Solutions — SWERC Polimi selection

The following are the solutions of Saturday morning's contest.

A: Windows — easy

Even though it was the first one only because of random chance, this task was actually the easiest.

int main() {

int n, h, w;

int prev = 0;

cin >> n >> h >> w;

long long ans = 0;

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

int l, r;

cin >> l >> r;

int overlap = l + r;

if(overlap > w)

overlap = w - (overlap - w);

ans += overlap \* h;

}

cout << ans;

}

B: Major Choice — hard

This was a very hard problem. An exaustive search of all possible course arrangements (CE or CS) is far too slow. The optimal solution required you to model the task as a graph, and then apply the MaxFlow algorithm on this graph to find the optimal cost of any arrangement.

const int inf = (int)1e5;

const int N = 1000010;

int ss[N], ff[N], c[N], f[N], obr[N], pred[N], last[N];

int m;

void add(int x, int y, int z, int xx, int yy) {

m++;

ss[m] = x;

ff[m] = y;

c[m] = z;

f[m] = xx;

obr[m] = yy;

pred[m] = last[x];

last[x] = m;

}

int x[N], d[N];

int fin;

bool expath() {

for (int i = 0; i <= fin; i++) d[i] = -1;

int b = 1, e = 1;

x[1] = 0;

d[0] = 0;

while (b <= e) {

int j = last[x[b]];

while (j > 0) {

if (c[j] > f[j] && d[ff[j]] == -1) {

e++;

x[e] = ff[j];

d[x[e]] = d[x[b]] + 1;

if (x[e] == fin) {

return true;

}

}

j = pred[j];

}

b++;

}

return false;

}

int rec(int v, int w) {

if (v == fin) {

return w;

}

int r = 0, j = last[v];

while (j > 0) {

last[v] = pred[j];

if (c[j] > f[j] && d[ff[j]] == d[v] + 1) {

int ww = c[j] - f[j];

if (w - r < ww) ww = w - r;

int t = rec(ff[j], ww);

if (t > 0) {

f[j] += t;

if (obr[j] != 0) f[obr[j]] -= t;

r += t;

if (r == w) break;

}

}

j = pred[j];

}

return r;

}

int st[N];

vector <int> words[12345];

char foo[1234567];

int main() {

int tt;

tt = 1;

for (int qq = 1; qq <= tt; qq++) {

int nn;

scanf("%d", &nn);

fgets(foo, 1234566, stdin);

map <string, int> qwe;

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

fgets(foo, 1234566, stdin);

stringstream ss;

for (int j = 0; foo[j]; j++) ss << foo[j];

words[i].clear();

string word;

while (ss >> word) {

if (qwe.find(word) == qwe.end()) {

int id = qwe.size();

qwe[word] = id;

}

words[i].push\_back(qwe[word]);

}

}

int word\_cnt = qwe.size();

fin = word\_cnt \* 2 + 1;

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

last[i] = 0;

}

m = 0;

for (int i = 0; i < word\_cnt; i++) {

add(2 \* i + 1, 2 \* i + 2, 1, 0, m + 2);

add(2 \* i + 2, 2 \* i + 1, 0, 0, m);

}

for (int i = 0; i < (int)words[0].size(); i++) {

int z = words[0][i];

add(0, 2 \* z + 1, inf, 0, 0);

}

for (int i = 0; i < (int)words[1].size(); i++) {

int z = words[1][i];

add(2 \* z + 2, fin, inf, 0, 0);

}

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

int sz = words[i].size();

for (int x = 0; x < sz; x++) {

for (int y = 0; y < sz; y++) {

if (x == y) {

continue;

}

int xx = words[i][x];

int yy = words[i][y];

add(2 \* xx + 2, 2 \* yy + 1, inf, 0, m + 2);

add(2 \* yy + 1, 2 \* xx + 2, 0, 0, m);

}

}

}

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

st[i] = last[i];

}

// cout << fin << " " << m << endl;

// for (int i = 1; i <= m; i++) cout << ss[i] << " " << ff[i] << " " << c[i] << " " << f[i] << " " << obr[i] << endl;

int ans = 0;

while (expath()) {

int t = rec(0, inf);

ans += t;

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

last[i] = st[i];

}

}

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

}

}

C: Double Ladder — hard

This problem can be easily solved in quadratic time, by trying all different combinations of two ladders and putting them in some container for later lookup. That would work in the given time constraint if the maximum array size was 10k or less, but in this case it's 200k, so we need to do better.

