

# 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

<b>1</b>	<b>Grafi</b>	<b>4</b>
1.1	Topološko sortiranje . . . . .	4
1.2	Najdaljša pot v DAGu . . . . .	4
1.3	Mostovi in prerezna vozlišča grafa . . . . .	5
1.4	Močno povezane komponente . . . . .	6
1.5	Najkrajša pot v grafu . . . . .	7
1.5.1	Dijkstra . . . . .	7
1.5.2	Dijkstra (kvadratičen) . . . . .	7
1.5.3	Bellman-Ford . . . . .	8
1.5.4	Floyd-Warhsall . . . . .	8
1.6	Minimalno vpeto drevo . . . . .	9
1.6.1	Prim . . . . .	9
1.6.2	Kruskal . . . . .	9
1.7	Najnižji skupni prednik . . . . .	10
1.8	Največji pretok in najmanjši prerez . . . . .	11
1.8.1	Edmonds-Karp . . . . .	11
1.9	Največje prirejanje in najmanjše pokritje . . . . .	12
<b>2</b>	<b>Podatkovne strukture</b>	<b>13</b>
2.1	Statično binarno iskalno drevo . . . . .	13
2.2	Drevo segmentov . . . . .	14
2.3	Avl drevo . . . . .	15
2.4	Fenwickovo drevo . . . . .	17
2.5	Fenwickovo drevo ( <b>n</b> -dim) . . . . .	18
<b>3</b>	<b>Algoritmi</b>	<b>19</b>
3.1	Najdaljše skupno podzaporedje . . . . .	19
3.2	Najdaljše naraščajoče podzaporedje . . . . .	20
3.3	Najdaljši strnjen palindrom . . . . .	21
3.4	Podseznam z največjo vsoto . . . . .	21
3.5	Leksikografsko minimalna rotacija . . . . .	22
3.6	BigInt in Karatsuba . . . . .	23
<b>4</b>	<b>Teorija števil</b>	<b>25</b>
4.1	Evklidov algoritem . . . . .	25
4.2	Razširjen Evklidov algoritem . . . . .	25
4.3	Kitajski izrek o ostankih . . . . .	25
4.4	Hitro potenciranje . . . . .	26
4.5	Številski sestavi . . . . .	26
4.6	Eulerjeva funkcija $\phi$ . . . . .	27
4.7	Eratostenovo rešeto . . . . .	27
4.8	Število deliteljev . . . . .	28
<b>5</b>	<b>Geometrija</b>	<b>28</b>
5.1	Osnove . . . . .	29
5.2	Konveksna ovojnica . . . . .	31
5.3	Ploščina unije pravokotnikov . . . . .	32

5.4	Najbližji par točk v ravnini . . . . .	33
<b>6</b>	<b>Matematika</b>	<b>34</b>

# 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

```
3  vector<int> topological_sort(const vector<vector<int>>& graf) {
4      int n = graf.size();
5      vector<int> ingoing(n, 0);
6      for (int i = 0; i < n; ++i)
7          for (const auto& u : graf[i])
8              ingoing[u]++;
9
10     queue<int> q; // morda priority_queue, če je vrstni red pomemben
11     for (int i = 0; i < n; ++i)
12         if (ingoing[i] == 0)
13             q.push(i);
14
15     vector<int> res;
16     while (!q.empty()) {
17         int t = q.front();
18         q.pop();
19
20         res.push_back(t);
21
22         for (int v : graf[t])
23             if (--ingoing[v] == 0)
24                 q.push(v);
25     }
26
27     return res; // če res.size() != n, ima graf cikle.
28 }
```

## 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

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

```

16         q.push(i); // topološko uredimo in gledamo maksimum
17
18     while (!q.empty()) {
19         int u = q.front();
20         q.pop();
21
22         for (const auto& edge : graf[u]) {
23             tie(v, w) = edge;
24             if (max_dist[u] >= 0) // da začnemo pri s-ju, sicer bi začeli na začetku, vsi pred s -1
25                 max_dist[v] = max(max_dist[v], max_dist[u] + w); // min za shortest path
26             if (--ind[v] == 0) q.push(v);
27         }
28     }
29     return max_dist[t];
30 }

```

### 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

```

3     namespace {
4         vector<int> low;
5         vector<int> dfs_num;
6         vector<int> parent;
7     }
8
9     void articulation_points_and_bridges_internal(int u, const vector<vector<int>>& graf,
10         vector<bool>& articulation_points_map, vector<pair<int, int>>& bridges) {
11         static int dfs_num_counter = 0;
12         low[u] = dfs_num[u] = ++dfs_num_counter;
13         int children = 0;
14         for (int v : graf[u]) {
15             if (dfs_num[v] == -1) { // unvisited
16                 parent[v] = u;
17                 children++;
18
19                 articulation_points_and_bridges_internal(v, graf, articulation_points_map, bridges);
20                 low[u] = min(low[u], low[v]); // update low[u]
21
22                 if (parent[u] == -1 && children > 1) // special root case
23                     articulation_points_map[u] = true;
24                 else if (parent[u] != -1 && low[v] >= dfs_num[u]) // articulation point
25                     articulation_points_map[u] = true; // assigned more than once
26                 if (low[v] > dfs_num[u]) // bridge
27                     bridges.push_back({u, v});
28             } else if (v != parent[u]) {
29                 low[u] = min(low[u], dfs_num[v]); // update low[u]
30             }
31         }
32     }
33
34     void articulation_points_and_bridges(int n, int m, const int E[][2],
35         vector<int>& articulation_points, vector<pair<int, int>>& bridges) {
36         vector<vector<int>> graf(n);
37         for (int i = 0; i < m; ++i) {
38             int a = E[i][0], b = E[i][1];
39             graf[a].push_back(b);
40             graf[b].push_back(a);
41         }
42
43         low.assign(n, -1);
44         dfs_num.assign(n, -1);
45         parent.assign(n, -1);
46     }

```

```

47     vector<bool> articulation_points_map(n, false);
48     for (int i = 0; i < n; ++i)
49         if (dfs_num[i] == -1)
50             articulation_points_and_bridges_internal(i, graf, articulation_points_map, bridges);
51
52     for (int i = 0; i < n; ++i)
53         if (articulation_points_map[i])
54             articulation_points.push_back(i); // actually return only articulation points
55 }

```

## 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](http://putka.upm.si/tasks/2012/2012_3kolo/zakladi)

```

3     namespace {
4         vector<int> low;
5         vector<int> dfs_num;
6         stack<int> S;
7         vector<int> component; // maps vertex to its component
8     }
9
10    void strongly_connected_components_internal(int u, const vector<vector<pair<int, int>>& graf,
11        vector<vector<int>>& comps) {
12        static int dfs_num_counter = 1;
13        low[u] = dfs_num[u] = dfs_num_counter++;
14        S.push(u);
15
16        for (const auto& v : graf[u]) {
17            if (dfs_num[v.first] == 0) // not visited yet
18                strongly_connected_components_internal(v.first, graf, comps);
19            if (dfs_num[v.first] != -1) // not popped yet
20                low[u] = min(low[u], low[v.first]);
21        }
22
23        if (low[u] == dfs_num[u]) { // extract the component
24            int cnum = comps.size();
25            comps.push_back({}); // start new component
26            int w;
27            do {
28                w = S.top(); S.pop();
29                comps.back().push_back(w);
30                component[w] = cnum;
31                dfs_num[w] = -1; // mark popped
32            } while (w != u);
33        }
34    }
35
36    void strongly_connected_components(const vector<vector<pair<int, int>>& graf,
37        vector<vector<int>>& comps, vector<map<int, int>>& dag) {
38        int n = graf.size();
39        low.assign(n, 0);
40        dfs_num.assign(n, 0);
41        component.assign(n, -1);
42
43        for (int i = 0; i < n; ++i)
44            if (dfs_num[i] == 0)
45                strongly_connected_components_internal(i, graf, comps);
46
47        dag.resize(comps.size()); // zgradimo kvocietni graf, teza povezave je vsota tez
48        for (int u = 0; u < n; ++u) {
49            for (const auto& v : graf[u]) {
50                if (component[u] != component[v.first]) {
51                    dag[component[u]][component[v.first]] += v.second; // ali max, kar zahteva naloga
52                }
53            }
54        }
55    }

```

## 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](http://putka.upm.si/tasks/2013/2013_1kolo/wolowitz)

```
3  typedef pair<int, int> pii;
4
5  int dijkstra(const vector<vector<pii>>& graf, int s, int t) {
6      int n = graf.size(), d, u;
7      priority_queue<pii, vector<pii>, greater<pii>> q;
8      vector<bool> visited(n, false);
9      vector<int> dist(n);
10
11      q.push({0, s}); // {cena, točka}
12      while (!q.empty()) {
13          tie(d, u) = q.top();
14          q.pop();
15
16          if (visited[u]) continue;
17          visited[u] = true;
18          dist[u] = d;
19
20          if (u == t) break; // ce iscemo do vseh točk spremeni v --n == 0
21
22          for (const auto& p : graf[u])
23              if (!visited[p.first])
24                  q.push({d + p.second, p.first});
25      }
26      return dist[t];
27 }
```

