**USACO 2016 US Open Contest, Platinum**

**Problem 1. 262144**

Bessie likes downloading games to play on her cell phone, even though she does find the small touch screen rather cumbersome to use with her large hooves.

She is particularly intrigued by the current game she is playing. The game starts with a sequence of NN positive integers (2≤N≤262,1442≤N≤262,144), each in the range 1…401…40. In one move, Bessie can take two adjacent numbers with equal values and replace them a single number of value one greater (e.g., she might replace two adjacent 7s with an 8). The goal is to maximize the value of the largest number present in the sequence at the end of the game. Please help Bessie score as highly as possible!

#### INPUT FORMAT (file 262144.in):

The first line of input contains NN, and the next NN lines give the sequence of NN numbers at the start of the game.

#### OUTPUT FORMAT (file 262144.out):

Please output the largest integer Bessie can generate.

#### SAMPLE INPUT:

4

1

1

1

2

#### SAMPLE OUTPUT:

3

In this example shown here, Bessie first merges the second and third 1s to obtain the sequence 1 2 2, and then she merges the 2s into a 3. Note that it is not optimal to join the first two 1s.

Problem credits: Mark Chen

Top of Form

Note: Many issues (e.g., uninitialized variables, out-of-bounds memory access) can cause a program to product different output when run multiple times; if your program behaves in a manner inconsistent with the official contest results, you should probably look for one of these issues. Timing can also differ slightly from run to run, so it is possible for a program timing out in the official results to occasionally run just under the time limit in analysis mode, and vice versa. Note also that we have recently changed grading servers, and since our new servers run at different speeds from the servers used during older contests, timing results for older contest problems may be slightly off until we manage to re-calibrate everything properly.

Bottom of Form

(Analysis by Mark Gordon)

A simple way to approach this problem would be to consider all ranges of the input array and determine the largest number that can be produced in that range. However, most ranges aren't actually interesting as they could never be combined into one.

To see this it helps to look at the equivalent problem where each of the array elements are powers of two and instead of combining x and x to produce x + 1 you produced 2x. Now it's clear that a range must sum to a power of two to be interesting. In fact, an interesting range can be better described by its starting position and the power of two it sums to.

This informs a simple Dynamic Programming solution. We let DP[p][i] give the ending index of the range starting at i that can combine to p, or -1 if it doesn't exist. DP[p + 1][i] is then calculated as DP[p + 1][i] = DP[p][DP[p][i]] provided DP[p][i] is valid.

Here's my solution to this problem.

#include <iostream>

#include <cstdio>

#include <vector>

using namespace std;

#define MAXN ((1 << 18) + 10)

#define MAXSZ 70

int dp[MAXSZ + 1][MAXN];

int A[MAXN];

int main() {

int N; cin >> N;

vector<int> A(N);

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

cin >> A[i];

}

int result = 0;

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

for (int j = 0; j < N; j++) {

if (A[j] == i) {

dp[i][j] = j + 1;

result = max(result, i);

} else {

if (i == 0 || dp[i - 1][j] == -1 || dp[i - 1][dp[i - 1][j]] == -1) {

dp[i][j] = -1;

} else {

dp[i][j] = dp[i - 1][dp[i - 1][j]];

result = max(result, i);

}

}

}

dp[i][N] = -1;

}

cout << result << endl;

return 0;

}

**Further analysis contributed by Kyle Liu:** There is an alternative O(N)O(N) greedy approach. An O(NlogN)O(Nlog⁡N) greedy solution is obvious. We can remove the lowest value (MM) by greedily combining KK consecutive pairs of MM into K/2K/2 pairs of (M+1M+1). In case that KK is odd, we can simply break the sequence into two and assign the K/2K/2 pairs of M+1M+1 to both sequences. Repeating this process will give us an O(NlogN)O(Nlog⁡N) solution, using appropriate data structures.

O(N)O(N) can be achieved since we don't have to always find lowest value to remove. Consider the sequence of numbers as heights of hills. We can simply find the "valley points" (point whose heights are below its neighbours') to remove. We first condense the sequence into consecutive intervals of same heights. We use a stack to keep track of the sequence and "valley point". As we go through the list of intervals, if the stack is empty or the incoming height is below the height in the top of stack (downhill), we simply push the incoming interval to the stack. If the incoming height is above the height in the top of stack (uphill), the point at the top of the stack is a "valley point", and it needs to be removed by combining into its neighbouring intervals. Its left neighbours are in the stack and its right neighbour is the incoming interval. If any combination needs to break into two sequences. We can calculate the optimal value of the first sequence by "collapsing" the stack. We then start the second sequence with only the "valley point" in the stack.

