = "acbacba"

Output: ababa

abaca

abcba

acaba

acaca

acbaa

acbca

This problem is an extension of longest common subsequence, We first find the length of LCS and store all LCS in a 2D table using Memoization (or Dynamic Programming). Then we search all characters from 'a' to 'z' (to output sorted order) in both strings. If a character is found in both strings and the current positions of the character lead to LCS, we recursively search all occurrences with current LCS length plus 1.

```
// Java program to find all LCS of two strings in  sorted order.
import java.io.*;
class GFG
{
static int MAX = 100;
static int lcslen = 0;
static int[][] dp = new int[MAX][MAX];
static int lcs(String str1, String str2, int len1, int len2, int i, int j)
{          // Write Code Here
                    ...        ....        ...
}
static void printAll(String str1, String str2, int len1, int len2,  char[] data, int indx1, int indx2,
                                                                                                    int currlcs)
{    // Write Code Here
                    ...        ....        ...
}

static void prinlAllLCSSorted(String str1, String str2)
{     // Write Code Here
                    ...        ....        ...
}

// Driver code
public static void main(String[] args)
{
          String str1 = "abcabcaa", str2 = "acbacba";
          prinlAllLCSSorted(str1, str2);
}
}
```

10.3 Print all Palindromic Partitions of a String using Bit Manipulation

Given a string, find all possible palindromic partitions of a given string.
Note that this problem is different from palindrome partitioning problem, there the task was to find the partitioning with minimum cuts in input string. Here we need to print all possible partitions.
Example:
Input: nitin
Output: n i t i n
n iti n
nitin
The idea is to consider the space between two characters in the string.

- If there **n** characters in the string, then there are **n-1** positions to put a space or say cut the string.

- Each of these positions can be given a binary number **1** (If a cut is made in this position) or **0** (If no cut is made in this position).
- This will give a total of **2n-1 partitions** and for each partition check whether this partition is palindrome or not.

```
import java.util.ArrayList;
import java.util.List;
class GFG {
	List<List<String>> ans = new ArrayList<>();
	boolean checkPalindrome(List<String> currPartition) {
		// Write Code Here
		....	...	..
		}
		return true;
	}
	void generatePartition(String s, String bitString) {
		// Write Code Here
		....	...	..
		}
	void bitManipulation(String s, String bitString) {
		// Write Code Here
		....	...	..
	}

	public static void main(String[] args) {
		GFG ob = new GFG();
		List<List<String>> ans;
		// Write Code Here
		....	...	..
}
```

10.4 Find shortest safe route in a path with landmines

Given a path in the form of a rectangular matrix having few landmines arbitrarily placed (marked as 0), calculate length of the shortest safe route possible from any cell in the first column to any cell in the last column of the matrix. We have to avoid landmines and their four adjacent cells (left, right, above and below) as they are also unsafe. We are allowed to move to only adjacent cells which are not landmines. i.e. the route cannot contains any diagonal moves.

Examples:

Input:
A 12 x 10 matrix with landmines marked as 0

[1 1 1 1 1 1 1 1 1 1]
[1 0 1 1 1 1 1 1 1 1]

[1 1 1 0 1 1 1 1 1 1]
[1 1 1 1 0 1 1 1 1 1]
[**1 1 1 1** 1 1 1 1 1 1]
[1 1 1 **1** 1 0 1 1 1 1]
[1 0 1 **1** 1 1 1 1 0 1]
[1 1 1 **1 1 1 1 1** 1 1]
[1 1 1 1 1 1 1 **1 1 1**]
[0 1 1 1 1 0 1 1 1 1]
[1 1 1 1 1 1 1 1 1 1]
[1 1 1 0 1 1 1 1 1 1]

Output:

Length of shortest safe route is 13 (Highlighted in **Bold**)

The idea is to use Backtracking. We first mark all adjacent cells of the landmines as unsafe. Then for each safe cell of first column of the matrix, we move forward in all allowed directions and recursively checks if they leads to the destination or not. If destination is found, we update the value of shortest path else if none of the above solutions work we return false from our function.

```
// Java program to find shortest safe Route in the matrix with landmines
import java.util.Arrays;
class GFG{
static final int R = 12;
static final int C = 10;
static int rowNum[] = { -1, 0, 0, 1 };
static int colNum[] = { 0, -1, 1, 0 };
static int min_dist;
// A function to check if a given cell (x, y) can be visited or not
static boolean isSafe(int[][] mat, boolean[][] visited, int x, int y)
{
          // Write Code Here
                    ....        ....        ....}
// A function to check if a given cell (x, y) is// a valid cell or not
static boolean isValid(int x, int y)
{
          // Write Code Here
                    ....        ....        ....}
static void markUnsafeCells(int[][] mat)
{            // Write Code Here
                    ....        ....        ....
}

static void findShortestPathUtil(int[][] mat, boolean[][] visited, int i, int j, int dist)
{        // Write Code Here
                    ....        ....        ....
}
```

```
// A wrapper function over findshortestPathUtil()
static void findShortestPath(int[][] mat)
{       // Write Code Here
                ....    ....    ....
        }
// Driver code
public static void main(String[] args)
{       // Input matrix with landmines shown with number 0
       // Write Code Here
                ....    ....    ....
        findShortestPath(mat);
}
}
```

10.5 Word Break Problem using Backtracking

Problem Statement

Given a valid sentence without any spaces between the words and a dictionary of valid English words, find all possible ways to break the sentence into individual dictionary words.

Example:

Consider the following dictionary

{ i, like, sam, sung, samsung, mobile, ice,
and, cream, icecream, man, go, mango}

Input: "ilikesamsungmobile"
Output: i like sam sung mobile
i like samsung mobile

Input: "ilikeicecreamandmango"
Output: i like ice cream and man go
i like ice cream and mango
i like icecream and man go

```
// print all possible partitions of a given string into dictionary words
import java.io.*;
import java.util.*;
class GFG {
// Prints all possible word breaks of given string
static void wordBreak(int n, List<String> dict, String s)
{
        String ans="";
        wordBreakUtil(n, s, dict, ans);
}
```

```
static void wordBreakUtil(int n, String s, List<String> dict, String ans)
{
        // Write code here
        ....    ....    ....
}
// main function
public static void main(String args[])
{       //Input str1 and str2
        // Write code here
        ....    ....    ....
        wordBreak(n1,dict,str1);
        System.out.println("\nSecond Test:");
        wordBreak(n2,dict,str2);
}
}
```

11. Branch and Bound

Branch and bound algorithms are used to find the optimal solution for combinatory, discrete, and general mathematical optimization problems.

A branch and bound algorithm provide an optimal solution to an NP-Hard problem by exploring the entire search space. Through the exploration of the entire search space, a branch and bound algorithm identify possible candidates for solutions step-by-step.

There are many optimization problems in computer science, many of which have a finite number of the feasible shortest path in a graph or minimum spanning tree that can be solved in polynomial time. Typically, these problems require a worst-case scenario of all possible permutations. The branch and bound algorithm create branches and bounds for the best solution.

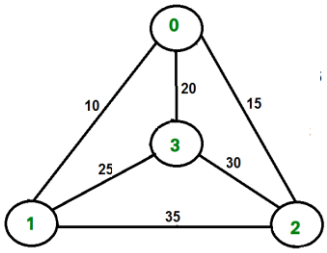
Classification of Branch and Bound Problems:

The Branch and Bound method can be classified into three types based on the order in which the state space tree is searched.

1. FIFO Branch and Bound
2. LIFO Branch and Bound
3. Least Cost-Branch and Bound

11.1 Travelling Salesperson Problem

Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible tour that visits every city exactly once and returns to the starting point.scheduling algorithm is applied to schedule the jobs on a single processor to maximize the profits.



For example, consider the graph shown in figure on right side. A TSP tour in the graph is 0-1-3-2-0. The cost of the tour is 10+25+30+15 which is 80.

Branch and Bound Solution

As seen in the previous articles, in Branch and Bound method, for current node in tree, we compute a bound on best possible solution that we can get if we down this node. If the bound on best possible solution itself is worse than current best (best computed so far), then we ignore the subtree rooted with the node.

Note that the cost through a node includes two costs.

1) Cost of reaching the node from the root (When we reach a node, we have this cost computed)

2) Cost of reaching an answer from current node to a leaf (We compute a bound on this cost to decide whether to ignore subtree with this node or not).

