

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
- uporabi `vector.assign` ne `vector.resize` med primeri
- uporabi `cin.sync_with_stdio(false);`
`cin.tie(nullptr);` in nikoli `endl` za hitrejši IO
- uporabi `numeric_limits<tip>::max()`
ali `infinity()`, `min()` za robne vrednosti
- uporabi `g++ -std=c++11 -Wall -pedantic -Wextra`
- v template dodaj `algorithm`, `array`, `complex`, `cmath`, `functional`, `iostream`, `io manip`, `limits`, `map`, `queue`, `set`, `stack`, `string`, `tuple`, `utility`, `vector`, namespace `std` in `cin` zadeve.
- Za izpis na fiksno število decimalk uporabi
`cout << fixed << setprecision(6);`

verzija: 11. oktober 2016

Kazalo

1	Grafi	4
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	Najširša pot med vsemi pari vozlišč	11
1.9	Največji pretok in najmanjši prerez	12
1.9.1	Ford-Fulkerson	12
1.9.2	Edmonds-Karp	13
1.10	Največje prirejanje in najmanjše pokritje	14
1.10.1	Največje prirejanje v neuteženih dvodelnih grafih	14
2	Podatkovne strukture	15
2.1	Statično binarno iskalno drevo	15
2.2	Statično drevo segmentov	15
2.3	Drevo segmentov	16
2.4	Avl drevo	18
2.5	Fenwickovo drevo	20
2.6	Fenwickovo drevo (n -dim)	20
2.7	Trie	21
3	Algoritmi	22
3.1	Najdaljše skupno podzaporedje	22
3.2	Najdaljše naraščajoče podzaporedje	23
3.3	Najdaljši strnjen palindrom	24
3.4	Podseznam z največjo vsoto	25
3.5	Leksikografsko minimalna rotacija	26
3.6	BigInt in Karatsuba	26
3.7	2-SAT	28
3.8	Knuth-Morris-Pratt	29
3.9	z -funkcija	30
3.10	Minimalna perioda niza	31
3.11	Minimalni element v podseznamu	31
4	Numerika	32
4.1	Gaussova eliminacija	32

5	Teorija števil	33
5.1	Evklidov algoritem	33
5.2	Razširjen Evklidov algoritem	33
5.3	Kitajski izrek o ostankih	33
5.4	Hitro potenciranje	34
5.5	Številski sestavi	34
5.6	Eulerjeva funkcija ϕ	35
5.7	Eratostenovo rešeto	35
5.8	Število deliteljev	36
5.9	Binomski koeficienti	36
5.10	Binomski koeficienti po modulu	37
6	Geometrija	38
6.1	Osnove	38
6.2	Konveksna ovojnica	41
6.3	Ploščina unije pravokotnikov	42
6.4	Najbližji par točk v ravnini	43
7	Matematika	44

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

```

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

```
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

```
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

```

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


```

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/problems/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 Najširša pot med med vsemi pari vozlišč

Vhod: Utežen usmerjen graf G z n vozlišči.

Izhod: $n \times n$ matrika, ki za vsak par vozlišč pove, koliko je širok najožji del najširše poti med tema vozliščema; ie. za vsak par vozlišč pove, kako širok je bottleneck med njima.

Časovna zahtevnost: $O(EV)$

Prostorska zahtevnost: $O(V + E)$

Testiranje na terenu: http://putka.upm.si/tasks/2016/2016_1kolo/bottleneck

```

3     vector<vector<int>> all_pairs_widest_path(const vector<vector<pair<int, int>>>& graf_) {
4         auto graf = graf_;
5         int n = graf.size();
6         // Prepisi vse povezave v en velik container
7         vector<tuple<int, int, int>> edges;
8         for (int i = 0; i < n; i++) {
9             sort(graf[i].begin(), graf[i].end(), [](const pair<int, int>& p, const pair<int, int>& q) {
10                 if (p.second == q.second) return p.first > q.first;
11                 return p.second > q.second; });
12             for (const auto& p : graf[i]) {
13                 edges.push_back(make_tuple(p.second, p.first, i)); // w, v, u (reversed for sorting)
14             }
15         }
16         sort(edges.begin(), edges.end(), greater<tuple<int, int, int>>());
17
18         vector<vector<int>> result(n, vector<int>(n, 0)); // note: morda ull
19         vector<bool> visited;
20         queue<int> q;
21         vector<int> vertex_degree; // koliko edgev, ki vodijo iz mesta si ze dodal
22         int u, v, w;
23
24         // start in each vertex
25         for (int i = 0; i < n; ++i) {

```

```

26     visited.assign(n, false); // Reset these
27     vertex_degree.assign(n, 0);
28     visited[i] = true; // I am visited
29
30     // add edges one by one from bigger to smaller weight
31     for (auto e : edges) {
32         tie(w, v, u) = e;
33         vertex_degree[u]++; // Oznamo, da smo dodali dodatno cesto
34
35         if (visited[u] && !visited[v]) { // Če smo iz že doseženega mesta prisli do novega
36             visited[v] = true; // Dodaj se med visited in dodaj svojo sirino v result
37             result[i][v] = w;
38             if (vertex_degree[v] > 0) { // Če je treba: flood fill
39                 q.push(v);
40                 while (!q.empty()) {
41                     int cur = q.front();
42                     q.pop();
43                     for (int k = 0; k < vertex_degree[cur]; ++k) {
44                         auto sosed = graf[cur][k].first; // samo prvih toliko je obiskanih
45                         if (!visited[sosed]) { // zato smo jih prej sortirali
46                             visited[sosed] = true;
47                             result[i][sosed] = w;
48                             q.push(sosed);
49                     }
50                 }
51             }
52         }
53     }
54 }
55 }
56 return result;
57 }

```

1.9 Največji pretok in najmanjši prerez

Če imamo omrežje s dopustnimi kapacitetami na povezavah, potem max flow pove, koliko največ lahko teže skozi od source s do sinka t . Min cut je najmanjša vsota tež povezav, ki jih moramo odstraniti, da se s in t popolnoma ločita. Ti dve vrednosti sta enaki.

1.9.1 Ford-Fulkerson

Vhod: Matrika kapacitet dimenzij $n \times n$ (tj. matrika sosednosti grafa z n vozlišči, kjer so vrednosti dopustni pretoki povezav). Vse kapacitete morajo biti nenegativne. Hitrejša (in daljša) verzija tega algoritma je Edmonds-Karpov algoritem.

Izhod: Vrne maksimalen pretok od s do t , ki je po vrednosti enak minimalnemu prerezu. Konstruira tudi matriko pretoka. Če pot od s do t sploh ne obstaja vrne 0.

Časovna zahtevnost: $O(Ef)$

Prostorska zahtevnost: $O(V^2)$

Testiranje na terenu: UPM 2016, 3. kolo deske

