**Computational Algorithms**

**Tower of Hanoi (Strategy: Recursion)**

It’s a game of Rods and discs in which certain number of discs of different sizes are to be transferred from one rod to other rod.

It has 3 rods and 3 discs in ascending order (smallest on the top)

Aim: To move all the discs to the destination rod in the same order.

---- One disc should be moved at one time.

---- Larger disc should not be placed on the top of a smaller disc.

package ComputationalAlgorithms;  
  
public class TowerHanoi  
{  
 public static void hanoi(int n , char rodFrom, char rodMiddle, char rodTo) {  
 if (n == 1) {  
  
 System.*out*.println("Disc 1 moved from : "+rodFrom+ " To "+ rodTo);  
 return;  
 }  
 //A to B  
 *hanoi*(n-1,rodFrom,rodTo,rodMiddle);  
 //A to C  
 System.*out*.println("Disc "+n+" moved from : "+rodFrom+ " To "+ rodTo);  
 // B to C  
 *hanoi*(n-1,rodMiddle, rodFrom, rodTo);  
 }  
 public static void main(String args[])  
 {  
 *hanoi*(3,'A','B','C');  
 }  
}

**Travelling Salesman Problem (Strategy: Dynamic Programming)**

The salesman need to travel to a number of cities and back to the original city from where he started, he must visit all the cities and he can start from any of the cities. We have to minimize the total distance.

A graph which is an adjacency matrix of distances.

Suppose

Distance[][];

So, if distance[i][j] ==0 that means the cities are not connected.

The salesman needs to complete a cycle that is a Hamiltonian cycle.

Example:

We have to use recursion and create a recursion tree.

package ComputationalAlgorithms;  
  
public class TravellingSalesman {  
 static int *n* = 4;  
 static int[][] *distance* = {  
 {0, 20, 42, 25},{20, 0, 30, 34}, {42, 30, 0, 10}, {25, 34, 10, 0}  
 };  
 static int *completed\_visit* = (1 << *n*) - 1;  
 static int[][] *DP* = new int[32][8];  
 static int TSP(int mark, int position) {  
 System.*out*.println("mark: " + mark + ", position: " + position);  
 if (mark == *completed\_visit*) {  
 System.*out*.println("Returning distance from " + position + " to 0: " + *distance*[position][0]);  
 return *distance*[position][0];  
 }  
 if (*DP*[mark][position]!= -1) {  
 System.*out*.println("Returning cached value: " + *DP*[mark][position]);  
 return *DP*[mark][position];  
 }  
 int answer = Integer.*MAX\_VALUE*;  
 System.*out*.println("Calculating minimum distance for mark " + mark + " and position " + position);  
 for (int city = 0; city < *n*; city++) {  
 if ((mark & (1 << city)) == 0) {  
 System.*out*.println("Visiting city " + city);  
 int newAnswer = *distance*[position][city] + *TSP*(mark | (1 << city), city);  
 System.*out*.println("New answer: " + newAnswer);  
 answer = Math.*min*(answer, newAnswer);  
 }  
 }  
 System.*out*.println("Minimum distance for mark " + mark + " and position " + position + ": " + answer);  
 *DP*[mark][position] = answer;  
 return answer;  
 }  
 public static void main(String[] args) {  
 for (int i = 0; i < (1 << *n*); i++) {  
 for (int j = 0; j < *n*; j++) {  
 *DP*[i][j] = -1;  
 }  
 }  
 System.*out*.println("Minimum Distance Travelled -> " + *TSP*(1, 0));  
 }  
}

**Job Sequencing Problem (Strategy: Greedy Method)**

Algorithm

----- Sort the jobs based on their profit.

----- Choose the uncompleted job with more profit and then create an array and add the job where the index is equal to the deadline day.

----- Choose the next uncompleted job with more profit and then check the deadline if the deadline is greater than the previous one then we complete it on the last day of the deadline, add the job where the index is equal or nearest to the deadline day if the array index is empty.

----- if we don’t have empty day before the deadline that is if the array index is filled before than we should not accept the job.

--- repeat the process at the last we will have array of the jobs of maximum profit.