- In cases of a **maximization problem**, an upper bound tells us the maximum possible solution if we follow the given node.
- In cases of a **minimization problem**, a lower bound tells us the minimum possible solution if we follow the given node.

In branch and bound, the challenging part is figuring out a way to compute a bound on best possible solution. Below is an idea used to compute bounds for Travelling salesman problem.

Cost of any tour can be written as below.

Cost of a tour T = (1/2) * ? (Sum of cost of two edges adjacent to u and in the tour T)
where u ? V

For every vertex u, if we consider two edges through it in T, and sum their costs. The overall sum for all vertices would be twice of cost of tour T (We have considered every edge twice.)

(Sum of two tour edges adjacent to u) >= (sum of minimum weight two edges adjacent to u)

Cost of any tour >= 1/2) * ? (Sum of cost of two minimum weight edges adjacent to u)
where u ? V

Now we have an idea about computation of lower bound. Let us see how to how to apply it state space search tree. We start enumerating all possible nodes (preferably in lexicographical order)

1. The Root Node: Without loss of generality, we assume we start at vertex "0" for which the lower bound has been calculated above.

Dealing with Level 2: The next level enumerates all possible vertices we can go to (keeping in mind that in any path a vertex has to occur only once) which are, 1, 2, 3… n (Note that the graph is complete). Consider we are calculating for vertex 1, Since we moved from 0 to 1, our tour has now included the edge 0-1. This allows us to make necessary changes in the lower bound of the root.

Lower Bound for vertex 1 =
Old lower bound - ((minimum edge cost of 0 + minimum edge cost of 1) / 2)
+ (edge cost 0-1)
How does it work? To include edge 0-1, we add the edge cost of 0-1, and subtract an edge weight such that the lower bound remains as tight as possible which would be the sum of the minimum edges of 0 and 1 divided by 2. Clearly, the edge subtracted can't be smaller than this.
Dealing with other levels: As we move on to the next level, we again enumerate all possible vertices. For the above case going further after 1, we check out for 2, 3, 4, ...n.
Consider lower bound for 2 as we moved from 1 to 1, we include the edge 1-2 to the tour and alter the new lower bound for this node.
Lower bound(2) = Old lower bound - ((second minimum edge cost of 1 + minimum edge cost of 2)/2)
+ edge cost 1-2)
Note: The only change in the formula is that this time we have included second minimum edge cost for 1, because the minimum edge cost has already been subtracted in previous level.

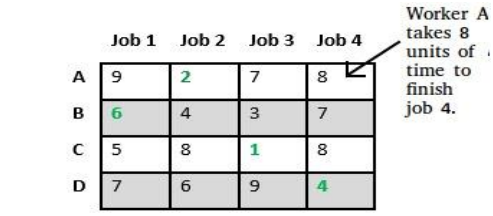
```
// Java program to solve Traveling Salesman Problem using Branch and Bound.
import java.util.*;
class GFG
{       static int N = 4;
        static int final_path[] = new int[N + 1];
        static boolean visited[] = new boolean[N];
        static int final_res = Integer.MAX_VALUE;
        static void copyToFinal(int curr_path[])
        {   // Write Code Here
                ...     ....    ....
        }
        // Function to find the minimum edge cost having an end at the vertex i
        static int firstMin(int adj[][], int i)
        {
            // Write Code Here
                ...     ....    ....}
        static int secondMin(int adj[][], int i)
        {
                // Write Code Here
                ...     ....    ....
        }
        static void TSPRec(int adj[][], int curr_bound, int curr_weight, int level, int curr_path[])
        {       // Write Code Here
                ...     ....    ....    }

        static void TSP(int adj[][])
        {
                // Write Code Here
                ...     ....    ....    }
```

```
        // Driver code
        public static void main(String[] args)
        {
                //Adjacency matrix for the given graph
                // Write Code Here
                …       ….      ….
                TSP(adj);
                System.out.printf("Minimum cost : %d\n", final_res);
                System.out.printf("Path Taken : ");
                for (int i = 0; i <= N; i++)
                {
                        System.out.printf("%d ", final_path[i]);
                }
        }
}
```

11.2 Job Assignment Problem

Problem Statement

Let there be N workers and N jobs. Any worker can be assigned to perform any job, incurring some cost that may vary depending on the work-job assignment. It is required to perform all jobs by assigning exactly one worker to each job and exactly one job to each agent in such a way that the total cost of the assignment is minimized.



An example job assignment problem. Green values show optimal job assignment that is A-Job2, B-Job1 C-Job3 and D-Job4

Finding Optimal Solution using Branch and Bound

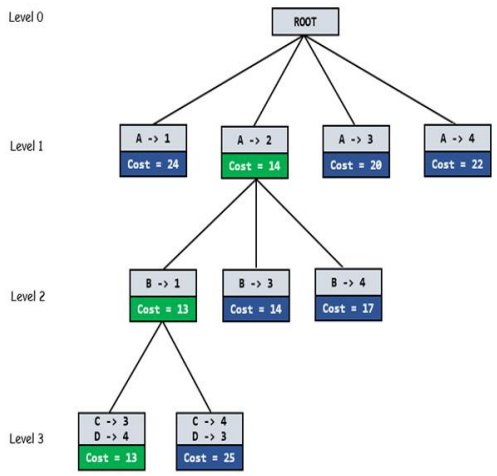
The selection rule for the next node in BFS and DFS is "blind". i.e. the selection rule does not give any preference to a node that has a very good chance of getting the search to an answer node quickly. The search for an optimal solution can often be speeded by using an "intelligent" ranking function, also called an approximate cost function to avoid searching in sub-trees that do not contain an optimal solution. It is similar to BFS-like search but with one major optimization. Instead of following FIFO order, we choose a live node with least cost. We may not get optimal solution by following node with least promising cost, but it will provide very good chance of getting the search to an answer node quickly.

There are two approaches to calculate the cost function:

1. For each worker, we choose job with minimum cost from list of unassigned jobs (take minimum entry from each row).

2. For each job, we choose a worker with lowest cost for that job from list of unassigned workers (take minimum entry from each column).

Below diagram shows complete search space diagram showing optimal solution path in green.



Complete Algorithm:

```
/* findMinCost uses Least() and Add() to maintain the list of live nodes
   Least() finds a live node with least cost, deletes  it from the list and returns it
   Add(x) calculates cost of x and adds it to the list of live nodes
   Implements list of live nodes as a min heap */
// Search Space Tree Node node
{
   int job_number;
   int worker_number;
   node parent;
   int cost;
}
// Input: Cost Matrix of Job Assignment problem Output: Optimal cost and Assignment of Jobs
algorithm findMinCost (costMatrix mat[][])
{
   // Initialize list of live nodes(min-Heap) with root of search tree i.e. a Dummy node
   while (true)
   {
      // Find a live node with least estimated cost
      E = Least();
      // The found node is deleted from the list of live nodes
      if (E is a leaf node)
      {
```

```
        printSolution();
        return;
      }

     for each child x of E
     {
        Add(x); // Add x to list of live nodes;
        x->parent = E; // Pointer for path to root
     }
   }
}
```

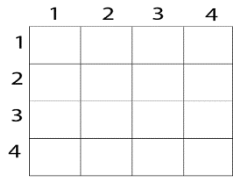
```
import java.util.*;
// Node class represents a job assignment
class Node {
    Node parent; // parent node
    int pathCost; // cost to reach this node
    int cost; // lower bound cost
    int workerID; // worker ID
    int jobID; // job ID
    boolean assigned[]; // keeps track of assigned jobs
    public Node(int N) {
        assigned = new boolean[N]; // initialize assigned jobs array
    }
}
public class Main {
    static final int N = 4; // number of workers and jobs

    // Function to create a new search tree node
    static Node newNode(int x, int y, boolean assigned[], Node parent) {
        Node node = new Node(N);
        for (int j = 0; j < N; j++)
            node.assigned[j] = assigned[j];
        if (y != -1) {
            node.assigned[y] = true;
        }
        node.parent = parent;
        node.workerID = x;
        node.jobID = y;
        return node;
    }
   static int calculateCost(int costMatrix[][], int x, int y, boolean assigned[]) {
        // Write Code Here
            ....        .....       ...   }
    // Function to print job assignment
```

```
    static void printAssignments(Node min) {
        // Write Code Here
            ....        .....        ...    }
    // Function to solve Job Assignment Problem using Branch and Bound
    static int findMinCost(int costMatrix[][]) {
// Write Code Here
            ....        .....        ...    }

    public static void main(String[] args) {
        // Write Code Here for input
            ....        .....        ...
        System.out.println("\nOptimal Cost is " + findMinCost(costMatrix));
    }
}
```

11.3 N-Queen Problem Using Branch and Bound

N-Queens Problem Statement:

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.

