

# TCR

git diff solution (Jens Heuseveldt, Ludo Pulles, Peter Ypma)

January 7, 2017

January 7, 2017

## Contents

vim ~/.vimrc

---

```
1 set nu sw=4 ts=4 noet ai hls
2 syntax on
3 colorscheme slate
```

---

template.cpp

---

```
1 #include<bits/stdc++.h>
2
3 #define x first
4 #define y second
5
6 using namespace std;
7
8 typedef long long ll;
9 typedef pair<int, int> pii;
10 typedef pair<ll, ll> pll;
11 typedef vector<int> vi;
12
13 const int INF = 2147483647; // (1 << 30) - 1 + (1 << 30)
14 const ll LLINF = (1LL << 62) - 1 + (1LL << 62); // = 9.223.372.036.854.775.807
15 const double PI = acos(-1.0);
16
17 // lambda-expression: [] (args) -> retType { body }
18
19 const bool LOG = false;
20 void Log() { if(LOG) cerr << "\n\n"; }
21 template<class T, class... S>
22 void Log(T t, S... s) {
23     if(LOG) cerr << t << "\t", Log(s...);
24 }
25
26 template<class T1, class T2>
27 ostream& operator<<(ostream& out, const pair<T1,T2> &p) {
28     return out << '(' << p.x << ", " << p.y << ')';
29 }
30
31 template<typename T1, typename T2>
32 ostream& operator<<(ostream &out, pair<T1, T2> p) {
33     return out << "(" << p.x << ", " << p.y << ")";
34 }
35
```

```

36 template<class T>
37 using min_queue = priority_queue<T, vector<T>, greater<T>>;
38
39 // Order Statistics Tree (if this is supported by the judge software)
40 #include <ext/pb_ds/assoc_container.hpp>
41 #include <ext/pb_ds/tree_policy.hpp>
42 using namespace __gnu_pbds;
43 template<class TIn, class TOut> // key, value types. TOut can be null_type
44 using order_tree = tree<TIn, TOut, less<TIn>,
45     rb_tree_tag, tree_order_statistics_node_update>;
46 // find_by_order(int r) (0-based)
47 // order_of_key(TIn v)
48 // use key pair<TIn,int> {value, counter} for multiset/multimap
49
50 int main() {
51     ios_base::sync_with_stdio(false); // faster IO
52     cin.tie(NULL); // faster IO
53     cerr << boolalpha; // (print true/false)
54     (cout << fixed).precision(10); // set floating point precision
55     // TODO: code
56     return 0;
57 }

```

---

Prime numbers: 982451653, 81253449,  $10^3 + \{-9, -3, 9, 13\}$ ,  $10^6 + \{-17, 3, 33\}$ ,  $10^9 + \{7, 9, 21, 33, 87\}$

## 0.1 De winnende aanpak

- Goed slapen & een vroeg ritme hebben
- Genoeg drinken & eten tijdens de wedstrijd
- Een lijst van alle problemen met info waar het over gaat, en wie het goed kan oplossen
- Ludo moet de opgave goed lezen
- Test de kleine voorbeeldgevallen
- Houd na 2 uur een pauze en overleg waar iedereen mee bezig is
- Maak zelf wat test-cases
- Na een WA, print het probleem, en probeer het ook weg te leggen
- Typ de dingen uit de TCR, die je zeker nodig hebt, alvast in
- Peter moet meer papier gebruiken om fouten te voorkomen
- Als iemand niks te doen heeft, kan hij nodige dingen uit de TCR typen.
- We moeten ook een voorbeeld test-case voor TCR algoritmes hebben om te testen of het goed overgetypt is
- Bij geometrie moeten we om kunnen gaan met meerdere input manieren (voor bv. lijnen)
- Gebruik veel long long's
- Bij een verkeerd antwoord, kijk naar genoeg debug output

## 0.2 Detecting overflow

These are GNU builtins, detect both over- and underflow. Returns a boolean upon failure, otherwise the result is present in ref. Follow the template:

```
_builtin_[u|s][add|mul|sub](ll)?_overflow(in, out, &ref)
```

## 0.3 Wrong Answer

- Edge cases:  $n \in \{-1, 0, 1, 2\}$ . Empty list/graph?
- Beware of typos
- Test sample input; make custom testcases
- Read carefully
- Check bounds (use long long or long double)

- Off by one error (in indices or loop bounds)
- Not enough precision
- Assertions
- Missing modulo operators
- Cases that need a (completely) different approach

## 0.4 Covering problems

*Minimum edge cover  $\iff$  Maximum independent set*

**Matching** A set of edges without common vertices (*Maximum is the **largest** such set, maximal is a set which you cannot add more edges to without breaking the property.*).

**Minimum Vertex Cover** A set vertices (cover) such that each edge in the graph is incident to at least one vertex of the set.

**Minimum Edge Cover** A set of edges (cover) such that every vertex is incident to at least one edge of the set.

**Maximum Independent Set** A set of vertices in a graph such that no two of them are adjacent.

**König's theorem** In any bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover

A useful identity:  $\bigoplus_{x=0}^{a-1} x = \{0, a-1, 1, a\}[a \bmod 4]$ .