-------- We will be implementing Comparable interface

package ComputationalAlgorithms;  
  
import java.util.ArrayList;  
import java.util.Collections;  
import java.util.List;  
  
class Job implements Comparable<Job> {  
 char id;  
 int deadLine;  
 int profit;  
  
 public Job(char id, int deadLine, int profit) {  
 this.id = id;  
 this.deadLine = deadLine;  
 this.profit = profit;  
 }  
  
 @Override  
 public int compareTo(Job o) {  
 return o.profit - this.profit;  
 }  
}  
  
public class JobSequencing {  
 List<Job> jobs;  
  
 public JobSequencing(List<Job> jobs) {  
 this.jobs = jobs;  
 }  
  
 public void best() {  
 char todo[] = new char[5];  
 int k;  
 for (Job j : jobs) {  
 k = j.deadLine - 1;  
 while (k >= 0 && todo[k] != '\0') {  
 k--;  
 }  
  
 if (k != -1)  
 todo[k] = j.id;  
 }  
 k = 0;  
  
 while (todo[k] != '\0') {  
 for (Job jj : jobs) {  
 if (jj.id == todo[k])  
 System.*out*.println("Id: " + jj.id + " Profit: " + jj.profit + " DeadLine: " + jj.deadLine + " ");  
 }  
 k++;  
 }  
 }  
  
 public void show() {  
 System.*out*.print("Job ID : \t");  
 for (Job j : jobs)  
 System.*out*.print(j.id + "\t");  
 System.*out*.println(" ");  
 System.*out*.print("DeadLine : \t");  
 for (Job j : jobs)  
 System.*out*.print(j.deadLine + "\t");  
 System.*out*.println(" ");  
 System.*out*.print("Profit : \t");  
 for (Job j : jobs)  
 System.*out*.print(j.profit + "\t");  
 System.*out*.println(" ");  
 }  
  
 public static void main(String args[]) {  
 List<Job> l1 = new ArrayList<>();  
 l1.add(new Job('c', 2, 100));  
 l1.add(new Job('b', 3, 200));  
 l1.add(new Job('d', 3, 150));  
 l1.add(new Job('e', 2, 130));  
 l1.add(new Job('f', 4, 80));  
 JobSequencing obj = new JobSequencing(l1);  
 System.*out*.println("Displaying All the Jobs");  
 obj.show();  
  
 Collections.*sort*(l1);  
 System.*out*.println(" ");  
 System.*out*.println("Jobs To Do ");  
 obj.best();  
  
 }  
}

**Fractional Knapsack Problem (Strategy: Greedy Method)**

We have a container (knapsack) with capacity and we have items with weight and profit.

The goal is to add the items to the knapsack maximizing the profit without exceeding the container weight.

In this Fractional Knapsack we can take the fractional part of the item to maximize the value.

Algorithm

--- calculate the ratio (profit/weight) of each item.

---- sort the items in descending order.

---- If the weight of the item is less than the capacity of the container than we add the item otherwise fraction of the item (as much as possible).

Container weight =10

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Items** | 1 | 2 | 3 | 4 | 5 |
| **Weights (in kg)** | 3 | 3 | 2 | 5 | 1 |
| **Profits** | 10 | 15 | 10 | 12 | 8 |

Solution

**Step 1**

Given, n = 5

Wi = {3, 3, 2, 5, 1}

Pi = {10, 15, 10, 12, 8}

Calculate P*i*/W*i* for all the items

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Items** | 1 | 2 | 3 | 4 | 5 |
| **Weights (in kg)** | 3 | 3 | 2 | 5 | 1 |
| **Profits** | 10 | 15 | 10 | 20 | 8 |
| **P*i*/W*i*** | 3.3 | 5 | 5 | 4 | 8 |

**Step 2**

Arrange all the items in descending order based on P*i*/W*i*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Items** | 5 | 2 | 3 | 4 | 1 |
| **Weights (in kg)** | 1 | 3 | 2 | 5 | 3 |
| **Profits** | 8 | 15 | 10 | 20 | 10 |
| **P*i*/W*i*** | 8 | 5 | 5 | 4 | 3.3 |

**Step 3**

Without exceeding the knapsack capacity, insert the items in the knapsack with maximum profit.

Knapsack = {5, 2, 3}

However, the knapsack can still hold 4 kg weight, but the next item having 5 kg weight will exceed the capacity. Therefore, only 4 kg weight of the 5 kg will be added in the knapsack.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Items** | 5 | 2 | 3 | 4 | 1 |
| **Weights (in kg)** | 1 | 3 | 2 | 5 | 3 |
| **Profits** | 8 | 15 | 10 | 20 | 10 |
| **Knapsack** | 1 | 1 | 1 | 4/5 | 0 |

Hence, the knapsack holds the weights = [(1 \* 1) + (1 \* 3) + (1 \* 2) + (4/5 \* 5)] = 10, with maximum profit of [(1 \* 8) + (1 \* 15) + (1 \* 10) + (4/5 \* 20)] = 37.

package ComputationalAlgorithms;  
  
import java.io.IOException;  
import java.util.Scanner;  
  
public class FractionalKnapsack  
{  
 public static void main(String args[]) throws IOException  
 {  
 int i,j=0,max\_qty,m,n;  
 float sum=0,max;  
 Scanner sc = new Scanner(System.*in*);  
 int array[][]=new int[2][20];  
 System.*out*.println("Enter no of items");  
 n=sc.nextInt();  
  