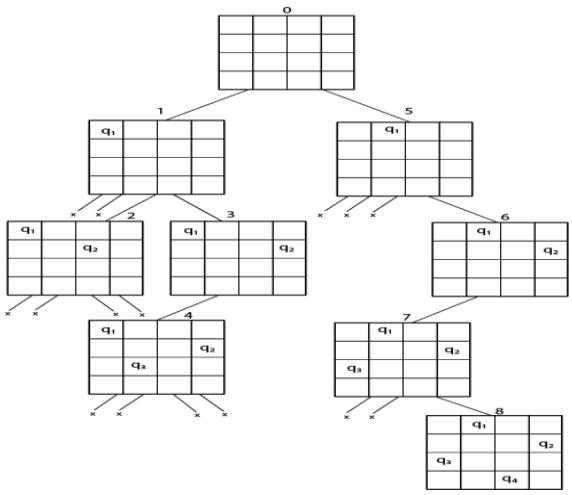


4x4 chessboard

Since, we have to place 4 queens such as q_1 q_2 q_3 and q_4 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."
Now, we place queen q_1 in the very first acceptable position (1, 1). Next, we put queen q_2 so that both these queens do not attack each other. We find that if we place q_2 in column 1 and 2, then the dead end is encountered. Thus the first acceptable position for q_2 in column 3, i.e. (2, 3) but then no position is left for placing queen 'q_3' safely. So we backtrack one step and place the queen 'q_2' in (2, 4), the next best possible solution. Then we obtain the position for placing 'q_3' which is (3, 2). But later this position also leads to a dead end, and no place is found where 'q_4' can be placed safely. Then we have to backtrack till 'q_1' and place it to (1, 2) and then all other queens are placed safely by moving q_2 to (2, 4), q_3 to (3, 1) and q_4 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.

	1	2	3	4
1			q_1	
2	q_2			
3				q_3
4		q_4		

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



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 x_1, x_2,....x_{k-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. }

```
// Java program to solve N Queen Problem  using Branch and Bound
import java.io.*;
import java.util.Arrays;
class GFG {
static int N = 8;
static void printSolution(int board[][])
{
        //  Code Here
                ...     ...     .... }

// A Optimized function to check if a queen  can be placed on board[row][col]
static boolean isSafe(int row, int col,  ....
//  Code Here
                ...     ...     ...}
// A recursive utility function to solve N Queen problem
static boolean solveNQueensUtil( .......
        { //  Code Here
                ...     ...     ... }
static boolean solveNQueens()
{
        //  Code Here
                ...     ...     ...
}
// Driver code
public static void main(String[] args)
{       solveNQueens();
}
}
```

11.4 0/1 Knapsack Problem

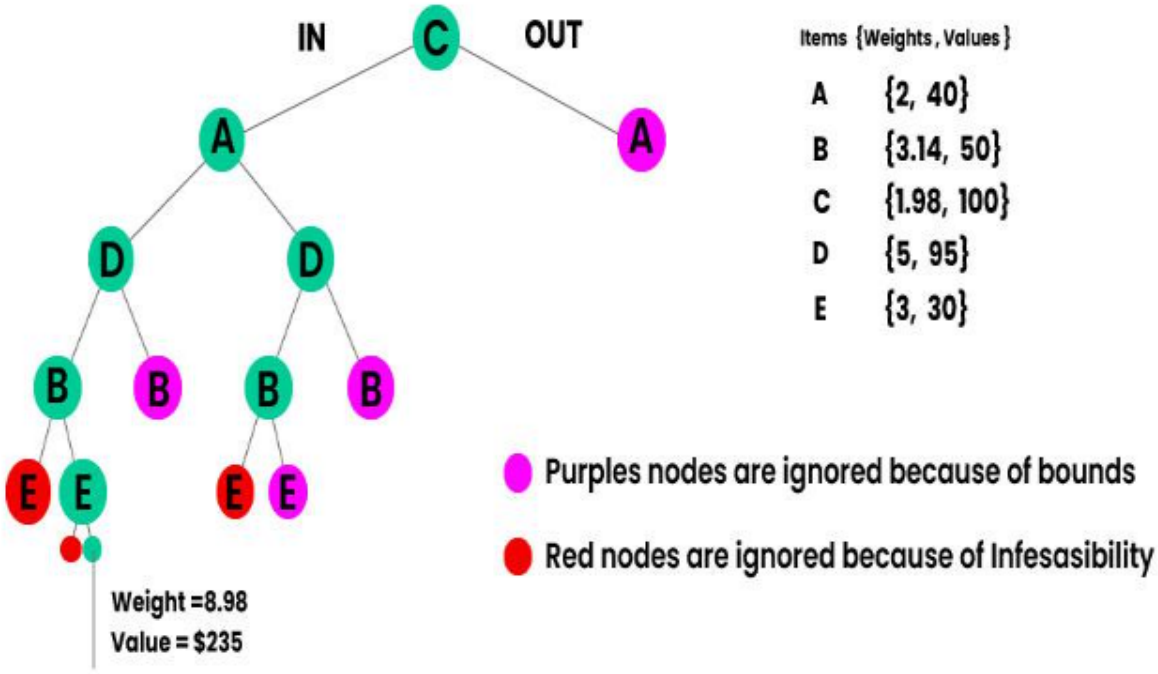
Problem Statement:

Given two arrays **v[]** and **w[]** that represent values and weights associated with n items respectively. Find out the maximum value subset(Maximum Profit) of **v[]** such that the sum of the weights of this subset is smaller than or equal to Knapsack capacity **Cap(W)**.

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.

Implementation of 0/1 Knapsack using Branch and Bound:

We will solve this problem using Branch and Bound technique.



Implementation of 0/1 Knapsack using Branch and Bound

Approach to find bound for every node for 0/1 Knapsack

The idea is to use the fact that the Greedy algorithm provides the best solution for Fractional Knapsack problem. To check if a particular node can give us a better solution or not, we compute the optimal solution (through the node) using Greedy approach. If the solution computed by Greedy approach itself is more than the best so far, then we can't get a better solution through the node.

Algorithm:

- Sort all items in decreasing order of ratio of value per unit weight so that an upper bound can be computed using Greedy Approach.
- Initialize maximum profit, maxProfit = 0
- Create an empty queue, Q.
- Create a dummy node of decision tree and enqueue it to Q. Profit and weight of dummy node are 0.
- Do following while Q is not empty.
    - Extract an item from Q. Let the extracted item be u.
    - Compute profit of next level node. If the profit is more than maxProfit, then update maxProfit.

- Compute bound of next level node. If bound is more than maxProfit, then add next level node to Q.
- Consider the case when next level node is not considered as part of solution and add a node to queue with level as next, but weight and profit without considering next level nodes.

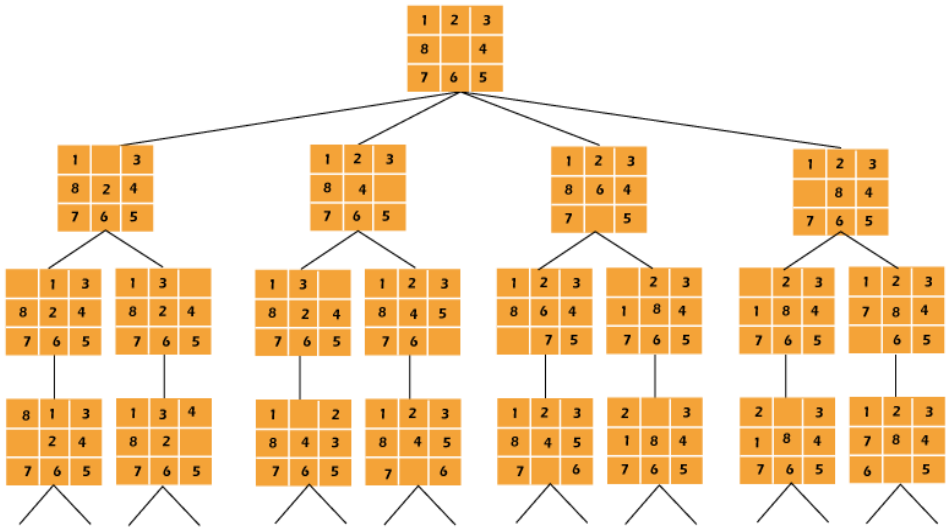
```
import java.util.Arrays;
import java.util.Comparator;
import java.util.PriorityQueue;
class Item {
	float weight;
	int value;
	Item(float weight, int value) {
		this.weight = weight;
		this.value = value;
	}
}
class Node {
	int level, profit, bound;
	float weight;

	Node(int level, int profit, float weight) {
		this.level = level;
		this.profit = profit;
		this.weight = weight;
	}
}

public class KnapsackBranchAndBound {
	// Write Code Here
	….   …..  ….   };
	static int bound(Node u, int n, int W, Item[] arr) {
		// Write Code Here
	….   …..  ….   }

	static int knapsack(int W, Item[] arr, int n) {
		// Write Code Here
	….   …..  ….   }

	public static void main(String[] args) {
		int W = 10;
		Item[] arr = {
			new Item(2, 40),
			new Item(3.14f, 50),
```

```
                new Item(1.98f, 100),
                new Item(5, 95),
                new Item(3, 30)
            };
            int n = arr.length;
            int maxProfit = knapsack(W, arr, n);
            System.out.println("Maximum possible profit = " +
maxProfit);
        }
}
```