### 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](http://putka.upm.si/tasks/2013/2013_1kolo/wolowitz)

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

```

22         prev[to] = u;
23     }
24 }
25 } // v dist so sedaj razdalje od s do vseh, ki so bližje kot t (in t)
26 vector<int> path; // ce je dist[t] == INF, je t v drugi komponenti kot s
27 for (int v = t; v != s; v = prev[v])
28     path.push_back(v);
29 path.push_back(s);
30 reverse(path.begin(), path.end());
31 return path;
32 }

```

### 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](http://putka.upm.si/tasks/2013/2013_1kolo/wolowitz)

```

3  vector<int> bellman_ford(const vector<vector<pair<int, int>>>& graf, int s) {
4      int INF = numeric_limits<int>::max();
5      int n = graf.size(), v, w;
6      vector<int> dist(n, INF);
7      vector<int> prev(n, -1);
8      vector<bool> visited(n, false);
9
10     dist[s] = 0;
11     for (int i = 0; i < n-1; ++i) { // i je trenutna dolžina poti
12         for (int u = 0; u < n; ++u) {
13             for (const auto& edge : graf[u]) {
14                 tie(v, w) = edge;
15                 if (dist[u] != INF && dist[u] + w < dist[v]) {
16                     dist[v] = dist[u] + w;
17                     prev[v] = u;
18                 }
19             }
20         }
21     }
22
23     for (int u = 0; u < n; ++u) { // cycle detection
24         for (const auto& edge : graf[u]) {
25             tie(v, w) = edge;
26             if (dist[u] != INF && dist[u] + w < dist[v])
27                 return {}; // graph has a negative cycle !!
28         }
29     }
30     return dist;
31 }

```

### 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](http://putka.upm.si/tasks/2013/2013_1kolo/wolowitz)



```

3  vector<vector<int>> floyd_warshall(int n, int m, const int E[][3]) {
4      int INF = numeric_limits<int>::max();
5      vector<vector<int>> d(n, vector<int>(n, INF));
6      // vector<vector<int>> next(n, vector<int>(n, -1)); // da dobimo pot
7      for (int i = 0; i < m; ++i) {
8          int u = E[i][0], v = E[i][1], c = E[i][2];
9          d[u][v] = c;
10         // next[u][v] = v
11     }
12
13     for (int i = 0; i < n; ++i)
14         d[i][i] = 0;
15
16     for (int k = 0; k < n; ++k)
17         for (int i = 0; i < n; ++i)
18             for (int j = 0; j < n; ++j)
19                 if (d[i][k] != INF && d[k][j] != INF && d[i][k] + d[k][j] < d[i][j])
20                     d[i][j] = d[i][k] + d[k][j];
21                 // next[i][j] = next[i][k];
22     return d; // ce je kateri izmed d[i][i] < 0, ima graf negativen cikel
23 }

```

## 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

```

3  typedef pair<int, int> pii;
4
5  int prim_minimal_spanning_tree(const vector<vector<pii>>& graf) {
6      int n = graf.size(), d, u;
7      vector<bool> visited(n, false);
8      priority_queue<pii, vector<pii>, greater<pii>> q; // remove greater for max-tree
9      q.push({0, 0});
10
11      int sum = 0; // sum of the mst
12      int edge_count = 0; // stevilo dodanih povezav
13      while (!q.empty()) {
14          tie(d, u) = q.top();
15          q.pop();
16
17          if (visited[u]) continue;
18          visited[u] = true;
19
20          sum += d;
21          if (++edge_count == n) break; // drevo, jebeš solato
22
23          for (const auto& edge : graf[u])
24              if (!visited[edge.first])
25                  q.push({edge.second, edge.first});
26      } // ce zelimo drevo si shranjujemo se previous vertex.
27      return sum;
28 }

```

### 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

```
3 namespace {
4 vector<int> parent;
5 vector<int> rank;
6 }
7
8 int find(int x) {
9     if (parent[x] != x)
10         parent[x] = find(parent[x]);
11     return parent[x];
12 }
13
14 bool unija(int x, int y) {
15     int xr = find(x);
16     int yr = find(y);
17
18     if (xr == yr) return false;
19     if (rank[xr] < rank[yr]) { // rank lahko tudi izpustimo, potem samo parent[xr] = yr;
20         parent[xr] = yr;
21     } else if (rank[xr] > rank[yr]) {
22         parent[yr] = xr;
23     } else {
24         parent[yr] = xr;
25         rank[xr]++;
26     }
27     return true;
28 }
29
30 int kruskal_minimal_spanning_tree(int n, int m, int E[][3]) {
31     rank.assign(n, 0);
32     parent.assign(n, 0);
33     for (int i = 0; i < n; ++i) parent[i] = i;
34     vector<tuple<int, int, int>> edges;
35     for (int i = 0; i < m; ++i) edges.emplace_back(E[i][2], E[i][0], E[i][1]);
36     sort(edges.begin(), edges.end());
37
38     int sum = 0, a, b, c, edge_count = 0;
39     for (int i = 0; i < m; ++i) {
40         tie(c, a, b) = edges[i];
41         if (unija(a, b)) {
42             sum += c;
43             edge_count++;
44         }
45         if (edge_count == n - 1) break;
46     }
47     return sum;
48 }
```

## 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/>

```
3 vector<vector<int>> preprocess(const vector<int>& parent) {
4     int n = parent.size();
5     int logn = 1;
6     while (1 << ++logn < n);
7     vector<vector<int>> P(n, vector<int>(logn, -1));
8
9     for (int i = 0; i < n; i++) // prvi prednik za i je parent[i]
10         P[i][0] = parent[i];
11 }
```

```

12     for (int j = 1; 1 << j < n; j++)
13         for (int i = 0; i < n; i++)
14             if (P[i][j - 1] != -1) // P[i][j] = 2^j-ti prednik i-ja
15                 P[i][j] = P[P[i][j - 1]][j - 1];
16     return P;
17 }
18
19 int level_internal(const vector<int>& parent, vector<int>& L, int v) {
20     if (L[v] != -1) return L[v];
21     return L[v] = (parent[v] == v) ? 0 : level_internal(parent, L, parent[v]) + 1;
22 }
23
24 vector<int> levels(const vector<int>& parent) {
25     vector<int> L(parent.size(), -1);
26     for (size_t i = 0; i < parent.size(); ++i) level_internal(parent, L, i);
27     return L;
28 }
29
30 int find_lca(const vector<int>& parent, int u, int v,
31             const vector<vector<int>>& P, const vector<int>& L) {
32     if (L[u] < L[v]) // if u is on a higher level than v then we swap them
33         swap(u, v);
34
35     int log = 1;
36     while (1 << ++log <= L[u]);
37     log--; // we compute the value of [log(L[u])]
38
39     for (int i = log; i >= 0; i--) // we find the ancestor of node u situated on
40         if (L[u] - (1 << i) >= L[v]) // the same level as v using the values in P
41             u = P[u][i];
42
43     if (u == v) return u;
44
45     for (int i = log; i >= 0; i--) // we compute LCA(u, v) using the values in P
46         if (P[u][i] != -1 && P[v][i] != -1)
47             u = P[u][i], v = P[v][i];
48
49     return parent[u];
50 }

```

## 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

```

3     namespace {
4         const int INF = numeric_limits<int>::max();
5         struct triple { int u, p, m; };
6     }
7
8     int edmonds_karp_maximal_flow(const vector<vector<int>>& capacity, int s, int t) {
9         int n = capacity.size();
10        vector<vector<int>> flow(n, vector<int>(n, 0));
11        int maxflow = 0;
12        while (true) {
13            vector<int> prev(n, -2); // hkrati tudi visited array
14            int bot = INF; // bottleneck
15            queue<triple> q;
16            q.push({s, -1, INF});
17            while (!q.empty()) { // compute a possible path, add its bottleneck to the total flow
18                int u = q.front().u, p = q.front().p, mini = q.front().m; // while such path exists
19                q.pop();
20
21                if (prev[u] != -2) continue;
22                prev[u] = p;
23
24                if (u == t) { bot = mini; break; }

```

```

25
26         for (int i = 0; i < n; ++i) {
27             int available = capacity[u][i] - flow[u][i];
28             if (available > 0) {
29                 q.push({i, u, min(available, mini)}); // kumulativni minimum
30             }
31         }
32     }
33
34     if (prev[t] == -2) break;
35
36     maxflow += bot;
37     for (int u = t; u != s; u = prev[u]) { // popravimo trenutni flow nazaj po poti
38         flow[u][prev[u]] -= bot;
39         flow[prev[u]][u] += bot;
40     }
41 }
42 return maxflow;
43 }