 System.*out*.println("Enter the weights of each items");  
 for(i=0;i<n;i++)  
 array[0][i]=sc.nextInt();  
  
 System.*out*.println("Enter the values of each items");  
 for(i=0;i<n;i++)  
 array[1][i]=sc.nextInt();  
  
 System.*out*.println("Enter maximum volume of knapsack :");  
 max\_qty=sc.nextInt();  
  
 m=max\_qty;  
 while(m>=0)  
 {  
 max=0;  
 for(i=0;i<n;i++)  
 {  
 if(((float)array[1][i])/((float)array[0][i])>max)  
 {  
 max=((float)array[1][i])/((float)array[0][i]);  
 j=i;  
 }  
 }  
 if(array[0][j]>m)  
 {  
 System.*out*.println("Quantity of item number: " + (j+1) + " added is " +m);  
 sum+=m\*max;  
 m=-1;  
 }  
 else  
 {  
 System.*out*.println("Quantity of item number: " + (j+1) + " added is " + array[0][j]);  
 m-=array[0][j];  
 sum+=(float)array[1][j];  
 array[1][j]=0;  
 }  
 }  
 System.*out*.println("The total profit is " + sum);  
 sc.close();  
 }  
}

**Dynamic Programming Algorithms**

**Knapsack Problem (Problem Type: Optimization)**

Knapsack – container

We have a container (knapsack) with capacity and we have items with weight and profit.

The goal is to add the items to the knapsack maximizing the profit without exceeding the container weight.

Recursive Approach

We create 2 arrays of weight and profit of items.

Weight[]

Profit[]

N = number of items

W = weight of the container.

We add the item if total weight of the item is less than the container weight.

package DynamicProgrammingAlgorithms;  
  
public class Knapsack  
{  
 public static int maximum( int a , int b)  
 {  
 if(a>b)  
 return a;  
 else  
 return b;  
 }  
 public static int knapSack(int w, int weight[],int profit[], int n)  
 {  
 if(n ==0 || w==0)  
 return 0;  
 if(weight[n-1]>w)  
 return *knapSack*(w,weight,profit,n-1);  
 else  
 return *maximum*((profit[n-1]+ *knapSack*(w-weight[n-1],weight,profit,n-1)), *knapSack*(w,weight,profit,n-1));  
 }  
 public static void main(String args[])  
 {  
 int weight[] = { 50, 90,110,60,100};  
 int profit[] = {10,20,30, 10,40};  
 int n = 5;  
 int w = 250;  
 int total = *knapSack*(w,weight,profit,n);  
 System.*out*.println("Total profit : "+ total);  
 }  
}

**Longest Common Subsequence (Problem Type: Subsequence)**

Longest common subsequence of characters is a sequence (group of characters) that is present in two or more strings.

https://interviewing.io/questions/longest-common-subsequence

Example

Abbcssdabcdes

Abcbssadabcesd

The longest common subsequence is “dabc”.

Algorithm

1. Two dimensional Arrays where the row stores the first string characters and the column stores the 2nd string characters.
2. Each character of the first string compare to the character of other string if they are same increment the value in the array and if they are not matched than take the maximum value between the value in the array at that position and the value in the array above to the left.

package DynamicProgrammingAlgorithms;  
  
public class LongestCommonSubsequence {  
 public static int lcs(char[]x,char[]y,int m, int n )  
 {  
 int [][]l = new int[m+1][n+1];  
 for(int i=0;i<=m;i++)  
 {  
 for(int j=0;j<=n;j++)  
 {  
 if(i==0 || j==0)  
 l[i][j]=0;  
 else if(x[i-1]==y[j-1])  
 l[i][j] = l[i-1][j-1]+1;  
 else  
 l[i][j] = Math.*max*(l[i-1][j],l[i][j-1]);  
  
 }  
 }  
 return l[m][n];  
 }  
  
 public static void main(String args[])  
 {  
 String s1 = "abccba";  
 String s2 = "abddba";  
  
 char f [] = s1.toCharArray();  
 char s[] = s2.toCharArray();  
 int m = s1.length();  
 int n = s2.length();  
 int lon = *lcs*(f,s,m,n);  
 System.*out*.println("The longest subsequence is : "+lon);  
 }  
}

**Longest Increasing Subsequence (Problem Type: Subsequence)**

A subsequence of an array is a list of elements of the array where some elements are deleted (or not deleted at all) and they should be in the same order in the subsequence as in the original array.