Here is my code implementing this approach:

#include <stdio.h>

#include <iostream>

#include <math.h>

using namespace std;

#define MAXN 262144+10

struct Node {

int val;

int tot;

};

Node ar[MAXN];

Node s[MAXN];

int N, top = 0, res = 0;

void collapse\_stack(void) // calculate value for first squence and reset stack

{

for (; top > 1; top--)

s[top-2].tot += s[top-1].tot / (1 << (s[top-2].val - s[top-1].val));

res = max(res, s[top-1].val + (int)log2(s[top-1].tot));

top--;

}

void combine\_left(int val) // combine the left side until height reaches val

{

for (; top > 1; top--) {

if(s[top-2].val > val) break;

int num = 1 << (s[top-2].val - s[top-1].val);

if (s[top-1].tot % num) {

Node tmp = s[top-1];

collapse\_stack();

s[top++] = tmp; // start second sequence with the "valley point"

break;

}

s[top-2].tot += s[top-1].tot / num;

}

}

int main(void)

{

freopen("262144.in","r",stdin);

freopen("262144.out","w",stdout);

cin >> N;

int st = 0;

for(int i=1; i<=N; i++) {

int a;

cin >> a;

res = max(res, a);

if(a == ar[st].val) ar[st].tot++;

else {

ar[++st].val = a;

ar[st].tot++;

}

}

for(int i=1; i<=st; i++) {

if (top == 0 || (ar[i].val < s[top-1].val)) { // downhill, add to stack

s[top++] = ar[i];

continue;

}

combine\_left(ar[i].val);

int num = 1 << (ar[i].val - s[top-1].val);

if (s[top-1].tot % num == 0) { // combine new interval into stack

s[top-1].val = ar[i].val;

s[top-1].tot = ar[i].tot + s[top-1].tot / num;

}

else { // new intervals cannot be merged to intervals already in stack

ar[i].tot += s[top-1].tot / num;

collapse\_stack();

s[top++] = ar[i];

}

}

collapse\_stack(); // obtain answer for remaining intervals in stack

cout << res << endl;

return 0;

}

## USACO 2016 US Open Contest, Platinum

## Problem 2. Bull in a China Shop

Farmer John has decided his home needs more decoration. Visiting the local china shop, he finds a delicate glass cow figurine that he decides to purchase, knowing that it will fit perfectly on the mantel above his fireplace.

The shape of the cow figurine is described by an N×MN×M grid of characters like the one below (3≤N,M≤5003≤N,M≤500), where lowercase letter characters are each part of the figurine (indicating different colors) and '.' characters are not.

...............

...............

x..x...........

xxxx...........

xxxxaaaaaaa...

.xx.aaaaaaaaa..

....aaaaaaa.aa.

....ll...ll....

....vv...vv....

...............

Unfortunately, right before FJ can make his purchase, a bull runs through the shop and breaks not only FJ's figurine, but many of the other glass objects on the shelves as well! FJ's figurine breaks into 3 pieces, which quickly become lost among KK total pieces lying on the ground (4≤K≤1004≤K≤100). Each of the KK pieces is described by a grid of characters, just like the original figurine.

Please help FJ determine how many sets of 3 pieces (out of the KK on the floor) could be glued back together to mend his broken figurine.

The pieces on the ground might have been flipped vertically or horizontally, or rotated by some multiple of 90 degrees. Therefore, given the original grid as well as KK grids describing pieces, you want to find sets of 3 pieces that can be joined together to form the original picture, allowing the pieces to be translated, flipped, or rotated multiples of 90 degrees. When then superimposed, the 3 pieces should exactly form the original picture, with each colored square in the original picture represented in exactly one of the pieces.

#### INPUT FORMAT (file bcs.in):

The first line contains a single integer KK. Following that will be K+1K+1 piece descriptions. The first description will describe the original glass cow, the following KK descriptions will be of the broken pieces.

