By : Prateek Bhaiya and Rajkishor

Greedy Algorithms

Greedy Algorithms are one of the most intuitive algorithms. Whenever we see a problem we first try to apply some greedy strategy to get the answer(we humans are greedy, aren't we :P ?).

Greedy approaches are quite simple and easy to understand/formulate.But many times the proving part might be difficult.

- Greedy Algorithm always makes the choice that looks best at **the moment.**
- You hope that by choosing a local optimum at each step, you will end up at a global optimum.

Counting Money

Problem Statement: Suppose you want to count out a certain amount of money,using the fewest possible notes or coins.

Greedy Approach:

At each step, take the largest possible note or coin that does not overshoot the sum of money and include it in the solution.

Example:

To make Rs 39 with fewest possible notes or coins.

Rs 10 note, to make 29

Rs 10 note, to make 19

Rs 10 note, to make 9

Rs 5 note, to make 4

Rs 2 coin, to make 2

Rs 2 coin, to make 0

Total is 4 notes and 2 coins, which is the optimum solution.

Note: For Indian currency, the greedy algorithm, even after Demonetization always gives the optimum solution.

Is this true for any currency?

Now that Donald Trump is the new President of US, he decides to do something extraordinary like PM Modi. So he decides to change all the currency notes to 1 dollars, 7 dollars and 10 dollars.

Suppose you went to US and used the same greedy approach to count minimum numbers of notes and coins in exchange for a Burger which costs $15.

To make $15:

1 $10 note

5 $1 notes

Total = 6 notes

Can we do better than 6?

$7 + $7 +$1 – Only 3 notes required!

Reason?

Ques. BusyMan, Activity Scheduling Problem

http://www.spoj.com/problems/BUSYMAN/

```
#include<iostream>
 #include<algorithm>
 #include<vector>
using namespace std;
struct cmpr{
bool operator()(pair<int,int> l, pair<int,int> r)
 {
  return l.second < r.second;
 }
}cmp;
int main()
{
int t,n,s,e,res;
scanf("%d",&t);
while(t--)
{
 res = 1;
 vector<pair<int,int> > activity;
 scanf("%d",&n);
 for(int i = 0;i<n;i++)
 {
 scanf("%d %d",&s,&e);
  activity.push_back(make_pair(s,e));
 }
 //Sort the activities according to their finish time
 sort(activity.begin(),activity.end(),cmp);
 //fin denotes the finish time of the choosen activity
 //i.e. activity with least finish time
 int fin = activity[0].second;
 for(int i = 1;i<activity.size();i++)
 {
 //To find the next compatible activity with
 //least finish time
 //Find the first activity whose start time
 //is more than finish time of previous activity
  if(activity[i].first >= fin)
  {
   //update the finish time as the finish time
   //of this activity
```

```
    fin = activity[i].second;
   //Since we have choosen a new activity,
   //increment the count by 1
   res++;
  }
 }
 printf("%d\n",res);
}
return 0;
}
```

Connecting Wires

- There are n white dots and n black dots, equally spaced, in a line.
- You want to connect each white dot with some one black dot, with a minimum total length of “wire”.

Greedy Approach: Suppose you have a sorted list of the white dot positions and the black dot positions. Then you should match the i-th white dot with the i-th black dot.

Why should greedy work?

Let b0 and w0 be the first black and white balls respectively and let bi

and wj be the next white and black balls.

Case 1: w0 and wj < b0

```
                   b0                bi
     w0   wj
```

(i) C(w0, bi) + C(wj, b0) = (bi - w0) + (b0 - wj)

(ii) C(w0, b0) + C(wi, bj) = (b0 - w0) + (bi - wj)

Both are same.

Case 2: w0 < b0 and b0 < wj < bi

```
             b0               bi
     w0               wj
```

(i) C(w0, bi) + C(wj, b0) = (bi - w0) + (wj - b0)

(ii) C(w0, b0) + C(wi, bj) = (b0 - w0) + (bi - wj)

(i) - (ii) = 2(wj - b0) is extra

Case 3: w0 < b0 and bi < wj

```
             b0               bi
     w0                               wj
```

(i) C(w0, bi) + C(wj, b0) = (bi - w0) + (wj - b0)

(ii) C(w0, b0) + C(wi, bj) = (b0 - w0) + (wj - bi)

(i) - (ii) = 2(bi - b0) is extra

Hence, in all the cases connecting 1st b to 1st w and 2nd b to 2nd w yeilds more optimal solution.

