A Airline Internet

Author:
Preparation:

While not theoretically too challenging, this problem required a lot of care to code properly. Many coders will have quickly seen that the way to proceed is by binary search on the transceiver radius R. This function is obviously monotonic, so as long as we can answer the question, "given a value of R, are the aircraft connected to the Internet for all time?" we will be able to converge to the correct answer.

We therefore need to consider how the connectivity of the graph changes in time. For time $T < 0$, we know that all the aircraft are sitting at airports, so they are all at distance of 0 from an airport, so must be connected. As time gets large, all the aircraft land, so they are once again connected. Now we just need to determine whether there is a period of time in between where some aircraft becomes disconnected. The way to do this is by realising that if the graph connectivity has changed, then it means that some aircraft has either come into range or gone out of range of an airport or another aircraft. So right at the point in time where the connectivity changes, the distance between two aircraft, or an aircraft and an airport must be exactly R. These "events" mark the only points in time where the graph connectivity can change. We can therefore divide time up into ranges by these points and we only need to due a single connectivity check in each range, since the connectivity must be constant for the duration of the range. Doing a connectivity check at a known point in time with a known transceiver radius is simple using either depth-first search or breadth-first search.

The time complexity of the above method is $O(N^4 \cdot \log M)$, where N is the number of aircraft and M represents the initial binary search limits. With the low constraints (N = 20), this runs in time reasonably comfortably.

B Weighing Scale

Author:
Preparation:

The solution for this problem is quite complicated so I'll just start from the basic observations, and see how it goes from there...

Iteration 1

Removing all the difficult bits from the statement we get "bla bla bla, return the number of different pairs of weights that, bla, bla, bla." There is only around 1100 of these pairs, so it seems doable to check them one by one.

```
int[] count(String[] measures, int A, int B){
    int[] result = {0,0,0};
    n = measures.length;
    for( {C, D} in {0..n-1}^2 ){ //all pairs of indices
        if( {A,B} & {C, D} != {} ) continue; //A and B are already taken
        if( something ) increment one position in result.
    }
    return result;
}
```

Iteration 2

Now we can look a bit closer at what exactly we need to check for each of the pairs.

The number of different pairs of weights you can put on the right pan to make it lighter than the left pan, the number of pairs to make both pans the same weight, and the number of pairs to make the left pan lighter.

Keeping in mind that each weight can have only one of the three values – 10, 20, or 30 grams. We can again check all the 81 combinations and directly see which side of the scale is heavier. The difficult bit is checking if the combination of the values is possible to achieve, considering the results in measures, but for now let's just focus on the part we know how to do.

```
//... We are inside the C,D loop
possibleOutcomes = {0,0,0};
for( {w1,w2,w3,w4} in {10,20,30}^4 ){
   //Somehow checks if weights A,B,C,D can tak values w1,w2,w3,w4 respectively:
   if( !possible({A,B,C,D}, {w1,w2,w3,w4}) ) continue;
   if( w1 + w2 > w3+ w4 ) possibleOutcomes[0] = 1; //left heavier
   if( w1 + w2 == w3+ w4 ) possibleOutcomes[1] = 1; //same
   if( w1 + w2 < w3+ w4 ) possibleOutcomes[2] = 1; //left lighter
}
if( sum(possibleOutcomes) == 1 ){ //checking if there is only one possible outcom
   result += possibleOutcomes;
}
```

So we loop through $\{w_1, w_2, w_3, w_4\}$ and, ignoring the combinations that are not possible, compare the total weight of the left pan with the right pan. The line number 10 implements the sentence "*You should only consider the pairs that make the result unambiguously predictable based on the*

results in measures." If we are able to predict the outcome unambiguously, that means that the left pan was always in the same relation to the right pan, thus `possibleOutcomes` has only a single 1 in it. The last step is to increment the `result` on the correct position.

Iteration 3

Now the solution is almost ready and we still haven't had to do anything difficult! The only part left is a function checking if the weights with indices $\{A, B, C, D\}$ can have values $\{w_1, w_2, w_3, w_4\}$. That's the time when we need to take a closer look at the experiments table. It should be obvious that if we have two weights x and y and `table[x][y] == '+'` then for sure x doesn't weigh 10 grams, nor y weigh 30 grams. Moreover, if we have a weight z such as `table[y][z] == '+'`, then there is no other way to assign values than ($x = 30$; $y = 20$; $z = 10$). To check the possible values for all the weights we can do the following:

- initialize `possibleWeights[measures.length][3]` to make all the elements `true`.
- For each node x run a `dfs` that traverses only the '`+`' and '`=`' edges. The `dfs` returns a maximum number of pluses it found on a single path from x.
- If `dfs` returns 0 it doesn't give us any new knowledge about the value of x. If it returned 1 we can set `possibleWeights[x][0] = false` (index zero represents 10 grams, 1 – 20 gram, etc), as it can not be 10 grams if it's heavier than something else. When the `dfs` returns 2 we can set both `possibleWeights[x][0]` and `possibleWeights[x][1]` to `false`, which leaves 30 grams as the only possible value. Note the it can't return anything above 2, as if it did there is a bug in our code.
- Do the same as previously, but now run through '`-`' and '`=`' edges and update the `possibleWeights` matrix accordingly.

