Greedy Algorithms

Assignment Solutions

PW SKILLS

Q1. Find the minimum sum of Products of two arrays of the same size, given that k modifications are allowed on the first array. In each modification, one array element of the first array can either be increased or decreased by 2.

Examples:
Input : a[] = {1, 2, -3}
b[] = {-2, 3, -5}
k = 5
Output : -31
Explanation:
Here n = 3 and k = 5.
So, we modified a[2], which is -3 and increased it by 10 (as 5 modifications are allowed).
Final sum will be :
(1 * -2) + (2 * 3) + (7 * -5)
-2 + 6 - 35
-31
(which is the minimum sum of the array with given conditions)

Input : a[] = {2, 3, 4, 5, 4}
b[] = {3, 4, 2, 3, 2}
k = 3
Output : 25
Explanation:
Here, total numbers are 5 and total modifications allowed are 3. So, modify a[1], which is 3 and decreased it by 6 (as 3 modifications are allowed).
Final sum will be :
(2 * 3) + (-3 * 4) + (4 * 2) + (5 * 3) + (4 * 2)
6 – 12 + 8 + 15 + 8
25
(which is the minimum sum of the array with given conditions)

Solution :

Code : ASS_Code1.java

Output :

```
-15
```

Approach :

- As in the original array A and B product is constant , then we make modification and there are 4 choices
  - min*2K
  - min*-2K
  - max*2K
  - max*-2K

- Now whatever is minimum is added original product
- Since we need to minimize the product sum, we find the maximum product and reduce it.
- We can observe that making 2*k changes to only one element is enough to get the minimum sum.
- Based on this observation, we consider every element as the element on which we apply all k operations and keep track of the element that reduces the result to minimum.

Q2. You are given n activities with their start and finish times. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a time.

Examples:
Input: start[] = {10, 12, 20}, finish[] = {20, 25, 30}
Output: 0 2
Explanation: A person can perform at most two activities. The maximum set of activities that can be executed is {0, 2} [These are indexes in start[] and finish[]]

Input: start[] = {1, 3, 0, 5, 8, 5}, finish[] = {2, 4, 6, 7, 9, 9};
Output: 0 1 3 4

Explanation: A person can perform at most four activities. The maximum set of activities that can be executed is {0, 1, 3, 4} [These are indexes in start[] and finish[]]

Solution :

Code : ASS_Code2.java

Output :

```
These activities are selected :
0 1 3 4
```

Approach :

- The greedy choice is to always pick the next activity whose finish time is the least among the remaining activities and the start time is more than or equal to the finish time of the previously selected activity.
- We can sort the activities according to their finishing time so that we always consider the next activity as the minimum finishing time activity.
- Select the first activity from the sorted array and print it.
- Do the following for the remaining activities in the sorted array:
  - If the start time of this activity is greater than or equal to the finish time of the previously selected activity then select this activity and print it
- Note: In the implementation, it is assumed that the activities are already sorted according to their finish time.

Q3. 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.

Example 1:
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.

Example 2:
Input: gas = [2,3,4], cost = [3,4,3]
Output: -1
Explanation:
You can't start at station 0 or 1, as there is not enough gas to travel to the next station.
Let's start at station 2 and fill up with 4 unit of gas. Your tank = 0 + 4 = 4
Travel to station 0. Your tank = 4 - 3 + 2 = 3
Travel to station 1. Your tank = 3 - 3 + 3 = 3
You cannot travel back to station 2, as it requires 4 units of gas but you only have 3.
Therefore, you can't travel around the circuit once no matter where you start.

Solution :

Code : ASS_Code3.java

Output :

```
The starting station is : 3
```

Approach :

- The function takes in two arrays: gas and cost, where gas[i] represents the amount of gas available at the i-th gas station, and cost[i] represents the amount of gas needed to travel from the i-th gas station to the next one.
- The function first calculates the total amount of gas available at all the gas stations (totalGas) and the total amount of gas needed to travel around all the gas stations (totalCost). If totalGas is less than totalCost, it means that it is not possible to travel around all the gas stations, so the function returns -1.
- Next, the function starts at the first gas station (start=0) and iterates through the gas stations. At each iteration, it calculates the remaining gas after visiting the current gas station (remainsGas) and adding the gas available at that station (gas[i]) and subtracting the gas needed to travel to the next gas station (cost[i]).
- If remainsGas becomes negative at any point, it means that it is not possible to travel from the current gas station to the next one without running out of gas. In this case, the function sets the starting gas station to the next one (start=i+1) and resets the remaining gas to 0 (remainsGas=0).
- Finally, the function returns the starting gas station that allows the travel around all the gas stations.