## 1 Mathematics

---

```

1 int abs(int x) { return x > 0 ? x : -x; }
2 int sign(int x) { return (x > 0) - (x < 0); }
3
4 // greatest common divisor
5 ll gcd(ll a, ll b) { while (b) { a %= b; swap(a, b); } return a; }
6 // least common multiple
7 ll lcm(ll a, ll b) { return a / gcd(a, b) * b; }
8 ll mod(ll a, ll m) { return ((a % b) + b) % b; }
9
10 // safe multiplication (ab % m) for m <= 4e18 in O(log b)
11 ll modmul(ll a, ll b, ll m) {
12     ll r = 0;
13     while (b) {
14         if (b & 1) r = mod(r + a, m);
15         a = mod(a + a, m);
16         b >>= 1;
17     }
18     return r;
19 }
20
21 // safe exponentiation (a^b % m) for m <= 2e9 in O(log b)
22 ll modpow(ll a, ll b, ll m) {
23     ll r = 1;
24     while (b) {
25         if (b & 1) r = (r * a) % m;
26         a = (a * a) % m;
27         b >>= 1;
28     }
29     return r;
30 }
31
32 // returns x, y such that ax + by = gcd(a, b)
33 ll egcd(ll a, ll b, ll &x, ll &y)
34 {
35     ll xx = y = 0, yy = x = 1;
36     while (b) {

```

```

37     x -= a / b * xx; swap(x, xx);
38     y -= a / b * yy; swap(y, yy);
39     a %= b; swap(a, b);
40 }
41 return a;
42 }
43
44 // Chinese remainder theorem
45 const pll NO_SOLUTION(0, -1);
46 // Returns (u, v) such that x = u % v <=> x = a % n and x = b % m
47 pll crt(ll a, ll n, ll b, ll m)
48 {
49     ll s, t, d = egcd(n, m, s, t), nm = n * m;
50     if (mod(a - b, d)) return NO_SOLUTION;
51     return pll(mod(s * b * n + t * a * m, nm) / d, nm / d);
52     /* when n, m > 10^6, avoid overflow:
53     return pll(mod(modmul(modmul(s, b, nm), n, nm) +
54                     modmul(modmul(t, a, nm), m, nm), nm) / d, nm / d); */
55 }
56
57 int phi[N]; // phi[i] = #{ j | gcd(i, j) = 1 }
58
59 void sievePhi() {
60     for (int i = 0; i < N; i++) phi[i] = i;
61     for (int i = 2; i < N; i++)
62         if (phi[i] == i)
63             for (int j = i; j < N; j += i)
64                 phi[j] -= phi[j] / i * (i - 1);
65 }
66
67 // calculate nCk % p (p prime!)
68 ll lucas(ll n, ll k, ll p) {
69     ll ans = 1;
70     while (n) {
71         ll np = n % p, kp = k % p;
72         if (np < kp) return 0;
73         ans = mod(ans * binom(np, kp), p); // (np C kp)
74         n /= p; k /= p;
75     }
76     return ans;
77 }

```

---

## 2 Datastructures

### 2.1 Segment tree $\mathcal{O}(\log n)$

```

1 typedef /* Tree element */ S;
2 const int n = 1 << 20;
3 S t[2 * n];
4
5 // sum segment tree
6 S combine(S l, S r) { return l + r; }
7 // max segment tree
8 S combine(S l, S r) { return max(l, r); }
9
10 void build() {
11     for (int i = n; --i > 0; )
12         t[i] = combine(t[2 * i], t[2 * i + 1]);
13 }
14
15 // set value v on position p
16 void update(int p, int v) {
17     for (t[p += n] = v; p /= 2; )
18         t[p] = combine(t[2 * p], t[2 * p + 1]);
19 }

```

---

```

20
21 // sum on interval [l, r)
22 S query(int l, int r) {
23     S resL, resR;
24     for (l += n, r += n; l < r; l /= 2, r /= 2) {
25         if (l & 1) resL = combine(resL, t[l++]);
26         if (r & 1) resR = combine(t[--r], resR);
27     }
28     return combine(resL, resR);
29 }

```

---

## 2.2 Binary Indexed Tree $\mathcal{O}(\log n)$

Use one-based indices!

---

```

1 int bit[MAXN];
2
3 // arr[idx] += val
4 void update(int idx, int val) {
5     while (idx < MAXN) bit[idx] += val, idx += idx & -idx;
6 }
7
8 // returns sum of arr[i], where i: [1, idx]
9 int query(int idx) {
10     int ret = 0;
11     while (idx) ret += bit[idx], idx -= idx & -idx;
12     return ret;
13 }

```

---

## 2.3 Trie

---

```

1 const int SIGMA = 26;
2
3 struct trie {
4     bool word;
5     trie **child;
6
7     trie() : word(false), child(new trie*[SIGMA]) {
8         for (int i = 0; i < SIGMA; i++) child[i] = NULL;
9     }
10
11     void addWord(const string &str)
12     {
13         trie *cur = this;
14         for (char ch : str) {
15             int idx = ch - 'a';
16             if (!cur->child[idx]) cur->child[idx] = new trie();
17             cur = cur->child[idx];
18         }
19         cur->word = true;
20     }
21
22     bool isWord(const string &str)
23     {
24         trie *cur = this;
25         for (char ch : str) {
26             int idx = ch - 'a';
27             if (!cur->child[idx]) return false;
28             cur = cur->child[idx];
29         }
30         return cur->word;
31     }
32 };

