

# 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: 9. april 2015

# Kazalo

<b>1</b>	<b>Grafi</b>	<b>3</b>
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
<b>2</b>	<b>Podatkovne strukture</b>	<b>11</b>
2.1	Statično binarno iskalno drevo . . . . .	11
2.2	Drevo segmentov . . . . .	12
2.3	Avl drevo . . . . .	14
2.4	Fenwickovo drevo . . . . .	15
2.5	Fenwickovo drevo ( <b>n</b> -dim) . . . . .	16
<b>3</b>	<b>Algoritmi</b>	<b>17</b>
3.1	Najdaljše skupno podzaporedje . . . . .	17
3.2	Najdaljše naraščajoče podzaporedje . . . . .	18
3.3	Najdaljši strnjen palindrom . . . . .	19
3.4	Podseznam z največjo vsoto . . . . .	19
3.5	Leksikografsko minimalna rotacija . . . . .	20
3.6	BigInt in Karatsuba . . . . .	21
<b>4</b>	<b>Teorija števil</b>	<b>23</b>
4.1	Evklidov algoritem . . . . .	23
4.2	Razširjen Evklidov algoritem . . . . .	23
4.3	Kitajski izrek o ostankih . . . . .	24
4.4	Hitro potenciranje . . . . .	24
4.5	Številski sestavi . . . . .	25
4.6	Eulerjeva funkcija $\phi$ . . . . .	25
4.7	Eratostenovo rešeto . . . . .	26
<b>5</b>	<b>Geometrija</b>	<b>26</b>
5.1	Osnove . . . . .	26
5.2	Konveksna ovojnica . . . . .	29
5.3	Ploščina unije pravokotnikov . . . . .	30
5.4	Najbližji par točk v ravnini . . . . .	31

# 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](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](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](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](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](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 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)

```

1  namespace {
2  int depth = 10; // depth of the tree
3  int n = 1 << 10; // number of different elements stored
4  vector<int> tree(2*n); // number of elements in the tree = tree[0]
5  }
6
7  void insert(int idx, int val = 1) {
8      int i = n - 1 + idx;
9      while (i > 0) {
10         tree[i] += val;
11         i--;
12         i >>= 1;
13     }
14     tree[0] += val;
15 }
16
17 int get_kth(int k) {
18     int i = 0;
19     while (i < n - 1) {
20         int lc = tree[2*i+1];
21         if (lc <= k) {
22             i = 2*i + 2;
23             k -= lc;
24         } else {
25             i = 2*i + 1;

```

```

26     }
27 }
28 return i - n + 1;
29 }
30
31 bool remove(int idx) {
32     if (tree[n-1 + idx] <= 0) return false;
33     insert(idx, -1);
34     return true;
35 }
36
37 void print() {
38     for (int i = 0; i <= depth; ++i) {
39         cout << string((1 << (depth - i)) - 1, ' ');
40         for (int j = (1 << i) - 1; j < (1 << (i+1)) - 1; ++j) {
41             cout << tree[j] << string((1 << (depth - i + 1)) - 1, ' ');
42         }
43         cout << '\n';
44     }
45 }

```

## 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>

```

1  struct Node { // static division points, [l, r] intervals
2      int l, m, r;
3      Node* left, *right;
4      int here_count, total_count;
5      Node(int ll, int rr) : l(ll), m((ll+rr) / 2), r(rr),
6                          left(nullptr), right(nullptr), here_count(0), total_count(0) {}
7      int insert(int f, int t, int inc = 1) { // insert inc elemnts into [f, t]. use for removal too
8          if (f <= l && r <= t) {
9              here_count += inc;
10             int lb = max(l, f), rb = min(r, t);
11             int inserted = (rb - lb + 1) * inc;
12             total_count += inserted;
13             return inserted;
14         }
15         int inserted = 0;
16         if (f <= m) {
17             if (left == nullptr) left = new Node(l, m);
18             inserted += left->insert(f, t, inc);
19         }
20         if (m + 1 <= t) {
21             if (right == nullptr) right = new Node(m + 1, r);
22             inserted += right->insert(f, t, inc);
23         }
24         total_count += inserted;
25
26         if (left != nullptr && right != nullptr) { // move full levels up (speedup, ne rabiš)
27             int child_here_count = min(left->here_count, right->here_count);
28             left->here_count -= child_here_count;
29             right->here_count -= child_here_count;