Each description begins with a line containing two integers RR and CC (1≤R,C≤1001≤R,C≤100). The following RR lines contain CClowercase alphabet characters describing the color of each cell. Each piece will be horizontally/vertically connected and have at least one non-empty cell.

#### OUTPUT FORMAT (file bcs.out):

Output the number of triples i,j,ki,j,k (i<j<ki<j<k) such that pieces ii, jj, and kk can be arranged to form the original glass cow.

#### SAMPLE INPUT:

5

5 5

aaaaa

..a..

bbabb

..a..

aaaaa

3 5

..abb

..a..

aaaaa

5 2

a.

a.

aa

a.

a.

1 2

bb

1 5

bbabb

2 5

aaaaa

..a..

#### SAMPLE OUTPUT:

3

The three solutions use pieces (0,1,2)(0,1,2), (0,2,4)(0,2,4), (1,3,4)(1,3,4).

Note that this problem has a time limit of 6 seconds per test case (and twice that for Java and Python submissions).

Problem credits: Brian Dean

Top of Form

Note: Many issues (e.g., uninitialized variables, out-of-bounds memory access) can cause a program to product different output when run multiple times; if your program behaves in a manner inconsistent with the official contest results, you should probably look for one of these issues. Timing can also differ slightly from run to run, so it is possible for a program timing out in the official results to occasionally run just under the time limit in analysis mode, and vice versa. Note also that we have recently changed grading servers, and since our new servers run at different speeds from the servers used during older contests, timing results for older contest problems may be slightly off until we manage to re-calibrate everything properly.

Bottom of Form

(Analysis by Mark Gordon)

The most basic approach to this problem would loop over tuples of three pieces and try all offsets and orientations of the pieces to try and make the figurine. Obviously, this approach would be too slow with k3k3 tuples and (8N2)3(8N2)3 ways to arrange them.

However, The need to try all offsets can be eliminated by simply trying the offset such that the bottom most (breaking ties by rightmost) uncovered cell is covered by the new piece. We can do this because this cell must eventually be covered and there is at most one way to do it for a given piece and orientation.

This observation alone can bring us down to a O(8k3N2)O(8k3N2) solution. The next piece of the puzzle is to use [polynomial hashing](https://www.reddit.com/r/usaco/comments/39qrta/string_hashing/) to eliminate a O(k)O(k) factor on the critical path. After placing the first two pieces we can calculate the appropriate offset of the final piece and quickly test if its hash is one of the input pieces.

The final bit of the puzzle is how to calculate offsets quickly. This amounts to being able to quickly calculate the position of the bottom most, right most uncovered cell. This can be done in O(logN)O(logN) time by precomputing the suffix sums in [Row Major Order](https://en.wikipedia.org/wiki/Row-major_order) and binary searching to find the last non-zero suffix sum when subtracting out the suffix sums of already placed pieces.0

Here's my solution to this problem annotated with what each section of code aims to accomplish.

#include <iostream>

#include <vector>

#include <unordered\_map>

#include <map>

#include <set>

#include <algorithm>

#include <cstring>

#include <cstdio>

using namespace std;

typedef vector<string> pat;

int POLYMOD[2] = {

975919579,

975979579,

};

int POLYMUL[2] = {

382737283,

382878283,

};

#define MAXCOL 1010

#define MAXROW 510

#define HASHES 2

#define MAXPOW (MAXROW \* MAXCOL)

int POWTAB[HASHES][MAXPOW];

void init\_tab() {

if (POWTAB[0][0]) {

return;

}

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

for (int j = POWTAB[i][0] = 1; j < MAXPOW; j++) {

POWTAB[i][j] = (1ll \* POWTAB[i][j - 1] \* POLYMUL[i]) % POLYMOD[i];

}

}

}

/\* Tracks a polynomial hash over a 2D array. \*/

struct xhash {

xhash() {

init\_tab();

memset(H, 0, sizeof(H));

}

xhash(const pat& p) {

init\_tab();

memset(H, 0, sizeof(H));

/\* Calculate the hash of the given input matrix. We linearize the array by

\* setting a[r \* MAXCOL + c] = p[i][j] and then apply a standard polynomial

\* hash. \*/

for (int i = 0; i < p.size(); i++) {

for (int j = 0; j < p[0].size(); j++) {

if (p[i][j] == '.') {

continue;