```

3  int dfs(int cur, int t, vector<bool>& visited, vector<vector<int>>& flow, int f = 20000) {
4      visited[cur] = true;
5      if (cur == t) return f;
6      int n = flow.size();
7      for (int y = 0; y < n; ++y) {
8          if (flow[cur][y] > 0 && !visited[y]) {
9              int df = dfs(y, t, visited, flow, min(f, flow[cur][y]));
10             if (df > 0) {
11                 flow[cur][y] -= df;
12                 flow[y][cur] += df;
13                 return df;
14             }
15         }
16     }

```

```

17     return 0;
18 }
19
20 int ford_fulkerson_maximal_flow(const vector<vector<int>>& capacity, int s, int t) {
21     vector<vector<int>> flow = capacity;
22     int max_flow = 0, df;
23     do {
24         vector<bool> visited(flow.size(), false);
25         df = dfs(s, t, visited, flow);
26         max_flow += df;
27     } while (df != 0);
28     return max_flow;
29 }

```

1.9.2 Edmonds-Karp

Vhod: Matrika kapacitet dimenzij $n \times n$ (tj. matrika sosednosti grafa z n vozlišči, kjer so vrednosti dopustni pretoki povezav). Graf je lahko (in ponavadi tudi je) usmerjen. Vse kapacitete morajo biti nenegativne.

Izhod: Vrne maksimalen pretok od s do t . Konstruira tudi matriko pretoka. Če pot od s do t sploh ne obstaja vrne 0.

Časovna zahtevnost: $O(VE^2)$

Prostorska zahtevnost: $O(V^2)$

Testiranje na terenu: UVa 820, UPM 2016, 3. kolo deske

```

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 }

```

1.10 Največje prirejanje in najmanjše pokritje

1.10.1 Največje prirejanje v neuteženih dvodelnih grafih

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 , pri 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     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

```
6 struct BST {
7     int depth, n; // 1 << depth mora biti >= številu možnih elementov
8     vector<int> tree; // number of elements in the tree = tree[0]
9
10    BST(int d) : depth(d), n(1 << d), tree(2*n, 0) {}
11
12    void insert(int idx, int val = 1) {
13        int i = n - 1 + idx;
14        while (i > 0) {
15            tree[i] += val;
16            i--;
17            i >>= 1;
18        }
19        tree[0] += val;
20    }
21
22    int get_kth(int k) {
23        int i = 0;
24        while (i < n - 1) {
25            int lc = tree[2*i+1];
26            if (lc <= k) {
27                i = 2*i + 2;
28                k -= lc;
29            } else {
30                i = 2*i + 1;
31            }
32        }
33        return i - n + 1;
34    }
35
36    bool remove(int idx) {
37        if (tree[n-1 + idx] <= 0) return false;
38        insert(idx, -1);
39        return true;
40    }
41 };
42 ostream& operator<<(ostream& os, const BST& t) {
43     for (int i = 0; i <= t.depth; ++i) {
44         os << string((1 << (t.depth - i)) - 1, ' ');
45         for (int j = (1 << i) - 1; j < (1 << (i+1)) - 1; ++j) {
46             os << t.tree[j] << string((1 << (t.depth - i + 1)) - 1, ' ');
47         }
48         os << '\n';
49     }
50     return os;
51 }
52 #endif // IMPLEMENTACIJA_PS_STATIC_BINARY_SEARCH_TREE_H
```

2.2 Statično drevo segmentov

Struktura za delat poizvedbe o neki statistiki na podseznamih nekega seznama. Spodaj je implementacija za minimum, toda `min` in `inf` se lahko zamenjata s poljubno asociativno operacijo. Implementacija je iterativna in zato hitrejša kot tista spodaj.

Operacije: • zgradi: naredi strukturo iz seznama

- spremeni: nastavi element na indeksu i na vrednost v
- išči: vrne statistiko podseznama $[a, b]$. Pozor, polodprto!

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

Prostorska zahtevnost: $O(n)$

Testiranje na terenu: http://putka.upm.si/tasks/2016/2016_3kolo/maxer