Now our code looks like this :

```
int dfs( int p, char sign, boolean[]used) {
  if(used[p]) return 0;
  used[p] = true;
  int res = 0;
  for(int q=0; q <n ;q++){
      //Edges with '=' don't add anything to the result, edges '+' or '-' do.
    if(measures[p][q] == '=' ) res = max(res, dfs( q, sign, used));
    if(measures[p][q] == sign ) res = max(res, 1 + dfs( q, sign, used));
  }
  return res;
}


int[] count(String[] measures, int A, int B){
    int[] result = {0,0,0};
    possibleWeights = {{true}};
    for( x =0; x<n; x++ ){
        int plusDepth = dfs(x, '+', new boolean[n]);
        int minuDepth = dfs(x, '-', new boolean[n]);
        //updating possibleWeight[x] based on the dephts;
    }
   //    ...
```

```
    //    Inside the loop with the four possible weight values
     if( ! possibleWeights[A][w1] ||  ! possibleWeights[B][w2] ||
         possibleWeights[C][w3] ||  ! possibleWeights[D][w4] ) continue;
     //the rest as before
}
```

And that's all.

But it disagrees on the example 3.

Iteration 4

Back to the drawing board. Let's look at example 4.

```
4
??+?
???+
-???
?-??
0
1
```

Answer:

```
1
0
0
```

We know that $w_0 > w_2$ and $w_1 > w_3$ thus $w_0 + w_1$ must be heavier than $w_2 + w_3$.

So, we have only one pair of weights to consider $\{2, 3\}$, our `possibleWeights` table looks like:

	10 grams	20 grams	30 grams
weight 0	false	true	true
weight 1	true	true	false
weight 2	false	true	true
weight 3	true	true	false

So in example we can have $30 + 20 > 20 + 10$, which makes the left pan heavier, but we can also have $20 + 20 = 20 + 20$, which makes the both pans the same weight. But wait! How can both w_0 and w_1 be 20 if the measures says that $w_1 > w_2$? Well it can't be, it's just that our program doesn't know it yet.

The example shows that it's not only important what values a single weight can have, but also if all the four values together match the relation defined by measures. To fix that we introduce one more matrix, `Re[n][n]`, where each cell can have one of the 4 values:

1. '`+`' – the i-th weight is heavier than the j-th weight.
2. '`-`' – the i-th weight is lighter than the j-th weight.
3. '`=`' – both weights have the same weight.
4. '`?`' – the two weights are not correlated.

Looks familiar, doesn't it? It is a transitive closure of the measure matrix. To compute it we can again run n pairs of `dfs`'s, just the same way as we did for computation of `possibleWeights`, only this time we are not interested in how deep we can go, but in what weights we can reach. After this is done, we check for every pair from $\{A, B, C, D\}$ and corresponding weights from $\{w_1, w_2, w_3, w_4\}$ if the values match the relation.

```
private boolean check(int a, int b, int w1, int w2) {
    if (Re[a][b] == ?)
        return true; //no relation
    if (Re[a][b] == '-' && w1 >= w2)
        return false;
    if (Re[a][b] == '=' && w1 != w2)
        return false;
    if (Re[a][b] == '+' && w1 <= w2)
        return false;
    return true;
}

/** This gets {A,B,C,D}, {w1,w2,w3,w4} **/
private boolean check(int[] ind, int[] w) {
    for (int i = 0; i < ind.length; i++)
        for (int j = i + 1; j < ind.length; j++) {
            if (ind[i] == ind[j])
                return false;
            if (!check(ind[i], ind[j], w[i], w[j]))
                return false;
        }
    return true;
}
```

Adding this check in the inner-most loop makes the solution pass. That was basically the reference solution, and the thought process behind it.

C Diligent Johnny

Author:
Preparation:

Note that we can just as well go from $(1, \ldots, n)$ to the next permutation until we arrive at the given one. In the sequel we will use this restatement.

Let $f(n)$ be the total number of swaps to proceed from $(1, \ldots, n)$ to $(n, \ldots, 1)$.

To do that, first we have to get from $(1, \ldots, n)$ to $(1, n, \ldots, 2)$. Next we have to change $(1, n, \ldots, 2)$ into $(2, 1, 3, \ldots, n)$, then to $(2, n, \ldots, 3, 1)$, and so on.