Ques: Biased Standings

In a competition, each team is be able to enter their preferred place in the ranklist. Suppose that we already have a ranklist. For each team, compute the distance between their preferred place and their place in the ranklist. The sum of these distances will be called the badness of this ranklist. Find one ranklist with the minimal possible badness.

http://www.spoj.com/problems/BAISED/

This is question is similar to the connecting wires problem, where the dseired ranks denote the black balls and the actual ranks denote the white balls. Hence, we can apply the greedy approach by sorting the desired ranks and calculating the distance between i-th desired rank and i-th actula rank.

Code :

```
#include <bits/stdc++.h>
using namespace std;
#define abs(a,b) a > b ? a - b : b - a
#define ll long long
int main()
{
int t;
cin>>t;
while(t--)
 {
  vector<int> v;
  string name;
  ll sum=0;
  cin>>n;
  for(int i=0;i<n;i++)
  {
   cin>>name;
   cin>>temp;
   //Insert the desired rank in vector v
   v.push_back(temp);
  }
  //Sort the vector containing the ranks
  sort(v.begin(),v.end());
  //This is the greedy approach
  for(int i=1;i<=n;i++)
  {
  //Take the difference between the i-th actual rank
  //and the i-th desired rank
  sum += abs(v[i-1],i);
  }
  printf("%lld\n",sum);
 }
 return 0;
 }
```

T = O(nlogn)

Can we do better?

```
#include <bits/stdc++.h>
 using namespace std;
 #define abs(a,b) a > b ? a - b : b - a
 #define ll long long
 int arr[100000+10];
 int main()
 {
 int t,n,temp;
 cin>>t;
 while(t--)
  {
    memset(arr,0,sizeof arr);
    string name;
    ll sum=0;
    cin>>n;
    for(int i=0;i<n;i++)
    {
     cin>>name;
     cin>>temp;
     //Increment the desired rank index in array arr
     arr[temp]++;
    }
   int pos = 1;
  //This is the greedy approach
  for(int i=1;i<=n;i++)
  {
 //If the i-th rank was desired by atleast one team
 //Assign and increment the actual rank index 'pos'
 //till this rank is desired
  while(arr[i])
  {
   sum += abs(pos,i);
   arr[i]--;
   pos++;
  }
 }
  printf("%lld\n",sum);
 }
  return 0;
 }
```

T = O(n)

Ques. Load Balancer

Rebalancing proceeds in rounds. In each round, every processor can transfer at most one job to each of its neighbors on the bus. Neighbors of the processor i are the processors i-1 and i+1 (processors 1 and N have only one neighbor each, 2 and N-1 respectively). The goal of rebalancing is to achieve that all processors have the same number of jobs.Determine the minimal number of rounds needed to achieve the state when every processor has the same number of jobs, or to determine that such rebalancing is not possible.

http://www.spoj.com/problems/BALIFE/

Greedy Approach: First find the final load that each processor will have. We can find it by sum(arr[1], arr[2],...,arr[n])/n, let us call it **load.**

So in the final state, each of the processor will have final load = load.

For each index i from 1 to n-1, create a partition of (1...i) and (i+1...n) and find the amount of load that is to be shared between these two partitions. The answer will be maximum of all the loads shared between all the partitions with i varying from 1 to n-1.

Example

```
Initial state: 4, 8, 12, 16
Final state: 10 10 10 10
i = 1:
partition: (4) (8,12,16)
Load --> (4) needs 6 and (8,12,16) needs to give 6.
max_load = 6
i = 2:
partition: (4,8) (12,16)
Load--> (4,8) needs 8 and (12,16) needs to give 8.
max_laod = 8
i = 3:
partition: (4,8,12) (16)
Load--> (4,8,12) needs 6 and (16) needs to give 6.
max_laod = 6
Final answer = 8
```

Why does it works?