11.5 8 Puzzle Problem

The 8 puzzle problem solution is covered in this article. A **3 by 3 board with 8 tiles** (each tile has a number from 1 to 8) and a **single empty space** is provided. The goal is to **use the vacant space to arrange the numbers on the tiles** such that they match the final arrangement. **Four neighbouring** (left, right, above, and below) **tiles** can be moved into the available area.

For instance,



By avoiding searching in sub-trees which do not include an answer node, an **"intelligent" ranking function**, also known as an **approximation costs function**, may frequently **speed up the search for an answer node**.

Basically, **Branch and Bound** involves three different kinds of nodes.

A **live node** is a created node whose children have not yet been formed.

The offspring of the **E-node** which is a live node, are now being investigated. Or to put it another way, an E-node is a node that is currently expanding.

A created node which is not to be developed or examined further is referred to as a **dead node**. A dead node has already extended all of its offspring.

Cost function:

In the search tree, each **node Y** has a corresponding cost. The next **E-node** may be found using the **cost function**. The E-node with the **lowest costs** is the next one. The function can be defined as:

C(Y) = g(Y) + h(Y) Where

g(Y) = the costs of reaching to the current node **from** the root.

h(Y) = the costs of reaching to an answer node **from** the Y.

The optimum costs function for an algorithm for 8 puzzles is :

We suppose that it will costs **one unit** to move a tile in any direction. In light of this, we create the following costs function for the 8-puzzle algorithm:

c(y) = f(y) + h(y) where

f(y) = the path's total length **from** the root y.

h(y) = the amount of the non-blank tiles which are **not in** their final goal position (misplaced tiles).

To change state y into a desired state, there are at least h(y) movements required.

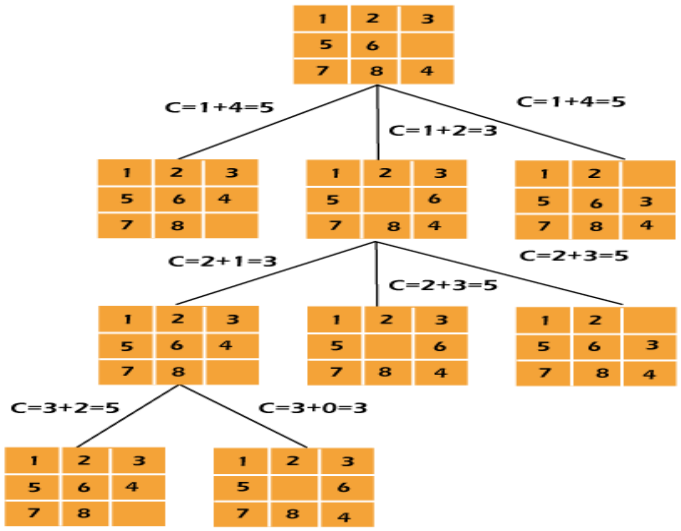
There is an algorithm for **estimating the unknown value of h(y),** which is accessible.

Final algorithm:

In order to maintain **the list of live nodes**, algorithm **LCSearch** employs the functions **Least()** and **Add().**

Least() identifies a live node with the **least c(y),** removes it from the list, and returns it.
Add(y) adds **y** to the list of live nodes.
Add(y) implements the list of live nodes as a **min-heap**.

The route taken by the aforementioned algorithm to arrive at the final configuration of the 8-Puzzle from the starting configuration supplied is shown in the **diagram below**. Keep in mind that only **nodes with the lowest costs function** value are extended.




```
// Java Program to print path from root node to destination node for N*N -1 puzzle
// algorithm using Branch and Bound The solution assumes that instance of puzzle is olvable
import java.io.*;
import java.util.*;
class GFG
{       public static int N = 3;
        public static class Node
        { // Code Here
        ...     ...     ...}
        // Function to print N x N matrix
        public static void printMatrix(int mat[][]){
                // Code Here
        ...     ...     ...}
        // Function to allocate a new node
        public static Node newNode(int mat[][], int x, int y,....
                // Code Here
        ...     ...     ... }

        public static int calculateCost(int initialMat[][], int finalMat[][])
        {
                // Code Here
        ...     ...     ...     }

        // Function to check if (x, y) is a valid matrix coordinate
        public static int isSafe(int x, int y)
        // Code Here
        ...     ...     ...}

        // Comparison object to be used to order the heap
        public static class comp implements Comparator<Node>{....
```

```
        public static void solve(int initialMat[][], int x, int y, int finalMat[][])
        {
        // Code Here
        ...     ...     ...     }

        //Driver Code
        public static void main (String[] args)
        {
                // Code Here for input
        ...     ...     ...

                solve(initialMat, x, y, finalMat);
        }
}
```

12. Branch and Bound

12.1 Generate Binary Strings of length N using Branch and Bound

Problem Statement and Example

The task is to generate a binary string of length N using branch and bound technique

Examples:

Input: N = 3

Output: 000 001 010 011 100 101 110 111

Explanation: Numbers with 3 binary digits are 0, 1, 2, 3, 4, 5, 6, 7

Input: N = 2

Output: 00 01 10 11

Approach: Generate Combinations using Branch and Bound :

- It starts with an empty solution vector .
- While Queue is not empty remove partial vector from queue.
- If it is a final vector print the combination else,
- For the next component of partial vector create k child vectors by fixing all possible states for the next component insert vectors into the queue.

```
// Java Program to generate Binary Strings using Branch and Bound
import java.io.*;
```

```
import java.util.*;
// Creating a Node class
class Node {
        int soln[];
        int level;
        ArrayList<Node> child;
        Node parent;
        Node(Node parent, int level, int N)
        {
                this.parent = parent;
                this.level = level;
                this.soln = new int[N];
        }
}

class GFG {
           // Write Code Here
        …       …       …       }

        // Driver code
        public static void main(String args[])
        {
                N = 3;
                Node root = new Node(null, 0, N);
                Q = new LinkedList<Node>();
                Q.add(root);
                while (Q.size() != 0) {
                        Node E = Q.poll();
                        generate(E);
                }
        }
}
```

12.2 Feature selection using branch and bound algorithm

A Feature selection is critical in the domains of machine learning and data analysis since it assists in identifying the most essential and informative features in a dataset. It is a procedure that seeks to extract relevant features that will help with analysis and modelling jobs. The branch and bound method is an effective feature selection tool. −

As the volume of data grows at an exponential rate, it is becoming increasingly vital to build efficient algorithms capable of quickly identifying the ideal subset of attributes. In this post, we will look at feature selection and how the branch and bound method may be used to improve the efficiency and accuracy of the feature selection process.

What is Feature Selection?

In machine learning and statistics, feature selection refers to the process of choosing a subset of relevant features that are most informative for a given task. By selecting the right features, we aim to improve the model's performance, reduce computational complexity, and mitigate the risk of overfitting.