In general, we have n steps of "swap the suffix" sort. Each of them take $f(n-1)$ steps.

Between these steps we have to change $(k, n, \ldots, k+1, k-1, \ldots, 1)$ to $(k+1, 1, \ldots, k, k+2, \ldots, n)$, where $k = 1, \ldots, n-1$.

The fact: The minimal number of swaps to change a permutation p into permutation q is equal to $n -$ (number of cycles in$p^{-1}q$).

One can check with some case analysis (or with brute-force for small numbers) that the number of swaps needed to change $(k, n, \ldots, k+1, k-1, \ldots, 1)$ into $(k+1, 1, \ldots, k, k+2, \ldots, n)$ doesn't depend on k and is equal to $\lceil \frac{n}{2} \rceil$.

Thus, we have $f(n) = nf(n-1) + (n-1)\lceil \frac{n}{2} \rceil$. Values of this recurrence can readily be found in $O(n)$ time.

Now to solve the "partial" problem: find the number of steps to obtain $(p_1, \ldots, p_n)$ from $(1, \ldots, n)$.

Suppose we have just obtained the correct prefix of length $l-1$, so all the rest elements $p_l, \ldots, p_n$ are currently in increasing order.

Let x_l be the index of p_l among the remaining elements if we order them by increasing. To place p_l in l-th position we have to do $x_l - 1$ repetitions of "swap the suffix" and "apply next permutation so that l-th element increases".

By previous arguments, we have to perform $(x_l - 1)(f(n-l) + \lceil \frac{n-l}{2} \rceil)$ swaps. After that, p_l is in its place, and all the latter elements are sorted, thus we reduce to a smaller problem. Values of x_l can be found using any kind of RSQ data structure in $O(n \log n)$.

D Cool Rectangles

Author:
Preparation:

There are a few crucial observations that allow us to dissect this problem properly.

First, notice that if one cool rectangle (call this rectangle A) contains another (call this rectangle B), then rectangle A won't be used to shrink an input rectangle. Why? Well, consider an optimal solution that uses rectangle A. Then whichever input rectangle is assigned to rectangle A also contains rectangle B. Thus, we may reassign this input rectangle to rectangle B. Since rectangle B has a smaller area than rectangle A, then it may lead to more input rectangles being shrunk, which would be a contradiction. Otherwise, this new solution is still better than the original, since the total area of used cool rectangles is smaller, which is also a contradiction.

Using this observation, we can focus our attention only on cool rectangles that contain no other cool rectangles (we call these awesome rectangles). We can find all awesome rectangles by observing that a cool rectangle is awesome if, and only if, there are no line segments that pass through the interior of the cool rectangle. Therefore, we can use a simple algorithm to determine if a cool rectangle is awesome by seeing if any line segments formed by input rectangles pass through its interior by intersecting its edges. However, we must also make sure that no input rectangles are contained within the cool rectangle.

Now that we have our set of awesome rectangles (none of which overlap, and thus are independent of one another), we must assign each input rectangle to an awesome rectangle, such that the sum of the areas of the awesome rectangles is minimum. In terms of graph theory, we construct a bipartite graph with partite set A consisting of all awesome rectangles, and partite set B consisting of all input rectangles. There is an edge between two vertices in opposite sets whenever an input rectangle contains an awesome rectangle. In addition, we assign each vertex in set A a weight that is equal to the area of the awesome rectangle that it represents. Therefore, we're looking for the maximum matching in a bipartite graph such that the sum of the vertex weights is minimum.

One way of solving this is with minimum-cost maximum flow, by assigning the edges weights that correspond to the areas of the awesome rectangles that they connect to. However, a simpler solution can be found by realizing that you can greedily match the awesome rectangles, which is done by computing the lexographically first maximum bipartite matching when set A is ordered by vertex weight.

While it is clear that there are $O(n^4)$ possible cool rectangles, there are only $O(n^2)$ possible awesome rectangles (these are known as the centered square numbers). We can find the awesome rectangles in $O(n^5)$, and since finding the lexographically first maximum bipartite matching can be solved in $O(VE)$, where $V = O(n^2)$ and $E = O(n^3)$, this solution has a time complexity of $O(n^5)$.

E Amoeba

Author:
Preparation:

First, we need to introduce a few definitions:

Amoeba set is a convex set which contains only cells with antimatter.