For example, for the array: [2,3,1] , the subsequences will be [{2},{3},{1},{2,3},{2,1},{3,1},{2,3,1}} but {3,2} is not a subsequence because its elements are not in the same order as the original array.

What is the Longest Increasing Subsequence?

The longest increasing subsequence is described as a subsequence of an array where:

* All elements of the subsequence are in increasing order.
* This subsequence itself is of the longest length possible.

Steps:

* First, make a 1D array of size n.
* Next, you must iterate it for each element from index 1 to n-1.
* You will iterate the elements with indices smaller than the current element in a nested loop for each element.
* In this nested loop, if you find the element’s value is lesser than the current element, then you will assign lis[i] with (lis[j]+1) and if (lis[j]+1) is greater than lis[i].
* Finally, you will traverse the entire lis[] array to find the maximum element, which will conclude your answer.

package DynamicProgrammingAlgorithms;  
  
public class LongestIncreasingSubsequence {  
 public static int lis(int arr[])  
 {  
 int n = arr.length;  
 int lis[] = new int[n];  
 for(int i=0;i<n;i++)  
 lis[i] = 1;  
 for(int i=1;i<n;i++)  
 {  
 for(int j=0;j<i;j++)  
 {  
 if(arr[i]>arr[j] && lis[i]< lis[j] +1)  
 lis[i]= lis[j]+1;  
 }  
  
 }  
 int m=0;  
 for( int i=0;i<n;i++)  
 {  
 if (m<lis[i])  
 m= lis[i];  
 }  
 return m;  
 }  
  
 public static void main(String args[])  
 {  
 int arr[] = {10, 22, 9, 33, 21, 50, 41,60};  
 int length = *lis*(arr);  
 System.*out*.println("Length of the Longest Increasing Subsequence is : "+ length);  
 }  
}

**Backtracking Algorithms**

**The Knight's Tour Problem**

The Knight's Tour Problem is a classic problem in computer science, mathematics, and chess. The Knight's Tour Problem involves finding a series of moves that a knight on a chessboard can make in order to visit every square only once.

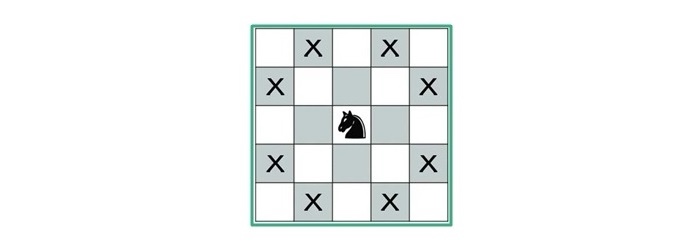
This Java program utilizes backtracking to solve this problem. The program will take an 8x8 chessboard as input and output a valid sequence of moves by the Knight that visits every square on the board exactly once.

There can be two ways in which a knight can finish its tour. In the first way, the knight moves one step and returns back to the starting position forming a loop which is called a closed tour. In the second way i.e. the open tour, it finishes anywhere in the board.

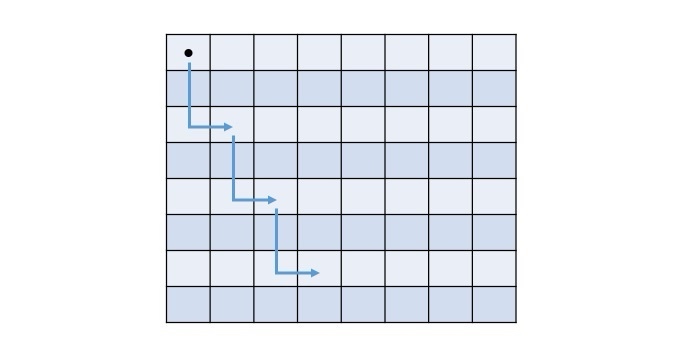
For a person who is not familiar with chess, note that the knight moves in a special manner. It can move either two squares horizontally and one square vertically or two squares vertically and one square horizontally in each direction. So, the complete movement looks like English letter 'L'.

        int[] xMove = {2, 1, -1, -2, -2, -1, 1, 2};

        int[] yMove = {1, 2, 2, 1, -1, -2, -2, -1};



Suppose the size of given chess board is 8 and the knight is at the top-left position on the board. The next possible moves are shown below –



Each cell in the above chess board holds a number, that indicates where to start and in how many moves the knight will reach a cell. The final values of the cell will be represented by the below matrix −

Backtracking Approach to Solve Knight's tour problem

The other way to solve this problem is to use backtracking. It is a technique that tries different possibilities until a solution is found or all options are tried. It involves choosing a move, making it, and then recursively trying to solve the rest of the problem. If the current move leads to a dead end, we backtrack and undo the move, then try another one.

