WOBURN CHALLENGE

2015-16 Online Round 2

Solutions

Automated grading is available for these problems at:
wcipeg.com

For problems to this contest and past contests, visit:
woburnchallenge.com

Problem IV: A New Hope

The first junior problem is always meant to test the bare basics of interpretation and implementation – i.e. can you understand what the problem is asking and use your language on an rudimentary level to do basic input/output accordingly? In this case, we just have to input a number *N* and output the sentence with *N* number of "`far`"s. The hardest part is following the output format *exactly*. Two ways to approach this are as follows.

The first way follows simple intuition – output the first part of the phrase, loop *N* times to output "`far`", and finally output the last part of the phrase. However, this raises a common problem with loops – we actually want the first and last iterations of the loop to do different things (the first "`far`" should be followed by a comma, but last "`far`" should not). The typical ways to handle this are: doing a check within the loop for the border case, or just doing the border case outside of the loop and only loop *N* – 1 times. The last option is usually easier because it removes the need for an extra if-statement. This method is implemented below in the official solution.

Official Solution (C++)

```
#include <iostream>
using namespace std;

int N;

int main() {
  cin >> N;
  cout << "A long time ago in a galaxy ";
  for (int i = 0; i < N - 1; i++) {
    cout << "far, ";
  }
  cout << "far away..." << endl;
  return 0;
}
```

Another possible solution comes from simply observing that the number of "`far`"s can only be as big as 5, which tells us that it's quite feasible to "hardcode" the answer. In other words, we can use 5 if/else-statements to check what *N* is equal to, and then go on to copy and paste the line of code which outputs the phrase for each of the 5 conditions, except with the number of "`far`"s manually modified in each case.

```
if (N == 1)
  cout << "A long time ago in a galaxy far away..." << endl;
else if (N == 2)
  cout << "A long time ago in a galaxy far, far away..." << endl;
else if (N == 3)
  cout << "A long time ago in a galaxy far, far, far away..." << endl;
...
```

This is also a perfectly fair solution, because some of you may be able to hardcode faster than obtaining a generic version using loops (which works for any sized *N*, and not just for *N* not exceeding 5). Although contests encourage taking advantage of small input sizes to speed up coding, just know that you should eventually be acquainted with writing generic solutions for harder problems that must scale.

Problem V: The Empire Strikes Back

Much like the first problem, this one tests basic skills in interpretation and implementation. The difference here is that neither looping nor if-statements can be avoided. Given two integers *N* and *M*, the question asks us to take *N* integer masses and add up all the ones which are less than or equal to *M*. If you solved the last problem, chances are you'll have no trouble with looping/if-statements and input. This is as easy as looping *N* times to read in the masses one by one, and having one if-statement within the loop to check if the number that was just read in is at most *M*.

Note that we don't have to overcomplicate the solution by storing the numbers into arrays. We can simply process the input one by one as we go, since *M* is given beforehand and a number won't be needed again after we're done with it for the first time. The only part that you might trip up on is interpreting the phrase "Luke's control of the Force is only strong enough to allow him to lift a rock if its mass is **no larger** than *M*." This means that we should add the number if it's *less than or equal to* (<=) *M*, and not if it's strictly *less than* (<) *M*. Otherwise, the relatively straightforward solution is given below.

Official Solution (C++)

```
#include <iostream>
using namespace std;

int N, M, W, ans = 0;

int main() {
  cin >> N >> M;
  for (int i = 0; i < N; i++) {
    cin >> W;
    if (W <= M) {
      ans += W;
    }
  }
  cout << ans << endl;
  return 0;
}
```

Problem VI: Revenge of the Sith

This problem required arrays and some slightly more complex control structures, but is ultimately nothing too daunting. We are given a distance *R*, *S* stormtroopers each associated with a type and position, as well as *E* Ewoks each associated with a position. First we must first count, for each Ewok, not how many stormtroopers are near him or her, rather how many *types* of stormtroopers are near him or her. Then for each Ewok, if the number of types of stormtroopers is greater than a certain number, then the Ewok is considered in danger and we increment the answer by 1. But what is this number? Once again, your interpretation skills are needed for the phrase "any **more than a single type** of stormtrooper poses a risk to them," which means greater than one (> 1).

Following the KISS principle, the first solution you should consider is to simply do exactly that – for each Ewok, loop and count the number of types of stormtroopers that are in range. Since there are only up to 100 Ewoks and 100 stormtroopers, the total time complexity will be proportional to $O(S \times E)$, which is at most 10000 steps and will certainly pass under the time limit of 2.00 seconds (30 million "steps" per 1 second of time limit is a good rule of

thumb). Although this thought might have not even crossed your mind, you should start looking into the running time analysis of algorithms, which will certainly be important for more advanced problems.

In any case, how should we check if an Ewok is within range of stormtrooper? The formula for Euclidean distance is already given in the question. We just have to see if the value of $\sqrt{(x^2 + y^2)}$ is *less than or equal to* (once again, not *less than*) the value of R, where x is the difference in x-coordinates and y is the difference in y-coordinates. Now, an important thing to point out is that, while we could use the square root function and replicate the distance formula exactly and compare it with R afterwards, this is not a good practice. The square root function always returns a real number (i.e. a floating point value). Floating-point arithmetic is inevitably inaccurate, and directly comparing them with the relational operators will often break down. In the general case, epsilon comparisons provide a fair solution, but there is a popular trick for the distance formula – since both sides of the comparison are guaranteed positive values, we can just get the same result as "$\sqrt{(x^2 + y^2)} \leq R$" by comparing their squares. In other words, we can avoid floating-points altogether with expression "$x^2 + y^2 \leq R^2$". Though the bounds in this problem may not have been able to expose this bug in many programs, you should start adapting this practice as you improve.