```

## 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

```

3 namespace {
4     vector<int> match, vis;
5 }
6
7 int augmenting_path(const vector<vector<int>>& graf, int left) {
8     if (vis[left]) return 0;
9     vis[left] = 1;
10    for (int right : graf[left]) {
11        if (match[right] == -1 || augmenting_path(graf, match[right])) {
12            match[right] = left;
13            match[left] = right;
14            return 1;
15        }
16    }
17    return 0;
18 }
19
20 void mark_vertices(const vector<vector<int>>& graf, vector<bool>& cover, int v) {
21     if (vis[v]) return;
22     vis[v] = 1;
23     cover[v] = false;
24     for (int r : graf[v]) {
25         cover[r] = true;
26         if (match[r] != -1)
27             mark_vertices(graf, cover, match[r]);
28     }
29 }
30
31 int bipartite_matching(const vector<vector<int>>& graf, int left_num) {
32     int n = graf.size();
33     match.assign(2*n, -1);
34     int mcbm = 0; // prvih left_num je v levem delu grafa
35     for (int left = 0; left < left_num; ++left) {
36         vis.assign(n, 0);
37         mcbm += augmenting_path(graf, left);
38     }
39 }

```

```

38     }
39     return mcbm;
40 }
41
42 vector<int> minimal_cover(const vector<vector<int>>& graf, int left_num) {
43     bipartite_matching(graf, left_num);
44     int n = graf.size();
45     vis.assign(2*n, 0);
46     vector<bool> cover(n, false);
47     fill(cover.begin(), cover.begin() + left_num, true);
48     for (int left = 0; left < n; ++left)
49         if (match[left] == -1)
50             mark_vertices(graf, cover, left);
51
52     vector<int> result; // ni potrebno, lahko se uporablja kar cover
53     for (int i = 0; i < n; ++i)
54         if (cover[i])
55             result.push_back(i);
56     return result;
57 }

```

## 2 Podatkovne strukture

### 2.1 Statično binarno iskalno drevo

**Operacije:** Klasično uravnoreženo 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](http://putka.upm.si/tasks/2011/2011_finale/kitajci)

```

3  namespace {
4  int depth = 10; // depth of the tree
5  int n = 1 << 10; // number of different elements stored
6  vector<int> tree(2*n); // number of elements in the tree = tree[0]
7  }
8
9  void insert(int idx, int val = 1) {
10     int i = n - 1 + idx;
11     while (i > 0) {
12         tree[i] += val;
13         i--;
14         i >>= 1;
15     }
16     tree[0] += val;
17 }
18
19 int get_kth(int k) {
20     int i = 0;
21     while (i < n - 1) {
22         int lc = tree[2*i+1];
23         if (lc <= k) {
24             i = 2*i + 2;
25             k -= lc;
26         } else {
27             i = 2*i + 1;
28         }
29     }
30     return i - n + 1;
31 }
32
33 bool remove(int idx) {
34     if (tree[n-1 + idx] <= 0) return false;
35     insert(idx, -1);
36     return true;
37 }
38

```

```

39 void print() {
40     for (int i = 0; i <= depth; ++i) {
41         cout << string((1 << (depth - i)) - 1, ' ');
42         for (int j = (1 << i) - 1; j < (1 << (i+1)) - 1; ++j) {
43             cout << tree[j] << string((1 << (depth - i + 1)) - 1, ' ');
44         }
45         cout << '\n';
46     }
47 }

```

## 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 odstraniti in node-i obstajajo). Prav tako lahko vrnemo število izbranih 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>

```

6  struct Node { // static division points, [l, r] intervals
7      int l, m, r;
8      Node* left, *right;
9      int here_count, total_count;
10     Node(int ll, int rr) : l(ll), m((ll+rr) / 2), r(rr),
11                          left(nullptr), right(nullptr), here_count(0), total_count(0) {}
12     int insert(int f, int t, int inc = 1) { // insert inc elemnts into [f, t]. use for removal too
13         if (f <= l && r <= t) {
14             here_count += inc;
15             int lb = max(l, f), rb = min(r, t);
16             int inserted = (rb - lb + 1) * inc;
17             total_count += inserted;
18             return inserted;
19         }
20         int inserted = 0;
21         if (f <= m) {
22             if (left == nullptr) left = new Node(l, m);
23             inserted += left->insert(f, t, inc);
24         }
25         if (m + 1 <= t) {
26             if (right == nullptr) right = new Node(m + 1, r);
27             inserted += right->insert(f, t, inc);
28         }
29         total_count += inserted;
30
31         if (left != nullptr && right != nullptr) { // move full levels up (speedup, ne rabiš)
32             int child_here_count = min(left->here_count, right->here_count);
33             left->here_count -= child_here_count;
34             right->here_count -= child_here_count;
35             here_count += child_here_count;
36             left->total_count -= (m - l + 1) * child_here_count;
37             right->total_count -= (r - m) * child_here_count;
38         } // end speed up
39         return inserted;
40     }
41
42     int count(int f, int t) { // count on interval [f, t]
43         if (f <= l && r <= t) return total_count;
44         int sum = 0, lb = max(l, f), rb = min(r, t);
45         if (f <= m && left != nullptr) sum += left->count(f, t);

```

```

46         if (m + 1 <= t && right != nullptr) sum += right->count(f, t);
47         return sum + (rb - lb + 1) * here_count;
48     }
49
50     int get_kth(int k, int parent_count = 0) { // zero based
51         int above_count = here_count + parent_count;
52         int lc = above_count * (m - 1 + 1) + get_cnt(left);
53         if (k < lc) {
54             if (left == nullptr) return 1 + k/above_count;
55             return left->get_kth(k, above_count);
56         } else {
57             k -= lc;
58             if (right == nullptr) return m + 1 + k/above_count;
59             return right->get_kth(k, above_count);
60         }
61     }
62
63     void print(int l, int r) {
64         printf("[%d, %d]: here: %d, total: %d\n", l, r, here_count, total_count);
65         if (left) left->print(l, m);
66         if (right) right->print(m+1, r);
67     }
68
69     private:
70     int get_cnt(Node* left) {
71         return (left == nullptr) ? 0 : left->total_count;
72     }
73 };
74
75 // poenostavitev, ce ne rabimo intervalov
76 struct SimpleNode { // static division points
77     SimpleNode* left, *right;
78     int cnt;
79     SimpleNode() : left(nullptr), right(nullptr), cnt(0) {}
80     void insert(int l, int r, int val) {
81         if (l == r) { cnt++; return; }
82         int mid = (l + r) / 2; // watch overflow
83         if (val <= mid) {
84             if (left == nullptr) left = new SimpleNode();
85             left->insert(l, mid, val);
86         } else {
87             if (right == nullptr) right = new SimpleNode();
88             right->insert(mid + 1, r, val);
89         }
90         cnt++;
91     }
92
93     int pop_kth(int l, int r, int k) { // one based
94         if (l == r) { cnt--; return l; }
95         int mid = (l + r) / 2;
96         if (k <= get_cnt(left)) {
97             cnt--;
98             return left->pop_kth(l, mid, k);
99         } else {
100             k -= get_cnt(left);
101             cnt--;
102             return right->pop_kth(mid + 1, r, k);
103         }
104     }
105
106     private:
107     int get_cnt(SimpleNode* x) {
108         if (x == nullptr) return 0;
109         return x->cnt;
110     }
111 };
112 #endif // IMPLEMENTACIJA_PS_SEGMENT_TREE_H_

```

## 2.3 Avl drevo

**Operacije:** Klasično uravnoreženo 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.

```

6  template<typename T>
7  class AvlNode {
8      public:
9          AvlNode<T>* left, *right;
10         size_t height, size, count;
11         T value;
12         AvlNode(const T& v) : left(nullptr), right(nullptr), height(1), size(1), count(1), value(v) {}
13         ostream& print(ostream& os, int indent = 0) {
14             if (right != nullptr) right->print(os, indent+2);
15             for (int i = 0; i < indent; ++i) os << ' '; // or use string(indent, ' ')
16             os << value << endl;
17             if (left != nullptr) left->print(os, indent+2);
18             return os;
19         }
20     };
21
22     template<typename T>
23     class AvlTree {
24     public:
25         AvlTree() : root(nullptr) {}
26         int size() const {
27             return size(root);
28         }
29         AvlNode<T>* insert(const T& val) {
30             return insert(val, root);
31         }
32         bool erase(const T& val) {
33             return erase(val, root);
34         }
35         const AvlNode<T>* get_nth(size_t index) const {
36             return get_nth(root, index);
37         }
38         const AvlNode<T>* find(const T& value) const {
39             return find(root, value);
40         }
41         template<typename U>
42         friend ostream& operator<<(ostream& os, const AvlTree<U>& tree);
43
44     private:
45         int size(const AvlNode<T>* const& node) const {
46             if (node == nullptr) return 0;
47             else return node->size;
48         }
49         size_t height(const AvlNode<T>* const& node) const {
50             if (node == nullptr) return 0;
51             return node->height;
52         }
53         int getBalance(const AvlNode<T>* const& node) const {
54             return height(node->left) - height(node->right);
55         }
56         void updateHeight(AvlNode<T>* const& node) {
57             node->height = max(height(node->left), height(node->right)) + 1;
58         }
59         void rotateLeft(AvlNode<T>*& node) {
60             AvlNode<T>* R = node->right;
61             node->size -= size(R->right) + R->count; R->size += size(node->left) + node->count;
62             node->right = R->left; R->left = node; node = R;
63             updateHeight(node->left); updateHeight(node);
64         }
65         void rotateRight(AvlNode<T>*& node) {
66             AvlNode<T>* L = node->left;
67             node->size -= size(L->left) + L->count; L->size += size(node->right) + node->count;
68             node->left = L->right; L->right = node; node = L;
69             updateHeight(node->right); updateHeight(node);
70         }
71         void balance(AvlNode<T>*& node) {
72             int b = getBalance(node);
73             if (b == 2) {
74                 if (getBalance(node->left) == -1) rotateLeft(node->left);

