

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

0.1	Covering problems . . . . .	2
<b>1</b>	<b>Mathematics</b>	<b>2</b>
<b>2</b>	<b>Datastructures</b>	<b>4</b>
2.1	Segment tree $\mathcal{O}(\log n)$ . . . . .	4
2.2	Binary Indexed Tree $\mathcal{O}(\log n)$ . . . . .	4
2.3	Trie . . . . .	5
2.4	Disjoint-Set / Union-Find $\mathcal{O}(\alpha(n))$ . . . . .	5
<b>3</b>	<b>Graph Algorithms</b>	<b>6</b>
3.1	Maximum matching $\mathcal{O}(nm)$ . . . . .	6
3.2	Strongly Connected Components $\mathcal{O}(V + E)$ . . . . .	6
3.3	Cycle Detection $\mathcal{O}(V + E)$ . . . . .	7
3.4	Shortest path . . . . .	8
3.4.1	BFS $\mathcal{O}(V + E)$ . . . . .	8
3.4.2	Dijkstra $\mathcal{O}(E + V \log V)$ . . . . .	8
3.4.3	Floyd-Warshall $\mathcal{O}(V^3)$ . . . . .	9
3.4.4	Bellman Ford $\mathcal{O}(VE)$ . . . . .	9
3.5	Max-flow min-cut . . . . .	9
3.5.1	Dinic's Algorithm $\mathcal{O}(V^2E)$ . . . . .	9
3.6	Min-cost max-flow . . . . .	11
3.7	Minimal Spanning Tree . . . . .	12
3.7.1	Prim $\mathcal{O}((E + V) \log V)$ . . . . .	12
3.7.2	Kruskal $\mathcal{O}(E \log V)$ . . . . .	12
<b>4</b>	<b>String algorithms</b>	<b>13</b>
4.1	Z-algorithm $\mathcal{O}(n)$ . . . . .	13
4.2	Longest Common Subsequence $\mathcal{O}(n^2)$ . . . . .	13
4.3	Levenshtein Distance $\mathcal{O}(n^2)$ . . . . .	14
4.4	Knuth-Morris-Pratt algorithm $\mathcal{O}(N + M)$ . . . . .	14
<b>5</b>	<b>Geometry</b>	<b>15</b>
5.1	Convex Hull $\mathcal{O}(n \log n)$ . . . . .	17
<b>6</b>	<b>Miscellaneous</b>	<b>18</b>
6.1	Binary search $\mathcal{O}(\log n)$ . . . . .	18
6.2	Minimum Assignment (Hungarian Algorithm) $\mathcal{O}(n^3)$ . . . . .	18

---

```

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;
14 const ll LLINF = 9223372036854775807LL;
15
16 // lambda-expression: [] (args) -> retType { body }
17
18 void run() {}
19
20 int main() {
21     ios_base::sync_with_stdio(false);
22     cin.tie(NULL);
23     cout.tie(NULL);
24     // cerr << boolalpha;
25     (cout << fixed).precision(10);
26     int ntc;
27     cin >> ntc;
28     while (ntc--> 0) { run(); }
29     return 0;
30 }

