1. Job Sequencing

Given a set of **N** jobs where each **job$_i$** has a deadline and profit associated with it.

Each job takes ***1*** unit of time to complete and only one job can be scheduled at a time. We earn the profit associated with job if and only if the job is completed by its deadline.

Find the number of jobs done and the **maximum profit**.

Note: Jobs will be given in the form (Job$_{id}$, Deadline, Profit) associated with that Job.

Input:
N = 4
Jobs = {(1,4,20),(2,1,10),(3,1,40),(4,1,30)}
Output:
2 60
Explanation:
Job1 and Job3 can be done with
maximum profit of 60 (20+40).

```
Arrays.sort(arr,(a,b)->{
            return b.profit-a.profit;
        });
        int count =0;
        int sum =0;
        int prev[] = new int[n+1];
        for(int i=0;i<=n;i++) prev[i]=i;
        for(int i=0;i<n;i++){
            int ded = arr[i].deadline;

            while(ded != prev[ded]){
                int temp = prev[ded];
                prev[ded] =temp;
                ded = temp;
            }
            if(prev[ded]!=0){
                count++;
                sum+=arr[i].profit;
                prev[ded] = prev[ded-1];
            }
        }

        int ans[] = {count,sum};

        return ans;
```

2. Fractional Knapsack

Given *weights* and *values* of **N** items, we need to put these items in a knapsack of capacity **W** to get the *maximum* total value in the knapsack.

Note: Unlike 0/1 knapsack, you are allowed to break the item.

```
    Arrays.sort(arr, new Comparator<Item>()
{
  @Override
  public int compare(Item o1, Item o2)
  {
      double rateProfitItem1 = (double)o1.value / o1.weight;
      double rateProfitItem2 = (double) o2.value / o2.weight;
      if (rateProfitItem2 > rateProfitItem1)
      {
        return 1;
      }
      else if (rateProfitItem2 < rateProfitItem1)
      {
        return -1;
      }
      else
      {
        return 0;
      }
  }
});
double sumProfit = 0d;
for (int i = 0; i < n; i++)
{
    if (W == 0)
    {
        break;
    }
    else if (arr[i].weight <= W)
    {
        sumProfit += arr[i].value;
        W -= arr[i].weight;
    }
    else
    {
        double fraction = (double)W / arr[i].weight;
        sumProfit += fraction * arr[i].value;
        W = (int)(W - (arr[i].weight * fraction));
    }
}
return sumProfit;
```

3. Min Platform

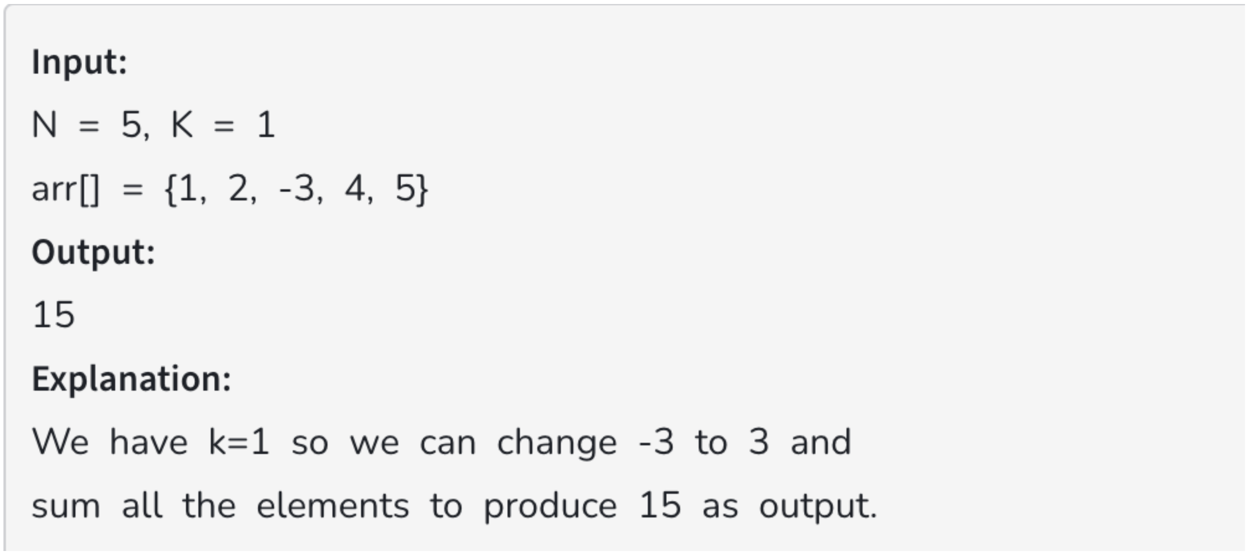
Given arrival and departure times of all trains that reach a railway station. Find the minimum number of platforms required for the railway station so that no train is kept waiting.
Consider that all the trains arrive on the same day and leave on the same day. Arrival and departure time can never be the same for a train but we can have arrival time of one train equal to departure time of the other. At any given instance of time, same platform can not be used for both departure of a train and arrival of another train. In such cases, we need different platforms**.**

```
Input: n = 6
arr[] = {0900, 0940, 0950, 1100, 1500, 1800}
dep[] = {0910, 1200, 1120, 1130, 1900, 2000}
Output: 3
Explanation:
Minimum 3 platforms are required to
safely arrive and depart all trains.
```

```
 Arrays.sort(arr);
Arrays.sort(dep);
 int platform=1;
  int i=1;
  int j=0;
   int  result=1;
    while(i<n && j<n){
         if(arr[i]<=dep[j]){
              platform++;
              i++;
               if(platform>result){
                   result=platform;
               }
         }
          else{
               platform--;
                j++;
          }
    }
     return result;
```

