Codebook

Pitoni++

Žiga Gosar, Maks Kolman, Jure Slak

- podrobno in pozorno preberi navodila
- pazi na `double` in `unsigned long long`
- počisti podatke med testnimi primeri

verzija: 15. april 2015

Kazalo

1 Grafi

1.1 Topološko sortiranje

Vhod: Usmerjen graf G brez ciklov. G ne sme imeti zank, če pa jih ima, se jih lahko brez škode odstrani.

Izhod: Topološka ureditev usmerjenega grafa G, to je seznam vozlišč v takem vrstnem redu, da nobena povezava ne kaže nazaj. Če je vrnjeni seznam krajši od n, potem ima G cikle.

Časovna zahtevnost: $O(V + E)$

Prostorska zahtevnost: $O(V + E)$

Testiranje na terenu: UVa 10305

```
vector<int> topological_sort(const vector<vector<int>>& graf) {
    int n = graf.size();
    vector<int> ingoing(n, 0);
    for (int i = 0; i < n; ++i)
        for (const auto& u : graf[i])
            ingoing[u]++;

    queue<int> q;  // morda priority_queue, če je vrstni red pomemben
    for (int i = 0; i < n; ++i)
        if (ingoing[i] == 0)
            q.push(i);

    vector<int> res;
    while (!q.empty()) {
        int t = q.front();
        q.pop();

        res.push_back(t);

        for (int v : graf[t])
            if (--ingoing[v] == 0)
                q.push(v);
    }

    return res;  // če res.size() != n, ima graf cikle.
}
```

1.2 Najdaljša pot v DAGu

Vhod: Usmerjen utežen graf G brez ciklov in vozlišči s in t. G ne sme imeti zank, če pa jih ima, se jih lahko brez škode odstrani.

Izhod: Dolžino najdaljše poti med s in t, oz. -1, če ta pot ne obstaja. Z lahkoto najdemo tudi dejansko pot (shranjujemo predhodnika) ali najkrajšo pot (max $\to$ min).

Časovna zahtevnost: $O(V + E)$

Prostorska zahtevnost: $O(V + E)$

Testiranje na terenu: UVa 103

```
int longest_path_in_a_dag(const vector<vector<pair<int, int>>>& graf, int s, int t) {
    int n = graf.size(), v, w;
    vector<int> ind(n, 0);
    vector<int> max_dist(n, -1);
    for (int i = 0; i < n; ++i)
        for (const auto& edge : graf[i])
            ind[edge.first]++;

    max_dist[s] = 0;

    queue<int> q;
    for (int i = 0; i < n; ++i)
        if (ind[i] == 0)
```

```
            q.push(i);  // topološko uredimo in gledamo maksimum

    while (!q.empty()) {
        int u = q.front();
        q.pop();

        for (const auto& edge : graf[u]) {
            tie(v, w) = edge;
            if (max_dist[u] >= 0)  // da začnemo pri s-ju, sicer bi začeli na začetku, vsi pred s -1
                max_dist[v] = max(max_dist[v], max_dist[u] + w);  // min za shortest path
            if (--ind[v] == 0) q.push(v);
        }
    }
    return max_dist[t];
}
```

1.3 Mostovi in prerezna vozlišča grafa

Vhod: Število vozlišč n in število povezav m ter seznam povezav E oblike $u \to v$ dolžine m. Neusmerjen graf G je tako sestavljen iz vozlišč z oznakami 0 do $n-1$ in povezavami iz E.

Izhod: Seznam prereznih vozlišč: točk, pri katerih, če jih odstranimo, graf razpade na dve komponenti in seznam mostov grafa G: povezav, pri katerih, če jih odstranimo, graf razpade na dve komponenti.

Časovna zahtevnost: $O(V+E)$

Prostorska zahtevnost: $O(V+E)$

Testiranje na terenu: UVa 315

```
namespace {
vector<int> low;
vector<int> dfs_num;
vector<int> parent;
}

void articulation_points_and_bridges_internal(int u, const vector<vector<int>>& graf,
        vector<bool>& articulation_points_map, vector<pair<int, int>>& bridges) {
    static int dfs_num_counter = 0;
    low[u] = dfs_num[u] = ++dfs_num_counter;
    int children = 0;
    for (int v : graf[u]) {
        if (dfs_num[v] == -1) {  // unvisited
            parent[v] = u;
            children++;

            articulation_points_and_bridges_internal(v, graf, articulation_points_map, bridges);
            low[u] = min(low[u], low[v]);  // update low[u]

            if (parent[u] == -1 && children > 1)  // special root case
                articulation_points_map[u] = true;
            else if (parent[u] != -1 && low[v] >= dfs_num[u])  // articulation point
                articulation_points_map[u] = true;  // assigned more than once
            if (low[v] > dfs_num[u])                // bridge
                bridges.push_back({u, v});
        } else if (v != parent[u]) {
            low[u] = min(low[u], dfs_num[v]);  // update low[u]
        }
    }
}

void articulation_points_and_bridges(int n, int m, const int E[][2],
        vector<int>& articulation_points, vector<pair<int, int>>& bridges) {
    vector<vector<int>> graf(n);
    for (int i = 0; i < m; ++i) {
        int a = E[i][0], b = E[i][1];
        graf[a].push_back(b);
        graf[b].push_back(a);
    }

    low.assign(n, -1);
    dfs_num.assign(n, -1);
    parent.assign(n, -1);

```

```cpp
    vector<bool> articulation_points_map(n, false);
    for (int i = 0; i < n; ++i)
        if (dfs_num[i] == -1)
            articulation_points_and_bridges_internal(i, graf, articulation_points_map, bridges);

    for (int i = 0; i < n; ++i)
        if (articulation_points_map[i])
            articulation_points.push_back(i);  // actually return only articulation points
}
```

1.4 Močno povezane komponente

Vhod: Seznam sosednosti s težami povezav.

Izhod: Seznam povezanih komponent grafa v obratni topološki ureditvi in kvocientni graf, to je DAG, ki ga dobimo iz grafa, če njegove komponente stisnemo v točke. Morebitnih več povezav med dvema komponentama seštejemo.

Časovna zahtevnost: $O(V + E)$

Prostorska zahtevnost: $O(V + E)$

Testiranje na terenu: `http://putka.upm.si/tasks/2012/2012_3kolo/zakladi`

```cpp
namespace {
vector<int> low;
vector<int> dfs_num;
stack<int> S;
vector<int> component;   // maps vertex to its component
}

void strongly_connected_components_internal(int u, const vector<vector<pair<int, int>>>& graf,
        vector<vector<int>>& comps) {
    static int dfs_num_counter = 1;
    low[u] = dfs_num[u] = dfs_num_counter++;
    S.push(u);

    for (const auto& v : graf[u]) {
        if (dfs_num[v.first] == 0)  // not visited yet
            strongly_connected_components_internal(v.first, graf, comps);
        if (dfs_num[v.first] != -1)  // not popped yet
            low[u] = min(low[u], low[v.first]);
    }

    if (low[u] == dfs_num[u]) {  // extract the component
        int cnum = comps.size();
        comps.push_back({});  // start new component
        int w;
        do {
            w = S.top(); S.pop();
            comps.back().push_back(w);
            component[w] = cnum;
            dfs_num[w] = -1;  // mark popped
        } while (w != u);
    }
}

void strongly_connected_components(const vector<vector<pair<int, int>>>& graf,
        vector<vector<int>>& comps, vector<map<int, int>>& dag) {
    int n = graf.size();
    low.assign(n, 0);
    dfs_num.assign(n, 0);
    component.assign(n, -1);

    for (int i = 0; i < n; ++i)
        if (dfs_num[i] == 0)
            strongly_connected_components_internal(i, graf, comps);

    dag.resize(comps.size());  // zgradimo kvocientni graf, teza povezave je vsota tez
    for (int u = 0; u < n; ++u) {
        for (const auto& v : graf[u]) {
            if (component[u] != component[v.first]) {
                dag[component[u]][component[v.first]] += v.second;  // ali max, kar zahteva naloga
            }
        }
    }
}
```

1.5 Najkrajša pot v grafu

1.5.1 Dijkstra

Vhod: Seznam sosednosti s težami povezav in dve točki grafa. Povezave morajo biti pozitivne.

Izhod: Dolžina najkrajša poti od prve do druge točke. Z lahkoto vrne tudi pot, glej kvadratično verzijo za implementacijo.

Časovna zahtevnost: $O(E \log(E))$

Prostorska zahtevnost: $O(V + E)$

Testiranje na terenu: `http://putka.upm.si/tasks/2013/2013_1kolo/wolowitz`

```
typedef pair<int, int> pii;

int dijkstra(const vector<vector<pii>>& graf, int s, int t) {
    int n = graf.size(), d, u;
    priority_queue<pii, vector<pii>, greater<pii>> q;
    vector<bool> visited(n, false);
    vector<int> dist(n);

    q.push({0, s});  // {cena, tocka}
    while (!q.empty()) {
        tie(d, u) = q.top();
        q.pop();

        if (visited[u]) continue;
        visited[u] = true;
        dist[u] = d;

        if (u == t) break;  // ce iscemo do vseh tock spremeni v --n == 0

        for (const auto& p : graf[u])
            if (!visited[p.first])
                q.push({d + p.second, p.first});
    }
    return dist[t];
}
```

1.5.2 Dijkstra (kvadratičen)

Vhod: Seznam sosednosti s težami povezav in dve točki grafa. Povezave morajo biti pozitivne.

Izhod: Najkrajša pot med danima točkama, dana kot seznam vmesnih vozlišč skupaj z obema krajiščema.

Časovna zahtevnost: $O(V^2)$, to je lahko bolje kot $O(E \log(E))$.

