Task 1: Collecting Mushrooms

Author: Lim Li
Singapore IOI Team 2017
limlirandom@gmail.com

Introduction

The abridged problem: You are given a grid of size $R \times C \leq 500000$. Each grid can contain a mushroom, a sprinkler, or neither. Sprinklers can water squares with distance $\leq D$, where distance is defined as $max(|X_s - X_m|, |Y_s - Y_m|)$. Find the number of mushrooms with $\geq K$ sprinklers watering it.

Subtasks

Subtask 1

In Subtask 1, $D = max(R, C), K = 1$, ie every sprinkler can reach the whole grid and a mushroom just needs one sprinkler to be harvestable. Since it is guaranteed that there is at least one sprinkler, we simply have to count the number of mushrooms on the grid.

Subtask 2

In Subtask 2, K is not limited, but the range of every sprinkler is still the whole grid. An addition step of counting the number of sprinklers is required. If the number of sprinklers is more than or equal to K, then output the number of mushrooms. Else, output 0.

Subtask 3

In Subtask 3, $D = 1, K = 1$. Since $D = 1$, we can look at the 3×3 grid surrounding each mushroom, and check if there is a sprinkler to determine if this mushroom can be harvested. This method has time complexity $O(RC + (\text{no. of mushrooms})D^2)$, which is $O(RC)$ since $D = 1$.

Subtask 4

In Subtask 4, K and D are not limited, but no. of mushrooms ≤ 500 and no. of sprinklers ≤ 500.

We can use a similar method as subtask 3, but instead of looping through the $(2D+1) \times (2D+1)$ box around each mushroom to count the number of sprinklers inside, we loop through the list of sprinklers to check if it is inside the $(2D+1) \times (2D+1)$ box. Now, counting the number of mushrooms in the $(2D+1) \times (2D+1)$ box takes $O(\text{no. of sprinklers})$ time. This method has time complexity $O(RC + (\text{no. of mushrooms}) \times (\text{no. of sprinklers}))$

Subtask 5

For Subtask 5, $R = 1$. Now the $(2D+1) \times (2D+1)$ box surrounding each mushroom is a line of length $2D+1$. To count the number of sprinklers in a line, we can use a prefix sum. Let S_i be the number of sprinklers in the range from column 0 to column i. Define S_{-1} to be 0. S_i can be precomputed in $O(C)$. Then the number sprinklers in the range a to b is $S_b - S_{a-1}$, which can be obtained in $O(1)$ after precomputation.

Subtask 6

Subtask 6 offers the full problem. The idea of prefix sum in subtask 5 can be generalised for a 2D grid. Let $S_{i,j}$ be the number of sprinklers in the region from $(0,0)$ to (i,j), where S is defined to be 0 for negative values of i, j. S can be precomputed. Then the number of sprinklers in the region from (A_x, A_y) to (B_x, B_y), $A_x < B_x, A_y < B_y$ is $(S_{B_x,B_y} - S_{A_x-1,B_y} - S_{B_x,A_y-1} + S_{A_x-1,A_y-1})$, which can be obtained in $O(1)$. This gives a complexity of $O(RC)$, which will obtain full marks for this problem.

Task 2: Journey

Author: Frank Stephan
Professor at NUS
fstephan@comp.nus.edu.sg

Introduction

The abridged problem: You are given a list of n cities. Each city has h flights leaving from that city. Each flight can be described by a pair of integers: (Destination, Minimum Stay). After taking a flight, Kuno can choose to stay in the city you arrived at for some number of nights that is at least the Minimum Stay of that flight. Count the number of ways for Kuno to fly to city $n-1$ from city 0 in k days for $k < m$, or determine if the answer is at least 500000001.

Full solution

Let $f(C, D)$ be the number of ways to reach city C in D days. To reach $f(C, D)$, Kuno has to either reach city C in less than D days and wait for the remaining days, or he can reach city C in exactly D days. This allows us to form the recurrence function for $f(C, D)$.

Let S_C contain the list of flights with destination to city C, (i, t), where the flight departs from city i, and takes t time to fly.

$$f(C,D) = \begin{cases} 0, & \text{if } D < 0 \\ \min(500000001, f(C, D-1) + \sum\limits_{(i,t)\in S_C} f(i, D-t)), & \text{otherwise} \end{cases}$$

This has $O(nm)$ states, and takes an average of $O(h)$ transition time. By using dynamic programming to memorize the answer for each state, we ensure that each state is only calculated once. This gives a total complexity of $O(nmh)$.

Task 3: Lightningrod

Author: Zhang Guangxuan
Singapore IOI Team 2016-2017
zhangguangxuan99@gmail.com

Introduction

The abridged problem:
There are N points on a plane. For each point in a selected subset, draw a right isosceles triangle with vertex at the selected point and hypothenuse parallel to the x-axis. Find the minimum number of selected points such that all N points lie within a triangle.

Subtasks

Subtask 1

In Subtask 1, we are given the additional constraint that $Y_i = 1$, where all points have the same Y-coordinate.