4. Maximize sum after K negations

Given an array of integers of size **N** and a number **K**., You must modify array **arr**[] exactly **K** number of times. Here modify array means in each operation you can replace any array element either **arr[i]** by **-arr[i]** or **-arr[i]** by **arr[i]**. You need to perform this operation in such a way that after K operations, the sum of the array must be maximum.



Input:

N = 5, K = 1

arr[] = {1, 2, -3, 4, 5}

Output:

15

Explanation:

We have k=1 so we can change -3 to 3 and sum all the elements to produce 15 as output.

```
Arrays.sort(a);
int i = 0;
while (k > 0 && i < n && a[i] < 0) {
   a[i] = -a[i];
   k--;
   i++;
}
if (k % 2 == 1) {
   Arrays.sort(a);
   a[0] = -a[0];
}
long sum = 0;
for (int j = 0; j < n; j++) {
   sum += a[j];
}
return sum;
```

5.Chocolate Distribution Problem

Given an array **A[ ]** of positive integers of size **N**, where each value represents the number of chocolates in a packet. Each packet can have a variable number of chocolates. There are **M** students, the task is to distribute chocolate packets among **M** students such that :

1. Each student gets **exactly** one packet.
2. The difference between maximum number of chocolates given to a student and minimum number of chocolates given to a student is minimum.

Input:
N = 8, M = 5
A = {3, 4, 1, 9, 56, 7, 9, 12}
Output: 6
Explanation: The minimum difference between maximum chocolates and minimum chocolates is 9 - 3 = 6 by choosing following M packets :{3, 4, 9, 7, 9}.

```
    long ans = Integer.MAX_VALUE;
  Collections.sort(a);
  for(int i=0; i<=n-m; i++){
     long diff = a.get(i+m-1) - a.get(i);
     ans = Math.min(ans,diff);
  }
  return ans;
```

6. Minimum Cost of ropes

There are given **N** ropes of different lengths, we need to connect these ropes into one rope. The cost to connect two ropes is equal to sum of their lengths.
The task is to connect the ropes with minimum cost. Given **N** size array **arr**[] contains the lengths of the ropes.

```
Input:
n = 5
arr[] = {4, 2, 7, 6, 9}
Output:
62
Explanation:
First, connect ropes 4 and 2, which makes
the array {6,7,6,9}. Cost of
this operation 4+2 = 6. Next, add ropes
6 and 6, which results in {12,7,9}.
Cost of this operation 6+6 = 12.
Then, add 7 and 9, which makes the array {12,16}.
Cost of this operation 7+9 = 16. And
finally, add these two which gives {28}.
Hence, the total cost is 6 + 12 + 16 +
28 = 62.
```

```
  PriorityQueue<Long> pq  = new PriorityQueue<>();
for(int i=0;i<arr.length;i++){
   pq.add(arr[i]);
}
long cost =0;
while(pq.size()>1){
   long min = pq.remove();
   long min2 = pq.remove();
   cost += min+min2;
   pq.add(min+min2);
}
return cost;
```

7. Gas Station

There are `n` gas stations along a circular route, where the amount of gas at the `ith` station is `gas[i]`.

