

Codebook

Pitoni++

Žiga Gosar, Maks Kolman, Jure Slak

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

verzija: 25. marec 2015

Kazalo

1	Grafi	3
1.1	Topološko sortiranje	3
1.2	Najdaljša pot v DAGu	3
1.3	Mostovi in prerezna vozlišča grafa	4
1.4	Močno povezane komponente	5
1.5	Najkrajša pot v grafu	6
1.5.1	Dijkstra	6
1.5.2	Dijkstra (kvadratičen)	6
1.5.3	Bellman-Ford	7
1.5.4	Floyd-Warhsall	7
1.6	Minimalno vpeto drevo	8
1.6.1	Prim	8
1.6.2	Kruskal	8
1.7	Največji pretok in najmanjši prerez	9
1.7.1	Edmonds-Karp	9
1.8	Največje prirejanje in najmanjše pokritje	10
2	Podatkovne strukture	11
2.1	Fenwick tree	11
2.2	Fenwick tree (n -dim)	12
3	Teorija števil	12
3.1	Evklidov algoritem	12
3.2	Razširjen Evklidov algoritem	13
3.3	Kitajski izrek o ostankih	13
3.4	Hitro potenciranje	14
3.5	Številski sestavi	14
3.6	Eulerjeva funkcija ϕ	15
3.7	Eratostenovo rešeto	15
4	Geometrija	16
4.1	Osnove	16
4.2	Konveksna ovojnica	19
4.3	Ploščina unije pravokotnikov	20
4.4	Najbližji par točk v ravnini	21

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

```
1  vector<int> topological_sort(const vector<vector<int>>& graf) {
2      int n = graf.size();
3      vector<int> ingoing(n, 0);
4      for (int i = 0; i < n; ++i)
5          for (const auto& u : graf[i])
6              ingoing[u]++;
7
8      queue<int> q; // morda priority_queue, če je vrstni red pomemben
9      for (int i = 0; i < n; ++i)
10         if (ingoing[i] == 0)
11             q.push(i);
12
13     vector<int> res;
14     while (!q.empty()) {
15         int t = q.front();
16         q.pop();
17         res.push_back(t);
18
19         for (int v : graf[t])
20             if (--ingoing[v] == 0)
21                 q.push(v);
22     }
23
24     return res; // če res.size() != n, ima graf cikle.
25 }
26 }
```

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 ($\max \rightarrow \min$).

Časovna zahtevnost: $O(V + E)$

Prostorska zahtevnost: $O(V + E)$

Testiranje na terenu: UVa 103

```
1  int longest_path_in_a_dag(const vector<vector<pair<int, int>>>& graf, int s, int t) {
2      int n = graf.size(), v, w;
3      vector<int> ind(n, 0);
4      vector<int> max_dist(n, -1);
5      for (int i = 0; i < n; ++i)
6          for (const auto& edge : graf[i])
7              ind[edge.first]++;
8
9      max_dist[s] = 0;
10
11     queue<int> q;
12     for (int i = 0; i < n; ++i)
13         if (ind[i] == 0)
```

```

14         q.push(i); // topološko uredimo in gledamo maksimum
15
16     while (!q.empty()) {
17         int u = q.front();
18         q.pop();
19
20         for (const auto& edge : graf[u]) {
21             tie(v, w) = edge;
22             if (max_dist[u] >= 0) // da začnemo pri s-ju, sicer bi začeli na začetku, vsi pred s -1
23                 max_dist[v] = max(max_dist[v], max_dist[u] + w); // min za shortest path
24             if (--ind[v] == 0) q.push(v);
25         }
26     }
27     return max_dist[t];
28 }

```

1.3 Mostovi in prerezna vozlišča grafa

Vhod: Število vozlišč n in število povezav m ter seznam povezav E oblike $u \rightarrow v$ 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

```

1  namespace {
2      vector<int> low;
3      vector<int> dfs_num;
4      vector<int> parent;
5  }
6
7  void articulation_points_and_bridges_internal(int u, const vector<vector<int>>& graf,
8      vector<bool>& articulation_points_map, vector<pair<int, int>>& bridges) {
9      static int dfs_num_counter = 0;
10     low[u] = dfs_num[u] = ++dfs_num_counter;
11     int children = 0;
12     for (int v : graf[u]) {
13         if (dfs_num[v] == -1) { // unvisited
14             parent[v] = u;
15             children++;
16
17             articulation_points_and_bridges_internal(v, graf, articulation_points_map, bridges);
18             low[u] = min(low[u], low[v]); // update low[u]
19
20             if (parent[u] == -1 && children > 1) // special root case
21                 articulation_points_map[u] = true;
22             else if (parent[u] != -1 && low[v] >= dfs_num[u]) // articulation point
23                 articulation_points_map[u] = true; // assigned more than once
24             if (low[v] > dfs_num[u]) // bridge
25                 bridges.push_back({u, v});
26         } else if (v != parent[u]) {
27             low[u] = min(low[u], dfs_num[v]); // update low[u]
28         }
29     }
30 }
31
32 void articulation_points_and_bridges(int n, int m, const int E[][2],
33     vector<int>& articulation_points, vector<pair<int, int>>& bridges) {
34     vector<vector<int>> graf(n);
35     for (int i = 0; i < m; ++i) {
36         int a = E[i][0], b = E[i][1];
37         graf[a].push_back(b);
38         graf[b].push_back(a);
39     }
40
41     low.assign(n, -1);
42     dfs_num.assign(n, -1);
43     parent.assign(n, -1);
44 }

```

```

45     vector<bool> articulation_points_map(n, false);
46     for (int i = 0; i < n; ++i)
47         if (dfs_num[i] == -1)
48             articulation_points_and_bridges_internal(i, graf, articulation_points_map, bridges);
49
50     for (int i = 0; i < n; ++i)
51         if (articulation_points_map[i])
52             articulation_points.push_back(i); // actually return only articulation points
53 }

```

1.4 Močno povezane komponente

Vhod: Seznam sosednosti s težami povezav.