}

/\* We set v this way to ensure that v\_1 - v\_2 could never represent a

\* valid character. This is important for ensuring the integrity of

\* subtracting two hashes. \*/

int v = 26 + (p[i][j] - 'a');

for (int k = 0; k < HASHES; k++) {

H[k] = (H[k] + 1ll \* POWTAB[k][i \* MAXCOL + j] \* v) %

POLYMOD[k];

}

}

}

}

void offset(int r, int c) {

/\* Offsetting the matrix by (r, c) translates into offsetting the

\* linearized array by r \* MAXCOL + c. Therefore we multiply each hash

\* by x^(r \* MAXCOL + c). \*/

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

H[i] = (1ll \* H[i] \* POWTAB[i][r \* MAXCOL + c]) % POLYMOD[i];

}

}

/\* Compute the difference of hashes. This gives you a hash of what would

\* remain in \*this if you got rid of everything present in x. In the case

\* that x isn't actually a subset of \*this the hash should just represent

\* garbage and won't get matched. \*/

xhash operator-(const xhash& x) const {

xhash nh;

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

nh.H[i] = H[i] - x.H[i];

if (nh.H[i] < 0) {

nh.H[i] += POLYMOD[i];

}

}

return nh;

}

bool operator==(const xhash& x) const {

return !memcmp(H, x.H, sizeof(H));

}

bool operator<(const xhash& x) const {

return memcmp(H, x.H, sizeof(H)) < 0;

}

int H[HASHES];

};

/\* For use in C++'s unordered\_map. \*/

struct xhash\_downhash {

int operator()(const xhash& h) const {

return h.H[0];

}

};

/\* Vertically flips the pattern. \*/

void vflip(pat& p) {

int R = p.size();

for (int i = 0; i < R - i - 1; i++) {

p[i].swap(p[R - i - 1]);

}

}

/\* Rotates the pattern 90 degrees. \*/

void rotate(pat& p) {

int R = p.size();

int C = p[0].size();

pat op(C, string(R, '.'));

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

for (int j = 0; j < C; j++) {

op[C - j - 1][i] = p[i][j];

}

}

p = op;

}

/\* Read in a pattern and canonicalize it. \*/

pat read\_pat() {

int R, C;

cin >> R >> C;

pat res(R);

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

cin >> res[i];

}

/\* Remove unneeded padding from the sides. \*/

int mnr = R, mxr = 0;

int mnc = C, mxc = 0;

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

for (int j = 0; j < C; j++) {

if (res[i][j] != '.') {

mnr = min(mnr, i);

mxr = max(mxr, i + 1);

mnc = min(mnc, j);

mxc = max(mxc, j + 1);

}

}

}

pat nres;

for (int i = mnr; i < mxr; i++) {

nres.push\_back(res[i].substr(mnc, mxc - mnc));

}

/\* Try all orientations and take the lexicographically least one. This

\* ensures that all equivalant piece representations are actually equal. \*/

res = nres;

for (int i = 0; i < 2; i++) {

for (int j = 0; j < 4; j++) {

if (nres < res) {

res = nres;

}

rotate(nres);

}

vflip(nres);

}

return res;

}

typedef vector<vector<int> > sum\_table;

sum\_table target\_sums;

vector<sum\_table> piece\_sums;

/\* Calculate the number of non-empty entries after every position in the

\* linearized array. This gets used in find\_offset later. \*/

sum\_table calc\_sums(const pat& p) {

int N = p.size();

int M = p[0].size();

sum\_table sums(N, vector<int>(M));

int lst = 0;

for (int i = N - 1; i >= 0; i--) {

for (int j = M - 1; j >= 0; j--) {

if (p[i][j] != '.') {

lst++;

}

sums[i][j] = lst;

}

}

return sums;

}

/\* Efficiently find the last non-zero position in base after subtracting

\* (possibly) several other matricies. \*/

pair<int, int> find\_offset(const sum\_table& base,

const vector<int>& subtract\_inds = {},

const vector<pair<int, int> >& offsets = {}) {

/\* Binary search over the linearized array. \*/

int N = base.size();

int M = base[0].size();

int lo = 0;

int hi = N \* M - 1;

while (lo < hi) {

int md = (lo + hi + 1) / 2;

int r = md / M;

int c = md % M;