You have a car with an unlimited gas tank and it costs `cost[i]` of gas to travel from the `ith` station to its next `(i + 1)th` station. You begin the journey with an empty tank at one of the gas stations.

Given two integer arrays `gas` and `cost`, return *the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return* `-1`. If there exists a solution, it is **guaranteed** to be **unique**

```
Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
Output: 3
Explanation:
Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4 = 4
Travel to station 4. Your tank = 4 - 1 + 5 = 8
Travel to station 0. Your tank = 8 - 2 + 1 = 7
Travel to station 1. Your tank = 7 - 3 + 2 = 6
Travel to station 2. Your tank = 6 - 4 + 3 = 5
Travel to station 3. The cost is 5. Your gas is just enough to travel back to station 3.
Therefore, return 3 as the starting index.
```

```
class Solution {
  public int canCompleteCircuit(int[] gas, int[] cost) {
     int totalCost = 0;
     int surplusGas = 0;
     int startFrom = 0;
     for(int i=0;i<gas.length;i++){
        totalCost += gas[i]-cost[i];
        surplusGas += gas[i]-cost[i];

        if(surplusGas < 0){
           startFrom = i+1;
           surplusGas = 0;
        }
     }
     if(totalCost>=0) return startFrom;
     return -1;

  }
}
```

8. Container with most water

You are given an integer array `height` of length `n`. There are `n` vertical lines drawn such that the two endpoints of the `i^th` line are `(i, 0)` and `(i, height[i])`.

Find two lines that together with the x-axis form a container, such that the container contains the most water.

Return *the maximum amount of water a container can store*.

Notice that you may not slant the container.

```java
class Solution {
  public int maxArea(int[] height) {
      int left = 0;
      int right = height.length-1;
      int maxArea = 0;
      while(left<right){
          int start = height[left];
          int end = height[right];
          int min = Math.min(start,end);
          int dist = right-left;
          int currArea = min * dist;
          maxArea = Math.max(currArea,maxArea);
          if(start>end){
              right--;
          }else{
              left++;
          }
      }
      return maxArea;
  }
}
```

9. Candy

There are `n` children standing in a line. Each child is assigned a rating value given in the integer array `ratings`.

You are giving candies to these children subjected to the following requirements:

- Each child must have at least one candy.
- Children with a higher rating get more candies than their neighbors.

Return *the minimum number of candies you need to have to distribute the candies to the children*.

```java
class Solution {
  public int candy(int[] ratings) {
      int n = ratings.length;
      int[] candies = new int[n];
      Arrays.fill(candies, 1);

      for (int i = 1; i < n; i++) {
          if (ratings[i] > ratings[i - 1]) {
              candies[i] = candies[i - 1] + 1;
          }
      }

      for (int i = n - 2; i >= 0; i--) {
          if (ratings[i] > ratings[i + 1]) {
              candies[i] = Math.max(candies[i], candies[i + 1] + 1);
          }
      }

      int totalCandies = 0;
      for (int candy : candies) {
          totalCandies += candy;
      }

      return totalCandies;
  }
}
```

10. Coin change

Given an integer array **coins**[ ] of size **N** representing different denominations of currency and an integer **sum**, find the number of ways you can make **sum** by using different combinations from coins[ ].
Note: Assume that you have an infinite supply of each type of coin. And you can use any coin as many times as you want.

```
Input:
N = 3, sum = 4
coins = {1,2,3}
Output: 4
Explanation: Four Possible ways are: {1,1,1,1},{1,1,2},{2,2},{1,3}.
```

```
    long dp[] = new long [sum + 1];
  dp[0] = 1;
 for (int i = 0; i < N; i++)
    for (int s = coins[i]; s <= sum; s++) {
        dp[s] = dp[s] + dp[s - coins[i]];


    }

 return dp[sum];
```

11. Partition equal subset sum

Given an integer array `nums`, return `true` *if you can partition the array into two subsets such that the sum of the elements in both subsets is equal or* `false` *otherwise*.

