1.**Lily’s Problem**

The sum for each **O ≤ i N** of |Ai — Ai-l| I will be minimal if the array is sorted in either ascending or descending order. Both will yield the minimum sum. You have to sort in both ways and check for which way the number of swaps is smaller. The number of swaps = N - cycles.

C++

```
#include <iostream>
#include <stdio.h>
#include <algorithm>
#include <cmath>
#include <vector>
#include <queue>

using namespace std;

#define pb push_back
#define mp make_pair
#define F first
#define S second

const int N = 100500;

int n;
int a[N];
bool used[N];
pair<int, int> b[N], c[N];

void goB(int v) {
  if (used[v]) return;
  used[v] = true;
  goB(b[v].S);
}


void goC(int v) {
  if (used[v]) return;
  used[v] = true;
  goC(c[v].S);
}
```

```
int main() {
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        scanf("%d", &a[i]);
        b[i] = c[i] = mp(a[i], i);
    }
    sort(b + 1, b + 1 + n);
    sort(c + 1, c + 1 + n);
    reverse(c + 1, c + 1 + n);
    for (int i = 1; i <= n; i++) {
        used[i] = false;
    }
    int ans = 0;
    for (int i = 1; i <= n; i++) {
        if (!used[i]) {
            ++ans;
            goB(i);
        }
    }
    int result = ans;
    for (int i = 1; i <= n; i++) {
        used[i] = false;
    }
    ans = 0;
    for (int i = 1; i <= n; i++) {
        if (!used[i]) {
            ++ans;
            goC(i);
        }
    }
    int answer = min(n - result, n - ans);
    printf("%d\n", answer);
    return 0;
}
```

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

const int N = int(1e5) + 5;
int n, a[N], p[N];
bool used[N];
bool cmp(int i, int j) {
  return a[i] < a[j];
}

int solve() {
  memset(used, 0, sizeof(used));
  int cur = 0;
  for (int i = 0; i < n; ++i) {
    int x = i;
    if (used[x])
      continue;
    while (!used[x]) {
      used[x] = true;
      x = p[x];
    }
    cur++;
  }
  return n - cur;
}

int main() {
  assert(scanf("%d", &n) == 1);
  for (int i = 0; i < n; ++i) {
    assert(scanf("%d", &a[i]) == 1);
    assert(1 <= a[i] && a[i] <= int(2e9));
    p[i] = i;
  }

  std::sort(p, p + n, cmp);
  for (int i = 0; i + 1 < n; ++i)
    assert(a[p[i]] != a[p[i + 1]]);

  int res = solve();
  std::reverse(p, p + n);
  res = std::min(res, solve());
  printf("%d\n", res);
  return 0;
}
```

2.Highest Value Pallindrome

Consider the given number, S, as an array of characters. To make S palindromic:

1. Create 2 pointers, **left** and **right** , where left initially points to position 0 and **right** initially points to position **n-1** .
2. Compare the digits referenced by **left** and **right**; if they do not match, then overwrite the smaller value with the larger value. For every modified digit, decrement **k** .
3. Increment **left** (moving it one position to the right), and decrement **right**(moving it one position to the left).

Repeat steps 2 and 3 until **left** and **right** meet (i.e., the number is a palindrome). If **k** is negative, then it's not possible to make **S** a palindrome and you must print -1.

In the event that you do not use all **k** moves, then you can further maximize the number. To do this, you again perform step **1** (above). Then you overwrite the values for both **left** and **right** for any digit **<9** until you fully deplete your **k** moves (observe that each time you perform this action, you are changing **2** digits). If you only have **1** move left and **n** is odd, you can use this move to increase the middle digit to **9**.

C++

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

using namespace std;

char ans[100005] = {'\0'};
bool mark[100005];

void solution() {

  memset(mark,0,sizeof(mark));
  int n, len, l, r, k;
  string str;
```

```
cin>>n>>k;
cin>>str;
len = str.size();

assert(n>0 && n<=100000);
assert(k>=0 && k<=100000);
assert(len>0 && len<=100000);

//Making palindrome
l=0; r=len-1;
while(l<=r)
{
        assert(str[l]>='0' && str[l]<='9');
        assert(str[r]>='0' && str[r]<='9');

        if(l==r)
        {
                ans[l] = str[l];
                break;
        }
        if(str[l] == str[r])
        {
                ans[l] = str[l];
                ans[r] = str[r];
        }
        else
        {
                if(str[l]>= str[r])
                {
                        mark[r] = 1;
                        k--;
                        ans[l] = ans[r] = str[l];
                }
                else
                {
                        mark[l] = 1;
                        k--;
                        ans[l] = ans[r] = str[r];
                }
        }
        l++;
        r--;
}
```

```
  if(k<0)
  {
        printf("-1\n");
        return;
  }

  //Maximizing number
  l=0; r=len-1;
  while(l<=r)
  {
        if(l==r)
        {
              if(ans[l]<'9' && k>=1)
                    ans[l] = '9';
              break;
        }
        if(ans[l]<'9')
        {
              if(mark[l] == 0 && mark[r] == 0 && k>=2) //not touch before
              {
                    k-=2;
                    ans[l] = ans[r] = '9';
              }
              else if((mark[l]==1 || mark[r]==1) && k>=1)
              {
                    k-=1;
                    ans[l] = ans[r] = '9';
              }
        }
        l++;
        r--;
  }
  ans[len] = '\0';
  printf("%s\n", ans);
}

int main () {

            solution();
      return 0;
}
```

3.Matrix Layer Problem

Suppose we have following *4x5* matrix.

```
a11  a12  a13  a14  a15

a21  a22  a23  a24  a25

a31  a32  a33  a34  a35

a41  a42  a43  a44  a45
```

In order to rotate it, we create strips of various levels. In the above example 3 strips will be created and they are as follows.