Prostorska zahtevnost: $O(V + E)$

Testiranje na terenu: `http://putka.upm.si/tasks/2013/2013_1kolo/wolowitz`

```
vector<int> dijkstra_square(const vector<vector<pair<int, int>>>& graf, int s, int t) {
    int INF = numeric_limits<int>::max();
    int n = graf.size(), to, len;
    vector<int> dist(n, INF),  prev(n);
    dist[s] = 0;
    vector<bool> visited(n, false);
    for (int i = 0; i < n; ++i) {
        int u = -1;
        for (int j = 0; j < n; ++j)
            if (!visited[j] && (u == -1 || dist[j] < dist[u]))
                u = j;                          // vertex with minimum dist
        if (u == -1 || dist[u] == INF) break;  // disconnected graph
        if (u == t) break;  // found shortest path to target
        visited[u] = true;

        for (const auto& edge : graf[u]) {
            tie(to, len) = edge;
            if (dist[u] + len < dist[to]) {  // if path can be improved via me
                dist[to] = dist[u] + len;
```

```
                prev[to] = u;
            }
        }
    }  // v dist so sedaj razdalje od s do vseh, ki so bližje kot t (in t)
    vector<int> path;  // ce je dist[t] == INF, je t v drugi komponenti kot s
    for (int v = t; v != s; v = prev[v])
        path.push_back(v);
    path.push_back(s);
    reverse(path.begin(), path.end());
    return path;
}
```

1.5.3 Bellman-Ford

Vhod: Seznam sosednosti s težami povezav in točka grafa. Povezave ne smejo imeti negativnega cikla (duh).

Izhod: Vrne razdaljo od dane točke do vseh drugih. Ni nič ceneje če iščemo samo do določene točke.

Časovna zahtevnost: $O(EV)$

Prostorska zahtevnost: $O(V + E)$

Testiranje na terenu: `http://putka.upm.si/tasks/2013/2013_1kolo/wolowitz`

```
vector<int> bellman_ford(const vector<vector<pair<int, int>>>& graf, int s) {
    int INF = numeric_limits<int>::max();
    int n = graf.size(), v, w;
    vector<int> dist(n, INF);
    vector<int> prev(n, -1);
    vector<bool> visited(n, false);

    dist[s] = 0;
    for (int i = 0; i < n-1; ++i) {  // i je trenutna dolžina poti
        for (int u = 0; u < n; ++u) {
            for (const auto& edge : graf[u]) {
                tie(v, w) = edge;
                if (dist[u] != INF && dist[u] + w < dist[v]) {
                    dist[v] = dist[u] + w;
                    prev[v] = u;
                }
            }
        }
    }

    for (int u = 0; u < n; ++u) {  // cycle detection
        for (const auto& edge : graf[u]) {
            tie(v, w) = edge;
            if (dist[u] != INF && dist[u] + w < dist[v])
                return {};  // graph has a negative cycle !!
        }
    }
    return dist;
}
```

1.5.4 Floyd-Warhsall

Vhod: Število vozlišč, število povezav in seznam povezav. Povezave ne smejo imeti negativnega cikla (duh).

Izhod: Vrne matriko razdalj med vsemi točkami, $d[i][j]$ je razdalja od i-te do j-te točke. Če je katerikoli diagonalen element negativen, ima graf negativen cikel. Rekonstrukcija poti je možna s pomočjo dodatne tabele, kjer hranimo naslednika.

Časovna zahtevnost: $O(V^3)$, dober za goste grafe.

Prostorska zahtevnost: $O(V^2)$

Testiranje na terenu: `http://putka.upm.si/tasks/2013/2013_1kolo/wolowitz`

```
vector<vector<int>> floyd_warshall(int n, int m, const int E[][3]) {
    int INF = numeric_limits<int>::max();
    vector<vector<int>> d(n, vector<int>(n, INF));
    // vector<vector<int>> next(n, vector<int>(n, -1));  // da dobimo pot
    for (int i = 0; i < m; ++i) {
        int u = E[i][0], v = E[i][1], c = E[i][2];
        d[u][v] = c;
        // next[u][v] = v
    }

    for (int i = 0; i < n; ++i)
        d[i][i] = 0;

    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                if (d[i][k] != INF && d[k][j] != INF && d[i][k] + d[k][j] < d[i][j])
                    d[i][j] = d[i][k] + d[k][j];
                    // next[i][j] = next[i][k];
    return d;  // ce je kateri izmed d[i][i] < 0, ima graf negativen cikel
}
```

1.6 Minimalno vpeto drevo

1.6.1 Prim

Vhod: Neusmerjen povezan graf s poljubnimi cenami povezav.

Izhod: Vrne ceno najmanjšega vpetega drevesa. Z lahkoto to zamenjamo z maksimalnim (ali katerokoli podobno operacijo) drevesom.

Časovna zahtevnost: $O(E\log(E))$, dober za goste grafe.

Prostorska zahtevnost: $O(V+E)$

Testiranje na terenu: UVa 11631

```
typedef pair<int, int> pii;

int prim_minimal_spanning_tree(const vector<vector<pii>>& graf) {
    int n = graf.size(), d, u;
    vector<bool> visited(n, false);
    priority_queue<pii, vector<pii>, greater<pii>> q;  // remove greater for max-tree
    q.push({0, 0});

    int sum = 0;            // sum of the mst
    int edge_count = 0;     // stevilo dodanih povezav
    while (!q.empty()) {
        tie(d, u) = q.top();
        q.pop();

        if (visited[u]) continue;
        visited[u] = true;

        sum += d;
        if (++edge_count == n) break;  // drevo, jebeš solato

        for (const auto& edge : graf[u])
            if (!visited[edge.first])
                q.push({edge.second, edge.first});
    }  // ce zelimo drevo si shranjujemo se previous vertex.
    return sum;
}
```

1.6.2 Kruskal

Vhod: Neusmerjen povezan graf s poljubnimi cenami povezav.

Izhod: Vrne ceno najmanjšega vpetega drevesa. Z lahkoto to zamenjamo z maksimalnim (ali katerokoli podobno operacijo) drevesom.

Časovna zahtevnost: $O(E\log(E))$, dober za redke grafe. Če so povezave že sortirane, samo $O(E\alpha(V))$.

Prostorska zahtevnost: $O(V + E)$

Testiranje na terenu: UVa 11631

```cpp
namespace {
vector<int> parent;
vector<int> rank;
}

int find(int x) {
    if (parent[x] != x)
        parent[x] = find(parent[x]);
    return parent[x];
}

bool unija(int x, int y) {
    int xr = find(x);
    int yr = find(y);

    if (xr == yr) return false;
    if (rank[xr] < rank[yr]) {      // rank lahko tudi izpustimo, potem samo parent[xr] = yr;
        parent[xr] = yr;
    } else if (rank[xr] > rank[yr]) {
        parent[yr] = xr;
    } else {
        parent[yr] = xr;
        rank[xr]++;
    }
    return true;
}

int kruskal_minimal_spanning_tree(int n, int m, int E[][3]) {
    rank.assign(n, 0);
    parent.assign(n, 0);
    for (int i = 0; i < n; ++i) parent[i] = i;
    vector<tuple<int, int, int>> edges;
    for (int i = 0; i < m; ++i) edges.emplace_back(E[i][2], E[i][0], E[i][1]);
    sort(edges.begin(), edges.end());

    int sum = 0, a, b, c, edge_count = 0;
    for (int i = 0; i < m; ++i) {
        tie(c, a, b) = edges[i];
        if (unija(a, b)) {
            sum += c;
            edge_count++;
        }
        if (edge_count == n - 1) break;
    }
    return sum;
}
```

1.7 Najnižji skupni prednik

Vhod: Drevo, podano s tabelo staršev. Vozlišče je koren, če je starš samemu sebi. Za queryje najprej potrebuješ pomožno tabelo skokov na višja vozlišča in tabelo nivojev.

Izhod: Za dani vozlišči u in v, vrne njunega najnižjega skupnega prednika, to je tako vozlišče p, da je p leži na poti od u do korena in od v do korena, ter je najdlje stran od korena drevesa.

Časovna zahtevnost: $O(\log(n))$ na query, s $O(n \log(n))$ predprocesiranja.

Prostorska zahtevnost: $O(n \log(n))$

Testiranje na terenu: `http://www.spoj.com/submit/LCA/`

```cpp
vector<vector<int>> preprocess(const vector<int>& parent) {
    int n = parent.size();
    int logn = 1;
    while (1 << ++logn < n);
    vector<vector<int>> P(n, vector<int>(logn, -1));

    for (int i = 0; i < n; i++)  // prvi prednik za i je parent[i]
        P[i][0] = parent[i];

```

```
    for (int j = 1; 1 << j < n; j++)
        for (int i = 0; i < n; i++)
            if (P[i][j - 1] != -1)                // P[i][j] = 2^j-ti prednik i-ja
                P[i][j] = P[P[i][j - 1]][j - 1];
    return P;
}

int level_internal(const vector<int>& parent, vector<int>& L, int v) {
    if (L[v] != -1) return L[v];
    return L[v] = (parent[v] == v) ? 0 : level_internal(parent, L, parent[v]) + 1;
}

vector<int> levels(const vector<int>& parent) {
    vector<int> L(parent.size(), -1);
    for (size_t i = 0; i < parent.size(); ++i) level_internal(parent, L, i);
    return L;
}

int find_lca(const vector<int>& parent, int u, int v,
             const vector<vector<int>>& P, const vector<int>& L) {
    if (L[u] < L[v])  // if u is on a higher level than v then we swap them
        swap(u, v);

    int log = 1;
    while (1 << ++log <= L[u]);
    log--;  // we compute the value of [log(L[u)]

      for (int i = log; i >= 0; i--)      // we find the ancestor of node u situated on
          if (L[u] - (1 << i) >= L[v])    // the same level as v using the values in P
              u = P[u][i];

      if (u == v) return u;

      for (int i = log; i >= 0; i--)  // we compute LCA(u, v) using the values in P
          if (P[u][i] != -1 && P[u][i] != P[v][i])
              u = P[u][i], v = P[v][i];

      return parent[u];
}
```

1.8 Največji pretok in najmanjši prerez

1.8.1 Edmonds-Karp

Vhod: Matrika kapacitet, vse morajo biti nenegativne.

Izhod: Vrne maksimalen pretok, ki je enak minimalnemu prerezu. Konstruira tudi matriko pretoka.