Significance of Feature Selection

Feature selection offers several advantages in the field of data analysis and machine learning −

- **Improved Model Performance** − By selecting the most relevant features, we can enhance the accuracy and predictive power of the model. Irrelevant or redundant features can introduce noise and hinder model performance.
- **Reduced Dimensionality** − Feature selection helps in reducing the number of dimensions or attributes in the dataset. This reduction simplifies the problem space, improves computational efficiency, and facilitates better model interpretability.
- **Elimination of Overfitting** − Including irrelevant features in the model can lead to overfitting, where the model becomes too specific to the training data and fails to generalize well on unseen data. Feature selection mitigates this risk by focusing on the most informative features.
- **Faster Training and Inference** − By reducing the dimensionality of the dataset, feature selection can significantly speed up the training and inference phases of the model. This is particularly important when dealing with large-scale datasets.

What is Branch and Bound Algorithm?

The Branch and Bound algorithm is a systematic approach to finding the optimal subset of features by exploring all possible feature combinations. It utilizes a divide-and-conquer strategy coupled with intelligent pruning to efficiently search the feature space. The algorithm begins with an initial bound and progressively explores different branches to narrow down the search space until the optimal subset is found.

Algorithm

Step 1: Initialization

The Branch and Bound algorithm starts by initializing the search process. This involves setting up the initial bounds, creating a priority queue to track the best feature subsets, and defining other necessary data structures.

Step 2: Generate Initial Bounds

To guide the search process, the algorithm generates initial bounds based on the evaluation criteria. These bounds provide an estimate of the best possible solution and help in pruning unpromising branches.

Step 3: Explore Branches

The algorithm explores different branches or paths in the search tree. Each branch represents a subset of features. It evaluates the quality of each branch based on a predefined evaluation metric and decides whether to further explore or prune the branch.

Step 4: Update Bounds

As the algorithm progresses and explores different branches, it updates the bounds dynamically. This allows for more accurate pruning decisions and helps in speeding up the search process.

Step 5: Pruning and Stopping Criteria

Branch and Bound employ pruning techniques to eliminate branches that are guaranteed to be suboptimal. This reduces the search space and focuses on more promising feature subsets. The algorithm continues the search until a stopping criterion is met, such as finding the optimal subset or reaching a predefined computational limit.

Example Demonstration

Let's consider a simple example to illustrate the working of the Branch and Bound algorithm. Suppose we have a dataset with 10 features, and we want to find the optimal subset of features for a classification task. The algorithm would systematically explore different feature combinations, evaluate their performance, and prune unpromising branches until it discovers the subset with the highest evaluation metrics, such as accuracy or information gain.

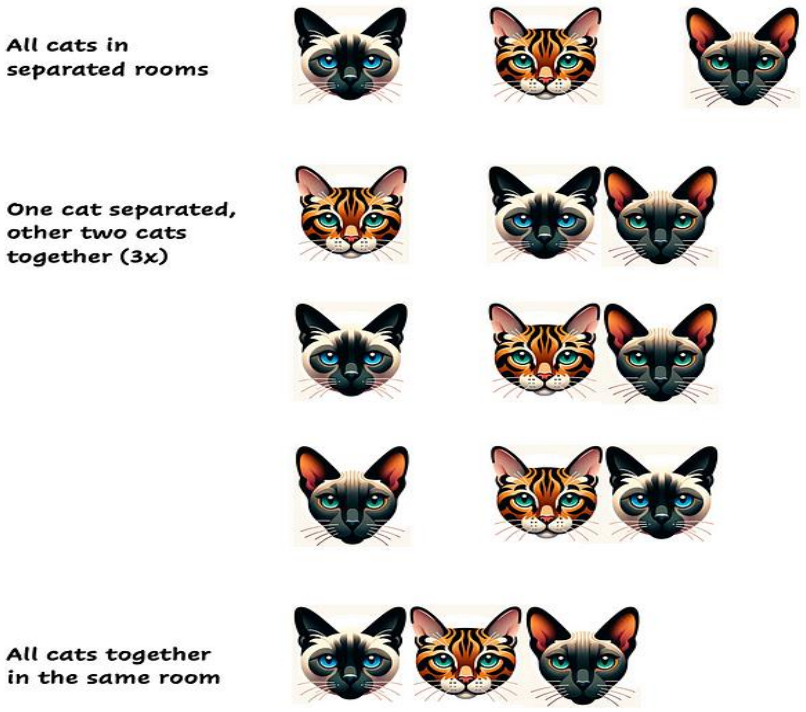
Example

Below is the program for the above example −

```
import itertools
def evaluate_subset(subset):
    return len(subset)
def branch_and_bound(features, k):
  n = len(features)
  best_subset = []
  best_score = 0.0
  def evaluate_branch(subset):
     Write Code Here
          ..................
  def backtrack(subset, idx):
Write Code Here
          ..................
return best_subset
# Example usage
if __name__ == '__main__':
    features = ['Feature A', 'Feature B', 'Feature C', 'FeatureD', 'Feature E', 'Feature F', 'Feature G', 'Feature H', 'Feature I', 'Feature J']
  k = 3  # Number of features to select
  selected_features = branch_and_bound(features, k)
  print(f"Selected Features: {selected_features}")
```

12.3 Mixed Integer Programming (MIP) using Branch and bound

Problem Statement:
Imagine you are the owner of a cat shelter. Every day, pet owners can bring their cats and you take care of them. Many people adopted a cat during COVID, but now everyone needs to be back at the office. Because of this your company is doing great.
Actually, a bit too great. You are having difficulties with placing all the cats in the rooms of your building. Sometimes you need to decline people because there are too many requests. That is why you decided to create an optimization algorithm to help you find the lowest number of rooms possible for all the cat registrations.
Let's take a look at an example. Imagine that 3 cats requested to stay at your shelter. Their names are Lily, Charlie, and Meowster. How can we divide these three cats in different rooms? We need at most three rooms, and here are all the possible solutions of grouping the cats:



Partitions of cats

Partitions and the Bell number
As you can see, there are 5 possible ways to group the 3 cats. In math, the name for one way of grouping elements of a set is a *partition*. The *Bell number* corresponds to the total number of possible partitions for a given set (in our case, with the three cats we can create 5 partitions). It's from the field of combinatorics. The recursive formula for calculating the next Bell number looks like this:

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$$

Recursive formula for calculating the Bell number. Image by author.
This number increases rapidly:

```
import org.apache.commons.math3.optim.linear.LinearObjectiveFunction;
import org.apache.commons.math3.optim.linear.Relationship;
import org.apache.commons.math3.optim.linear.LinearConstraint;
import org.apache.commons.math3.optim.linear.NoFeasibleSolutionException;
import org.apache.commons.math3.optim.linear.SimplexSolver;
import org.apache.commons.math3.optim.linear.NonNegativeConstraint;
import org.apache.commons.math3.optim.linear.UnboundedSolutionException;
import org.apache.commons.math3.optim.MaxIter;
import org.apache.commons.math3.optim.OptimizationData;
import java.util.ArrayList;
import java.util.List;
public class MIPSolver {
    public static void main(String[] args) {
        // Write Code Here
                        ....        ....        ...
        };

        try {
            SimplexSolver solver = new SimplexSolver();
            double[] solution = solver.optimize(f, constraints, optData).getPoint();
            System.out.println("Optimal solution: x1 = " + solution[0] + ", x2 = " + solution[1]);
        } catch (NoFeasibleSolutionException e) {
            System.out.println("No feasible solution found.");
        } catch (UnboundedSolutionException e) {
            System.out.println("Solution is unbounded.");
        }
    }
}
```

Solving the Cat Problem

With this knowledge, let's get back to our initial problem. Let's code branch and bound in Python. We will use breadth first search, but feel free to reuse this code to try other exploration strategies.
Some rules we will implement:

- The score of a solution will be equal to the number of rooms we use. Obviously we want to minimize this.
- Feasibility: A room can never have more than 5 cats, and the total weight of the cats in a room should not exceed 25 kilograms. Also, a room can have at most one cat with an 'angry' character, otherwise we will have cat fights…
- The lower bound of a node is calculated by the current score of the node plus the minimum number of extra rooms we need based on the number of angry cats.