Left border of an amoeba set S is a set of cells $L(S)$ for which the following conditions are satisfied:

1. all elements of $L(S)$ belong to S;
2. if cell (r, c) belongs to $L(S)$, then either $c = 0$ or $(r, c-1)$ doesn't belong to $L(S)$

Right border of an amoeba set S is a set of cells $R(S)$ for which the following conditions are satisfied:

1. all elements of $R(S)$ belong to S
2. if cell (r, c) belongs to $R(S)$, then either c is on the right border of table or $(r, c+1)$ doesn't belong to $R(S)$

Amoeba set S is called *left decreased* if there exist two cells (r_1, c_1) and $(r_1 + 1, c_2)$ from $L(S)$ and $c_2 > c_1$.

Amoeba set S is called *right decreased* if there exist two cells (r_1, c_1) and $(r_1 + 1, c_2)$ from $R(S)$ and $c_2 > c_1$.

Let's take some amoeba set S and let L^C be the sequence of column numbers of cells from $L(S)$ ordered by ascending order of their row number. We will define a sequence R^C in the same way for the $R(S)$.

Claim 1. There exists such number i that $L_0^C \le L_1^C \le \ldots \le L_i^C \ge L_{i+1}^C \ge \ldots \ge L_{n-1}^C$, where n is the length of the sequence L^C. This is true because the set is convex.

Claim 2. There exists such number i that $R_0^C \ge R_1^C \ge \ldots \ge R_i^C \le R_{i+1}^C \le \ldots \le R_{n-1}^C$.

Let `f(row, left, right, isLeftDecreased, isRightDecreased)` be the number of amoeba sets S with the following properties:

- all cells lie in the first row rows
- all cells in the row row between columns with numbers left and right inclusive, belong to the set
- left and right belong to $L(S)$ and $R(S)$, respectively
- if the set is left decreased `isLeftDecreased` equals 1, otherwise `isLeftDecreased` equals 0
- if the set is right decreased `isRightDecreased` equals 1, otherwise `isRightDecreased` equals 0

How to find the number of such sets? If `left` > `right` or there is a cell with matter in row `row` between columns `left` and `right`, this number is, obviously, 0. Now we will consider only cases when `left` $\le$ `right` and all cells in row `row` between columns `left` and `right` contain only antimatter. One possible set consists from all cells in row `row` in columns between `left` and `right`, inclusive.

All other sets contain the cells from row `row` -1 That makes us think that we can calculate `f(row, left, right, isLeftDecreased, isRightDecreased)` using `f(row - 1, , , , )`.

First, we need to decide what happens with left and right borders after transition from row `row` -1 to `row`. Here are 4 simple observations:

- If a set in the previous row was left decreased then its left border can not move strictly to the left after the transition to row row.
- If a set in the previous row was right decreased then its right border can not move strictly to the right.
- If the left border has moved to the left after the transition then the set couldn't be left decreased before the transition.
- If the right border has moved to the right after the transition then the set couldn't be right decreased before it.

Let's fix the types of movement after the transition from row row-1 to row of both left and right borders. For each of them there are three types of movement:

1. The border moves to the left.
2. The border moves to the right.
3. The border remains in the same column.

After fixing these types we can determine intervals of possible positions of each of the borders (let this interval be [`left1'`,`left2'`] for the left border and [`right1'`,`right2'`] for the right border). For example, if the right border moves to the left after transition then it could be on every position between `left` and `right`-1 in row `row`-1. Also we can determine if the set was left decreased or right decreased before the transition (let these states be `isLeftDecreased'` and `isRightDecreased'`). Now we need to add `f(row - 1, left', right', isLeftDecreased', isRightDecreased')` for all `left'` in [`left1'`,`left2'`] and all `right'` in [`right1'`,`right2'`] to the value of `f(row, left, right, isLeftDecreased, isRightDecreased)` we are currently calculating.

The solution described above has the complexity of $O(N \cdot M^4)$, where N and M are the number of rows and columns in the table, which is too slow for the given constraints. To speed it up one need to notice that at each transition after fixing the types of borders movement point with coordinates (`left'`, `right'`) lies in the rectangle with coordinates of the lower left corner (`left1'`, `right1'`) and the upper right corner (`left2'`, `right2'`). All we need to do is to find the sum of the values `f(row - 1, left', right', isLeftDecreased', isRightDecreased')`. To do it in $O(1)$ we need to calculate the partial sums of the values of `f(row - 1, , , isLeftDecreased', isRightDecreased')`. It can be done after finishing calculating `f(row-1, , , , )` in $O(M^2)$. Now the transition can be done in $O(1)$, thus the total complexity becomes $O(N \cdot M^2)$.

Common misconceptions:

One can think that if the intervals [`left1'`,`left2'`] and [`right1'`,`right2'`] are intersecting, we will include the values `f(row - 1, left', right', isLeftDecreased', isRightDecreased')` in which `left'` $>$ `right'`. It is true, but as it was mentioned above, these values are equal to 0.