```

6  const int inf = 1e9;
7  struct RangeQuery { // Change inf and min below to your operation and initial value
8      int n; // size of the tree
9      vector<int> tree; // 1-based
10     RangeQuery(const vector<int>& a) : // Constructs a queryable structure over you list
11         n(1 << static_cast<int>(ceil(log2(a.size())))),
12         tree(2*n, inf) {
13         for (size_t i = 0; i < a.size(); ++i) tree[n+i] = a[i];
14         for (int i = n - 1; i; --i) tree[i] = min(tree[2*i], tree[2*i+1]);
15     }
16     void set(int pos, int x) {
17         for (int i = pos + n; i; i >>= 1) tree[i] = min(tree[i], x);
18     }
19     int get(int s, int e) const { // [s, e] i.e. s <= i < e
20         int res = inf;
21         for (s += n, e += n; s < e; s >>= 1, e >>= 1) {
22             if (s & 1) res = min(res, tree[s++]);
23             if (e & 1) res = min(res, tree[--e]);
24         }
25         return res;
26     }
27 };
28 ostream& operator<<(ostream& os, const RangeQuery& rq) { // fancy print
29     int depth = static_cast<int>(log2(rq.n));
30     for (int i = 0; i <= depth; ++i) {
31         os << string((1 << (depth - i)) - 1, ' ');
32         for (int j = (1 << i); j < (1 << (i+1)); ++j) {
33             os << rq.tree[j] << string((1 << (depth - i + 1)) - 1, ' ');
34         }
35         os << '\n';
36     }
37     return os;
38 }
39 #endif // IMPLEMENTACIJA_PS_STATIC_SEGMENT_TREE_H

```

2.3 Drevo segmentov

Operacije: Segment tree deljen po fiksni točki 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 <= 1 && r <= t) {
14         here_count += inc;
15         int lb = max(1, 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(1, 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 - 1 + 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 <= 1 && r <= t) return total_count;
44     int sum = 0, lb = max(1, 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_

```

2.4 Avl drevo

Operacije: Klasično uravnoteženo binarno iskalno drevo.

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

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

Prostorska zahtevnost: $O(n)$

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

Opombe: To je lepa implementacija. V praksi ne rabimo vsega public interface-a je dovolj samo imeti neke 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     AVLNode<T>* insert(const T& val, AVLNode<T>* node) {
84         if (node == nullptr) return node = new AVLNode<T>(val);
85         node->size++;
86         AVLNode<T>* return_node = node;
87         if (val < node->value) return_node = insert(val, node->left);
88         else if (node->value == val) node->count++;
89         else if (node->value < val) return_node = insert(val, node->right);
90         balance(node);
91         return return_node;
92     }
93     bool erase(const T& val, AVLNode<T>* node) {
94         if (node == nullptr) return false;
95         if (val < node->value) {
96             if (erase(val, node->left)) {
97                 node->size--;
98                 return true;
99             }
100         } else if (node->value < val) {
101             if (erase(val, node->right)) {
102                 node->size--;
103                 return true;
104             }
105         } else if (node->value == val && node->count > 0) {
106             node->count--;
107             node->size--;
108             return true;
109         }
110         return false;
111     }
112     const AVLNode<T>* get_nth(const AVLNode<T>* const& node, size_t n) const {
113         size_t left_size = size(node->left);
114         if (n < left_size) return get_nth(node->left, n);
115         else if (n < left_size + node->count) return node;
116         else if (n < node->size) return get_nth(node->right, n - left_size - node->count);
117         else return nullptr;
118     }
119     const AVLNode<T>* find(const AVLNode<T>* const& node, const T& value) const {

```

```

120         if (node == nullptr) return nullptr;
121         if (value < node->value) return find(node->left, value);
122         else if (value == node->value) return node;
123         else return find(node->right, value);
124     }
125
126     AvlNode<T>* root;
127 };
128
129 template<typename T>
130 ostream& operator<<(ostream& os, const AvlTree<T>& tree) {
131     if (tree.root == nullptr) os << "Tree empty";
132     else tree.root->print(os);
133     return os;
134 }
135 #endif // IMPLEMENTACIJA_PS_AVL_TREE_H_

```

2.5 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>

```

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 }

```

2.6 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 spremeni n za 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

```

3  typedef vector<vector<vector<int>>> vvv;
4
5  int sum(int x, int y, int z, const vvv& 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, vvv& 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 vvv& 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 }

```

2.7 Trie

Operacije: Prefix tree, hranimo besede, črko po črko na nivoju, črke so iz neke končne abecede Σ , pri implementaciji $\Sigma = \{a, \dots, z\}$.

- vstavi besedo
- največji skupen prefix z dano besedo
- največji skupen prefix med besedami v drevesu (vrne ena preveč)

Časovna zahtevnost: $O(\ell)$ za add in common_prefix, ki tju uporabimo na besedi dolžine ℓ ter $O(|T|)$ za najdaljši prefix med vsemi besedami, kjer je $|T|$ število vozlišč v drevesu.

Prostorska zahtevnost: $O(|T|) = O(n|\Sigma|)$, kjer je n število besed, v praksi manj, ker se prekrivajo.

Testiranje na terenu: <http://www.spoj.com/problems/PRHYME/> TODO

```

6  struct Node {
7      int words, total; // koliko besed se konča tukaj, koliko besed je v poddrevesu
8      Node* nodes[26];
9      Node() : words(0), total(0) {
10         for (int i = 0; i < 26; ++i) nodes[i] = nullptr;
11     }

```

```

12 void add(const string& word, size_t idx = 0) {
13     if (idx == word.size()) { words++; return; }
14     int p = word[idx] - 'a';
15     if (nodes[p] == nullptr) nodes[p] = new Node();
16     total++;
17     nodes[p] -> add(word, idx+1);
18 }
19 void print(int n, string& prefix) {
20     if (n == 0) return;
21     for (int j = 0; j < words; ++j)
22         printf("%s\n", prefix.c_str());
23
24     for (int i = 0; i < 26; ++i) {
25         if (nodes[i] != nullptr) {
26             prefix.push_back('a'+i);
27             nodes[i] -> print(n-1, prefix);
28             prefix.pop_back();
29         }
30     }
31 }
32 int longest_common_prefix_length() { // najdaljši skupen prefix med besedami v drevesu + 1
33     int childc = 0;
34     int maxl = 0;
35     for (int i = 0; i < 26; ++i) {
36         if (nodes[i] != nullptr) {
37             childc++;
38             maxl = max(maxl, nodes[i] -> longest_common_prefix_length());
39         }
40     }
41     if (maxl > 0) return maxl+1; // ce je match v poddrevesu, imamo match+1
42     return childc > 1 || (childc == 1 && words > 0) || (words > 1);
43 } // sicer imamo match ce sta dve isti ali imamo otroka in mi eno besedo ali dva otroka
44 int common_prefix(const string& s, size_t x = 0) { // koliko crk imata skupnih
45     if (nodes[s[x] - 'a'] == nullptr || x == s.size()) return x;
46     return nodes[s[x] - 'a'] -> common_prefix(s, x+1);
47 }
48 int size() { return total+words; }
49 ~Node() {
50     for (int i = 0; i < 26; ++i)
51         delete nodes[i];
52 }
53 };
54
55 #endif // IMPLEMENTACIJA_PS_TRIE_H_

```

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 tudi 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] = dolzina 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         if (L[i] > maxLPSLength) { // Track maxLPSLength // Length by ONE If even
31             maxLPSLength = L[i]; // position, we just increment
32             maxLPSCenterPosition = i; // LPS by ONE without any
33         } // 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 - maxLPSELength)/2;
41 end = start + maxLPSELength;
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 }