Let's start with a Cat class.

```
import random
from typing import List
import numpy as np
class Cat:
    """
    A cat has a name, a weight and a character.
    """

    def __init__(self, name: str, weight: int, character: str):
        self.name = name
        self.weight = weight
        self.character = character


def generate_n_cats(n: int) -> List[Cat]:
    """
    Generate n cats with random names, weights and characters.
    """
    cats = []
    for i in range(n):
        name = f"cat_{i}"
        weight = random.randint(1, 10)
        character = np.random.choice(["sweet", "angry"], p=[0.8, 0.2])
        cats.append(Cat(name, weight, character))
    return cats
```

With the generate_n_cats function, we can generate as many cats as we like.
In the next code snippet, we will code the branch and bound algorithm:

```
import time
from typing import List
from cat_generator import Cat, generate_n_cats
class Node:
    def __init__(self, partition, remaining_cats, score):
        self.partition = partition
        self.remaining_cats = remaining_cats
        self.score = score

    def is_feasible(self):
            Write Code Here
            ..........
       return True
    def calculate_lower_bound(self):
        Write Code Here
              ..........................
        return lower_bound
```

```
    @staticmethod
    def calculate_weight(group):
        """
        Calculate the weight of a group.
        """
        return sum([cat.weight for cat in group])

    @staticmethod
    def count_angry_cats(group):
        """
        Check how many angry cats are in a group.
        """
        angry_cats = [cat for cat in group if cat.character == "angry"]
        return len(angry_cats)
class BranchAndBound:
    def __init__(self, cats: List[Cat]) -> None:
        self.cats = cats
        self.solution_type = "Branch and Bound"

    def create_solution(self, timelimit: int = 300):
            Write Code Here
            ..................
        return best_partition

    def score(self, partition) -> int:
        """
        Calculate the score of a partition.
        """
        return len(partition)
if __name__ == "__main__":
    cats = generate_n_cats(15)
    builder = BranchAndBound(cats)
    solution = builder.create_solution()
    print(f"Score of {builder.solution_type}: {builder.score(solution)}")
    print(f"Time of {builder.solution_type}: {builder.time}")
    for group in solution:
        print(group)
    print(builder.score(solution))
```

12.4 Generate Binary Strings of length N using Branch and Bound

The task is to generate a binary string of length N using branch and bound technique **Examples:**

Input: N = 3 **Output:** 000 001 010 011 100 101 110 111 **Explanation:** Numbers with 3 binary digits are 0, 1, 2, 3, 4, 5, 6, 7 **Input:** N = 2 **Output:** 00 01 10 11

Approach: Generate Combinations using Branch and Bound:

- It starts with an empty solution vector.
- While Queue is not empty remove partial vector from queue.
- If it is a final vector print the combination else,
- For the next component of partial vector create k child vectors by fixing all possible states for the next component insert vectors into the queue.

Below is the implementation of the above approach

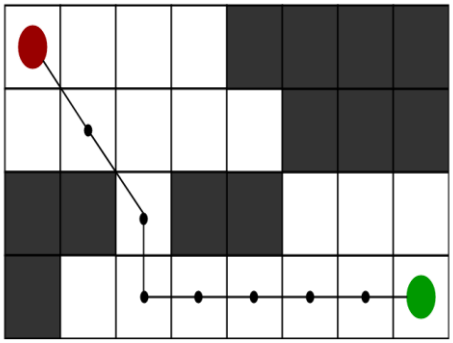
```
// Java Program to generate Binary Strings using Branch and Bound
import java.io.*;
import java.util.*;
// Creating a Node class
class Node {
	int soln[];
	int level;
	ArrayList<Node> child;
	Node parent;
	Node(Node parent, int level, int N)
	{
		this.parent = parent;
		this.level = level;
		this.soln = new int[N];
	}
}

class GFG {
	 // Write Code Here
	…	…	…	}

	// Driver code
	public static void main(String args[])
	{
		N = 3;
		Node root = new Node(null, 0, N);
		Q = new LinkedList<Node>();
		Q.add(root);
		while (Q.size() != 0) {
			Node E = Q.poll();
			generate(E);
		}
	}
}
```

12.5 A* Search Algorithm

Motivation

To approximate the shortest path in real-life situations, like- in maps, games where there can be many hindrances.

We can consider a 2D Grid having several obstacles and we start from a source cell (colored red below) to reach towards a goal cell (colored green below)



What is A* Search Algorithm?

A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.

Why A* Search Algorithm?

Informally speaking, A* Search algorithms, unlike other traversal techniques, it has "brains". What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections.

And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

Explanation

Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* Search Algorithm comes to the rescue.

What A* Search Algorithm does is that at each step it picks the node according to a value-'**f**' which is a parameter equal to the sum of two other parameters – '**g**' and '**h**'. At each step it picks the node/cell having the lowest '**f**', and process that node/cell.

We define '**g**' and '**h**' as simply as possible below

g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.

h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don't know the actual distance until we find the path, because all sorts of things can be in the

way (walls, water, etc.). There can be many ways to calculate this 'h' which are discussed in the later sections.

Algorithm

We create two lists – Open List and Closed List (just like Dijkstra Algorithm)

// A* Search Algorithm

1. Initialize the open list
2. Initialize the closed list
   put the starting node on the open
   list (you can leave its **f** at zero)
3. while the open list is not empty
   a) find the node with the least **f** on
      the open list, call it "q"
   b) pop q off the open list

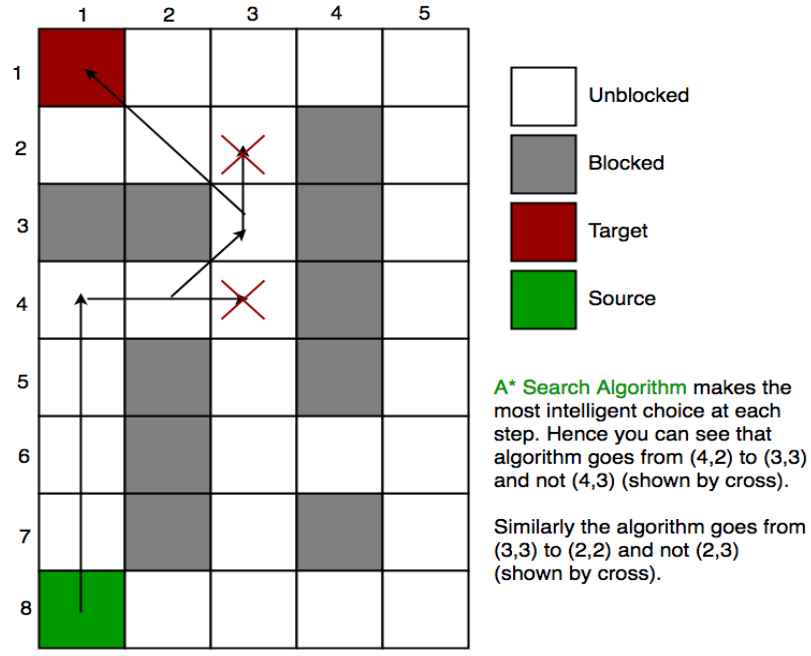
   c) generate q's 8 successors and set their
      parents to q

   d) for each successor
      i) if successor is the goal, stop search

      ii) else, compute both **g** and **h** for successor
        successor.**g** = q.**g** + distance between
                          successor and q
        successor.**h** = distance from goal to
        successor (This can be done using many
        ways, we will discuss three heuristics-
        Manhattan, Diagonal and Euclidean
        Heuristics)
        successor.**f** = successor.**g** + successor.**h**
      iii) if a node with the same position as
          successor is in the OPEN list which has a
          lower **f** than successor, skip this successor
      iV) if a node with the same position as
          successor is in the CLOSED list which has
          a lower **f** than successor, skip this successor
          otherwise, add the node to the open list
    end (for loop)

   e) push q on the closed list
   end (while loop)

So suppose as in the below figure if we want to reach the target cell from the source cell, then the A* Search algorithm would follow path as shown below. Note that the below figure is made by considering Euclidean Distance as a heuristics.



Heuristics

We can calculate **g** but how to calculate h ?

We can do things.

A) Either calculate the exact value of h (which is certainly time consuming).

OR

B) Approximate the value of h using some heuristics (less time consuming).

We will discuss both of the methods.

A) **Exact Heuristics** –

We can find exact values of h, but that is generally very time consuming.

Below are some of the methods to calculate the exact value of h.

1) Pre-compute the distance between each pair of cells before running the A* Search Algorithm.

2) If there are no blocked cells/obstacles then we can just find the exact value of h without any pre-computation using the distance formula/Euclidean Distance

B) Approximation Heuristics –

There are generally three approximation heuristics to calculate h –

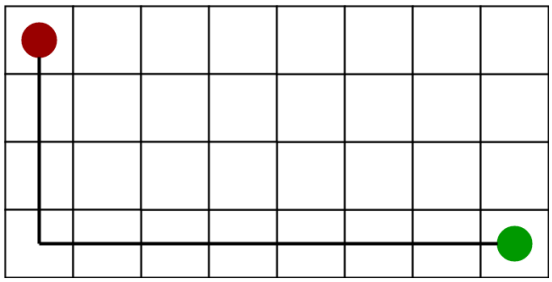
1) Manhattan Distance –

- It is nothing but the sum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

h = abs (current_cell.x – goal.x) +
abs (current_cell.y – goal.y)

- When to use this heuristic? – When we are allowed to move only in four directions only (right, left, top, bottom)

The Manhattan Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).