One can think that if the intervals [`left1'`,`left2'`] and [`right1'`,`right2'`] contain cells with matter, we will include amoeba sets with matter inside. That is not true, because respective values of `f(row - 1, left', right', isLeftDecreased', isRightDecreased')` will be equal to 0.

F Blackjohn

Author:
Preparation:

In this problem, it is necessary to select a subset of fractions (cards) such that their sum equals one. Thus, this problem is a special case of the knapsack problem. One possible approach to solving the problem is to convert all fractions to a common denominator. However, the least common multiple (LCM) of all possible denominators is 232 792 560, which is too large for further solving the resulting "classic" knapsack problem.

To solve the given problem, it is necessary to divide it into independent subproblems. To do this, the fractions should be divided into five groups based on their denominators: fractions with denominators 11, 13, 17, 19, and all other fractions. Note that if the selected fractions from any of these groups do not sum up to an integer, they cannot be complemented by fractions from other groups to form an integer. This fact is related to the fact that large prime denominators are allocated to separate groups, and there cannot be other denominators that are multiples of them except for themselves.

In each group, the sum of the selected fractions must be an integer, even if it is zero. By solving the knapsack problem for each group, we can determine whether it is possible to select fractions in the group to obtain a certain integer. Only integers from -100 to 100 are of interest, as the sum of no more than 100 fractions, each of which does not exceed one in absolute value, will not exceed 100 in absolute value. Also, note that the LCM of all possible denominators in the last group is only 5040, which means that the number of possible numerator options in the sum does not exceed 1 008 001. This fact allows solving the knapsack problem in each group within the allocated time.

Knowing which integers can be obtained in each group, it is necessary to check whether it is possible to select one integer from each group to obtain a sum equal to one. This problem can be solved using dynamic programming. Let $a_{i,j}$ be a value that represents whether it is possible to obtain a sum of exactly j using the first i groups of fractions. The base case is defined as $a_{0,0} =$ `1` and $a_{0,j} =$ `false` for j from -100 to 100, excluding zero. The transition is made as follows: $a_{i,j} =$ `true` if there exists at least one k such that $a_{i-1,j-k} \wedge b_{i,k}$ is true, and `false` otherwise, where $b_{i,k}$ represents whether it is possible to select fractions in group i with a sum of k. The answer to the problem is the value of $a_{5,1}$. To find the set of fractions that need to be taken, it is sufficient to use the found dynamic programming values and, for each group, use the standard method of reconstructing the answer for the knapsack problem.

G Shuffling the Deck

Author:
Preparation:

Let's call a situation where two identical cards are placed in a row a *conflict*, and an x-conflict when two cards with the value x are placed in a row.

Let's call the original sequence of cards the *body*, which we will gradually reduce by moving cards to the *tail*. The tail will be formed in such a way that the cards in it do not form conflicts.

We will transform the original sequence into a sequence of conflicts and work with the new sequence. We will call the number written on the cards that form a conflict its *color*. Each time we move a segment of cards to the end of the deck, we will choose its endpoints inside conflicts.

Let's denote the number of conflicts as k. Since one move operation removes at most two conflicts, the problem cannot be solved in less than $\lceil \frac{k}{2} \rceil$ operations.

Let's denote m as the maximum number of conflicts of one color and call these conflicts *maximal*. Let's learn how to solve the problem in the case when $m = \lceil \frac{k}{2} \rceil$.

- k is even. Color all conflicts preceding the *maximal* ones with color a, the *maximal* conflicts with color b, and the ones following them with color c. $|a|+|b|+|c| = k$, therefore $|a|+|c| = |b| = m$. We will eliminate conflicts in pairs, first the pairs (b, c), and then, when there are no such pairs left, the pairs (a, b). Note that the tail will have the form $(b, c)^*(a, b)^*$, where $*$ means one or more repetitions. Obviously, two conflicts of the same color do not occur consecutively. When combined with the body, everything will be fine: if $|c| > 0$, then the last card with the value c will remain in place, and the tail will start with b. If $|c| = 0$, then the tail has the form $(a, b)^*$, and the last card with the value b will remain in place.
- k is odd. Apply the same coloring. If there is at least one card of color a, then we reduce the problem to the previous case by adding a virtual card and an a-conflict at the beginning. If there is a card of color c, then we add a new virtual card and a c-conflict at the end, reducing it to the previous case.

The solutions for these cases are optimal, as they achieve the lower bound of $\lceil \frac{k}{2} \rceil$ operations. In total, we know how to solve the problem optimally when the number of *maximal* conflicts is half of all conflicts (rounded up). We will call this case *basic*.

Now let's classify the initial situations based on the maximum number of consecutive cards of the same value, which we will call *maximal* cards, and denote their count as Q.