Outer strip:

```
a11  a12  a13  a14  a15

a21                 a25

a31                 a35

a41  a42  a43  a44  a45
```

->

a11 <- a12 <- a13 <- a14 <- a15 <- a25 <- a35 <- a45 <- a44 <- a43 <- a42 <- a41 <- a31 <- a21

Inner strip:

```
  a22  a23  a24

  a32  a33  a34
```

->

a22 <- a23 <- a24 <- a34 <- a33 <- a32

Since *R* can be very large, we will rotate each strip, of length *k*, by *R%k* times. After rotating each strips we will recreate the original matrix.

Let's rotate above strip by one step, *R = 1*, and the creating the matrix will involve the following steps.

Outer strip:

a12 <- a13 <- a14 <- a15 <- a25 <- a35 <- a45 <- a44 <- a43 <- a42 <- a41 <- a31 <- a21 <- a11

->

```
a12  a13  a14  a15  a25

a11                 a35

a21                 a45

a31  a41  a42  a43  a44
```

Inner strip:

a23 <- a24 <- a34 <- a33 <- a32 <- a22

->

```
  a23  a24  a34

  a22  a32  a33
```

So the final configuration of the above matrix after 1 rotation will be:

```
a12  a13  a14  a15  a25

a11  a23  a24  a34  a35

a21  a22  a32  a33  a45

a31  a41  a42  a43  a44
```

Scala

```
import scala.collection.mutable.HashMap;

object Solution{
    def main(args: Array[String]) = {
        val in = readLine.split(" ").map(_.toInt);
        var arr = new Array[Array[Int]](in(0));
        for(i <- 1 to in(0))
```

```
            arr(i - 1) = readLine.split(" ").map(_.toInt);
        println(rotate(arr, in(2)));
    }

    def rotate(arr:Array[Array[Int]], r:Int):String = {
        var b = Array.fill(arr.size){ new Array[Int](arr(0).size) };
        var m = arr.size;
        var n = arr(0).size;
        val min = Math.min(m/2, n/2);
        for(i <- 1 to min){
            val map = getMap(m, n);
            val len = map.keys.size;
            for(j <- map.keys){
                val cur = map(j);
                val next = map((j + r) % len);
                b(next._1 + i - 1)(next._2 + i - 1) = arr(cur._1 + i - 1)(cur._2 + i - 1);
            }
            m -= 2;
            n -= 2;
        }
        return b.map(x => x.mkString(" ")).mkString("\n");
    }

    def getMap(m:Int, n:Int):HashMap[Int, Tuple2[Int, Int]] = {
        var map = new HashMap[Int, Tuple2[Int, Int]]();
        val len = 2 * (m + n - 2) - 1;
        for(i <- 0 to len) map += (i -> getLoc(i, m, n));
        return map;
    }

    def getLoc(i:Int, m:Int, n:Int):Tuple2[Int, Int] = {
        if(i < m - 1)
            return (i, 0);
        else if(i < m + n - 2)
            return (m - 1, (i - m + 1) % n);
        else if(i < 2 * m + n - 3)
            return (2 * m + n - 3 - i, n - 1);
        else
            return (0, 2 * (m + n - 2) - i);
    }
}
```

Python 2

```
from copy import deepcopy
m, n, r = map(int, raw_input().split())
matrix = []
for i in xrange(m):
    matrix.append(map(int, raw_input().split()))
k = min(m, n) / 2
rows = []
for ii in xrange(k):
    row = []
    for i in xrange(ii, m - 1 - ii):
        row.append(matrix[i][ii])
    for i in xrange(ii, n - 1 - ii):
        row.append(matrix[m - 1 - ii][i])
    for i in xrange(m - 1 - ii, ii, -1):
        row.append(matrix[i][n - 1 - ii])
    for i in xrange(n - 1 - ii, ii, -1):
        row.append(matrix[ii][i])
    rows.append(row)

result = deepcopy(matrix)

for ii in xrange(k):
    row = rows[ii]
    shift = r % len(row)
    idx = -shift
    for i in xrange(ii, m - 1 - ii):
        result[i][ii] = row[idx]
        idx += 1
        idx %= len(row)
    for i in xrange(ii, n - 1 - ii):
        result[m - 1 - ii][i] = row[idx]
        idx += 1
        idx %= len(row)
    for i in xrange(m - 1 - ii, ii, -1):
        result[i][n - 1 - ii] = row[idx]
        idx += 1
        idx %= len(row)
    for i in xrange(n - 1 - ii, ii, -1):
        result[ii][i] = row[idx]
        idx += 1
        idx %= len(row)
for i in result:
    print " ".join(map(str, i))
```

4. Minimum average waiting time

This problem can be solved using a **greedy approach.**

We have to remember that Tieu cannot know about future orders beforehand. So he will have to serve one of the orders that are currently waiting to be served. But how should he decide which order to serve now ?

This is where the greedy approach comes. He should always serve the order which has minimum cooking time i.e. the order with the minimum L . Let's prove this approach to be the optimal strategy using proof by contradiction.