Example 1:

```
Input: nums = [1,5,11,5]
Output: true
Explanation: The array can be partitioned as [1, 5, 5] and [11].
```

```
class Solution {
    public boolean canPartition(int[] nums) {
        int sum = 0;
        int n = nums.length;
        for(int i : nums) sum+=i;
        if(sum%2!=0) return false;
        sum /= 2;
        boolean[] dp = new boolean[sum+1];
        dp[0] = true;
        for (int j : nums) {
            for (int i = sum; i > 0; i--) {
                if (i >= j) {
                    dp[i] = dp[i] || dp[i-j];
                }
            }
        }
        return dp[sum];
    }
}
```

12. Max path sum in matrix

Given a NxN matrix of positive integers. There are only three possible moves from a cell **Matrix[r][c]**.

1. Matrix [r+1] [c]
2. Matrix [r+1] [c-1]
3. Matrix [r+1] [c+1]

Starting from any column in row 0 return the largest sum of any of the paths up to row N-1.

NOTE: We can start from any column in zeroth row and can end at any column in (N-1)th row.

```
int dp[] = new int[N];

        //for 1st row
        for(int i = 0 ; i<N ; i++){
          dp[i] = Matrix[0][i];
        }

        for(int row = 1 ; row<N ; row++){
            int temp[] = new int[N];
            for(int col = 0 ; col<N ; col++){

                int down = Matrix[row][col] + dp[col];

                int left =   Matrix[row][col];
                if(col-1 >= 0) left += dp[col-1];
                else left += (int)(-1e9 + 7);

                int right =Matrix[row][col];
                if(col+1 < N) right += dp[col+1];
                else right += (int)(-1e9 + 7);

                temp[col] = Math.max(down , Math.max(left , right));
            }
            dp = temp;
        }

        int ans = 0;
        for(int i = 0 ; i<N ; i++){
           ans = Math.max(dp[i] , ans);
       }

       return ans;
```

13. Knapsack with Duplicate Items

Given a set of **N** items, each with a weight and a value, represented by the array **w** and **val** respectively. Also, a knapsack with weight limit **W**.

The task is to fill the knapsack in such a way that we can get the maximum profit. Return the maximum profit.

Note: Each item can be taken any number of times.

```
Input:
N = 2
W = 3
val = {1, 1}
wt = {2, 1}
Output:
3
Explanation:
1.Pick the 2nd element thrice.
2.Total profit = 1 + 1 + 1 = 3. Also the total weight = 1 + 1 + 1  = 3 which is <= 3.
```

```
int dp[][] = new int [N+1][W+1];

        for(int i=1; i<N+1; i++){
            for(int j=1; j<W+1; j++){
                //check if valid
                if(wt[i-1] <= j){
                    //include
                    int include = val[i-1]+dp[i][j-wt[i-1]];
                    //exclude
                    int exclude = dp[i-1][j];

                    dp[i][j] = Math.max(include,exclude);
                }else{
                    dp[i][j] = dp[i-1][j];
                }
            }
        }
        return dp[N][W];
```

14. Largest square in matrix

Given a binary matrix **mat** of size **n** * **m**, find out the maximum size square sub-matrix with all 1s.

Example 1:

```
Input: n = 2, m = 2
mat = {{1, 1},
       {1, 1}}
Output: 2
Explaination: The maximum size of the square
sub-matrix is 2. The matrix itself is the
maximum sized sub-matrix in this case.
```

```
 int[] maxi = new int[] { 0 };

solveDP2(n,m,mat,maxi);
        return maxi[0];

static void solveDP2(int n, int m, int mat[][], int[] maxi){
        int curr = 0;       // row i column j
        int curr_plus = 0;  // row i column j+1
        int next = 0;       // row i+1 column j
        int next_plus = 0;  // row i+1 columns j+1

        for(int i=n-1;i>=0;i--){
            for(int j=m-1;j>=0;j--){
                int right = curr_plus;
                int diagonal = next_plus;
                int down = next;
                if (mat[i][j] == 1) {
                    int ans = 1 + Math.min(right, Math.min(diagonal, down));
                    maxi[0] = Math.max(maxi[0], ans);
                    curr = ans;
                }
                else
                    curr = 0;
            }
            next = curr;
            next_plus = curr_plus;
        }
    }
```

15. Stone Game

Alice and Bob play a game with piles of stones. There are an **even** number of piles arranged in a row, and each pile has a **positive** integer number of stones `piles[i]`.