In all the partitions where less than max_load is transferred, we can internally transfer the load between these partitions when max_laod is being transferred between the max_load partition.

```
t = 2s:
when load = 2 is transferred between (4,8) and (12,16)
we can transfer load = 2 between (8) --> (4) and (16)--> (12)
State: 6 8 12 14
t = 4s:
when load = 2 is transferred between (6,8) and (12,14)
we can transfer load = 2 between (8) --> (6) and (14) --> (12)
State: 8 8 12 12
t = 6s:
when load = 2 is transferred between (8,8) and (12,12)
State: 8 10 10 12
t = 8s:
when load = 2 is transferred between (8,10) and (10,12)
we can transfer load = 2 between (10) --> (8) and (12) --> (10)
State: 10 10 10 10
```

Code

```
#include <bits/stdc++.h>

using namespace std;
int main()
{
int arr[9000],n, i, val, diff;
while(1)
 {
```

```
 int max_load = 0, load = 0;
 cin>>n;
 if(n == -1)
  break;
 for(int i = 0;i<n;i++)
 {
  cin>>arr[i];
  load += arr[i];
 }
 //If we cannot divide the load equally
if(load % n)
{
 cout<<-1<<endl;
 continue;
}
//Find the load that is to be divided equally
load /= n;
//Greedy step
for(int i = 0;i<n;i++)
{
 //At each iteration, find the value
 //of difference between final load to be assigned
 //and current load
 //Keep adding this difference in 'diff'
 diff += (arr[i] - load);
 //If the net difference is negative i.e.
 //we need diff amount till i-th index
 if(diff < 0)
   max_load = max(max_load, -1*diff);
 //If diff is positive i.e. we have to
 //give diff amout to (n-i) processors
 else
```

```
     max_load = max(max_load, diff);
    //calculate the max of load that can be given or
     //taken at each iteration .
    }
    cout<<max_load<<endl;
   }
 return 0;
 }
```

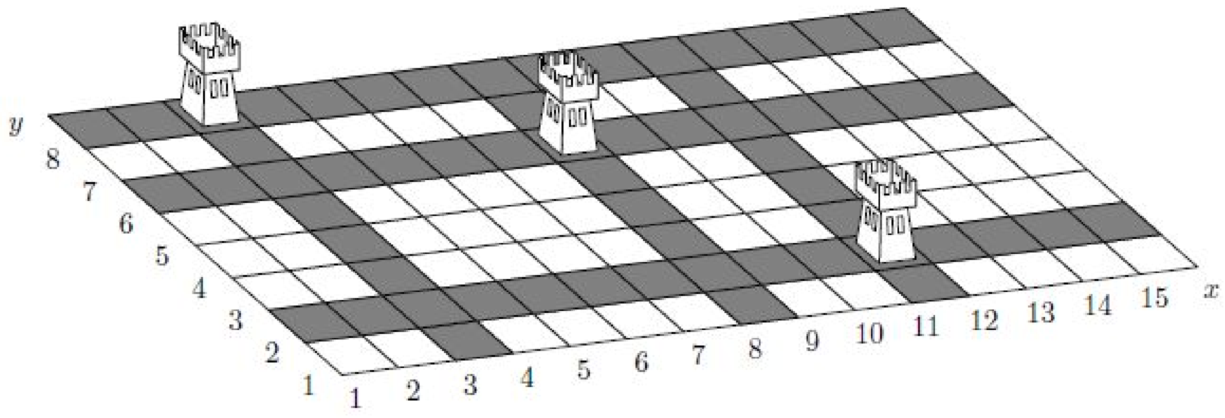
T = O(n)

Ques. Defense of a Kingdom, Spoj

Problem Statement

Given H,W of a field, and location of towers which guard the horizontal and the vertical lines corresponding to their positions. Find out the largest unbounded area(white rect). Refer the diagram.

http://www.spoj.com/problems/DEFKIN/



Greedy Approach: Given w, h as width and height of the playing field, and the coordinates of the towers as (x1,y1) ... (xN,yN), split the coordinates into two lists x1...xN, y1...yN, sort both of those coordinate lists.

Then calculate the empty spaces, e.g. dx[] = { x1, x2-x1, ..., xN - xN-1, (w + 1) - xN }. Do the same for the y coordinates: dy[] = { y1, y2-y1,..., yN - yN-1, (h + 1) - yN }. Multiply max(dx)-1 by max(dy)-1 and you should have the largest uncovered rectangle. You have to decrement the delta values by one because the line covered by the higher coordinate tower is included in it, but it is not uncovered.