```

---

Two prime numbers: 982451653, 81253449

## 0.1 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

## 1 Mathematics

---

```

1
2 int abs(int x) { return x > 0 ? x : -x; }
3 int sign(int x) { return (x > 0) - (x < 0); }
4
5 // greatest common divisor
6 ll gcd(ll a, ll b) {
7     while (b) {
8         ll c = a % b;
9         a = b; b = c;
10    }
11    return a;
12 }
13
14 // least common multiple
15 ll lcm(ll a, ll b) { return a / gcd(a, b) * b; }
16
17 // ax + by = gcd(a, b)
18 ll egcd(ll a, ll b, ll &x, ll &y) {
19     ll r = b, rr = a, s = 0, ss = 1, t = 1, tt = 0, tmp;
20     while (r) {
21         ll q = rr / r;
22         tmp = r; r = rr - q * r; rr = tmp;
23         tmp = s; s = ss - q * s; ss = tmp;
24         tmp = t; t = tt - q * t; tt = tmp;
25     }
26     x = ss; y = tt;
27     return rr; // gcd
28 }
29
30 ll mod(ll a, ll m) {
31     a %= m;
32     return (a < 0) ? a + m : a;
33 }
34
35 const pll INVALID_CRT(0, -1);
36
37 // x = a0 mod b0 = a1 mod b1
38 pll crt(ll a1, ll m1, ll a2, ll m2) {
39     ll x, y, gcd = egcd(m1, m2, x, y);
40     if (a1 % gcd != a2 % gcd) return INVALID_CRT;
41     return pll(mod(x * a2 * m1 + y * a1 * m2, m1 * m2) / gcd, m1 / gcd * m2);
42 }
43
44 // totient function for values 1 to N
45 int phi[N];
46
47 void sievePhi() {
48     for (int i = N; i--;) phi[i] = i;
49     for (int i = 2; i <= N; i++)
50         if (phi[i] == i)
51             for (int j = i; j <= N; j += i)
52                 phi[j] -= phi[j] / i * (i - 1);
53 }
54
55 // modular exponentiation: r = b^e mod m
56 ll modpow(ll b, ll e, ll m) {
57     ll r = 1;
58     while (e) {

```

---

```

59         if (e & 1) r = (r * b) % m;
60         e >>= 1;
61         b = (b * b) % m;
62     }
63     return r;
64 }

```

---

## 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 $\mathcal{O}(\log n)$

---

```

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 struct trie {
2     bool word;
3     trie **child;
4
5     trie() : word(false), child() {
6         child = new trie*[26];
7         for (int i = 26; i--;) child[i] = NULL;
8     }
9
10    void addWord(const string &str)
11    {
12        trie *cur = this;
13        for (char ch : str) {
14            int idx = ch - 'a';
15            if (cur->child[idx] == NULL)
16                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] == NULL) 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     int pa = uf_find(a), ra = rnk[pa];

```

---

```

14     int pb = uf_find(b), rb = rnk[pb];
15     if (pa == pb) return;
16
17     if (ra > rb) {
18         par[pb] = pa;
19     } else if (rb > ra) {
20         par[pa] = pb;
21     } else {
22         par[pa] = pb;
23         rnk[pb]++;
24     }
25 }

```

---

## 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 bool vis[nodesRight]; // vis[rightnodes]
2 int par[nodesRight]; // par[rightnode] = leftnode
3 vector<int> adj[nodesLeft]; // adj[leftnode][i] = rightnode
4
5 bool match(int cur) {
6     for (int nxt : adj[cur]) {
7         if (vis[nxt]) continue;
8         vis[nxt] = true;
9         if (par[nxt] == -1 || match(par[nxt])) {
10             par[nxt] = cur;
11             return true;
12         }
13     }
14     return false;
15 }
16
17 int matches = 0;
18 for (int i = 0; i < nodesLeft; i++) {
19     memset(vis, false, sizeof(vis));
20     if (match(i)) matches++;
21 }

```

---

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

---

```

1 vector<vi> adj; // adjacency matrix
2 vi index, lowlink; // lowest index reachable
3 stack<int> tarjanStack;
4 vector<bool> inStack; // true iff in tarjanStack
5 int newId;
6 vector<vi> scc; // Output: collection of vertex sets
7
8 void tarjan(int v) {
9     index[v] = lowlink[v] = newId++;

```

---

```

10     tarjanStack.push(v);
11     inStack[v] = true;
12     for (int w : adj[v]) {
13         if (index[w] == 0) {
14             tarjan(w);
15             lowlink[v] = min(lowlink[v], lowlink[w]);
16         } else if (inStack[w]) {
17             lowlink[v] = min(lowlink[v], index[w]);
18         }
19     }
20
21     if (lowlink[v] == index[v]) {
22         scc.push_back(v);
23         int w;
24         do {
25             w = tarjanStack.top();
26             scc.back().push_back(w);
27             inStack[w] = false;
28             tarjanStack.pop();
29         } while (w != v);
30     }
31 }
32
33 int findSCC() {
34     newId = 1;
35     index.clear(); index.resize(n + 1, 0);
36     lowlink.clear(); lowlink.resize(n + 1, 0);
37     inStack.clear(); inStack.resize(n + 1, false);
38     while (!tarjanStack.empty()) tarjanStack.pop();
39     scc.clear();
40
41     for (int i = 0; i < n; i++) {
42         if (index[i] == 0) tarjan(i);
43     }
44     return scc.size();
45 }

```

---

### 3.3 Cycle Detection $\mathcal{O}(V + E)$

---