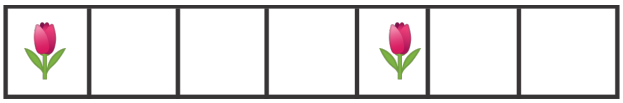
Q4. You have a long flowerbed in which some of the plots are planted, and some are not. However, flowers cannot be planted in adjacent plots.Given an integer array flowerbed containing 0's and 1's, where 0 means empty and 1 means not empty, and an integer n, return if n new flowers can be planted in the flowerbed without violating the no-adjacent-flowers rule.

Example 1:
Input: flowerbed = [1,0,0,0,1], n = 1
Output: true

Example 2:
Input: flowerbed = [1,0,0,0,1], n = 2
Output: false

Solution :

Code : ASS_Code4.java

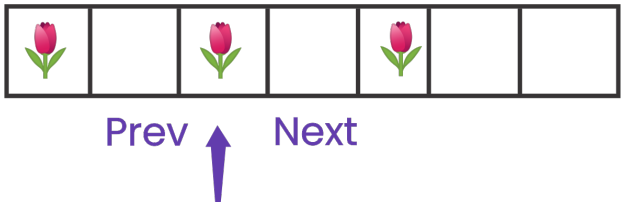


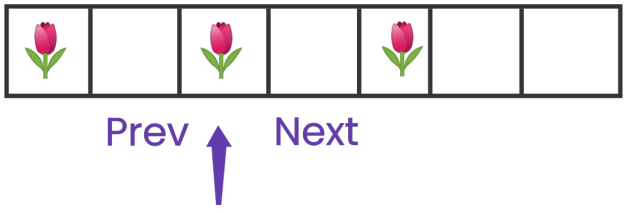
Output :

```
The desired output is :  true
```

Approach :

- Suppose [1,0,0,0,0,1,0] this is a flowerbed given to us.
- Let's forget about how many flowers we can plant into this & we will find the maximum no. of flowers that we can plant.
- So, in order to find that what we can do is, if we see it visually we can see there are two flowers planted in the flower bed of 7 flowers.
- So, we need to iterate over all the flowerbed and we need to focus our attention on the fact where we can plant flowers. The flower can only be planted where there is no flower yet. So, we need to search for the position where the value is 0.
- We start from the 1st index and we will move till we find a value i.e. 0 & going forward we find a place which has no flower.
- Now the condition says there should not be any flower adjacent to this flower. So, in order to place this flower we have to make sure that there is no adjacent flower.
- So, we need to check these two positions i.e. previous position & next position.
- As we can see a flower in its previous position, we can't plant a flower at this place. So, we move our pointer to the next index. As we move here we again check for adjacent flowers.
- As there are no flowers present adjacent to this empty place, we will plant a flower. And also keep a count of the fact that we have planted a flower at this place.





- We move ahead with the same concept checking every two adjacent edges of the empty place & move ahead till we reach the end of the array.

Q5. Given an array of intervals where intervals[i] = [starti, endi], return the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

Example 1:
Input: intervals = [[1,2],[2,3],[3,4],[1,3]]
Output: 1
Explanation: [1,3] can be removed and the rest of the intervals are non-overlapping.

Example 2:
Input: intervals = [[1,2],[1,2],[1,2]]
Output: 2
Explanation: You need to remove two [1,2] to make the rest of the intervals non-overlapping.

Example 3:
Input: intervals = [[1,2],[2,3]]
Output: 0
Explanation: You don't need to remove any of the intervals since they're already non-overlapping.

Solution :

Code : ASS_Code5.java

Output :

```
The desired output is :  1
```

Approach :

- Consider the intervals representing the starting and ending time of meetings you need to schedule in only one meeting room, then the question becomes what is the maximum number of meetings you can host.
- The corresponding greedy approach will be:
  (1) Sort the intervals (meeting times) according to their end time.
  (2) Schedule the meeting with the earliest end time.
  (3) After scheduling a meeting, remove any meetings that overlap with it.