Finally, to keep track of the types, we make a 0-initialized array for each Ewok we input, storing the counts of each type of stormtrooper. This array won't be a burden to the efficiency of the program since there are only up to 4 types, which can be effectively considered a constant. For each of the stormtroopers we loop through and find to be near the current Ewok, we set the count of that type of stormtrooper to 1 (read: *not* increment the count, just set it to 1). Finally, we add up all of the 4 counts to see if the total number is greater than 1. If so, then we increment the answer.

Official Solution (C++)

```
#include <iostream>
using namespace std;

int S, E, R, x, y, ans = 0;
int W[105], X[105], Y[105];

int main() {
  cin >> S >> E >> R;
  for (int i = 0; i < S; i++) {
    cin >> W[i] >> X[i] >> Y[i];
  }
  for (int e = 0; e < E; e++) {
    cin >> x >> y;
    int cnt[5] = {0};
    for (int i = 0; i < S; i++) {
      if ((X[i] - x) * (X[i] - x) + (Y[i] - y) * (Y[i] - y) <= R * R) {
        cnt[W[i]] = 1;
      }
    }
    if (cnt[1] + cnt[2] + cnt[3] + cnt[4] > 1) {
      ans++;
    }
  }
  cout << ans << endl;
  return 0;
}
```

Problem VII: The Force Awakens

This is the last and most interesting problem of the junior contest, since it's the only problem which actually forces us to think about cutting down on the time complexity of our program. We are given a grid of empty cells, mirrors, and blocked cells. We must count the number of empty cells such that if we were to shoot a laser beam into at least one direction from that cell, the beam would eventually bounce back to the same cell.

Let's first consider some simple, yet inefficient solutions. The most straightforward solution is simulation. In other words, loop through all of the $N \times M$ cells in the grid. For each cell, try to fire a shot into each of the four cardinal directions – left, right, up, and down, one after the other. For each shot fired, we will simply follow its path through the grid until either it reaches a wall/non-reflective barrier (in which case the original cell isn't deadly) or it returns to the starting position (in which case the cell is deadly). In each state of this simulation, we will need to keep track of the following pieces of information:

- the row that the laser beam is current at,
- the column that the laser beam is current at, and
- the direction that the laser beam is current traveling in.

The row and column can simply be stored as a pair of integers (r, c). The direction is the trickier part. Some may decide to store it as a single number/letters (i.e. arbitrarily assign directions to the numbers from 0 to 3, or perhaps the letters 'L', 'R', 'U' and 'D') and check these values each time to make changes to the current position. However, a popular, more elegant method is just to keep a pair (dr, dc) – the *change* in the row and column, respectively, per unit of time. Left is (0, –1), right is (0, 1), up is (–1, 0), and down is (1, 0). This is much cleaner, because all we have to do is add the values of dr and dc to the current coordinates to get the new coordinates.

Every time we encounter a mirror, we must change dr and dc accordingly. As it turns out, there exists an elegant way to do so which avoids painstakingly long if-statement chains. Observe that whenever the laser hits a `/` or `\` mirror, the axis of the laser changes – i.e. horizontal beams become vertical, and vice versa. That means, we can simply swap whichever of dr and dc is currently equal to 0 when encountering these mirrors. However, by simply swapping the two numbers, the sign of the non-zero directional component may be wrong. When is it wrong? A quick check in our head shows that only for the `/` mirror, we have to negate the non-zero value – rightward (0, 1) beams will be reflected upward (–1, 0), downward (1, 0) beams will be reflected leftward (0, –1), etc. Which of dr or dc to negate? Since negating 0 does nothing to it and exactly one of them is always going to be zero, we might as well skip the check and negate both. Finally, observe that the `X` mirror will just reverse both components, so it's the exact same case as the check for the `/` mirror and we can combine them into one. Altogether, the handling of changing directions can be done in two harmless-looking if-conditions:

```
if (g[r][c] == '/' || g[r][c] == '\\')
  swap(dr, dc);
if (g[r][c] == '/' || g[r][c] == 'X') {
  dr = -dr;
  dc = -dc;
}
```

Note that in many programming languages the backslash character '`\`' is an escape character. To represent a single backslash, we'll need to escape it using yet another backslash. Finally, we may want to simplify checking of walls and barriers. If we completely pad our 2D character array with blocks ('`#`' characters) before reading in the grid, and also read the grid into 1-based indices (from (1, 1) to (N, M) rather than from (0, 0) to ($N-1$, $M-1$)) we've effectively created a literal "wall" of blocks around the grid for the laser beams to run into. This is not necessary, but removes the need for range checks. With these implementation details out of the way, the rest of the simulation should be very straightforward.

Solution – Straightforward Simulation, 80% of Points (C++)

```
#include <algorithm>
#include <cstring>
#include <iostream>
using namespace std;

const int DR[] = {0, 0, -1, 1};
const int DC[] = {-1, 1, 0, 0};

int N, M, ans = 0;
char g[2005][2005];

bool simulate(int sr, int sc, int dr, int dc) {
  int r = sr, c = sc;
  while (g[r += dr][c += dc] != '#') {
    if (r == sr && c == sc)
      return true;
    if (g[r][c] == '/' || g[r][c] == '\\')
      swap(dr, dc);
    if (g[r][c] == '/' || g[r][c] == 'X') {
      dr = -dr;
      dc = -dc;
    }
  }
  return false;
}

int main() {
  memset(g, '#', sizeof g);
  cin >> N >> M;
  for (int r = 1; r <= N; r++)
    for (int c = 1; c <= M; c++)
      cin >> g[r][c];
  for (int r = 1; r <= N; r++)
    for (int c = 1; c <= M; c++)
      if (g[r][c] == '.')
        for (int d = 0; d < 4; d++)
          if (simulate(r, c, DR[d], DC[d])) {
            ans++;
            break;
          }
  cout << ans << endl;
  return 0;
}
```

```
/\/\/\X
.......
.......
.\/\/\/
```