* Start from any cell on the board and mark it as visited by the knight.
* Move the knight to a valid unvisited cell and mark it visited. From any cell, a knight can take a maximum of 8 moves.
* If the current cell is not valid or not taking to the solution, then backtrack and try other possible moves that may lead to a solution.
* Repeat this process until the moves of knight are equal to 8 x 8 = 64.

package BacktrackingAlgorithms;  
  
import java.util.Arrays;  
import java.util.List;  
  
public class KnightTour {  
 private static final int *BOARD\_SIZE* = 8;  
 private static final int *MOVE\_COUNT* = *BOARD\_SIZE* \* *BOARD\_SIZE*;  
 private static final int[][] *X\_MOVES* = {{2, 1, -1, -2, -2, -1, 1, 2}};  
 private static final int[][] *Y\_MOVES* = {{1, 2, 2, 1, -1, -2, -2, -1}};  
  
 private static boolean isValidMove(int row, int col, int[][] solution) {  
 return row >= 0 && row < *BOARD\_SIZE* && col >= 0 && col < *BOARD\_SIZE* && solution[row][col] == -1;  
 }  
  
 private static void displaySolution(int[][] solution) {  
 System.*out*.println("The possible solution:");  
 for (int[] row : solution) {  
 for (int val : row) {  
 System.*out*.printf("%3d ", val);  
 }  
 System.*out*.println();  
 }  
 }  
  
 private static boolean knightTour(int row, int col, int move, int[][] solution) {  
 if (move == *MOVE\_COUNT*) {  
 return true;  
 }  
  
 for (int k = 0; k < 8; k++) {  
 int newRow = row + *X\_MOVES*[0][k];  
 int newCol = col + *Y\_MOVES*[0][k];  
  
 if (*isValidMove*(newRow, newCol, solution)) {  
 solution[newRow][newCol] = move;  
 if (*knightTour*(newRow, newCol, move + 1, solution)) {  
 return true;  
 } else {  
 solution[newRow][newCol] = -1;  
 }  
 }  
 }  
  
 return false;  
 }  
  
 public static boolean findKnightTourSolution() {  
 int[][] solution = new int[*BOARD\_SIZE*][*BOARD\_SIZE*];  
 for (int[] row : solution) {  
 Arrays.*fill*(row, -1);  
 }  
  
 solution[0][0] = 0;  
  
 if (!*knightTour*(0, 0, 1, solution)) {  
 System.*out*.println("Solution does not exist");  
 return false;  
 } else {  
 *displaySolution*(solution);  
 }  
  
 return true;  
 }  
  
 public static void main(String[] args) {  
 *findKnightTourSolution*();  
 }  
}

**Rat in a Maze (Problem Type: Path Finding)**

The rat in a maze problem is a path finding puzzle in which our objective is to find an optimal path from a starting point to an exit point. In this puzzle, there is a rat which is trapped inside a maze represented by a square matrix. The maze contains different cells through which that rat can travel in order to reach the exit of maze.

Backtracking Approach

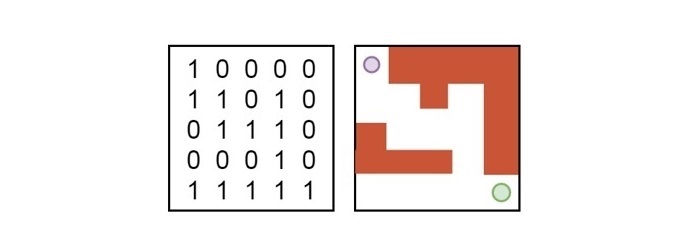
Maze is of size = NxN, where cells can either be marked as 1 or 0.

1 - valid path, 0 - blocked cell.

Rat can move in up, down, left, or right directions, but it can only visit each cell once.

Source - Top left corner (0,0)

Destination – Bottom Right Corner (n-1,n-1).



Steps:

1. Create method isValid() – to check the cell is blocked(0) or unblocked(1) and the position (x,y) is inside the maze.
2. Backtracking – Direction(Right, left, up, down) String – DLRU.
3. findPath() – find all the possible paths.

package BacktrackingAlgorithms;  
  
import java.util.ArrayList;  
import java.util.List;  
  
public class RatInMaze  
{  
 static String *direction* = "DLRU";  
 static int[] *dr* = {1,0,0,-1};  
 static int[]*dc* = {0,-1,1,0};  
 public static boolean isValid(int r, int c, int n, int[][] maze)  
 {  
 return r>=0 && c>=0 && r<n && c<n && maze[r][c]==1;  
 }  
  
 public static void findPath(int r, int c, int n, int[][] maze, List<String> res,StringBuilder cp)  
 {  
 if(r ==n-1 && c==n-1) {  
 res.add(cp.toString());  
 return;  
 }  
 maze[r][c] =0 ;  
 for(int i=0;i<n-1;i++)  
 {  
 int nextr = r+*dr*[i];  
 int nextc = c+*dc*[i];  
 if(*isValid*(nextr,nextc,n,maze))  
 {  
 cp.append(*direction*.charAt(i)); // appending cp(currentpath)  
 *findPath*(nextr,nextc,n,maze,res,cp); // calling the findPath()  
 cp.deleteCharAt(cp.length()-1);// backtracking  
  