/\* Calculate how many non-zero entries there are after (r, c) in the

\* linearized array. \*/

int val = base[r][c];

for (int i = 0; i < subtract\_inds.size(); i++) {

int j = subtract\_inds[i];

int er = r - offsets[i].first;

int ec = c - offsets[i].second;

int PN = piece\_sums[j].size();

int PM = piece\_sums[j][0].size();

if (ec >= PM) {

er++;

ec = 0;

}

if (er >= PN) {

/\* Do nothing. \*/

} else if (er < 0) {

val -= piece\_sums[j][0][0];

} else {

val -= piece\_sums[j][er][max(0, ec)];

}

}

/\* Search right if there are more non-zero entries, otherwise search left.

\*/

if (val) {

lo = md;

} else {

hi = md - 1;

}

}

return make\_pair(lo / M, lo % M);

}

int main() {

freopen("bcs.in", "r", stdin);

freopen("bcs.out", "w", stdout);

int K; cin >> K;

pat target = read\_pat();

int R = target.size();

int C = target[0].size();

xhash target\_hash = target;

target\_sums = calc\_sums(target);

/\* Read in the pieces into a map after canonicalization. Keep track of

\* how many times an equivalant piece occurs and deduplicate. \*/

map<pat, int> piece\_map;

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

piece\_map[read\_pat()]++;

}

/\* Build data structures based on the deduplicated pieces in each of

\* their orientations. \*/

vector<int> piece\_counts;

vector<pair<int, int> > piece\_offsets;

vector<int> piece\_index;

vector<xhash> piece\_hashes;

unordered\_map<xhash, int, xhash\_downhash> the\_hash;

for (auto it : piece\_map) {

pat p = it.first;

int index = piece\_counts.size();

piece\_counts.push\_back(it.second);

for (int j = 0; j < 2; j++) {

for (int k = 0; k < 4; k++) {

if (p.size() > R || p[0].size() > C) {

rotate(p);

continue;

}

xhash h(p);

piece\_sums.push\_back(calc\_sums(p));

piece\_offsets.push\_back(find\_offset(piece\_sums.back()));

piece\_hashes.push\_back(h);

piece\_index.push\_back(index);

h.offset(R - piece\_offsets.back().first,

C - piece\_offsets.back().second);

/\* Deduplication ensures that no two pieces should have the same hash.

\*/

the\_hash[h] = index;

rotate(p);

}

vflip(p);

}

}

set<tuple<int, int, int> > sols;

pair<int, int> base\_1 = find\_offset(target\_sums);

for (int i = 0; i < piece\_hashes.size(); i++) {

/\* Find the offset so that the last item in piece\_hashes[i] covers the last

\* item in the target. \*/

pair<int, int> off\_1 = base\_1;

off\_1.first -= piece\_offsets[i].first;

off\_1.second -= piece\_offsets[i].second;

if (off\_1.first < 0 || off\_1.second < 0) {

continue;

}

xhash hash\_1 = piece\_hashes[i];

hash\_1.offset(off\_1.first, off\_1.second);

pair<int, int> base\_2 = find\_offset(target\_sums, {i}, {off\_1});

for (int j = 0; j < piece\_hashes.size(); j++) {

/\* Find the offset so that the last uncovered item in \* piece\_hashes[j]

\* covers the last still uncovered item in the target. \*/

pair<int, int> off\_2 = base\_2;

off\_2.first -= piece\_offsets[j].first;

off\_2.second -= piece\_offsets[j].second;

if (off\_2.first < 0 || off\_2.second < 0) {

continue;

}

xhash hash\_2 = piece\_hashes[j];

hash\_2.offset(off\_2.first, off\_2.second);

/\* Canonicalize the position of the last still uncovered item of the

\* target so we can look up if we have a matching hash. \*/

pair<int, int> off\_3 = find\_offset(target\_sums, {i, j}, {off\_1, off\_2});

if (off\_3.first < 0 || off\_3.second < 0) {

continue;

}

xhash hash\_remains = target\_hash - hash\_1 - hash\_2;

hash\_remains.offset(R - off\_3.first, C - off\_3.second);

/\* Check if we have a match, if we do insert it into the solutions list.