Proof by contradiction:

Let's say currently we have X orders that need to be served. Let Lmin be the cooking time of the order with the minimum cooking time. Then if we decide to serve this order now, it will contribute (X * Lmin + C1) time to the total waiting time,where C1 is the time waited by new customers who have placed orders while we were cooking this order.

Now let's say, it was actually optimal to serve the j^th^ order where Lj > Lmin- Then this will contribute (X x Lj+ C2) time to the total waiting time, where C2 is the time waited by new customers who have placed orders while we were cooking the j^th^ order. Now we have,

1.Lj > Lmin

2. C2 > C1 (Since the j^th^ order takes more time to be cooked, the number of new customers who have placed order in the mean time will be higher or equal to the first scenario and they will have to wait more time as Lj > Lmin)

So we can conclude that,

(X x Lmin + C1) <(X x Lj + C2)

But this is a contradiction to the claim that serving the j^th^ order with Lj > Lmin, is optimal. Hence, using proof by contradiction, serving the order with the lowest cooking time is proved to be the optimal strategy.

Storing the data efficiently:

At first, we need to sort the orders according to increasing order time. An array is sufficient here. But then we need to store the currently available orders according to increasing cooking time. We can use a **min heap** for this purpose. We will loop through the array. Upon arriving at the ith order, we will insert it into the min heap if its order time is less than or equal to the current time. Otherwise, we will keep serving from the top of the heap until current time becomes larger than or equal to the order time of the ith order or the heap becomes empty. We will update the current time while inserting an order into the heap (if needed) and also while serving an order.

C++

```
#include<bits/stdc++.h>
using namespace std;

typedef long long int LL;

int n;
struct order{
   LL T, L;
}customer[100010];

bool operator < (order a, order b)
{
   return a.T < b.T;
}

bool compare(order a, order b)
{
   return a.L > b.L;
}
priority_queue<order, vector<order>, function<bool(order, order)>> min_heap(compare);

void add_order(order current_order, LL &current_time)
{
   if(min_heap.empty() == true)
      current_time = max(current_time, current_order.T);
   min_heap.push(current_order);
}

LL serve_order(LL &current_time)
{
   order current_order = min_heap.top();
   min_heap.pop();
```

```
    current_time += current_order.L;
    return current_time - current_order.T;
}

LL solve(int n)
{
    LL current_time = 0, total_waiting_time = 0;
    for(int i = 1; i<=n; i++)
    {
        if(i == 1 || customer[i].T <= current_time)
            add_order(customer[i], current_time);
        else if(min_heap.empty() == false)
        {
            while(min_heap.empty() == false && customer[i].T > current_time)
                total_waiting_time += serve_order(current_time);
            add_order(customer[i], current_time);
        }
    }
    while(min_heap.empty() == false)
        total_waiting_time += serve_order(current_time);
    return total_waiting_time/n;
}
int main()
{
    cin >> n;
    for(int i = 1; i<=n; i++)
        cin >> customer[i].T >> customer[i].L ;
    sort(customer+1, customer+n+1);
    cout << solve(n) << endl;
    return 0;
}
```

```
/*
Solution: pick the order that takes shortest time to cook.
*/
#include <cstdio>
#include <iostream>
#include <fstream>
#include <algorithm>
#include <vector>
#include <queue>
#include <stack>
#include <set>
```

```
#include <map>
#include <cstring>
#include <cstdlib>
#include <cmath>
#include <string>
#include <memory.h>
#include <sstream>
#include <complex>
#include <cassert>
#include <climits>

#define REP(i,n) for(int i = 0, _n = (n); i < _n; i++)
#define REPD(i,n) for(int i = (n) - 1; i >= 0; i--)
#define FOR(i,a,b) for (int i = (a), _b = (b); i <= _b; i++)
#define FORD(i,a,b) for (int i = (a), _b = (b); i >= _b; i--)
#define FORN(i,a,b) for(int i=a;i<b;i++)
#define FOREACH(it,c) for (__typeof((c).begin()) it=(c).begin();it!=(c).end();it++)