```

---

## 2.4 Disjoint-Set / Union-Find $\mathcal{O}(\alpha(n))$

---

```

1 int par[MAXN], rnk[MAXN];
2
3 void uf_init(int n) {
4     fill_n(par, n, -1);
5     fill_n(rnk, n, 0);
6 }
7
8 int uf_find(int v) {
9     return par[v] < 0 ? v : par[v] = uf_find(par[v]);
10 }
11
12 void uf_union(int a, int b) {
13     if ((a = uf_find(a)) == (b = uf_find(b))) return;
14     if (rnk[a] < rnk[b]) swap(a, b);
15     if (rnk[a] == rnk[b]) rnk[a]++;
16     par[b] = a;
17 }

```

---

## 3 Graph Algorithms

### 3.1 Maximum matching $\mathcal{O}(nm)$

This problem could be solved with a flow algorithm like Dinic's algorithm which runs in  $\mathcal{O}(\sqrt{V}E)$ , too.

---

```

1 const int nodesLeft = 1e4, nodesRight = 1e4;
2 bool vis[nodesRight]; // vis[rightnodes]
3 int par[nodesRight]; // par[rightnode] = leftnode
4 vector<int> adj[nodesLeft]; // adj[leftnode][i] = rightnode
5
6 bool match(int cur) {
7     for (int nxt : adj[cur]) {
8         if (vis[nxt]) continue;
9         vis[nxt] = true;
10        if (par[nxt] == -1 || match(par[nxt])) {
11            par[nxt] = cur;
12            return true;
13        }
14    }
15    return false;
16 }
17
18 // perfect matching iff matches == nodesLeft && matches == nodesRight
19 int maxmatch() {
20     int matches = 0;
21     for (int i = 0; i < nodesLeft; i++) {
22         fill_n(vis, nodesRight, false);
23         if (match(i)) matches++;
24     }
25     return matches;
26 }

```

---

### 3.2 Strongly Connected Components $\mathcal{O}(V + E)$

---

```

1 vector<vi> adj, comps;
2 vi tidx, lnk, cnr, st;
3 vector<bool> vis;
4 int age, ncomps;
5
6 void tarjan(int v) {
7     tidx[v] = lnk[v] = ++age;
8     vis[v] = true;

```

```

9     st.push_back(v);
10
11     for (int w : adj[v]) {
12         if (!tidx[w]) tarjan(w), lnk[v] = min(lnk[v], lnk[w]);
13         else if (vis[w]) lnk[v] = min(lnk[v], tidx[w]);
14     }
15
16     if (lnk[v] != tidx[v]) return;
17
18     comps.push_back(vi());
19     int w;
20     do {
21         vis[w = st.back()] = false;
22         cnr[w] = ncomps;
23         comps.back().push_back(w);
24         st.pop_back();
25     } while (w != v);
26     ncomps++;
27 }
28
29 void findComps(int n) {
30     age = ncomps = 0;
31     vis.assign(n, false);
32     tidx.assign(n, 0);
33     lnk.resize(n);
34     cnr.resize(n);
35     comps.clear();
36
37     for (int i = 0; i < n; i++)
38         if (tidx[i] == 0) tarjan(i);
39 }

```

---

### 3.2.1 2-SAT $\mathcal{O}(V + E)$

```

1 void init2sat(int n) { adj.assign(2 * n, vi()); }
2
3 // vl, vr = true -> variable l, variable r should be negated.
4 void imply(int xl, bool vl, int xr, bool vr) {
5     adj[2 * xl + vl].push_back(2 * xr + vr);
6     adj[2 * xr + !vr].push_back(2 * xl + !vl);
7 }
8
9 void satOr(int xl, bool vl, int xr, bool vr) {
10     imply(xl, !vl, xr, vr);
11 }
12 void satConst(int x, bool v) {
13     imply(x, !v, x, v);
14 }
15 void satIff(int xl, bool vl, int xr, bool vr) {
16     imply(xl, vl, xr, vr);
17     imply(xr, vr, xl, vl);
18 }
19
20 bool solve2sat(int n, vector<bool> &sol) {
21     findComps(2 * n);
22     for (int i = 0; i < n; i++)
23         if (cnr[2 * i] == cnr[2 * i + 1]) return false;
24     vector<bool> seen(n, false);
25     sol.assign(n, false);
26     for (vi &comp : comps) {
27         for (int v : comp) {
28             if (seen[v / 2]) continue;
29             seen[v / 2] = true;
30             sol[v / 2] = v & 1;
31         }
32     }

```

---

```

33     return true;
34 }

```

---

### 3.3 Shortest path

#### 3.3.1 Floyd-Warshall $\mathcal{O}(V^3)$

---

```

1 int n = 100, d[MAXN][MAXN];
2 for (int i = 0; i < n; i++) fill_n(d[i], n, INF / 3);
3 // set direct distances from i to j in d[i][j] (and d[j][i])
4 for (int i = 0; i < n; i++)
5     for (int j = 0; j < n; j++)
6         for (int k = 0; k < n; k++)
7             d[j][k] = min(d[j][k], d[j][i] + d[i][k]);

```

---

#### 3.3.2 Bellman Ford $\mathcal{O}(VE)$

This is only useful if there are edges with weight  $w_{ij} < 0$  in the graph.

