2015-2016 ACM-ICPC Nordic Collegiate Programming Contest (NCPC 2015)

Chun-Hung Tseng

June 5, 2016

Problem A
Adjoin the Networks
Problem ID: adjoin

Wikimedia, cc-by-sa

One day your boss explains to you that he has a bunch of computer networks that are currently unreachable from each other, and he asks you, the cable expert's assistant, to adjoin the networks to each other using new cables. Existing cables in the network cannot be touched.

He has asked you to use as few cables as possible, but the length of the cables used does not matter to him, since the cables are optical and the connectors are the expensive parts. Your boss is rather picky on cable usage, so you know that the already existing networks have as few cables as possible.

Due to your humongous knowledge of computer networks, you are of course aware that the latency for an information packet travelling across the network is proportional to the number of *hops* the packet needs, where a hop is a traversal along a single cable. And since you believe a good solution to your boss' problem may earn you that long wanted promotion, you decide to minimise the maximum number of hops needed between any pair of network nodes.

Input

On the first line, you are given two positive integers, the number $1 \leq c \leq 10^5$ of computers and the number $0 \leq \ell \leq c - 1$ of existing cables. Then follow ℓ lines, each line consisting of two integers a and b, the two computers the cables connect. You may assume that every computer has a unique name between 0 and $n - 1$.

Output

The maximum number of hops in the resulting network.

Sample Input 1	Sample Output 1
6 4 0 1 0 2 3 4 3 5	3

Sample Input 2	Sample Output 2

<pre>11 9
0 1
0 3
0 4
1 2
5 4
6 4
7 8
7 9
7 10</pre>
	<pre>4</pre>

Problem A

This is a very interesting problem, which I got 12 WA during virtual participation on Codeforces. Draw the diagram out, you will realize that this problem is about tree diameter and radii pretty quickly.

You are given a forest with n vertices to start with, and you have to add one edge for each tree in the forest to connect all of them together eventually, such that the forest will become a single tree, where the furthest two leaves are of the shortest distance.

Finding the diameter and radii is the part that got me 12 WA, because I didn't implement it the right way for the first 10+ attempts. Here is a great article on Codeforces that have great explanations on this.

Let's denote D for the largest diameter of all trees in the forest, A for the largest radius, B for the second-largest radius(if exists), and C for the third-largest radius of all trees(if exists). The answer will be $max(D, A+B+1, B+C+2)$. We get this formula by two crucial observations: connect to trees using their center(s) only, and always connect to the center of the largest diameter. In this way, we can minimize the distance between two furthest leaves.

The solution is $O(n)$, because finding the diameter and radii are all linear time algorithms.

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

using namespace std;

#define N 100010
vector<int> g[N];
int deg[N];
bool seen[N];

vector<int> group;
void dfs(int u)
{
    if (seen[u] == true)
        return;
    seen[u] = true;
    group.push_back(u);

    for (auto i : g[u])
        dfs(i);
}

int main()
{
    int n, k;
    scanf("%d %d", &n, &k);
    if (k == 0) {
        printf("2\n");
        return 0;
    }

    for (int i = 0; i < n; i++) {
        g[i].clear();
        deg[i] = 0;
    }
```

```
    for (int i = 0; i < k; i++) {
        int u, v;
        scanf("%d %d", &u, &v);

        g[u].push_back(v);
        g[v].push_back(u);
        deg[u]++;
        deg[v]++;
    }

    vector<int> ans;
    for (int i = 0; i < n; i++) {
        if (seen[i] == false) {
            // find all nodes in the subtree
            group.clear();
            dfs(i);

            // add leaf to queue
            queue<int> q;
            for (auto v : group) {
                if (deg[v] == 1) {
                    q.push(v);
                }
            }