- $Q = 0$. There are no conflicts, nothing needs to be done.
- If $Q > \lceil \frac{n}{2} \rceil$, then the problem has no solution. In this case, we have less than $Q-1$ non-maximal cards, which is not enough to separate the maximal value cards.
- $Q = \lceil \frac{n}{2} \rceil$. In this case, there are enough non-maximal cards, but every second card must be maximal. By coloring the cards in three colors, we get:
  - a A-cards preceding the maximal ones
  - $b = Q$ B-cards that are maximal

  - c C-cards following the maximal ones.

  We assume that all a and c cards form conflicts. Let's consider the cases:

  - $a = 0$. Therefore, $c = n - Q = \lfloor \frac{n}{2} \rfloor$, so b differs from c by at most one, which means we reduced it to the basic case.
  - $c = 0$. Same as the previous case.
  - $a > 0$, $b > 0$. Among the A-cards, there are $a - 1$ conflicts, among the B-cards: $b - 1$, and among the C-cards: $c - 1$. $(a - 1) + (c - 1) = n - Q - 2$, which differs from $Q - 1$ by at most 1, so it is the basic case.

- $m < \lfloor \frac{n+1}{2} \rfloor$. If it is possible to add a certain number of virtual conflicts (conflicts between cards with different values) to reduce the problem to a base case, we add them and solve it as a base case. Now, we will repeatedly remove the last pair of conflicts that can be removed (the last two conflicts of different colors) (c_1, c_2). If we encounter a base case during this process, we solve it as we would solve a base case. If there are no more conflicts left, the problem is solved. We will prove that no new conflicts will arise. Let's consider the color of the last conflict z. We will transfer pairs of conflicts (x, z) until z runs out, resulting in a tail of z $(x_1, z)(x_2, z)\ldots$. If the z conflicts run out, there will be a transition to (x, y) conflicts, i.e., a pair $(x_k, z)(x_{k+1}, y)$, where $x_{k+1} \neq z$. When transitioning to the base case, no conflicts will arise because after transferring the pair (x, z), neither x nor z can become maximum (m). Since $m \neq z$, either z runs out and the conflict $(x, z)(z, \ldots)$ will not arise, or z is located after m, resulting in a continuation $(x, z)(m, z)$.

Since we remove a pair of conflicts at each step or use the base case, the constructed solution is optimal and works in $\lceil \frac{k}{2} \rceil$ operations.

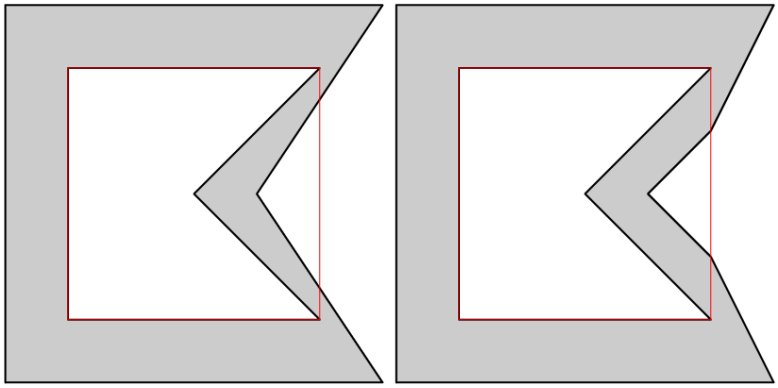
It can be noticed that there is no need to store which numbers are in the tail and in what order they are written, as there are no conflicts there. The sequence of values can be maintained using a Cartesian tree with an implicit key, and the total runtime will be $O(n \log n)$. Also, if at some point we determine that there are no conflicts at the end of the body, we can reduce the length of the body and accordingly increase the tail. We can consider all cases when a segment is moved and make sure that the body always has the form: a prefix of the original array with no more than two cut-out subsegments. Based on this fact, the solution can be implemented in $O(n)$.

H Race Track

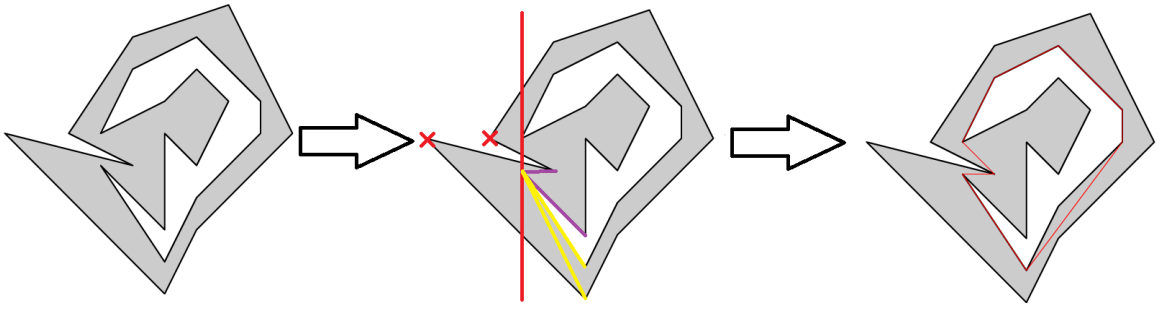
Author:
Preparation:

Let's divide the solution of the problem into two stages and consider each one separately. The first stage will be the construction of a graph, where the vertices are all the vertices of the track edges. Two vertices are connected by an edge if the segment drawn between them does not go beyond the track boundaries. The graph will be weighted, and the weight of an edge between two vertices will be equal to the distance between them.