Časovna zahtevnost: $O(VE^2)$

Prostorska zahtevnost: $O(V^2)$

Testiranje na terenu: UVa 820

```
namespace {
const int INF = numeric_limits<int>::max();
struct triple { int u, p, m; };
}

int edmonds_karp_maximal_flow(const vector<vector<int>>& capacity, int s, int t) {
    int n = capacity.size();
    vector<vector<int>> flow(n, vector<int>(n, 0));
    int maxflow = 0;
    while (true) {
        vector<int> prev(n, -2);  // hkrati tudi visited array
        int bot = INF;            // bottleneck
        queue<triple> q;
        q.push({s, -1, INF});
        while (!q.empty()) {      // compute a possible path, add its bottleneck to the total flow
            int u = q.front().u, p = q.front().p, mini = q.front().m;  // while such path exists
            q.pop();

            if (prev[u] != -2) continue;
            prev[u] = p;

            if (u == t) { bot = mini; break; }
```

```

            for (int i = 0; i < n; ++i) {
                int available = capacity[u][i] - flow[u][i];
                if (available > 0) {
                    q.push({i, u, min(available, mini)});  // kumulativni minimum
                }
            }
        }

        if (prev[t] == -2) break;

        maxflow += bot;
        for (int u = t; u != s; u = prev[u]) {  // popravimo tretnurni flow nazaj po poti
            flow[u][prev[u]] -= bot;
            flow[prev[u]][u] += bot;
        }
    }
    return maxflow;
}
```

1.9 Največje prirejanje in najmanjše pokritje

V angleščini: *maximum cardinality bipartite matching* (če bi dodali še kakšno povezavo bi se dve stikali) in *minimum vertex cover* (če bi vzeli še kakšno točko stran, bi bila neka povezava brez pobarvane točke na obeh koncih).

Vhod: Dvodelen neutežen graf, dan s seznamom sosedov. Prvih `left` vozlišč je na levi strani.

Izhod: Število povezav v $MCBM$ = število točk v MVC, prvi MVC vrne tudi neko minimalno pokritje. Velja tudi $MIS = V - MCBM$, MIS pomeni *maximum independent set*.

Časovna zahtevnost: $O(VE)$

Prostorska zahtevnost: $O(V + E)$

Testiranje na terenu: UVa 11138

```
namespace {
vector<int> match, vis;
}

int augmenting_path(const vector<vector<int>>& graf, int left) {
    if (vis[left]) return 0;
    vis[left] = 1;
    for (int right : graf[left]) {
        if (match[right] == -1 || augmenting_path(graf, match[right])) {
            match[right] = left;
            match[left] = right;
            return 1;
        }
    }
    return 0;
}

void mark_vertices(const vector<vector<int>>& graf, vector<bool>& cover, int v) {
    if (vis[v]) return;
    vis[v] = 1;
    cover[v] = false;
    for (int r : graf[v]) {
        cover[r] = true;
        if (match[r] != -1)
            mark_vertices(graf, cover, match[r]);
    }
}

int bipartite_matching(const vector<vector<int>>& graf, int left_num) {
    int n = graf.size();
    match.assign(2*n, -1);
    int mcbm = 0;              // prvih left_num je v levem delu grafa
    for (int left = 0; left < left_num; ++left) {
        vis.assign(n, 0);
        mcbm += augmenting_path(graf, left);
```

```
    }
    return mcbm;
}

vector<int> minimal_cover(const vector<vector<int>>& graf, int left_num) {
    bipartite_matching(graf, left_num);
    int n = graf.size();
    vis.assign(2*n, 0);
    vector<bool> cover(n, false);
    fill(cover.begin(), cover.begin() + left_num, true);
    for (int left = 0; left < n; ++left)
        if (match[left] == -1)
            mark_vertices(graf, cover, left);

    vector<int> result;  // ni potrebno, lahko se uporablja kar cover
    for (int i = 0; i < n; ++i)
        if (cover[i])
            result.push_back(i);
    return result;
}
```

2 Podatkovne strukture

2.1 Statično binarno iskalno drevo

Operacije: Klasično uravnoteženo binarno iskalno drevo.

- vstavi: doda +1 k countu na mestu `idx`
- briši: vrne `true`/`false` glede na to ali element obstaja in če, zmanjša njegov count za 1
- najdi k-tega: vrne indeks k-tega elementa. Zero based.

Časovna zahtevnost: $O(\log(n))$ na operacijo

Prostorska zahtevnost: $O(n)$

Testiranje na terenu: `http://putka.upm.si/tasks/2011/2011_finale/kitajci`

```
namespace {
int depth = 10;  // depth of the tree
int n = 1 << 10;  // number of different elements stored
vector<int> tree(2*n);  // numer of elements in the tree = tree[0]
}

void insert(int idx, int val = 1) {
    int i = n - 1 + idx;
    while (i > 0) {
        tree[i] += val;
        i--;
        i >>= 1;
    }
    tree[0] += val;
}

int get_kth(int k) {
    int i = 0;
    while (i < n - 1) {
        int lc = tree[2*i+1];
        if (lc <= k) {
            i = 2*i + 2;
            k -= lc;
        } else {
            i = 2*i + 1;
        }
    }
    return i - n + 1;
}

bool remove(int idx) {
    if (tree[n-1 + idx] <= 0) return false;
    insert(idx, -1);
    return true;
}

```

```
void print() {
    for (int i = 0; i <= depth; ++i) {
        cout << string((1 << (depth - i)) - 1, ' ');
        for (int j = (1 << i) - 1; j < (1 << (i+1)) - 1; ++j) {
            cout << tree[j] << string((1 << (depth - i + 1)) - 1, ' ');
        }
        cout << '\n';
    }
}
```

2.2 Drevo segmentov

Operacije: Segment tree deljen po fiksnih točkah z dinamično alokacijo node-ov. Ob ustvarjanju roota povemo razpon vstavljanja, končne točke so postavljene po celih številih.

Za remove, ki ne zagotavlja nujno, da obstajajo stvari ki jih brišemo, se je treba malo bolj potruditi. Najprej odstranimo vse na trenutnem levelu, kolikor lahko, nato pa se v vsakem primeru pokličemo dalje (če je še kaj za odstranit in node-i obstajajo). Prav tako lahko vrnemo število izbrisanih stvari.

- vstavi neko vrednost na intervalu $[a, b]$
- briši na intervalu $[a, b]$
- dobi vrednost na intervalu $[a, b]$
- najdi k-tega

Časovna zahtevnost: $O(\log(n))$ na operacijo

Prostorska zahtevnost: $O(n)$

Testiranje na terenu: `http://putka.upm.si/competitions/upm2014-finale/izstevanka`

```
struct Node {  // static division points, [l, r] intervals
    int l, m, r;
    Node* left, *right;
    int here_count, total_count;
    Node(int ll, int rr) : l(ll), m((ll+rr) / 2), r(rr),
                           left(nullptr), right(nullptr), here_count(0), total_count(0) {}
    int insert(int f, int t, int inc = 1) {  // insert inc elemnts into [f, t]. use for removal too
        if (f <= l && r <= t) {
            here_count += inc;
            int lb = max(l, f), rb = min(r, t);
            int inserted = (rb - lb + 1) * inc;
            total_count += inserted;
            return inserted;
        }
        int inserted = 0;
        if (f <= m) {
            if (left == nullptr) left = new Node(l, m);
            inserted += left->insert(f, t, inc);
        }
        if (m + 1 <= t) {
            if (right == nullptr) right = new Node(m + 1, r);
            inserted += right->insert(f, t, inc);
        }
        total_count += inserted;

        if (left != nullptr && right != nullptr) {  // move full levels up (speedup, ne rabiš)
            int child_here_count = min(left->here_count, right->here_count);
            left->here_count -= child_here_count;
            right->here_count -= child_here_count;
            here_count += child_here_count;
            left->total_count -= (m - l + 1) * child_here_count;
            right->total_count -= (r - m) * child_here_count;
        }  // end speed up
        return inserted;
    }

    int count(int f, int t) {  // count on interval [f, t]
        if (f <= l && r <= t) return total_count;
        int sum = 0, lb = max(l, f), rb = min(r, t);
        if (f <= m && left != nullptr) sum += left->count(f, t);
```

```
        if (m + 1 <= t && right != nullptr) sum += right->count(f, t);
        return sum + (rb - lb + 1) * here_count;
    }

    int get_kth(int k, int parent_count = 0) {  // zero based
        int above_count = here_count + parent_count;
        int lc = above_count * (m - l + 1) + get_cnt(left);
        if (k < lc) {
            if (left == nullptr) return l + k/above_count;
            return left->get_kth(k, above_count);
        } else {
            k -= lc;
            if (right == nullptr) return m + 1 + k/above_count;
            return right->get_kth(k, above_count);
        }
    }

    void print(int l, int r) {
        printf("[%d, %d]: here: %d, total: %d\n", l, r, here_count, total_count);
        if (left) left->print(l, m);
        if (right) right->print(m+1, r);
    }

  private:
    int get_cnt(Node* left) {
        return (left == nullptr) ? 0 : left->total_count;
    }
};

// poenostavitev, ce ne rabimo intervalov
struct SimpleNode {  // static division points
    SimpleNode* left, *right;
    int cnt;
    SimpleNode() : left(nullptr), right(nullptr), cnt(0) {}
    void insert(int l, int r, int val) {
        if (l == r) { cnt++; return; }
        int mid = (l + r) / 2;  // watch overflow
        if (val <= mid) {
            if (left == nullptr) left = new SimpleNode();
            left->insert(l, mid, val);
        } else {
            if (right == nullptr) right = new SimpleNode();
            right->insert(mid + 1, r, val);
        }
        cnt++;
    }

    int pop_kth(int l, int r, int k) {  // one based
        if (l == r) { cnt--; return l; }
        int mid = (l + r) / 2;
        if (k <= get_cnt(left)) {
            cnt--;
            return left->pop_kth(l, mid, k);
        } else {
            k -= get_cnt(left);
            cnt--;
            return right->pop_kth(mid + 1, r, k);
        }
    }

  private:
    int get_cnt(SimpleNode* x) {
        if (x == nullptr) return 0;
        return x->cnt;
    }
};
#endif  // IMPLEMENTACIJA_PS_SEGMENT_TREE_H_
```

2.3 Avl drevo

Operacije: Klasično uravnoteženo binarno iskalno drevo.