\*/

auto it = the\_hash.find(hash\_remains);

if (it != the\_hash.end()) {

int L[3];

L[0] = piece\_index[i];

L[1] = piece\_index[j];

L[2] = it->second;

sort(L, L + 3);

sols.insert(make\_tuple(L[0], L[1], L[2]));

}

}

}

/\* Convert the solutions list into an actual result on the input array. This

\* takes into account the pieces that were deduplicated in the beginning. \*/

int result = 0;

for (auto sol : sols) {

int c0 = piece\_counts[get<0>(sol)];

int c1 = piece\_counts[get<1>(sol)];

int c2 = piece\_counts[get<2>(sol)];

if (get<0>(sol) == get<2>(sol)) {

result += c0 \* (c0 - 1) \* (c0 - 2) / 6;

} else if (get<0>(sol) == get<1>(sol)) {

result += c0 \* (c0 - 1) / 2 \* c2;

} else if (get<1>(sol) == get<2>(sol)) {

result += c0 \* c1 \* (c1 - 1) / 2;

} else {

result += c0 \* c1 \* c2;

}

}

cout << result << endl;

return 0;

}

## USACO 2016 US Open Contest, Platinum

## Problem 3. Landscaping

Farmer John is building a nicely-landscaped garden, and needs to move a large amount of dirt in the process.

The garden consists of a sequence of NN flowerbeds (1≤N≤100,0001≤N≤100,000), where flowerbed ii initially contains AiAi units of dirt. Farmer John would like to re-landscape the garden so that each flowerbed ii instead contains BiBi units of dirt. The AiAi's and BiBi's are all integers in the range 0…100…10.

To landscape the garden, Farmer John has several options: he can purchase one unit of dirt and place it in a flowerbed of his choice for XX units of money. He can remove one unit of dirt from a flowerbed of his choice and have it shipped away for YYunits of money. He can also transport one unit of dirt from flowerbed ii to flowerbed jj at a cost of ZZ times |i−j||i−j|. Please compute the minimum total cost for Farmer John to complete his landscaping project.

#### INPUT FORMAT (file landscape.in):

The first line of input contains NN, XX, YY, and ZZ (0≤X,Y≤108;0≤Z≤10000≤X,Y≤108;0≤Z≤1000). Line i+1i+1 contains the integers AiAi and BiBi.

#### OUTPUT FORMAT (file landscape.out):

Please print the minimum total cost FJ needs to spend on landscaping.

#### SAMPLE INPUT:

4 100 200 1

1 4

2 3

3 2

4 0

#### SAMPLE OUTPUT:

210

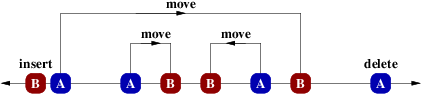
Note that this problem has been asked in a previous USACO contest, at the silver level; however, the limits in the present version have been raised considerably, so one should not expect many points from the solution to the previous, easier version.

Problem credits: Brian Dean

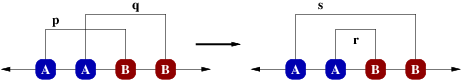
(Analysis by Brian Dean)

Letting K=10K=10 be the maximum amount of dirt in any cell, we can ultimately solve this problem using dynamic programming in O(NK)O(NK) time, which runs in less than 0.1 seconds on every test case. However, it takes a number of structural insights to get to this point. There are several ways to attack this problem, many based on more advanced network flow techniques like min-cost flows, which are perhaps slightly beyond the typical scope of USACO contests. For simplicity, we focus here on a solution that only uses dynamic programming, and no advanced flow techniques. We may soon add extra detail from other solutions that use flow-based techniques, for completeness.

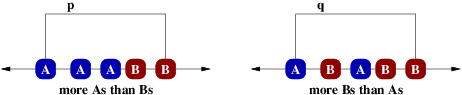
For starters, it may help to review the O((NK)2)O((NK)2) [solution](http://www.usaco.org/current/data/sol_landscape.html) of the earlier version of this problem, which appeared on the silver 2012 March USACO contest. As in that problem, we first unpack each array into a list of O(NK)O(NK) dirt locations. For example, a landscape with heights 3,1,4,1 turns into the sequence 0,0,0,1,2,2,2,2,3 (e.g., there are 4 units of dirt at position 2). The previous solution involved using an "edit distance" style DP algorithm to transform the initial sequence of dirt locations into the target sequence in quadratic time.