All right angle triangles cover exactly 1 point, so the answer is N.

Subtask 2

In Subtask 2, there are exactly 2 points ($N = 2$).

If $|X_0 - X_1| \leq |Y_0 - Y_1|$, one point can cover the other point, so the answer is 1. Otherwise, the points cannot cover each other, so the answer is 2.

Subtask 3

In Subtask 3, there are at most 20 points ($N \leq 20$).

We can do a $O(2^N)$ brute force for selecting a subset of points, $O(N^2)$ to check if the selection is valid.

Sample C++ code (with time complexity of $O(2^N N^2)$):

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

```
using namespace std;

int x[21],y[21];

int main(){
    int n;scanf("%d",&n);
    for(int i=0;i<n;i++){
        scanf("%d%d",&x[i],&y[i]);
    }
    int ans=n;
    for(int bm=0;bm<(1<<n);bm++){
        int cans=0;
        for(int i=0;i<n;i++){
            cans+=(bm&(1<<i))>0;
            bool safe=(bm&(1<<i))>0;
            for(int j=0;j<n;j++){
                safe|=(bm&(1<<j))&&(abs(x[j]-x[i])<=(y[j]-y[i]));
            }
            if(!safe){cans=n;break;}
        }
        ans=min(cans,ans);
    }
    printf("%d",ans);
}
```

Subtask 4

In Subtask 4, there are at most 2000 points ($N \leq 2000$)

Observe that if we process points by decreasing Y_i, then we have to pick the current point if and only if previous points have not covered this point. We can check this for every point in $O(N)$ by looping through previous points, giving a total complexity of $O(N^2)$.

Sample C++ code (with time complexity of $O(N^2)$):

```
#include <bits/stdc++.h>
using namespace std;
pair<int,int> y[5009];

int main(){
    int n;scanf("%d",&n);
    for(int i=0;i<n;i++){
        int a,b;scanf("%d%d",&a,&b);
        y[i]=make_pair(b,a);
    }
    sort(y,y+n,greater<pair<int,int> >());
    int ans=0;
    for(int i=0;i<n;i++){
        bool pick=1;
        for(int j=0;j<i;j++){
            pick&=(y[j].first-y[i].first)<abs(y[i].second-y[j].second);
        }
        ans+=pick;
    }
    printf("%d",ans);
}
```

Subtask 5

In Subtask 5, there are at most 200 000 points ($N \leq 200\,000$)

We can improve on the subtask 4 solution. Instead of looping through all previous points to check if the current point is already covered, we just have to look for the highest point on the left and right of the current point.

Let the current point be (X_i, Y_i). For points to the left, we look for the largest $X_j + Y_j$, and compare it with $X_i + Y_i$. For points to the right, we look for the largest $Y_j - X_j$, and compare it with $Y_i - X_i$. The current point is chosen if and only if no points cover the current point.

This can be in $O(logN)$ using 2 range max point update segment trees or fenwick trees, where the first tree stores $X_i + Y_i$, second tree stores $Y_i - X_i$. For each point, we do a prefix query on the first tree, suffix query on the second tree. If the current node is chosen, update the current position with $X_i + Y_i$ in the first tree, $Y_i - X_i$ in the second tree.

Sample C++ code (with time complexity of $O(NlogN)$):

```
#include <bits/stdc++.h>
using namespace std;
pair<int,int> yy[200009];
bool p[200009];
int dc[200009];

struct node{
    int s,e,m,v;
    node *l,*r;
    node(int _s,int _e):s(_s),e(_e),m((_s+_e)/2),v(-2e9){
        if(s==e)return;
        l=new node(s,m);r=new node(m+1,e);
    }
    void up(int x,int uv){
        if(s==e){v=max(v,uv);return;}
        if(x>m)r->up(x,uv);
        else l->up(x,uv);
        v=max(l->v,r->v);
    }
    int qu(int x,int y){
        if(s==x&&e==y)return v;
        if(x>m)return r->qu(x,y);
        if(y<=m)return l->qu(x,y);
        return max(l->qu(x,m),r->qu(m+1,y));
    }
}*root1,*root2;

int main(){
    int n;scanf("%d",&n);
    for(int i=0;i<n;i++){
        int a,b;scanf("%d%d",&a,&b);
        dc[i]=a;
        yy[i]=make_pair(b,a);
    }
    sort(dc,dc+n);
    sort(yy,yy+n,greater<pair<int,int> >());
    int ans=0;
    root1=new node(0,n-1);//prefix
    root2=new node(0,n-1);//suffix
    for(int i=0;i<n;i++){
        int y=yy[i].first,x=yy[i].second;
        int x2=lower_bound(dc,dc+n,x)-dc;
        int uv1=x+y,uv2=y-x;
        int q1=-1e9,q2=-1e9;
        if(x2)q1=root1->qu(0,x2-1);
        if(x2!=n-1)q2=root2->qu(x2+1,n-1);
        if(q1<uv1&&q2<uv2){
            ans++;
            root1->up(x2,uv1);
            root2->up(x2,uv2);
        }
    }
    printf("%d",ans);
}
```

Subtask 6

In Subtask 6, each point can only be covered by the point directly on the left or right. It can be easily checked in $O(1)$ for each point.

