1. In this question, depending on which city we start, optimum cost will be different We calculate two situation whether the start point NY or SF.

When both calculations are finished, we'll find the minimum cost.

Let's say we are in NY in nth month. There are 2 options for optimum cost:

→ Current month's cost + Optimal cost ending in NY for n-1 month

→ Current month's cost + Optimal cost ending in SF for n-1 month + M

We calculate these operations from first month to nth month.

So we have 2 optimum costs. Minimum of them is the result

Worst case running time → O(n) n: number of months

2- We can clearly see that it's an Activity Selection Problem. The greedy approach always pick the an activity start time is bigger than or equal to previous activity's finish time.

- We can sort the activities by their finish time.
- So, we accept the next activity as minimum finishing time activity.
- Also, we add a counter to find maximum number of sessions.

Algorithm

1- We sort the activities by their finish times.
2 We select the first activity.
3. We pick the activity that is the time is bigger than or equal to the previous activity's finish time.
4- We calculate maximum number of sessions by a counter.

Worst-case running time → $\Theta(n\log n)$
It occurs when input activities are not sorted.

3- In this problem first of all we need to seperate positive integers to an array and negative integers to an array. Also we need to create an array that contains non-zero elements. Then we check if there is exist a subset summing to the targetSum, in this case it's zero, and including the index, one of three following must hold:

→ if Index == targetSum, Index is a subset that we want
→ there is a solution using only $k_1 \ldots k_{\{Index-1\}}$
→ there is a solution by adding k_{Index} to problem's solution results in a solution to the current problem.

Worst case running time → $\Theta(n)$
(Because $T(n) = T(n-1) + 1$)

5- In this problem, we need to find sum of the array. Because sum of the array is equal to the minimum number of operations. But there is one tricky part, if any element is less than zero, we take the absolute value, then calculate.

For example:

arr[ ] = { 2, 3, 7, 5 } → operation number: $2+3+7+5=17$

arr[ ] = { 2, 3, -2, -3 } → operation number: $2+3+|-2|+|-3|=10$

Worst case running time: $O(n)$

4- First of all, we create a matrix that all elements are zero. Then we fill the matrix in the right order. We put the maximum value to the array, by recurrence. We track the cell and find the cost. Cost is the largest score in our array.

Worst case running time: $O(n^2)$

Because we use 2 for (inner)