```

```

30         here_count += child_here_count;
31         left->total_count -= (m - l + 1) * child_here_count;
32         right->total_count -= (r - m) * child_here_count;
33     } // end speed up
34     return inserted;
35 }
36
37 int count(int f, int t) { // count on interval [f, t]
38     if (f <= l && r <= t) return total_count;
39     int sum = 0, lb = max(l, f), rb = min(r, t);
40     if (f <= m && left != nullptr) sum += left->count(f, t);
41     if (m + 1 <= t && right != nullptr) sum += right->count(f, t);
42     return sum + (rb - lb + 1) * here_count;
43 }
44
45 int get_kth(int k, int parent_count = 0) { // zero based
46     int above_count = here_count + parent_count;
47     int lc = above_count * (m - l + 1) + get_cnt(left);
48     if (k < lc) {
49         if (left == nullptr) return l + k/above_count;
50         return left->get_kth(k, above_count);
51     } else {
52         k -= lc;
53         if (right == nullptr) return m + 1 + k/above_count;
54         return right->get_kth(k, above_count);
55     }
56 }
57
58 void print(int l, int r) {
59     printf("[%d, %d]: here: %d, total: %d\n", l, r, here_count, total_count);
60     if (left) left->print(l, m);
61     if (right) right->print(m+1, r);
62 }
63
64 private:
65     int get_cnt(Node* left) {
66         return (left == nullptr) ? 0 : left->total_count;
67     }
68 };
69
70 // poenostavitev, ce ne rabimo intervalov
71 struct SimpleNode { // static division points
72     SimpleNode* left, *right;
73     int cnt;
74     SimpleNode() : left(nullptr), right(nullptr), cnt(0) {}
75     void insert(int l, int r, int val) {
76         if (l == r) { cnt++; return; }
77         int mid = (l + r) / 2; // watch overflow
78         if (val <= mid) {
79             if (left == nullptr) left = new SimpleNode();
80             left->insert(l, mid, val);
81         } else {
82             if (right == nullptr) right = new SimpleNode();
83             right->insert(mid + 1, r, val);
84         }
85         cnt++;
86     }
87
88     int pop_kth(int l, int r, int k) { // one based
89         if (l == r) { cnt--; return l; }
90         int mid = (l + r) / 2;
91         if (k <= get_cnt(left)) {
92             cnt--;
93             return left->pop_kth(l, mid, k);
94         } else {
95             k -= get_cnt(left);
96             cnt--;
97             return right->pop_kth(mid + 1, r, k);
98         }
99     }
100
101     private:
102         int get_cnt(SimpleNode* x) {
103             if (x == nullptr) return 0;
104             return x->cnt;
105         }
106 };
107 #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 neke globalen root in private metode.

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