To construct the graph, it is necessary to check if the segment goes beyond the track boundaries. This can be done by intersecting the segment with all the links of both polylines. It is also important to pay special attention to the case when the segment goes beyond the track boundaries at a vertex of the polyline. To avoid this, it can be assumed that the segment, except for its endpoints, should not have any common points with the track edges. Another case of an invalid segment is a segment that lies entirely within the polygon of the inner track edge or outside the polygon of the outer track edge. Such segments can be eliminated by carefully considering the angles at which the segment and two links of the corresponding endpoint of the polyline extend.



The leftmost vertex of the inner track edge will always lie on the optimal trajectory and will be the leftmost point of the optimal trajectory. It is required to exclude all vertices to the left of this vertex from the graph, and split the vertex itself. At the same time, the edges need to be divided between the two copies of the vertex according to whether they are above or below the links of the inner track polyline. This can also be done by considering the angles at the vertex.



The second stage of the solution will be finding the shortest path in this graph between the two copies of the split vertex. The shortest path will be the optimal track, and its length will be the length of the track. The shortest path can be found using Dijkstra's algorithm.

Constructing the graph takes $O(n^3)$ time, as each of the $O(n^2)$ possible edges needs to be intersected with $O(n)$ links of the polyline. The time complexity of finding the shortest path in a graph with $O(n)$ vertices and $O(n^2)$ edges is $O(n^2)$. Thus, the total runtime of the program is $O(n^3)$.

I The Robot

Author:
Preparation:

In this problem, the task was to find the expected length of the shortest path from one corner of the field to another. Some cells on the field were painted white, and when the robot landed on such a cell, it would be moved to a random white cell on the field.

Let's take a closer look at how the optimal path selection strategy works. For each cell on the field, we can calculate the expected number of moves required to reach the bottom right cell. The answer for the problem will correspond to the distance written in the top left cell. How can we calculate these values? Let's consider a specific cell. Let's consider all its neighbors. If a neighbor is painted black, then the answer for the cell will be at least (1 + the value written in the neighboring cell). If the cell has a white neighbor, then the answer for it is at least (1 + the arithmetic mean of the answers for all white cells).

It is clear that if the robot lands on the same cell twice, there exists an optimal strategy in which the next move is the same in both cases. If this is not the case, then in one of the moves, the robot made a mistake and went to a cell with a higher expected number of moves to the end, which means the strategy can be improved. There are two fundamentally different actions that the robot can take while in a certain cell. It can either go directly to the bottom right corner through black cells (if such a path exists), or it can go to a neighboring white cell. Obviously, among all white cells that can be reached, the closest one should be chosen.

Let's calculate the distance to the bottom right through black cells and the distance to the nearest white cell for each cell. Note that for any white cell, the distance to the nearest white cell is not greater than two. Now let's divide all white cells into two sets - those from which the robot should immediately go to the bottom right cell, and those from which the robot should go to the neighboring white cell. Once this division is made, it is not difficult to calculate the answer. By writing a recurrence relation for the average answer for all white cells, we can find it. It will be a fraction with a denominator no greater than the number of white cells. Now the answer is min(distance through black cells to the bottom right, distance to the nearest white + average answer for all white cells).

Now we need to learn how to find the correct division of white cells into sets. Let's sort all white cells according to the criterion (distance through black cells to the bottom right - distance to the nearest white). It is claimed that the optimal division is as follows - from all cells in a certain prefix of the sorted array, we should immediately go to the bottom right cell, and from the rest, we should go to the nearest white cell. This fact can be easily proven by contradiction (assume that the optimal division has a different form, consider the average answer for white cells, and see that some cells can be moved from one set to another without worsening the answer, but bringing the division to the desired form).

In the end, the solution comes down to the following. Calculate the distances discussed earlier for all cells. Sort the white cells. Iterate over the division into sets, recalculating the average answer for white cells each time in $O(1)$. The answer to the problem is the minimum of the distance through black cells and the distance to the nearest white cell + average answer for white cells. If the white cells are sorted using counting sort, the overall complexity of the solution will be O(total number of cells).