```



```

75         rotateRight(node);
76     } else if (b == -2) {
77         if (getBalance(node->right) == 1) rotateRight(node->right);
78         rotateLeft(node);
79     } else {
80         updateHeight(node);
81     }
82 }
83
84 AvlNode<T>* insert(const T& val, AvlNode<T>*& node) {
85     if (node == nullptr) return node = new AvlNode<T>(val);
86     node->size++;
87     AvlNode<T>* return_node = node;
88     if (val < node->value) return_node = insert(val, node->left);
89     else if (node->value == val) node->count++;
90     else if (node->value < val) return_node = insert(val, node->right);
91     balance(node);
92     return return_node;
93 }
94
95 bool erase(const T& val, AvlNode<T>*& node) {
96     if (node == nullptr) return false;
97     if (val < node->value) {
98         if (erase(val, node->left)) {
99             node->size--;
100             return true;
101         }
102     } else if (node->value < val) {
103         if (erase(val, node->right)) {
104             node->size--;
105             return true;
106         }
107     } else if (node->value == val && node->count > 0) {
108         node->count--;
109         node->size--;
110         return true;
111     }
112     return false;
113 }
114
115 const AvlNode<T>* get_nth(const AvlNode<T>* const& node, size_t n) const {
116     size_t left_size = size(node->left);
117     if (n < left_size) return get_nth(node->left, n);
118     else if (n < left_size + node->count) return node;
119     else if (n < node->size) return get_nth(node->right, n - left_size - node->count);
120     else return nullptr;
121 }
122
123 const AvlNode<T>* find(const AvlNode<T>* const& node, const T& value) const {
124     if (node == nullptr) return nullptr;
125     if (value < node->value) return find(node->left, value);
126     else if (value == node->value) return node;
127     else return find(node->right, value);
128 }
129
130 AvlNode<T>* root;
131 };
132
133 template<typename T>
134 ostream& operator<<(ostream& os, const AvlTree<T>& tree) {
135     if (tree.root == nullptr) os << "Tree empty";
136     else tree.root->print(os);
137     return os;
138 }
139
140 #endif // IMPLEMENTACIJA_PS_AVL_TREE_H_

```

## 2.4 Fenwickovo drevo

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

```
3 namespace {
4   const int MAX_INDEX = 16;
5   vector<int> tree(MAX_INDEX+1, 0); // global tree, 1 based!!
6 }
7
8 void update(int idx, int val) { // increments idx for value
9   while (idx <= MAX_INDEX) {
10     tree[idx] += val;
11     idx += (idx & -idx);
12   }
13 }
14
15 int read(int idx) { // read sum of [1, x], read(0) == 0, duh.
16   int sum = 0;
17   while (idx > 0) {
18     sum += tree[idx];
19     idx -= (idx & -idx);
20   }
21   return sum;
22 }
23
24 int readSingle(int idx) { // read a single value, readSingle(x) == read(x)-read(x-1)
25   int sum = tree[idx];
26   if (idx > 0) {
27     int z = idx - (idx & -idx);
28     idx--;
29     while (idx != z) {
30       sum -= tree[idx];
31       idx -= (idx & -idx);
32     }
33   }
34   return sum;
35 }
```

## 2.5 Fenwickovo drevo (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](http://putka.upm.si/tasks/2010/2010_3kolo/stanovanja)

```
3 typedef vector<vector<vector<int>>> vvvvi;
4
5 int sum(int x, int y, int z, const vvvvi& tree) { // [0,0,0 - x,y,z] vključno
6   int result = 0;
7   for (int i = x; i >= 0; i = (i & (i+1)) - 1)
8     for (int j = y; j >= 0; j = (j & (j+1)) - 1)
9       for (int k = z; k >= 0; k = (k & (k+1)) - 1)
10         result += tree[i][j][k];
11   return result;
12 }
13
14 void inc(int x, int y, int z, int delta, vvvvi& tree) { // povečaj na koordinatah, 0 based
15   int n = tree.size(); // lahko so tudi različni n-ji za posamezno dimenzijo
16   for (int i = x; i < n; i |= i+1)
17     for (int j = y; j < n; j |= j+1)
18       for (int k = z; k < n; k |= k+1)
19         tree[i][j][k] += delta;
```

```

20 }
21
22 int subsum(int x1, int y1, int z1,
23           int x2, int y2, int z2, const vvvi& tree) { // vsota na [x1,y1,z1 - x2,y2,z2], vključno
24     x1--; y1--; z1--;
25     return sum(x2, y2, z2, tree) -
26           sum(x1, y2, z2, tree) -
27           sum(x2, y1, z2, tree) - // pravilo vključitev in izključitev
28           sum(x2, y2, z1, tree) +
29           sum(x1, y1, z2, tree) +
30           sum(x1, y2, z1, tree) +
31           sum(x2, y1, z1, tree) -
32           sum(x1, y1, z1, tree);
33 }

```

## 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

```

3 // lahko pridemo na  $O(n \sqrt{n})$ 
4 vector<int> longest_common_subsequence(const vector<int>& a, const vector<int>& b) {
5     int n = a.size(), m = b.size();
6     vector<vector<int>>> c(n + 1, vector<int>(m + 1, 0));
7     for (int i = 1; i <= n; ++i)
8         for (int j = 1; j <= m; ++j)
9             if (a[i-1] == b[j-1])
10                 c[i][j] = c[i-1][j-1] + 1;
11             else
12                 c[i][j] = max(c[i][j-1], c[i-1][j]);
13     vector<int> sequence;
14     int i = n, j = m;
15     while (i > 0 && j > 0) {
16         if (a[i-1] == b[j-1]) {
17             sequence.push_back(a[i-1]);
18             i--; j--;
19         } else if (c[i][j-1] > c[i-1][j]) {
20             j--;
21         } else {
22             i--;
23         }
24     }
25     reverse(sequence.begin(), sequence.end());
26     return sequence;
27 }
28
29 //  $O(n)$  prostora, lahko tudi zgornjo verzijo, ce je dovolj spomina.
30 int longest_common_subsequence_length(const vector<int>& a, const vector<int>& b) {
31     int n = a.size(), m = b.size(); // po možnosti transponiraj tabelo, ce je malo spomina
32     vector<vector<int>>> c(2, vector<int>(m + 1, 0));
33     bool f = 0;
34     for (int i = 1; i <= n; ++i) {
35         for (int j = 1; j <= m; ++j)
36             if (a[i-1] == b[j-1])
37                 c[f][j] = c[!f][j-1] + 1;
38             else
39                 c[f][j] = max(c[f][j-1], c[!f][j]);
40         f = !f;
41     }
42     return c[!f][m];
43 }

```

## 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.

```
3  vector<int> longest_increasing_subsequence(const vector<int>& a) {
4      vector<int> p(a.size()), b;
5      int u, v;
6
7      if (a.empty()) return {};
8      b.push_back(0);
9
10     for (size_t i = 1; i < a.size(); i++) {
11         if (a[b.back()] < a[i]) {
12             p[i] = b.back();
13             b.push_back(i);
14             continue;
15         }
16
17         for (u = 0, v = b.size()-1; u < v; ) {
18             int c = (u + v) / 2;
19             if (a[b[c]] < a[i]) u = c + 1;
20             else v = c;
21         }
22
23         if (a[i] < a[b[u]]) {
24             if (u > 0) p[i] = b[u-1];
25             b[u] = i;
26         }
27     }
28
29     for (u = b.size(), v = b.back(); u--; v = p[v]) b[u] = a[v];
30     return b; // b[u] = v, če želiš indekse, ali ce ima a neinteger elemente
31 }
32
33 vector<int> longest_increasing_subsequence_square(const vector<int>& a) {
34     if (a.size() == 0) return {};
35     int max_length = 1, best_end = 0;
36     int n = a.size();
37     vector<int> m(n, 0), prev(n, -1); // m[i] = dolžina lis, ki se konca pri i
38     m[0] = 1;
39     prev[0] = -1;
40
41     for (int i = 1; i < n; i++) {
42         m[i] = 1;
43         prev[i] = -1;
44
45         for (int j = i-1; j >= 0; --j) {
46             if (m[j] + 1 > m[i] && a[j] < a[i]) {
47                 m[i] = m[j] + 1;
48                 prev[i] = j;
49             }
50
51             if (m[i] > max_length) {
52                 best_end = i;
53                 max_length = m[i];
54             }
55         }
56     }
57     vector<int> lis;
58     for (int i = best_end; i != -1; i = prev[i]) lis.push_back(a[i]);
59     reverse(lis.begin(), lis.end());
60     return lis;
61 }
```