```

58     updateHeight(node->left); updateHeight(node);
59 }
60 void rotateRight(AvlNode<T>*& node) {
61     AvlNode<T>* L = node->left;
62     node->size -= size(L->left) + L->count; L->size += size(node->right) + node->count;
63     node->left = L->right; L->right = node; node = L;
64     updateHeight(node->right); updateHeight(node);
65 }
66 void balance(AvlNode<T>*& node) {
67     int b = getBalance(node);
68     if (b == 2) {
69         if (getBalance(node->left) == -1) rotateLeft(node->left);
70         rotateRight(node);
71     } else if (b == -2) {
72         if (getBalance(node->right) == 1) rotateRight(node->right);
73         rotateLeft(node);
74     } else {
75         updateHeight(node);
76     }
77 }
78 AvlNode<T>* insert(const T& val, AvlNode<T>*& node) {
79     if (node == nullptr) return node = new AvlNode<T>(val);
80     node->size++;
81     AvlNode<T>* return_node = node;
82     if (val < node->value) return_node = insert(val, node->left);
83     else if (node->value == val) node->count++;
84     else if (node->value < val) return_node = insert(val, node->right);
85     balance(node);
86     return return_node;
87 }
88 bool erase(const T& val, AvlNode<T>*& node) {
89     if (node == nullptr) return false;
90     if (val < node->value) {
91         if (erase(val, node->left)) {
92             node->size--;
93             return true;
94         }
95     } else if (node->value < val) {
96         if (erase(val, node->right)) {
97             node->size--;
98             return true;
99         }
100     } else if (node->value == val && node->count > 0) {
101         node->count--;
102         node->size--;
103         return true;
104     }
105     return false;
106 }
107 const AvlNode<T>* get_nth(const AvlNode<T>* const& node, size_t n) const {
108     size_t left_size = size(node->left);
109     if (n < left_size) return get_nth(node->left, n);
110     else if (n < left_size + node->count) return node;
111     else if (n < node->size) return get_nth(node->right, n - left_size - node->count);
112     else return nullptr;
113 }
114 const AvlNode<T>* find(const AvlNode<T>* const& node, const T& value) const {
115     if (node == nullptr) return nullptr;
116     if (value < node->value) return find(node->left, value);
117     else if (value == node->value) return node;
118     else return find(node->right, value);
119 }
120
121 AvlNode<T>* root;
122 };
123
124 template<typename T>
125 ostream& operator<<(ostream& os, const AvlTree<T>& tree) {
126     if (tree.root == nullptr) os << "Tree empty";
127     else tree.root->print(os);
128     return os;
129 }
130 #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 posodablјati elemente in odgovarјati 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 }
```

## 2.5 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)

```

1  typedef vector<vector<vector<int>>> vvvi;
2
3  int sum(int x, int y, int z, const vvvi& 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, vvvi& 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 vvvi& 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 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

```

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

```

```

33     for (int j = 1; j <= m; ++j)
34         if (a[i-1] == b[j-1])
35             c[f][j] = c[!f][j-1] + 1;
36         else
37             c[f][j] = max(c[f][j-1], c[!f][j]);
38     f = !f;
39 }
40 return c[!f][m];
41 }

```

## 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.

```

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

```

```

47         }
48
49         if (m[i] > max_length) {
50             best_end = i;
51             max_length = m[i];
52         }
53     }
54 }
55 vector<int> lis;
56 for (int i = best_end; i != -1; i = prev[i]) lis.push_back(a[i]);
57 reverse(lis.begin(), lis.end());
58 return lis;
59 }