It will be convenient to visualize this as matching a set of points on a number line, where the As represent locations of the source dirt and Bs are the target dirt locations. If we match an A with a B (at distance dd), this costs dZdZ and corresponds to moving a piece of dirt. Unmatched As correspond to deletions (cost YY) and unmatched Bs correspond to insertions (cost XX).   
  


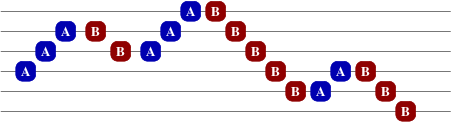
It will be helpful to assume that we match all of the smaller set of points; for example, if there are fewer As, we'd like to assume that we want to match all the As, leaving only Bs unmatched. To do this, we set the cost of matching a distance-dd pair to be min(dZ,X+Y)min(dZ,X+Y), reflecting now the possibility that the matched pair might not be an actual move, but may instead be a "teleport" caused by a delete plus an insert. We can now assume that all elements of the smaller set must be matched.

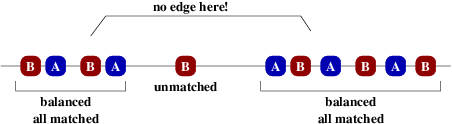
A key structural property is that there always exists an optimal matching with a "nesting" structure, where pairings can nest but not "cross". This follows from the fact that any set of crossing edges can be "uncrossed", for example as shown below, without harming (increasing) the total cost of the matching. By doing this repeatedly, we can turn any optimal matching into an optimal matching with no crossings.   
  


Recall that the cost of a distance-dd edge in the matching is min(dZ,X+Y)min(dZ,X+Y). If the cost is X+YX+Y, we'll call the edge "long"; otherwise it is "short". We can prove that cost doesn't increase by the uncrossing operation by breaking it into three cases: (i) both p and q are short, (ii) one of p and q is long, (iii) both p and q are long. In all cases, you can easily show that the total cost does not increase. For example, in case (iii), edge s must also be long, so the contribution before uncrossing is 2(X+Y)2(X+Y) and the contribution after is X+Y+min(dr,X+Y)≤2(X+Y)X+Y+min(dr,X+Y)≤2(X+Y).

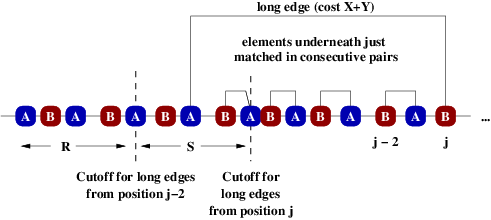
Non-crossing implies also that underneath each edge must be the same number of As and Bs. Assume for a moment (without loss of generality) that there are more Bs than As, so all As must be matched. In this case, if there are more As than Bs underneath edge p as shown below, then one of the As would need to be matched in a way that crosses p.   
  


Similarly, if there are more Bs than As underneath edge q, then one of them must remain unmatched (since otherwise there would be a crossing, just as before), and then we could decrease the length of edge q by linking with the unmatched B inside the range, giving an even more optimal solution (or at least, not a worse solution).

Let us now assign heights to our points, with successive As moving upward and Bs moving downward:   
  


The fact that the points underneath each edge must be balanced therefore implies that we can decompose our problem and consider each height level independently. This simplifies things quite a bit, since within any given height level we have alternating As and Bs. Either we have the same number of As and Bs and they are all matched, or we have one excess element that is unmatched while the rest are matched. In this second case, one can show that an optimal solution will not involve an edge across the unmatched element, so it further decomposes into a prefix of balanced As and Bs (all matched to each-other), followed by the unmatched element, followed by a suffix of balanced As and Bs, all matched:   
  


We therefore have reduced our problem to: given an alternating set of points, find the optimal way to match every balanced prefix (and by symmetry, suffix). We show how to do this in linear time with dynamic programming, after which another linear-time scan can be used to find the right way to combine the two solutions as in the "one element unmatched" case above.