---

```

1 vector< pair<pii,int> > edges; // ((from, to), weight)
2 vector<int> dist (MAXN);
3
4 // when undirected, add back edges
5 bool bellman_ford(int source) {
6     fill_n(dist, MAXN, INF / 3);
7     dist[source] = 0;
8
9     bool updated = true;
10    int loops = 0;
11    while (updated && loops < n) {
12        updated = false;
13        for (auto e : edges) {
14            int alt = dist[e.x.x] + e.y;
15            if (alt < dist[e.x.y]) {
16                dist[e.x.y] = alt;
17                updated = true;
18            }
19        }
20    }
21    return loops < n; // loops >= n: negative cycles
22 }

```

---

### 3.4 Max-flow min-cut

#### 3.4.1 Dinic's Algorithm $\mathcal{O}(V^2E)$

Let's hope this algorithm works correctly! ...

---

```

1 // http://www.slideshare.net/KuoE0/acmicpc-dinics-algorithm
2 struct edge {
3     int to, rev;
4     ll cap, flow;
5     edge(int t, int r, ll c) : to(t), rev(r), cap(c), flow(0) {}
6 };
7
8 int s, t, level[MAXN]; // s = source, t = sink
9 vector<edge> g[MAXN];
10
11 bool dinic_bfs() {
12     fill_n(level, MAXN, 0);
13     level[s] = 1;
14
15     queue<int> q;

```



```

16     q.push(s);
17     while (!q.empty()) {
18         int cur = q.front();
19         q.pop();
20         for (edge e : g[cur]) {
21             if (level[e.to] == -1 && e.flow < e.cap) {
22                 level[e.to] = level[cur] + 1;
23                 q.push(e.to);
24             }
25         }
26     }
27     return level[t] != -1;
28 }
29
30 ll dinic_dfs(int cur, ll maxf) {
31     if (cur == t) return maxf;
32
33     ll f = 0;
34     bool isSat = true;
35     for (edge &e : g[cur]) {
36         if (level[e.to] != level[cur] + 1 || e.flow == e.cap)
37             continue;
38         ll df = dinic_dfs(e.to, min(maxf - f, e.cap - e.flow));
39         f += df;
40         e.flow += df;
41         g[e.to][e.rev].flow -= df;
42         isSat &= e.flow == e.cap;
43         if (maxf == f) break;
44     }
45     if (isSat) level[cur] = 0;
46     return f;
47 }
48
49 ll dinic_maxflow() {
50     ll f = 0;
51     while (dinic_bfs()) f += dinic_dfs(s, LLINF);
52     return f;
53 }
54
55 void add_edge(int fr, int to, ll cap) {
56     g[fr].push_back(edge(to, g[to].size(), cap));
57     g[to].push_back(edge(fr, g[fr].size() - 1, 0));
58 }

```

---

### 3.5 Min-cost max-flow

Find the cheapest possible way of sending a certain amount of flow through a flow network.

```

1 struct edge {
2     // to, rev, flow, capacity, weight
3     int t, r;
4     ll f, c, w;
5     edge(int _t, int _r, ll _c, ll _w) : t(_t), r(_r), f(0), c(_c), w(_w) {}
6 };
7
8 int n, par[MAXN];
9 vector<edge> adj[MAXN];
10 ll dist[MAXN];
11
12 bool findPath(int s, int t)
13 {
14     fill_n(dist, n, LLINF);
15     fill_n(par, n, -1);
16
17     priority_queue<pii, vector<pii>, greater<pii> > q;
18     q.push(pii(dist[s] = 0, s));
19

```

---

```

20     while (!q.empty()) {
21         int d = q.top().x, v = q.top().y;
22         q.pop();
23         if (d > dist[v]) continue;
24
25         for (edge e : adj[v]) {
26             if (e.f < e.c && d + e.w < dist[e.t]) {
27                 q.push(pii(dist[e.t] = d + e.w, e.t));
28                 par[e.t] = e.r;
29             }
30         }
31     }
32     return dist[t] < INF;
33 }
34
35 pair<ll, ll> minCostMaxFlow(int s, int t)
36 {
37     ll cost = 0, flow = 0;
38     while (findPath(s, t)) {
39         ll f = INF, c = 0;
40         int cur = t;
41         while (cur != s) {
42             const edge &rev = adj[cur][par[cur]], &e = adj[rev.t][rev.r];
43             f = min(f, e.c - e.f);
44             cur = rev.t;
45         }
46         cur = t;
47         while (cur != s) {
48             edge &rev = adj[cur][par[cur]], &e = adj[rev.t][rev.r];
49             c += e.w;
50             e.f += f;
51             rev.f -= f;
52             cur = rev.t;
53         }
54         cost += f * c;
55         flow += f;
56     }
57     return pair<ll, ll>(cost, flow);
58 }
59
60 inline void addEdge(int from, int to, ll cap, ll weight)
61 {
62     adj[from].push_back(edge(to, adj[to].size(), cap, weight));
63     adj[to].push_back(edge(from, adj[from].size() - 1, 0, -weight));
64 }