Why is this solution not enough? Let's study its running time. If we have to simulate a laser beam from each of the $N \times M$ cells, then the running time should be proportional to $4NM$ multiplied by the largest possible number of steps taken by a laser beam. A little thinking, we should be able to come up with a test case like the one depicted on the right. If we project the laser beam upwards from the bottom-left corner, it will travel to every other cell twice before returning to the original cell. This tells us that *each simulation* of a single laser also has the worst-case running time proportional to $N \times M$, the size of the grid. Putting it together, our straight-simulation solution takes $4N^2M^2$ steps. In the worst case when the grid is 2000 by 2000, our algorithm will have to take $4 \times 2000^2 \times 2000^2 = 64{,}000{,}000{,}000{,}000$ (64 quadrillion) steps. This is clearly going to exceed the time limit by our 30 million steps/second rule, but a correct implementation like the one above is given 80% of the points as guaranteed by the problem statement ($4 \times 50^2 \times 50^2 = 25$ million).

The next step that one might consider is some optimization. As it turns out, there are a few ways we can go about optimizing the straight-simulation.

One way is by noticing that in the worst-case example we provided above, there exists many empty cells we pass through. It's not hard to see that we would be able to make our program much faster by simply "skipping-over" those cells. In other words, we would like to make each "reflection" have $O(1)$ time complexity by moving directly to the opposing mirror, rather than $O(\max(N, M))$ through a bunch of empty cells. This is doable with precomputation.

We can store a 3D array $next[r][c][d]$ of coordinates. This array shall store the coordinates of the next mirror that a beam at (r, c) should hit if it was travelling in direction d. To precompute *next* for empty cells where $d = (0, -1)$ or $(-1, 0)$, we can loop upwards from (1, 1) to (N, M) and take the *next* value of the cells left and above of it not on the left or top borders. For cells on the left or top border, *next* can simply point to the wall block. For non-empty cells (cells will a mirror or block), *next* will point to its own co-ordinates so that other *next* values to the right and top of it can propagate through. Then to get *next* values for $d = (0, 1)$ or $(1, 0)$, we do the same thing but in reverse – i.e. loop downwards from (N, M) to (1, 1). During our simulation, we simply set our current coordinates to the *next* coordinates after each step to "skip" to the next non-empty cell, making sure to check for blocks both before and after the skip. For further clarification, see the implementation below for details.

While this is a good optimization, the solution is still not good enough. Consider the case on the right. In a test case like this, a laser traveling leftwards from the 4th column of the bottom-most row will hit all of the mirrors before returning to the same cell. In fact, a single laser beam shot anywhere on the cycle will always encounter $2N$ mirrors. For such a case, we've shown the time complexity to be *at least* $O(N^3)$, which requires roughly $2000^3 = 8$ billion steps. This is still too slow. However, implemented efficiently, it is awarded an extra 10% of points.

```
/......\
./....\.
../..\..
.../\...
.../\./..
..\.../.
.\...../
\.../...
```

Solution – Optimized Simulation, 90% of Points (C++)

```
#include <algorithm>
#include <cstring>
#include <iostream>
#include <utility>
using namespace std;

const int DR[4] = {0, 0, -1, 1};
const int DC[4] = {-1, 1, 0, 0};

inline int dir(int dr, int dc) {
  if (dr ==  0 && dc == -1) return 0;
  if (dr ==  0 && dc ==  1) return 1;
  if (dr == -1 && dc ==  0) return 2;
  return 3;
}

int N, M, r2, c2, dr, dc, ans = 0;
char g[2002][2002];
pair<int, int> p, next[2002][2002][4];

bool intersect(int R1, int C1, int R2, int C2, int r, int c) {
  if (R1 == R2) {
    if (r != R1) return false;
    if (C1 > C2) swap(C1, C2);
    return C1 <= c && c <= C2;
  }
  if (c != C1) return false;
  if (R1 > R2) swap(R1, R2);
  return R1 <= r && r <= R2;
}
```

```
bool simulate(int sr, int sc, int dr, int dc) {
  int r = sr, c = sc;
  while (g[r][c] != '#') {
    if (g[r][c] == '/' || g[r][c] == '\\')
      swap(dr, dc);
    if (g[r][c] == '/' || g[r][c] == 'X') {
      dr = -dr;
      dc = -dc;
    }
    if (g[r += dr][c += dc] == '#') break;
    p = next[r][c][dir(dr, dc)];
    if (intersect(r, c, p.first, p.second, sr, sc))
      return true;
    r = p.first;
    c = p.second;
  }
  return false;
}

int main() {
  memset(g, '#', sizeof g);
  cin >> N >> M;
  for (int r = 1; r <= N; r++)
    for (int c = 1; c <= M; c++)
      cin >> g[r][c];
  for (int r = 1; r <= N; r++)
    for (int c = 1; c <= M; c++)
      for (int d = 0; d < 4; d++)
        if (DR[d] < 0 || DC[d] < 0) {
          if (g[r][c] != '.') {
            p = make_pair(r, c);
          } else {
            if ((r2 = r + DR[d]) < 1 || (c2 = c + DC[d]) < 1)
              p = make_pair(r2, c2);
            else
              p = next[r2][c2][dir(DR[d], DC[d])];
          }
          next[r][c][dir(DR[d], DC[d])] = p;
        }
  for (int r = N; r >= 1; r--)
    for (int c = M; c >= 1; c--)
      for (int d = 0; d < 4; d++)
        if (DR[d] > 0 || DC[d] > 0) {
          if (g[r][c] != '.') {
            p = make_pair(r, c);
          } else {
            if ((r2 = r + DR[d]) > N || (c2 = c + DC[d]) > M)
              p = make_pair(r2, c2);
            else
              p = next[r2][c2][dir(DR[d], DC[d])];
          }
          next[r][c][dir(DR[d], DC[d])] = p;
        }
  for (int r = 1; r <= N; r++)
    for (int c = 1; c <= M; c++)
      if (g[r][c] == '.')
        for (int d = 0; d < 4; d++)
          if (simulate(r, c, DR[d], DC[d])) {
            ans++;
            break;
          }
  cout << ans << endl;
  return 0;
}
```

There are other optimizations we can do, such as noticing whenever a cycle has been reached, all cells in the cycle (going in the correct direction) must also be deadly. However, as long as we refuse to abandon the idea of projecting lasers outwards of each empty square, it all seems futile. Unless we can optimize the time of each simulation to a whole *order* of complexity less than $O(N)$ (like $O(1)$ or $O(\log N)$), this problem just seems impossible. As it turns out, we must altogether abandon the idea of simulating from all the empty squares.