```

### 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/>

```

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

```

### 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>

```
1  int maximum_subarray(const vector<int>& a) { // glej komentarje ce ne dovolimo prazne vsote
2      int max_ending_here = 0, max_so_far = 0; // A[0]
3      for (int x : a) { // a[1:]
4          max_ending_here = max(0, max_ending_here + x);
5          max_so_far = max(max_so_far, max_ending_here);
6      }
7      return max_so_far;
8  }
9
10 vector<int> maximum_subarray_extract(const vector<int>& a) {
11     int max_ending_here = 0, max_so_far = 0;
12     int idx_from = 0, total_to = 0, total_from = 0;
13     for (size_t i = 0; i < a.size(); ++i) {
14         if (max_ending_here + a[i] > 0) {
15             max_ending_here += a[i];
16         } else {
17             idx_from = i + 1;
18         }
19         if (max_ending_here > max_so_far) {
20             total_from = idx_from;
21             total_to = i + 1;
22             max_so_far = max_ending_here;
23         }
24     }
25     return vector<int>(a.begin() + total_from, a.begin() + total_to);
26 }
27
28 int maximum_subarray(const vector<int>& a, int k) { // k = 1 ... n(n-1)/2
29     int n = a.size();
30     vector<int> s(n+1, 0);
31     vector<pair<int, int>> p(n+1);
32     priority_queue<tuple<int, int, int>> q;
33     for (int i = 0, m = 0; i < n; ++i) {
34         s[i+1] = s[i] + a[i];
35         if (s[m] > s[i]) m = i;
36         p[i+1] = make_pair(s[i+1], i+1);
37         q.push(make_tuple(s[i+1]-s[m], i+1, m));
38     }
39     sort(p.begin(), p.end());
40     vector<int> ss(n+1);
41     for (int i = 0; i <= n; ++i)
42         ss[p[i].second] = i;
43
44     int v = -1, i, j;
45     for (int l = 1; l <= k; ++l) {
46         tie(v, i, j) = q.top();
47         q.pop();
48
49         for (int m = ss[j] + 1; m <= n; ++m) {
50             if (p[m].second < i) {
51                 q.push(make_tuple(s[i]-p[m].first, i, p[m].second));
52                 break;
53             }
54         }
55     }
56     return v;
57 }
```

### 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  $n$ .

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

## 3.6 BigInt in Karatsuba

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

**Operacije:** Seštevanje, odštevanje, množenje, primerjanje.

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

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

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

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

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

```
1  template<typename T>
2  struct Number {
3      bool sign = true; // true = 1 = +, false = 0 = -
4      deque<T> data;
5      static const int base = 10; // bi se lahko spremenilo samo I/O nebi delal, matematika bi
6      static const int KARATSUBA_LIMIT = 2; // kdaj preklopi na brute force množenje
7
8      Number() {} // default constructor, positive zero
9      Number(const deque<T>& a, bool s = 1) : sign(s), data(a) { clear_zeros(); }
10     Number(deque<T>&& a, bool s = 1) : sign(s), data(a) { clear_zeros(); }
11     Number(const string& s) { from_string(s); }
12     void from_string(const string& s) {
13         if (s.size() == 0) data = {0};
14         int i = 0;
15         if (s[0] == '+') sign = 1, i = 1;
16         if (s[0] == '-') sign = 0, i = 1;
17         int l = s.size(); data.resize(l-i);
18         for (; i < l; ++i) {
19             if (!('0' <= s[i] && s[i] <= '9')) return; // silent quit po prvi ne številki
20             data[l-i-1] = s[i] - '0';
21         }
22         clear_zeros();
23     }
24     string to_string() const {
25         if (data.empty()) return "0";
26         string s = (sign ? "" : "-");
27         for (int i = data.size() - 1; i >= 0; --i)
28             s.push_back('0' + data[i]);
29         return s;
30     }
31     Number operator+(const Number& o) const { // remove signs if using for positive only
32         if (sign == 0) return -((- *this) + (-o));
```

```

33     if (sign == 1 && o.sign == 0) return (*this < -o) ? -((-o) - *this) : *this - (-o);
34     bool carry = false;
35     int i = 0, j = 0, n = data.size(), m = o.data.size();
36     deque<T> r;
37     while (i < n || j < m) {
38         T c = ((i < n) ? data[i++] : 0) + ((j < m) ? o.data[j++] : 0) + carry;
39         carry = (c >= base);
40         r.push_back(c % base);
41     }
42     if (carry) r.push_back(1);
43     return Number(move(r));
44 }
45 Number operator-() const { return Number(data, !sign); }
46 bool operator==(const Number& o) const { return sign == o.sign && data == o.data; }
47 bool operator<(const Number& o) const {
48     if (sign == o.sign) {
49         if (sign == 0) return -o < -*this;
50         if (data.size() == o.data.size()) {
51             for (int i = data.size() - 1; i >= 0; --i)
52                 if (data[i] == o.data[i]) continue;
53                 else return data[i] < o.data[i];
54         }
55         return data.size() < o.data.size();
56     }
57     return sign < o.sign;
58 }
59 Number& operator+=(const Number& o) { // lahko tudi s +. Samo za pozitivne.
60     bool carry = false;
61     int i = 0, j = 0, n = data.size(), m = o.data.size();
62     while (i < n || j < m) {
63         if (i < n && j < m) data[i] += o.data[j++] + carry;
64         else if (i < n) data[i] += carry;
65         else data.push_back(o.data[j++] + carry);
66         carry = data[i] / base;
67         data[i++] %= base;
68     }
69     if (carry) data.push_back(1);
70     clear_zeros();
71     return *this;
72 }
73 Number operator*(const T& o) const { // z eno številko
74     deque<T> r;
75     int carry = 0, n = data.size();
76     for (int i = 0; i < n; ++i) {
77         T c = data[i]*o + carry;
78         carry = c / base;
79         c %= base;
80         r.push_back(c);
81     }
82     if (carry) r.push_back(carry);
83     return Number(move(r), sign);
84 }
85 Number operator<<(int n) const { // na začetek dodamo n ničel
86     deque<T> r(n, 0);
87     r.insert(r.end(), data.begin(), data.end());
88     return Number(move(r));
89 }
90 Number operator*(const Number<T>& o) const {
91     if (sign == 0 && o.sign == 0) return ((*this) * (-o));
92     if (sign == 0 && o.sign == 1) return -((*this) * o);
93     if (sign == 1 && o.sign == 0) return -(*this * (-o));
94     Number r;
95     int m = o.data.size();
96     for (int i = 0; i < m; ++i)
97         r += (*this*o.data[i] << i);
98     return r;
99 }
100 Number operator-(const Number& o) const {
101     deque<T> r;
102     bool carry = false;
103     int i = 0, j = 0, n = data.size(), m = o.data.size();
104     while (i < n || j < m) {
105         T c = data[i++] + base - ((j < m) ? o.data[j++] : 0) - carry;
106         carry = 1 - c / base;
107         c %= base;
108         r.push_back(c);
109     }
110     return Number(move(r));
111 }
112
113 private:

```

```

114     void clear_zeros() {
115         while (data.size() > 0 && data.back() == 0) data.pop_back();
116         if (data.empty()) sign = 1;
117     }
118 };
119
120 template<typename T> // karatsuba algorithm
121 Number<T> karatsuba(const Number<T>& a, const Number<T>& b) {
122     if (a.data.size() <= Number<T>::KARATSUBA_LIMIT || b.data.size() <= Number<T>::KARATSUBA_LIMIT)
123         return a*b;
124
125     if (a.sign == 0 && b.sign == 0) return ((-a) * (-b));
126     if (a.sign == 0 && b.sign == 1) return -((-a) * b);
127     if (a.sign == 1 && b.sign == 0) return -(a * (-b));
128
129     Number<T> a0, a1, b0, b1, c0, c1, c2;
130     int m = min(a.data.size(), b.data.size())/2; // choose m carefully
131
132     a0.data.assign(a.data.begin(), a.data.begin()+m);
133     a1.data.assign(a.data.begin()+m, a.data.end());
134     b0.data.assign(b.data.begin(), b.data.begin()+m);
135     b1.data.assign(b.data.begin()+m, b.data.end());
136
137     c2 = karatsuba(a1, b1);
138     c0 = karatsuba(a0, b0);
139     c1 = karatsuba(a0+a1, b0+b1) - c0 - c2;
140
141     return (c2 << 2*m) + (c1 << m) + c0;
142 }
143 #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:**  $O(1)$

```

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  }

```

### 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

```

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 reminder
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 }

```

### 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. 23)

**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_remainder_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  }