#define RESET(c,x) memset (c, x, sizeof (c))

#define PI acos(-1)
#define sqr(x) ((x) * (x))
#define PB push_back
#define MP make_pair
#define F first
#define S second
#define Aint(c) (c).begin(), (c).end()
#define SIZE(c) (c).size()

#define DEBUG(x) { cerr << #x << " = " << x << endl; }
#define PR(a,n) {cerr<<#a<<" = "; FOR(_,1,n) cerr << a[_] << ' '; cerr <<endl;}
#define PR0(a,n) {cerr<<#a<<" = ";REP(_,n) cerr << a[_] << ' '; cerr << endl;}
#define LL long long

using namespace std;

#define maxn 100111
int n;
pair<int, int> order[maxn];
multiset<pair<int, int> > available_order;

bool getChar(char& c) {
  return (scanf("%c", &c) == 1);
}
```

```
void nextInt(long long& u, char endline, int l, int r) {
    int sign = 1;
    long long sum = 0;
    char c;
    assert(getChar(c));
    if (c == '-') sign = -1;
    else {
        assert('0' <= c && c <= '9');
        sum = (c - '0');
    }

    int cnt = 0;

    for (int i = 1; i <= 15; i++) {
        assert(getChar(c));
        if (c == endline) break;
        assert('0' <= c && c <= '9');
        sum = sum * 10 + (c - '0');
        cnt ++;
        assert(cnt <= 15);
    }

    sum = sign * sum;
    assert (l <= sum && sum <= r);
    u = sum;
}

void nextInt(int& u, char endline, int l, int r) {
    long long tmp;
    nextInt(tmp, endline, l, r);
    u = tmp;
}

int main() {
    //freopen("a.in", "rb", stdin); freopen("a.out", "wb", stdout);
    nextInt(n, '\n', 1, 1e5);
    for (int i = 0; i < n; i++) {
        nextInt(order[i].first, ' ', 0, 1e9);
        nextInt(order[i].second, '\n', 1, 1e9);
    }

    //sort the orders in the increaseing order of the submit time.
```

```
  sort(order, order + n);

  //we maintain the available_order set, which stores the submitted order in the increasing order of the
time to cook
  int pos = 0;//pos is the next order to be considered to be push in available_order
  long long current_time = order[0].first;
  //the total waiting time may be out of 64bit integer range so we maintain two values average and mod
so that total waiting time = average * n + mod.
  long long average = 0, mod = 0;

  for (int i = 0; i < n; i++) {
    //push all submitted orders into available_order
    while (pos < n && order[pos].first <= current_time) {
      //note that in available_order, the order is stored in the increasing order of time to cook
      available_order.insert(make_pair(order[pos].second, order[pos].first));
      pos++;
    }

    //take out the order that has a shortest time to cook
    pair<int, int> item = *available_order.begin();

    available_order.erase(available_order.begin());

    //update current_time, average and mod
    current_time += item.first;
    //this is the waiting_time of the current order
    long long waiting_time = current_time - item.second;

    mod += waiting_time % n;
    average += waiting_time / n;

    if (mod >= n) {
      average += mod / n;
      mod %= n;
    }
  }

  cout << average << endl;
  char __c;
  assert(!getChar(__c));
  return 0;
}
```

```
#include<bits/stdc++.h>
#define assn(n,a,b) assert(n>=a && n<=b)
#define F first
#define S second
#define mp make_pair
using namespace std;
int main()
{
   int n,i;
   vector < pair < int, int > > ar;
   long long cur=0,ans=0;
   cin >> n;
   ar.resize(n);
   assn(n,1,1e5);
   for(i=0; i<n; i++)
   {
   cin >> ar[i].F >> ar[i].S;
   assn(ar[i].F, 0, 1e9);
   assn(ar[i].S, 1, 1e9);
   }

   sort(ar.begin(),ar.end());
   priority_queue< pair < int, int > , vector< pair < int, int> >, greater<  pair < int, int > > > mheap;
   i=1;
   mheap.push(mp(ar[0].S,ar[0].F));
   cur=ar[0].F;
   while(!mheap.empty() || i<n)
   {
   while(i<n && ar[i].F<=cur)
   {
      mheap.push(mp(ar[i].S,ar[i].F));
      i++;
   }
   if(mheap.empty() && i<n)
   {
      cur=ar[i].F;
      mheap.push(mp(ar[i].S,ar[i].F));
      i++;
   }
   pair < int, int > p = mheap.top();
   mheap.pop();
   cur+=(long long)(p.F);
//  printf("cur:%d cook-time:%d coming-time:%d\n",cur,p.F,p.S);
```

```
    ans+=(long long)(cur)-(long long)(p.S);
    }
    cout << ans/n << endl;
    return 0;
}
```