So here's a crazy idea. How about we simulate from the *blocked* squares instead?

Turns out, it's not so crazy at all. Consider this "obvious" observation – every possible laser beam in the grid (at some coordinate of an empty cell, travelling in some direction) has exactly two possible fates:

1. It will end up back in the same coordinates, travelling in the same direction. We've already established that this means every cell in the cycle is considered deadly.

2. It will hit a wall or blocked square.

We know that any laser beam's path is "deterministic," in the sense that if we were to shoot a laser-beam from a given cell which eventually ends up hitting a wall/block in a particular direction, then shooting the beam in *reverse* from the coordinates of that wall/block will eventually pass through the original cell.

Much more importantly, it is guaranteed that a laser beam coming out of a wall/block will never be able to enter into an infinite cycle. In other words, the laser beam will *always* end up at yet another wall/block. Convince yourself that this is true by trying to come up with counterexamples (you won't be able to). This presents the follow approach:

First consider each neighbouring cell of a wall/block which is not blocked (either an empty cell or a mirror). We will simulate a laser coming out of that wall/block in the direction of the non-blocked cell. For *each* of the simulations, we keep track of which cells in the grid we've encountered *in that particular simulation only*. If we notice that a cell has already been visited in the same simulation, then it must be deadly.

How? We could keep a 2D Boolean array of all the cells visited in the current simulation, but resetting it each time is too slow and will still make the running time quadratic. Instead, we can number each simulation (just count upwards from 1) and keep an array called $last[r][c]$, which stores the number of the last simulation where we encountered cell (r, c). If *last* is equal to the current simulation number, then we must have encountered it twice and the cell is deadly. Otherwise, just make sure to update $last[r][c]$ to the current simulation number.

However, we may still not be counting the deadly cells which are infinitely involved in a cycle (and never hits a wall). Let's consider each of the 4 directions of each empty cell. If after the previous simulations, a wall laser beam has never traveled in that direction at that cell, then that cell *must* be part of a cyclic path (we've shown that there are only these 2 cases).

So, we can keep yet another Boolean array $done[r][c][d]$, storing whether a beam has previously moved through that cell, in that direction. For each cell (r, c) that is empty, if even one of the 4 directions has not been previously visited by a wall/block-originated laser beam, then it must be deadly.

Note that since wall/block laser beams are deterministic, it is unnecessary to follow a beam going in the same direction at a given cell more than once. Therefore, we notice that if $done[r][c][d]$ is already true while we're following a laser beam, we might as well stop the simulation right there. We never reset *done* across any of the simulations, so *done* guarantees that every possible laser beam is examined at most once in the entire program. Magically, this ensures that our entire solution only takes at most $4NM$ steps, yielding a $O(NM)$ running time.

The 100% solution is presented as follows.

Official Solution (C++)

```
#include <algorithm>
#include <cstring>
#include <iostream>
using namespace std;

const int DR[] = {0, 0, -1, 1};
const int DC[] = {-1, 1, 0, 0};

inline int dir(int dr, int dc) {
  if (dr ==  0 && dc == -1) return 0;
  if (dr ==  0 && dc ==  1) return 1;
  if (dr == -1 && dc ==  0) return 2;
  return 3;
}

int N, M, ans = 0, last[2005][2005] = {0};
char g[2005][2005];
bool dead[2005][2005] = {0}, done[2005][2005][4] = {0};

void simulate(int r, int c, int dr, int dc, int curr) {
  while (g[r += dr][c += dc] != '#') {
    if (g[r][c] == '.') {
      if (last[r][c] == curr) dead[r][c] = true;
      last[r][c] = curr;
    }
    int d = dir(dr, dc);
    if (done[r][c][d]) return;
    done[r][c][d] = true;
    if (g[r][c] == '/' || g[r][c] == '\\')
      swap(dr, dc);
    if (g[r][c] == '/' || g[r][c] == 'X') {
      dr = -dr;
      dc = -dc;
    }
  }
}

int main() {
  memset(g, '#', sizeof g);
  cin >> N >> M;
  for (int r = 1; r <= N; r++)
    for (int c = 1; c <= M; c++)
      cin >> g[r][c];
  int curr = 1;
  for (int r = 0; r <= N + 1; r++)
    for (int c = 0; c <= M + 1; c++)
      if (g[r][c] == '#')
        for (int d = 0; d < 4; d++) {
          int r2 = r + DR[d], c2 = c + DC[d];
          if (r2 >= 0 && c2 >= 0 && g[r2][c2] != '#')
            simulate(r, c, DR[d], DC[d], curr++);
        }
  int ans = 0;
  for (int r = 1; r <= N; r++)
    for (int c = 1; c <= M; c++)
      if (g[r][c] == '.' && (dead[r][c] ||
            done[r][c][0] + done[r][c][1] +
            done[r][c][2] + done[r][c][3] < 4))
        ans++;
  cout << ans << endl;
  return 0;
}
```

Problem I: The Phantom Menace

Given an array S of N ordered integers each between 0 and 4, we must replace the 0's with other numbers between 1 and 4 such that no two adjacent numbers are the same, and that the resulting array is lexicographically smallest. You may be tempted to try some recursive solution, but this is unnecessarily complicated and may very well exceed the time limit. Instead, we can take advantage of the second requirement to construct a greedy solution.