```

### 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)

```

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 }

```

## 4.5 Številski sestavi

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

**Izhod:** Število  $n$  ali  $\frac{p}{q}$  predstavljeno v izbranem sestavu z izbranimi 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.

```

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 }

```

## 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>

```
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 }
```

## 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

```
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 }
```

## 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...

```

1  const double pi = M_PI;
2  const double eps = 1e-9;
3  const double inf = numeric_limits<double>::infinity();
4
5  enum ITYPE : char { OK, NO, EQ };
6  typedef complex<double> P;
7
8  template<typename T>
9  struct line_t { // premica, dana z enačbo ax + by = c ali z dvema točkama
10     double a, b, c; // lahko tudi int
11     line_t() : a(0), b(0), c(0) {}
12     line_t(int A, int B, int C) {
13         if (A < 0 || (A == 0 && B < 0)) a = -A, b = -B, c = -C;
14         else a = A, b = B, c = C;
15         int d = gcd(gcd(abs(a), abs(b)), abs(c)); // same sign as A, if nonzero, else B, else C
16         if (d == 0) d = 1; // in case of 0 0 0 input
17         a /= d;
18         b /= d;
19         c /= d;
20     }
21     line_t(T A, T B, T C) {
22         if (A < 0 || (A == 0 && B < 0)) a = -A, b = -B, c = -C;
23         else a = A, b = B, c = C;
24     }
25     line_t(const P& p, const P& q) : line_t(imag(q-p), real(p-q), cross(p, q)) {}
26     P normal() const { return {a, b}; }
27     double value(const P& p) const { return dot(normal(), p) - c; }
28     bool operator<(const line_t<T>& line) const { // da jih lahko vržemo v set, če T = int
29         if (a == line.a) {
30             if (b == line.b) return c < line.c;
31             return b < line.b;
32         }
33         return a < line.a;
34     }
35     bool operator==(const line_t<T>& line) const {
36         return cross(normal(), line.normal()) < eps && c*line.b == b*line.c;
37     }
38 };
39 template<typename T>
40 ostream& operator<<(ostream& os, const line_t<T>& line) {
41     os << line.a << "x + " << line.b << "y == " << line.c; return os;
42 }
43
44 typedef line_t<double> L;
45
46 #endif // IMPLEMENTACIJA_GEOM_BASICS_H_

```

```

1  double dot(const P& p, const P& q) {
2      return p.real() * q.real() + p.imag() * q.imag();
3  }
4  double cross(const P& p, const P& q) {
5      return p.real() * q.imag() - p.imag() * q.real();
6  }
7  double cross(const P& p, const P& q, const P& r) {
8      return cross(q - p, r - q); // > 0 levo, < 0 desno, = 0 naravnost
9  }
10 // true is p->q->r is a left turn, straight line is not, if so, change to -eps
11 bool left_turn(const P& p, const P& q, const P& r) {