Note that for subtasks where $N \leq 10\,000\,000$, fast input is required to ensure the programme runs within 1 second.

Sample C++ code (with time complexity of $O(N)$):

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

int x[10000009],y[10000009];

inline int readInt() {
    int x=0; char ch=getchar_unlocked(); bool s=1;
    while(ch<'0'||ch>'9'){if(ch=='-')s=0;ch=getchar_unlocked();}
    while(ch>='0'&&ch<='9'){x=(x<<3)+(x<<1)+ch-'0';ch=getchar_unlocked();}
    return s?x:-x;
}

int main(){
    int n=readInt();
    for(int i=0;i<n;i++){
        x[i]=readInt(),y[i]=readInt();
    }
    int ans=0;
    for(int i=0;i<n;i++){
        if(y[i]==1)ans++;
        else if((i==0||y[i-1]==0)&&(i==n-1||y[i+1]==0))ans++;
    }
    printf("%d",ans);
}
```

Subtask 7

In Subtask 7, there are at most 10 000 000 points ($N \leq 10\,000\,000$)

We observe that we can first assume all points are chosen, then pick out points which are not covered by other points. We can perform a similar check in subtask 5. We pick a point (X_i, Y_i) if $X_i + Y_i$ is higher than all of $X_j + Y_j$ for $i > j$, and $Y_i - X_i$ is higher than all of $Y_j - X_j$ for $i < j$. This can be done with a prefix max of $X_i + Y_i$ and suffix max of $Y_i - X_i$, for an overall complexity of $O(N)$.

Sample C++ code (with time complexity of $O(N)$):

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

inline int readInt() {
    int x=0; char ch=getchar_unlocked(); bool s=1;
    while(ch<'0'||ch>'9'){if(ch=='-')s=0;ch=getchar_unlocked();}
    while(ch>='0'&&ch<='9'){x=(x<<3)+(x<<1)+ch-'0';ch=getchar_unlocked();}
    return s?x:-x;
}
```

```
int pref[10000009],suff[10000009];

int main(){
    int n=readInt();
    for(int i=0;i<n;i++){
        int x=readInt(),y=readInt();
        pref[i]=x+y;
        suff[i]=y-x;
        if(i)pref[i]=max(pref[i],pref[i-1]);
    }
    int ans=0;
    for(int i=n-1;i>=0;i--){
        ans+=((i==0)||pref[i]>pref[i-1])&&((i==n-1)||suff[i]>suff[i+1]);
        if(i!=n-1)suff[i]=max(suff[i],suff[i+1]);
    }
    printf("%d",ans);
}
```

Alternative solution

If we make further observations, we notice that we can store chosen points in a stack, where the top is the rightmost chosen point. Considering each point from left to right, either the new point is covered by the top of the stack, or the new point covers a number of points from the top of the stack, which can be recursively popped. This leads us to a solution that only iterates through the array once.

Sample C++ code (with time complexity of $O(N)$):

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

inline int readInt() {
    int x=0; char ch=getchar_unlocked(); bool s=1;
    while(ch<'0'||ch>'9'){if(ch=='-')s=0;ch=getchar_unlocked();}
    while(ch>='0'&&ch<='9'){x=(x<<3)+(x<<1)+ch-'0';ch=getchar_unlocked();}
    return s?x:-x;
}

stack<pair<int,int> >s;