```

```

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 $+$, \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 in 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 brute force 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 zacetek 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_

```

3.7 2-SAT

Vhod: Formula $\varphi(x_1, \dots, x_n)$ v 2-CNF obliki, torej

$$\varphi = S_1 \wedge \dots \wedge S_n, \quad S_i = L_{i1} \vee L_{i2}, \quad L_{ij} \in \{x_1, \dots, x_n, \neg x_1, \dots, \neg x_n\}.$$

Za naše namene je predstavljena kot seznam parov števil od ± 1 do $\pm n$, kjer pozitivna število i pomeni literal x_i , negativno število $-i$ pa literal $\neg x_i$. Na primer,

$$[(-2, 3), (4, 5), (-3, -1)]$$

predstavlja formulo

$$(\neg x_2 \vee x_3) \wedge (x_4 \vee x_5) \wedge (\neg x_3 \vee \neg x_1).$$

Izhod: Nabor n vrednosti za x_i , pri katerih je formula resnična. Če tak nabor ne obstaja, vrne $(-1, \dots, -1)$.

Časovna zahtevnost: $O(V + E)$

Prostorska zahtevnost: $O(V + E)$

Testiranje na terenu: Uva 11294

```
3  int var2node(int var, int n) { // sprem. od ±1 do ±n spremeni v vozlišče od 0 do 2n-1
4      return (var > 0) ? var - 1 : -var - 1 + n;
5  }
6
7  namespace {
8      int dfscount;
9      vector<int> postorder;
10     vector<int> comp; // comp[i] pove v kateri komponenti je i. Komponente so top. urejene.
11     vector<bool> visited;
12 }
13
14 void get_postorder(const vector<vector<int>>& G, int v) { // izračuna čase odhoda iz vozlišč
15     visited[v] = true;
16     for (int u : G[v])
17         if (!visited[u])
18             get_postorder(G, u);
19     postorder[dfscount++] = v;
20 }
21
22 void mark_comp(const vector<vector<int>>& G, int v, int scccount) { // najde povezane komponente
23     comp[v] = scccount;
24     for (int u : G[v])
25         if (comp[u] == -1)
26             mark_comp(G, u, scccount);
27 }
28
29 vector<int> solve_2sat(const vector<pair<int, int>>& formula, int n) {
30     vector<vector<int>> G(2*n), GR(2*n);
31     int x, y;
32     for (const auto& term : formula) { // Imamo stavke x v y.
33         tie(x, y) = term; // Naredimo dva sklepa: -x => y, -y => x
34         int tx = var2node(x, n), ty = var2node(y, n);
35         int nx = var2node(-x, n), ny = var2node(-y, n);
36         G[nx].push_back(ty); G[ny].push_back(tx); // naredimo graf implikacij
37         GR[ty].push_back(nx); GR[tx].push_back(ny); // in obraten graf
38     }
39
40     dfscount = 2*n-1;
41     postorder.resize(2*n);
42     visited.assign(2*n, false);
43     for (int i = 0; i < 2*n; ++i)
44         if (!visited[i])
45             get_postorder(G, i);
46
47     comp.assign(2*n, -1);
48     int scccount = 0;
49     for (int v : postorder)
50         if (comp[v] == -1)
51             mark_comp(GR, v, scccount++);
52
53     for (int i = 0; i < n; ++i) {
54         if (comp[i] == comp[n+i])
55             return vector<int>(n, -1); // ali false
56     } // ce ne rabis resitve lahko das tukaj return true;
57
58     vector<int> solution(n);
59     for (int i = 0; i < n; ++i)
60         solution[i] = comp[i] > comp[n+i];
61     return solution;
62 }
63
64 bool evaluate(const vector<pair<int, int>>& formula, const vector<int>& values) {
65     int x, y; // izračuna vrednost formule pri danem naboru vrednosti spremenljivk
66     for (const auto& p : formula) { // pričakujemo da so v formuli številke od \pm 1 do \pm n
67         tie(x, y) = p; // in da imamo n vrednosti, ki so 0 ali 1
68         bool vx = values[abs(x)-1] ^ (x < 0);
69         bool vy = values[abs(y)-1] ^ (y < 0);
70         if (!(vx || vy)) return false;
71     }
72     return true;
73 }
```

3.8 Knuth-Morris-Pratt

Vhod: Niz znakov s dolžine n in niz znakov p dolžine m . Posebej lahko izračunamo tudi failure function ali podamo indeks, da išče po nizu samo od neke naprej.