```

---

## 3.6 Minimal Spanning Tree

### 3.6.1 Kruskal $\mathcal{O}(E \log V)$

---

```

1 struct edge { int x, y, w; };
2 edge edges[MAXM];
3
4 ll kruskal(int n, int m) { // n: #vertices, m: #edges
5     uf_init(n);
6     sort(edges, edges + m, [] (edge a, edge b) -> bool { return a.w < b.w; });
7     ll ret = 0;
8     while (m--) {
9         if (uf_find(edges[m].x) == uf_find(edges[m].y)) continue;
10        ret += edges[m].w;
11        uf_union(edges[m].x, edges[m].y);
12    }
13    return ret;
14 }

```

---

## 4 String algorithms

### 4.1 Z-algorithm $\mathcal{O}(n)$

---

```

1 // z[i] = length of longest substring starting from s[i] which is also a prefix of s.
2 vector<int> z_function(const string &s) {
3     int n = (int) s.length();
4     vector<int> z(n);
5     for (int i = 1, l = 0, r = 0; i < n; ++i) {
6         if (i <= r)
7             z[i] = min (r - i + 1, z[i - l]);
8         while (i + z[i] < n && s[z[i]] == s[i + z[i]])
9             ++z[i];
10        if (i + z[i] - 1 > r)
11            l = i, r = i + z[i] - 1;
12    }
13    return z;
14 }
```

---

### 4.2 Suffix array $\mathcal{O}(n \log^2 n)$

This creates an array  $P[0], P[1], \dots, P[n-1]$  such that the suffix  $S[i \dots n]$  is the  $P[i]^{th}$  suffix of  $S$  when lexicographically sorted.

---

```

1 typedef pair<pii, int> tii;
2
3 const int maxlogn = 17, int maxn = 1 << maxlogn;
4
5 tii make_triple(int a, int b, int c) { return tii(pii(a, b), c); }
6
7 int p[maxlogn + 1][maxn];
8 tii L[maxn];
9
10 int suffixArray(string S)
11 {
12     int N = S.size(), stp = 1, cnt = 1;
13     for (int i = 0; i < N; i++) p[0][i] = S[i];
14     for (; cnt < N; stp++, cnt <= 1) {
15         for (int i = 0; i < N; i++) {
16             L[i] = tii(pii(p[stp-1][i], i + cnt < N ? p[stp-1][i + cnt] : -1), i);
17         }
18         sort(L, L + N);
19         for (int i = 0; i < N; i++) {
20             p[stp][L[i].y] = i > 0 && L[i].x == L[i-1].x ? p[stp][L[i-1].y] : i;
21         }
22     }
23     return stp - 1; // result is in p[stp - 1][0 .. (N - 1)]
24 }
```

---

### 4.3 Longest Common Subsequence $\mathcal{O}(n^2)$

SUBSTRING: *consecutive characters* !!!

---

```

1 int table[STR_SIZE][STR_SIZE]; // DP problem
2
3 int lcs(const string &w1, const string &w2) {
4     int n1 = w1.size(), n2 = w2.size();
5     for (int i = 0; i <= n1; i++) table[i][0] = 0;
6     for (int j = 0; j <= n2; j++) table[0][j] = 0;
7
8     for (int i = 1; i < n1; i++) {
9         for (int j = 1; j < n2; j++) {
10             table[i][j] = w1[i - 1] == w2[j - 1] ?
```

```

11         (table[i - 1][j - 1] + 1) :
12         max(table[i - 1][j], table[i][j - 1]);
13     }
14 }
15 return table[n1][n2];
16 }
17
18 // backtrack
19 string getLCS(const string &w1, const string &w2) {
20     int i = w1.size(), j = w2.size();
21     string ret = "";
22     while (i > 0 && j > 0) {
23         if (w1[i - 1] == w2[j - 1]) ret += w1[--i], j--;
24         else if (table[i][j - 1] > table[i - 1][j]) j--;
25         else i--;
26     }
27     reverse(ret.begin(), ret.end());
28     return ret;
29 }

```

---

#### 4.4 Levenshtein Distance $\mathcal{O}(n^2)$

```

1 int costs[MAX_SIZE][MAX_SIZE]; // DP problem
2
3 int levDist(const string &w1, const string &w2) {
4     int n1 = w1.size(), n2 = w2.size();
5     for (int i = 0; i <= n1; i++) costs[i][0] = i; // removal
6     for (int j = 0; j <= n2; j++) costs[0][j] = j; // insertion
7     for (int i = 1; i <= n1; i++) {
8         for (int j = 1; j <= n2; j++) {
9             costs[i][j] = min(
10                 min(costs[i - 1][j] + 1, costs[i][j - 1] + 1),
11                 costs[i - 1][j - 1] + (w1[i - 1] != w2[j - 1])
12             );
13         }
14     }
15     return costs[n1][n2];
16 }