One solution, which makes this task very hard, is to use Fast Fourier Transform: we consider the ladder heights as exponents of a polynomial in x, and then we square that polynomial, obtaining a new set of exponents, that is exactly the set of reachable floor levels. This solution has the same running time as the FFT algorithm, so NlogN.

#define LOG2 19

#define SIZE (1 << LOG2)

typedef double complex\_t[2];

static complex\_t distance[SIZE];

static int reachable[SIZE];

static void swap(complex\_t x, complex\_t y)

{

complex\_t z;

z[0] = x[0];

z[1] = x[1];

x[0] = y[0];

x[1] = y[1];

y[0] = z[0];

y[1] = z[1];

}

static void multiply(complex\_t z, complex\_t x, complex\_t y)

{

complex\_t dst;

dst[0] = x[0] \* y[0] - x[1] \* y[1];

dst[1] = x[1] \* y[0] + x[0] \* y[1];

z[0] = dst[0];

z[1] = dst[1];

}

static void set(complex\_t x, double r, double i)

{

x[0] = r;

x[1] = i;

}

/\* http://paulbourke.net/miscellaneous/dft/ \*/

static void fft(complex\_t \*xy, int m, int direction)

{

int n, i, i1, j, k, i2, l, l1, l2;

complex\_t c, t, u;

n = 1;

for (i = 0; i < m; i++)

n <<= 1;

i2 = n >> 1;

j = 0;

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

if (i < j)

swap(xy[i], xy[j]);

k = i2;

while (k <= j) {

j -= k;

k >>= 1;

}

j += k;

}

set(c, -1.0, 0.0);

l2 = 1;

for (l = 0; l < m; l++) {

l1 = l2;

l2 <<= 1;

set(u, 1.0, 0.0);

for (j = 0; j < l1; j++) {

for (i = j; i < n; i += l2) {

i1 = i + l1;

multiply(t, u, xy[i1]);

xy[i1][0] = xy[i][0] - t[0];

xy[i1][1] = xy[i][1] - t[1];

xy[i][0] += t[0];

xy[i][1] += t[1];

}

multiply(u, u, c);

}

c[1] = sqrt((1.0 - c[0]) / 2.0);

if (direction > 0)

c[1] = -c[1];

c[0] = sqrt((1.0 + c[0]) / 2.0);

}

if (direction > 0) {

for (i = 0; i < n; i++) {

xy[i][0] /= n;

xy[i][1] /= n;

}

}

}

int main(void)

{

int v, N, Q;

int possible;

memset(distance, 0, sizeof(distance));

memset(reachable, 0, sizeof(reachable));

scanf("%d%d", &N, &Q);

for (; N > 0; N--) {

scanf("%d", &v);

distance[v][0] = 1.0;

reachable[v] = 1;

}

fft(distance, LOG2, -1);

for (v = 0; v < SIZE; v++)

multiply(distance[v], distance[v], distance[v]);

fft(distance, LOG2, +1);

possible = 0;

for (; Q > 0; Q--) {

scanf("%d", &v);

if ((distance[v][0] > 0.5) || (reachable[v] != 0))

possible++;

}

printf("%d\n", possible);

return 0;

}

D: Tunnel Digging — medium

Let's consider a graph where the nodes are the N buildings and there are N² arcs, one for each pair of buildings, where the arc weight is computed with the Euclidean distance formula.

Once you have this graph, the answer is the weight of the heaviest arc in a Minimum Spanning Tree of the graph. There are many famous algorithms for solving the MST problem, like Kruskal or Prim. You can choose one and use that.

using ll = long long;

using ull = unsigned long long;

using pii = pair<int,int>;

using vi = vector<int>;

using vl = vector<ll>;

using vvl = vector<vl>;

const int INF = 2e9+5;

// this N²logN solution uses Kruskal's Algorithm, but

// it's also possible to use Prim's Algorithm instead

ll square\_distance(pii &a, pii &b){

ll x = a.first - b.first;

ll y = a.second - b.second;

return x\*x + y\*y;

}

struct dsu{

vi fs;

dsu(int size) : fs(size, -1){}

int find(int i){

if(fs[i] < 0) return i;

return fs[i] = find(fs[i]);

}

bool add\_edge(int a, int b, ll c){

a = find(a);

b = find(b);

if(a == b) return 0;

// size is stored negative

int size\_a = -fs[a];

int size\_b = -fs[b];

// to make sure a is bigger than b

if(size\_a < size\_b) swap(a, b);