Izhod: Najmanjši indeks $0 \leq i < n$, tako da se v s na mestih $s[i:i+m]$ nahaja p . Če tak indeks ne obstaja vrne -1 . Program torej najde prvo pojavitev p v s . Hkrati izračuna tudi *failure_function* ff , ki pove nekaj o samopodobnosti niza. Vrednost $ff[i-1]$ pove indeks naslednje črke, ki jo moramo preveriti, če vemo, da smo na i -tem znaku ravno failali match podniza. Drugače, to je dolžina največjega pravega podniza $p[:i+1]$, ki je hkrati prefix in suffix za niz $p[:i+1]$. Primer:

p	A	B	C	D	A	B	D
i	0	1	2	3	4	5	6
ff	0	0	0	0	1	2	0

Časovna zahtevnost: $O(n+m)$

Prostorska zahtevnost: $O(m)$

Testiranje na terenu: <http://www.spoj.com/problems/NHAY/>

```

3  vector<int> compute_failure_function(const string& p) {
4      int m = p.size();
5      vector<int> ff(m, 0);
6      for (int k = 0, i = 1; i < m; ++i) {
7          while (k > 0 && p[i] != p[k]) k = ff[k-1];
8          if (p[i] == p[k]) k++;
9          ff[i] = k;
10     }
11     return ff;
12 }
13
14 int knuth_morris_pratt(const string& s, const string& p, const vector<int>& ff, int start) {
15     int k = 0, n = s.size(), m = p.size();
16     for (int i = start; i < n; i++) {
17         while (k > 0 && p[k] != s[i]) k = ff[k-1];
18         if (s[i] == p[k]) k++;
19         if (k == m) return i - k + 1;
20     }
21     return -1;
22 }
23
24 int knuth_morris_pratt(const string& s, const string& p) {
25     vector<int> ff = compute_failure_function(p);
26     return knuth_morris_pratt(s, p, ff, 0);
27 }
28
29 vector<int> find_all_occurrences(const string& s, const string& p) {
30     vector<int> ff = compute_failure_function(p), result;
31     int i = -1;
32     while ((i = knuth_morris_pratt(s, p, ff, i + 1)) != -1) {
33         result.push_back(i); // or do something else
34     }
35     return result;
36 }
37
38 vector<int> find_non_overlapping_occurrences(const string& s, const string& p) {
39     vector<int> ff = compute_failure_function(p), result;
40     int i = -p.size();
41     while ((i = knuth_morris_pratt(s, p, ff, i + p.size())) != -1) {
42         result.push_back(i); // or do something else
43     }
44     return result;
45 }
46
47 int minimal_period(const string& s) {
48     int n = s.size();
49     vector<int> ff = compute_failure_function(s);
50     int candidate = n - ff.back();
51     if (n % candidate == 0) return candidate;
52     return n;
53 }

```

3.9 z-funkcija

Vhod: Niz znakov s dolžine n .

Izhod: z -funkcija niza. Vrednost $z[i]$ pove največji skupni prefix med s in $s[i:]$.

Primer: za $s = \text{"aaaaa"}$ je $z = [0, 4, 3, 2, 1]$, za $s = \text{"aaabaab"}$ je $z = [0, 2, 1, 0, 2, 1, 0]$, za $s = \text{"abacaba"}$ je $z = [0, 0, 1, 0, 3, 0, 1]$.

Časovna zahtevnost: $O(n)$

Prostorska zahtevnost: $O(n)$

Testiranje na terenu: TODO

```
3  vector<int> z_function(const string& s) {
4      int n = s.length();
5      vector<int> z(n);
6      for (int i = 1, l = 0, r = 0; i < n; ++i) {
7          if (i <= r) z[i] = min(r - i + 1, z[i - l]);
8          while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
9          if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
10     }
11     return z;
12 }
13 int match(const string& s, const string& p) {
14     string t = p + '#' + s; // # does not appear in s or p
15     vector<int> z = z_function(t);
16     int n = s.size(), m = p.size();
17     for (int i = 0; i < n; ++i)
18         if (z[i+m+1] == m)
19             return i;
20     return -1;
21 }
22 int minimal_period_zf(const string& s) {
23     int n = s.size();
24     vector<int> z = z_function(s);
25     for (int i = 1; i < n; ++i)
26         if (n % i == 0 && i + z[i] == n) // ce ni pogoja n % i == 0 najdemo tudi necele periode
27             return i; // če najdemo vse take i, najdemo vse periode
28     return n;
29 }
```

3.10 Minimalna perioda niza

Vhod: Niz znakov s dolžine n .

Izhod: Dolžina minimalne periode s , tj. tak k , da je $(s[:k])^{\frac{n}{k}} = s$.

Primer: minimalna perioda $s = \text{"abcabcabc"}$ je "abc" , dolžine 3.

Časovna zahtevnost: $O(n)$

Prostorska zahtevnost: $O(n)$

Testiranje na terenu: TODO

Implementacija: Glej z -funkcija, str. 30 ali Knuth-Morris-Pratt, str. 29.

3.11 Minimalni element v podseznamu

Po angleško RMQ ali Range Minimum Query. Želimo odgovarjati na poizvedbe v seznam dolžine n oblike: "Koliko je minimum med a in b ." Če se elementi seznama spreminjajo, potem je najboljša uporaba statičnega drevesa segmentov 2.2. Če pa imamo dan seznam vnaprej, potem si lahko zgradimo strukturo, kjer za vsak indeks i vemo odgovor na vprašanje "Koliko je minimum na intervalu $[i, i + 2^j]$ za vse j , da je interval še v seznamu. Tako lahko query $[a, b]$ odgovorimo v konstantnem času tako, da najdemo največji r , da je $a + 2^r \leq b$ in primerjamo minimuma na $[a, a + 2^r]$, $[b - 2^r, b]$, ki sta oba poračunana vnaprej.

4 Numerika

4.1 Gaussova eliminacija

Vhod: Matrika $A' = [A \mid b]$, ki predstavlja $m \times n$ sistem $Ax = b$.

Izhod: Vektor x (dolga n), ki reši sistem. Poleg tega vrne tudi -1 , če rešitev ne obstaja, 0 , če je enolična, sicer pa dimenzionalnost prostora rešitev (tj. število splošnih konstant). Enačbo rešimo z LU razcepom z delnim pivotiranjem. Druga funkcija izračuna determinanto matrike, prav tako z uporabo delnega pivotiranja. Obe se da uporabiti z ulomki namesto z decimalnimi števili.