Note that for any given i between 1 and N such that $S_i = 0$, it is always more important to minimize the value at i than any index greater than i. This is because the goal is to ultimately find a solution which minimizes the lexicographical rank of the entire array, and earlier indices take precedence. Now note that it will always be possible to fill in any 0-value in the array, no matter what its neighbouring values are (at most 2 of the 4 possible values will be prohibited). This makes it so we don't have to worry about "consequences" of our decisions – namely, whether the current choice will eventually lead to no possible way to fill in a cell later on. Putting these two observations together, we know that as long as we minimize the current value, there will always be an answer which is lexicographically smallest.

Now, we can loop through each unassigned scene in order from 1 to N and try to assign it the smallest value in the range 1..4 which is unequal to that of the previous scene (if any) and also unequal to that of the following scene (if it exists and is pre-assigned to a setting). We must be careful to not access the array out of bounds at the border iterations. This greedy process (implemented below) takes $4N$ steps, which means that it runs in $O(N)$ time.

Official Solution (C++)

```
#include <iostream>
using namespace std;

int N, S[10005];

int main() {
  cin >> N;
  for (int i = 0; i < N; i++) {
    cin >> S[i];
  }
  for (int i = 0; i < N; i++) {
    if (S[i] == 0) {
      for (int j = 1; j <= 4; j++) {
        if ((i == 0 || S[i - 1] != j) && (i == N - 1 || S[i + 1] != j)) {
          S[i] = j;
          break;
        }
      }
    }
  }
  for (int i = 0; i < N; i++) {
    cout << S[i];
  }
  cout << endl;
  return 0;
}
```

Problem II: Attack of the Clones

This problem proved to be surprisingly difficult, since it had fewer full-solutions in the live contest than the third senior problem. Given N points on a grid where odd-numbered streets can only move right/down and even-numbered streets can only move left/down, we would like to calculate the minimum distance to visit the first x points starting from (1, 1) for all x between 1 and N.

A naïve solution would be to position each coordinate in a 2D array as they're being inputted, and then for each x, looping through each row and column of the grid to count the steps between each location. For odd-numbered rows, find the east-most avenue which has a location either in the current street, or the street below it (so we don't miss that location when we go south) and go south from there. For even-numbered rows, it's the same except we search for the west-most avenue. Since each time requires looping through the max size of the grid and we do this N times, the time complexity will be $O(N \times \max(S_i) \times \max(A_i))$. In the subtask where N, S_i, and A_i are all at-most 100, this will take roughly $100^3 = 1$ million steps and pass in time. This solution is given 30% of points.

Let's consider another naïve solution based on an important observation. If we want to minimize the total distance traveled, any given set of x points can only be visited in one possible order following the zig-zagging streets and avenues downwards and sideways. How do we determine this ordering? We know that north-most points are always visited before south-most points, meanwhile for points on the same row, west-most points are always visited before east-most points if and only if the row number is odd. So, if we were to sort the points using a comparison-based sort, we could use the following comparison function to determine whether the point is less.

```
bool is_less(pair<int, int> a, pair<int, int> b) {
  if (a.first != b.first)
    return a.first < b.first;
  if (a.first % 2 == 1)
    return a.second < b.second;
  return a.second > b.second;
}
```

In C++, the `std::sort()` function in `<algorithm>` allows us to specify a custom-defined comparator. Many other languages also support special comparators for sorting functions. To sort an `std::vector` of N points (stored as `std::pairs`), we would simply have to call `sort(points.begin(), points.end(), is_less)`.

After the points are sorted, we'll need to find the Manhattan distance between each pair of adjacent points and add them all up. The Manhattan distance between two points (s_1, a_1) and (s_2, a_2) is $|s_1 - s_2| + |a_1 - a_2|$, and obviously represents the minimum distance between any two adjacent points in this grid. This works because, evidently, the overall distance will be minimal if the distance between each adjacent pair of locations in the order are minimized.

We also propose a trick to eliminate the need for comparators altogether. Simply negate the value of a whenever the value of s is even, and the implicit comparator for pairs will automatically take care of it during sorting (comparing by s if they are different, breaking ties by comparing a). This will work in languages like Python, where tuples are compared lexicographically by default. The only time we'll be using a values again is when we're calculating the Manhattan distance. There, we can just use the formula $|s_1 - s_2| + ||a_1| - |a_2||$, since $|x - y| = ||x| - |y||$.

Efficient sorting functions have a worst-case time complexity of $O(N \log N)$, and finding the Manhattan distance requires looping through all of the N points, and will require $O(N)$ time. Therefore, finding the total distance for just one set of N points takes $O(N \log N + N)$, dominated by $O(N \log N)$ time. The problem is that we have to repeat this for each x in the range $1..N$, re-sorting the array each time. So, the running time is effectively $O(N^2 \log N)$. Since $N \leq 200{,}000$, this is far too slow. However, for $N \leq 2000$, there will be roughly $2000^2 \log_2 \approx 43$ million steps and should pass in time. Such a solution is given 75% of points, and is implemented below.