2) Diagonal Distance-

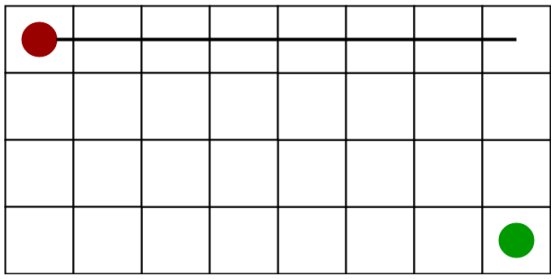
- It is nothing but the maximum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

dx = abs(current_cell.x – goal.x)
dy = abs(current_cell.y – goal.y)

h = D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
where D is length of each node(usually = 1) and D2 is diagonal distance between each node (usually = sqrt(2)).

- When to use this heuristic? – When we are allowed to move in eight directions only (similar to a move of a King in Chess)

The Diagonal Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).



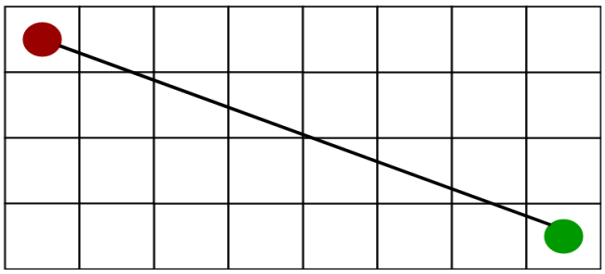
3) Euclidean Distance-

- As it is clear from its name, it is nothing but the distance between the current cell and the goal cell using the distance formula

h = sqrt ((current_cell.x – goal.x)2 +
(current_cell.y – goal.y)2)

- When to use this heuristic? – When we are allowed to move in any directions.

The Euclidean Distance Heuristics is shown by the below figure (assume red spot as source cell and green spot as target cell).



Relation (Similarity and Differences) with other algorithms-

Dijkstra is a special case of A* Search Algorithm, where h = 0 for all nodes.

Implementation

We can use any data structure to implement open list and closed list but for best performance, we use a **set** data structure of C++ STL(implemented as Red-Black Tree) and a boolean hash table for a closed list.

The implementations are similar to Dijkstra's algorithm. If we use a Fibonacci heap to implement the open list instead of a binary heap/self-balancing tree, then the performance will become better (as Fibonacci heap takes O(1) average time to insert into open list and to decrease key)

```
import java.util.*;
class Node {
   int x, y;
   int g, h;
   Node parent;
   public Node(int x, int y) {
      this.x = x;
      this.y = y;
   }
   public int getF() {
      return g + h;
   }
}

public class AStar {
   public static final int[][] GRID = {
         // Write Input Here
                        …          ….          ….    };
   public static final int[][] DIRS = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
```

```
  public static int heuristic(int x, int y, int tx, int ty) {
    return Math.abs(tx - x) + Math.abs(ty - y);
  }
  public static List<Node> astar(int[][] grid, int sx, int sy, int tx, int ty) {
    // Write code Here
         ....        ....        ..          }
    return null; // No path found
  }

  public static void main(String[] args) {
    int startX = 0, startY = 0;
    int targetX = 4, targetY = 4;

    List<Node> path = astar(GRID, startX, startY, targetX, targetY);

    if (path != null) {
      System.out.println("Path found:");
      for (int i = path.size() - 1; i >= 0; i--) {
        Node node = path.get(i);
        System.out.println("(" + node.x + ", " + node.y + ")");
      }
    } else {
      System.out.println("No path found.");
    }
  }
}
```

13. Algorithm Design Strategies

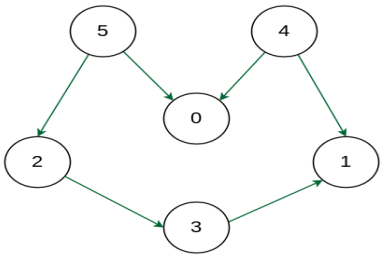
13.1 Topological Sort

Topological sorting for **Directed Acyclic Graph (DAG)** is a linear ordering of vertices such that for every directed edge u-v, vertex **u** comes before **v** in the ordering.

Note: Topological Sorting for a graph is not possible if the graph is not a **DAG**.

Example:

Input: Graph



Output: 5 4 2 3 1 0
Explanation: The first vertex in topological sorting is always a vertex with an in-degree of 0 (a vertex with no incoming edges). A topological sorting of the following graph is "5 4 2 3 1 0". There can be more than one topological sorting for a graph. Another topological sorting of the following graph is "4 5 2 3 1 0"
Approach:

- Create a **stack** to store the nodes.
- Initialize **visited** array of size **N** to keep the record of visited nodes.
- Run a loop from **0** till **N :**
- if the node is not marked **True** in **visited** array then call the recursive function for topological sort and perform the following steps:
    - Mark the current node as **True** in the **visited** array.
    - Run a loop on all the nodes which has a directed edge to the current node
    - if the node is not marked **True** in the **visited** array:
    - Recursively call the topological sort function on the node
    - Push the current node in the stack.
- Print all the elements in the stack.

```
public class TopologicalSort {

    // Function to perform DFS and topological sorting
    static void
    topologicalSortUtil(int v, List<List<Integer> > adj,
                        boolean[] visited,
                        Stack<Integer> stack)
    {
        // Mark the current node as visited
        visited[v] = true;

        // Write code here
        ……
    }

    // Function to perform Topological Sort
    static void topologicalSort(List<List<Integer> > adj,
                                int V)
    {
        // Write code here
        ……

        // Print contents of stack
        System.out.print(
            "Topological sorting of the graph: ");
        while (!stack.empty()) {
```

```
            System.out.print(stack.pop() + " ");
        }
    }

    // Driver code
    public static void main(String[] args)
    {
        // Number of nodes
        int V = 4;

        // Edges
        List<List<Integer> > edges = new ArrayList<>();
        edges.add(Arrays.asList(0, 1));
        edges.add(Arrays.asList(1, 2));
        edges.add(Arrays.asList(3, 1));
        edges.add(Arrays.asList(3, 2));

        // Graph represented as an adjacency list
        List<List<Integer> > adj = new ArrayList<>(V);
        for (int i = 0; i < V; i++) {
            adj.add(new ArrayList<>());
        }

        for (List<Integer> i : edges) {
            adj.get(i.get(0)).add(i.get(1));
        }

        topologicalSort(adj, V);
    }
}
```

13.2 Egg Dropping Puzzle

Given n cities and distances between every pair of cities, select k cities to place warehouses (or ATMs or
The following is a description of the instance of this famous puzzle involving N = 2 eggs and a building with K = 36 floors.
Suppose that we wish to know which stories in a 36-story building are safe to drop eggs from, and which will cause the eggs to break on landing. We make a few assumptions:

- An egg that survives a fall can be used again.
- A broken egg must be discarded.
- The effect of a fall is the same for all eggs.
- If an egg breaks when dropped, then it would break if dropped from a higher floor.
- If an egg survives a fall then it would survive a shorter fall.
- It is not ruled out that the first-floor windows break eggs, nor is it ruled out that the 36th-floor does not cause an egg to break.

If only one egg is available and we wish to be sure of obtaining the right result, the experiment can be carried out in only one way. Drop the egg from the first-floor window; if it survives, drop it from the

second-floor window. Continue upward until it breaks. In the worst case, this method may require 36 droppings. Suppose 2 eggs are available. What is the least number of egg droppings that are guaranteed to work in all cases?
The problem is not actually to find the critical floor, but merely to decide floors from which eggs should be dropped so that the total number of trials is minimized.
The solution is to try dropping an egg from every floor(from 1 to K) and recursively calculate the minimum number of droppings needed in the worst case. The floor which gives the minimum value in the worst case is going to be part of the solution.
In the following solutions, we return the minimum number of trials in the worst case; these solutions can be easily modified to print the floor numbers of every trial also.

What is worst case scenario?