- vstavi: doda +1 k countu, če obstaja
- najdi: vrne pointer na node ali `nullptr`, če ne obstaja
- briši: vrne `true`/`false` glede na to ali element obstaja in samo zmanjša njegov count (memory overhead, ampak who cares)

- najdi n-tega, vrne `nullptr` če ne obstaja

Časovna zahtevnost: $O(\log(n))$ na operacijo
Prostorska zahtevnost: $O(n)$
Testiranje na terenu: `http://putka.upm.si/competitions/upm2014-finale/izstevanka`
Opombe: To je lepa implementacija. V praksi ne rabimo vsega public interface-a je dovolj samo imeti nekje globalen root in private metode.

```cpp
template<typename T>
class AvlNode {
  public:
    AvlNode<T>* left, *right;
    size_t height, size, count;
    T value;
    AvlNode(const T& v) : left(nullptr), right(nullptr), height(1), size(1), count(1), value(v) {}
    ostream& print(ostream& os, int indent = 0) {
        if (right != nullptr) right->print(os, indent+2);
        for (int i = 0; i < indent; ++i) os << ' ';  // or use string(indent, ' ')
        os << value << endl;
        if (left != nullptr) left->print(os, indent+2);
        return os;
    }
};

template<typename T>
class AvlTree {
  public:
    AvlTree() : root(nullptr) {}
    int size() const {
        return size(root);
    }
    AvlNode<T>* insert(const T& val) {
        return insert(val, root);
    }
    bool erase(const T& val) {
        return erase(val, root);
    }
    const AvlNode<T>* get_nth(size_t index) const {
        return get_nth(root, index);
    }
    const AvlNode<T>* find(const T& value) const {
        return find(root, value);
    }
    template<typename U>
    friend ostream& operator<<(ostream& os, const AvlTree<U>& tree);

  private:
    int size(const AvlNode<T>* const& node) const {
        if (node == nullptr) return 0;
        else return node->size;
    }
    size_t height(const AvlNode<T>* const& node) const {
        if (node == nullptr) return 0;
        return node->height;
    }
    int getBalance(const AvlNode<T>* const& node) const {
        return height(node->left) - height(node->right);
    }
    void updateHeight(AvlNode<T>* const& node) {
        node->height = max(height(node->left), height(node->right)) + 1;
    }
    void rotateLeft(AvlNode<T>*& node) {
        AvlNode<T>* R = node->right;
        node->size -= size(R->right) + R->count; R->size += size(node->left) + node->count;
        node->right = R->left; R->left = node; node = R;
        updateHeight(node->left); updateHeight(node);
    }
    void rotateRight(AvlNode<T>*& node) {
        AvlNode<T>* L = node->left;
        node->size -= size(L->left) + L->count; L->size += size(node->right) + node->count;
        node->left = L->right; L->right = node; node = L;
        updateHeight(node->right); updateHeight(node);
    }
    void balance(AvlNode<T>*& node) {
        int b = getBalance(node);
        if (b == 2) {
            if (getBalance(node->left) == -1) rotateLeft(node->left);
```

```
            rotateRight(node);
        } else if (b == -2) {
            if (getBalance(node->right) == 1) rotateRight(node->right);
            rotateLeft(node);
        } else {
            updateHeight(node);
        }
    }
    AvlNode<T>* insert(const T& val, AvlNode<T>*& node) {
        if (node == nullptr) return node = new AvlNode<T>(val);
        node->size++;
        AvlNode<T>* return_node = node;
        if (val < node->value) return_node = insert(val, node->left);
        else if (node->value == val) node->count++;
        else if (node->value < val) return_node = insert(val, node->right);
        balance(node);
        return return_node;
    }
    bool erase(const T& val, AvlNode<T>*& node) {
        if (node == nullptr) return false;
        if (val < node->value) {
            if (erase(val, node->left)) {
                node->size--;
                return true;
            }
        } else if (node->value < val) {
            if (erase(val, node->right)) {
                node->size--;
                return true;
            }
        } else if (node->value == val && node->count > 0) {
            node->count--;
            node->size--;
            return true;
        }
        return false;
    }
    const AvlNode<T>* get_nth(const AvlNode<T>* const& node, size_t n) const {
        size_t left_size = size(node->left);
        if (n < left_size) return get_nth(node->left, n);
        else if (n < left_size + node->count) return node;
        else if (n < node->size) return get_nth(node->right, n - left_size - node->count);
        else return nullptr;
    }
    const AvlNode<T>* find(const AvlNode<T>* const& node, const T& value) const {
        if (node == nullptr) return nullptr;
        if (value < node->value) return find(node->left, value);
        else if (value == node->value) return node;
        else return find(node->right, value);
    }

    AvlNode<T>* root;
};

template<typename T>
ostream& operator<<(ostream& os, const AvlTree<T>& tree) {
    if (tree.root == nullptr) os << "Tree empty";
    else tree.root->print(os);
    return os;
}
#endif  // IMPLEMENTACIJA_PS_AVL_TREE_H_
```

2.4 Fenwickovo drevo

Operacije: Imamo tabelo z indeksi $1 \leqslant x \leqslant 2^k$ v kateri hranimo števila. Želimo hitro posodabljati elemente in odgovarjati na queryje po vsoti podseznamov.

- preberi vsoto do indeksa x (za poljuben podseznam, $read(b) - read(a)$)
- posodobi število na indeksu x
- preberi število na indeksu x.

Časovna zahtevnost: $O(k)$ na operacijo
Prostorska zahtevnost: $O(2^k)$

Testiranje na terenu: `http://putka.upm.si/competitions/upm2013-finale/safety`

```
namespace {
const int MAX_INDEX = 16;
vector<int> tree(MAX_INDEX+1, 0);  // global tree, 1 based!!
}

void update(int idx, int val) {  // increments idx for value
    while (idx <= MAX_INDEX) {
        tree[idx] += val;
        idx += (idx & -idx);
    }
}

int read(int idx) {  // read sum of [1, x], read(0) == 0, duh.
    int sum = 0;
    while (idx > 0) {
        sum += tree[idx];
        idx -= (idx & -idx);
    }
    return sum;
}

int readSingle(int idx) {  // read a single value, readSingle(x) == read(x)-read(x-1)
    int sum = tree[idx];
    if (idx > 0) {
        int z = idx - (idx & -idx);
        idx--;
        while (idx != z) {
            sum -= tree[idx];
            idx -= (idx & -idx);
        }
    }
    return sum;
}
```

2.5 Fenwickovo drevo (n-dim)

Operacije: Imamo n-dim tabelo dimenzij $d_1 \times d_2 \times \cdots \times d_n$ z zero-based indeksi v kateri hranimo števila. Želimo hitro posodabljati elemente in odgovarjati na queryje po vsoti podkvadrov.

- preberi vsoto do vključno indeksa $\underline{x}$
- posodobi število na indeksu $\underline{x}$
- preberi vsoto na podkvadru (pravilo vključitev in izključitev)

Funkcije so napisane za 3D, samo dodaj ali odstrani for zanke za višje / nižje dimenzije in na ne kockasto tabelo.

Časovna zahtevnost: kumulativna vsota in update $O(\log(d_1 + \cdots + d_n))$, za vsoto podkvadra $O(2^d \log(d_1 + \cdots + d_n))$.

Prostorska zahtevnost: $O(d_1 \cdots d_n)$

Testiranje na terenu: `http://putka.upm.si/tasks/2010/2010_3kolo/stanovanja`

```
typedef vector<vector<vector<int>>> vvvi;

int sum(int x, int y, int z, const vvvi& tree) {  // [0,0,0 - x,y,z] vključno
    int result = 0;
    for (int i = x; i >= 0; i = (i & (i+1)) - 1)
        for (int j = y; j >= 0; j = (j & (j+1)) - 1)
            for (int k = z; k >= 0; k = (k & (k+1)) - 1)
                result += tree[i][j][k];
    return result;
}

void inc(int x, int y, int z, int delta, vvvi& tree) {  // povečaj na koordinatah, 0 based
    int n = tree.size();  // lahko so tudi različni n-ji za posamezno dimenzijo
    for (int i = x; i < n; i |= i+1)
        for (int j = y; j < n; j |= j+1)
            for (int k = z; k < n; k |= k+1)
                tree[i][j][k] += delta;
```

```
}

int subsum(int x1, int y1, int z1,
           int x2, int y2, int z2, const vvvi& tree) {  // vsota na [x1,y1,z1 - x2,y2,z2], vključno
    x1--; y1--; z1--;
    return sum(x2, y2, z2, tree) -
           sum(x1, y2, z2, tree) -
           sum(x2, y1, z2, tree) -  // pravilo vključitev in izključitev
           sum(x2, y2, z1, tree) +
           sum(x1, y1, z2, tree) +
           sum(x1, y2, z1, tree) +
           sum(x2, y1, z1, tree) -
           sum(x1, y1, z1, tree);
}
```

3 Algoritmi

3.1 Najdaljše skupno podzaporedje

Vhod: Dve zaporedji a in b dolžin n in m.

Izhod: Najdaljše skupno podzaporedje (ne nujno strnjeno) LCS. Lahko dobimo samo njegovo dolžino. Problem je povezan z najkrajšim skupnim nadzaporedjem (SCS). Velja $SCS + LCS = n + m$.

Časovna zahtevnost: $O(nm)$

Prostorska zahtevnost: $O(nm)$ za podzaporedje, $O(m)$ za dolžino.

Testiranje na terenu: UVa 10405

```
// lahko pridemo na O(n sqrt(n))
vector<int> longest_common_subsequence(const vector<int>& a, const vector<int>& b) {
    int n = a.size(), m = b.size();
    vector<vector<int>> c(n + 1, vector<int>(m + 1, 0));
    for (int i = 1; i <= n; ++i)
        for (int j = 1; j <= m; ++j)
            if (a[i-1] == b[j-1])
                c[i][j] = c[i-1][j-1] + 1;
            else
                c[i][j] = max(c[i][j-1], c[i-1][j]);
    vector<int> sequence;
    int i = n, j = m;
    while (i > 0 && j > 0) {
        if (a[i-1] == b[j-1]) {
            sequence.push_back(a[i-1]);
            i--; j--;
        } else if (c[i][j-1] > c[i-1][j]) {
            j--;
        } else {
            i--;
        }
    }
    reverse(sequence.begin(), sequence.end());
    return sequence;
}

//  O(n) prostora, lahko tudi zgornjo verzijo, ce je dovolj spomina.
int longest_common_subsequence_length(const vector<int>& a, const vector<int>& b) {
    int n = a.size(), m = b.size();  // po moznosi transponiraj tabelo, ce je malo spomina
    vector<vector<int>> c(2, vector<int>(m + 1, 0));
    bool f = 0;
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j)
            if (a[i-1] == b[j-1])
                c[f][j] = c[!f][j-1] + 1;
            else
                c[f][j] = max(c[f][j-1], c[!f][j]);
        f = !f;
    }
    return c[!f][m];
}
```