Časovna zahtevnost: $O(\min\{n, m\}nm)$

Prostorska zahtevnost: $O(nm)$

Testiranje na terenu: TODO

```
3 namespace {
4     const double eps = 1e-9;
5 }
6
7 int gauss(vector<vector<double>> A, vector<double>& ans) {
8     int n = A.size(), m = A[0].size() - 1;
9
10    vector<int> where(m, -1);
11    for (int col = 0, row = 0; col < m && row < n; ++col) {
12        int sel = row;
13        for (int i = row; i < n; ++i)
14            if (abs(A[i][col]) > abs(A[sel][col])) sel = i;
15        if (abs(A[sel][col]) < eps) continue;
16        for (int i = col; i <= m; ++i) swap(A[sel][i], A[row][i]);
17        where[col] = row;
18
19        for (int i = 0; i < n; ++i)
20            if (i != row) {
21                double c = A[i][col] / A[row][col];
22                for (int j = col; j <= m; ++j) A[i][j] -= A[row][j] * c;
23            }
24        ++row;
25    } // tukaj je v A spravljen LU razcep PA = LU.
26
27    ans.assign(m, 0);
28    for (int i = 0; i < m; ++i)
29        if (where[i] != -1)
30            ans[i] = A[where[i]][m] / A[where[i]][i];
31
32    // Preverjamo rank in to, da rešitev res reši sistem
33    for (int i = 0; i < n; ++i) {
34        double sum = 0;
35        for (int j = 0; j < m; ++j) sum += ans[j] * A[i][j];
36        if (abs(sum - A[i][m]) > eps) return -1;
37    }
38
39    return count(where.begin(), where.end(), -1);
40 }
41
42 double det(vector<vector<double>> a) {
43     double det = 1;
44     int n = a.size();
45     for (int i = 0; i < n; ++i) {
46         int k = i;
47         for (int j = i + 1; j < n; ++j)
48             if (abs(a[j][i]) > abs(a[k][i]))
49                 k = j;
50         if (abs(a[k][i]) < eps) {
51             det = 0;
52             break;
53         }
54         swap(a[i], a[k]);
55         if (i != k) det = -det;
56         det *= a[i][i];
57         for (int j = i + 1; j < n; ++j) {
58             a[i][j] /= a[i][i];
59         }
60         for (int j = 0; j < n; ++j) {
61             if (j != i && abs(a[j][i]) > eps) {
```



```

62         for (int k = i + 1; k < n; ++k) {
63             a[j][k] -= a[i][k] * a[j][i];
64         }
65     }
66 }
67 }
68 return det;
69 }

```

5 Teorija števil

5.1 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 }

```

5.2 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 }

```

5.3 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. 33)

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  // sprejme seznam [(a_i, m_i)], za enačbe x == a_i (mod m_i)
10 int chinese_remainder_theorem(const vector<pair<int, int>>& cong) {
11     int M = 1;
12     for (size_t i = 0; i < cong.size(); ++i) {
13         M *= cong[i].second;
14     }
15     int x = 0, a, m;
16     for (const auto& p : cong) {
17         tie(a, m) = p;
18         x += a * M / m * mul_inverse(M/m, m);
19         x %= M;
20     }
21     return (x + M) % M;
22 }
```

5.4 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

```
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 }
```

5.5 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

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 }
```

5.6 Eulerjeva funkcija ϕ

Vhod: Število $n \in \mathbb{N}$.

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

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

Časovna zahtevnost: $O(\sqrt{n})$

Prostorska zahtevnost: $O(1)$

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

```
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 }
```

5.7 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 }
```

5.8 Število deliteljev

Vhod: Število $n \in \mathbb{N}$.

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

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

Časovna zahtevnost: $O(\sqrt{n})$

Prostorska zahtevnost: $O(1)$

Testiranje na terenu: <https://projecteuler.net/problem=12>

```

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) { // i je prafaktor n, s potenco p
9             n /= i;
10            p++;
11        }
12        tau *= p + 1;
13        i++;
14    }
15    if (n > 1) tau *= 2;
16    return tau;
17 }
```

5.9 Binomski koeficienti

Vhod: Števili $n, k \in \mathbb{Z}$.

Izhod: Binomski koeficient $\binom{n}{k} = \begin{cases} \frac{n!}{k!(n-k)!} & n, k \geq 0 \\ 0 & \text{sicer} \end{cases}$

Za velike vrednosti lahko izračunamo aproksimacijo s pomočjo logaritma gama funkcije. Če rabiš isto vrednost $\binom{n}{k}$ več kot enkrat, se morda splača shraniti cel Pascalov trikotnik.

Časovna zahtevnost: $O(\min\{k, n - k\})$ za enega, $O(n^2)$ za Pascalov trikotnik.

Prostorska zahtevnost: $O(1)$ za enega, $O(n^2)$ za Pascalov trikotnik

Testiranje na terenu: http://putka.upm.si/tasks/2015/2015_3kolo/minsko_polje