### 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/>

```
3 pair<int, int> find_longest_palindrome(const string& str) { // returns [start, end]
4     int n = str.length();
5     if (n == 0) return {0, 0};
6     if (n == 1) return {0, 1};
7     n = 2*n + 1; // Position count
8     int L[n]; // LPS Length Array
9     L[0] = 0;
10    L[1] = 1;
11    int C = 1; // centerPosition
12    int R = 2; // centerRightPosition
13    int i = 0; // currentRightPosition
14    int iMirror; // currentLeftPosition
15    int maxLPSLength = 0, maxLPSCenterPosition = 0;
16    int start = -1, end = -1, diff = -1;
17
18    for (i = 2; i < n; i++) {
19        iMirror = 2*C-i; // get currentLeftPosition iMirror for currentRightPosition i
20        L[i] = 0;
21        diff = R - i; // If currentRightPosition i is within centerRightPosition R
22        if (diff > 0) L[i] = min(L[iMirror], diff);
23
24        while ( ((i + L[i]) < n && (i - L[i]) > 0) && ( // Attempt to expand
25                ((i + L[i] + 1) % 2 == 0) || // palindrome centered at
26                (str[(i + L[i] + 1)/2] == str[(i - L[i] - 1)/2])) { // currentRightPosition i Here
27            L[i]++; // for odd positions, we
28        } // compare characters and if
29        // match then increment LPS
30        // Length by ONE If even
31        if (L[i] > maxLPSLength) { // Track maxLPSLength // position, we just increment
32            maxLPSLength = L[i]; // LPS by ONE without any
33            maxLPSCenterPosition = i; // character comparison
34        }
35        if (i + L[i] > R) { // If palindrome centered at currentRightPosition i
36            C = i; // expand beyond centerRightPosition R,
37            R = i + L[i]; // adjust centerPosition C based on expanded palindrome.
38        }
39    }
40    start = (maxLPSCenterPosition - maxLPSLength)/2;
41    end = start + maxLPSLength;
42    return {start, end};
43 }
```

### 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>

```
3 int maximum_subarray(const vector<int>& a) { // glej komentarje ce ne dovolimo prazne vsote
4     int max_ending_here = 0, max_so_far = 0; // A[0]
5     for (int x : a) { // a[1:]
6         max_ending_here = max(0, max_ending_here + x);
7         max_so_far = max(max_so_far, max_ending_here);
8     }
9     return max_so_far;
```

```

10 }
11
12 vector<int> maximum_subarray_extract(const vector<int>& a) {
13     int max_ending_here = 0, max_so_far = 0;
14     int idx_from = 0, total_to = 0, total_from = 0;
15     for (size_t i = 0; i < a.size(); ++i) {
16         if (max_ending_here + a[i] > 0) {
17             max_ending_here += a[i];
18         } else {
19             idx_from = i + 1;
20         }
21         if (max_ending_here > max_so_far) {
22             total_from = idx_from;
23             total_to = i + 1;
24             max_so_far = max_ending_here;
25         }
26     }
27     return vector<int>(a.begin() + total_from, a.begin() + total_to);
28 }
29
30 int maximum_subarray(const vector<int>& a, int k) { // k = 1 ... n(n-1)/2
31     int n = a.size();
32     vector<int> s(n+1, 0);
33     vector<pair<int, int>> p(n+1);
34     priority_queue<tuple<int, int, int>> q;
35     for (int i = 0, m = 0; i < n; ++i) {
36         s[i+1] = s[i] + a[i];
37         if (s[m] > s[i]) m = i;
38         p[i+1] = make_pair(s[i+1], i+1);
39         q.push(make_tuple(s[i+1]-s[m], i+1, m));
40     }
41     sort(p.begin(), p.end());
42     vector<int> ss(n+1);
43     for (int i = 0; i <= n; ++i)
44         ss[p[i].second] = i;
45
46     int v = -1, i, j;
47     for (int l = 1; l <= k; ++l) {
48         tie(v, i, j) = q.top();
49         q.pop();
50
51         for (int m = ss[j] + 1; m <= n; ++m) {
52             if (p[m].second < i) {
53                 q.push(make_tuple(s[i]-p[m].first, i, p[m].second));
54                 break;
55             }
56         }
57     }
58     return v;
59 }

```

### 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  $n$ .

```

3 int minimal_rotation(string s) {
4     s += s;
5     int len = s.size(), i = 0, j = 1, k = 0;
6     while (i + k < len && j + k < len) {
7         if (s[i+k] == s[j+k]) k++;
8         else if (s[i+k] > s[j+k]) { i = i+k+1; if (i <= j) i = j+1; k = 0; }
9         else if (s[i+k] < s[j+k]) { j = j+k+1; if (j <= i) j = i+1; k = 0; }
10    }
11    return min(i, j);
12 }

```

## 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  $+$ ,  $<$  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  $+$ ,  $-$ ,  $*$  števka,  $O(n^2)$  za  $*$ ,  $O(n^{1.585})$  za karatsubo.

**Prostorska zahtevnost:**  $O(n)$

**Testiranje na terenu:** [http://putka.upm.si/tasks/test\\_okolja/odstevanje](http://putka.upm.si/tasks/test_okolja/odstevanje) in [http://putka.upm.si/tasks/test\\_okolja/sestevanje](http://putka.upm.si/tasks/test_okolja/sestevanje)

```
6  template<typename T>
7  struct Number {
8      bool sign = true; // true = 1 = +, false = 0 = -
9      deque<T> data;
10     static const int base = 10; // bi se lahko spremenilo samo I/O nebi delal, matematika bi
11     static const int KARATSUBA_LIMIT = 2; // kdaj preklopi na bruteforce množenje
12
13     Number() {} // default constructor, positive zero
14     Number(const deque<T>& a, bool s = 1) : sign(s), data(a) { clear_zeros(); }
15     Number(deque<T>&& a, bool s = 1) : sign(s), data(a) { clear_zeros(); }
16     Number(const string& s) { from_string(s); }
17     void from_string(const string& s) {
18         if (s.size() == 0) data = {0};
19         int i = 0;
20         if (s[0] == '+') sign = 1, i = 1;
21         if (s[0] == '-') sign = 0, i = 1;
22         int l = s.size(); data.resize(l-i);
23         for (; i < l; ++i) {
24             if (!('0' <= s[i] && s[i] <= '9')) return; // silent quit po prvi ne številki
25             data[l-i-1] = s[i] - '0';
26         }
27         clear_zeros();
28     }
29     string to_string() const {
30         if (data.empty()) return "0";
31         string s = (sign) ? "" : "-";
32         for (int i = data.size() - 1; i >= 0; --i)
33             s.push_back('0' + data[i]);
34         return s;
35     }
36     Number operator+(const Number& o) const { // remove signs if using for positive only
37         if (sign == 0) return -((- *this) + (-o));
38         if (sign == 1 && o.sign == 0) return (*this < -o) ? -((-o) - *this) : *this - (-o);
39         bool carry = false;
40         int i = 0, j = 0, n = data.size(), m = o.data.size();
41         deque<T> r;
42         while (i < n || j < m) {
43             T c = ((i < n) ? data[i++] : 0) + ((j < m) ? o.data[j++] : 0) + carry;
44             carry = (c >= base);
45             r.push_back(c % base);
46         }
47         if (carry) r.push_back(1);
48         return Number(move(r));
49     }
50     Number operator-() const { return Number(data, !sign); }
51     bool operator==(const Number& o) const { return sign == o.sign && data == o.data; }
52     bool operator<(const Number& o) const {
53         if (sign == o.sign) {
54             if (sign == 0) return -o < - *this;
```

```

55         if (data.size() == o.data.size()) {
56             for (int i = data.size() - 1; i >= 0; --i)
57                 if (data[i] == o.data[i]) continue;
58                 else return data[i] < o.data[i];
59         }
60         return data.size() < o.data.size();
61     }
62     return sign < o.sign;
63 }
64 Number& operator+=(const Number& o) { // lahko tudi s +. Samo za pozitivne.
65     bool carry = false;
66     int i = 0, j = 0, n = data.size(), m = o.data.size();
67     while (i < n || j < m) {
68         if (i < n && j < m) data[i] += o.data[j++] + carry;
69         else if (i < n) data[i] += carry;
70         else data.push_back(o.data[j++] + carry);
71         carry = data[i] / base;
72         data[i++] %= base;
73     }
74     if (carry) data.push_back(1);
75     clear_zeros();
76     return *this;
77 }
78 Number operator*(const T& o) const { // z eno števkco
79     deque<T> r;
80     int carry = 0, n = data.size();
81     for (int i = 0; i < n; ++i) {
82         T c = data[i]*o + carry;
83         carry = c / base;
84         c %= base;
85         r.push_back(c);
86     }
87     if (carry) r.push_back(carry);
88     return Number(move(r), sign);
89 }
90 Number operator<<(int n) const { // na začetek dodamo n ničel
91     deque<T> r(n, 0);
92     r.insert(r.end(), data.begin(), data.end());
93     return Number(move(r));
94 }
95 Number operator*(const Number<T>& o) const {
96     if (sign == 0 && o.sign == 0) return ((*this) * (-o));
97     if (sign == 0 && o.sign == 1) return -((*this) * o);
98     if (sign == 1 && o.sign == 0) return -(*this * (-o));
99     Number r;
100     int m = o.data.size();
101     for (int i = 0; i < m; ++i)
102         r += (*this*o.data[i] << i);
103     return r;
104 }
105 Number operator-(const Number& o) const {
106     deque<T> r;
107     bool carry = false;
108     int i = 0, j = 0, n = data.size(), m = o.data.size();
109     while (i < n || j < m) {
110         T c = data[i++] + base - ((j < m) ? o.data[j++] : 0) - carry;
111         carry = 1 - c / base;
112         c %= base;
113         r.push_back(c);
114     }
115     return Number(move(r));
116 }
117
118 private:
119     void clear_zeros() {
120         while (data.size() > 0 && data.back() == 0) data.pop_back();
121         if (data.empty()) sign = 1;
122     }
123 };
124
125 template<typename T> // karatsuba algorithm
126 Number<T> karatsuba(const Number<T>& a, const Number<T>& b) {
127     if (a.data.size() <= Number<T>::KARATSUBA_LIMIT || b.data.size() <= Number<T>::KARATSUBA_LIMIT)
128         return a*b;
129
130     if (a.sign == 0 && b.sign == 0) return ((-a) * (-b));
131     if (a.sign == 0 && b.sign == 1) return -((-a) * b);
132     if (a.sign == 1 && b.sign == 0) return -(a * (-b));
133
134     Number<T> a0, a1, b0, b1, c0, c1, c2;
135     int m = min(a.data.size(), b.data.size())/2; // choose m carefully