Worst case scenario gives the user the surety of the threshold floor. For example- If we have '1' egg and 'K' floors, we will start dropping the egg from the first floor till the egg breaks suppose on the 'Kth' floor so the number of tries to give us surety is 'K'.
To solve the problem follow the below idea:
Since the same subproblems are called again, this problem has the Overlapping Subproblems property. So Egg Dropping Puzzle has both properties of a dynamic programming problem. Like other typical DP problem, re-computations of the same subproblems can be avoided by constructing a temporary array eggFloor[][] in a bottom-up manner.
In this approach, we work on the same idea as described above neglecting the case of calculating the answers to sub-problems again and again. The approach will be to make a table that will store the results of sub-problems so that to solve a sub-problem, would only require a look-up from the table which will take constant time, which earlier took exponential time.
Formally for filling DP[i][j] state where 'i' is the number of eggs and 'j' is the number of floors:
We have to traverse for each floor 'x' from '1' to 'j' and find a minimum of:
(1 + max(DP[i-1][j-1], DP[i][j-x])).
Below is the illustration of the above approach:
i => Number of eggs
j => Number of floors
Look up find maximum
Lets fill the table for the following case:
Floors = '4'
Eggs = '2'
1 2 3 4
1 2 3 4 => 1
1 2 2 3 => 2
For 'egg-1' each case is the base case so the
number of attempts is equal to floor number.
For 'egg-2' it will take '1' attempt for 1st
floor which is base case.
For floor-2 =>
Taking 1st floor 1 + max(0, DP[1][1])

Taking 2nd floor 1 + max(DP[1][1], 0)
DP[2][2] = min(1 + max(0, DP[1][1]), 1 + max(DP[1][1], 0))
For floor-3 =>
Taking 1st floor 1 + max(0, DP[2][2])
Taking 2nd floor 1 + max(DP[1][1], DP[2][1])
Taking 3rd floor 1 + max(0, DP[2][2])
DP[2][3]= min('all three floors') = 2
For floor-4 =>
Taking 1st floor 1 + max(0, DP[2][3])
Taking 2nd floor 1 + max(DP[1][1], DP[2][2])
Taking 3rd floor 1 + max(DP[1][2], DP[2][1])
Taking 4th floor 1 + max(0, DP[2][3])
DP[2][4]= min('all four floors') = 3

```
import java.io.*;

class EggDrop {

    // A utility function to get maximum of two integers
    static int max(int a, int b) { return (a > b) ? a : b; }

    /* Function to get minimum number of trials needed in worst
    case with n eggs and k floors */
    static int eggDrop(int n, int k)
    {
            //Write Code Here
      …     ….    ….    ….
    }

    /* Driver code */
    public static void main(String args[])
    {
        int n = 2, k = 36;
        System.out.println(
            "Minimum number of trials in worst"
            + " case with " + n + "  eggs and " + k
            + " floors is " + eggDrop(n, k));
    }
}
```

13.3 Policemen Catch Thieves

Given an array of size n that has the following specifications:

1. Each element in the array contains either a policeman or a thief.
2. Each policeman can catch only one thief.
3. A policeman cannot catch a thief who is more than K units away from the policeman.

We need to find the maximum number of thieves that can be caught.

Algorithm:

1. Initialize the current lowest indices of policeman in pol and thief in thi variable as -1.
2 Find the lowest index of policeman and thief.
3 If lowest index of either policeman or thief remain -1 then return 0.
4 If |pol – thi| <=k then make an allotment and find the next policeman and thief.
5 Else increment the min(pol , thi) to the next policeman or thief found.
6 Repeat the above two steps until we can find the next policeman and thief.
7 Return the number of allotments made.

```
// Java program to find maximum number of thieves caught
import java.io.*;
import java.util.*;
class GFG {
        // Returns maximum number of thieves that can be caught.
        static int policeThief(char arr[], int n, int k)
        {
                // Code Here
        .....   .....   ....                }

        // return 0 if no police OR no thief found
                if (thi == -1 || pol == -1)
                        return 0;
        // loop to increase res if distance between police and thief <= k
                while (pol < n && thi < n) {
                        // Write Code Here
                .....   ......  .....}
        }

        // Driver code starts
        public static void main(String[] args)
        {       char arr1[] = { 'P', 'T', 'T', 'P', 'T' };
                int n = arr1.length;
                int k = 2;
                System.out.println("Maximum thieves caught: " + policeThief(arr1, n, k));

                char arr2[] = { 'T', 'T', 'P', 'P', 'T', 'P' };
                n = arr2.length;
                k = 2;
```

```
                System.out.println("Maximum thieves caught: " + policeThief(arr2, n, k));

                char arr3[] = { 'P', 'T', 'P', 'T', 'T', 'P' };
                n = arr3.length;
                k = 3;
                System.out.println("Maximum thieves caught: " + policeThief(arr3, n, k));
        }
}
```

13.4 Fitting Shelves Problem

Given length of wall w and shelves of two lengths m and n, find the number of each type of shelf to be used and the remaining empty space in the optimal solution so that the empty space is minimum. The larger of the two shelves is cheaper so it is preferred. However cost is secondary and first priority is to minimize empty space on wall.
A simple and efficient approach will be to try all possible combinations of shelves that fit within the length of the wall.
To implement this approach along with the constraint that larger shelf costs less than the smaller one, starting from 0, we increase no of larger type shelves till they can be fit. For each case we calculate the empty space and finally store that value which minimizes the empty space. if empty space is same in two cases we prefer the one with more no of larger shelves.

```
// Java program to count all rotation divisible by 4.
public class GFG {
        static void minSpacePreferLarge(int wall, int m, int n)
        {
                // For simplicity, Assuming m is always smaller than n initializing output variables
                int num_m = 0, num_n = 0, min_empty = wall;
                // p and q are no of shelves of length m and n rem is the empty space
                int p = wall/m, q = 0, rem=wall%m;
                num_m=p;
                num_n=q;
                min_empty=rem;
                while (wall >= n) {
                        // Code Here
                ...     .....     ....
                        }
                System.out.println(num_m + " " + num_n + " " + min_empty);
        }

        // Driver Code
        public static void main(String[] args)
        {
```

```
                int wall = 24, m = 3, n = 5;
                minSpacePreferLarge(wall, m, n);

                wall = 24;
                m = 4;
                n = 7;
                minSpacePreferLarge(wall, m, n);
        }
}
```

14. Final Notes

The only way to learn Design and Analyzing algorithm is writing algorithms for solving real time applications and challenging problems optimally through programming. The problems in this tutorial are certainly NOT challenging. There are tens of thousands of challenging problems available – used in training for various programming contests (such as International Collegiate Programming Contest (ICPC), International Olympiad in Informatics (IOI)). Check out these sites:

- The ACM - ICPC International collegiate programming contest (https://icpc.global/)
- The Topcoder Open (TCO) annual programming and design contest (https://www.topcoder.com/)
- Universidad de Valladolid's online judge (https://uva.onlinejudge.org/).
- Peking University's online judge (http://poj.org/).
- USA Computing Olympiad (USACO) Training Program @ http://train.usaco.org/usacogate.
- Google's coding competitions (https://codingcompetitions.withgoogle.com/codejam, https://codingcompetitions.withgoogle.com/hashcode)
- The ICFP programming contest (https://www.icfpconference.org/)
- BME International 24-hours programming contest (https://www.challenge24.org/)
- The International Obfuscated C Code Contest (https://www0.us.ioccc.org/main.html)
- Internet Problem Solving Contest (https://ipsc.ksp.sk/)
- Microsoft Imagine Cup (https://imaginecup.microsoft.com/en-us)
- Hewlett Packard Enterprise (HPE) Codewars (https://hpecodewars.org/)
- OpenChallenge (https://www.openchallenge.org/)

IV. REFERENCE BOOKS:

1. Levitin A, "Introduction to the Design and Analysis of Algorithms", Pearson Education, 2008.
2. Goodrich, M.T. R Tomassia, "Algorithm Design foundations Analysis and Internet Examples", John Wiley and Sons, 2006.
3. Base Sara, Allen Van Gelder, "Computer Algorithms Introduction to Design and Analysis", Pearson, 3rd edition, 1999.

V. WEB REFERENCE:

1. http://www.personal.kent.edu/~rmuhamma/Algorithms/algorithm.html
2. http://openclassroom.stanford.edu/MainFolder/CoursePage.php?course=IntroToAlgorithms
3. http://www.facweb.iitkgp.ernet.in/~sourav/daa.html