3.2 Najdaljše naraščajoče podzaporedje

Vhod: Zaporedje elementov na katerih imamo linearno urejenost.
Izhod: Najdaljše naraščajoče podzaporedje.
Časovna zahtevnost: $O(n \log(n))$ in $O(n^2)$
Prostorska zahtevnost: $O(n)$
Testiranje na terenu: UVa 103
Opomba: Za hitro verzijo je zaradi bisekcije potrebna linearna urejenost elementov. Pri n^2 verziji je dovolj delna urejenost. V tem primeru je elemente morda treba urediti, tako da je potem potrebno za urejanje izbrati neko linearno razširitev dane delne urejenosti. Pri obeh verzijah elementi niso omejeni na števila, vendar pri prvi ne moremo samo zamenjati tipa, ki ga funkcija vrača, lažje je spremeniti, da vrača indekse elementov namesto dejanskega zaporedja.

```cpp
vector<int> longest_increasing_subsequence(const vector<int>& a) {
    vector<int> p(a.size()), b;
    int u, v;

    if (a.empty()) return {};
    b.push_back(0);

    for (size_t i = 1; i < a.size(); i++) {
        if (a[b.back()] < a[i]) {
            p[i] = b.back();
            b.push_back(i);
            continue;
        }

        for (u = 0, v = b.size()-1; u < v;) {
            int c = (u + v) / 2;
            if (a[b[c]] < a[i]) u = c + 1;
            else v = c;
        }

        if (a[i] < a[b[u]]) {
            if (u > 0) p[i] = b[u-1];
            b[u] = i;
        }
    }

    for (u = b.size(), v = b.back(); u--; v = p[v]) b[u] = a[v];
    return b;       // b[u] = v, če želiš indekse, ali ce ima a neinteger elemente
}

vector<int> longest_increasing_subsequence_square(const vector<int>& a) {
    if (a.size() == 0) return {};
    int max_length = 1, best_end = 0;
    int n = a.size();
    vector<int> m(n, 0), prev(n, -1);  // m[i] = dolzina lis, ki se konca pri i
    m[0] = 1;
    prev[0] = -1;

    for (int i = 1; i < n; i++) {
        m[i] = 1;
        prev[i] = -1;

        for (int j = i-1; j >= 0; --j) {
            if (m[j] + 1 > m[i] && a[j] < a[i]) {
                m[i] = m[j] + 1;
                prev[i] = j;
            }

            if (m[i] > max_length) {
                best_end = i;
                max_length = m[i];
            }
        }
    }
    vector<int> lis;
    for (int i = best_end; i != -1; i = prev[i]) lis.push_back(a[i]);
    reverse(lis.begin(), lis.end());
    return lis;
}
```

3.3 Najdaljši strnjen palindrom

Vhod: Niz s dolžine n.

Izhod: Števili f in t, tako da je niz $s[f : t]$ palindrom največje dolžine, ki ga je možno najti v s. No nujno edini, niti prvi. Uporablja Mancherjev algoritem.

Časovna zahtevnost: $O(n)$

Prostorska zahtevnost: $O(n)$

Testiranje na terenu: `http://www.spoj.com/problems/LPS/`

```
pair<int, int> find_longest_palindrome(const string& str) {  // returns [start, end)
    int n = str.length();
    if (n == 0) return {0, 0};
    if (n == 1) return {0, 1};
    n = 2*n + 1;  // Position count
    int L[n];   // LPS Length Array
    L[0] = 0;
    L[1] = 1;
    int C = 1;    // centerPosition
    int R = 2;    // centerRightPosition
    int i = 0;    // currentRightPosition
    int iMirror;  // currentLeftPosition
    int maxLPSLength = 0, maxLPSCenterPosition = 0;
    int start = -1, end = -1, diff = -1;

    for (i = 2; i < n; i++) {
        iMirror  = 2*C-i;  // get currentLeftPosition iMirror for currentRightPosition i
        L[i] = 0;
        diff = R - i;  // If currentRightPosition i is within centerRightPosition R
        if (diff > 0) L[i] = min(L[iMirror], diff);
                                                               // Attempt to expand
        while ( ((i + L[i]) < n && (i - L[i]) > 0) && (        // palindrome centered at
                ((i + L[i] + 1) % 2 == 0) ||                   // currentRightPosition i Here
                (str[(i + L[i] + 1)/2] == str[(i - L[i] - 1)/2]))) {  // for odd positions, we
            L[i]++;                                            // compare characters and if
        }                                                      // match then increment LPS
                                                               // Length by ONE If even
        if (L[i] > maxLPSLength) {  // Track maxLPSLength      // position, we just increment
            maxLPSLength = L[i];                               // LPS by ONE without any
            maxLPSCenterPosition = i;                          // character comparison
        }

        if (i + L[i] > R) {  // If palindrome centered at currentRightPosition i
            C = i;           // expand beyond centerRightPosition R,
            R = i + L[i];    // adjust centerPosition C based on expanded palindrome.
        }
    }
    start = (maxLPSCenterPosition - maxLPSLength)/2;
    end = start + maxLPSLength;
    return {start, end};
}
```

3.4 Podseznam z največjo vsoto

Vhod: Zaporedje elementov a_i dolžine n.

Izhod: Največja možna vsota strnjenega podzaporedja a (lahko je tudi prazno). Alternativna verzija tudi vrne iskano zaporedje (najkrajše tako). Tretja verzija poišče k-to največjo vsoto.

Časovna zahtevnost: $O(n)$, $O(n \log(n) + nk)$

Prostorska zahtevnost: $O(n)$

Testiranje na terenu: `http://www.codechef.com/problems/KSUBSUM`

```
int maximum_subarray(const vector<int>& a) {  // glej komentarje ce ne dovolimo prazne vsote
    int max_ending_here = 0, max_so_far = 0;  // A[0]
    for (int x : a) {  // a[1:]
        max_ending_here = max(0, max_ending_here + x);
        max_so_far = max(max_so_far, max_ending_here);
    }
    return max_so_far;
```

```
}

vector<int> maximum_subarray_extract(const vector<int>& a) {
    int max_ending_here = 0, max_so_far = 0;
    int idx_from = 0, total_to = 0, total_from = 0;
    for (size_t i = 0; i < a.size(); ++i) {
        if (max_ending_here + a[i] > 0) {
            max_ending_here += a[i];
        } else {
            idx_from = i + 1;
        }
        if (max_ending_here > max_so_far) {
            total_from = idx_from;
            total_to = i + 1;
            max_so_far = max_ending_here;
        }
    }
    return vector<int>(a.begin() + total_from, a.begin() + total_to);
}

int maximum_subarray(const vector<int>& a, int k) {  // k = 1 ... n(n-1)/2
    int n = a.size();
    vector<int> s(n+1, 0);
    vector<pair<int, int>> p(n+1);
    priority_queue<tuple<int, int, int>> q;
    for (int i = 0, m = 0; i < n; ++i) {
        s[i+1] = s[i] + a[i];
        if (s[m] > s[i]) m = i;
        p[i+1] = make_pair(s[i+1], i+1);
        q.push(make_tuple(s[i+1]-s[m], i+1, m));
    }
    sort(p.begin(), p.end());
    vector<int> ss(n+1);
    for (int i = 0; i <= n; ++i)
        ss[p[i].second] = i;

    int v = -1, i, j;
    for (int l = 1; l <= k; ++l) {
        tie(v, i, j) = q.top();
        q.pop();

        for (int m = ss[j] + 1; m <= n; ++m) {
            if (p[m].second < i) {
                q.push(make_tuple(s[i]-p[m].first, i, p[m].second));
                break;
            }
        }
    }
    return v;
}
```

3.5 Leksikografsko minimalna rotacija

Vhod: Niz znakov s dolžine n.

Izhod: Indeks i, tako da je string $s[i:] + s[:i]$ leksikografsko najmanjši, izmed vseh možnih rotacij s.

Časovna zahtevnost: $O(n)$

Prostorska zahtevnost: $O(n)$

Testiranje na terenu: UVa 719

Opomba: Če smo res na tesnem s prostorom, lahko funkcija sprejme dejanski string in ga ne podvoji, ter dela vse indekse po modulu n.

```
int minimal_rotation(string s) {
    s += s;
    int len = s.size(), i = 0, j = 1, k = 0;
    while (i + k < len && j + k < len) {
        if (s[i+k] == s[j+k]) k++;
        else if (s[i+k] > s[j+k]) { i = i+k+1; if (i <= j) i = j+1; k = 0; }
        else if (s[i+k] < s[j+k]) { j = j+k+1; if (j <= i) j = i+1; k = 0; }
    }
    return min(i, j);
}
```

3.6 BigInt in Karatsuba

Class za računanje z velikimi števili, v poljubni bazi. IO deluje samo v desetiški.

Operacije: Seštevanje, odštevanje, množenje, primerjanje.

- seštevanje: samostojno, za negativne rabi $-$ in $<$.
- odštevanje: samostojno, če bo razlika pozitivna. Za negativne prevedi na seštevanje $a + (-b)$.
- množenje: rabi $+$, $\ll$ in $*$ s števko. Za negativne samo malo manipulacije predznakov. Lahko uporabiš tudi karatsubo.
- primerjanje: samostojno, za negativne samo malo manipulacije predznakov.

Jasno ni treba implementirati vsega. $+$ in $*$ nista tako zelo počasna, tako da verzije $+=$ ipd. niso nujno potrebne.

Časovna zahtevnost: $O(n)$ za $+$, $-$, $*$ stevka, $O(n^2)$ za $*$, $O(n^{1.585})$ za karatsubo.