Solution – Sorting, 75% of Points (C++)

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

typedef pair<int, int> pii;

inline int dist(const pii & a, const pii & b) {
  return abs(a.first - b.first) + abs(abs(a.second) - abs(b.second));
}

int N, S, A, ans;
vector<pii> points;

int main() {
  points.push_back(make_pair(1, 1));
  cin >> N;
  for (int i = 0; i < N; i++) {
    cin >> S >> A;
    if (S % 2 == 0) A = -A;
    points.push_back(make_pair(S, A));
    sort(points.begin(), points.end());
    ans = 0;
    for (int j = 1; j < (int)points.size(); j++)
      ans += dist(points[j - 1], points[j]);
    cout << ans << '\n';
  }
  return 0;
}
```

Now, we can consider some optimizations to this solution. The bottle-neck right now is the N log N sorting function. We can actually reduce sorting to just $O(N)$ time by using sorting algorithms such as insertion sort that work really well on nearly-sorted lists. Alternatively, can store the points in a linked-list where we can iterate through to insert each new point (up to N iterations to find the position we want, and $O(1)$ time to insert into the linked list) also in $O(N)$ time. Ultimately though, both of these methods still require $O(N^2)$ total running time, and for $N \leq 200{,}000$, this is just not feasible.

How do we insert faster than $O(N)$ into an ordered set of values? Why, using a balanced binary search tree of course! Many programming languages has this data structure built-in – C++ has `std::set`, and Java has `java.util.TreeSet`. We can insert any element into it worst-case $O(\log N)$ time, where N is the number of element currently in the set. The code itself won't get much more complex from this change. Now, the bottleneck is recomputing all of the Manhattan distances, which would still take $O(N)$ and make the entire algorithm $O(N^2)$.

Notice that whenever a single point is inserted, most of the adjacent Manhattan distances won't change. Namely, only the ongoing sum should be maintained, and when a new point p is inserted between two points a and b in the list, we can just subtract the Manhattan distance from a to b, and add the distances from a to p and from p to b to our running total. Since for each of the N points, addition and subtraction takes $O(1)$ while insertion takes $O(\log N)$, this approach has a total time complexity of $O(N \log N)$.

Note that we'll need to obtain iterators to before and after the current element that was just inserted, and this requires some variety of a binary search operation on the tree that runs in $O(\log N)$. In C++, this can be done with `set.insert()`, `set.lower_bound()`, or `set.upper_bound()`. In Java, there are the functions `TreeSet.lower()` and `TreeSet.higher()` which can be used to get the elements just before and just after.

Official Solution (C++)

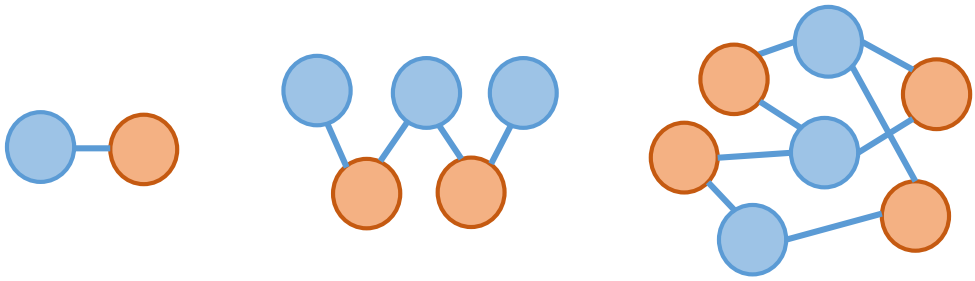
```
#include <algorithm>
#include <iostream>
#include <set>
#include <utility>
using namespace std;

typedef pair<int, int> pii;

inline int dist(const pii & a, const pii & b) {
  return abs(a.first - b.first) + abs(abs(a.second) - abs(b.second));
}

int N, S, A, ans = 0;
set<pii> points;
set<pii>::iterator it, prev, next;

int main() {
  points.insert(make_pair(1, 1));
  cin >> N;
  for (int i = 0; i < N; i++) {
    cin >> S >> A;
    if (S % 2 == 0) A = -A;
    pii p = make_pair(S, A);
    it = points.lower_bound(p);
    if (it == points.end()) {
      ans += dist(*--it, p);
    } else {
      next = it;
      prev = --it;
      ans += dist(*prev, p) + dist(p, *next) - dist(*prev, *next);
    }
    points.insert(p);
    cout << ans << '\n';
  }
  return 0;
}
```

Problem III: Revenge of the Sith

We are given an undirected graph with N nodes and M edges. If Anakin and Obi-Wan are placed at two random nodes on the graph (call them nodes x and y), we would like to know the probability that they will eventually reach the same node by each traveling one edge per second (if they can). First we should consider what determines whether or not a pair of starting locations x and y will yield an eventual encounter between them.

If x and y are not reachable from one another at all (in other words, if the two nodes are in different connected components of the graph), then clearly they won't ever encounter each other.

Otherwise if they are part of the same connected component, they mostly likely will encounter each other. However, it actually depends on the layout of the connected component. To figure out in which type of components they will never encounter each other, we can try drawing ourselves some examples. Here are some you might come up with. Say x starts on a blue node, and y starts on an orange node. Each person will never end up on a chamber of the opposite colour than the colour he started on. Therefore, they will never meet if they start on opposite colours.



Note that in all of these examples, there is never an edge connecting two nodes of the same colour. This is because if such an edge were to exist, then the "parity" of their opposing movements would be broken, and it would be possible to eventually maneuver them onto the same node. In fact, this type of graph is well-known as "bipartite." A component is bipartite if and only if there exists a way to label each of its nodes with one of two colours such that no edge connects two nodes with the same color.

If the component is bipartite and x and y both start on different colours, then this pair of starting locations will never result in a duel, since Obi-Wan and Anakin's nodes will be guaranteed to have different labels at every point in time (they'll simultaneously alternate between blue and orange every minute). If x and y would have the same label as one another or if their component is not bipartite, then an encounter can occur (which means that it certainly will within an infinite amount of time). Convince yourself that bipartite components are actually the *only* type of components in which a duel can never occur.

The probability of a random pair of starting positions (x, y) eventually leading to a duel is equal to the total number of starting pairs (x, y) that results in a duel, divided by the total number of possible starting pairs (equal to N^2, since x, y can each take on the nodes from 1 to N). We can do this by trying to classify each component of the graph as either bipartite or not. Breadth-first search (BFS) or depth-first search (DFS) are popular ways to do so.

Consider each connected component of the graph in turn – we will start BFS or DFSing from any node within the component, meaning trying to greedily label the component's nodes with alternating colours (say blue is 1 and orange is 2), until they've all been labelled or a conflict has been found (with a node adjacent to nodes with multiple different labels). In the latter case, the component is not bipartite, meaning that it contains c^2 valid pairs of starting locations (where c is the number of nodes in it). Otherwise, it contains $c_1^2 + c_2^2$ valid pairs (where c_1 and c_2 are the number of nodes labelled with 1 and 2, respectively). Traversing the graph to find the connected components and attempt to label their nodes can be done in $O(N + M)$ time, since a node or edge never needs to be examined more than once. The following program implements this solution exactly using an adjacency list, which is an efficient way to store and traverse graphs. More readings on bipartite graphs can be found online with a quick search.

Official Solution (C++)

