

Greedy II

Analyze the problem E of the contest:

Given several segments of line (int the X axis) with coordinates $[Li, Ri]$. You are to choose the minimal amount of them, such that they would completely cover the segment $[0, M]$.

Input:

The first line is the number of test cases, followed by a blank line.

Each test case in the input should contains an integer M ($1 \leq M \leq 5000$), followed by pairs "Li Ri"

($|Li|, |Ri| \leq 50000$, $i \leq 100000$), each on a separate line. Each test case of input is terminated by pair '0 0'. Each test case will be separated by a single line.

Output:

For each test case, in the first line of output your program should print the minimal number of line segments which can cover segment $[0, M]$. In the following lines, the coordinates of segments, sorted by their left end (Li), should be printed in the same format as in the input. Pair '0 0' should not be printed. If $[0, M]$ can not be covered by given line segments, your program should print '0' (without quotes).

Print a blank line between the outputs for two consecutive test cases.

Sample Input:

```
2
1
-1 0
-5 -3
2 5
0 0
1
-1 0
0 1
0 0
```

Sample Output:

```
0
1
0 1
```

Code Resolution:

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
    int t;
    cin >> t;

    while (t--> 0) {
        int M;
        cin >> M;

        vector<pair<int, int>> coordinates;
        int l, r;

        while (true) {
            cin >> l >> r;
            if (l == 0 && r == 0) {
                break;
            }
            coordinates.push_back({l, r});
        }

        sort(coordinates.begin(), coordinates.end());

        vector<pair<int, int>> ans;
        int current = 0, i = 0;

        while (current < M && i < coordinates.size()) {
            int maxR = -1, index = -1;

            while (i < coordinates.size() && coordinates[i].first <= current) {
                if (coordinates[i].second > maxR) {
                    maxR = coordinates[i].second;
                    index = i;
                }
                i++;
            }

            current = maxR;
            ans.push_back({current, index});
        }
    }
}
```

```

    if (index == -1) {
        break;
    }

    ans.push_back(coordinates[index]);
    current = maxR;
}

if (current < M) {
    cout << 0 << endl;
} else {
    cout << ans.size() << endl;
    for (const auto& p : ans) {
        cout << p.first << " " << p.second << endl;
    }
}