Izhod: Seznam povezanih komponent grafa v obratni topološki ureditvi in kvoci-
entni 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

```

1  namespace {
2  vector<int> low;
3  vector<int> dfs_num;
4  stack<int> S;
5  vector<int> component; // maps vertex to its component
6  }
7
8  void strongly_connected_components_internal(int u, const vector<vector<pair<int, int>>& graf,
9      vector<vector<int>>& comps) {
10     static int dfs_num_counter = 1;
11     low[u] = dfs_num[u] = dfs_num_counter++;
12     S.push(u);
13
14     for (const auto& v : graf[u]) {
15         if (dfs_num[v.first] == 0) // not visited yet
16             strongly_connected_components_internal(v.first, graf, comps);
17         if (dfs_num[v.first] != -1) // not popped yet
18             low[u] = min(low[u], low[v.first]);
19     }
20
21     if (low[u] == dfs_num[u]) { // extract the component
22         int cnum = comps.size();
23         comps.push_back({}); // start new component
24         int w;
25         do {
26             w = S.top(); S.pop();
27             comps.back().push_back(w);
28             component[w] = cnum;
29             dfs_num[w] = -1; // mark popped
30         } while (w != u);
31     }
32 }
33
34 void strongly_connected_components(const vector<vector<pair<int, int>>& graf,
35     vector<vector<int>>& comps, vector<map<int, int>>& dag) {
36     int n = graf.size();
37     low.assign(n, 0);
38     dfs_num.assign(n, 0);
39     component.assign(n, -1);
40
41     for (int i = 0; i < n; ++i)
42         if (dfs_num[i] == 0)
43             strongly_connected_components_internal(i, graf, comps);
44
45     dag.resize(comps.size()); // zgradimo kvocietni graf, teza povezave je vsota tez
46     for (int u = 0; u < n; ++u) {
47         for (const auto& v : graf[u]) {
48             if (component[u] != component[v.first]) {
49                 dag[component[u]][component[v.first]] += v.second; // ali max, kar zahteva naloga
50             }
51         }
52     }
53 }

```

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

```
1  typedef pair<int, int> pii;
2
3  int dijkstra(const vector<vector<pii>>& graf, int s, int t) {
4      int n = graf.size(), d, u;
5      priority_queue<pii, vector<pii>, greater<pii>> q;
6      vector<bool> visited(n, false);
7      vector<int> dist(n);
8
9      q.push({0, s}); // {cena, točka}
10     while (!q.empty()) {
11         tie(d, u) = q.top();
12         q.pop();
13
14         if (visited[u]) continue;
15         visited[u] = true;
16         dist[u] = d;
17
18         if (u == t) break; // ce iscemo do vseh točk spremeni v --n == 0
19
20         for (const auto& p : graf[u])
21             if (!visited[p.first])
22                 q.push({d + p.second, p.first});
23     }
24     return dist[t];
25 }
```

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

```
1  vector<int> dijkstra_square(const vector<vector<pair<int, int>>>& graf, int s, int t) {
2      int INF = numeric_limits<int>::max();
3      int n = graf.size(), to, len;
4      vector<int> dist(n, INF), prev(n);
5      dist[s] = 0;
6      vector<bool> visited(n, false);
7      for (int i = 0; i < n; ++i) {
8          int u = -1;
9          for (int j = 0; j < n; ++j)
10             if (!visited[j] && (u == -1 || dist[j] < dist[u]))
11                 u = j; // vertex with minimum dist
12             if (u == -1 || dist[u] == INF) break; // disconnected graph
13             if (u == t) break; // found shortest path to target
14             visited[u] = true;
15
16             for (const auto& edge : graf[u]) {
17                 tie(to, len) = edge;
18                 if (dist[u] + len < dist[to]) { // if path can be improved via me
19                     dist[to] = dist[u] + len;
```

```

20         prev[to] = u;
21     }
22 }
23 } // v dist so sedaj razdalje od s do vseh, ki so bližje kot t (in t)
24 vector<int> path; // ce je dist[t] == INF, je t v drugi komponenti kot s
25 for (int v = t; v != s; v = prev[v])
26     path.push_back(v);
27 path.push_back(s);
28 reverse(path.begin(), path.end());
29 return path;
30 }