int main(){
    int n=readInt();
    for(int i=0;i<n;i++){
        int x=readInt(),y=readInt();
        bool add=1;
        while(s.size()){
            int tx=s.top().first,ty=s.top().second;
            if(x-tx<=ty-y){add=0;break;}
            if(x-tx<=y-ty)s.pop();
            else break;
        }
        if(add)s.push(make_pair(x,y));
    }
    printf("%d",(int)s.size());
}
```

Task 4: City Mapping

Author: Pang Wen Yuen
Singapore IOI Team 2015-2017
pwypeanut@gmail.com

Introduction

The abridged problem: You are not given a weighted tree of up to $N = 1000$ vertices, and each node in the tree has at most degree 3. You are allowed $Q = 6500$ queries to find the entire tree, and each query gives you the sum of weights of the shortest path between two nodes.

Subtasks

Subtask 1

In Subtask 1, we see that we are allowed $Q = 500000$ queries, and the weights of the graph are all 1. This means we are allowed to query all pairs of nodes as $\binom{1000}{2} = 499500 < 500000$. Since the weights of the tree are all 1, the pairs of nodes seperated by distance 1 are simply nodes that are connected. So, we can check through all pairs of nodes and output the pairs which are distance 1 apart as edges. This should be a simple subtask to obtain, as long as the contestant understands the problem and knows how to utilise the function call interface.

Subtask 2

In Subtask 2, the tree is now weighted, but we are still allowed effectively unlimited queries. Notice that this subtask reduces to the Minimum Spanning Tree (MST) problem. We can construct a complete graph, with the edge weight between two nodes x and y equal to $query\_distance(x, y)$. Then, we can utilise an MST algorithm such as Kruskal's or Prim's to obtain the MST of this graph, which will be the correct tree. This subtask, together with the first, gives 25 points.

Subtask 3

Subtask 3 is the first subtask where the number of queries is limited. Now we have $Q = 12000$. But the additional constraint ensures the tree is an unweighted line. Let's arbitrarily root the line at node 1. The line will now extend to the *left* and *right* of node 1, assuming it is not a leaf node (the algorithm covers the other case as well).

We can first query the distance between node 1 and all other nodes in the line, then process these nodes in increasing order of distance from node 1, adding them to the *found tree* one node at a time. For each new node to be processed, notice that since it has the shortest distance to the root among the unprocessed nodes, its path to the root must only pass through processed nodes, so it can be connected onto the *found tree.* This idea is key to solving the entire problem.

In addition, since the graph is a line, it must be either connected to the leftmost node or the rightmost node on the *found tree.* We can check this by querying the distance between the leftmost node on the *found tree* and the node to be processed. If the distance is 1, it is connected to the leftmost node, otherwise it is connected to the rightmost node.

This costs a total of $Q = 2000$ queries when $N = 1000$, which is far lower than the $Q = 12000$ required by the subtask constraints. This is so as to allow for algorithms which solve the more general version of the problem (involving a tree instead of a line) to solve this subtask as well.

Subtask 4

Subtask 4 has the same constraints as Subtask 3, except the line is now weighted. The algorithm to solve this subtask is as follows. First we arbitrarily root the tree at node 1, then we query the distances from node 1 to all other nodes on the line. We pick the furthest node away from node 1, and call it node X. Node X must lie on either end of the line, as otherwise there would be a node further away from node 1.

Let's take node X as the leftmost node, and imagine the line as extending from left to right. We query the distance from node X to all other nodes, and then sort the nodes in increasing order of their distance from node X. Notice that this order of nodes corresponds to their order from left to right. Therefore, there exists an edge between any two adjacent nodes in this order, and the edge weight would be the difference in their distances from node X.

This costs a total of $Q = 2000$ queries when $N = 1000$ as well, which would easily pass under the subtask constraints. The first 4 subtasks will yield a total of 57 marks.

Subtask 5

Subtask 5 offers the full problem, finding a weighted tree. It is a partial scoring subtask, allowing up to 25000 queries. The score varies with Q, giving 0 to 10 marks on a linear scale from $Q = 25000$ to $Q = 12000$, then 10 to 40 marks on a linear scale from $Q = 12000$ to $Q = 6501$, with the final 3 marks of the problem only given to solutions which pass under the $Q = 6500$ bound. Several subquadratic solutions lie within this range, with all of them having an expected $NlogN$ complexity in terms of Q, but with different constant factors.

The *Tree Binary Search* Approach

We arbitrarily root the tree at node 1, and query the distance from node 1 to all other nodes, then sort them by increasing order of distance. Similar to the idea for Subtask 3, we process these nodes one by one and connect it to our *found tree*. So, for each new node, we need to find the node on the *found tree* which it is directly connected to, and then connect it there. Let's call the node to be connected C.

First, let's consider the *found tree*. We do a Depth-First Search (DFS) on the *found tree* to obtain the number of found nodes each node has in its subtree. We now pick a node on the *found tree* such that the number of nodes in its subtree is as close to half of the total number of nodes in the tree as possible. We can always find a node where the number of nodes in its subtree is close to half. (Due to the limited degree of the tree, the best node will not deviate from half beyond a certain bound.) Let's call this node X.