Code :

```
#include<iostream>
 #include<algorithm>
 using namespace std;
 int point_x[40000+10], point_y[40000+10];
 int main()
 {
  int t,w,h,n,x,y;
  scanf("%d",&t);
  while(t--)
  {
   scanf("%d %d %d",&w,&h,&n);
   for(int i = 0;i<n;i++)
   {
   scanf("%d %d",&point_x[i],&point_y[i]);
   }
   //sort the x-coordinates of the list
   sort(point_x, point_x + n);
   //sort the y-coordinates of the list
   sort(point_y, point_y + n);
   //dx --> maximum uncovered tiles in x coordinate
   //dy --> maximum uncocered tiles in y coordinate
   //Initially dx and dy are the first guars's position
   int dx = point_x[0],dy = point_y[0];
  //calculate the maximum uncovered gap
   //in x and y coordinate
   for(int i = 1;i<n;i++)
   {
    dx = max(dx,point_x[i] - point_x[i-1]);
    dy = max(dy,point_y[i] - point_y[i-1]);
   }
   dx = max(dx, w + 1 - point_x[n-1]);
   dy = max(dy, h + 1 - point_y[n-1]);
   printf("%d\n",((dx-1) * (dy-1)));
  }
  return 0;
 }
```

T = O(nlogn)

Ques. Chopsticks

Given N sticks of length L[1], L[2], ..., L[N] and a positive integer D. Two sticks can be paired if the difference of their length is at most D.

Find the max number of pairs.

https://www.codechef.com/problems/TACHSTCK

Greedy Appraoch:

- Sort the list of sticks according to their lengths.
- If L[1] and L[2] cannot be paired, L[1] is useless.
- Else pair L[1] and L[2] and remove them from the list.
- Repeat steps 2-3 till the list become empty.

Why does this works?

- By pairing starting stick to its immediate next, we have max number of options left for next pairing.
- If L[1] and L[2] can be paired, but instead we pair (L[1], L[M]) and (L[2], L[N]).

Case 1: L[N] < L[M]

```
    L[1]                L[2]          L[N]              L[M]
```

```

L[M] - L[1] <= D
L[N] - L[2] <= D

L[M] - L[1] = (L[M] - L[N]) + (L[N] - L[1])
(L[M] - L[N]) + (+ve number) <= D
L[M] - L[N] <= D

Hence, we can also pair L[N] and L[M]
```

Case 2: L[N] > L[M]

```
L[1]        L[2]             L[M]             L[N]

L[N] - L[2] <= D
L[M] - L[1] <= D

L[N] - L[2] = (L[N] - L[M]) + (L[M] - L[1]) <= D
L[N] - L[M] <= D

Hence, L[M] and L[N] can be paired
```

Code :

```
#include <iostream>
 #include <vector>
 #include <algorithm>
using namespace std;
typedef long long ll;
int main()
{
ll n,d,a;
vector<ll> v;
cin>>n>>d;
for(int i = 0;i<n;i++)
{
 cin>>a;
 v.push_back(a);
}
//Sort the list of sticks
sort(v.begin(),v.end());
int res = 0;
for(int i = 0;i<n-1;)
{
 //If the adjacent difference is less than d
 //we can make a pair
 //Now remove these two sticks from the list
 if(v[i+1] - v[i] <= d)
 {
  res++;
  i+=2;
 }
 //If we cannot make a pair with adjacent stick
 //This stick is useless, remove this stick from the list
 else
   i++;
}
cout<<res<<endl;
return 0;
}
```

T = O(nlogn)

Ques. Expedition, Spoj

A damaged truck needs to travel a distance of L. The truck leaks one unit of fuel for every unit distance it travels.There are N (<=10^4)fuel stops on the path, what is the minimum number of refueling stops that the truck needs to make given that it has P amout of fuel initially.

http://www.spoj.com/problems/EXPEDI/

The first observation is, if you want to reach the city, say, point L, you **have to ensure that every single point between the current position and city must also be reachable.**

Now, the task is to minimize the number of stoppages for fuel, which is at most 10000. So, we sort the fuel stations, and start from current **position. For every fuel station, if we want to reach it, we must have fuel f more than or equal to the**

distance d. Also, using the larger capacities will always reduce the number of stations we must stop.

How does greedy work?

If we make sure that we only stop at the largest capicity fuel stations

upto the point where the fuel capacity of truck becomes 0, number of

stops will be minimized.