            // find tree center / radius
            int radius = 0;
            int level[100010] = {0};
            while (q.empty() == false) {
                int cur = q.front();
                q.pop();

                for (auto v : g[cur]) {
                    deg[v]--;
                    if (deg[v] == 1) {
                        level[v] = level[cur] + 1;
                        radius = max(radius, level[v]);
                        q.push(v);
                    }
                }
            }

            int cntCenter = 0;
            for (int u : group)
                if (level[u] == radius)
                    cntCenter++;
            ans.push_back(2 * radius + cntCenter - 1);
        }
    }

    sort(ans.begin(), ans.end());
    reverse(ans.begin(), ans.end());

    int mx = ans[0];
    if (ans.size() > 1)
        mx = max(mx, (ans[0] + 1) / 2 + (ans[1] + 1) / 2 + 1);
    if (ans.size() > 2)
        mx = max(mx, (ans[1] + 1) / 2 + (ans[2] + 1) / 2 + 2);
    printf("%d\n", mx);

    return 0;
```

```
}
```

A/main.cpp

Problem C
Cryptographer's Conundrum
Problem ID: conundrum

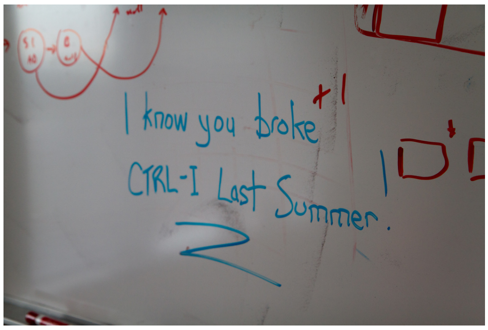


Photo by Alan Wu

The walls of the corridors at the Theoretical Computer Science group (TCS) at KTH are all but covered with whiteboards. Some of the faculty members are cryptographers, and like to write cryptographic puzzles on the whiteboards. A new puzzle is added whenever someone discovers a solution to the previous one.

When Per walked in the corridor two weeks ago, he saw that the newest puzzle read "`GuvfVfNGrfg`". After arriving at his computer, he quickly figured out that this was a simple ROT13 encryption of "`ThisIsATest`".

The series of lousy puzzles continued next week, when a new puzzle read "`VmkgdGFyIHPDpGtlcmhldGVuIHDDpSBzdMO2cnN0YSBhbGx2YXIK`". This was just base64-encoded text! "Enough with these pranks", Per thought; "I'm going to show you!"

Now Per has come up with a secret plan: every day he will erase one letter of the cipher text and replace it with a different letter, so that, in the end, the whole text reads "`PerPerPerPerPerPerPer`". Since Per will change one letter each day, he hopes that people will not notice.

Per would like to know how many days it will take to transform a given cipher text into a text only containing his name, assuming he substitutes one letter each day. You may assume that the length of the original cipher text is a multiple of 3.

For simplicity, you can ignore the case of the letters, and instead assume that all letters are upper-case.

Input

The first and only line of input contains the cipher text on the whiteboard. It consists of at most 300 upper-case characters, and its length is a multiple of 3.

Output

Output the number of days needed to change the cipher text to a string containing only Per's name.

Sample Input 1	Sample Output 1
`SECRET`	`4`

Problem C

The easiest problem in this problem set.

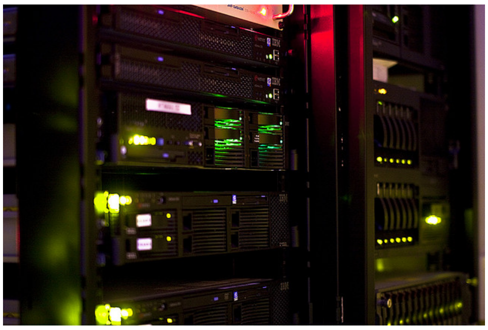
Find the total differences between the input string and string "PERPERPER..." ("PER" repeats forever), where the input string is given in a length of a multiple of 3.

The solution is $O(n)$, just linear scan and check the differences.