```

1 vector<vi> adj; // assumes bidirected graph, adjust accordingly
2 vector<bool> vis(MAXN, false);
3 vector<int> par(MAXN, -1);
4
5 bool cycle_detection() {
6     stack<int> s;
7     s.push(0);
8     vis[0] = true;
9     while(!s.empty()) {
10         int cur = s.top();
11         s.pop();
12         for(int i : adj[cur]) {
13             if(vis[i] && par[cur] != i) return true;
14             s.push(i);
15             par[i] = cur;
16             vis[i] = true;
17         }

```

---

```

18     }
19     return false;
20 }

```

---

## 3.4 Shortest path

### 3.4.1 BFS $\mathcal{O}(V + E)$

---

```

1 int n, dist[MAXN];
2 vector<int> edges[MAXN]; // (to, cost)
3
4 // faster than dijkstra when all edge costs are the same
5 int bfs(int from, int to) {
6     fill_n(dist, n, -1);
7     dist[from] = 0;
8
9     queue<int> q;
10    q.push(from);
11    while (!q.empty()) {
12        int cur = q.front();
13        q.pop();
14        for (int nxt : edges[cur]) {
15            if (dist[nxt] >= 0) {
16                dist[nxt] = dist[cur] + 1;
17                if (nxt == to) return dist[nxt];
18            }
19            q.push(nxt);
20        }
21    }
22    return -1;
23 }

```

---

### 3.4.2 Dijkstra $\mathcal{O}(E + V \log V)$

---

```

1 int n; // number of nodes
2
3 vector<pii> edges[MAXN]; // (to, cost)
4 int dist[MAXN];
5 bool vis[MAXN];
6
7 void dijkstra() {
8     fill_n(vis, n, false);
9     priority_queue<pii, vector<pii>, greater<pii> > q; // (dist, id)
10    q.push(pii(0, 0));
11
12    while (!q.empty()) {
13        pii v = q.top();
14        q.pop();
15
16        if (vis[v.second]) continue;
17        vis[v.second] = true;
18
19        for (const pii e : edges[v.second]) {
20            q.push(pii(v.first + e.second, e.first));
21        }
22    }
23 }

```

---



---

```

21 }
22 dist[v.second] = v.first;
23 }
24 }

```

---

### 3.4.3 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]);

```

---

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

---

```

1 vector< pair<pii,int> > edges; // ((from, to), cost)
2 vector<int> dist (MAXN);
3
4 bool bellman_ford(int source) {
5     for (int i = 0; i < MAXN; i++) dist[i] = INF / 3;
6     dist[source] = 0;
7
8     bool updated;
9     int loops = 0;
10    do {
11        updated = false;
12        for (auto e : edges) {
13            int alt = dist[e.first.first] + e.second;
14            if (alt < dist[e.first.second]) {
15                dist[e.first.second] = alt;
16                updated = true;
17            }
18            // if undirected graph:
19            int alt = dist[e.first.second] + e.second;
20            if (UNDIRECTED && alt < dist[e.first.first]) {
21                dist[e.first.first] = alt;
22                updated = true;
23            }
24        }
25    } while(updated && loops < n);
26    return loops < n; // loops >= n: negative cycles
27 }

```

---

## 3.5 Max-flow min-cut

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

---

```

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] == 0 && e.flow < e.cap) {
22                 level[e.to] = level[cur] + 1;
23                 q.push(e.to);
24             }
25         }
26     }
27     return level[t] != 0;
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 }

```

---

### 3.6 Min-cost max-flow

---

```

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().first, v = q.top().second;
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 }