Example:

Initial capacity = 11

Fuel Stations: 3 8 10 12

Capacity: 4 10 2 3

Initially, we don't have enough fuel to reach 12.

So we can reach upto maximum 11th city.

In order to minimize our stops, we will stop at 8th city

where fuel capacity is maximum.

Code:

```
#include<iostream>
#include<vector>
#include<queue>
#include<algorithm>
using namespace std;
bool cmpr(pair<int,int> l,pair<int,int> r)
{
return l.first > r.first;
}
int main()
{
int n,t,x,d,f,D,F,prev = 0;
scanf("%d",&t);
while(t--)
```

```
{
int flag = 0,ans = 0;
vector<pair<int,int> > v;
priority_queue<int> pq;
scanf("%d",&n);
for(int i = 0;i<n;i++)
{
 scanf("%d %d",&d,&f);
 //Insert the city index and fuel capacity
 v.push_back(make_pair(d,f));
}
//Sort the cities according to their location
sort(v.begin(),v.end(),cmpr);
scanf("%d %d",&D,&F);
//Calculate the difference between the current city and the
//destination i.e. v[i] = j means that we need to travel j
//units to rach our destination
for(int i = 0;i<n;i++)
{
 v[i].first = D - v[i].first;
}
//prev denotes the previous city visited
 prev = 0;
//x will denote the current city that we are in
 x = 0;
while(x < n)
{
 //cout<<x<<" "<<F<<" "<<v[x].first<<" "<<prev<<" "<<endl;
 //If we have enough fuel to travel from prev to current city
 //Push this fuel station to priority_queue
 //Reduce the amount of fuel used
 //update the previous city
 if(F >= (v[x].first - prev))
 {
   F -= (v[x].first - prev);
  pq.push(v[x].second);
  prev = v[x].first;
 }
 //If we dont have enough fuel to visit
 //the current city
```

```
//Find the max capacity fuel station between
//prev and current city and use it to refuel
else
{
 //If no fuel station is left
 //i.e. we have used all of
//the fuel stations and still not able
//to reach the city
//return FAIL!
if(pq.empty())
{
 flag = 1;
 break;
}
//Increment the fuel capacity of truck
 //by the maximum fuel station capacity
 F += pq.top();
//Remove that fuel station from heap
 pq.pop();
//Increment the number of used fuel
//station by 1
  ans++;
  continue;
}
//If we have visited the current city
 //Visit next city
x++;
}
if(flag){
 printf("-1\n");
 continue;
}
//Find the distance between the destination
//and last city
//Check if it is possible to visit the destination
//from the last city.
D = D - v[n-1].first;
if(F >= D)
{
 printf("%d\n",ans);
 continue;
}
while(F < D)
{
if(pq.empty()){
  flag = 1;
  break;
  }
 F += pq.top();
pq.pop();
ans++;
}
if(flag){
printf("-1\n");
continue;
 }
 printf("%d\n",ans);
}
return 0;
}
```

T = O(NlogN)

Ques. Greedy Knapsack Problem, Codechef

You are given N items, each item has two parameters: the weight and the cost. Let's denote M as the sum of the weights of all the items. Your task is to determine the most expensive cost of the knapsack, for every capacity 1, 2, ..., M. The capacity C of a knapsack means that the sum of weights of the chosen items can't exceed C.

https://www.codechef.com/problems/KNPSK

The key observation here is that weight of each item is either 1 or 2.

Greedy Approach: Greedily pickup the most costliest item with weight <= 2, which can be taken in the knapsack.

Case 1: W is even

Select the most expensive item with sum of weight = 2. This can be done in two ways:

- Take the most expensive item of weight 2
- Or take at most two most expensive item of weight 1

Note that after picking up the most expensive item with sum of weight <= 2, we will remove the item taken and will recursively select the items to fill the most expensive elements.

Case 2: W is odd

We can simply select the most expensive item of weight 1. Now, we don't consider this item again. Now we have to select the most expensive weights from the remaining items. Since W-1 is even, we can solve this problem similar to case 1.