We then query the distance from node C to node X. Using this information, we can actually find out whether the new node lies within the subtree of X or not. Since we know the distance from the root to all nodes, we can consider the distance between the root and X, the root and C, and between C and X.

If $d(root, X) + d(X, C) = d(root, C)$, then C must be in the subtree of X. Otherwise, C must not be in the subtree of X. Using this, we can prune approximately half of the tree away as potential candidates as connecting points for C. If we repeat this approximately $logN$ times, each time pruning half of the tree away, we can obtain the connecting point for C. Since there are N nodes to connect, the overall complexity is $NlogN$.

In the context of this problem, this solution obtains about $Q = 10000$, which gains approximately 77 marks in total.

The *Random Branching* Approach

This solution is a *randomised* algorithm that performs surprisingly well on this problem. We first start with a random root, let's call this node R. We then query the distances from node R to all other nodes. We pick a random auxiliary node in this pool. Let's call this node A, and we query the distance from node A to all other nodes as well.

Let's consider all nodes on the path between R and A. We can call this path the *main branch*. Let's consider a particular node X. X lies on the *main branch* iff $d(R, A) = d(R, X) + d(X, A)$. Therefore, we can find all the nodes and edges within the *main branch*, as we can sort these nodes in increasing order of their distance from R, and that would be the order of nodes from R to A.

In addition, for every node not on the *main branch*, we are able to obtain the point at which their path from R deviates from the *main branch.* How? Let's call the node at which the path deviates from the main branch M. Since we know $d(R, X)$, $d(R, A)$ and $d(X, A)$, and these three conditions hold: $d(R, X) = d(R, M) + d(M, X)$, $d(R, A) = d(R, M) + d(M, A)$ and $d(X, A) = d(X, M) + d(M, A)$, we can solve for $d(M, A)$, $d(M, R)$ and $d(M, X)$. This allows us to identify the node M on the *main branch* where the path deviates.

From there, we can split the problem up. For the *subtree* branching out at each node on the *main branch*, we have the distance from the node on the *main branch* to each node in that *subtree.* This reduces to the original problem, just on a smaller scale, where the new root is now the node on the *main branch.* We can pick another *auxiliary node* within that *subtree*, and repeat this process to obtain the entire *subtree.*

This algorithm works very well in practice, giving about $Q = 7800$ on the testset and obtaining about 90 points. Heuristics can be added to the algorithm, for example picking one of the 3 deepest nodes as *auxiliary nodes* to improve the algorithm's performance. Such heuristics cut Q to about 7000, obtaining about 94 points.

The Solution

Let's go back to the original idea proposed in the *Tree Binary Search* approach. The reason why that algorithm has a large constant factor is because each distance query only gives us one of two outcomes: in the subtree or not in the subtree, and thus we can only obtain one bit of information. This algorithm eliminates this weakness.

Again, we return to the idea of connecting each new node X to the *found tree.* We do the same thing, calculating the number of nodes in the subtree for each node in the *found tree.* Now, we start from the root R, and at every step, we traverse down toward the child with the maximum number of nodes in its subtree. We do this until we reach a leaf node. Let's call this leaf node L.

We now use one query to find the distance between nodes L and X. Consider the relationship between the nodes L, X and R. Again there will be a middle node M where the paths R-X and R-L diverge. Using $d(R, L)$, $d(L, X)$ and $d(R, X)$, we can obtain $d(M, R)$, $d(M, L)$ and $d(M, X)$.

We can now "reroot" the tree at M, and since we have $d(M, X)$, we can eliminate all nodes not in the subtree of M or in the path M-L. We then run the same algorithm outlined in paragraph 2 of this section, until the total number of possible nodes that X can be directly connected to drops to 1, then we simply connect X to that node and restart with a new node.

The method of finding L ensures that at least half of the tree is pruned out at each iteration (ignoring the edge case where the root has 3 children, in that case a third will be pruned out), and so it is $NlogN$. The fact that we can obtain more than one bit of information from each query also makes the constant factor of this algorithm much smaller. In the testset, which includes binary trees which is the worst case for this algorithm, the worst case found was about $Q = 6300$, well below the $Q = 6500$ bound required to obtain full marks for this problem.

Task 5: Safety

Author: Bernard Teo
Singapore IOI Team 2012-2013
bernardteo@u.nus.edu

Introduction

The abridged problem:
Given an array of N non-negative integers, find the minimum number of increment/decrement operations (on elements in the array) needed such that the all pairs of adjacent elements differ by no larger than H.

Subtasks

For all subtasks

For the analysis below, K represents the maximum possible stack height, i.e. $K = \max_{1 \le i \le N} \{S[i]\}$.

Observe that for any instance of this problem, if $H \ge K$, setting H to K will not affect the required output. More precisely, original artwork is definitely already safe when $H \ge K$, since the difference in height between any two stacks cannot exceed K. Hence, for subtasks which have a time complexity that depends on H, one can set H to $\min\{H, K\}$, or simply output `0` if $H \ge K$, in order to pass the time limit if the constraint on K is tighter than the constraint on H.

Subtask 1

Time complexity: $O(K^N N)$

Subtask 1 contains extremely small limits for N and K, which will allow a simple brute force solution to pass. We can search all possible $(K+1)^N$ possible sequences of length N, and if the sequence is safe, calculate the number of steps needed to get to it from the input sequence.

Subtask 2

Time complexity: $O((2H)^{N-1} NK)$