```

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

```

1  vector<int> bellman_ford(const vector<vector<pair<int, int>>>& graf, int s) {
2      int INF = numeric_limits<int>::max();
3      int n = graf.size(), v, w;
4      vector<int> dist(n, INF);
5      vector<int> prev(n, -1);
6      vector<bool> visited(n, false);
7
8      dist[s] = 0;
9      for (int i = 0; i < n-1; ++i) { // i je trenutna dolžina poti
10         for (int u = 0; u < n; ++u) {
11             for (const auto& edge : graf[u]) {
12                 tie(v, w) = edge;
13                 if (dist[u] != INF && dist[u] + w < dist[v]) {
14                     dist[v] = dist[u] + w;
15                     prev[v] = u;
16                 }
17             }
18         }
19     }
20
21     for (int u = 0; u < n; ++u) { // cycle detection
22         for (const auto& edge : graf[u]) {
23             tie(v, w) = edge;
24             if (dist[u] != INF && dist[u] + w < dist[v])
25                 return {}; // graph has a negative cycle !!
26         }
27     }
28     return dist;
29 }

```

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

```

1  vector<vector<int>> floyd_warshall(int n, int m, const int E[][3]) {
2      int INF = numeric_limits<int>::max();
3      vector<vector<int>> d(n, vector<int>(n, INF));
4      // vector<vector<int>> next(n, vector<int>(n, -1)); // da dobimo pot
5      for (int i = 0; i < m; ++i) {
6          int u = E[i][0], v = E[i][1], c = E[i][2];
7          d[u][v] = c;
8          // next[u][v] = v
9      }
10
11     for (int i = 0; i < n; ++i)
12         d[i][i] = 0;
13
14     for (int k = 0; k < n; ++k)
15         for (int i = 0; i < n; ++i)
16             for (int j = 0; j < n; ++j)
17                 if (d[i][k] != INF && d[k][j] != INF && d[i][k] + d[k][j] < d[i][j])
18                     d[i][j] = d[i][k] + d[k][j];
19                     // next[i][j] = next[i][k];
20     return d; // ce je kateri izmed d[i][i] < 0, ima graf negativen cikel
21 }

```

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

```

1  typedef pair<int, int> pii;
2
3  int prim_minimal_spanning_tree(const vector<vector<pii>>& graf) {
4      int n = graf.size(), d, u;
5      vector<bool> visited(n, false);
6      priority_queue<pii, vector<pii>, greater<pii>> q; // remove greater for max-tree
7      q.push({0, 0});
8
9      int sum = 0; // sum of the mst
10     int edge_count = 0; // stevilo dodanih povezav
11     while (!q.empty()) {
12         tie(d, u) = q.top();
13         q.pop();
14
15         if (visited[u]) continue;
16         visited[u] = true;
17
18         sum += d;
19         if (++edge_count == n) break; // drevo, jebeš solato
20
21         for (const auto& edge : graf[u])
22             if (!visited[edge.first])
23                 q.push({edge.second, edge.first});
24     } // ce zelimo drevo si shranjujemo se previous vertex.
25     return sum;
26 }

```

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

```
1 namespace {
2 vector<int> parent;
3 vector<int> rank;
4 }
5
6 int find(int x) {
7     if (parent[x] != x)
8         parent[x] = find(parent[x]);
9     return parent[x];
10 }
11
12 bool unija(int x, int y) {
13     int xr = find(x);
14     int yr = find(y);
15
16     if (xr == yr) return false;
17     if (rank[xr] < rank[yr]) { // rank lahko tudi izpustimo, potem samo parent[xr] = yr;
18         parent[xr] = yr;
19     } else if (rank[xr] > rank[yr]) {
20         parent[yr] = xr;
21     } else {
22         parent[yr] = xr;
23         rank[xr]++;
24     }
25     return true;
26 }
27
28 int kruskal_minimal_spanning_tree(int n, int m, int E[][3]) {
29     rank.assign(n, 0);
30     parent.assign(n, 0);
31     for (int i = 0; i < n; ++i) parent[i] = i;
32     vector<tuple<int, int, int>> edges;
33     for (int i = 0; i < m; ++i) edges.emplace_back(E[i][2], E[i][0], E[i][1]);
34     sort(edges.begin(), edges.end());
35
36     int sum = 0, a, b, c, edge_count = 0;
37     for (int i = 0; i < m; ++i) {
38         tie(c, a, b) = edges[i];
39         if (unija(a, b)) {
40             sum += c;
41             edge_count++;
42         }
43         if (edge_count == n - 1) break;
44     }
45     return sum;
46 }
```

1.7 Največji pretok in najmanjši prerez

1.7.1 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

```
1 namespace {
2 const int INF = numeric_limits<int>::max();
3 struct triple { int u, p, m; };
4 }
5
6 int edmonds_karp_maximal_flow(const vector<vector<int>>& capacity, int s, int t) {
7     int n = capacity.size();
8     vector<vector<int>> flow(n, vector<int>(n, 0));
9     int maxflow = 0;
10    while (true) {
11        vector<int> prev(n, -2); // hkrati tudi visited array
```

```

12     int bot = INF; // bottleneck
13     queue<triple> q;
14     q.push({s, -1, INF});
15     while (!q.empty()) { // compute a possible path, add its bottleneck to the total flow
16         int u = q.front().u, p = q.front().p, mini = q.front().m; // while such path exists
17         q.pop();
18
19         if (prev[u] != -2) continue;
20         prev[u] = p;
21
22         if (u == t) { bot = mini; break; }
23
24         for (int i = 0; i < n; ++i) {
25             int available = capacity[u][i] - flow[u][i];
26             if (available > 0) {
27                 q.push({i, u, min(available, mini)}); // kumulativni minimum
28             }
29         }
30     }
31
32     if (prev[t] == -2) break;
33
34     maxflow += bot;
35     for (int u = t; u != s; u = prev[u]) { // popravimo trenutni flow nazaj po poti
36         flow[u][prev[u]] -= bot;
37         flow[prev[u]][u] += bot;
38     }
39 }
40 return maxflow;
41 }