```cpp
#include <iostream>
#include <vector>
using namespace std;

int N, M, a, b, c1, c2, ans = 0;
int col[46000] = {0};
vector<int> adj[46000];

bool bipartite;

void dfs(int n, int c) {
  col[n] = c;
  (c == 1 ? c1 : c2)++;
  for (int j = 0; j < adj[n].size(); j++) {
    if (col[adj[n][j]] == 0) {
      dfs(adj[n][j], c == 1 ? 2 : 1);
    } else if (col[adj[n][j]] == c) {
      bipartite = false;
    }
  }
}

int main() {
  cin >> N >> M;
  for (int i = 0; i < M; i++) {
    cin >> a >> b;
    adj[a].push back(b);
    adj[b].push_back(a);
  }
  for (int i = 1; i <= N; i++) {
    if (col[i] == 0) {
      c1 = 0;
      c2 = 0;
      bipartite = true;
      dfs(i, 1);
      if (bipartite) {
        ans += (c1 * c1) + (c2 * c2);
      } else {
        ans += (c1 + c2) * (c1 + c2);
      }
    }
  }
  cout.precision(6);
  cout << fixed << (double)ans / (N * N) << endl;
  return 0;
}
```

Problem VII: The Champ Awakens

This is a data structures problem. We are given a fixed rectangle on the plane representing John Cena's "blast zone," along with the straight-line paths of N ships and M different "steps". Each ship has an value of how much damage has been done to it. Steps with $A_i = 1$ can be considered "update" operations, while steps with $A_i = 2$ can be considered "query" operations. Each update adds C_i damage all of the ships in the blast zone rectangle after a certain time. Each query asks to print out the sum of the damages in a range of ships with consecutive indices.

A naïve solution would simply do exactly as the problem asks. We keep track of the current time. For each update operation, we move the time forward and loop through all of the ships. Given a particular time, we can easily tell where a ship will be by simply multiplying the time with the velocity (dx and dy) and adding the result to the starting coordinates respectively. If these new coordinates are within the blast zone, then we incur damage onto the ship. For query operations, we just loop between the two given indices, add up the damages, and report the total.

Updating and querying each has a worst-case running time of $O(N)$, making the overall running time of the program $O(NM)$. The following implementation is clearly too slow for $N, M \leq 400{,}000$ but is good enough to pass the $N \leq 500$ and $M \leq 2000$ subtask to obtain an easy 20% of the points.

Naïve Solution – 20% of Points (C++)

```
#include <iostream>
using namespace std;

const int MAXN = 400000;

int N, M, X1, Y1, X2, Y2, A, B, C, t = 0;
int X[MAXN], Y[MAXN], DX[MAXN], DY[MAXN];
long long x, y, damage[MAXN] = {0};

int main() {
  ios_base::sync_with_stdio(false);
  cin >> N >> M >> X1 >> Y1 >> X2 >> Y2;
  for (int i = 1; i <= N; i++) {
    cin >> X[i] >> Y[i] >> DX[i] >> DY[i];
  }
  for (int m = 0; m < M; m++) {
    cin >> A >> B >> C;
    if (A == 1) {
      t += B;
      for (int i = 1; i <= N; i++) {
        x = X[i] + (long long)t * DX[i];
        y = Y[i] + (long long)t * DY[i];
        if (x >= X1 && x <= X2 && y >= Y1 && y <= Y2)
          damage[i] += C;
      }
    } else {
      int ans = 0;
      for (int i = B; i <= C; i++)
        ans += damage[i];
      cout << ans << endl;
    }
  }
  return 0;
}
```

For a full solution, we have to try something significantly cleverer. One key observation is that each ship will either never be within the blast zone, or will be within the blast zone for a single consecutive window of time. This interval of time (if non-empty) can be found by calculating the intersection points of the edges of the blast zone rectangle with the line along which the ship will fly. Since we're dealing with a straight line being projected through a rectangle, there are at most two intersection points.

Finding the intersection points is merely drudging geometry, the details of which will be omitted from this analysis. Instead, we suggest trying to work it out yourself, referring to the solution program below if necessary. However, since each of the four edges is either horizontal or vertical, a general line intersection algorithm is not needed. Your math can thus be significantly simpler.

Now, if the line intersects with the rectangle at all, the time interval of interest can then be computed by considering the ship's velocity. Note that the interval may start or even end in the past, which is fine. There are at most $2N$ events in which ships either enter or leave the blast zone. All of these events should then be sorted by their time of occurrence, in $O(N \log N)$ time.

Now, we can simulate the M steps of the plan, somehow efficiently maintaining information about two things: which ships are currently inside the blast zone, and how much damage each ship has sustained so far. At each step, we should first update the current time, and iterate over all of the previously-computed "events" which have occurred since the previous step (or, for the first step, since the beginning of time), processing which ships are entering and leaving the blast zone, one by one. We must then be able to either update the damage taken by all ships currently within the blast zone, or query the sum of a range of such values. An advanced data structure is needed in order to handle these updates and queries in less than $O(N)$ time each!

A segment tree with lazy propagation will do the trick. Once again, the internet is filled with great articles that you can peruse. The segment tree data structure can be used to solve problems like the dynamic range sum query (efficiently reporting the sum of any consecutive segment in an array, as the array is being constantly changed). The lazy updating, or "lazy-propagation" technique adds the support to update an entire range of values efficiently.

In this particular problem, we will need to adapt it slightly. Each node in the tree will be responsible for a consecutive interval of ships like in normal segment trees, except here, it will store three values:

(A) the number of ships in this interval which are currently within the blast zone,
(B) the total amount of damage sustained so far by the ships in this interval, and
(C) a "lazy value" of how much damage still needs to be dealt to each ship in the interval which are currently within the blast zone.

The lazy propagation aspect works as follows: when necessary, the (C) value is passed down from a node to each of its children, while its (B) value should increase by the product of its (A) and (C) values (as each ship currently within the blast zone will lazily take the appropriate amount of damage).

With all of that in place, handling the entry/exit of a single ship to/from the blast zone with this segment tree is no problem – we'll need to add either +1 or –1 to the (A) value of the appropriate leaf node, as well as all of its ancestors.

Dealing C damage to all ships currently within the blast zone is as simple as it gets - simply add C to the root node's (C) value, and let the lazy propagation handle it later! Finally, querying the total amount of damage done to an interval of ships can be done by adding together the appropriate nodes' (B) values, in standard segment tree fashion.

There are at most N events to process and M operations to handle, each invoking a corresponding operation on the segment tree, each of which taking no more than $O(\log N)$ amortized time. Therefore, the total time complexity of the algorithm is $O((N + M) \log N)$. The following solution implements this idea. Note that epsilon comparisons and 96-bit `long doubles` are necessary to preserve precision in calculations.

Official Solution (C++)