 }  
  
 }  
 maze[r][c]= 1;  
  
 }  
  
 public static void main(String args[])  
 {  
 int maze[][] = {{1,0,0,0},{1,1,0,1},{1,1,0,0},{0,1,1,1}};  
 int n = maze.length;  
 List<String> res = new ArrayList<>();  
 StringBuilder cp = new StringBuilder();  
 if(maze[0][0]!=0 && maze[n-1][n-1] !=0)  
 *findPath*(0,0,n,maze,res,cp);  
 if(res.isEmpty())  
 System.*out*.println(-1-1);  
 else  
 for(String p : res)  
 System.*out*.println(p+ " ");  
 System.*out*.println(" ");  
 }  
}

**N Queen Problem (Problem Type: Arrangement)**

What is Backtracking?

In backtracking, we start with one possible move out of many avail­able moves. We then try to solve the prob­lem.

If we are able to solve the prob­lem with the selected move then we will print the solu­tion. Else we will back­track and select some other move and try to solve it.

If none of the moves works out us claim that there is no solu­tion for the problem.

What is the N-Queens Problem?

How can N queens be placed on an NxN chessboard so that no two of them attack each other?

This problem is commonly seen for N=4 and N=8.

Let’s look at an example where N=4

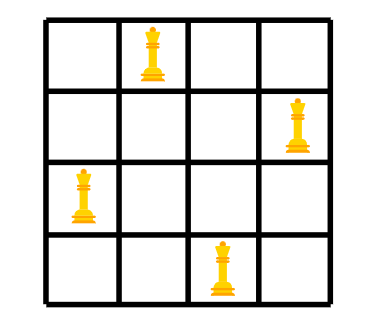
Before solving the problem, you need to know about the movement of the queen in chess.

A queen can move any number of steps in any direction. The only constraint is that it can’t change its direction while it’s moving.

One thing that is clear by looking at the queen’s movement is that no two queens can be in the same row or column.

That allows us to place only one queen in each row and each column.

When N=4, the solution looks like:



Steps:

1. N\*N chess board
2. Initialize it with 0.
3. Place the queens in each column.
4. Check if the position is safe by calling the isSafe() method
5. If we got the valid position print it otherwise backtracking.

package BacktrackingAlgorithms;  
  
public class NQueenProblem {  
 final int N = 4;  
 void print(int board[][])  
 {  
 for(int i=0;i<N;i++)  
 {  
 for(int j =0;j<N;j++)  
 System.*out*.print(""+ board[i][j]+" ");  
 System.*out*.println(" ");  
 }  
 }  
 boolean isSafe(int board[][], int r, int c)  
 {  
 for(int i=0;i<c;i++)  
 {  
 if(board[r][i]==1)  
 return false;  
 if(r-i>0 && board[r-i][c-i]==1)  
 return false;  
 if(r+i<N && board[r+i][c-i]==1)  
 return false;  
  
 }  
 return true;  
 }  
 boolean solveNQUtil(int board[][],int c)  
 {  
 if(c>=N)  
 return true;  
 for(int i=0;i<N;i++)  
 {  
 if (isSafe(board, i, c)) {  
 board[i][c] = 1;  
 if (solveNQUtil(board, c + 1))  
 return true;  
 board[i][c] =0 ; // Back tracking  
 }  
 }  
 return false;  
 }  
 boolean solveNQ()  
 {  
 int board[][] = new int[N][N];  
 if(!solveNQUtil(board,0))  
 {  
 System.*out*.println("Solution not found");  
 return false;  
 }  
 print(board);  
 return true;  
 }  
 public static void main(String args[])  
 {  
 NQueenProblem obj = new NQueenProblem();  
 obj.solveNQ();  
 }  
}