```

1.8 Največje prirejanje in najmanjše pokritje

V angleščini: *maximum cardinality bipartite matching* (če bi dodali še kakšno pove-zavo 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

```

1  namespace {
2  vector<int> match, vis;
3  }
4
5  int augmenting_path(const vector<vector<int>>& graf, int left) {
6      if (vis[left]) return 0;
7      vis[left] = 1;
8      for (int right : graf[left]) {
9          if (match[right] == -1 || augmenting_path(graf, match[right])) {
10             match[right] = left;
11             match[left] = right;
12             return 1;
13         }
14     }
15     return 0;
16 }
17
18 void mark_vertices(const vector<vector<int>>& graf, vector<bool>& cover, int v) {
19     if (vis[v]) return;
20     vis[v] = 1;
21     cover[v] = false;
22     for (int r : graf[v]) {
23         cover[r] = true;
24         if (match[r] != -1)

```

```

25         mark_vertices(graf, cover, match[r]);
26     }
27 }
28
29 int bipartite_matching(const vector<vector<int>>& graf, int left_num) {
30     int n = graf.size();
31     match.assign(2*n, -1);
32     int mcbm = 0; // prvih left_num je v levem delu grafa
33     for (int left = 0; left < left_num; ++left) {
34         vis.assign(n, 0);
35         mcbm += augmenting_path(graf, left);
36     }
37     return mcbm;
38 }
39
40 vector<int> minimal_cover(const vector<vector<int>>& graf, int left_num) {
41     bipartite_matching(graf, left_num);
42     int n = graf.size();
43     vis.assign(2*n, 0);
44     vector<bool> cover(n, false);
45     fill(cover.begin(), cover.begin() + left_num, true);
46     for (int left = 0; left < n; ++left)
47         if (match[left] == -1)
48             mark_vertices(graf, cover, left);
49
50     vector<int> result; // ni potrebno, lahko se uporablja kar cover
51     for (int i = 0; i < n; ++i)
52         if (cover[i])
53             result.push_back(i);
54     return result;
55 }

```

2 Podatkovne strukture

2.1 Fenwick tree

Operacije: Imamo tabelo z indeksi $1 \leq x \leq 2^k$ v kateri hranimo števila. Želimo hitro posodabljanje 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>

```

1  namespace {
2  const int MAX_INDEX = 16;
3  vector<int> tree(MAX_INDEX+1, 0); // global tree, 1 based!!
4  }
5
6  void update(int idx, int val) { // increments idx for value
7      while (idx <= MAX_INDEX) {
8          tree[idx] += val;
9          idx += (idx & -idx);
10     }
11 }
12
13 int read(int idx) { // read sum of [1, x], read(0) == 0, duh.
14     int sum = 0;
15     while (idx > 0) {
16         sum += tree[idx];
17         idx -= (idx & -idx);
18     }
19     return sum;
20 }
21
22 int readSingle(int idx) { // read a single value, readSingle(x) == read(x)-read(x-1)
23     int sum = tree[idx];

```

```

24     if (idx > 0) {
25         int z = idx - (idx & -idx);
26         idx--;
27         while (idx != z) {
28             sum += tree[idx];
29             idx -= (idx & -idx);
30         }
31     }
32     return sum;
33 }

```

2.2 Fenwick tree (n-dim)

Operacije: Imamo n -dim tabelo dimenzij $d_1 \times d_2 \times \dots \times d_n$ z zero-based indeksi v kateri hranimo števila. Želimo hitro posodablјati elemente in odgovarјati 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 $O(\log(d_1 + \dots + d_n))$, za vsoto podkvadra $O(2^d \log(d_1 + \dots + d_n))$.

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

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

```

1  typedef vector<vector<vector<int>>> vvv;
2
3  int sum(int x, int y, int z, const vvv& tree) { // [0,0,0 - x,y,z] vključno
4      int result = 0;
5      for (int i = x; i >= 0; i = (i & (i+1)) - 1)
6          for (int j = y; j >= 0; j = (j & (j+1)) - 1)
7              for (int k = z; k >= 0; k = (k & (k+1)) - 1)
8                  result += tree[i][j][k];
9      return result;
10 }
11
12 void inc(int x, int y, int z, int delta, vvv& tree) { // povečaj na koordinatah, 0 based
13     int n = tree.size(); // lahko so tudi različni n-ji za posamezno dimenzijo
14     for (int i = x; i < n; i |= i+1)
15         for (int j = y; j < n; j |= j+1)
16             for (int k = z; k < n; k |= k+1)
17                 tree[i][j][k] += delta;
18 }
19
20 int subsum(int x1, int y1, int z1,
21            int x2, int y2, int z2, const vvv& tree) { // vsota na [x1,y1,z1 - x2,y2,z2], vključno
22     x1--; y1--; z1--;
23     return sum(x2, y2, z2, tree) -
24            sum(x1, y2, z2, tree) -
25            sum(x2, y1, z2, tree) - // pravilo vključitev in izključitev
26            sum(x2, y2, z1, tree) +
27            sum(x1, y1, z2, tree) +
28            sum(x1, y2, z1, tree) +
29            sum(x2, y1, z1, tree) -
30            sum(x1, y1, z1, tree);
31 }

```

3 Teorija števil

3.1 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: $O(1)$

```

1  int gcd(int a, int b) {
2      int t;
3      while (b != 0) {
4          t = a % b;
5          a = b;
6          b = t;
7      }
8      return a;
9  }
```

3.2 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 \mathbb{Z}_b^* .