5. Maximizing Mission Points

First sort the cities by their heights and get rid of this dimension. Now we essentially have a list of weighted points in the 2-d plane, and we need to find the subsequence with maximum score which satisfies the following constraints:

- The absolute difference in the x- coordinates between adjacent points in his subsequence must be at most .
- The absolute difference in the y- coordinates between adjacent points in his subsequence must be at most .

This can be solved using dynamic programming + divide and conquer.

Let dp(i)= best score I can obtain when my subsequence has *ith* point as the last one.

Basic Idea (divide and conquer):

- Solve for the left half first. After this step dp(i) for all *i* in the left half is complete.
- Then merge the subproblems. What this means is that we calculate dp(i) for all *i* in the right half, assuming that the previous adjacent point (if any) was neccesarily from the left half. After this step, dp(i) for all *i* in right half is only partially complete. (We are yet to process the cases when *i* is in right half and so is the previous point)
- Then solve for the right half. Now dp(i) is calculated for the whole range

Merging the subproblems: Sort the points (in left and right) by their x co-ordinates, and iterate over the points in the right half in increasing order.

For a point in the right half:

- Remove all points in left half which are at a distance greater than X behind it, and add all points in the right half which are at a distance less than or equal to X ahead of it. In other words, only points (of the left half) which are within absolute distance (x - coordinate only) X of this point (of the right half), are "active".
- Among all active points, query the one which has y - coordinate within Y, and has maximum value.