```



```

136
137     a0.data.assign(a.data.begin(), a.data.begin()+m);
138     a1.data.assign(a.data.begin()+m, a.data.end());
139     b0.data.assign(b.data.begin(), b.data.begin()+m);
140     b1.data.assign(b.data.begin()+m, b.data.end());
141
142     c2 = karatsuba(a1, b1);
143     c0 = karatsuba(a0, b0);
144     c1 = karatsuba(a0+a1, b0+b1) - c0 - c2;
145
146     return (c2 << 2*m) + (c1 << m) + c0;
147 }
148 #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:**  $O(1)$

```

3  int gcd(int a, int b) {
4      int t;
5      while (b != 0) {
6          t = a % b;
7          a = b;
8          b = t;
9      }
10     return a;
11 }

```

### 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  $\mathbb{Z}_b^*$ .

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

**Prostorska zahtevnost:**  $O(1)$

**Testiranje na terenu:** UVa 756

```

3  int ext_gcd(int a, int b, int& retx, int& rety) {
4      int x = 0, px = 1, y = 1, py = 0, r, q;
5      while (b != 0) {
6          r = a % b; q = a / b; // quotient and reminder
7          a = b; b = r;        // gcd swap
8          r = px - q * x;      // x swap
9          px = x; x = r;
10         r = py - q * y;      // y swap
11         py = y; y = r;
12     }
13     retx = px; rety = py;    // return
14     return a;
15 }

```

### 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

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

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. 25)

**Testiranje na terenu:** UVa 756

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

```
3  int mul_inverse(int a, int m) {
4      int x, y;
5      ext_gcd(a, m, x, y);
6      return (x + m) % m;
7  }
8
9  int chinese_remainder_theorem(const vector<pair<int, int>>& cong) {
10     int M = 1;
11     for (size_t i = 0; i < cong.size(); ++i) {
12         M *= cong[i].second;
13     }
14     int x = 0, a, m;
15     for (const auto& p : cong) {
16         tie(a, m) = p;
17         x += a * M / m * mul_inverse(M/m, m);
18         x %= M;
19     }
20     return (x + M) % M;
21 }
```

## 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](http://putka.upm.si/tasks/2010/2010_3kolo/nicle)

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

## 4.5 Številski sestavi

**Vhod:** Število  $n \in \mathbb{N}_0$  ali  $\frac{p}{q} \in \mathbb{Q}$  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](http://putka.upm.si/tasks/2010/2010_finale/ulomki)

**Opomba:** Zgornja meja za bazo  $b$  je dolžina niza STEVILSKI\_SESTAVI\_ZNAKI.

```

3  char STEVILSKI_SESTAVI_ZNAKI[] = "0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ";
4
5  string convert_int(int n, int baza) {
6      if (n == 0) return "0";
7      string result;
8      while (n > 0) {
9          result.push_back(STEVILSKI_SESTAVI_ZNAKI[n % baza]);
10         n /= baza;
11     }
12     reverse(result.begin(), result.end());
13     return result;
14 }
15
16 string convert_fraction(int stevec, int imenovalec, int base) {
17     div_t d = div(stevec, imenovalec);
18     string result = convert_int(d.quot, base);
19     if (d.rem == 0) return result;
20
21     string decimalke; // decimalni del
22     result.push_back('.');
23     int mesto = 0;
24     map<int, int> spomin;
25     spomin[d.rem] = mesto;
26     while (d.rem != 0) { // pisno deljenje
27         mesto++;
28         d.rem *= base;
29         decimalke += STEVILSKI_SESTAVI_ZNAKI[d.rem / imenovalec];
30         d.rem %= imenovalec;
31         if (spomin.count(d.rem) > 0) { // periodično
32             result.append(decimalke.begin(), decimalke.begin() + spomin[d.rem]);
33             result.push_back('(');
34             result.append(decimalke.begin() + spomin[d.rem], decimalke.end());
35             result.push_back(')');
36             return result;
37         }
38         spomin[d.rem] = mesto;
39     }
40     result += decimalke;
41     return result; // končno decimalno stevilo
42 }

```

## 4.6 Eulerjeva funkcija $\phi$

**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>

```

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

```

## 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 praštevski delitelj. To se lahko uporablja za faktorizacijo števil in testiranje praštevskosti.

**Časovna zahtevnost:**  $O(n \log(n))$

**Prostorska zahtevnost:**  $O(n)$

**Testiranje na terenu:** UVa 10394

```

3 void eratosthenes_sieve(int n, vector<int>& is_prime, vector<int>& primes) {
4     is_prime.resize(n);
5     for (int i = 2; i < n+1; ++i) {
6         if (is_prime[i] == 0) {
7             is_prime[i] = i;
8             primes.push_back(i);
9         }
10        size_t j = 0;
11        while (j < primes.size() && primes[j] <= is_prime[i] && i * primes[j] <= n) {
12            is_prime[i * primes[j]] = primes[j];
13            j++;
14        }
15    }
16 }

```

## 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:**

```

3 int number_of_divisors(int n) {
4     int tau = 1;
5     int i = 2;
6     while (i * i <= n) {
7         int p = 0;
8         while (n % i == 0) {
9             n /= i;
10            p++;
11        }
12        tau *= p + 1;
13        i++;
14    }
15    if (n > 1) tau *= 2;
16    return tau;
17 }

```

## 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:  $O(\text{št. točk})$

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

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

```
6  const double pi = M_PI;
7  const double eps = 1e-9;
8  const double inf = numeric_limits<double>::infinity();
9
10 enum ITYPE : char { OK, NO, EQ };
11 typedef complex<double> P;
12
13 template<typename T>
14 struct line_t { // premica, dana z enačbo ax + by = c ali z dvema točkama
15     double a, b, c; // lahko tudi int
16     line_t() : a(0), b(0), c(0) {}
17     line_t(int A, int B, int C) {
18         if (A < 0 || (A == 0 && B < 0)) a = -A, b = -B, c = -C;
19         else a = A, b = B, c = C;
20         int d = gcd(gcd(abs(a), abs(b)), abs(c)); // same sign as A, if nonzero, else B, else C
21         if (d == 0) d = 1; // in case of 0 0 0 input
22         a /= d;
23         b /= d;
24         c /= d;
25     }
26     line_t(T A, T B, T C) {
27         if (A < 0 || (A == 0 && B < 0)) a = -A, b = -B, c = -C;
28         else a = A, b = B, c = C;
29     }
30     line_t(const P& p, const P& q) : line_t(imag(q-p), real(p-q), cross(p, q)) {}
31     P normal() const { return {a, b}; }
32     double value(const P& p) const { return dot(normal(), p) - c; }
33     bool operator<(const line_t<T>& line) const { // da jih lahko vržemo v set, če T = int
34         if (a == line.a) {
35             if (b == line.b) return c < line.c;
36             return b < line.b;
37         }
38         return a < line.a;
39     }
40     bool operator==(const line_t<T>& line) const {
41         return cross(normal(), line.normal()) < eps && c*line.b == b*line.c;
42     }
43 };
44 template<typename T>
45 ostream& operator<<(ostream& os, const line_t<T>& line) {
46     os << line.a << "x + " << line.b << "y == " << line.c; return os;
47 }
48
49 typedef line_t<double> L;
50
51 #endif // IMPLEMENTACIJA_GEOM_BASICS_H_

```