Prostorska zahtevnost: $O(n)$

Testiranje na terenu: `http://putka.upm.si/tasks/test_okolja/odstevanje` in `http://putka.upm.si/tasks/test_okolja/sestevanje`

```cpp
template<typename T>
struct Number {
    bool sign = true;  // true = 1 = +, false = 0 = -
    deque<T> data;
    static const int base = 10;  // bi se lahko spremenilo samo I/O nebi delal, matematika bi
    static const int KARATSUBA_LIMIT = 2;  // kdaj preklopi na bruteforce množenje

    Number() {}  // default constructor, positive zero
    Number(const deque<T>& a, bool s = 1) : sign(s), data(a) { clear_zeros(); }
    Number(deque<T>&& a, bool s = 1) : sign(s), data(a) { clear_zeros(); }
    Number(const string& s) { from_string(s); }
    void from_string(const string& s) {
        if (s.size() == 0) data = {0};
        int i = 0;
        if (s[0] == '+') sign = 1, i = 1;
        if (s[0] == '-') sign = 0, i = 1;
        int l = s.size(); data.resize(l-i);
        for (; i < l; ++i) {
            if (!('0' <= s[i] && s[i] <= '9')) return;  // silent quit po prvi ne stevilki
            data[l-i-1] = s[i] - '0';
        }
        clear_zeros();
    }
    string to_string() const {
        if (data.empty()) return "0";
        string s = (sign) ? "" : "-";
        for (int i = data.size() - 1; i >= 0; --i)
            s.push_back('0' + data[i]);
        return s;
    }
    Number operator+(const Number& o) const {  // remove signs if using for positive only
        if (sign == 0) return -((-*this) + (-o));
        if (sign == 1 && o.sign == 0) return (*this < -o) ? -((-o) - *this) : *this - (-o);
        bool carry = false;
        int i = 0, j = 0, n = data.size(), m = o.data.size();
        deque<T> r;
        while (i < n || j < m) {
            T c = ((i < n) ? data[i++] : 0) + ((j < m) ? o.data[j++] : 0) + carry;
            carry = (c >= base);
            r.push_back(c % base);
        }
        if (carry) r.push_back(1);
        return Number(move(r));
    }
    Number operator-() const { return Number(data, !sign); }
    bool operator==(const Number& o) const { return sign == o.sign && data == o.data; }
    bool operator<(const Number& o) const {
        if (sign == o.sign) {
            if (sign == 0) return -o < -*this;
```

```
            if (data.size() == o.data.size()) {
                for (int i = data.size() - 1; i >= 0; --i)
                    if (data[i] == o.data[i]) continue;
                    else return data[i] < o.data[i];
            }
            return data.size() < o.data.size();
        }
        return sign < o.sign;
    }
    Number& operator+=(const Number& o) {  // lahko tudi s +. Samo za pozitivne.
        bool carry = false;
        int i = 0, j = 0, n = data.size(), m = o.data.size();
        while (i < n || j < m) {
            if (i < n && j < m) data[i] += o.data[j++] + carry;
            else if (i < n)     data[i] += carry;
            else                data.push_back(o.data[j++] + carry);
            carry = data[i] / base;
            data[i++] %= base;
        }
        if (carry) data.push_back(1);
        clear_zeros();
        return *this;
    }
    Number operator*(const T& o) const {  // z eno števko
        deque<T> r;
        int carry = 0, n = data.size();
        for (int i = 0; i < n; ++i) {
            T c = data[i]*o + carry;
            carry = c / base;
            c %= base;
            r.push_back(c);
        }
        if (carry) r.push_back(carry);
        return Number(move(r), sign);
    }
    Number operator<<(int n) const {  // na zacetek dodamo n ničel
        deque<T> r(n, 0);
        r.insert(r.end(), data.begin(), data.end());
        return Number(move(r));
    }
    Number operator*(const Number<T>& o) const {
        if (sign == 0 && o.sign == 0) return ((-*this) * (-o));
        if (sign == 0 && o.sign == 1) return -((-*this) * o);
        if (sign == 1 && o.sign == 0) return -(*this * (-o));
        Number r;
        int m = o.data.size();
        for (int i = 0; i < m; ++i)
            r += (*this*o.data[i] << i);
        return r;
    }
    Number operator-(const Number& o) const {
        deque<T> r;
        bool carry = false;
        int i = 0, j = 0, n = data.size(), m = o.data.size();
        while (i < n || j < m) {
            T c = data[i++] + base - ((j < m) ? o.data[j++] : 0) - carry;
            carry = 1 - c / base;
            c %= base;
            r.push_back(c);
        }
        return Number(move(r));
    }

  private:
    void clear_zeros() {
        while (data.size() > 0 && data.back() == 0) data.pop_back();
        if (data.empty()) sign = 1;
    }
};

template<typename T>  // karatsuba algorithm
Number<T> karatsuba(const Number<T>& a, const Number<T>& b) {
    if (a.data.size() <= Number<T>::KARATSUBA_LIMIT || b.data.size() <= Number<T>::KARATSUBA_LIMIT)
        return a*b;

    if (a.sign == 0 && b.sign == 0) return ((-a) * (-b));
    if (a.sign == 0 && b.sign == 1) return -((-a) * b);
    if (a.sign == 1 && b.sign == 0) return -(a * (-b));

    Number<T> a0, a1, b0, b1, c0, c1, c2;
    int m = min(a.data.size(), b.data.size())/2;  // choose m carefully
```

```

    a0.data.assign(a.data.begin(), a.data.begin()+m);
    a1.data.assign(a.data.begin()+m, a.data.end());
    b0.data.assign(b.data.begin(), b.data.begin()+m);
    b1.data.assign(b.data.begin()+m, b.data.end());

    c2 = karatsuba(a1, b1);
    c0 = karatsuba(a0, b0);
    c1 = karatsuba(a0+a1, b0+b1) - c0 - c2;

    return (c2 << 2*m) + (c1 << m) + c0;
}
#endif  // IMPLEMENTACIJA_ALGO_BIGINT_H_
```

4 Teorija števil

4.1 Evklidov algoritem

Vhod: $a, b \in \mathbb{Z}$

Izhod: Največji skupni delitelj a in b. Za pozitivna števila je pozitiven, če je eno število 0, je rezultat drugo število, pri negativnih je predznak odvisen od števila iteracij.

Časovna zahtevnost: $O(\log(a) + \log(b))$

Prostorska zahtevnost: $O(1)$

```
int gcd(int a, int b) {
    int t;
    while (b != 0) {
        t = a % b;
        a = b;
        b = t;
    }
    return a;
}
```

4.2 Razširjen Evklidov algoritem

Vhod: $a, b \in \mathbb{Z}$. Števili $retx$, $rety$ sta parametra samo za vračanje vrednosti.

Izhod: Števila x, y, d, pri čemer $d = \gcd(a, b)$, ki rešijo Diofantsko enačbo $ax + by = d$. V posebnem primeru, da je b tuj a, je x inverz števila a v multiplikativni grupi Z_b^*.

Časovna zahtevnost: $O(\log(a) + \log(b))$

Prostorska zahtevnost: $O(1)$

Testiranje na terenu: UVa 756

```
int ext_gcd(int a, int b, int& retx, int& rety) {
    int x = 0, px = 1, y = 1, py = 0, r, q;
    while (b != 0) {
        r = a % b; q = a / b;  // quotient and reminder
        a = b; b = r;          // gcd swap
        r = px - q * x;        // x swap
        px = x; x = r;
        r = py - q * y;        // y swap
        py = y; y = r;
    }
    retx = px; rety = py;      // return
    return a;
}
```

4.3 Kitajski izrek o ostankih

Vhod: Sistem n kongruenc $x \equiv a_i \pmod{m_i}$, m_i so paroma tuji.

Izhod: Število x, ki reši ta sistem dobimo po formuli

$$x = \left[\sum_{i=1}^{n} a_i \frac{M}{m_i} \left[\left(\frac{M}{mi}\right)^{-1}\right]_{m_i}\right]_M, \qquad M = \prod_{i=1}^{n} m_i,$$

kjer $[x^{-1}]_m$ označuje inverz x po modulu m. Vrnjeni x je med 0 in M.

Časovna zahtevnost: $O(n \log(\max\{m_i, a_i\}))$

Prostorska zahtevnost: $O(n)$

Potrebuje: Evklidov algoritem (str. 25)

Testiranje na terenu: UVa 756

Opomba: Pogosto potrebujemo `unsigned long long` namesto `int`.

```
int mul_inverse(int a, int m) {
    int x, y;
    ext_gcd(a, m, x, y);
    return (x + m) % m;
}

int chinese_reminder_theorem(const vector<pair<int, int>>& cong) {
    int M = 1;
    for (size_t i = 0; i < cong.size(); ++i) {
        M *= cong[i].second;
    }
    int x = 0, a, m;
    for (const auto& p : cong) {
        tie(a, m) = p;
        x += a * M / m * mul_inverse(M/m, m);
        x %= M;
    }
    return (x + M) % M;
}
```

4.4 Hitro potenciranje

Vhod: Število g iz splošne grupe in $n \in \mathbb{N}_0$.

Izhod: Število g^n.

Časovna zahtevnost: $O(\log(n))$

Prostorska zahtevnost: $O(1)$

Testiranje na terenu: `http://putka.upm.si/tasks/2010/2010_3kolo/nicle`

```
int fast_power(int g, int n) {
    int r = 1;
    while (n > 0) {
        if (n & 1) r *= g;
        g *= g;
        n >>= 1;
    }
    return r;
}
```

4.5 Številski sestavi

Vhod: Število $n \in \mathbb{N}_0$ ali $\frac{p}{q} \in Q$ ter $b \in [2, \infty) \cap \mathbb{N}$.

Izhod: Število n ali $\frac{p}{q}$ predstavljeno v izbranem sestavu z izbranimi števkami in označeno periodo.

Časovna zahtevnost: $O(\log(n))$ ali $O(q \log(q))$

Prostorska zahtevnost: $O(n)$ ali $O(q)$

Testiranje na terenu: `http://putka.upm.si/tasks/2010/2010_finale/ulomki`

Opomba: Zgornja meja za bazo b je dolžina niza `STEVILSKI_SESTAVI_ZNAKI`.

```
char STEVILSKI_SESTAVI_ZNAKI[] = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";

string convert_int(int n, int baza) {
    if (n == 0) return "0";
    string result;
    while (n > 0) {
        result.push_back(STEVILSKI_SESTAVI_ZNAKI[n % baza]);
        n /= baza;
    }
    reverse(result.begin(), result.end());
    return result;
}

string convert_fraction(int stevec, int imenovalec, int base) {
    div_t d = div(stevec, imenovalec);
    string result = convert_int(d.quot, base);
    if (d.rem == 0) return result;

    string decimalke;  // decimalni del
    result.push_back('.');
    int mesto = 0;
    map<int, int> spomin;
    spomin[d.rem] = mesto;
    while (d.rem != 0) {  // pisno deljenje
        mesto++;
        d.rem *= base;
        decimalke += STEVILSKI_SESTAVI_ZNAKI[d.rem / imenovalec];
        d.rem %= imenovalec;
        if (spomin.count(d.rem) > 0) {  // periodicno
            result.append(decimalke.begin(), decimalke.begin() + spomin[d.rem]);
            result.push_back('(');
            result.append(decimalke.begin() + spomin[d.rem], decimalke.end());
            result.push_back(')');
            return result;
        }
        spomin[d.rem] = mesto;
    }
    result += decimalke;
    return result;  // koncno decimalno stevilo
}
```

4.6 Eulerjeva funkcija ϕ

Vhod: Število $n \in \mathbb{N}$.