For a point in the left half: maintain a segment tree which supports point updates and range max queries:

- To add it, set Y[i] = DP[i]
- To remove it, set Y[i] = -infinity

Refer to the setter's code in case the above explanation is not clear (easier to do, than to explain).

```
--------------
#include <bits/stdc++.h>
using namespace std;
const int MAXN = 200000;
const long long INF = 1e15;

long long tree[MAXN * 4];

void makeTree() {
  for (int i = 1; i < MAXN * 4; ++i) {
    tree[i] = -INF;
  }
}
void update(int x, long long val, int u = 1, int l = 1, int r = MAXN) {
  if (l == r) {
    tree[u] = val;
    return;
  }
  int m = (l + r) / 2;
  if (x <= m) {
    update(x, val, 2 * u, l, m);
  }
  else {
    update(x, val, 2 * u + 1, m + 1, r);
  }
  tree[u] = max(tree[2 * u], tree[2 * u + 1]);
```

```
}

long long query(int x, int y, int u = 1, int l = 1, int r = MAXN) {
   if (x > r or y < l) {
      return -INF;
   }
   if (x <= l and r <= y) {
      return tree[u];
   }
   int m = (l + r) / 2;
   return max(query(x, y, 2 * u, l, m), query(x, y, 2 * u + 1, m + 1, r));
}

struct Point {
   int x, y, z, w;
   long long dp;
   bool operator < (const Point &o) const {
      return z < o.z;
   }
};
Point p[MAXN + 1];
long long DP[MAXN + 1];
int X_LIM, Y_LIM;
void merge(int l, int r) {
   int m = (l + r) / 2;

   vector<pair<int, int> > left;
   vector<pair<int, int> > right;
   for (int i = l; i <= m; ++i) {
      left.push_back({p[i].x, p[i].z});
   }
   for (int i = m + 1; i <= r; ++i) {
      right.push_back({p[i].x, p[i].z});
   }
   sort(left.begin(), left.end());
   sort(right.begin(), right.end());

   int left_l = 0;
   int left_r = -1;
   for (auto u : right) {
      int i = u.second;
      int x = u.first;
      while (left_r + 1 < left.size() and left[left_r + 1].first - x <= X_LIM) {
            ++left_r;
            int who = left[left_r].second;
            update(p[who].y, p[who].dp);
        }
        while (left_l < left.size() and x - left[left_l].first > X_LIM) {
            int who = left[left_l].second;
            update(p[who].y, -INF);
            ++left_l;
        }
        int yLow = max(1, p[i].y - Y_LIM);
        int yHigh = min(MAXN, p[i].y + Y_LIM);
        p[i].dp = max(p[i].dp, p[i].w + query(yLow, yHigh));
    }
    while (left_l <= left_r) {
        int who = left[left_l].second;
        update(p[who].y, -INF);
        ++left_l;
    }
}
void solve(int l, int r) {
    if (l == r) {
        p[l].dp = max(p[l].dp, (long long)p[l].w);
        return;
    }
    int m = (l + r) / 2;
    solve(l, m);
    merge(l, r);
    solve(m + 1, r);
}
int main() {
    makeTree();
    int n;
    scanf("%d %d %d", &n, &X_LIM, &Y_LIM);
    for (int i = 0; i < n; ++i) {
        scanf("%d %d %d %d", &p[i].x, &p[i].y, &p[i].z, &p[i].w);
        p[i].dp = -INF;
    }
    sort(p, p + n);
    for (int i = 0; i < n; ++i) {
        p[i].z = i;
    }
    solve(0, n - 1);
    long long ans = 0;
    for (int i = 0; i < n; ++i) {
        ans = max(ans, p[i].dp);
    }
    printf("%lld\n", ans);
}
```

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int const N = 800600;
ll const INFL = 1e18 + 111;

struct Item {
    int x, y, h;
    ll cost;
};

struct Max {
    vector<pair<ll, ll>> a, b;

    Max() {
        clear();
    }

    void clear() {
        a.clear();
        a.emplace_back(-INFL, -INFL);
        b.clear();
        b.emplace_back(-INFL, -INFL);
    }

    void add(ll x) {
        a.emplace_back(x, max(a.back().second, x));
    }

    ll get_max() const {
        return max(a.back().second, b.back().second);
    }
    void pop() {
        if (b.size() == 1u) {
            while (a.size() > 1u) {
                b.emplace_back(a.back().first, max(a.back().first, b.back().second));
                a.pop_back();
            }
        }
        b.pop_back();
        assert(a.size() >= 1u);
        assert(b.size() >= 1u);
    }
};

int n, dx, dy;
Item items[N];
ll ans[N];
Max leafs[N];
ll tree[N];

void build(int v, int l, int r) {
    tree[v] = -INFL;
    if (r - l > 1) {
        int m = (l + r) / 2;
        build(2 * v + 1, l, m);
        build(2 * v + 2, m, r);
    } else {
        leafs[l].clear();
    }
}

void add_number(int v, int l, int r, int pos, ll num) {
    if (r - l == 1) {
        if (num == INFL)
            leafs[l].pop();
        else
            leafs[l].add(num);
        tree[v] = leafs[l].get_max();
    } else {
        int m = (l + r) / 2;
        if (pos < m)
            add_number(2 * v + 1, l, m, pos, num);
        else
            add_number(2 * v + 2, m, r, pos, num);
        tree[v] = max(tree[2 * v + 1], tree[2 * v + 2]);
    }
}

ll get_max(int v, int l, int r, int from, int to) {
    if (r <= from || to <= l)
        return -INFL;
    if (from <= l && r <= to)
        return tree[v];
    int m = (l + r) / 2;
    return max(get_max(2 * v + 1, l, m, from, to), get_max(2 * v + 2, m, r, from, to));
}

struct Event {
    int x, y1, y2, type, i; /// -1 - open, 0 - point, 1 - close
};

bool operator<(Event const& e1, Event const& e2) {
    if (e1.x != e2.x)
        return e1.x < e2.x;
    return e1.type < e2.type;
}