J Chessboard

Author:
Preparation:

There are two types of regular chessboards, so we can iterate two cases, try both boards.

If the final board coloring is known, then for each cell and diagonal the color is also known. For each cell, if it was not painted correctly, we need to repaint at least one of the two diagonals it is lying on.

So let's try to formalize the problem.

Consider a bipartite graph, each diagonal is a vertex, each part contains diagonals of a single type. There is an edge from one diagonal to another, if there is a cell lying on both diagonals and this cell is colored incorrectly. Then the solution would be a minimum vertex cover of this graph. To find it there is an algorithm using bipartite matching.

K Long long trip

Author:
Preparation:

Edges are small, t is large

A straightforward idea would be to do a graph search. Since the actual value of the distance of the path is important, we would make a special graph between special vertices (x, i) where i is a node in the original graph and x is the distance from 1 to i after traveling a specific path. So if (t, n) is reachable, there is a solution. The issue is that t can be excessively large (10^{18}).

Another thing to take from the constraints, however, is that the edge lengths (single roads) have a much smaller constraint in comparison — Just 10 000.

The solution to this problem is to find properties that allow us to exploit the small edge lengths.

If there was a cycle

There can be cycles in the graph and the cycles are a tool that allows us to make trips from 1 to n longer, we should analyze how they affect the problem.

For example, imagine that there is a cycle of length c that starts at a vertex x. Then for every path of length L from 1 to n that includes vertex x, we can create a new path of length $L + c$, just add the cycle to the path. We can repeat it and conclude that a path of length $L + 2c$ is also possible. In fact, all of $L, L + c, L + 2c, L + 3c, L + 4c, \ldots$ are possible. Any length K such that $K = L \pmod{c}$ and $K \geq L$ will be possible. It follows that if we find at least one path from 1 to n that visits x and has a length K such that $K \leq t$ and $K = t \pmod{c}$ then a path of length t that connects 1 and n and visits x exists.

The most important observation in this problem is that it also works the other way around. Imagine absolutely no path following those properties exists, then we can show that it is impossible to find any path of length t from 1 to n that visits x. In order to prove this, we can use a proof by contrapositive: If a path of length t that visits x exists, then at least one path of length $Klet$ with $K = t \pmod{c}$ must also exist. Indeed: Let $K = t$, this value of K fulfills the conditions.

There is always one cycle

Now consider that the graph is undirected, this means that every edge in the graph allows a cycle. An edge of length d from a to b allows a cycle of length $2d : a \to b \to a$.

Except for trivial cases in which 1 and n are disconnected, there should always be a path from 1 to n, it means there should always be at least one edge connecting 1 with some other vertex. We can take one of these edges, any of these edges, and let d be its weight. Then we can claim there is a cycle of length $2d$ that starts at vertex 1.

Apply the rule we found in the first section, we know that $x = 0$ is an edge that will always be visited in any path from 1 to n, so the cycle of length $2d$ applies: There is a path from 1 to n of exact length t if and only if there exists at least one path of length $K \leq t$ such that $K = t \pmod{2d}$.

This is a interesting part of the solution. Note that we can use any value of d as long as it is the weight of an edge connected to 1. This is regardless of whether the edge in question will actually be used in the final solution. In effect, we are using d to cut the complexity through the special property described above. The actual shape of the solution path can involve anything, including paths that do not use d, but the math above will still work.

Minimum path

Imagine a new special graph, connecting vertices of the kind (m, i). Vertex (m, i) means we reached vertex i in the original graph using a path of some length $L = m \pmod{2d}$. Note that we only need values of m smaller than $2d$. This graph is therefore not as big as the first graph we tried. Up to $20000 \cdot 51$ vertices in the worst case, each with up to 50 edges (moving from a city to another).

There is an edge from special vertex (m, i) to (n, j) if there is an edge connecting vertices i and j in the original graph and $n = m + e \pmod{2d}$ if e is the length of the edge.

It is possible to find the minimum path from $(0, 1)$ to $(t \bmod (2d), n)$ in this graph. Since it is just a weighted path we can use Dijkstra or another shortest path algorithm. The minimum distance from $(0, 1)$ to $(t \bmod (2d), n)$ is the minimum $K = t \pmod{2d}$ such that there is a path of length K from 1 to n. If this value doesn't exist or if it is greater than t, the answer is "Impossible".

L System Administrator

Author:
Preparation:

For each vertex v store three bits:

1. is its left child is connected to its right child, if we remove v,
2. is its left child is connected to its parent if we remove v,
3. is its right child is connected to its parent if we remove v

If at least two bits are set, then the computer is not important.

Simulate adding the edges to the graph, each vertex has no more that $\log_2 n$ ancestors, update the state for each ancestor.