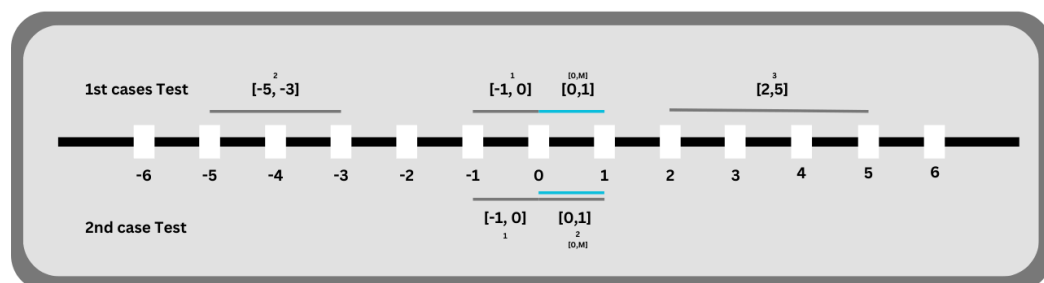
if (t > 0) {
    cout << endl;
}

return 0;
}

```

Explication Problem:

This problem is about finding the minimum number of line segments needed to cover the segment $[0, M]$. Each segment is represented by a pair of coordinates $[Li, Ri]$. The objective is to select a set of segments such that they cover the entire segment $[0, M]$, and then print these segments sorted by their left coordinate (Li).



Logic Code:

In essence, the code solves the problem of selecting the smallest possible number of line segments to completely cover a given interval $[0, M]$. First, it arranges the coordinates of the segments in increasing order according to their initial position. Then, it uses an iterative algorithm to choose segments, looking for those that cover the current point on the X-axis and have the largest possible right-hand coordinate. This process continues until the entire interval $[0, M]$ is covered. The output provides the number of selected segments and their ordered coordinates, complying with the required format. In case it is not possible to cover the interval, a "0" is printed.

Questionnaire:

- Identify the greedy choice

Greedy case:

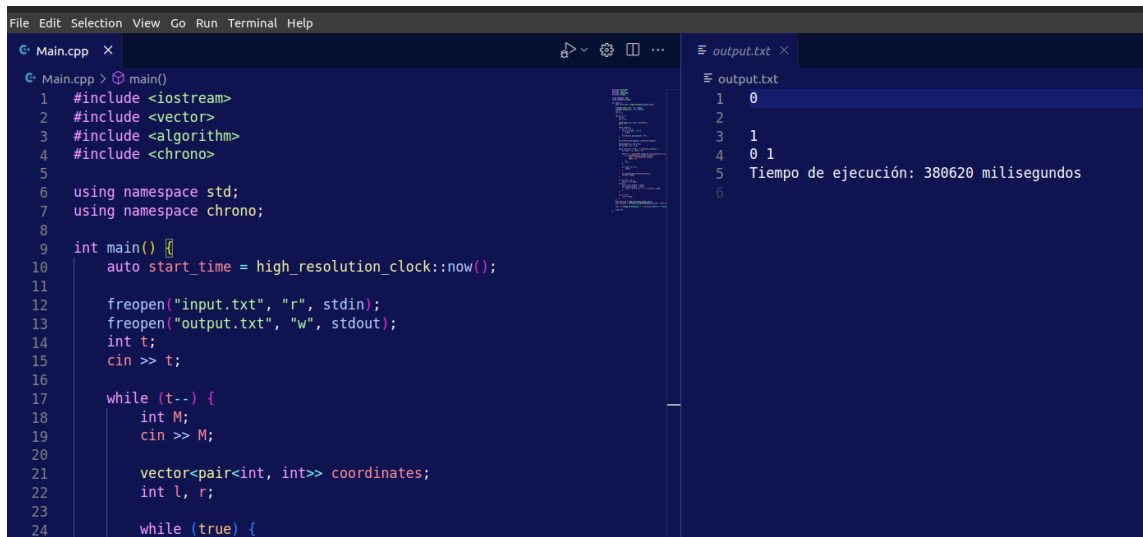
In the greedy case, the algorithm makes local decisions at each step to maximize the coverage of the segment $[0, M]$. Instead of evaluating all possible combinations, the algorithm selects the rightmost segments first, which may not lead to the optimal solution. Although the solution is not necessarily the best possible solution, it is a valid solution according to the greedy approach. The output reflects the number of segments selected and their ordered coordinates.

- Identify the optimal substructure

Optimal case:

In the optimal case, the algorithm evaluates all possible combinations of line segments and selects the one that completely covers the segment $[0, M]$ with the least number of segments. Each test case is considered separately, and a solution is sought that minimizes the total number of segments required. The output reflects the minimum number of segments and their ordered coordinates.

- Make the proof of your greedy algorithm



```
File Edit Selection View Go Run Terminal Help
Main.cpp x
Main.cpp > main()
1 #include <iostream>
2 #include <vector>
3 #include <algorithm>
4 #include <chrono>
5
6 using namespace std;
7 using namespace chrono;
8
9 int main() {
10     auto start_time = high_resolution_clock::now();
11
12     freopen("input.txt", "r", stdin);
13     freopen("output.txt", "w", stdout);
14     int t;
15     cin >> t;
16
17     while (t--) {
18         int M;
19         cin >> M;
20
21         vector<pair<int, int>> coordinates;
22         int l, r;
23
24         while (true) {
```