```

---

#### 4.5 Knuth-Morris-Pratt algorithm $\mathcal{O}(N + M)$

```

1 int kmp_search(const string &word, const string &text) {
2     int n = word.size();
3     vector<int> table(n + 1, 0);
4     for (int i = 1, j = 0; i < n; ) {
5         if (word[i] == word[j]) table[++i] = ++j; // match
6         else if (j > 0) j = table[j]; // fallback
7         else i++; // no match, keep zero
8     }
9     int matches = 0;
10    for (int i = 0, j = 0; i < text.size(); ) {
11        if (text[i] == word[j]) {
12            i++;
13            if (++j == n) { // match at interval [i - n, i)
14                matches++;
15                j = table[j];
16            }
17        } else if (j > 0) j = table[j];
18        else i++;
19    }
20    return matches;
21 }

```

---

## 4.6 Aho-Corasick Algorithm $\mathcal{O}(N + \sum_{i=1}^m |S_i|)$

All given patterns must be unique!

---

```

1  const int MAXP = 100, MAXLEN = 200, SIGMA = 26, MAXTRIE = MAXP * MAXLEN;
2
3
4  int npatterns;
5  string patterns[MAXP], S;
6
7  int wordIdx[MAXTRIE], to[MAXTRIE][SIGMA], sLink[MAXTRIE], dLink[MAXTRIE], nnodes;
8
9  void ahoCorasick()
10 {
11     // 1. Make a tree, 2. create sLinks and dLinks, 3. Walk through S
12
13     fill_n(wordIdx, MAXTRIE, -1);
14     for (int i = 0; i < MAXTRIE; i++) fill_n(to[i], SIGMA, 0);
15     fill_n(sLink, MAXTRIE, 0);
16     fill_n(dLink, MAXTRIE, 0);
17     nnodes = 1;
18
19     for (int i = 0; i < npatterns; i++) {
20         int cur = 0;
21         for (char c : patterns[i]) {
22             int idx = c - 'a';
23             if (to[cur][idx] == 0) to[cur][idx] = nnodes++;
24             cur = to[cur][idx];
25         }
26         wordIdx[cur] = i;
27     }
28
29     queue<int> q;
30     q.push(0);
31     while (!q.empty()) {
32         int cur = q.front(); q.pop();
33         for (int c = 0; c < SIGMA; c++) {
34             if (to[cur][c]) {
35                 int sl = sLink[to[cur][c]] = cur == 0 ? 0 : to[sLink[cur]][c];
36                 // if all strings have equal length, remove this:
37                 dLink[to[cur][c]] = wordIdx[sl] >= 0 ? sl : dLink[sl];
38                 q.push(to[cur][c]);
39             } else to[cur][c] = to[sLink[cur]][c];
40         }
41     }
42
43     for (int cur = 0, i = 0, n = S.size(); i < n; i++) {
44         int idx = S[i] - 'a';
45         cur = to[cur][idx];
46         for (int hit = wordIdx[cur] >= 0 ? cur : dLink[cur]; hit; hit = dLink[hit]) {
47             cerr << "Match for " << patterns[wordIdx[hit]] << " at " << (i + 1 - patterns[
48                 wordIdx[hit]].size()) << endl;
49         }
50     }
51 }

```

---

## 5 Geometry

---

```

1  const double EPS = 1e-7;
2
3  #define x first
4  #define y second
5
6  typedef double NUM; // EITHER double OR long long
7  typedef pair<NUM, NUM> pt;