```
3  int binomial(int n, int k) {
4      if (k < 0 || k > n) return 0;
5      if (k == 0 || k == n) return 1;
6      k = min(k, n - k);
7      int r = 1;
8      for (int i = 0; i < k; ++i) {
9          r *= n - i;
10         r /= i + 1; // deljenje se vedno izide
11     }
12     return r;
13 }
14
15 double binomial_approx(double n, double k) {
16     return exp(lgamma(n+1) - lgamma(k+1) - lgamma(n-k+1));
17 }
18
19 vector<vector<int>> pascal_triangle(int n) {
20     vector<vector<int>> ret(n, vector<int>(n, 0));
21     ret[0][0] = 1;
22     for (int i = 1; i < n; ++i) {
23         ret[i][0] = 1;
24         for (int j = 1; j <= i; ++j) {
25             ret[i][j] = ret[i-1][j] + ret[i-1][j-1];
26         }
27     }
28     return ret;
29 }
```

5.10 Binomski koeficienti po modulu

Vhod: Števili $n, k \in \mathbb{Z}$ in $p \in \mathbb{P}$. Lahko posplošimo na poljubno število, ki v praštevilskem razcepu nima potenc praštevil.

Izhod: Ostanek pri deljenju binomskega koeficienta $\binom{n}{k}$ s p . Če je p praštevilo, lahko uporabiš `binomial_modp`, sicer pa `binomial_mod`. Pri deljenju s praštevilom uporabimo Lucasov izrek:

$$\binom{n}{k} \equiv \binom{n_r}{k_r} \binom{n_{r-1}}{k_{r-1}} \dots \binom{n_1}{k_1} \binom{n_0}{k_0},$$

kjer sta $n = n_r \dots n_0$ in $k = k_r \dots k_0$ zapisa števil n in k v p -jiškem sistemu.

Sicer število razcepimo na tuja si števila (prafaktorje) in uporabimo skupaj za vsako praštevilo posebej zgornji postopek, nato pa združimo s kitajskim izrekom o ostankih.

Časovna zahtevnost: $O(p \cdot (\log_p(n) + \log_p(k)))$ za praštevilo p .

Prostorska zahtevnost: $O(1)$

Potrebuje: Kitajski izrek o ostankih (str. 33)

Testiranje na terenu: http://putka.upm.si/tasks/2015/2015_3kolo/minsko_polje

```
3  int binomial_modp(int n, int k, int p) {
4      if (k < 0 || k > n) return 0;
5      if (k == 0 || k == n) return 1;
6      int r = 1;
7      while (n > 0) {
8          int ni = n % p, ki = k % p;
9          r *= binomial(ni, ki); // če imaš zračunano vnaprej, uporabi tisto
10         r %= p; // sicer pa delaj modulo tudi znotraj binomial
11         n /= p;
12         k /= p;
13     }
14     return r;
15 }
```

```

16
17 // funkcija je tu bolj za demonstracijo, v praksi racunaj vse rezultate po modulih
18 int binomial_mod(int n, int k, int m) { // prafaktorjev in sele na koncu rekunstruiraj
19     if (m == 1) return 0;
20     if (k < 0 || k > n) return 0;
21     if (k == 0 || k == n) return 1;
22     vector<pair<int, int>> mods;
23     int i = 2;
24     while (i * i <= m) {
25         int p = 0;
26         while (m % i == 0) { // i je prafaktor n, s potenco p
27             m /= i;
28             p++;
29         }
30         assert(p <= 1 && "Invalid number, numbers containing prime powers not supported.");
31         if (p == 1) mods.emplace_back(binomial_modp(n, k, i), i);
32
33         i++;
34     } // kar ostane mora biti praštevilko ali 1
35     if (m != 1) mods.emplace_back(binomial_modp(n, k, m), m);
36
37     return chinese_remainder_theorem(mods);
38 }

```

6 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.

6.1 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(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 presečišče morajo biti o1, o2, o3, o4 != 0
111     // vemo da presečišče obstaja, tudi ce veljata samo prva dva pogoja
112     if (o1 != 0 && o3 != 0 && o1 != 0 && o2 != 0 && o3 != 0 && o4 != 0) {
113         return line_line_intersection(L(p1, p2), L(q1, q2));
114     }
115
116     // EQ = se dotika samo z ogliscem ali sta vzporedni
117     if (o1 == 0 && point_in_rect(q1, p1, p2)) return {EQ, q1}; // q1 lezi na p
118     if (o2 == 0 && point_in_rect(q2, p1, p2)) return {EQ, q2}; // q2 lezi na p
119     if (o3 == 0 && point_in_rect(p1, q1, q2)) return {EQ, p1}; // p1 lezi na q
120     if (o4 == 0 && point_in_rect(p2, q1, q2)) return {EQ, p2}; // p2 lezi na q

```



```

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 oglisce, 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 }

```

6.2 Konveksna ovojnica

Vhod: Seznam n točk.

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

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

Prostorska zahtevnost: $O(n)$

Potrebuje: Vektorski produkt, str. 38.

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) { // točke, 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 }

```

6.3 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; // y-coordinates of rect sides (can be double)
15
16 struct Node { // segment tree, s in e sta zacetek in konec intervala, torej en node pokrije [s, e]
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 dani tako, ce v nalogi niso, zamenjaj x1 <-> x2
57     for (int i = 0; i < n; ++i) {
58         tie(levo_spodaj, desno_zgoraj) = rects[i];

```

```

59     int a = levo_spodaj.real(); // +-----+ (c, d)
60     int c = desno_zgoraj.real(); // / /
61     int b = levo_spodaj.imag(); // / /
62     int d = desno_zgoraj.imag(); // (a, b) +-----+
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() - 1); // sweepline segment tree, po celem seznamu
74
75 int area = 0, height = 0; // area = total area. height = trenutno pokrita višina
76 int px = -(1 << 30); // value smaller than smallest x coordinate
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) height = sl->add(s, e); // segment tree sprejme indekse, ne koordinat
83     else height = sl->remove(s, e);
84     px = verts[i].x;
85 }
86
87 return area;
88 }