```cpp
#include <bits/stdc++.h>
// LLONG_MIN LLONG_MAX INT_MIN INT_MAX

#ifdef _WIN32
#define lld "I64d"
#else
#define lld "lld"
#endif

typedef long long int ll;

using namespace std;

int main()
{
    char inp[1000];
    while (scanf("%s", inp) == 1) {
        int len = strlen(inp);
        int ans = 0;
        const char *str = "PER";
        for (int i = 0; i < len / 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (inp[i * 3 + j] != str[j])
                    ans++;
            }
        }
        printf("%d\n", ans);
    }

    return 0;
}
```

C/main.cpp

Problem D
Disastrous Downtime
Problem ID: downtime



You're investigating what happened when one of your computer systems recently broke down. So far you've concluded that the system was overloaded; it looks like it couldn't handle the hailstorm of incoming requests. Since the incident, you have had ample opportunity to add more servers to your system, which would make it capable of handling more concurrent requests. However, you've simply been too lazy to do it—until now. Indeed, you shall add all the necessary servers …very soon!

To predict future requests to your system, you've reached out to the customers of your service, asking them for details on how they will use it in the near future. The response has been pretty impressive; your customers have sent you a list of the exact timestamp of every request they will ever make!

You have produced a list of all the n upcoming requests specified in milliseconds. Whenever a request comes in, it will immediately be sent to one of your servers. A request will take exactly 1000 milliseconds to process, and it must be processed right away.

Each server can work on at most k requests simultaneously. Given this limitation, can you calculate the minimum number of servers needed to prevent another system breakdown?

Input

The first line contains two integers $1 \leq n \leq 100\,000$ and $1 \leq k \leq 100\,000$, the number of upcoming requests and the maximum number of requests per second that each server can handle.

Then follow n lines with one integer $0 \leq t_i \leq 100\,000$ each, specifying that the ith request will happen t_i milliseconds from the exact moment you notified your customers. The timestamps are sorted in chronological order. It is possible that several requests come in at the same time.

Output

Output a single integer on a single line: the minimum number of servers required to process all the incoming requests, without another system breakdown.

Sample Input 1	Sample Output 1
2 1 0 1000	1

Sample Input 2	Sample Output 2
3 2 1000 1010 1999	2

Problem D

Finding the minimal number of server required to process all requests is equal to finding the maximal request at a specific time.

We can tackle the problem by splitting each request into two parts: when the request is received, we create a tuple (time, 1); when the request is finished, we create a tuple (time + 1000, 0). Sort all tuples by time (non-decreasing order) and status (non-decreasing order). Iterate over all tuples, and keep track of the maximal sum at a specific moment.

The maximal sum will tell us what is the maximal number of requests being processed at a specific moment. Divide maximal sum by the number of requests a server can handle per second, take the ceiling of the result, and will get the answer.

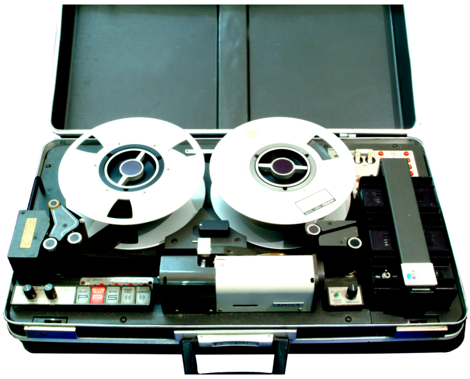
The solution is $O(nlog(n))$ because sorting is involved.