void go(int L, int R) {
    if (R - L == 1) {
    } else {
        int M = (L + R) / 2;
        go(L, M);
        static int ally[N];
        int cn = 0;
        for (int i = L; i < M; ++i) {
            ally[cn++] = items[i].y;
        }
        sort(ally, ally + cn);
        cn = unique(ally, ally + cn) - ally;
        #define comp(q) (lower_bound(ally, ally + cn, q) - ally)
        static Event events[N];
        int es = 0;
        for (int i = L; i < M; ++i) {
            int q = comp(items[i].y);
            events[es++] = {items[i].x - dx, q, -1, -1, i};
            events[es++] = {items[i].x + dx, q, -1, 1, i};
        }
        for (int i = M; i < R; ++i) {
            int from = comp(items[i].y - dy);
            int to = comp(items[i].y + dy + 1);
            events[es++] = {items[i].x, from, to, 0, i};
        }
        build(0, 0, cn);
        sort(events, events + es);
        for (int i = 0; i < es; ++i) {
            auto e = events[i];
            if (e.type == -1) {
                add_number(0, 0, cn, e.y1, ans[e.i]);
            } else if (e.type == 1) {
                add_number(0, 0, cn, e.y1, INFL);
            } else {
                ll cur = get_max(0, 0, cn, e.y1, e.y2) + items[e.i].cost;
                ans[e.i] = max(ans[e.i], cur);
            }
        }
        go(M, R);
    }
}

bool solve() {
    if (scanf("%d%d%d", &n, &dx, &dy) < 0) {
        return false;
    }
    assert(n >= 1 && n <= 200000);
    assert(dx >= 1 && dx <= 200000);
    assert(dy >= 1 && dy <= 200000);
    for (int i = 0; i < n; ++i) {
        int x, y, h, cost;
        scanf("%d%d%d%d", &x, &y, &h, &cost);
        assert(x >= 1 && x <= 200000);
        assert(y >= 1 && y <= 200000);
        assert(h >= 1 && h <= 200000);
        assert(cost >= -200000 && cost <= 200000);
        items[i] = {x, y, h, (ll)cost};
    }
    sort(items, items + n, [](Item const &a, Item const &b) {
        return a.h < b.h;
    });
    for (int i = 0; i < n; ++i) {
        ans[i] = items[i].cost;
    }
    go(0, n);
    ll res = 0;
    for (int i = 0; i < n; ++i) {
        res = max(res, ans[i]);
    }
    cout << res << '\n';
    return true;
}

int main() {
    while (solve());
}
```

6. Two Subarrays

Note: In the editoral author using numeration of arrays from 1 instead of 0

Subtask for 20% of score:

In this subtask we can calculate value of function **f** for all subarrays in the given array.

First we should notice that function **sum** has next property :

sum(l,r) = sum(1,r) — sum(1,l — 1) , so we can precalculate all values **sum(l,i)** (prefix sums) and find for the given subarray A[l, r] value of function sum in **O(1).**

Now we need to calculate value of function inc(l, r). We will use dynamic programming and make next **DP** array :

DP[i][j][k] is denoting maximum sum of some strictly increasing subsequence which didn't start before i and finished at most at postion j (subarray **A[i, j]**) and last element is equal to **k**.

DP[i][j][k] = DP[i][j — 1][k] for Aj ≠ AK

DP[i][j]|k] = max(DP[i][j — 1][k], DP[i][j — 1[p] + Aj) for Aj = k, p = 0,1,2,3,...,Aj — 1

Now our answer for subarray A[l, r] is equal to sum(1, r) — sum(1,l — 1) + max(DP[l][r][p]),

p=0,1,2,3,..., maw(Ai). At the end just find the biggest value over all subarrays and count amount of such subarrays with minimum possible length.

Total complexity **$O(n^2$ max(Ai)).**

You can see a little different approach for 20% of score in the setter section for code. Main idea that we can caluclate given value DP[i][j][k], similar as calculating Longest Increasing Subsequence (LIS), only in our **DP** states we are saving maximum value of some increasing subsequence instead of maximum length of that subsequence. Also we can reduce memory in the solution above - notice that is enough to save only second and third coordinate and everytime for different **i** you need to set all **DP** values to zero.

Complexity of setter code is $O(n^2 logn)$

Solution for 60% of score:

This solution contains nearly all things as solution for whole score, but we do not need to notice some little properties of our answer. Again we will define **3D** dynamic programming array **DP**.

Now **DP[i][s][k]** is denoting the right most position such that exists strictly increasing subsequence starting from that postion and finishing at most at postion **i**, has sum of all its elements equal to **s** and last element in the sequence is **k**. How this values help us to find our answer ?

Well, let's suppose we calucated all **DP** values for position **i**. We want to calculate maximum answer for all possible arrays which has right end point at position **i** (all subarrays[I,i] l ≤ i). Now we will find all subarrays with maximum possible sum (**maxS**) which is finishing on postion i. That is all subarrays with left point l,0 < 1 ≤ max(DP[i][maxS]|[p])

(p = 1,2,..., max(Ai) - actually here is only possible that p = max(Ai), but never mind) . So our answer for that fixed sum maxS and fixed subarray with right end point **i** is ans = sum(1, i) — sum(1,l— 1) — max(DP[i|[maxS][p]).

Only value **sum(1, l- 1)** is not constant(we precalculate all other values before), so our answer will be maximum in case when **sum(1,1 — 1**) is minimum. Such value we can find using segment tree and Range Minimum Queries for prefix sums in the array.

Now we will repeat the same process for total sum equal to max.S — 1, only interval for left end point of subarray will be changed to **max(DP[i][maxS][p1]) < 1 ≤max(DP[i][maxS — 1][p2]) (p1,p2 = 1...max(Ai))**. Please notice that in case max**(DP[i][maxS][p1]) > maa(DP[i][maxS — 1][p2])** we actually do not have subarray with value of maximum sum for increasing subsequnce equal to **maxS — 1.**

We will repeat the process for number maxS — 2 and so on till 1.

When we saw how DP array helps us to calculate answer, we will see how to find this array :

DP[i][s][k] = DP[i — 1][s][k] for k ≢ Ai

DP[il[s][k] = max(DP[i — 1][s — Ai][p]) for k = Ai and p=1,2,...,Ai —1

Total complexity of this solution is **O(n max(Ai)^3 +n maa(Ai)^2 log n)**. Maybe solution looks that has bigger complexity than formula above, but please notice we are finding all **DP** values in **O(1)** instead of case when **k = Ai**, where we are find optimal answer in complexity **O(max(Ai))** - and this second case will have again at most n* max(Ai)^3 for whole array...

Also, for better understanding we could use another matrix **best[s][k]** which is denoting maximum value of DP on postion **i — 1** with total sum of increasing subsequence s and last element not bigger than k... We would calculate

best[s|[k] = max(best[s][k — 1], dp[i — 1][s][k]).

Soltuion for whole score:

First why we are using 3D matrix for DP ? Obviosly our DP on postion only depends from previous postion and it is changing

in case **Ai = k**. . . So we will use 2D array like for calculating array best[s][k].

DP[s](k] = DP[s]|[k] fork ≠ Ai

DP[s](k] = best[s — Ai][Ai] k = Ai

We reduced memory. Now what we can do next? We can destroy **log** factor from previous solution. For Range Minimum Queries we do not have to use Segment Tree, we can use Sparse Table and answer queries in complexity O(1) with precalculations in complexity O(n log n).

Next thing:

We do not have to calculate DP for all values, only for **k = A**i (otherwise it won't be changed and formula

DP{s][k] = DP[s][k] looks funny). So this part for calulating DP reduced our complexity for fixed position i to O(max(Ai)^2)- we are changing only first coordinate.

The last optimization in the solution looks very simple and at the first view it doesn't cut complexity of our code, but actually with small constraints for **Ai** it will finish the job ! (till now we are calculating matrix **best** in the complexity O(max(Ai)^3).

Here is optimized code for calculating matrix best :