```

---

## 3.7 Minimal Spanning Tree

### 3.7.1 Prim $\mathcal{O}((E + V) \log V)$

---

```

1 // minimum spanning forest actually...
2 vector<pii> edges[MAXN]; // or set
3 int dist[MAXN];
4 bool done[MAXN];
5
6 ll prim(int n) {
7     fill_n(dist, n, INF);
8     fill_n(done, n, false);
9     ll ret = 0, trees = 0;
10    set<pii> q; // (to MST, vertex)
11    for (int i = 0; i < n; i++) {
12        if (done[i]) continue;
13        trees++;
14        q.insert(pii(dist[i] = 0, i));
15        while (!q.empty()) {
16            ret += q.begin()->first;
17            int cur = q.begin()->second;
18            q.erase(q.begin());
19            done[cur] = true;
20            for (pii pr : edges[cur]) {
21                if (!done[pr.first] && pr.second < dist[pr.first]) {
22                    q.erase(pii(dist[pr.first], pr.first));
23                    dist[pr.first] = pr.second;
24                    q.insert(pii(dist[pr.first], pr.first));
25                }
26            }
27        }
28    }
29    // if (trees > 1) return -1; // forest
30    return ret;
31 }

```

---

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

---

```

1 struct edge {
2     int x, y, s;
3     void read() { cin >> x >> y >> s; }
4 };
5
6 edge edges[MAXM];
7

```

---

```

8 int kruskal(int n, int m) {
9     uf_init(n);
10    sort(edges, edges + m, [] (const edge &a, const edge &b)
11        -> bool { return a.s > b.s; });
12    ll ret = 0;
13    while (m--) {
14        if (uf_find(edges[m].x) != uf_find(edges[m].y)) {
15            ret += edges[m].s;
16            uf_union(edges[m].x, edges[m].y);
17        }
18    }
19    return ret;
20 }

```

---

## 4 String algorithms

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

---

```

1 // z[i] = length of longest substring starting from s[i],
2 // which is also a prefix of s.
3 vector<int> z_function(const string &s) {
4     int n = (int) s.length();
5     vector<int> z(n);
6     for (int i = 1, l = 0, r = 0; i < n; ++i) {
7         if (i <= r)
8             z[i] = min(r - i + 1, z[i - l]);
9         while (i + z[i] < n && s[z[i]] == s[i + z[i]])
10             ++z[i];
11         if (i + z[i] - 1 > r)
12             l = i, r = i + z[i] - 1;
13     }
14     return z;
15 }

```

---

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

SUBSTRING: *consecutive characters* !!!

---

```

1 int table[STR_SIZE][STR_SIZE];
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 // backtrace
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 }

```

---

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

---

```

1 int costs[MAX_SIZE][MAX_SIZE];
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 }

```

---

### 4.4 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]) {
6             table[++i] = ++j;
7         } else if (j > 0) {
8             j = table[j];
9         } else i++;
10    }
11    int matches = 0;
12    for (int i = 0, j = 0; i < text.size(); ) {
13        if (text[i] == word[j]) {
14            i++;
15            if (++j == n) {
16                matches++;
17                // match at interval [i - j, i)
18                j = table[j];
19            }

```

```

20         } else if (j > 0) j = table[j];
21         else i++;
22     }
23     return matches;
24 }

```

---

## 5 Geometry

```

1  typedef double NUM; // either double or long long
2
3  struct pt {
4      NUM x, y;
5
6      pt() : x(0), y(0) {}
7      pt(NUM _x, NUM _y) : x(_x), y(_y) {}
8      pt(const pt &p) : x(p.x), y(p.y) {}
9
10     pt operator*(NUM scalar) const {
11         return pt(scalar * x, scalar * y); // scalar
12     }
13     NUM operator*(const pt &rhs) const {
14         return x * rhs.x + y * rhs.y; // dot product
15     }
16     NUM operator^(const pt &rhs) const {
17         return x * rhs.y - y * rhs.x; // cross product
18     }
19     pt operator+(const pt &rhs) const {
20         return pt(x + rhs.x, y + rhs.y); // addition
21     }
22     pt operator-(const pt &rhs) const {
23         return pt(x - rhs.x, y - rhs.y); // subtraction
24     }
25     bool operator==(const pt &rhs) const {
26         return x == rhs.x && y == rhs.y;
27     }
28     bool operator!=(const pt &rhs) const {
29         return x != rhs.x || y != rhs.y;
30     }
31 };
32
33 // distance SQUARED from pt a to pt b
34 NUM sqDist(const pt &a, const pt &b) {
35     return (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y);
36 }
37
38 // distance SQUARED from pt a to line bc
39 double sqDistPointLine(pt a, pt b, pt c) {
40     NUM a_x = a.x, a_y = a.y;
41     NUM b_x = b.x, b_y = b.y;
42     NUM c_x = c.x, c_y = c.y;
43     NUM dx = c_x - b_x, dy = c_y - b_y;
44     NUM dxdx = dx * dx, dxdy = dx * dy, dydy = dy * dy;
45     NUM t = (a_x - b_x) * dx + (a_y - b_y) * dy;
46     if (t < 0) t = 0;
47     if (t > dxdx + dydy) t = dxdx + dydy;
48     NUM x = b_x + dx * t / (dxdx + dydy);
49     NUM y = b_y + dy * t / (dxdx + dydy);
50     return (a_x - x) * (a_x - x) + (a_y - y) * (a_y - y);
51 }

```