Izhod: Število $\phi(n)$, to je število števil manjših ali enakih n in tujih n. Direktna formula:

$$\phi(n) = n \cdot \prod_{p|n}(1 - \frac{1}{p})$$

Časovna zahtevnost: $O(\sqrt{n})$

Prostorska zahtevnost: $O(1)$

Testiranje na terenu: `https://projecteuler.net/problem=69`

```
int euler_phi(int n) {
    int res = n;
    for (int i = 2; i*i <= n; ++i) {
        if (n % i == 0) {
            while (n % i == 0) n /= i;
            res -= res / i;
        }
    }
    if (n > 1) res -= res / n;
    return res;
}
```

4.7 Eratostenovo rešeto

Vhod: Število $n \in \mathbb{N}$.

Izhod: Seznam praštevil manjših od n in seznam, kjer je za vsako število manjše od n notri njegov najmanjši praštevilski delitelj. To se lahko uporablja za faktorizacijo števil in testiranje praštevilskosti.

Časovna zahtevnost: $O(n \log(n))$

Prostorska zahtevnost: $O(n)$

Testiranje na terenu: UVa 10394

```
void erastothenes_sieve(int n, vector<int>& is_prime, vector<int>& primes) {
    is_prime.resize(n);
    for (int i = 2; i < n+1; ++i) {
        if (is_prime[i] == 0) {
            is_prime[i] = i;
            primes.push_back(i);
        }
        size_t j = 0;
        while (j < primes.size() && primes[j] <= is_prime[i] && i * primes[j] <= n) {
            is_prime[i * primes[j]] = primes[j];
            j++;
        }
    }
}
```

4.8 Število deliteljev

Vhod: Število $n \in \mathbb{N}$.

Izhod: Število pozitivnih deliteljev n, $\tau(n)$. Velja da za $n = p_1^{\alpha_1} \cdots p_k^{\alpha_k}$, je

$$\tau(n) = (\alpha_1 + 1) \cdots (\alpha_k + 1).$$

Časovna zahtevnost: $O(\sqrt(n))$

Prostorska zahtevnost: $O(1)$

Testiranje na terenu:

```
int number_of_divisors(int n) {
    int tau = 1;
    int i = 2;
    while (i * i <= n) {
        int p = 0;
        while (n % i == 0) {
            n /= i;
            p++;
        }
        tau *= p + 1;
        i++;
    }
    if (n > 1) tau *= 2;
    return tau;
}
```

5 Geometrija

Zaenkrat obravnavamo samo ravninsko geometrijo. Točke predstavimo kot kompleksna števila. Daljice predstavimo z začetno in končno točko. Premice s koeficienti v enačbi $ax + by = c$. Premico lahko konstruiramo iz dveh točk in po želji hranimo točko in smerni vektor. Pravokotnike predstavimo z spodnjim levim in zgornjim desnim ogliščem. Večkotnike predstavimo s seznamom točk, kot si sledijo, prve točke ne ponavljamo. Tip `ITYPE` predstavlja različne vrste presečišč ali vsebovanosti: `OK` pomeni, da se lepo seka oz. je točka v notranjosti. `NO` pomeni, da se ne seka oz. da točna ni vsebovana, `EQ` pa pomeni, da se premici prekrivata, daljici sekata v krajišču ali se pokrivata, oz. da je točka na robu.

5.1 Osnove

Funkcije:

- skalarni in vektorski produkt
- pravokotni vektor in polarni kot
- ploščina trikotnika in enostavnega mnogokotnika
- razred za premice
- razdalja do premice, daljice, po sferi
- vsebovanost v trikotniku, pravokotniku, enostavnem mnogokotniku
- presek dveh premic, premice in daljice in dveh daljic
- konstrukcije krogov iz treh točk, iz dveh točk in radija

Vhod: Pri argumentih funkcij.
Izhod: Pri argumentih funkcij.
Časovna zahtevnost: O(št. točk)
Prostorska zahtevnost: O(št. točk)
Testiranje na terenu: Bolj tako, ima pa obsežne unit teste. . .

```cpp
const double pi = M_PI;
const double eps = 1e-9;
const double inf = numeric_limits<double>::infinity();

enum ITYPE : char { OK, NO, EQ };
typedef complex<double> P;

template<typename T>
struct line_t {  // premica, dana z enačbo ax + by = c ali z dvema točkama
    double a, b, c;  // lahko tudi int
    line_t() : a(0), b(0), c(0) {}
    line_t(int A, int B, int C) {
        if (A < 0 || (A == 0 && B < 0)) a = -A, b = -B, c = -C;
        else a = A, b = B, c = C;
        int d = gcd(gcd(abs(a), abs(b)), abs(c));  // same sign as A, if nonzero, else B, else C
        if (d == 0) d = 1;                          // in case of 0 0 0 input
        a /= d;
        b /= d;
        c /= d;
    }
    line_t(T A, T B, T C) {
        if (A < 0 || (A == 0 && B < 0)) a = -A, b = -B, c = -C;
        else a = A, b = B, c = C;
    }
    line_t(const P& p, const P& q) : line_t(imag(q-p), real(p-q), cross(p, q)) {}
    P normal() const { return {a, b}; }
    double value(const P& p) const { return dot(normal(), p) - c; }
    bool operator<(const line_t<T>& line) const {  // da jih lahko vržemo v set, če T = int
        if (a == line.a) {
            if (b == line.b) return c < line.c;
            return b < line.b;
        }
        return a < line.a;
    }
    bool operator==(const line_t<T>& line) const {
        return cross(normal(), line.normal()) < eps && c*line.b == b*line.c;
    }
};
template<typename T>
ostream& operator<<(ostream& os, const line_t<T>& line) {
    os << line.a << "x + " << line.b << "y == " << line.c; return os;
}

typedef line_t<double> L;

#endif  // IMPLEMENTACIJA_GEOM_BASICS_H_
```

```cpp
double dot(const P& p, const P& q) {
    return p.real() * q.real() + p.imag() * q.imag();
}
```

```
double cross(const P& p, const P& q) {
    return p.real() * q.imag() - p.imag() * q.real();
}
double cross(const P& p, const P& q, const P& r) {
    return cross(q - p, r - q);  // > 0 levo, < 0 desno, = 0 naravnost
}
// true is p->q->r is a left turn, straight line is not, if so, change to -eps
bool left_turn(const P& p, const P& q, const P& r) {
    return cross(q-p, r-q) > eps;
}
P perp(const P& p) {  // get left perpendicular vector
    return P(-p.imag(), p.real());
}
int sign(double x) {
    if (x < -eps) return -1;
    if (x > eps) return 1;
    return 0;
}
double polar_angle(const P& p) {  // phi in [0, 2pi) or -1 for (0,0)
    if (p == P(0, 0)) return -1;
    double a = arg(p);
    if (a < 0) a += 2*pi;
    return a;
}
double area(const P& a, const P& b, const P& c) {  // signed
    return 0.5 * cross(a, b, c);
}
double area(const vector<P>& poly) {  // signed
    double A = 0;
    int n = poly.size();
    for (int i = 0; i < n; ++i) {
        int j = (i+1) % n;
        A += cross(poly[i], poly[j]);
    }
    return A/2;
}
double dist_to_line(const P& p, const L& line) {
    return abs(line.value(p)) / abs(line.normal());
}
double dist_to_line(const P& t, const P& p1, const P& p2) {  // t do premice p1p2
    return abs(cross(p2-p1, t-p1)) / abs(p2-p1);
}
double dist_to_segment(const P& t, const P& p1, const P& p2) {  // t do daljice p1p2
    P s = p2 - p1;
    P w = t - p1;
    double c1 = dot(s, w);
    if (c1 <= 0) return abs(w);
    double c2 = norm(s);
    if (c2 <= c1) return abs(t-p2);
    return dist_to_line(t, p1, p2);
}
double great_circle_dist(const P& a, const P& b) {  // pairs of (latitude, longitude) in radians
    double R = 6371.0;                              // compute great circle distance
    double u[3] = { cos(a.real()) * sin(a.imag()), cos(a.real()) * cos(a.imag()), sin(a.real()) };
    double v[3] = { cos(b.real()) * sin(b.imag()), cos(b.real()) * cos(b.imag()), sin(b.real()) };
    double dot = u[0]*v[0] + u[1]*v[1] + u[2]*v[2];
    bool flip = false;
    if (dot < 0.0) {
        flip = true;
        for (int i = 0; i < 3; i++) v[i] = -v[i];
    }
    double cr[3] = { u[1]*v[2] - u[2]*v[1], u[2]*v[0] - u[0]*v[2], u[0]*v[1] - u[1]*v[0] };
    double theta = asin(sqrt(cr[0]*cr[0] + cr[1]*cr[1] + cr[2]*cr[2]));
    double len = theta * R;
    if (flip) len = pi * R - len;
    return len;
}
bool point_in_rect(const P& t, const P& p1, const P& p2) {  // ali je t v pravokotniku p1p2
    return min(p1.real(), p2.real()) <= t.real() && t.real() <= max(p1.real(), p2.real()) &&
           min(p1.imag(), p2.imag()) <= t.imag() && t.imag() <= max(p1.imag(), p2.imag());
}
bool point_in_triangle(const P& t, const P& a, const P& b, const P& c) {  // orientation independant
    return abs(abs(area(a, b, t)) + abs(area(a, c, t)) + abs(area(b, c, t))  // edge inclusive
               - abs(area(a, b, c))) < eps;
}
pair<ITYPE, P> line_line_intersection(const L& p, const L& q) {
    double det = cross(p.normal(), q.normal());  // če imata odvisni normali (ali smerna vektorja)
    if (abs(det) < eps) {  // paralel
        if (abs(p.b*q.c - p.c*q.b) < eps && abs(p.a*q.c - p.c*q.a) < eps) {
            return {EQ, P()};  // razmerja koeficientov se ujemajo
        } else {
```

```
            return {NO, P()};
        }
    } else {
        return {OK, P(q.b*p.c - p.b*q.c, p.a*q.c - q.a*p.c) / det};
    }
}
pair<ITYPE, P> line_segment_intersection(const L& p, const P& u, const P& v) {
    double u_on = p.value(u);
    double v_on = p.value(v);
    if (abs(u_on) < eps && abs(v_on) < eps) return {EQ, u};
    if (abs(u_on) < eps) return {OK, u};
    if (abs(v_on) < eps) return {OK, v};
    if ((u_on > eps && v_on < -eps) || (u_on < -eps && v_on > eps)) {
        return line_line_intersection(p, L(u, v));
    }
    return {NO, P()};
}
pair<ITYPE, P> segment_segment_intersection(const P& p1, const P& p2, const P& q1, const P& q2) {
    int o1 = sign(cross(p1, p2, q1));  // daljico p1p1 sekamo z q1q2
    int o2 = sign(cross(p1, p2, q2));
    int o3 = sign(cross(q1, q2, p1));
    int o4 = sign(cross(q1, q2, p2));

    // za pravo presecisce morajo biti o1, o2, o3, o4 != 0
    // vemo da presečišče obstaja, tudi ce veljata samo prva dva pogoja
    if (o1 != o2 && o3 != o4 && o1 != 0 && o2 != 0 && o3 != 0 && o4 != 0)
        return line_line_intersection(L(p1, p2), L(q1, q2));

    // EQ = se dotika samo z ogliscem ali sta vzporedni
    if (o1 == 0 && point_in_rect(q1, p1, p2)) return {EQ, q1};  // q1 lezi na p
    if (o2 == 0 && point_in_rect(q2, p1, p2)) return {EQ, q2};  // q2 lezi na p
    if (o3 == 0 && point_in_rect(p1, q1, q2)) return {EQ, p1};  // p1 lezi na q
    if (o4 == 0 && point_in_rect(p2, q1, q2)) return {EQ, p2};  // p2 lezi na q

    return {NO, P()};
}
ITYPE point_in_poly(const P& t, const vector<P>& poly) {
    int n = poly.size();
    int cnt = 0;
    double x2 = rand() % 100;
    double y2 = rand() % 100;
    P dalec(x2, y2);
    for (int i = 0; i < n; ++i) {
        int j = (i+1) % n;
        if (dist_to_segment(t, poly[i], poly[j]) < eps) return EQ;  // boundary
        ITYPE tip = segment_segment_intersection(poly[i], poly[j], t, dalec).first;
        if (tip != NO) cnt++;  // ne testiramo, ali smo zadeli oglisce, upamo da nismo
    }
    if (cnt % 2 == 0) return NO;
    else return OK;
}
pair<P, double> get_circle(const P& p, const P& q, const P& r) {  // circle through 3 points
    P v = q-p;
    P w = q-r;
    if (abs(cross(v, w)) < eps) return {P(), 0};
    P x = (p+q)/2.0, y = (q+r)/2.0;
    ITYPE tip;
    P intersection;
    tie(tip, intersection) = line_line_intersection(L(x, x+perp(v)), L(y, y+perp(w)));
    return {intersection, abs(intersection-p)};
}
// circle through 2 points with given r, to the left of pq
P get_circle(const P& p, const P& q, double r) {
    double d = norm(p-q);
    double h = r*r / d - 0.25;
    if (h < 0) return P(inf, inf);
    h = sqrt(h);
    return (p+q) / 2.0 + h * perp(q-p);
}
```