Subtask 2 contains slightly larger limits for N that will fail a solution that tries all possible sequences. However, the additional constraint that $H \leq 1$ means that we can search all possible *safe* sequences with a reduced branching factor (as compared to iterating all sequences then checking whether each of them is safe). There are $K(2H+1)^{N-1}$ safe sequences to check, giving us the required time complexity.

Subtask 3

Time complexity: $O((2H)^{N-1}N^2)$

The major change from Subtask 2 to Subtask 3 is the removal of the constraint for K. This means that stacks can be much higher and much more varied in height; but as with Subtask 2, the branching factor is still very small.

We thus need to find the height, after modification of the artwork, of just a single stack. After obtaining this value, we can search all possible safe sequences from there as with Subtask 2.

Finding this stack (and its height after modification) is the key observation in this subtask. We will show that at least one of the original stacks need not be modified (so we can try fixing the height of each stack, one at a time, and running the search).

Suppose we have found a safe sequence that requires all stacks to be modified (i.e. no stack retains its original height). Let x be the number of stacks that will decrease in height, and y be the number of stacks that will increase in height (so $x+y=N$). There are three cases:

1. $x < y$
2. $x = y$
3. $x > y$

The reasoning for Cases 1 and 3 are similar, so only Case 1 and 2 will be described in detail here.

Case 1: Consider a new safe sequence formed from removing a single cube from each stack in the safe sequence we have found. By doing this, we accrue one additional step for each stack counted in x, but we save one step for each stack counted in y, so there is an overall change of $x-y$. Since $x<y$, we know that $x-y<0$, i.e. there is an overall reduction in number of steps needed. This means that the safe sequence we have found cannot be one that minimizes the number of steps required.

Case 2: Again, consider a new safe sequence formed from removing a single cube from each stack in the safe sequence we have found. There will be no change in the number of steps required. However, since $x+y=N>0$, we know that $y>0$, so by repeatedly removing a single cube from each stack, we will eventually reach a sequence where at least one stack will not be modified. Hence, the required number of steps for the original safe sequence we have found would been equal to the required number of steps for at least one safe sequence that preserves the height of at least one stack.

Thus, we need to only search all possible safe sequences in which at least one stack is left unchanged, and there are $N(2H+1)^{N-1}$ such sequences.

Subtask 4

Time complexity: $O(N \log N)$ (but only valid when $H = 0$)

Unlike all other subtasks, Subtask 4 is not merely a slower solution that will eventually determine the correct answer. Subtask 4 is a special case of this question where we want to *equalise* the heights of all stacks.

By using a similar argument as Subtask 3 (where we increment or decrement the equalised height by one), it can be shown that the optimal equalised height is the median height of the given stacks.

A simple sorting-based median-finding algorithm with $O(N \log N)$ time complexity suffices for this subtask.

Subtask 5

Time complexity: $O(NHK)$

Subtask 5 and beyond requires the use of a dynamic programming solution.

Let $\text{best}(n, k)$ be the minimum number of increment/decrement operations required on stacks 1 to n such that the n^{th} stack has height exactly k.

Then we can construct the following recurrence relation (when $n > 1$):

$$\text{best}(n, k) = \min_{k'=\max(k-H,0)}^{\min(k+H,K)} \{\text{best}(n-1, k')\} + |k - S[n]|$$

When $n = 1$, the expression is simply: $\text{best}(1, k) = |k - S[1]|$.

Since the state complexity is $O(NK)$ and the transition time complexity is $O(H)$, we arrive at an overall time complexity of $O(NHK)$.

Subtask 6

Time complexity: $O(NK \log H)$

Building on our solution for the previous subtask, Subtask 6 requires a reduction of the transition time complexity to $O(\log H)$.

Computing $\text{best}(n, k)$ for a fixed n as we iterate from $k = 1$ to K, we notice that the set in which we have to pick the minimum element from, i.e. $\{\text{best}(n-1, k') \mid k' \in [\max(k-H, 0), \min(k+H, K)]\}$, only changes by the insertion of at most one element and removal of at most one element. We may hence use a multiset data structure to store these elements, allowing $O(\log H)$ insertion, removal, and minimum query.

Subtask 7

Time complexity: $O(NK)$

Subtask 7 requires a further reduction in transition time complexity to $O(1)$ (amortized). This may be implemented with a sliding range minimum query algorithm that maintains a double-ended queue (deque) of candidate values in increasing order, allowing the computation of $\min\limits_{k'=\max(k-H,0)}^{\min(k+H,K)} \{\text{best}(n-1, k')\}$ for all k, given $\text{best}(n-1, k)$ for all k, in $O(K)$ time complexity.

When a new element is to be added (to the back of the deque), all existing elements with value larger than or equal to this element are removed (from the back) before the new element is added to the back of the deque. In this way, the elements in the deque (from front to back) are always in increasing order of value and in decreasing order of age.

When the minimum value is to be queried, all elements that are now out of the required range (i.e. too old) are removed (from the front of the deque), and the oldest (i.e. frontmost) element remaining holds the minimum valid value, which is the answer for the minimum value query.

Since all K elements are added exactly once to this deque, the time complexity of processing all K elements (from $k = 1$ to K) is $O(K)$, resulting in the amortized $O(1)$ transition time complexity required.

As this is a rather well-known algorithm, further details may be found elsewhere.

Sample C++ code for Subtask 7:

```cpp
#include <cstdlib>
#include <iostream>
#include <utility>
#include <algorithm>
#include <deque>
#define MAX_HEIGHT 5000
using namespace std;
int back_arr[2][MAX_HEIGHT + 1];
inline void consider_insert(deque<pair<int, int>>& dq, int index, int value) {
    while(!dq.empty() && dq.back().second >= value) dq.pop_back();
    dq.emplace_back(index, value);
}
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, h;
    cin >> n >> h;
    int* arrprev = back_arr[0];
    int* arrcurr = back_arr[1];
    {
        int x;
        cin >> x;
        for(int j = 0; j <= MAX_HEIGHT; ++j) {
            arrprev[j] = abs(j - x);
        }
    }
    for(int i = 1; i < n; ++i) {
        int x;
        cin >> x;
        deque<pair<int, int>> dq; // index, value
        for(int j = 0; j < min(MAX_HEIGHT, h); ++j) {
            consider_insert(dq, j, arrprev[j]);
        }
        for(int j = 0; j <= MAX_HEIGHT; ++j){
            if(j + h <= MAX_HEIGHT) consider_insert(dq, j + h, arrprev[j + h]);
            arrcurr[j] = dq.front().second;
            arrcurr[j] += abs(j - x);
            if(dq.front().first == j - h) dq.pop_front();
        }
        swap(arrprev, arrcurr);
    }
    cout<< *min_element(arrprev, arrprev + MAX_HEIGHT + 1) << endl;
}
```

Subtask 8

Time complexity: $O(N^2)$

Subtasks 8 and 9 require a further observation that allows us to obtain best(n, k) for all k, given best$(n-1, k)$ for all k, in sub-$O(K)$ time complexity — meaning that we cannot store all the values of best(n, k) as a usual array.

For a given (fixed) n, plot a graph of best(n, k) against k.

It is easy to see that as $k \to -\infty$ or $+\infty$, best$(n, k) \to +\infty$. More specifically, when $k < 0$, best(n, k) increases as k decreases, and when $k > K$, best(n, k) increases as k increases. This means that best(n, k) when $k \notin [0, K]$ will never be in the path of an optimal solution (i.e. will never contribute to an optimal solution). This leads to a simplification of our original dynamic programming transition function to:

$$f(k) \coloneqq \text{best}(n, k) = \min_{k'=k-H}^{k+H} \{\text{best}(n-1, k')\} + |k - S[n]|$$

Decompose the dynamic programming transition function into two separate functions:

1. $g(k) = \min_{k'=k-H}^{k+H} \{\text{best}(n-1, k')\}$
2. $\text{best}(n, k) = g(k) + |k - S[n]|$

(For the analysis below, we use graph *to refer to the function $f(k)$ and related functions, in order to disambiguate from the functions produced by the decomposition, which are referred to as* functions.*)*

Assume, for now, that the graph is concave upwards (i.e. $f(k)$ is concave upwards). This function $f(k)$ may then be stored as a list of *critical* points $(k, \text{best}(n, k))$ — points at which the gradient of $f(k)$ changes.

Furthermore, the minimum points of this graph must be a continguous range inside the range $[0, K]$, and to the left of the minimum points $f(k)$ is decreasing, while to the right of the minimum points $f(k)$ is increasing.

Next, observe that both functions may be applied to our list of points in $O(P)$ time complexity, where P is the number of points in the graph:

1. Function 1 translates each point on the list by H units along the k-axis *away* from the minimum points of $f(x)$, expanding the range of minimum points by H units in both directions in the process.

   (This means that for the decreasing part of $f(k)$ (on the left of the minimum points) are translated to the left, while the increasing part of $f(k)$ (on the right of the minimum points) are translated to the right.)

2. Function 2 is an addition of the graph resulting from Function 1 with an absolute value function that has been shifted along the k-axis.

   Geometrically, Function 2 translates each point upwards by an amount equal to the distance between the k-coordinate of that point and $S[n]$, and adds a new point $(S[n], z)$ where z is the value of $g(S[n])$ (i.e. before applying Function 2).

Since both functions will produce an output graph that remains concave upwards (assuming the input graph is concave upwards already), and the case where $n = 1$ (which is just a shifted absolute value function) is concave upwards, by mathematical induction, our assumption (that the input graph is concave upwards) is valid for all $1 \leq n \leq N$.

While this concavity observation is not strictly necessary for Subtask 8, it may simplify the implementation in code. It will also be necessary for the Subtask 9.

Lastly, note that both Function 1 and Function 2 increases the number of points by at most one, so the number of points we need to store at a given n is at most $2n$. Hence, calculating $\text{best}(n, k)$ for all k, given $\text{best}(n-1, k)$ for all k, is an $O(N)$ operation, giving us an overall time complexity of $O(N^2)$ (since we need to process $n = 1, \ldots, N$).

Subtask 9

Time complexity: $O(N \log N)$

Subtask 9 requires the concavity observation from the previous subtask, as well as a different way of storing the points, to allow both functions to be applied to the graph in at most $O(\log N)$ time complexity.

We first expand the list of points, by adding duplicate points in the list, such that the gradient of the line segment between each adjacent pair of points differ by exactly 1 unit when compared to adjacent line segments. (More precisely, to the left of the minimum points, each line segment has a gradient of 1 unit less than the line segment to its right (i.e. gradients get more negative as we go to the left); and to the right of the minimum points, each line segment has a gradient of 1 unit more than the line segment to its left (i.e. gradients get more positive as we go to the right).)

Next, observe that the storing the dependent coordinate (i.e. the value of $f(k)$ for each k) is redundant; we are able to reconstruct the dependent coordinate based on the points stored in our (expanded) list, as long as we store the minimum value of $f(k)$.

Thus, we store these critical k-coordinates in two separate lists — the *left* list contains those on the left of the minimum points, and the *right* list contains those on the right of the minimum points. Both lists are kept sorted in increasing order of distance from the minimum points. (Note that the leftmost and rightmost minimum points are considered critical k-coordinates, and so they are stored at the front of the left and right lists respectively.) The value of $f(k)$ at the minimum point is stored separately as a single integer.

By storing our points in this manner, both functions (as defined in Subtask 8) become very straightforward:

1. Function 1 is the subtraction of H to all elements in the left list, and the addition of H to all elements in the right list.
2. Function 2 is the insertion of the value $S[n]$ twice into the correct list at the correct position (to preserve ordering of the k-coordinates), followed by a "rotation" from the modified list to the other list (removing the first

element of the modified list, and then inserting it to the front of the second list).

Function 1 may be realised as an $O(1)$ operation by storing an "offset" for each list that applies to all elements in that list.

By using a priority queue or multiset data structure, Function 2 may be implemented efficiently in $O(\log N)$.

Hence, a priority queue (or multiset) with offset allows both operations can be done in at most $O(\log N)$ time complexity, achieving an overall time complexity of $O(N \log N)$.

As we do not store the dependent coordinate explicitly, we can no longer immediately locate the minimum value required for output. To obtain this value, we need to keep track of the minimum value through each iteration of n (for $n = 1, \ldots, N$), and increase the minimum value by the correct amount after every application of Function 2. Given $S[n]$ and the left and right k-coordinates of the (existing) minimum range, computing the correct amount to add to the minimum value is trivial.

(The keen reader might notice that the offset may be determined from n, since the offset is increased by H on every iteration of n. By doing this, we need not explicitly store the offset.)

Sample C++ code for the full solution (using `std::priority_queue`):