```
3  double dot(const P& p, const P& q) {
4      return p.real() * q.real() + p.imag() * q.imag();
5  }
```

```

6  double cross(const P& p, const P& q) {
7      return p.real() * q.imag() - p.imag() * q.real();
8  }
9  double cross(const P& p, const P& q, const P& r) {
10     return cross(q - p, r - q); // > 0 levo, < 0 desno, = 0 naravnost
11 }
12 // true is p->q->r is a left turn, straight line is not, if so, change to -eps
13 bool left_turn(const P& p, const P& q, const P& r) {
14     return cross(q-p, r-q) > eps;
15 }
16 P perp(const P& p) { // get left perpendicular vector
17     return P(-p.imag(), p.real());
18 }
19 int sign(double x) {
20     if (x < -eps) return -1;
21     if (x > eps) return 1;
22     return 0;
23 }
24 double polar_angle(const P& p) { // phi in [0, 2pi) or -1 for (0,0)
25     if (p == P(0, 0)) return -1;
26     double a = arg(p);
27     if (a < 0) a += 2*pi;
28     return a;
29 }
30 double area(const P& a, const P& b, const P& c) { // signed
31     return 0.5 * cross(a, b, c);
32 }
33 double area(const vector<P>& poly) { // signed
34     double A = 0;
35     int n = poly.size();
36     for (int i = 0; i < n; ++i) {
37         int j = (i+1) % n;
38         A += cross(poly[i], poly[j]);
39     }
40     return A/2;
41 }
42 double dist_to_line(const P& p, const L& line) {
43     return abs(line.value(p)) / abs(line.normal());
44 }
45 double dist_to_line(const P& t, const P& p1, const P& p2) { // t do premice p1p2
46     return abs(cross(p2-p1, t-p1)) / abs(p2-p1);
47 }
48 double dist_to_segment(const P& t, const P& p1, const P& p2) { // t do daljice p1p2
49     P s = p2 - p1;
50     P w = t - p1;
51     double c1 = dot(s, w);
52     if (c1 <= 0) return abs(w);
53     double c2 = norm(s);
54     if (c2 <= c1) return abs(t-p2);
55     return dist_to_line(t, p1, p2);
56 }
57 double great_circle_dist(const P& a, const P& b) { // pairs of (latitude, longitude) in radians
58     double R = 6371.0; // compute great circle distance
59     double u[3] = { cos(a.real()) * sin(a.imag()), cos(a.real()) * cos(a.imag()), sin(a.real()) };
60     double v[3] = { cos(b.real()) * sin(b.imag()), cos(b.real()) * cos(b.imag()), sin(b.real()) };
61     double dot = u[0]*v[0] + u[1]*v[1] + u[2]*v[2];
62     bool flip = false;
63     if (dot < 0.0) {
64         flip = true;
65         for (int i = 0; i < 3; i++) v[i] = -v[i];
66     }
67     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] };
68     double theta = asin(sqrt(cr[0]*cr[0] + cr[1]*cr[1] + cr[2]*cr[2]));
69     double len = theta * R;
70     if (flip) len = pi * R - len;
71     return len;
72 }
73 bool point_in_rect(const P& t, const P& p1, const P& p2) { // ali je t v pravokotniku p1p2
74     return min(p1.real(), p2.real()) <= t.real() && t.real() <= max(p1.real(), p2.real()) &&
75            min(p1.imag(), p2.imag()) <= t.imag() && t.imag() <= max(p1.imag(), p2.imag());
76 }
77 bool point_in_triangle(const P& t, const P& a, const P& b, const P& c) { // orientation independant
78     return abs(area(a, b, t)) + abs(area(a, c, t)) + abs(area(b, c, t)) // edge inclusive
79            - abs(area(a, b, c)) < eps;
80 }
81 pair<ITYPE, P> line_line_intersection(const L& p, const L& q) {
82     double det = cross(p.normal(), q.normal()); // če imata odvisni normali (ali smerna vektorja)
83     if (abs(det) < eps) { // paralel
84         if (abs(p.b*q.c - p.c*q.b) < eps && abs(p.a*q.c - p.c*q.a) < eps) {
85             return {EQ, P()}; // razmerja koeficientov se ujemajo
86         } else {

```

```

87         return {NO, P()};
88     }
89     } else {
90         return {OK, P(q.b*p.c - p.b*q.c, p.a*q.c - q.a*p.c) / det};
91     }
92 }
93 pair<ITYPE, P> line_segment_intersection(const L& p, const P& u, const P& v) {
94     double u_on = p.value(u);
95     double v_on = p.value(v);
96     if (abs(u_on) < eps && abs(v_on) < eps) return {EQ, u};
97     if (abs(u_on) < eps) return {OK, u};
98     if (abs(v_on) < eps) return {OK, v};
99     if ((u_on > eps && v_on < -eps) || (u_on < -eps && v_on > eps)) {
100         return line_line_intersection(p, L(u, v));
101     }
102     return {NO, P()};
103 }
104 pair<ITYPE, P> segment_segment_intersection(const P& p1, const P& p2, const P& q1, const P& q2) {
105     int o1 = sign(cross(p1, p2, q1)); // daljico p1p1 sekamo z q1q2
106     int o2 = sign(cross(p1, p2, q2));
107     int o3 = sign(cross(q1, q2, p1));
108     int o4 = sign(cross(q1, q2, p2));
109
110     // za pravo presečisce morajo biti o1, o2, o3, o4 != 0
111     // vemo da presečišče obstaja, tudi ce veljata samo prva dva pogoja
112     if (o1 != o2 && o3 != o4 && o1 != 0 && o2 != 0 && o3 != 0 && o4 != 0)
113         return line_line_intersection(L(p1, p2), L(q1, q2));
114
115     // EQ = se dotika samo z oglišcem ali sta vzporedni
116     if (o1 == 0 && point_in_rect(q1, p1, p2)) return {EQ, q1}; // q1 lezi na p
117     if (o2 == 0 && point_in_rect(q2, p1, p2)) return {EQ, q2}; // q2 lezi na p
118     if (o3 == 0 && point_in_rect(p1, q1, q2)) return {EQ, p1}; // p1 lezi na q
119     if (o4 == 0 && point_in_rect(p2, q1, q2)) return {EQ, p2}; // p2 lezi na q
120
121     return {NO, P()};
122 }
123 ITYPE point_in_poly(const P& t, const vector<P>& poly) {
124     int n = poly.size();
125     int cnt = 0;
126     double x2 = rand() % 100;
127     double y2 = rand() % 100;
128     P dalec(x2, y2);
129     for (int i = 0; i < n; ++i) {
130         int j = (i+1) % n;
131         if (dist_to_segment(t, poly[i], poly[j]) < eps) return EQ; // boundary
132         ITYPE tip = segment_segment_intersection(poly[i], poly[j], t, dalec).first;
133         if (tip != NO) cnt++; // ne testiramo, ali smo zadeli oglišce, upamo da nismo
134     }
135     if (cnt % 2 == 0) return NO;
136     else return OK;
137 }
138 pair<P, double> get_circle(const P& p, const P& q, const P& r) { // circle through 3 points
139     P v = q-p;
140     P w = q-r;
141     if (abs(cross(v, w)) < eps) return {P(), 0};
142     P x = (p+q)/2.0, y = (q+r)/2.0;
143     ITYPE tip;
144     P intersection;
145     tie(tip, intersection) = line_line_intersection(L(x, x+perp(v)), L(y, y+perp(w)));
146     return {intersection, abs(intersection-p)};
147 }
148 // circle through 2 points with given r, to the left of pq
149 P get_circle(const P& p, const P& q, double r) {
150     double d = norm(p-q);
151     double h = r*r / d - 0.25;
152     if (h < 0) return P(inf, inf);
153     h = sqrt(h);
154     return (p+q) / 2.0 + h * perp(q-p);
155 }

```

## 5.2 Konveksna ovojnica

**Vhod:** Seznam  $n$  točk.

**Izhod:** Najkrajši seznam  $h$  točk, ki napenjaajo 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

```
3  typedef complex<double> P; // ali int
4
5  bool compare(const P& a, const P& b, const P& m) {
6      double det = cross(a, m, b);
7      if (abs(det) < eps) return abs(a-m) < abs(b-m);
8      return det < 0;
9  }
10
11 vector<P> convex_hull(vector<P>& points) { // vector is modified
12     if (points.size() <= 2) return points;
13     P m = points[0]; int mi = 0;
14     int n = points.size();
15     for (int i = 1; i < n; ++i) {
16         if (points[i].imag() < m.imag() ||
17             (points[i].imag() == m.imag() && points[i].real() < m.real())) {
18             m = points[i];
19             mi = i;
20         }
21     } // m = spodnja leva
22
23     swap(points[0], points[mi]);
24     sort(points.begin()+1, points.end(),
25          [&m](const P& a, const P& b) { return compare(a, b, m); });
26
27     vector<P> hull;
28     hull.push_back(points[0]);
29     hull.push_back(points[1]);
30
31     for (int i = 2; i < n; ++i) { // tocke, ki so na ovojnici spusti, ce jih hoces daj -eps
32         while (hull.size() >= 2 && cross(hull.end()[-2], hull.end()[-1], points[i]) < eps) {
33             hull.pop_back(); // right turn
34         }
35         hull.push_back(points[i]);
36     }
37
38     return hull;
39 }
```

### 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>