// a's size increases

fs[a] += fs[b];

// b's father becomes a

fs[b] = a;

// edges was added

return 1;

}

};

struct edge {

int from, to;

ll c;

edge(int from, int to, ll c) : from(from), to(to), c(c){}

bool operator<(const edge &other) const{

return c < other.c;

}

};

int main(){

cin.tie(0), cin.sync\_with\_stdio(0), cout.tie(0), cout.sync\_with\_stdio(0);

int n; cin >> n;

vector<pii> v(n);

for(pii &p : v)

cin >> p.first >> p.second;

vector<edge> edges;

for(int i = 0; i < n; i++)

for(int j = i+1; j < n; j++)

edges.push\_back(edge(i, j, square\_distance(v[i], v[j])));

sort(edges.begin(), edges.end());

dsu g(n);

ll ans = 0;

for(edge &e : edges)

if(g.add\_edge(e.from, e.to, e.c))

ans = e.c;

cout << ceil(sqrt(ans));

return cout << '\n', 0;

}

E: Sorting Algorithm — medium

When adding a new paper in the sequence, you can efficiently look up the new location with a hash map. If the location doesn't exist, you can easily create it as it will be equal to the number of locations in the hash map plus one.

int main() {

int N, C;

cin >> N >> C;

unordered\_map<int, int> dist;

long long answer = 0;

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

int x;

cin >> x;

if (!dist.count(x)) {

int new\_dist = dist.size();

dist[x] = new\_dist + 1;

}

int num\_swaps = dist[x] - 1;

answer += num\_swaps;

}

cout << answer << endl;

}

F: Valid Parentheses — medium-hard

To solve this problem you should use a well-known technique known as **Dynamic Programming**. More explanations can be found in the code itself:

using ll = long long;

using ull = unsigned long long;

using pii = pair<int,int>;

using vi = vector<int>;

using vvi = vector<vi>;

const int INF = 2e9+5;

// Given a string, determine the number of valid substrings.

// We call a (sub)string valid if we can replace the characters

// with open or closed parenthesis - ( or ) - so that they form

// a balanced parenthesis string and if we paired parenthesis

// that were on the same character

// Core idea is that if both s(a, b) and s(b+1, c) are valid,

// then s(a, c) is also valid.

// We'll solve this problem using dynamic programming

// dp[i] = # of valid substrings starting in index i

// how to compute dp[i] ? we want to know the smallest index j

// for which s(i, j) is balanced (let it be f[i])

// so dp[i] = 1 + dp[f[i] + 1], or 0 if f[i] == INF

// how to compute f[i] ?

// if s[i+1] == s[i], then f[i] = i+1

// otherwise, things are a little more complicated

// as we need to match s[i+1] till some index j

// such that s[j+1] == s[i]

// we can see that j must be the value of f[k]

// for some index k > i

// so we can just keep track of this value z[i][c]

// for each character as we compute f[i] values

// to recap, f[i] =

// i+1, when s[i+1] == s[i]

// z[c], otherwise

int main(){

cin.tie(0), cin.sync\_with\_stdio(0), cout.tie(0), cout.sync\_with\_stdio(0);

string s; cin >> s;

// shift in (0, 25)

for(char &c : s) c -= 'a';

int n = s.size();

// compute f[i] as described above, bottomup

vi f(n, INF);

vvi z(n, vi(26, INF));

// skip last character, it can never be balanced

for(int i = n - 2; i >= 0; i--){

// compute f[i]

if(s[i+1] == s[i]){

// free match

f[i] = i+1;

}

else if(z[i+1][s[i]] != INF){

// chain to next matchable

f[i] = z[i+1][s[i]] + 1;

}

else{

// unmatchable

f[i] = INF;

}

// update z[i]

if(f[i] + 1 < n){

// copy values from earliest match

z[i] = z[f[i]+1];

// update new match

z[i][s[f[i]+1]] = f[i];

}

}

ll ans = 0;

// dp to compute solution

vi dp(n+1);

// skip last

for(int i = n-2; i >= 0; i--){

dp[i] = f[i] == INF ? 0 : 1 + dp[f[i]+1];

ans += dp[i];

}

cout << ans;

return cout << '\n', 0;

}

G: Zoning Milan — medium-hard

To solve this problem you need a well-known data structure known as **Segment Tree** (or sometimes referred to as Range Tree).