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

Prostorska zahtevnost: $O(1)$

Testiranje na terenu: UVa 756

```

1  int ext_gcd(int a, int b, int& retx, int& rety) {
2      int x = 0, px = 1, y = 1, py = 0, r, q;
3      while (b != 0) {
4          r = a % b; q = a / b; // quotient and remainder
5          a = b; b = r;        // gcd swap
6          r = px - q * x;      // x swap
7          px = x; x = r;
8          r = py - q * y;      // y swap
9          py = y; y = r;
10     }
11     retx = px; rety = py;    // return
12     return a;
13 }
```

3.3 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

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

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. 12)

Testiranje na terenu: UVa 756

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

```

1  int mul_inverse(int a, int m) {
2      int x, y;
3      ext_gcd(a, m, x, y);
4      return (x + m) % m;
5  }
6
7  int chinese_reminder_theorem(const vector<pair<int, int>>& cong) {
8      int M = 1;
9      for (size_t i = 0; i < cong.size(); ++i) {
10         M *= cong[i].second;
11     }
12     int x = 0, a, m;
13     for (const auto& p : cong) {
14         tie(a, m) = p;
15         x += a * M / m * mul_inverse(M/m, m);
16         x %= M;
17     }
18     return (x + M) % M;
19 }

```

3.4 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

```

1  int fast_power(int g, int n) {
2      int r = 1;
3      while (n > 0) {
4          if (n & 1) {
5              r *= g;
6          }
7          g *= g;
8          n >>= 1;
9      }
10     return r;
11 }

```

3.5 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 števki 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.

```

1  char STEVILSKI_SESTAVI_ZNAKI[] = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
2
3  string convert_int(int n, int baza) {
4      if (n == 0) return "0";
5      string result;
6      while (n > 0) {
7          result.push_back(STEVILSKI_SESTAVI_ZNAKI[n % baza]);
8          n /= baza;
9      }
10     reverse(result.begin(), result.end());
11     return result;
12 }
13
14 string convert_fraction(int stevec, int imenovalec, int base) {
15     div_t d = div(stevec, imenovalec);
16     string result = convert_int(d.quot, base);

```

```

17     if (d.rem == 0) return result;
18
19     string decimalke; // decimalni del
20     result.push_back('.');
21     int mesto = 0;
22     map<int, int> spomin;
23     spomin[d.rem] = mesto;
24     while (d.rem != 0) { // pisno deljenje
25         mesto++;
26         d.rem *= base;
27         decimalke += STEVILSKI_SESTAVI_ZNAKI[d.rem / imenovalec];
28         d.rem %= imenovalec;
29         if (spomin.count(d.rem) > 0) { // periodično
30             result.append(decimalke.begin(), decimalke.begin() + spomin[d.rem]);
31             result.push_back('.');
32             result.append(decimalke.begin() + spomin[d.rem], decimalke.end());
33             result.push_back('.');
34             return result;
35         }
36         spomin[d.rem] = mesto;
37     }
38     result += decimalke;
39     return result; // končno decimalno stevilo
40 }

```

3.6 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:

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

Časovna zahtevnost: $O(\sqrt{n})$

Prostorska zahtevnost: $O(1)$

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

```

1  int euler_phi(int n) {
2      int res = n;
3      for (int i = 2; i*i <= n; ++i) {
4          if (n % i == 0) {
5              while (n % i == 0) {
6                  n /= i;
7              }
8              res -= res / i;
9          }
10     }
11     if (n > 1) res -= res / n;
12     return res;
13 }

```

3.7 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 praštevilstosti.

Časovna zahtevnost: $O(n \log(n))$

Prostorska zahtevnost: $O(n)$

Testiranje na terenu: UVa 10394

```

1  void eratosthenes_sieve(int n, vector<int>& is_prime, vector<int>& primes) {
2      is_prime.resize(n);
3      for (int i = 2; i < n+1; ++i) {

```

```

4         if (is_prime[i] == 0) {
5             is_prime[i] = i;
6             primes.push_back(i);
7         }
8         size_t j = 0;
9         while (j < primes.size() && primes[j] <= is_prime[i] && i * primes[j] <= n) {
10             is_prime[i * primes[j]] = primes[j];
11             j++;
12         }
13     }
14 }

```

4 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.