```
#include <cstdlib>
#include <iostream>
#include <vector>
#include <queue>
#include <functional>
using namespace std;
priority_queue<long long, vector<long long>, less<long long>> lefts;
priority_queue<long long, vector<long long>, greater<long long>> rights;
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    int n, h;
    cin >> n >> h;
    {
        int x;
        cin >> x;
        lefts.push(x);
        rights.push(x);
    }
    long long minval = 0;
    for(int i = 1; i < n; ++i) {
        int x;
        cin >> x;
        long long shift = static_cast<long long>(i) * h;
        long long leftborderval = lefts.top() - shift;
        long long rightborderval = rights.top() + shift;
        if(x < leftborderval) {
            lefts.push(x + shift);
            lefts.push(x + shift);
            lefts.pop();
            rights.push(leftborderval - shift);
            minval += abs(leftborderval - x);
        }
        else if(x > rightborderval) {
            rights.push(x - shift);
            rights.push(x - shift);
            rights.pop();
            lefts.push(rightborderval + shift);
            minval += abs(rightborderval - x);
        }
        else {
            lefts.push(x + shift);
            rights.push(x - shift);
        }
    }
    cout << minval << endl;
}
```

Sample C++ code for the full solution (using `std::multiset`):

```
#include <cstdlib>
#include <iostream>
#include <algorithm>
#include <set>
#include <functional>
using namespace std;
multiset<long long, greater<long long>> lefts;
multiset<long long, less<long long>> rights;
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
    int n,h;
    cin >> n >> h;
    {
        int x;
        cin >> x;
        lefts.insert(x);
        rights.insert(x);
    }
    long long minval = 0;
    for(int i = 1; i < n; ++i) {
        int x;
        cin >> x;
        long long shift = static_cast<long long>(i) * h;
        auto leftborder = lefts.begin();
        long long leftborderval = *leftborder - shift;
        auto rightborder = rights.begin();
        long long rightborderval = *rightborder + shift;
        if(x < leftborderval) {
            auto hint = lefts.insert(x + shift);
            lefts.insert(hint, x + shift);
            lefts.erase(leftborder);
            rights.insert(rightborder, leftborderval - shift);
            minval += abs(leftborderval - x);
        }
        else if(x > rightborderval) {
            auto hint = rights.insert(x - shift);
            rights.insert(hint, x - shift);
            rights.erase(rightborder);
            lefts.insert(leftborder, rightborderval + shift);
            minval += abs(rightborderval - x);
        }
        else {
            lefts.insert(leftborder, x + shift);
            rights.insert(rightborder, x - shift);
        }
    }
    cout << minval << endl;
}
```