const int INF = 2e9 + 5;

const int MOD = 1e9 + 7;

typedef vector<int> vi;

typedef long long ll;

typedef pair<int, int> pii;

constexpr int left(int i){ return i\*2 + 1;}

constexpr int right(int i){ return i\*2 + 2;}

constexpr int lsb(int i) { return i&(-i);}

// using 4 segment trees

// Range min e range max are implemented separately

// the code could be shortened a lot using C++ templates

struct seg\_min{

vector<pii> seg;

int n;

pii merge(pii a, pii b){

return min(a, b);

}

pii null\_val = {INF, -1};

seg\_min(vi &v){

int size = v.size();

n = ++size;

size = 1<<(int)(log2(size-1)+2);

size = size \* 2 - 1;

seg.resize(size, null\_val);

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

seg[i + size/2] = {v[i], i};

}

for(int i = size/2 - 1; i >= 0; i--){

seg[i] = merge(seg[left(i)], seg[right(i)]);

}

/\*for(ll &l : seg)

cout << l << ' ';

cout << endl;\*/

}

pii \_query(int i, int a, int b, int l, int r){

//cout << i << ' ' << a << ' ' << b << ' ' << l << ' ' << r << '\n';

if(a > r || b < l) return null\_val;

if(a >= l && b <= r) return seg[i];

int m = (a + b) / 2;

return merge(\_query(left(i), a, m, l, r), \_query(right(i), m+1, b, l, r));

}

int min\_index(int l, int r){

//cout << "mul " << l << ' ' << r << '\n';

return \_query(0, 0, seg.size()/2, l, r).second;

}

int double\_query(int l, int i, int r){

int a, b;

if(l == i) a = b = min\_index(i+1, r);

else if(r == i) a = b = min\_index(l, i-1);

else a = min\_index(l, i-1), b = min\_index(i+1, r);

if(min(a, b) == -1) return INF;

a += seg.size() / 2, b += seg.size()/2;

return merge(seg[a], seg[b]).first;

}

};

struct seg\_max{

vector<pii> seg;

int n;

pii null\_val = {-INF, -1};

pii merge(pii a, pii b){

return max(a, b);

}

seg\_max(vi &v){

int size = v.size();

n = ++size;

size = 1<<(int)(log2(size-1)+2);

size = size \* 2 - 1;

seg.resize(size, null\_val);

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

seg[i + size/2] = {v[i], i};

}

for(int i = size/2 - 1; i >= 0; i--){

seg[i] = merge(seg[left(i)], seg[right(i)]);

}

/\*for(ll &l : seg)

cout << l << ' ';

cout << endl;\*/

}

pii \_query(int i, int a, int b, int l, int r){

//cout << i << ' ' << a << ' ' << b << ' ' << l << ' ' << r << '\n';

if(a > r || b < l) return null\_val;

if(a >= l && b <= r) return seg[i];

int m = (a + b) / 2;

return merge(\_query(left(i), a, m, l, r), \_query(right(i), m+1, b, l, r));

}

int max\_index(int l, int r){

//cout << "mul " << l << ' ' << r << '\n';

return \_query(0, 0, seg.size()/2, l, r).second;

}

int double\_query(int l, int i, int r){

int a, b;

if(l == i) a = b = max\_index(i+1, r);

else if(r == i) a = b = max\_index(l, i-1);

else a = max\_index(l, i-1), b = max\_index(i+1, r);

if(min(a, b) == -1) return INF;

a += seg.size() / 2, b += seg.size()/2;

return merge(seg[a], seg[b]).first;

}

};

signed main(){

cin.tie(0), cin.sync\_with\_stdio(0);

//ofstream fout("k.out");

int n, q;

cin >> n >> q;

if(n == 1) return 0;

vi x(n), y(n);

for(int i = 0; i < n; i++)

cin >> x[i] >> y[i];

seg\_max xmax(x), ymax(y);

seg\_min xmin(x), ymin(y);

while(q--){

int l, r;

cin >> l >> r;

l--, r--;

//we try to remove them one by one

vi remove{

xmax.max\_index(l, r),

xmin.min\_index(l, r),

ymax.max\_index(l, r),

ymin.min\_index(l, r)

};

int ans = INF;

for(int i : remove){

int top, bottom, right, left;

top = ymax.double\_query(l, i, r);

bottom = ymin.double\_query(l, i, r);

right = xmax.double\_query(l, i, r);

left = xmin.double\_query(l, i, r);

ans = min(ans, max(right - left, top - bottom));

//cout << top << ' ' << bottom << ", " << left << ' ' << right << "\n";

}

cout << ans << '\n';

}

}