4.1 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: $O(\text{št. točk})$

Prostorska zahtevnost: $O(\text{št. točk})$

Testiranje na terenu: Bolj tako, ima pa obsežne unit teste...

```

1  const double pi = M_PI;
2  const double eps = 1e-7;
3  const double inf = numeric_limits<double>::infinity();
4
5  enum ITYPE : char { OK, NO, EQ };
6  typedef complex<double> P;
7
8  double dot(const P& p, const P& q) {
9      return p.real() * q.real() + p.imag() * q.imag();
10 }
11 double cross(const P& p, const P& q) {
12     return p.real() * q.imag() - p.imag() * q.real();
13 }

```



```

14 double cross(const P& p, const P& q, const P& r) {
15     return cross(q - p, r - q); // > 0 levo, < 0 desno, = 0 naravnost
16 }
17 // true is p->q->r is a left turn, straight line is not, if so, change to -eps
18 bool left_turn(const P& p, const P& q, const P& r) {
19     return cross(q-p, r-q) > eps;
20 }
21 P perp(const P& p) { // get left perpendicular vector
22     return P(-p.imag(), p.real());
23 }
24 int sign(double x) {
25     if (x < -eps) return -1;
26     if (x > eps) return 1;
27     return 0;
28 }
29 double polar_angle(const P& p) { // phi in [0, 2pi) or -1 for (0,0)
30     if (p == P(0, 0)) return -1;
31     double a = arg(p);
32     if (a < 0) a += 2*pi;
33     return a;
34 }
35 double area(const P& a, const P& b, const P& c) { // signed
36     return 0.5 * cross(a, b, c);
37 }
38 double area(const vector<P>& poly) { // signed
39     double A = 0;
40     int n = poly.size();
41     for (int i = 0; i < n; ++i) {
42         int j = (i+1) % n;
43         A += cross(poly[i], poly[j]);
44     }
45     return A/2;
46 }
47 // struct L { // premica, dana z enacbo ax + by = c ali z dvema točkama
48 //     double a, b, c; // lahko tudi int
49 L::L() : a(0), b(0), c(0) {}
50 L::L(int A, int B, int C) {
51     if (A < 0 || (A == 0 && B < 0)) a = -A, b = -B, c = -C;
52     else a = A, b = B, c = C;
53     int d = gcd(gcd(abs(a), abs(b)), abs(c)); // same sign as A, if nonzero, else B, else C
54     if (d == 0) d = 1; // in case of 0 0 0 input
55     a /= d;
56     b /= d;
57     c /= d;
58 }
59 L::L(double A, double B, double C) {
60     if (A < 0 || (A == 0 && B < 0)) a = -A, b = -B, c = -C;
61     else a = A, b = B, c = C;
62 }
63 L::L(const P& p, const P& q) : L(imag(q-p), real(p-q), cross(p, q)) {}
64 P L::normal() const { return {a, b}; }
65 double L::value(const P& p) const { return dot(normal(), p) - c; }
66 bool L::operator<(const L& line) const {
67     if (a == line.a) {
68         if (b == line.b) return c < line.c;
69         return b < line.b;
70     }
71     return a < line.a;
72 }
73 bool L::operator==(const L& line) const {
74     return cross(normal(), line.normal()) < eps && c*line.b == b*line.c;
75 }
76 // }; // end struct L
77 ostream& operator<<(ostream& os, const L& line) {
78     os << line.a << "x + " << line.b << "y == " << line.c; return os;
79 }
80
81 double dist_to_line(const P& p, const L& line) {
82     return abs(line.value(p)) / abs(line.normal());
83 }
84 double dist_to_line(const P& t, const P& p1, const P& p2) { // t do premice p1p2
85     return abs(cross(p2-p1, t-p1)) / abs(p2-p1);
86 }
87 double dist_to_segment(const P& t, const P& p1, const P& p2) { // t do daljice p1p2
88     P s = p2 - p1;
89     P w = t - p1;
90     double c1 = dot(s, w);
91     if (c1 <= 0) return abs(w);
92     double c2 = norm(s);
93     if (c2 <= c1) return abs(t-p2);
94     return dist_to_line(t, p1, p2);