```
3  typedef complex<int> P;
4
5  struct vert { // vertical sweep line element
6      int x, s, e;
7      bool start;
8      vert(int a, int b, int c, bool d) : x(a), s(b), e(c), start(d) {}
9      bool operator<(const vert& o) const {
10         return x < o.x;
11     }
12 };
13
14 vector<int> points;
15
16 struct Node { // segment tree
17     int s, e, m, c, a; // start, end, middle, count, area
18     Node *left, *right;
19     Node(int s_, int e_) : s(s_), e(e_), m((s+e)/2), c(0), a(0), left(nullptr), right(nullptr) {
20         if (e-s == 1) return;
21         left = new Node(s, m);
22         right = new Node(m, e);
23     }
24     int add(int f, int t) { // returns area
```



```

25     if (f <= s && e <= t) {
26         c++;
27         return a = points[e] - points[s];
28     }
29     if (f < m) left->add(f, t);
30     if (t > m) right->add(f, t);
31     if (c == 0) a = left->a + right->a; // če nimam lastnega intervala, izračunaj
32     return a;
33 }
34 int remove(int f, int t) { // returns area
35     if (f <= s && e <= t) {
36         c--;
37         if (c == 0) { // če nima lastnega intervala
38             if (left == nullptr) a = 0; // če je list je area 0
39             else a = left->a + right->a; // če ne je vsota otrok
40         }
41         return a;
42     }
43     if (f < m) left->remove(f, t);
44     if (t > m) right->remove(f, t);
45     if (c == 0) a = left->a + right->a;
46     return a;
47 }
48 };
49
50 int rectangle_union_area(const vector<pair<P, P>>& rects) {
51     int n = rects.size();
52
53     vector<vert> verts; verts.reserve(2*n);
54     points.resize(2*n); // vse točke čez katere napenjamo intervale (stranice)
55
56     P levo_spodaj, desno_zgoraj; // pravokotniki so podani tako
57     for (int i = 0; i < n; ++i) {
58         tie(levo_spodaj, desno_zgoraj) = rects[i];
59         int a = levo_spodaj.real();
60         int c = desno_zgoraj.real();
61         int b = levo_spodaj.imag();
62         int d = desno_zgoraj.imag();
63         verts.push_back(vert(a, b, d, true));
64         verts.push_back(vert(c, b, d, false));
65         points[2*i] = b;
66         points[2*i+1] = d;
67     }
68
69     sort(verts.begin(), verts.end());
70     sort(points.begin(), points.end());
71     points.resize(unique(points.begin(), points.end())-points.begin()); // zberemo enake
72
73     Node * sl = new Node(0, points.size()); // sweepline segment tree
74
75     int area = 0, height = 0; // area = total area. height = trenutno pokrita višina
76     int px = -(1 << 30);
77     for (int i = 0; i < 2*n; ++i) {
78         area += (verts[i].x-px)*height; // trenutno pometena area
79
80         int s = lower_bound(points.begin(), points.end(), verts[i].s)-points.begin();
81         int e = lower_bound(points.begin(), points.end(), verts[i].e)-points.begin();
82         if (verts[i].start)
83             height = sl->add(s, e); // segment tree sprejme indexe, ne koordinat
84         else
85             height = sl->remove(s, e);
86         px = verts[i].x;
87     }
88
89     return area;
90 }

```

## 5.4 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

```
3 typedef complex<double> P;
4 typedef vector<P>::iterator RAI; // or use template
5
6 bool byx(const P& a, const P& b) { return a.real() < b.real(); }
7 bool byy(const P& a, const P& b) { return a.imag() < b.imag(); }
8
9 double najblizji_tocki_bf(RAI s, RAI e) {
10     double m = numeric_limits<double>::max();
11     for (RAI i = s; i != e; ++i)
12         for (RAI j = i+1; j != e; ++j)
13             m = min(m, norm(*i - *j));
14     return m;
15 }
16 double najblizji_tocki_divide(RAI s, RAI e, const vector<P>& py) {
17     if (e - s < 50) return najblizji_tocki_bf(s, e);
18
19     size_t m = (e-s) / 2;
20     double d1 = najblizji_tocki_divide(s, s+m, py);
21     double d2 = najblizji_tocki_divide(s+m, e, py);
22     double d = min(d1, d2);
23     // merge
24     double meja = (s[m].real() + s[m+1].real()) / 2;
25     int n = py.size();
26     for (double i = 0; i < n; ++i) {
27         if (meja-d < py[i].real() && py[i].real() <= meja+d) {
28             double j = i+1;
29             double c = 0;
30             while (j < n && c < 7) { // navzdol gledamo le 7 ali dokler ni dlje od d
31                 if (meja-d < py[j].real() && py[j].real() <= meja+d) {
32                     double nd = norm(py[j]-py[i]);
33                     d = min(d, nd);
34                     if (py[j].imag() - py[i].imag() > d) break;
35                     ++c;
36                 }
37                 ++j;
38             }
39         }
40     }
41     return d;
42 }
43 double najblizji_tocki(const vector<P>& points) {
44     vector<P> px = points, py = points;
45     sort(px.begin(), px.end(), byx);
46     sort(py.begin(), py.end(), byy);
47     return najblizji_tocki_divide(px.begin(), px.end(), py);
48 }
```

## 6 Matematika

Vrste:

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} \quad \sum_{i=1}^n i^2 = \frac{n(n+1)(2n+1)}{6} \quad \sum_{i=1}^n i^3 = \frac{n^2(n+1)^2}{4}$$
$$\sum_{i=1}^n q^i = q \frac{q^n - 1}{q - 1} \quad \sum_{i=1}^n i q^i = \frac{n q^{n+2} - (n+1) q^{n+1} + q}{(q-1)^2}$$

Kombinatorika:

$$\binom{n}{k} = \sum_{i=1}^k \frac{n-i}{i} \quad \binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$
$$C_n = \frac{1}{n+1} \binom{2n}{n} = \sum_{i=0}^{n-1} C_i C_{n-1-i} = \prod_{i=2}^n \frac{n+i}{i} = \frac{2(2n-1)}{n+1} C_{n-1}$$
$$F_n = F_{n-1} + F_{n-2} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}_{11}^{n-1} \quad F_m F_n + F_{m-1} F_{n-1} = F_{m+n-1} \quad F_m F_{n+1} + F_{m-1} F_n = F_{m+n}$$

### Številске ocene:

$i$	$2^i$	$\log_{10}(2^i)$	$i!$	$\log_{10}(i!)$	$10^i$ int-ov	$C_i$	$F_i$
1	2	0,30	1	0,00	40 B	1	1
2	4	0,60	2	0,30	400 B	2	1
3	8	0,90	6	0,77	3,9 kiB	5	2
4	16	1,20	24	1,38	39 kiB	14	3
5	32	1,50	120	2,07	400 kiB	42	5
6	64	1,80	720	2,85	3,8 MiB	132	8
7	128	2,10	5040	3,70	38 MiB	429	13
8	256	2,40	40320	4,60	380 MiB	1430	21
9	512	2,70	362880	5,55	3,7 GiB	4862	34
10	1.024	3,01	3628800	6,55	.	16796	55
11	2.048	3,31	39916800	7,60	.	58786	89
12	4.096	3,61	.	8,68	.	208012	144
13	8.192	3,91	.	9,79		742900	233
14	16.384	4,21	.	10,94		2674440	377
15	32.768	4,51		12,11		9694845	610
16	65.536	4,81		13,32		.	987
17	131.072	5,11		14,55		.	1597
18	262.144	5,41		15,80		.	2584
19	524.288	5,71		17,08			4181
20	1.048.576	6,02		18,38			6765
21	2.097.152	6,32		19,70			10946
22	4.194.304	6,62		21,05			17711
23	8.388.608	6,92		22,41			28657
24	16.777.216	7,22		23,79			46368
25	33.554.432	7,52		25,19			75025
26	67.108.864	7,82		26,60			121393
27	134.217.728	8,12		28,03			196418
28	268.435.456	8,42		29,48			317811
29	536.870.912	8,72		30,94			514229
30	1.073.741.824	9,03		32,42			832040
31	2.147.483.648	9,33		33,91			1346269
32	4.294.967.296	9,63		35,42	Google		2178309

### Pascalov trikotnik:

0:								1											
1:								1		1									
2:							1		2		1								
3:						1		3		3		1							
4:					1		4		6		4		1						
5:				1		5		10		10		5		1					
6:				1		6		15		20		15		6		1			
7:			1		7		21		35		35		21		7		1		
8:			1		8		28		56		70		56		28		8		1
9:			1		9		36		84		126		126		84		36		9
10:			1		10		45		120		210		252		210		120		45
11:			1		11		55		165		330		462		462		330		165
12:			1		12		66		220		495		792		924		792		495
13:			1		13		78		286		715		1287		1716		1716		1287