```

48     c = c - b;
49     NUM dot = a * c, len = c * c;
50     if (dot <= 0) return a * a;
51     if (dot >= len) return (a - c) * (a - c);
52     return a * a - dot * dot / ((double) len);
53     // pt proj = c * dot / ((double) len);
54 }
55
56 bool between(NUM a, NUM b, NUM n) {
57     return min(a, b) <= n && n <= max(a, b);
58 }
59 bool collinear(pt a, pt b, pt c) {
60     return ((a - b) ^ (a - c)) == 0;
61 }
62
63 // point a on segment bc
64 bool pointOnSegment(pt a, pt b, pt c)
65 {
66     return collinear(a, b, c) &&
67         between(b.x, c.x, a.x) && between(b.y, c.y, a.y);
68 }
69
70 // Line segment a1 -- a2 intersects with b1 -- b2?
71 // returns 0: no, 1: yes at i1, 2: yes at i1 -- i2
72 int segmentsIntersect(pt a1, pt a2, pt b1, pt b2, pt &i1, pt &i2) {
73     if ((a2 - a1) ^ (b2 - b1) < 0) swap(a1, a2);
74     // assert(a1 != a2 && b1 != b2);
75     pt q = a2 - a1, r = b2 - b1, s = b1 - a1;
76     NUM cross = q ^ r, c1 = s ^ r, c2 = s ^ q;
77     if (cross == 0) {
78         // line segments are parallel
79         if ((q ^ s) != 0) return 0; // no intersection
80         NUM v1 = s * q, v2 = (b2 - a1) * q, v3 = q * q;
81         if (v2 < v1) swap(v1, v2), swap(b1, b2);
82
83         if (v1 > v3 || v2 < 0) return 0; // intersection empty
84         i1 = v2 > v3 ? a2 : b2;
85         i2 = v1 < 0 ? a1 : b1;
86         return i1 == i2 ? 1 : 2; // one point or overlapping
87     } else { // cross > 0
88         i1 = pt(a1) + pt(q) * (1.0 * c1 / cross); // needs double
89         return 0 <= c1 && c1 <= cross && 0 <= c2 && c2 <= cross;
90         // intersection inside segments
91     }
92 }
93
94 // complete intersection check
95 int segmentsIntersect2(pt a1, pt a2, pt b1, pt b2, pt &i1, pt &i2) {
96     if (a1 == a2 && b1 == b2) {
97         i1 = a1;
98         return a1 == b1;
99     } else if (a1 == a2) {
100         i1 = a1;
101         return pointOnSegment(a1, b1, b2);
102     } else if (b1 == b2) {
103         i1 = b1;
104         return pointOnSegment(b1, a1, a2);
105     } else return segmentsIntersect(a1, a2, b1, b2, i1, i2);

```