```

```

12     return cross(q-p, r-q) > eps;
13 }
14 P perp(const P& p) { // get left perpendicular vector
15     return P(-p.imag(), p.real());
16 }
17 int sign(double x) {
18     if (x < -eps) return -1;
19     if (x > eps) return 1;
20     return 0;
21 }
22 double polar_angle(const P& p) { // phi in [0, 2pi) or -1 for (0,0)
23     if (p == P(0, 0)) return -1;
24     double a = arg(p);
25     if (a < 0) a += 2*pi;
26     return a;
27 }
28 double area(const P& a, const P& b, const P& c) { // signed
29     return 0.5 * cross(a, b, c);
30 }
31 double area(const vector<P>& poly) { // signed
32     double A = 0;
33     int n = poly.size();
34     for (int i = 0; i < n; ++i) {
35         int j = (i+1) % n;
36         A += cross(poly[i], poly[j]);
37     }
38     return A/2;
39 }
40 double dist_to_line(const P& p, const L& line) {
41     return abs(line.value(p)) / abs(line.normal());
42 }
43 double dist_to_line(const P& t, const P& p1, const P& p2) { // t do premice p1p2
44     return abs(cross(p2-p1, t-p1)) / abs(p2-p1);
45 }
46 double dist_to_segment(const P& t, const P& p1, const P& p2) { // t do daljice p1p2
47     P s = p2 - p1;
48     P w = t - p1;
49     double c1 = dot(s, w);
50     if (c1 <= 0) return abs(w);
51     double c2 = norm(s);
52     if (c2 <= c1) return abs(t-p2);
53     return dist_to_line(t, p1, p2);
54 }
55 double great_circle_dist(const P& a, const P& b) { // pairs of (latitude, longitude) in radians
56     // compute great circle distance
57     double u[3] = { cos(a.real()) * sin(a.imag()), cos(a.real()) * cos(a.imag()), sin(a.real()) };
58     double v[3] = { cos(b.real()) * sin(b.imag()), cos(b.real()) * cos(b.imag()), sin(b.real()) };
59     double dot = u[0]*v[0] + u[1]*v[1] + u[2]*v[2];
60     bool flip = false;
61     if (dot < 0.0) {
62         flip = true;
63         for (int i = 0; i < 3; i++) v[i] = -v[i];
64     }
65     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] };
66     double theta = asin(sqrt(cr[0]*cr[0] + cr[1]*cr[1] + cr[2]*cr[2]));
67     double len = theta * R;
68     if (flip) len = pi * R - len;
69     return len;
70 }
71 bool point_in_rect(const P& t, const P& p1, const P& p2) { // ali je t v pravokotniku p1p2
72     return min(p1.real(), p2.real()) <= t.real() && t.real() <= max(p1.real(), p2.real()) &&
73         min(p1.imag(), p2.imag()) <= t.imag() && t.imag() <= max(p1.imag(), p2.imag());
74 }
75 bool point_in_triangle(const P& t, const P& a, const P& b, const P& c) { // orientation independant
76     return abs(area(a, b, t)) + abs(area(a, c, t)) + abs(area(b, c, t)) // edge inclusive
77         - abs(area(a, b, c)) < eps;
78 }
79 pair<ITYPE, P> line_line_intersection(const L& p, const L& q) {
80     double det = cross(p.normal(), q.normal()); // če imata odvisni normali (ali smerna vektorja)
81     if (abs(det) < eps) { // paralel
82         if (abs(p.b*q.c - p.c*q.b) < eps && abs(p.a*q.c - p.c*q.a) < eps) {
83             return {EQ, P()}; // razmerja koeficientov se ujemajo
84         } else {
85             return {NO, P()};
86         }
87     } else {
88         return {OK, P(q.b*p.c - p.b*q.c, p.a*q.c - q.a*p.c) / det};
89     }
90 }
91 pair<ITYPE, P> line_segment_intersection(const L& p, const P& u, const P& v) {
92     double u_on = p.value(u);

```

```

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

```

## 5.2 Konveksna ovojnica

**Vhod:** Seznam  $n$  točk.

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

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

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

**Potrebuje:** Vektorski produkt, str. 26.

**Testiranje na terenu:** UVa 681

```

1 typedef complex<double> P; // ali int
2

```

```

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

```

### 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>

```

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(nullptr), right(nullptr) {
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 (f <= s && 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 (f <= s && e <= t) {
34             c--;
35             if (c == 0) { // če nima lastnega intervala
36                 if (left == nullptr) a = 0; // če je list 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); // use 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 indexe, ne koordinat
82         else
83             height = sl->remove(s, e);
84         px = verts[i].x;
85     }
86
87     return area;
88 }

```

## 5.4 Najbližji par točk v ravnini

**Vhod:** Seznam  $n \geq 2$  točk v ravnini.

**Izhod:** Kvadrat razdalje med najbližjima točkama. Z lahko 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 }

```