```

```

95 }
96 double great_circle_dist(const P& a, const P& b) { // pairs of (latitude, longitude) in radians
97     double R = 6371.0; // compute great circle distance
98     double u[3] = { cos(a.real()) * sin(a.imag()), cos(a.real()) * cos(a.imag()), sin(a.real()) };
99     double v[3] = { cos(b.real()) * sin(b.imag()), cos(b.real()) * cos(b.imag()), sin(b.real()) };
100     double dot = u[0]*v[0] + u[1]*v[1] + u[2]*v[2];
101     bool flip = false;
102     if (dot < 0.0) {
103         flip = true;
104         for (int i = 0; i < 3; i++) v[i] = -v[i];
105     }
106     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] };
107     double theta = asin(sqrt(cr[0]*cr[0] + cr[1]*cr[1] + cr[2]*cr[2]));
108     double len = theta * R;
109     if (flip) len = pi * R - len;
110     return len;
111 }
112 bool point_in_rect(const P& t, const P& p1, const P& p2) { // ali je t v pravokotniku p1p2
113     return min(p1.real(), p2.real()) <= t.real() && t.real() <= max(p1.real(), p2.real()) &&
114         min(p1.imag(), p2.imag()) <= t.imag() && t.imag() <= max(p1.imag(), p2.imag());
115 }
116 bool point_in_triangle(const P& t, const P& a, const P& b, const P& c) { // orientation independant
117     return abs(abs(area(a, b, t)) + abs(area(a, c, t)) + abs(area(b, c, t)) // edge inclusive
118         - abs(area(a, b, c))) < eps;
119 }
120 pair<ITYPE, P> line_line_intersection(const L& p, const L& q) {
121     double det = cross(p.normal(), q.normal()); // če imata odvisni normali (ali smerna vektorja)
122     if (abs(det) < eps) { // paralel
123         if (abs(p.b*q.c - p.c*q.b) < eps && abs(p.a*q.c - p.c*q.a) < eps) {
124             return {EQ, P()}; // razmerja koeficientov se ujemajo
125         } else {
126             return {NO, P()};
127         }
128     } else {
129         return {OK, P(q.b*p.c - p.b*q.c, p.a*q.c - q.a*p.c) / det};
130     }
131 }
132 pair<ITYPE, P> line_segment_intersection(const L& p, const P& u, const P& v) {
133     double u_on = p.value(u);
134     double v_on = p.value(v);
135     if (abs(u_on) < eps && abs(v_on) < eps) return {EQ, u};
136     if (abs(u_on) < eps) return {OK, u};
137     if (abs(v_on) < eps) return {OK, v};
138     if ((u_on > eps && v_on < -eps) || (u_on < -eps && v_on > eps)) {
139         return line_line_intersection(p, L(u, v));
140     }
141     return {NO, P()};
142 }
143 pair<ITYPE, P> segment_segment_intersection(const P& p1, const P& p2, const P& q1, const P& q2) {
144     int o1 = sign(cross(p1, p2, q1)); // daljico p1p1 sekamo z q1q2
145     int o2 = sign(cross(p1, p2, q2));
146     int o3 = sign(cross(q1, q2, p1));
147     int o4 = sign(cross(q1, q2, p2));
148
149     // za pravo presečišče morajo biti o1, o2, o3, o4 != 0
150     // vemo da presečišče obstaja, tudi ce veljata samo prva dva pogoja
151     if (o1 != o2 && o3 != o4 && o1 != 0 && o2 != 0 && o3 != 0 && o4 != 0)
152         return line_line_intersection(L(p1, p2), L(q1, q2));
153
154     // EQ = se dotika samo z oglišcem ali sta vzporedni
155     if (o1 == 0 && point_in_rect(q1, p1, p2)) return {EQ, q1}; // q1 lezi na p
156     if (o2 == 0 && point_in_rect(q2, p1, p2)) return {EQ, q2}; // q2 lezi na p
157     if (o3 == 0 && point_in_rect(p1, q1, q2)) return {EQ, p1}; // p1 lezi na q
158     if (o4 == 0 && point_in_rect(p2, q1, q2)) return {EQ, p2}; // p2 lezi na q
159
160     return {NO, P()};
161 }
162 ITYPE point_in_poly(const P& t, const vector<P>& poly) {
163     int n = poly.size();
164     int cnt = 0;
165     double x2 = rand() % 100;
166     double y2 = rand() % 100;
167     P dalec(x2, y2);
168     for (int i = 0; i < n; ++i) {
169         int j = (i+1) % n;
170         if (dist_to_segment(t, poly[i], poly[j]) < eps) return EQ; // boundary
171         ITYPE tip = segment_segment_intersection(poly[i], poly[j], t, dalec).first;
172         if (tip != NO) cnt++; // ne testiramo, ali smo zadeli oglišce, upamo da nismo
173     }
174     if (cnt % 2 == 0) return NO;
175     else return OK;

```

```

176 }
177 pair<P, double> get_circle(const P& p, const P& q, const P& r) { // circle through 3 points
178     P v = q-p;
179     P w = q-r;
180     if (abs(cross(v, w)) < eps) return {P(), 0};
181     P x = (p+q)/2.0, y = (q+r)/2.0;
182     ITYPE tip;
183     P intersection;
184     tie(tip, intersection) = line_line_intersection(L(x, x+perp(v)), L(y, y+perp(w)));
185     return {intersection, abs(intersection-p)};
186 }
187 // circle through 2 points with given r, to the left of pq
188 P get_circle(const P& p, const P& q, double r) {
189     double d = norm(p-q);
190     double h = r*r / d - 0.25;
191     if (h < 0) return P(inf, inf);
192     h = sqrt(h);
193     return (p+q) / 2.0 + h * perp(q-p);
194 }

```

4.2 Konveksna ovojnica

Vhod: Seznam n točk.

Izhod: Najkrajši seznam h točk, ki napenja 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. 16.

Testiranje na terenu: UVa 681

```

1  typedef complex<double> P; // ali int
2  double eps = 1e-9;
3
4  bool compare(const P& a, const P& b, const P& m) {
5      double det = cross(a, m, b);
6      if (abs(det) < eps) return abs(a-m) < abs(b-m);
7      return det < 0;
8  }
9
10 vector<P> convex_hull(vector<P>& points) { // vector is modified
11     if (points.size() <= 2) return points;
12     P m = points[0]; int mi = 0;
13     int n = points.size();
14     for (int i = 1; i < n; ++i) {
15         if (points[i].imag() < m.imag() ||
16             (points[i].imag() == m.imag() && points[i].real() < m.real())) {
17             m = points[i];
18             mi = i;
19         }
20     } // m = spodnja leva
21
22     swap(points[0], points[mi]);
23     sort(points.begin()+1, points.end(),
24         [&m](const P& a, const P& b) { return compare(a, b, m); });
25
26     vector<P> hull;
27     hull.push_back(points[0]);
28     hull.push_back(points[1]);
29
30     for (int i = 2; i < n; ++i) { // tocke, ki so na ovojnici spusti, ce jih hoces daj -eps
31         while (hull.size() >= 2 && cross(hull.end()[-2], hull.end()[-1], points[i]) < eps) {
32             hull.pop_back(); // right turn
33         }
34         hull.push_back(points[i]);
35     }
36
37     return hull;
38 }

