**Mohit and his Sum**

Mohit is preparing for campus placements. So, this week, he started to solve the problems related to arrays. But some time ago, he solved a problem related to minimum contiguous subarray sum X and maximum contiguous subarray sum Y. But he forgot the method of how to calculate X and Y. So, he gives you an array and tells you to find X and Y for the given array.

So, can you help him ?

**Input Format**

* First line contains the number of testcases T
* Next T lines, each line contains N and on the next line N space separated integers Ai.

**Constraints**

* 1 <= T <= 100
* 1 <= N <= 50
* -100000 <= Ai <= 100000

**Output Format**

* Print X and Y separated by spaces.

**Sample Input 0**

2

9

-2 1 -3 4 -1 2 1 -5 4

7

3 -4 2 -3 -1 7 -5

**Sample Output 0**

-5 6

-6 7

**Explanation 0**

1. Testcase 1: The contiguous subarray (-5) has the minimum sum and subarray (4, -1, 2, 1) has the maximum sum as 6.

**Solution:**

How to find the subarray with the largest sum in a given array? This is a repeated common question one might encounter while solving CP problems. Kadane’s algorithm is used to solve this problem. Before we reach into the implementation, we should try to solve this with a brute force approach.

1. **Naive approach:**

The most basic thing we can do is, start with each index of the array and keep a track of sums. Then, we compare the sums that we have obtained and return the largest subarray.

1. Generate a list of all subarrays.
2. Generate sums for all subarrays.
3. Return subarray with largest sum.

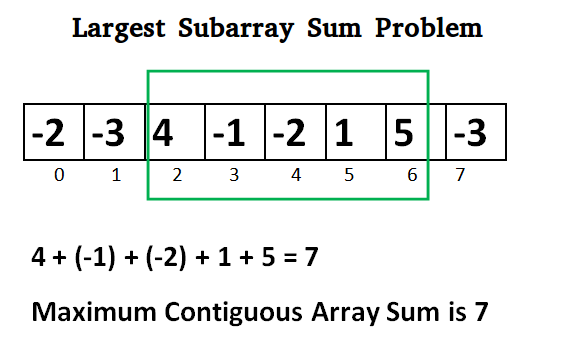
The time complexity for this approach is O(n²). This is a very inefficient solution.

Hence, this problem can be solved using Kandane’s Algorithm. This algorithm is used to find the sum of contiguous subarray within a one-dimensional array of numbers that has the largest sum.

When we start out, we assume that the subarray starts from the beginning of given array and current sum is 0. For each element in the array, we then check the following:

1. If the current element, added to the current sum, yields a value which is *actually* lesser than the element itself, then there’s no point in considering the previous sum. We might as well start the largest subarray from this element onwards and continue in the provided fashion.
2. If the current element, added to the current sum, yields a value greater than the item itself, then we should cross check it with the largest sum that we have found so far. If the sum exceeds the largest sum, then the subarray starts somewhere and ends at this element. Otherwise, we just continue.

What this means is that, if adding a sum to the given index’s element is lesser than the element, then it must mean that the sum is negative, and the item is positive or negative, but it is larger than the current sum. Since we are after the largest sum, we can discard our current progress and continue!



In the above figure, we can see, starting with -2, the largest sum is -2. If we add -3 to the sum, then we get -5, which is smaller than -3, so we discard -2 and start our subarray from -3. Then we add 4, which gives us value 1, which is still, smaller than 4. So, we discard -3 and start subarray from 4. Then we add -1. We get 3, which is larger than -1, so we check the sum(3) against the largest sum recorded so far, which is 1. As 3 > 1, we mark this as the end of the subarray. So far, the maximal sum subarray contains [4, -1]. Then we add -2. As 3 + -2 = 1, which is greater than -2, but lower than max recorded sum(3), we will not mark our array to end here. So far, the maximal subarray contains [4, -1]. Then we reach 1. Still, we won’t mark the end of the subarray. Then we reach 5. At this point, the sum becomes 7, which is greater than 3. So, our maximal subarray ends at this point so far. Then we consider -3. Adding it to 7 gives us 4, which is lesser than the sum recorded so far.

**Kadane’s Algorithm:**

Initialize:

max\_so\_far = INT\_MIN

max\_ending\_here = 0

Loop for each element of the array

(a) max\_ending\_here = max\_ending\_here + a[i]

(b) if(max\_so\_far < max\_ending\_here)

max\_so\_far = max\_ending\_here

(c) if(max\_ending\_here < 0)

max\_ending\_here = 0

return max\_so\_far

The simple idea of Kadane’s algorithm is to look for all positive contiguous segments of the array (max\_ending\_here is used for this). And keep track of maximum sum contiguous segment among all positive segments (max\_so\_far is used for this). Each time we get a positive-sum compare it with max\_so\_far and update max\_so\_far if it is greater than max\_so\_far.

See: <https://www.geeksforgeeks.org/largest-sum-contiguous-subarray/> for example.

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Java Solution:**

import java.util.\*;

public class Subarray

{

public static int Kadane(int a[], int n)

{

int max=Integer.MIN\_VALUE, mend=0;

for(int i=0;i<n;i++)

{

mend=mend+a[i];

if(max<mend)

max=mend;

if(mend<0)

mend=0;

}

return max;

}

public static int Minimum(int a[], int n)

{

int min=Integer.MAX\_VALUE, mend=Integer.MAX\_VALUE;

for(int i=0;i<n;i++)

{

if(mend>0)

mend=a[i];

else

mend=mend+a[i];

min=Math.min(min,mend);

}

return min;

}

public static void main (String[] args)

{

Scanner sc=new Scanner(System.in);

int t=sc.nextInt();

int i=0, j=0;

for(i=0;i<t;i++)

{

int n=sc.nextInt();

int a[]=new int[n];

for(j=0;j<n;j++)

a[j]=sc.nextInt();

int x=Minimum(a,n);

int y=Kadane(a,n);

System.out.println(x+" "+y);

}

}

} **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Python Solution:**

t = int(input())

for i in range(t):

n = int(input())

mylist = list(map(int, input().split()))

maxsum = currentsum = mylist[0]

for i in range(1,n):

currentsum = max(currentsum + mylist[i], mylist[i])

maxsum = max(maxsum, currentsum)

minsum = currentmin = mylist[0]

for i in range(1,n):

currentmin = min(currentmin + mylist[i], mylist[i])

minsum = min(minsum, currentmin)

print(minsum, maxsum)

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**C++ Solution:**

#include <cmath>

#include <cstdio>

#include <vector>

#include <iostream>

#include <algorithm>

#include<bits/stdc++.h>

using namespace std;

int main() {

/\* Enter your code here. Read input from STDIN. Print output to STDOUT \*/

int t;

cin>>t;

while(t--){

int n;

cin>>n;

int a[n];

for(int i=0;i<n;i++){

cin>>a[i];

}

int maxsofar = INT\_MIN;

int maxendhere = 0;

int minsofar = INT\_MAX;

int minendhere = INT\_MAX;

for(int i=0;i<n;i++){

maxendhere += a[i];

maxsofar = max(maxsofar , maxendhere);

maxendhere = max(maxendhere , 0);

if(minendhere > 0){

minendhere = a[i];

}

else{

minendhere += a[i];

}

minsofar = min(minsofar , minendhere);

}

cout<<minsofar<<" "<<maxsofar<<"\n";

}

return 0;

}