The objective of the game is to end with the most stones. The **total** number of stones across all the piles is **odd**, so there are no ties.

Alice and Bob take turns, with **Alice starting first**. Each turn, a player takes the entire pile of stones either from the **beginning** or from the **end** of the row. This continues until there are no more piles left, at which point the person with the **most stones wins**.

Assuming Alice and Bob play optimally, return `true` *if Alice wins the game, or* `false` *if Bob wins*.

```
class Solution {
        public boolean stoneGame(int[] p) {
        int n = p.length;
        int[][] dp  = new int[n][n];
        for (int i = 0; i < n; i++) dp[i][i] = p[i];
        for (int d = 1; d < n; d++)
            for (int i = 0; i < n - d; i++)
                dp[i][i + d] = Math.max(p[i] - dp[i + 1][i + d], p[i + d] - dp[i][i + d - 1]);
        return dp[0][n - 1] > 0;
    }
}
```

16. Non negative no without consecutive ones

Given a positive integer `n`, return the number of the integers in the range `[0, n]` whose binary representations **do not** contain consecutive ones.

Example 1:

```
Input: n = 5
Output: 5
Explanation:
Here are the non-negative integers <= 5 with their corresponding binary representations:
0 : 0
1 : 1
2 : 10
3 : 11
4 : 100
5 : 101
Among them, only integer 3 disobeys the rule (two consecutive ones) and the other 5 satisfy the rule.
```

```
class Solution {
    public int findIntegers(int n) {
        String binary = Integer.toBinaryString(n);
        int k = binary.length();

        int[] fib = new int[k+1];
        fib[0] = 1;
        fib[1] = 2;
        for(int i=2;i<=k;i++){
            fib[i] = fib[i-1]+fib[i-2];
        }

        boolean isLastBitOne = false;
        int res=0;
        int bit = k-1;
        while(bit>=0){
            if((n & (1<<bit))==0){
                isLastBitOne=false;
            } else {
                res+=fib[bit];
                if(isLastBitOne){
                    return res;
                }
                isLastBitOne=true;
            }
            bit--;
        }
        return res+1;
    }
}
```

17. Smallest sufficient team

In a project, you have a list of required skills `req_skills`, and a list of people. The `i`th person `people[i]` contains a list of skills that the person has.

Consider a sufficient team: a set of people such that for every required skill in `req_skills`, there is at least one person in the team who has that skill. We can represent these teams by the index of each person.

- For example, `team = [0, 1, 3]` represents the people with skills `people[0]`, `people[1]`, and `people[3]`.

Return *any sufficient team of the smallest possible size, represented by the index of each person*. You may return the answer in **any order**.

It is **guaranteed** an answer exists.

Example 1:

```
Input: req_skills = ["java","nodejs","reactjs"], people = [["java"],["nodejs"],["nodejs","reactjs"]]
Output: [0,2]
```

```
class Solution {
        public int[] smallestSufficientTeam(String[] req_skills, List<List<String>> people)
{
        int n = req_skills.length, m = people.size();
        HashMap<String, Integer> skill_index = new HashMap<>();
        for (int i = 0; i < n; ++i)
            skill_index.put(req_skills[i], i);
        List<Integer>[] dp = new List[1 << n];
        dp[0] = new ArrayList<>();
        for (int i = 0; i < m; ++i) {
            int cur_skill = 0;
            for (String s : people.get(i))
                cur_skill |= 1 << skill_index.get(s);
            for (int prev = 0; prev < dp.length; ++prev) {
                if (dp[prev] == null) continue;
                int comb = prev | cur_skill;
                if (dp[comb] == null || dp[prev].size() + 1 < dp[comb].size()) {
                    dp[comb] = new ArrayList<>(dp[prev]);
                    dp[comb].add(i);
                }
            }
        }
        return dp[(1 << n) - 1].stream().mapToInt(i -> i).toArray();
    }
}
```