```
output.txt x
output.txt
1 0
2
3 1
4 0 1
5 Tiempo de ejecución: 380620 milisegundos
6
```

- Implement your code and submit it to the judge

It was successfully uploaded on the first attempt in the judge, I performed it in C++.

- What is the time complexity of your solution?

The complexity of the problem is:

- Data Input: $O(t)$ - t is a constant (It is the number of test cases).
- Sorting: Sort with a sort that has a complexity of $O(n \log n)$.
- Main Loop: The but case loop is $O(n^2)$ the best case of $O(n)$.
- Data output: $O(t)$ - t is a constant (It is the number of test cases).

The complexity could be expressed as:

- $O(t * (n \log n + n^2))$ or in a simplified way $O(n^2)$ the worst case, and the best case $O(n \log n)$

Solve the problems on the pdf:

[Coco_S_theme_party.pdf](#) / [CocosBirthday.pdf](#)

[Coco_S_theme_party.pdf](#)

Analyze the problem:

Coco is organizing a theme party at her house. Each of her guests has a value that indicates their current level of happiness.

Coco knows that if two guests are next to each other, they will talk to each other, and the difference between them will be how much they enjoy the party.

Coco wants to maximize the guests' enjoyment at the party, so she must decide how to accommodate them.

Input

A number N, followed by n numbers that represent the level of happiness of each guest.

Output

The max value of enjoyment.

Sample 1:

Input:

3
2 2 10

Output:

16

Sample 2:

Input:

3
10 3 4

Output:

13

Code Resolution:

```
package vjudge;

import java.util.Arrays;
import java.util.Scanner;

public class TematicParty {

    public static int maximizeEnjoyment(int n, int[] happiness) {
        Arrays.sort(happiness);

        int[] sortedArray = new int[n];
        int left = 0, right = n - 1, maxEnjoyment = 0;;

        for (int i = 0; i < n; i++) {
            if (i % 2 == 0) {
                sortedArray[i] = happiness[left];
                left++;
            } else {
                sortedArray[i] = happiness[right];
                right--;
            }
        }
    }
}
```

```

    for (int i = 1; i < n; i++) {
        int enjoyment = Math.abs(sortedArray[i] - sortedArray[i - 1]);
        maxEnjoyment += enjoyment;
    }
    return maxEnjoyment;
}

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);

    int N = scanner.nextInt();
    int[] happinessLevels = new int[N];
    for (int i = 0; i < N; i++) {
        happinessLevels[i] = scanner.nextInt();
    }