Our dynamic programming formulation looks like the following. For each balanced prefix (say, up to position jj), we compute the optimal way to match all its elements, and also the optimal way to match its elements such that jj is matched with a long edge. For matching element jj with a long edge, candidates to which we can match are those elements in regions RR and SS in the figure below:  
  


We can afford to check all the candidates in region SS explicitly, since for all values of jj this will involve monotonically scanning the entire line just once. Each candidate ii in SS can be checked in O(1)O(1) time, since we have already computed the best way to match everyone up to i−1i−1, and since the elements underneath the long ii-jj edge can be assumed to be simply paired up consecutively (so this cost can be computed in constant time using the difference of two prefix sums). This follows from the observation that an optimal solution will never involve a long edge nested within another long edge, so underneath any long edge we have only short edges; furthermore, nested short edges can always be uncrossed with no change in objective value, so we can assume the short edges underneath a long edge are just consecutively paired, with no nesting at all. To evaluate candidates in the RR region, we note that the best such candidate is the same as the best long-edge match from element j−2j−2, the only difference being that its cost has increased by the additional cost of pairing elements j−2j−2 and j−1j−1.

For matching element jj with a short edge, we can assume (by the reasoning above, since short edges don't nest), that jj would be paired with j−1j−1, so this is easy to evaluate in O(1)O(1) time. This completes the description of the DP algorithm.

My code is below. The input part in the main function arranges elements by level, then the solve() function runs independently within each level, calling the DP code once or twice (twice if we need to combine a prefix and a suffix solution, having to deal with an odd number of elements on some level). The DP code is a bit cryptic, but follows essentially the approach outlined above.

#include <iostream>

#include <fstream>

#include <vector>

#include <algorithm>

using namespace std;

typedef long long LL;

#define MAX\_N 100000

#define MAX\_VAL 10

#define MAX\_TOT (MAX\_N\*MAX\_VAL)

int N, X, Y, Z, rtype[MAX\_TOT\*2+1];

LL res1[MAX\_TOT\*2+1], res2[MAX\_TOT\*2+1];

vector<int> rows[MAX\_TOT\*2+1];

void dp(vector<int> &v, LL \*results)

{

LL M=v.size(), i=-2, prefcost\_i=0, prefcost\_j=0, prev\_longrange=999999999999999LL;

for (int j=1; j<M; j+=2) {

if (j>1) prefcost\_j += Z\*abs(v[j-2]-v[j-1]);

if (j>1) prev\_longrange += Z\*abs(v[j-2]-v[j-1]);

while (i+2 < j && X+Y <= Z\*abs(v[j]-v[i+2])) {

i += 2;

if (i>0) prefcost\_i += Z\*abs(v[i]-v[i-1]);

prev\_longrange = min(prev\_longrange, X+Y+prefcost\_j-prefcost\_i+(i>0?results[i-1]:0));

}

results[j] = min(prev\_longrange, Z\*abs(v[j]-v[j-1]) + (j>1?results[j-2]:0));

}

}

LL solve(vector<int> &v, int ecost)

{

LL M = v.size(), best;

if (M == 0) return 0;

if (M == 1) return ecost; // cost of insert/delete for just single item

dp(v, res1);

reverse(v.begin(), v.end());

dp(v, res2);

reverse(res2, res2+M);

if (M%2 == 0) best = res1[M-1]; // even: all matched; only 1 DP pass needed

else { // odd: one left out -- piece together prefix + missing item + suffix

best = ecost + min(res1[M-2], res2[1]);

for (int i=2; i<=M-3; i+=2) best = min(best, res1[i-1] + ecost + res2[i+1]);

}

return best;

}

int main(void)

{

ifstream fin ("landscape.in");

fin >> N >> X >> Y >> Z;

for (int last\_d=0, level=MAX\_TOT, i=0; i<N; i++) {

int a, b, m, d;

fin >> a >> b;

d = (max(a,b)==a) ? +1 : -1;

m = max(a,b) - min(a,b);

while (m-- > 0) {

if (last\_d == d) level += d;

if (rtype[level]==0) rtype[level] = d;

rows[level].push\_back(i);

last\_d = d;

}

}

fin.close();

ofstream fout ("landscape.out");

LL total = 0;

for (int level=0; level<MAX\_TOT\*2+1; level++)

total += solve(rows[level], rtype[level]>0 ? Y : X);

fout << total << "\n";

fout.close();

return 0;

}