**Subset Sum (Problem Type: Subset)**

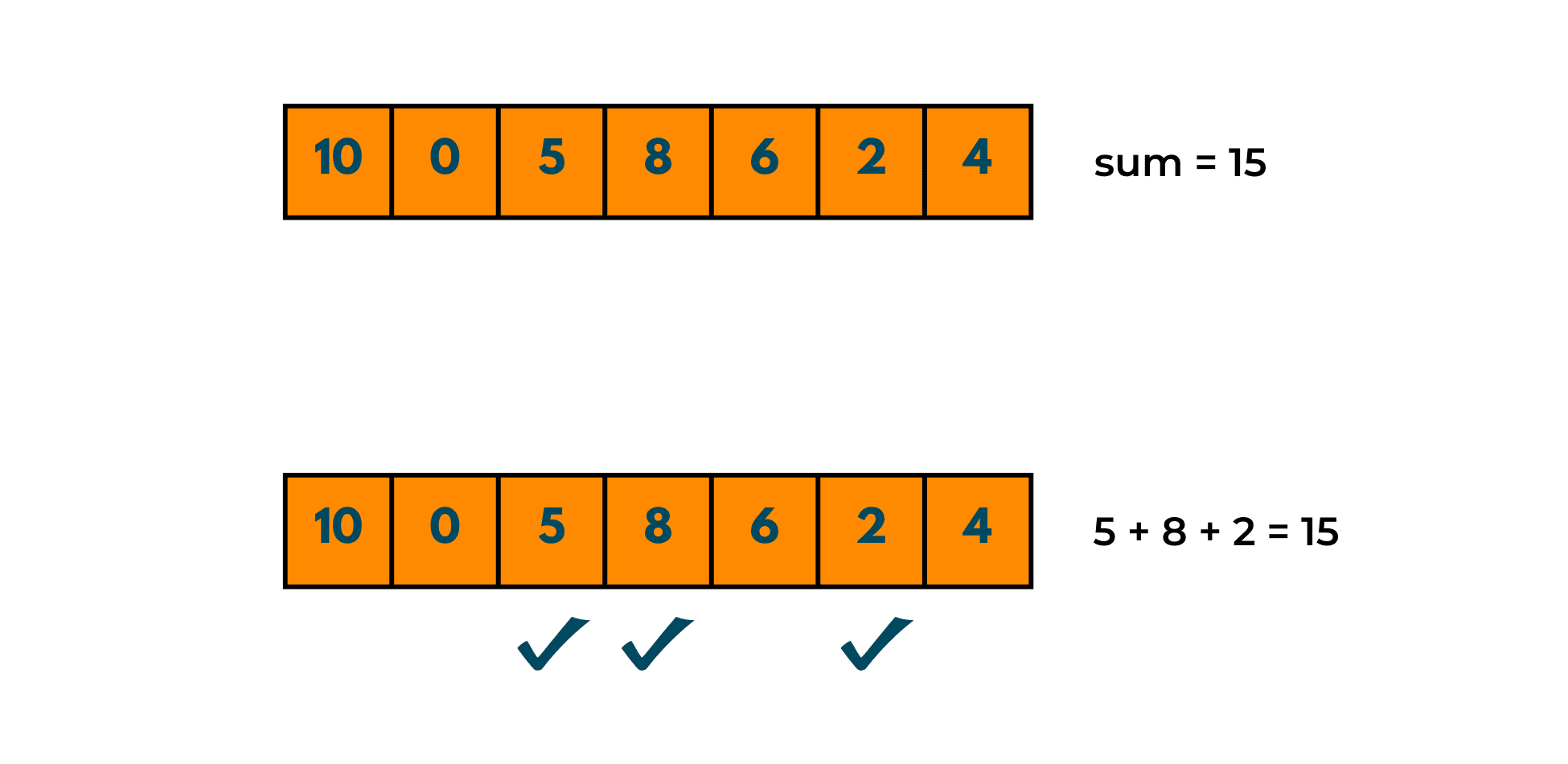
You are given an array of non-negative numbers and a value 'sum'. You have to find out whether a subset of the given array is present whose sum is equal to the given value."

Let's look at an example:

Input: {10, 0, 5, 8, 6, 2, 4}, 15

Output: True

Explanation: The sum of the subset {5,8,2} gives the sum as 15. Therefore, the answer comes out to be true.