```
#include <algorithm>
#include <cmath>
#include <iostream>
#include <vector>
using namespace std;

const int MAXN = 400000;
const int SIZE = 1048576; //2 ^ (ceil(log2(MAXN)) + 1)
const long double EPS = 1E-12;

inline bool between(long double x, long double l, long double h) {
  return l - EPS < x && x < h + EPS;
}

inline long double dist(long double x, long double y) {
  return sqrt(x * x + y * y);
}

int size = 1;
long long sum[SIZE], mult[SIZE], lazy[SIZE];

void propagate(int n, int l, int h) {
  sum[n] += lazy[n] * mult[n];
  if (n < size) {
    lazy[2*n + 1] += lazy[n];
    lazy[2*n + 2] += lazy[n];
  }
  lazy[n] = 0;
}

void update1(int n, int l, int h, int a, int v) {
  propagate(n, l, h);
  mult[n] += v;
  if (n >= size) return;
  int m = (l + h) / 2;
  if (a <= m)
    update1(2*n + 1, l, m, a, v);
  else
    update1(2*n + 2, m + 1, h, a, v);
}

void update2(int v) {
  propagate(0, 1, size);
  lazy[0] += v;
}

long long query(int n, int l, int h, int a, int b) {
  propagate(n, l, h);
  if (a <= l && h <= b) return sum[n];
  long long res = 0;
  int m = (l + h) / 2;
  if (a <= m)
    res += query(2*n + 1, l, m, a, b);
  if (b > m)
    res += query(2*n + 2, m + 1, h, a, b);
  return res;
}
```

```
int N, M, X1, Y1, X2, Y2, x, y, dx, dy, A, B, C;
vector< long double > curr;
vector< pair< long double, pair<int, int> > > events;

int main() {
  ios_base::sync_with_stdio(false);
  cin >> N >> M >> X1 >> Y1 >> X2 >> Y2;
  for (int i = 1; i <= N; i++) {
    cin >> x >> y >> dx >> dy;
    curr.clear();
    if (dx == 0) {
      if (x < X1 || x > X2) continue;
      curr.push_back(Y1 - y);
      curr.push_back(Y2 - y);
      if (dy < 0) {
        for (int j = 0; j < (int)curr.size(); j++)
          curr[j] = -curr[j];
      }
    } else {
      long double m = (long double)dy / dx;
      long double px = -1E6, py = y - m * (x + 1E6);
      long double d = dist(x - px, y - py);
      y = py + m * (X1 - px);
      if (between(y, Y1, Y2)) curr.push_back(dist(X1 - px, y - py) - d);
      y = py + m * (X2 - px);
      if (between(y, Y1, Y2)) curr.push_back(dist(X2 - px, y - py) - d);
      if (dy != 0) {
        x = (m * px + Y1 - py) / m;
        if (between(x, X1, X2)) curr.push_back(dist(x - px, Y1 - py) - d);
        x = (m * px + Y2 - py) / m;
        if (between(x, X1, X2)) curr.push_back(dist(x - px, Y2 - py) - d);
      }
      if (dx < 0) {
        for (int j = 0; j < (int)curr.size(); j++)
          curr[j] = -curr[j];
      }
    }
    if (curr.size() >= 2) {
      sort(curr.begin(), curr.end());
      events.push_back(make_pair(curr[0] / dist(dx, dy) - EPS, make_pair(i, 1)));
      events.push_back(make_pair(curr.back() / dist(dx, dy) + EPS, make_pair(i, -1)));
    }
  }
  sort(events.begin(), events.end());
  while (size <= N) size *= 2;
  int j = 0, t = 0;
  for (int m = 0; m < M; m++) {
    cin >> A >> B >> C;
    if (A == 1) t += B;
    while (j < (int)events.size() && events[j].first < t) {
      update1(0, 1, size, events[j].second.first, events[j].second.second);
      j++;
    }
    if (A == 1) {
      update2(C);
    } else {
      cout << query(0, 1, size, B, C) << endl;
    }
  }
  return 0;
}
```