```

```

8
9 pt operator+(pt p, pt q) { return pt(p.x + q.x, p.y + q.y); }
10 pt operator-(pt p, pt q) { return pt(p.x - q.x, p.y - q.y); }
11
12 pt& operator+=(pt &p, pt q) { return p = p + q; }
13 pt& operator-=(pt &p, pt q) { return p = p - q; }
14
15 pt operator*(pt p, NUM l) { return pt(p.x * l, p.y * l); }
16 pt operator/(pt p, NUM l) { return pt(p.x / l, p.y / l); }
17
18 NUM operator*(pt p, pt q) { return p.x * q.x + p.y * q.y; }
19 NUM operator^(pt p, pt q) { return p.x * q.y - p.y * q.x; }
20
21 istream& operator>>(istream &in, pt &p) { return in >> p.x >> p.y; }
22 ostream& operator<<(ostream &out, pt p) { return out << '(' << p.x << ", " << p.y << ')'; }
23
24 NUM lenSq(pt p) { return p * p; }
25 NUM lenSq(pt p, pt q) { return lenSq(p - q); }
26 double len(pt p) { return hypot(p.x, p.y); } // more overflow safe
27 double len(pt p, pt q) { return len(p - q); }
28
29 // square distance from pt a to line bc
30 double distPtLineSq(pt a, pt b, pt c) {
31     a -= b, c -= b;
32     return (a ^ c) * (a ^ c) / (double)(c * c);
33 }
34
35 // square distance from pt a to segment bc
36 double distPtSegmentSq(pt a, pt b, pt c) {
37     a -= b; c -= b;
38     NUM dot = a * c, len = c * c;
39     if (dot <= 0) return a * a;
40     if (dot >= len) return (a - c) * (a - c);
41     return a * a - dot * dot / ((double) len);
42     // pt proj = b + c * dot / ((double) len);
43 }
44
45 bool between(NUM x, NUM a, NUM b) { return min(a, b) <= x && x <= max(a, b); }
46 bool collinear(pt a, pt b, pt c) { return ((a - b) ^ (a - c)) == 0; }
47
48 // point a on segment bc
49 bool pointOnSegment(pt a, pt b, pt c) {
50     return collinear(a, b, c) && between(a.x, b.x, c.x) && between(a.y, b.y, c.y);
51 }
52
53 // REQUIRES DOUBLES
54 pt lineLineIntersection(pt a, pt b, pt c, pt d, bool &cross)
55 {
56     NUM det = (a - b) ^ (c - d);
57     pt ret = (c - d) * (a ^ b) - (a - b) * (c ^ d);
58     return (cross = det != 0) ? (ret / det) : ret;
59 }
60
61 // REQUIRES DOUBLES
62 // Line segment a1 -- a2 intersects with b1 -- b2?
63 // returns 0: no, 1: yes at i1, 2: yes at i1 -- i2
64 int segmentsIntersect(pt a1, pt a2, pt b1, pt b2, pt &i1, pt &i2) {
65     if (((a2 - a1) ^ (b2 - b1)) < 0) swap(a1, a2);
66     // assert(a1 != a2 && b1 != b2);
67     pt q = a2 - a1, r = b2 - b1, s = b1 - a1;
68     NUM cross = q ^ r, c1 = s ^ r, c2 = s ^ q;
69     if (cross == 0) {
70         // line segments are parallel
71         if ((q ^ s) != 0) return 0; // no intersection
72         NUM v1 = s * q, v2 = (b2 - a1) * q, v3 = q * q;
73         if (v2 < v1) swap(v1, v2), swap(b1, b2);
74
75         if (v1 > v3 || v2 < 0) return 0; // intersection empty

```

```

76     i1 = v2 > v3 ? a2 : b2;
77     i2 = v1 < 0 ? a1 : b1;
78     return i1 == i2 ? 1 : 2; // one point or overlapping
79 } else { // cross > 0
80     i1 = pt(a1) + pt(q) * (1.0 * c1 / cross); // needs double
81     return 0 <= c1 && c1 <= cross && 0 <= c2 && c2 <= cross;
82     // intersection inside segments
83 }
84 }
85
86 // REQUIRES DOUBLES
87 // TODO: Needs shortening
88 // complete intersection check
89 int segmentsIntersect2(pt a1, pt a2, pt b1, pt b2, pt &i1, pt &i2) {
90     if (a1 == a2 && b1 == b2) {
91         i1 = a1;
92         return a1 == b1;
93     } else if (a1 == a2) {
94         i1 = a1;
95         return pointOnSegment(a1, b1, b2);
96     } else if (b1 == b2) {
97         i1 = b1;
98         return pointOnSegment(b1, a1, a2);
99     } else return segmentsIntersect(a1, a2, b1, b2, i1, i2);
100 }
101
102 // Returns TWICE the area of a polygon to keep it an integer
103 NUM polygonTwiceArea(const vector<pt> &pts) {
104     NUM area = 0;
105     for (int N = pts.size(), i = 0, j = N - 1; i < N; j = i++)
106         area += pts[i] ^ pts[j];
107     return abs(area); // area < 0 ==> pts ccw
108 }
109
110 bool pointInPolygon(pt p, const vector<pt> &pts)
111 {
112     double sum = 0;
113     for (int N = pts.size(), i = 0, j = N - 1; i < N; j = i++) {
114         if (pointOnSegment(p, pts[i], pts[j])) return true; // boundary
115         double angle = acos((pts[i] - p) * (pts[j] - p) / len(pts[i], p) / len(pts[j], p));
116         sum += ((pts[i] - p) ^ (pts[j] - p)) < 0 ? angle : -angle;
117     }
118     return abs(abs(sum) - 2 * PI) < EPS;
119 }

```

---

## 5.1 Convex Hull $\mathcal{O}(n \log n)$

```

1 // points are given by: pts[ret[0]], pts[ret[1]], ... pts[ret[ret.size()-1]]
2 vector<int> convexHull(const vector<pt> &pts) {
3     if (pts.empty()) return vector<int>();
4     vector<int> ret;
5     int bestIndex = 0, n = pts.size();
6     pt best = pts[0];
7     for(int i = n; i--; ) {
8         if (pts[i] < best) {
9             best = pts[bestIndex = i];
10        }
11    }
12    ret.push_back(bestIndex);
13    pt refr = pts[bestIndex];
14
15    vector<int> ordered; // index into pts
16    for (int i = n; i--; ) {
17        if (pts[i] != refr) ordered.push_back(i);
18    }
19    sort(ordered.begin(), ordered.end(), [&pts, &refr] (int a, int b) -> bool {

```

---

```

20     NUM cross = (pts[a] - refr) ^ (pts[b] - refr);
21     return cross != 0 ? cross > 0 : lenSq(refr, pts[a]) < lenSq(refr, pts[b]);
22 });
23 for (int i : ordered) {
24     // NOTE: > INCLUDES points on the hull-line, >= EXCLUDES
25     while (ret.size() > 1 && ((pts[ret[ret.size() - 2]] - pts[ret.back()]) ^ (pts[i] - pts
26         [ret.back()]))) >= 0) {
27         ret.pop_back();
28     }
29     ret.push_back(i);
30 }
31 return ret;
32 }

```

---

## 5.2 Rotating Calipers $\mathcal{O}(n)$

Finds the longest distance between two points in a convex hull.