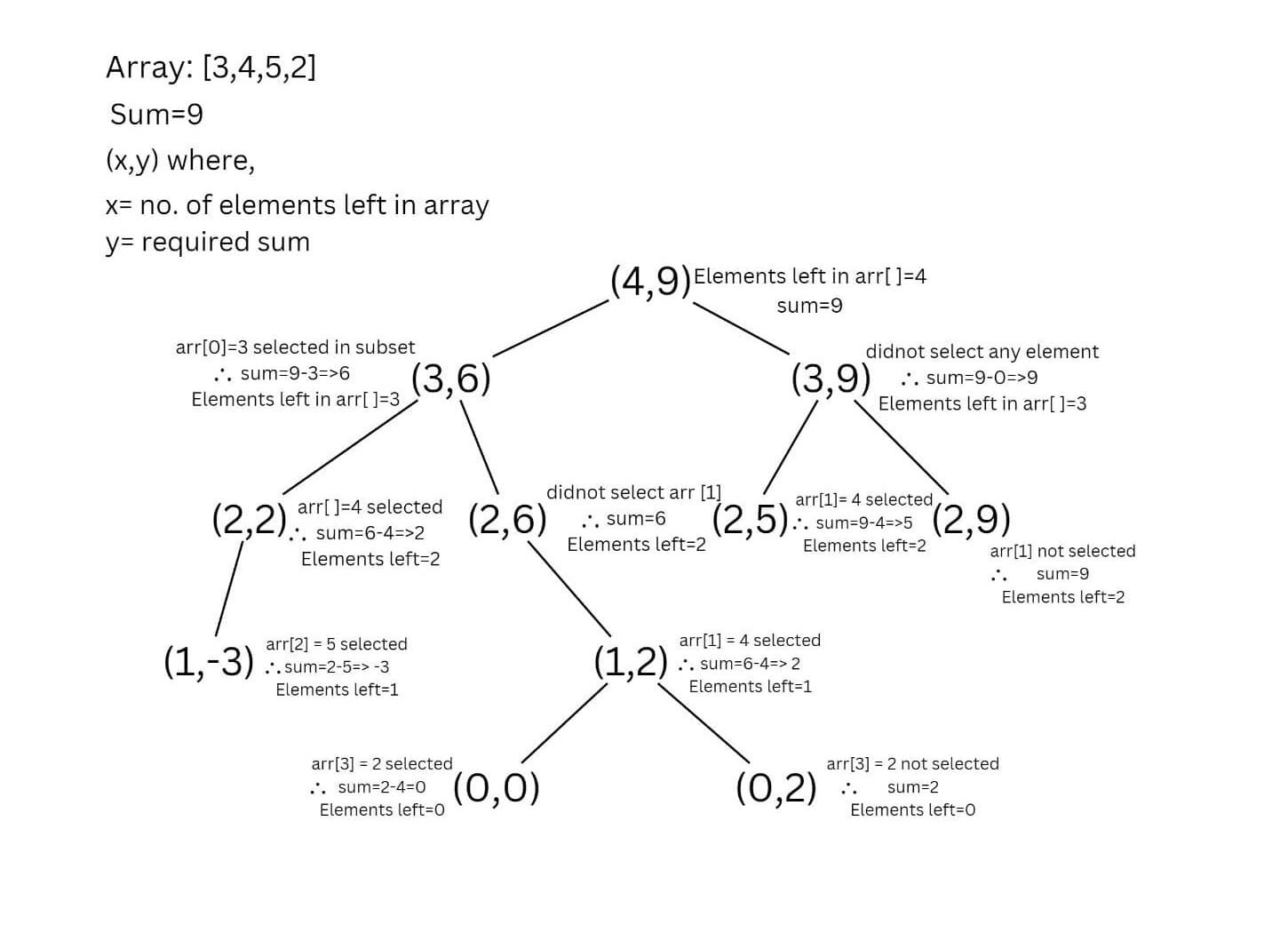


Recursion

Recursion is a widely known concept and is majorly famous in questions where you need to process the same data again and again till you hit the base condition. So, in this method, we'll be using recursion only. Let's understand the algorithm and try to decode the question with some illustrations.

Algorithm:

1. Take input of the array and the value 'sum'.
2. Find the size of the array.
3. Create a function whose return type is Boolean so that it returns true if the sum is found otherwise false.
4. In this function, check whether the last element of the array is greater than the 'sum' or not.
5. If it is greater than skip to the next element and if not, consider it in the subset.
6. Keep decreasing the value of the sum by subtracting the value of the last element (if considered).
7. Keep the base condition to be true till the elements are present and till the given sum is not equal to 0.



package BacktrackingAlgorithms;  
  
public class SubsetSum {  
 static boolean isSubSet(int set[],int n, int sum)  
 {  
 if(sum==0)  
 return true;  
 if(n==0 && sum!=0)  
 return false;  
 if(set[n-1]>sum)  
 return *isSubSet*(set,n-1,sum);  
 return(*isSubSet*(set,n-1,sum-set[n-1]) || *isSubSet*(set,n-1,sum));  
 }  
  
 public static void main(String args[])  
 {  
 int set[] ={1, 9, 7, 5, 18, 12, 20, 15};  
 int sum = 35;  
 int n = set.length;  
 if(*isSubSet*(set,n,sum))  
 System.*out*.println("Subset is present");  
 else  
 System.*out*.println("SubSet Not present");  
 }  
}