Example:

1 7 1 100 2 70

2 44 1 56 2 44

1 56 1 33 2 1

2 1 1 18

1 18

1 33

2 70

Case 1: Even

W = 2: (1,100) + (1,56) = 156

W = 4: 156 + (2,70) = 226

W = 6: 226 + (1,33) + (1,18) = 277

W = 8: 277 + (2,44) = 321

W = 10: 321 + (2,1) = 322

Case 2: Odd

W = 1: (1,100) = 100

W = 3: 100 + (1,56) + (1,33) = 189

W = 5: 189 + (2,70) = 259

W = 7: 259 + (2,44) = 303

W = 9: 321 + (1,18) = 321

Code

```
#include<iostream>
 #include<vector>
 #include<algorithm>
 #include<stdio.h>
using namespace std;
long long cost[200000+10];
int main()
{
int n,w,c,m,W = 0;
//Separate one and two for even and odd casses
vector<int> one,two,One,Two;
scanf("%d",&n);
for(int i = 0;i<n;i++)
{
 scanf("%d %d",&w,&c);
 //Insert weight 1 items in one vector
 if(w == 1)
  one.push_back(c);
 //Insert weight 2 items in two vector
 else
   two.push_back(c);
 W += w;
}
 //sort the two vectors
 sort(one.begin(),one.end());
 sort(two.begin(),two.end());
One = one;
Two = two;
long long sum = 0,cur = 0,res = 0;
```

```
//even case
for(int i = 2;i<=W;i+=2)
{
 //res1 --> when we take atmost two 1 wight items
 //res2 --> when we take single 2 weight item
 long long res1 = 0, res2 = 0;
 //calc res2
 if(two.size() > 0)
 {
  res2 = two[two.size()-1];
 }
 //calculate res1
 if(one.size() > 1)
 {
  res1 = one[one.size()-1] + one[one.size()-2];
 }
 else if(one.size() > 0)
 {
  res1 = one[one.size() - 1];
 }
 //if res2 > res1 remove the 2 weight item
 if(res2 > res1)
 {
  two.pop_back();
  res += res2;
  }
 //remove at most two 1 weight items
 else
 {
  if(one.size() > 1)
  {
   one.pop_back();
   one.pop_back();
   }
   else
     one.pop_back();
  res += res1;
  }
 //update the max cost for weight i
 cost[i] = res;
 }
 //odd case
 //subtract 1 to make the weight sum even
 res = 0;
 //Find the most expensive 1 weight item
 //and remove it from the list
 if(One.size() > 0){
 res = One[One.size()-1];
 One.pop_back();
}
cost[1] = res;
//Similar to even case
for(int i = 3;i<=W;i+=2)
{
 long long res1 = 0, res2 = 0;
  if(Two.size() > 0)
  {
   res2 = Two[Two.size()-1];
  }
  if(One.size() > 1)
  {
   res1 = One[One.size()-1] + One[One.size()-2];
  }
  else if(One.size() > 0)
  {
   res1 = One[One.size()-1];
  }
  if(res2 > res1)
  {
   Two.pop_back();
   res += res2;
  }
  else
  {
   if(One.size() > 1)
   {
    One.pop_back();
    One.pop_back();
   }
   else
     One.pop_back();
   res += res1;
  }
 cost[i] = res;
}
for (int i = 1; i <= W; i++)
{
 if (i > 1) printf(" ");
 printf("%lld", cost[i]);
}
 printf("\n");
return 0;
}
```

T = O(nlogn)

PROBLEMS TO TRY

1) **Fractional Knapsack**

Given weights and values of n items, we need put these items in a knapsack of capacity W to get the maximum total value in the knapsack.We can break items for maximizing the total value of knapsack.

2) **Huffman Coding**

Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters. The most frequent character gets the smallest code and the least frequent character gets the largest code. Read about this problem and implement it yourself.

3) **Maximum Circles (HackerBlocks)**

There are *n* circles arranged on x-y plane. All of them have their centers on x-axis. You have to remove some of them, such that no two circles are overlapping after that. Find the minimum number of circles that need to be removed.

4) <u>Maximum Unique Segment(Codechef)</u>

You are given 2 arrays W = (W_1, W_2, .., W_N) and C = (C_1, C_2, .., C_N) with N elements each. A range [l, r] is *unique* if all the elements C_l, C_{l+1}, .., C_r are unique (ie. no duplicates). The *sum* of the range is $W_l + W_{l+1} + ... + W_r$

You want to find an *unique* range with the maximum *sum* possible, and output this sum.

<u>5) Station Balance - UVa 410</u>

Read and solve the problem statement Online

https://uva.onlinejudge.org/external/4/410.pdf