```

106 }
107
108 // Returns TWICE the area of a polygon to keep it an integer
109 NUM polygonTwiceArea(const vector<pt> &polygon) {
110     NUM area = 0;
111     for (int i = 0, N = polygon.size(), j = N - 1; i < N; j = i++)
112         area += polygon[i] ^ polygon[j];
113     return abs(area);
114 }
115
116 // returns 0 outside, 1 inside, 2 on boundary
117 int pointInPolygon(pt p, const vector<pt> &polygon) {
118     // Check crossings with horizontal semi-line through p to +x
119     int crosscount = 0, N = polygon.size();
120     for (int i = 0, j = N - 1; i < N; j = i++) {
121         if (pointOnSegment(p, polygon[i], polygon[j])) return 2;
122
123         // check if it crosses the vertical y = p.y line
124         NUM l = (p.x - polygon[i].x)*(polygon[j].y - polygon[i].y);
125         NUM r = (p.y - polygon[i].y)*(polygon[j].x - polygon[i].x);
126         if (polygon[j].y > p.y) {
127             if (polygon[i].y <= p.y && l < r) crosscount++;
128         } else {
129             if (polygon[i].y <= p.y && l > r) crosscount++;
130         }
131     }
132     return crosscount & 1;
133 }
134
135 // Assumption: polygon has unique points
136 int pointInConvex(pt p, const vector<pt> &polygon) {
137     // the cross product should always have the same sign,
138     // when the point is inside the convex
139     int N = polygon.size(), sgn = 0;
140     bool onBoundary = false;
141     for (int i = 0, j = N - 1; i < N; j = i++) {
142         NUM cross = (polygon[j] - p) ^ (polygon[i] - p);
143         if (cross == 0) onBoundary = true;
144         else if (sgn == 0) sgn = sign(cross);
145         else if (sgn != sign(cross)) return 0;
146     }
147     return onBoundary ? 2 : 1;
148 }

```

---

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

```

1 // output contains indices of the points on the hull
2 void convex_hull(const vector<pt> &pts, vector<int> &output) {
3     output.clear();
4     if (pts.size() < 3) {
5         if (pts.size() >= 1) output.push_back(0);
6         if (pts.size() >= 2) output.push_back(1);
7         return;
8     }
9
10    unsigned int bestIndex = 0;

```

---

```

11     NUM minX = pts[0].x, minY = pts[0].y;
12     for(unsigned int i = 1; i < pts.size(); ++i) {
13         if (pts[i].x < minX || (pts[i].x == minX && pts[i].y < minY)) {
14             bestIndex = i;
15             minX = pts[i].x;
16             minY = pts[i].y;
17         }
18     }
19     vector<int> ordered; //index into pts
20     for(unsigned int i = 0; i < pts.size(); ++i) {
21         if (i != bestIndex) ordered.push_back(i);
22     }
23
24     pt refr = pts[bestIndex];
25     sort(ordered.begin(), ordered.end(), [&pts,&refr] (int a, int b) -> bool {
26         NUM cross = (pts[a] - refr) ^ (pts[b] - refr);
27         return cross != 0 ? cross > 0 : sqDist(refr, pts[a]) < sqDist(refr, pts[b]);
28     });
29
30     output.push_back(bestIndex);
31     output.push_back(ordered[0]);
32     output.push_back(ordered[1]);
33     for(unsigned int i = 2; i < ordered.size(); ++i) {
34         //NOTE: > INCLUDES and >= EXCLUDES points on the hull-line
35         while (output.size() > 1 && ((pts[output[output.size() - 2]] - pts[output.back()]) ^ (pts[ordered[i]] - pts[output.back()])) > 0) {
36             output.pop_back();
37         }
38         output.push_back(ordered[i]);
39     }
40     return;
41 }

```

---

## 6 Miscellaneous

### 6.1 Binary search $\mathcal{O}(\log n)$

Inclusive, Exclusive	Inclusive, Inclusive
<pre> 1 bool test(int n); 2 3 int min = 0, max = n; 4 // assert(test(min) &amp;&amp; !test(max)); 5 while (max - min &gt; 1) { 6     int c = (min + max) / 2; 7     if (test(c)) min = c; 8     else max = c; 9 } 10 return min; </pre>	<pre> 1 bool test(int n); 2 3 int lo = 0, hi = n - 1; 4 // assert(test(lo) &amp;&amp; !test(hi + 1)); 5 while (lo &lt; hi) { 6     int mid = (lo + hi + 1) / 2; 7     if (test(mid)) lo = mid; 8     else hi = mid - 1; 9 } 10 return lo; </pre>

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

---

```

1 int n, m; // n rows, m columns

```

```
2 int a[MAXN + 1][MAXM + 1]; // matrix, 1-based
3 int minimum_assignment() {
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    // for (int j = 1; j <= m; ++ j) ans[p[j]] = j;
34    return -v[0];
35 }
```

---