---

```

1 NUM rotatingCalipers(vector<pt> &hull) {
2     int n = hull.size(), a = 0, b = 1;
3     if (n <= 1) return 0.0;
4     while (((hull[1] - hull[0]) ^ (hull[(b + 1) % n] - hull[b])) > 0) b++;
5     cerr << a << " " << b << endl;
6     NUM ret = 0.0;
7     while (a < n) {
8         ret = max(ret, lenSq(hull[a], hull[b]));
9         if (((hull[(a + 1) % n] - hull[a % n]) ^ (hull[(b + 1) % n] - hull[b])) <= 0) a++;
10        else if (++b == n) b = 0;
11    }
12    return ret;
13 }

```

---

## 6 Miscellaneous

### 6.1 Binary search $\mathcal{O}(\log(hi - lo))$

---

```

1 bool test(int n);
2
3 int search(int lo, int hi) {
4     // assert(test(lo) && !test(hi));
5     while (hi - lo > 1) {
6         int c = (lo + hi) / 2;
7         if (test(c)) lo = c;
8         else hi = c;
9     }
10    // assert(test(lo) && !test(hi));
11    return lo;
12 }

```

---

### 6.2 Fast Fourier Transform $\mathcal{O}(n \log n)$

Given two polynomials  $A(x) = a_0 + a_1x + \dots + a_{n/2}x^{n/2}$  and  $B(x) = b_0 + b_1x + \dots + b_{n/2}x^{n/2}$ , FFT calculates all coefficients of  $C(x) = A(x) \cdot B(x) = c_0 + c_1x + \dots + c_nx^n$ , with  $c_i = \sum_{j=0}^i a_j b_{i-j}$ .

---

```

1
2 typedef complex<double> cpx;
3 const int logmaxn = 20, maxn = 1 << logmaxn;
4
5 cpx a[maxn] = {}, b[maxn] = {}, c[maxn];
6
7 void fft(cpx *src, cpx *dest)

```



```

8 {
9     for (int i = 0, rep = 0; i < maxn; i++, rep = 0) {
10         for (int j = i, k = logmaxn; k-- >= 1; rep = (rep << 1) | (j & 1);
11             dest[rep] = src[i];
12         }
13         for (int s = 1, m = 1; m <= maxn; s++, m *= 2) {
14             cpx r = exp(cpx(0, 2.0 * PI / m));
15             for (int k = 0; k < maxn; k += m) {
16                 cpx cr(1.0, 0.0);
17                 for (int j = 0; j < m / 2; j++) {
18                     NUM t = cr * dest[k + j + m / 2];
19                     dest[k + j + m / 2] = dest[k + j] - t;
20                     dest[k + j] += t;
21                     cr *= r;
22                 }
23             }
24         }
25     }
26
27 void multiply()
28 {
29     fft(a, c);
30     fft(b, a);
31     for (int i = 0; i < maxn; i++) b[i] = conj(a[i] * c[i]);
32     fft(b, c);
33     for (int i = 0; i < maxn; i++) c[i] = conj(c[i]) / (1.0 * maxn);
34 }

```

---

### 6.3 Minimum Assignment (Hungarian Algorithm) $\mathcal{O}(n^3)$

```

1 int a[MAXN + 1][MAXM + 1]; // matrix, 1-based
2
3 int minimum_assignment(int n, int m) { // n rows, m columns
4     vector<int> u(n + 1), v(m + 1), p(m + 1), way(m + 1);
5
6     for (int i = 1; i <= n; i++) {
7         p[0] = i;
8         int j0 = 0;
9         vector<int> minv(m + 1, INF);
10        vector<char> used(m + 1, false);
11        do {
12            used[j0] = true;
13            int i0 = p[j0], delta = INF, j1;
14            for (int j = 1; j <= m; j++)
15                if (!used[j]) {
16                    int cur = a[i0][j] - u[i0] - v[j];
17                    if (cur < minv[j]) minv[j] = cur, way[j] = j0;
18                    if (minv[j] < delta) delta = minv[j], j1 = j;
19                }
20            for (int j = 0; j <= m; j++) {
21                if (used[j]) u[p[j]] += delta, v[j] -= delta;
22                else minv[j] -= delta;
23            }
24            j0 = j1;
25        } while (p[j0] != 0);
26        do {
27            int j1 = way[j0];
28            p[j0] = p[j1];
29            j0 = j1;
30        } while (j0);
31    }
32
33    // column j is assigned to row p[j]
34    // for (int j = 1; j <= m; ++j) ans[p[j]] = j;
35    return -v[0];
36 }

```

---