5.2 Konveksna ovojnica

Vhod: Seznam n točk.

Izhod: Najkrajši seznam h točk, ki napenjajo konveksno ovojnico, urejen naraščajoče po kotu glede na spodnjo levo točko.

Časovna zahtevnost: $O(n \log n)$, zaradi sortiranja

Prostorska zahtevnost: $O(n)$
Potrebuje: Vektorski produkt, str. 29.
Testiranje na terenu: UVa 681

```
typedef complex<double> P;  // ali int

bool compare(const P& a, const P& b, const P& m) {
    double det = cross(a, m, b);
    if (abs(det) < eps) return abs(a-m) < abs(b-m);
    return det < 0;
}

vector<P> convex_hull(vector<P>& points) {  // vector is modified
    if (points.size() <= 2) return points;
    P m = points[0]; int mi = 0;
    int n = points.size();
    for (int i = 1; i < n; ++i) {
        if (points[i].imag() < m.imag() ||
           (points[i].imag() == m.imag() && points[i].real() < m.real())) {
            m = points[i];
            mi = i;
        }
    }   // m = spodnja leva

    swap(points[0], points[mi]);
    sort(points.begin()+1, points.end(),
         [&m](const P& a, const P& b) { return compare(a, b, m); });

    vector<P> hull;
    hull.push_back(points[0]);
    hull.push_back(points[1]);

    for (int i = 2; i < n; ++i) {  // tocke, ki so na ovojnici spusti, ce jih hoces daj -eps
        while (hull.size() >= 2 && cross(hull.end()[-2], hull.end()[-1], points[i]) < eps) {
            hull.pop_back();  // right turn
        }
        hull.push_back(points[i]);
    }

    return hull;
}
```

5.3 Ploščina unije pravokotnikov

Vhod: Seznam n pravokotnikov P_i danih s spodnjo levo in zgornjo desno točko.
Izhod: Ploščina unije danih pravokotnikov.
Časovna zahtevnost: $O(n \log n)$
Prostorska zahtevnost: $O(n)$
Testiranje na terenu: `http://putka.upm.si/competitions/upm2013-2/kolaz`

```
typedef complex<int> P;

struct vert {  // vertical sweep line element
    int x, s, e;
    bool start;
    vert(int a, int b, int c, bool d) : x(a), s(b), e(c), start(d) {}
    bool operator<(const vert& o) const {
        return x < o.x;
    }
};

vector<int> points;

struct Node {  // segment tree
    int s, e, m, c, a;  // start, end, middle, count, area
    Node *left, *right;
    Node(int s_, int e_) : s(s_), e(e_), m((s+e)/2), c(0), a(0), left(nullptr), right(nullptr) {
        if (e-s == 1) return;
        left = new Node(s, m);
        right = new Node(m, e);
    }
    int add(int f, int t) {  // returns area
```

```
        if (f <= s && e <= t) {
            c++;
            return a = points[e] - points[s];
        }
        if (f < m) left->add(f, t);
        if (t > m) right->add(f, t);
        if (c == 0) a = left->a + right->a;  // če nimam lastnega intervala, izračunaj
        return a;
    }
    int remove(int f, int t) {  // returns area
        if (f <= s && e <= t) {
            c--;
            if (c == 0) {  // če nima lastnega intervala
                if (left == nullptr) a = 0;  // če je list je area 0
                else a = left->a + right->a;  // če ne je vsota otrok
            }
            return a;
        }
        if (f < m) left->remove(f, t);
        if (t > m) right->remove(f, t);
        if (c == 0) a = left->a + right->a;
        return a;
    }
};

int rectangle_union_area(const vector<pair<P, P>>& rects) {
    int n = rects.size();

    vector<vert> verts; verts.reserve(2*n);
    points.resize(2*n);  // vse točke čez katere napenjamo intervale (stranice)

    P levo_spodaj, desno_zgoraj;  // pravokotniki so podani tako
    for (int i = 0; i < n; ++i) {
        tie(levo_spodaj, desno_zgoraj) = rects[i];
        int a = levo_spodaj.real();
        int c = desno_zgoraj.real();
        int b = levo_spodaj.imag();
        int d = desno_zgoraj.imag();
        verts.push_back(vert(a, b, d, true));
        verts.push_back(vert(c, b, d, false));
        points[2*i] = b;
        points[2*i+1] = d;
    }

    sort(verts.begin(), verts.end());
    sort(points.begin(), points.end());
    points.resize(unique(points.begin(), points.end())-points.begin());  // zbrišemo enake

    Node * sl = new Node(0, points.size());  // sweepline segment tree

    int area = 0, height = 0;  // area = total area. height = trenutno pokrita višina
    int px = -(1 << 30);
    for (int i = 0; i < 2*n; ++i) {
        area += (verts[i].x-px)*height;  // trenutno pometena area

        int s = lower_bound(points.begin(), points.end(), verts[i].s)-points.begin();
        int e = lower_bound(points.begin(), points.end(), verts[i].e)-points.begin();
        if (verts[i].start)
            height = sl->add(s, e);  // segment tree sprejme indexe, ne koordinat
        else
            height = sl->remove(s, e);
        px = verts[i].x;
    }

    return area;
}
```

5.4 Najbližji par točk v ravnini

Vhod: Seznam $n \geqslant 2$ točk v ravnini.

Izhod: Kvadrat razdalje med najbližjima točkama. Z lahkoto se prilagodi, da vrne tudi točki.

Časovna zahtevnost: $O(n \log n)$, nisem sure. . . :

Prostorska zahtevnost: $O(n \log n)$

Testiranje na terenu: UVa 10245

```cpp
typedef complex<double> P;
typedef vector<P>::iterator RAI;  // or use template

bool byx(const P& a, const P& b) { return a.real() < b.real(); }
bool byy(const P& a, const P& b) { return a.imag() < b.imag(); }

double najblizji_tocki_bf(RAI s, RAI e) {
    double m = numeric_limits<double>::max();
    for (RAI i = s; i != e; ++i)
        for (RAI j = i+1; j != e; ++j)
            m = min(m, norm(*i - *j));
    return m;
}
double najblizji_tocki_divide(RAI s, RAI e, const vector<P>& py) {
    if (e - s < 50) return najblizji_tocki_bf(s, e);

    size_t m = (e-s) / 2;
    double d1 = najblizji_tocki_divide(s, s+m, py);
    double d2 = najblizji_tocki_divide(s+m, e, py);
    double d = min(d1, d2);
    // merge
    double meja = (s[m].real() + s[m+1].real()) / 2;
    int n = py.size();
    for (double i = 0; i < n; ++i) {
        if (meja-d < py[i].real() && py[i].real() <= meja+d) {
            double j = i+1;
            double c = 0;
            while (j < n && c < 7) {  // navzdol gledamo le 7 ali dokler ni dlje od d
                if (meja-d < py[j].real() && py[j].real() <= meja+d) {
                    double nd = norm(py[j]-py[i]);
                    d = min(d, nd);
                    if (py[j].imag() - py[i].imag() > d) break;
                    ++c;
                }
                ++j;
            }
        }
    }
    return d;
}
double najblizji_tocki(const vector<P>& points) {
    vector<P> px = points, py = points;
    sort(px.begin(), px.end(), byx);
    sort(py.begin(), py.end(), byy);
    return najblizji_tocki_divide(px.begin(), px.end(), py);
}
```