```
for (int j=a[i];j<=(a[i]*(a[i]+1))/2;j++)
   for (int k=a[i];k<=me;k++)
      best[k][j]=max(best[k-1][j],dp[k][j]);
```

In case last element is equal to Ai, obviously maximum sum of strictly increasing subsequence can not be smaller than **Ai** (choose only last element), also that sum can not be bigger than (Ai.(Ai +l))/2 (we can choose all numbers **1+ 2+3-+---+ Ai).**So we do not need to iterate over all sums in matrix best, only sums between this two values. Same thing for the last element, in case we put Ai on that last place we will changest only postions in matrix **best[s][k]** where **Ai ≤ k.**

This looks as some easy optimisation which won't change complexity, but actully we will have at most (2/27)max(Ai)^3 (you should find extrem value of the code above, and it can be done writing function and checking local maximum and minimum).

So at the end we got complexity **O((2/27)nmax(Ai)^3**), which is near to **O(nmax(Ai)^2)** for given constraint for array **A.**

Also some other approaches with reducing complexity from **O(nmax(Ai)^3** will pass system testing, we know solution in complexity O(n maz(Ai)^2 √max(Ai)).

Note : We didn't disscuss finding amount of subarrays with maximum goodness and minimum length, we can do it directly during **DP** calculation. In case we update answer, we set new length and again put amount of subarrays to 1, otherwise in case current **DP** value and answer are same we need to check whether we are changing minimum length or not.

It depends of implementation, but possible you need to consider one special case, when answer is equal to **0**. In this case amount of good subarrays is equal to **n**.

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

using namespace std;

//O(2/27 * (n A[i]^3))

const int sm=18;
const int maxi=3e5+5;
const int big=1e7;
const int me=40;
const int ms=821;

int lg[maxi],a[maxi],st[sm], p[maxi][sm], dp[me+1][ms],s[maxi],best[me+1][ms];
int n;

void calc()
{
   st[0]=1;
   lg[1]=0;
   for (int i=1;i<sm;i++)
   {
      st[i]=st[i-1]*2;
      lg[st[i]]=i;
   }

   for (int i=2;i<=maxi;i++)
      if (!lg[i]) lg[i]=lg[i-1];
}

void update_sparse()
{

   for (int i=1;i<sm;i++)
  for (int poz=0;poz<=n;poz++)
   if (s[p[poz][i-1]]<=s[p[poz+st[i-1]][i-1]])
    p[poz][i]=p[poz][i-1];  else
     p[poz][i]=p[poz+st[i-1]][i-1];


}

int get_min(int l, int r)
{
  int range=(r-l)+1;
  if (s[p[l][lg[range]]]<=s[p[r-st[lg[range]]+1][lg[range]]])
     return p[l][lg[range]]; else
      return p[r-st[lg[range]]+1][lg[range]];
}

int main()
{
  scanf("%d",&n);

  calc();

 p[0][0]=0;
 for (int i=1;i<=n;i++)
 {
   scanf("%d",&a[i]);
   s[i]=s[i-1]+a[i];
   p[i][0]=i;
 }

 update_sparse();

 int ans=0;
 int am=0;
 int len=0;

 for (int i=1;i<=n;i++)
   if (a[i]>0)
    {
      dp[a[i]][a[i]]=i;
      for (int j=a[i]+1;j<=(a[i]*(a[i]+1))/2;j++)
      dp[a[i]][j]=best[a[i]-1][j-a[i]];

      for (int j=a[i];j<=(a[i]*(a[i]+1))/2;j++)
      for (int k=a[i];k<=me;k++)
          best[k][j]=max(best[k-1][j],dp[k][j]);

     int lef=0;
     for (int k=ms-1;k>=0;k--)
      if (best[me][k]>lef)
     {
         int idx=get_min(lef,best[me][k]-1);
         if (ans<s[i]-s[idx]-k)
         {
           ans=s[i]-s[idx]-k;
           am=1;
           len=i-idx+1;
         } else
         if (ans==s[i]-s[idx]-k && len>i-idx+1)
         {
           len=i-idx+1;
           am=1;
         }
         else
          if (ans==s[i]-s[idx]-k && len==i-idx+1)  am++;

         lef=best[me][k];
     }
   }
 if (ans==0) am=n;
 printf("%d %d\n",ans,am);

 return 0;
}
```

```
#include <iostream>
#include <cmath>
#include <algorithm>
#include <vector>
#include <cstring>
#include <deque>
#include <stack>
#include <stdio.h>
#include <map>
#include <set>
#include <time.h>
#include <string>
#include <fstream>
#include <queue>
#include <bitset>
#include <cstdlib>
#include <assert.h>
#define X first
#define Y second
#define mp make_pair
```

```
#define pb push_back
#define pdd pair<double,double>
#define pii pair<ll,ll>
#define PI 3.14159265358979323846
#define MOD 1000000007
#define MOD2 1000000009
#define INF ((ll)1e+18)
#define x1 fldgjdflgjhrthrl
#define x2 fldgjdflgrtyrtyjl
#define y1 fldggfhfghjdflgjl
#define y2 ffgfldgjdflgjl
#define N 262134
#define SUM 23423
#define MAG 100000000
#define OPEN 0
#define CLOSE 1
typedef int ll;
typedef long double ld;
using namespace std;
ll i,j,n,k,l,m,tot, flag,r,ans,z, K,x1,y1,x2,y2,x3,y3,x,y,h,num,h2,timer,sz,q,c;
ll dp[41][821], a[200500], b[41][825], pa[200500];
pii t[500500];
void build() {  // build the tree
  for (int i = N - 1; i > 0; --i) t[i] = min(t[i<<1], t[i<<1|1]);
}

pii query(int l, int r) {  // sum on interval [l, r)
  pii res = mp(MOD,0);
  for (l += N, r += N; l < r; l >>= 1, r >>= 1) {
    if (l&1) res = min(res, t[l++]);
    if (r&1) res = min(res, t[--r]);
  }
  return res;
}


int main() {
          //freopen("input.txt","r",stdin);
          //freopen("output.txt","w",stdout);
          h = 820;
          for (i = 0; i <= 40; i++)
                    for (j = 0; j <= h; j++)
                              dp[i][j] = b[i][j] = -1;
```

```
cin >> n;
assert(n >= 1 && n <= 200000);
for (i = 0; i < n; i++)
{
        cin >> a[i];
        assert(-40 <= a[i] && a[i] <= 40);
        pa[i+1] = pa[i] + a[i];
        t[N+i+1] = mp(pa[i+1], -i-1);
}
build();
ll ans = 0, len = 0, cnt = 0;
for (i = 0; i < n; i++)
if (a[i] > 0)
{
        dp[a[i]][a[i]] = i;
        b[a[i]][a[i]] = i;
        ll val = a[i]*(a[i]-1)/2;
        ll to = a[i];
        //for (j = 1; j < to; j++)
                for (k = 1; k <= val; k++)
                {
                        dp[to][k+to] = max(dp[to][k+to], b[to-1][k]);
                        b[to][k+to] = max(b[to][k+to], dp[to][k+to]);
                }
        for (j = to+1; j <= 40; j++)
                for (k = to; k <= val+to; k++)
                        b[j][k] = max(b[j][k], b[j-1][k]);
        ll cur = -1, lst = -1, cur_sum = -1;
        for (j = h; j >= a[i]; j--)
        {
                if (b[40][j] > cur)
                {
                  lst = cur, cur = b[40][j], cur_sum = j;
                        pii mn = query(lst+1, cur+1);
                        mn.Y = -mn.Y;
                        ll val1 = pa[i+1]-mn.X-cur_sum, val2 = i-mn.Y;
                        if (ans < val1)
                        {
                          ans = val1, cnt = 1, len = val2;
                        }
                        else
                        if (ans == val1)
                        {
```

```
                                                  if (val2 < len)
                                                           cnt = 1, len = val2;
                                                  else
                                                  if (val2 == len)
                                                            cnt++;
                                                }
                                      }
                            }
                  }
                  if (ans == 0)
                    cnt = n;
                  cout << ans << " " << cnt << endl;
        return 0;
}
```