```
#include <bits/stdc++.h>
// LLONG_MIN LLONG_MAX INT_MIN INT_MAX

#ifdef _WIN32
#define lld "I64d"
#else
#define lld "lld"
#endif

using namespace std;
typedef long long int ll;
typedef pair<int, int> ii;

int main()
{
    int n, k;
    while (scanf("%d %d", &n, &k) == 2) {
        vector<ii> inp;
        for (int i = 0; i < n; i++) {
            int t;
            scanf("%d", &t);
            inp.push_back(ii(t, 1));
            inp.push_back(ii(t + 1000, 0));
        }
        sort(inp.begin(), inp.end());

        int cnt = 0, mx = 0;
        for (int i = 0; i < (int)inp.size(); i++) {
            if (inp[i].second == 1)
                cnt++;
            else
                cnt--;
            mx = max(mx, cnt);
            // printf("%d %d\n", cnt, mx);
        }
        // printf("%d %d\n", mx / k, mx % k);
        printf("%d\n", mx / k + (mx % k == 0 ? 0 : 1));
    }

    return 0;
```

```
}
```

D/main.cpp

Problem E
Entertainment Box
Problem ID: entertainmentbox



Wikimedia, cc-by-sa

Ada, Bertrand and Charles often argue over which TV shows to watch, and to avoid some of their fights they have finally decided to buy a video tape recorder. This fabulous, new device can record k different TV shows simultaneously, and whenever a show recorded in one the machine's k slots ends, the machine is immediately ready to record another show in the same slot.

The three friends wonder how many TV shows they can record during one day. They provide you with the TV guide for today's shows, and tell you the number of shows the machine can record simultaneously. How many shows can they record, using their recording machine? Count only shows that are recorded in their entirety.

Input

The first line of input contains two integers n, k ($1 \leq k < n \leq 100\,000$). Then follow n lines, each containing two integers x_i, y_i, meaning that show i starts at time x_i and finishes by time y_i. This means that two shows i and j, where $y_i = x_j$, can be recorded, without conflict, in the same recording slot. You may assume that $0 \leq x_i < y_i \leq 1\,000\,000\,000$.

Output

The output should contain exactly one line with a single integer: the maximum number of full shows from the TV guide that can be recorded with the tape recorder.

Sample Input 1	Sample Output 1
3 1 1 2 2 3 2 3	2

Sample Input 2	Sample Output 2
4 1 1 3 4 6 7 8 2 5	3

Problem E

We are trying to record as many TV shows as we can, so we first sort the TV shows by end time, in non-decreasing order, and then we try to add them to the tracks greedily.

Add the TV shows to the track using the following principle: extend the track that has the end time closest to your start time.

Because the data structure that we used for maintaining the data is c++ multiset, the solution is $O(nlog(k))$, where n is the total number of TV shows and k is the total tracks we have.

```
#include <bits/stdc++.h>
// LLONG_MIN LLONG_MAX INT_MIN INT_MAX

#ifdef _WIN32
#define lld "I64d"
#else
#define lld "lld"
#endif

using namespace std;
typedef long long int ll;
typedef pair<int, int> ii;

ii inp[100010];

bool cmp(ii a, ii b)
{
    if (a.second == b.second)
        return a.first < b.first;
    return a.second < b.second;
}
int main()
{
    int n, k;
    while (scanf("%d %d", &n, &k) == 2) {
        for (int i = 0; i < n; i++) {
            int x, y;
            scanf("%d %d", &x, &y);

            inp[i] = ii(x, y);
        }
        sort(inp, inp + n, cmp);

        multiset<int, greater<int>> s;
        int ans = 0;
        for (int i = 0; i < k; i++)
            s.insert(0);
        // for(auto i : s)
        // printf("s %d\n", i);
        for (int i = 0; i < n; i++) {
            auto it = s.lower_bound(inp[i].first);
            // printf("%d lb %d\n", inp[i].first, it == s.end() ? -1 : *it);
            if (it != s.end()) {
                s.erase(it);
                s.insert(inp[i].second);
                ans++;
            }
        }
```