    int result = maximizeEnjoyment(N, happinessLevels);
    System.out.println(result);
}
}

```

Logic Code:

The logic of the code addresses the problem of maximizing enjoyment at a themed party by arranging guests strategically. After sorting the happiness levels of the guests, a new arrangement is created where guests are alternated from the left and right ends of the original arrangement. This aims to place the happiest guests in the center to maximize overall enjoyment. The solution is evaluated by calculating the sum of the absolute differences between the consecutive happiness levels in the new arrangement. Finally, the result, which represents the maximum possible enjoyment at the theme party, is printed.

Questionnaire:

- Identify the greedy choice.

Greedy case:

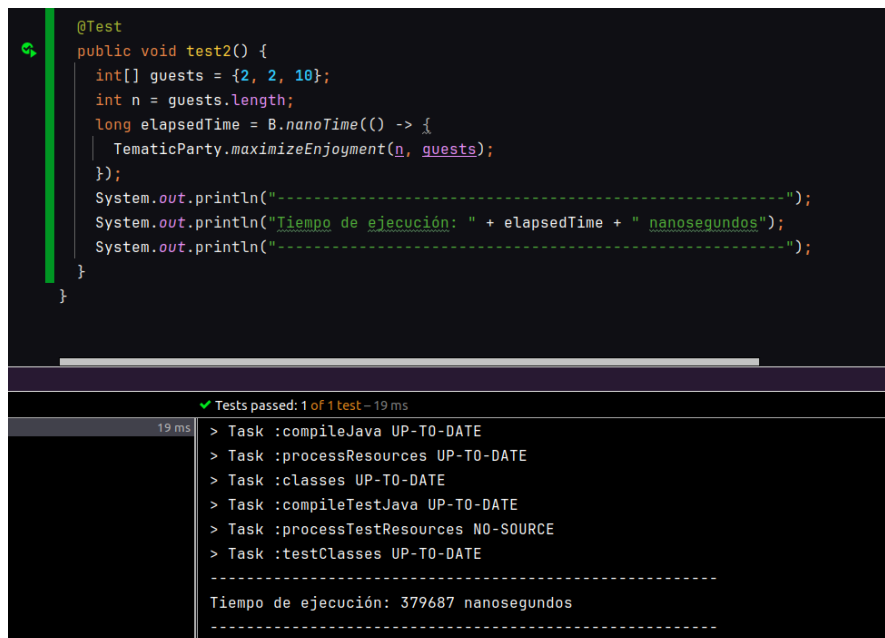
It occurs by filling the "sortedArray" by alternately choosing guests from the left and right ends of the array sorted by happiness. This approach prioritizes placing guests with higher happiness levels toward the center, potentially maximizing enjoyment at the party. The decision to alternate between the left and right ends is the voracious choice.

- Identify the optimal substructure.

The optimal substructure for maximizing enjoyment at a theme party is in the calculation of total enjoyment. After sorting the happiness levels and arranging the guests in the sortedArray, the code iterates through this sorted array and calculates the total enjoyment by summing the absolute differences between consecutive happiness levels. This process inherently exhibits an optimal substructure because the enjoyment of the entire array is composed of optimal enjoyments within smaller subproblems. The total enjoyment can be constructed by combining optimal solutions to subproblems, specifically the enjoyment between pairs of adjacent guests in the ordered array. Therefore, the optimal substructure is reflected in the recursive nature of the enjoyment computation along the sortedArray.

- Implement your solution.

It has already been successfully implemented.



```

@Test
public void test2() {
    int[] guests = {2, 2, 10};
    int n = guests.length;
    long elapsedTime = B.nanoTime() -> {
        TematicParty.maximizeEnjoyment(n, guests);
    };
    System.out.println("-----");
    System.out.println("Tiempo de ejecución: " + elapsedTime + " nanosegundos");
    System.out.println("-----");
}
}

```

✓ Tests passed: 1 of 1 test - 19 ms

```

> Task :compileJava UP-TO-DATE
> Task :processResources UP-TO-DATE
> Task :classes UP-TO-DATE
> Task :compileTestJava UP-TO-DATE
> Task :processTestResources NO-SOURCE
> Task :testClasses UP-TO-DATE
-----
Tiempo de ejecución: 379687 nanosegundos
-----

```

- What is the time complexity of your solution?

The complexity of the problem is:

- Data Input: $O(t)$ - t is a constant (It is the number of test cases).
- Sorting: Sort with a sort that has a complexity of $O(n \log n)$.
- Creation of sortedArray: Iterate over sortedArray therefore $O(n)$
- Calculation of Total Enjoyment: Iterate over sortedArray so it is $O(n)$
- Data output: $O(t)$ - t is a constant (It is the number of test cases).

The complexity could be expressed as:

- $O(t * (n \log n))$ or in a simplified way $O(n \log n)$ the worst case, and the best case $O(n \log n)$

CocosBirthday.pdf

Analyze the problem:

Coco is organizing a birthday party and has invited n of his friends.

It is the day of the party, and all the guests are in line; each guest takes one minute to enter; the first to enter enters at minute 1, the second to enter enters at minute 2, and so on. Her guests have a happiness x that decreases every minute they do not enter the party. Coco knows that guests will leave the party if they have negative happiness. So she wants to maximize the number of people that join the party.

Coco notices that sometimes it is convenient for people to enter the party in a different order than in which they are in line. However, if at any time t , m other guests enter the party before the guest in the front row, the guest in the front row will decrease their happiness by m additional points.

Help Coco maximize the number of people that enter her party.

Input

The first line has N . The following line has n numbers that represent their happiness x .

Output

An integer that represents the number of people that joins the party.

Sample 1:

Input:

4
2 2 3 3

Output:

3

Sample 2:

Input:

4
7 5 3 1

Output:

4

Code Resolution:

```
package vjudge;

import java.util.Arrays;
import java.util.Scanner;
```

```

public class CocoBirthday {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        int n = scanner.nextInt();
        int[] happiness = new int[n];

        for (int i = 0; i < n; i++) {
            happiness[i] = scanner.nextInt();
        }

        int quantityMaxGuest = Guests(n, happiness);
        System.out.println(quantityMaxGuest);
    }

    public static int Guests(int n, int[] happiness) {
        Arrays.sort(happiness);
        int maxGuests = 0;
        for (int i = 0; i < n; i++) {
            if (happiness[i] - i > 0) {
                maxGuests++;
            } else {
                break;
            }
        }
        return maxGuests;
    }
}

```

Logic Code:

The code logic for Coco to maximize participation in her birthday party by considering the decreasing happiness level of friends who have not yet joined.

After sorting the happiness levels of friends, the code iterates over the resulting arrangement and determines the maximum number of people who can join without experiencing negative happiness levels. The logic is based on the premise that friends can enter in ascending order of happiness, as long as the happiness level minus the index is not negative. The result, represented by the variable max Guests, is printed as the solution to the problem, indicating the optimal number of friends Coco can have in her party without any of them having a negative happiness level.

Questionnaire:

- Identify the greedy choice.

Greedy case:

The greedy choice lies in the process of determining which friends can enter the party in a way that maximizes the number of attendees. The key greedy decision is produced by iterating through the ordered array of happiness levels. For each friend, the code checks whether his happiness level minus his index in the ordered matrix is greater than zero. If true, indicating that the friend's happiness is sufficient to withstand the wait, he is considered eligible to enter. This greedy choice gives priority to inviting friends with higher happiness levels earlier in the sorting order, with the goal of maximizing the total number of attendees by accommodating those who can tolerate a longer wait.order, aiming to maximize the overall number of attendees by accommodating those who can tolerate waiting longer.

- Identify the optimal substructure.

The optimal substructure is in the calculation of the maximum number of guests that can join the party without experiencing negative happiness levels. This is set in the Guests method where it iterates over the ordered array of happiness levels. The optimal substructure lies in the fact that the maximum number of guests for the entire party is built upon the optimal solution of the subproblems, specifically the decision of whether or not each friend with a given happiness level and position in the ordered array can join. The recursive nature of the algorithm and the dependence on the optimal solution of the subproblems characterize the optimal substructure.

- Implement your solution.

It has already been successfully implemented.

```
@Test
public void test() {
    int[] happiness = {2, 2, 3, 3};
    int n = happiness.length;
    long elapsedTime = B.nanoTime() -> {
        CocoBirthday.Guests(n, happiness);
    };
    System.out.println("-----");
    System.out.println("Tiempo de ejecución: " + elapsedTime + " nanosegundos");
    System.out.println("-----");
}

@Test
public void test2() {
    // ...
}
```

✓ Tests passed: 1 of 1 test - 23 ms

23 ms

```
> Task :compileJava UP-TO-DATE
> Task :processResources UP-TO-DATE
> Task :classes UP-TO-DATE
> Task :compileTestJava
> Task :processTestResources NO-SOURCE
> Task :testClasses

-----
Tiempo de ejecución: 430766 nanosegundos
-----
```

- What is the time complexity of your solution?

The complexity of the problem is:

- Data Input: $O(t)$ - t is a constant (It is the number of test cases).
- Sorting: Sort with a sort that has a complexity of $O(n \log n)$.
- Data output: $O(t)$ - t is a constant (It is the number of test cases).

The complexity could be expressed as:

- $O(t * (n \log n))$ or in a simplified way $O(n \log n)$ the worst case, and the best case $O(n \log n)$