```

4.3 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>

```
1  typedef complex<int> P;
2
3  struct vert { // vertical sweep line element
4      int x, s, e;
5      bool start;
6      vert(int a, int b, int c, bool d) : x(a), s(b), e(c), start(d) {}
7      bool operator<(const vert& o) const {
8          return x < o.x;
9      }
10 };
11
12 vector<int> points;
13
14 struct Node { // segment tree
15     int s, e, m, c, a; // start, end, middle, count, area
16     Node *left, *right;
17     Node(int s_, int e_) : s(s_), e(e_), m((s+e)/2), c(0), a(0), left(NULL), right(NULL) {
18         if (e-s == 1) return;
19         left = new Node(s, m);
20         right = new Node(m, e);
21     }
22     int add(int f, int t) { // returns area
23         if (s >= f && e <= t) {
24             c++;
25             return a = points[e] - points[s];
26         }
27         if (f < m) left->add(f, t);
28         if (t > m) right->add(f, t);
29         if (c == 0) a = left->a + right->a; // če nimam lastnega intervala, izračunaj
30         return a;
31     }
32     int remove(int f, int t) { // returns area
33         if (s >= f && e <= t) {
34             c--;
35             if (c == 0) { // če nima lastnega intervala
36                 if (left == NULL) a = 0; // če je otrok je area 0
37                 else a = left->a + right->a; // če ne je vsota otrok
38             }
39             return a;
40         }
41         if (f < m) left->remove(f, t);
42         if (t > m) right->remove(f, t);
43         if (c == 0) a = left->a + right->a;
44         return a;
45     }
46 };
47
48 int rectangle_union_area(const vector<pair<P, P>>& rects) {
49     int n = rects.size();
50
51     vector<vert> verts; verts.reserve(2*n);
52     points.resize(2*n); // vse točke čez katere napenjamo intervale (stranice)
53
54     P levo_spodaj, desno_zgoraj; // pravokotniki so podani tako
55     for (int i = 0; i < n; ++i) {
56         tie(levo_spodaj, desno_zgoraj) = rects[i];
57         int a = levo_spodaj.real();
58         int c = desno_zgoraj.real();
59         int b = levo_spodaj.imag();
60         int d = desno_zgoraj.imag();
61         verts.push_back(vert(a, b, d, true));
62         verts.push_back(vert(c, b, d, false));
63         points[2*i] = b;
64         points[2*i+1] = d;
65     }
66
67     sort(verts.begin(), verts.end());
68     sort(points.begin(), points.end());
69     points.resize(unique(points.begin(), points.end())-points.begin()); // zberemo enake
```

```

70
71     Node * sl = new Node(0, points.size()); // sweepline segment tree
72
73     int area = 0, height = 0; // area = total area. height = trenutno pokrita višina
74     int px = -(1 << 30);
75     for (int i = 0; i < 2*n; ++i) {
76         area += (verts[i].x-px)*height; // trenutno pometena area
77
78         int s = lower_bound(points.begin(), points.end(), verts[i].s)-points.begin();
79         int e = lower_bound(points.begin(), points.end(), verts[i].e)-points.begin();
80         if (verts[i].start)
81             height = sl->add(s, e); // segment tree sprejme indekse, ne koordinat
82         else
83             height = sl->remove(s, e);
84         px = verts[i].x;
85     }
86
87     return area;
88 }

```

4.4 Najbližji par točk v ravnini

Vhod: Seznam $n \geq 2$ 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

```

1  typedef complex<double> P;
2  typedef vector<P>::iterator RAI; // or use template
3
4  bool byx(const P& a, const P& b) { return a.real() < b.real(); }
5  bool byy(const P& a, const P& b) { return a.imag() < b.imag(); }
6
7  double najblizji_tocki_bf(RAI s, RAI e) {
8      double m = numeric_limits<double>::max();
9      for (RAI i = s; i != e; ++i)
10         for (RAI j = i+1; j != e; ++j)
11             m = min(m, norm(*i - *j));
12     return m;
13 }
14 double najblizji_tocki_divide(RAI s, RAI e, const vector<P>& py) {
15     if (e - s < 50) return najblizji_tocki_bf(s, e);
16
17     size_t m = (e-s) / 2;
18     double d1 = najblizji_tocki_divide(s, s+m, py);
19     double d2 = najblizji_tocki_divide(s+m, e, py);
20     double d = min(d1, d2);
21     // merge
22     double meja = (s[m].real() + s[m+1].real()) / 2;
23     int n = py.size();
24     for (double i = 0; i < n; ++i) {
25         if (meja-d < py[i].real() && py[i].real() <= meja+d) {
26             double j = i+1;
27             double c = 0;
28             while (j < n && c < 7) { // navzdol gledamo le 7 ali dokler ni dlje od d
29                 if (meja-d < py[j].real() && py[j].real() <= meja+d) {
30                     double nd = norm(py[j]-py[i]);
31                     d = min(d, nd);
32                     if (py[j].imag() - py[i].imag() > d) break;
33                     ++c;
34                 }
35                 ++j;
36             }
37         }
38     }
39     return d;
40 }
41 double najblizji_tocki(const vector<P>& points) {
42     vector<P> px = points, py = points;
43     sort(px.begin(), px.end(), byx);
44     sort(py.begin(), py.end(), byy);
45     return najblizji_tocki_divide(px.begin(), px.end(), py);
46 }

```