```
        printf("%d\n", ans);
    }

    return 0;
}
```

E/main.cpp

Problem G
Goblin Garden Guards
Problem ID: goblingardenguards

Felipe Escobar Bravo, cc-by-nc-nd

In an unprecedented turn of events, goblins recently launched an invasion against the Nedewsian city of Mlohkcots. Goblins—small, green critters—love nothing more than to introduce additional entropy into the calm and ordered lives of ordinary people. They fear little, but one of the few things they fear is water.

The goblin invasion has now reached the royal gardens, where the goblins are busy stealing fruit, going for joyrides on the lawnmower and carving the trees into obscene shapes, and King Lrac Fatsug has decreed that this nonsense stop immediately!

Thankfully, the garden is equipped with an automated sprinkler system. Enabling the sprinklers will soak all goblins within range, forcing them to run home and dry themselves.

Serving in the royal garden guards, you have been asked to calculate how many goblins will remain in the royal garden after the sprinklers have been turned on, so that the royal gardeners can plan their next move.

Input

The input starts with one integer $1 \leq g \leq 100\,000$, the number of goblins in the royal gardens.

Then, for each goblin follows the position of the goblin as two integers, $0 \leq x_i \leq 10\,000$ and $0 \leq y_i \leq 10\,000$. The garden is flat, square and all distances are in meters. Due to quantum interference, several goblins can occupy exactly the same spot in the garden.

Then follows one integer $1 \leq m \leq 20\,000$, the number of sprinklers in the garden.

Finally, for each sprinkler follows the location of the sprinkler as two integers $0 \leq x_i \leq 10\,000$ and $0 \leq y_i \leq 10\,000$, and the integer radius $1 \leq r \leq 100$ of the area it covers, meaning that any goblin at a distance of at most r from the point (x_i, y_i) will be soaked by this sprinkler. There can be several sprinklers in the same location.

Output

Output the number of goblins remaining in the garden after the sprinklers have been turned on.

Sample Input 1	Sample Output 1
5 0 0 100 0 0 100 100 100 50 50 1 0 0 50	4

Problem G

If we use brute force for the solution: for every sprinkler, we need to iterate over all goblins to check if any of the them are within the range, this will be a $O(g * m) = O(10^9)$ solution, which is too slow to pass.

Because r is at most 100, so we can optimize the search by storing all goblins' location according to their x-coordinate. So now, for every sprinkler, we just need to check the x-coordinate range $[x - r, x + r]$, where for every x being checked, we binary search for the maximal and minimal y, $(x, max(y))$ and $(x, min(y))$ within the radius of the sprinkler's location, and remove those goblins.

The solution will be $O(nlogn)$, because we are using STL map as the underlying data structure.

```
#include <bits/stdc++.h>
// LLONG_MIN LLONG_MAX INT_MIN INT_MAX

#ifdef _WIN32
#define lld "I64d"
#else
#define lld "lld"
#endif

using namespace std;

typedef long long int ll;
typedef pair<int, int> ii;

int dist(int x, int y, int a, int b)
{
    int dx = x - a;
    int dy = y - b;
    return dx * dx + dy * dy;
}

typedef map<ii, int> data;
data loc[10010]; // for every x-coor, store location and count
set<ii> sloc;
int main()
{
    int n;
    while (scanf("%d", &n) == 1) {
        for (int i = 0; i < n; i++) {
            int x, y;
            scanf("%d %d", &x, &y);

            loc[x][(ii(x, y))]++;
        }

        int k;
        scanf("%d", &k);
        for (int i = 0; i < k; i++) {
            int x, y, r;
            scanf("%d %d %d", &x, &y, &r);
            if (sloc.find(ii(x, y)) != sloc.end())
                continue;

```

```
            for (int j = (x - r >= 0 ? x - r : 0);
                 j <= (x + r <= 10000 ? x + r : 10000); j++) {
                int dy = sqrt(r * r - (j - x) * (j - x));
                int uppery = y + dy, lowery = y - dy;

                auto it_begin = loc[j].lower_bound(ii(j, lowery));
                auto it_end = loc[j].upper_bound(ii(j, uppery));

                loc[j].erase(it_begin, it_end);
            }
        }

        int ans = 0;
        for (int i = 0; i < 10010; i++) {
            for (auto j : loc[i])
                ans += j.second;
        }
        printf("%d\n", ans);
    }
    return 0;
}
```

G/main.cpp