```

6.4 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 }

```

7 Matematika

Vrste:

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

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

Geometrija:

Trikotnik (stranice a, b, c , oglišča A, B, C s koordinatami (x_i, y_i) , ploščina p , polobseg s , r radij včrtanega in R očrtanega kroga):

$$p = \text{abs} \left(\frac{1}{2} \det \begin{pmatrix} x_2 - x_1 & y_2 - y_1 \\ x_3 - x_1 & y_3 - y_1 \end{pmatrix} \right) = \frac{(B-A) \times (B-C)}{2} = \frac{c \cdot v_c}{2} = \frac{a \cdot b \cdot \sin \gamma}{2} =$$

$$= \sqrt{s(s-a)(s-b)(s-c)} = rs = \frac{abc}{4R}$$

Pravilni mnogokotnik (stranica a , obseg o , ploščina p , $\varphi = \frac{2\pi}{n}$ središčni kot, r radij včrtanega in R očrtanega kroga):

$$p = \frac{nar}{2} = \frac{nR^2 \sin \varphi}{2} = \frac{na^2}{4 \tan \frac{\varphi}{2}} = nr^2 \tan \frac{\varphi}{2}$$

Linearna algebra: Dano imamo rekurzivno zvezo $z_{n+2} = az_{n+1} + bz_n$ in pogoja $z_0 = c_0$ in $z_1 = c_1$. Tako zvezo lahko napišemo v matriko (dodamo trivialne enakosti, če je potrebno):

$$\begin{bmatrix} z_{n+2} \\ z_{n+1} \end{bmatrix} = \begin{bmatrix} a & b \\ 0 & 1 \end{bmatrix} \begin{bmatrix} z_{n+1} \\ z_n \end{bmatrix}$$

Krajše zapisano dobimo $\vec{x}_{n+1} = A\vec{x}_n$, kar odvijemo do $\vec{x}_{n+1} = A^{n+1}\vec{x}_0$, kjer so vse količine znane. S hitrim potenciranjem lahko izračunamo \vec{x}_{n+1} v $O(\log n)$ časa.

Kombinatorika:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} = \prod_{i=0}^{k-1} \frac{n-i}{i+1} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

$$C_n = \frac{1}{n+1} \binom{2n}{n} = \sum_{i=0}^{n-1} C_i C_{n-1-i} = \prod_{i=2}^n \frac{n+i}{i} = \frac{2(2n-1)}{n+1} C_{n-1}$$

$$F_n = F_{n-1} + F_{n-2} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}^{n-1}_{11} \quad F_m F_n + F_{m-1} F_{n-1} = F_{m+n-1} \quad F_m F_{n+1} + F_{m-1} F_n = F_{m+n}$$

Izbori k elementov iz n množice:

urejeni/ponavljanje	DA/DA	DA/NE	NE/DA	NE/NE
število	n^k	$n^{\underline{k}}$	$\binom{n+k-1}{k}$	$\binom{n}{k}$

Binomska in multinomska števila:

$$\text{Velja: } (a+b)^n = \sum_{k=0}^n \binom{n}{k} a^{n-k} b^k \quad \sum_{k=0}^n \binom{n}{k} = 2^n \quad \sum_{k=0}^n \binom{n}{k}^2 = \binom{2n}{n}$$

Pravilo vključitev in izključitev: $|A_1 \cup \dots \cup A_n| = \alpha_1 - \alpha_2 + \dots + (-1)^{n+1} \alpha_n$

α_i = vsota moči vseh možnih presekov po i množic.

V posebnem, če so vsi preseki po i množic enako močni: $|\bigcup_{i=1}^n A_i| = \sum_{i=1}^n (-1)^{i+1} \binom{n}{i} |\bigcap_{j=1}^i A_j|$

Če je problem lep, je to možno implementirati v $O(n^2)$ časa.

Stirlingova števila 2. vrste:

$S(n, k)$ je število možnih razbitij n -množice na k nepraznih kosov.

Definiramo $S(0, 0) = 1$ in $S(n, 0) = 0$ za $n \geq 1$.

Rekurzivna zveza: $S(n, k) = S(n-1, k-1) + k \cdot S(n-1, k)$

Velja: $x^n = \sum_{k=1}^n S(n, k) x^{\underline{k}} \quad S(n+1, m+1) = \sum_{k=1}^n \binom{n}{k} S(k, m)$

Število surjekcij: $k! S(n, k) = \sum_{i=1}^n (-1)^i \binom{k}{i} (k-i)^n$

Lahova števila:

$L(n, k)$ je število možnih razbitij n -množice na k linearne urejenih nepraznih kosov.

Definiramo $L(0, 0) = 1$ in $L(n, 0) = 0$ za $n \geq 1$.

Rekurzivna zveza: $L(n, k) = L(n-1, k-1) + (n+k-1) \cdot L(n-1, k)$

Eksplisitna formula: $L(n, k) = \frac{n!}{k!} \binom{n-1}{k-1} = \frac{(n-1)!}{(k-1)!} \binom{n}{k}$.

Velja: $x^{\bar{n}} = \sum_{k=1}^n L(n, k) x^{\underline{k}}$

Stirlingova števila 1. vrste:

$s(n, k)$ je število permutacij n množice, ki se zapišejo kot produkt k disjunktnih ciklov.

Definiramo $s(0, 0) = 1$ in $s(n, 0) = 0$ za $n \geq 1$.

Rekurzivna zveza: $s(n, k) = s(n-1, k-1) + (n-1) \cdot s(n-1, k)$

Velja: $x^{\bar{n}} = \sum_{k=1}^n s(n, k) x^{\underline{k}}$

Bellova števila:

$B(n)$ je število vseh možnih razbitij n množice. Očitno velja: $\sum_{k=0}^n S(n, k) = B(n)$.

Rekurzivna zveza: $B(n+1) = \sum_{k=0}^n \binom{n}{k} B(k)$

Particije števila:

Particija števila n je zapis $n = \lambda_1 + \dots + \lambda_k$, kjer velja $0 < \lambda_1 \leq \lambda_2 \leq \dots \leq \lambda_k$. λ_i so kosi.

Rekurzivna zveza: $p(n; k) = p(n-1; k-1) + p(n-k; k)$, št. particij n na k kosov.

$p(n; k) = p(n-k; \leq k) = \sum_{i=1}^{n-k} p(n-k; i)$

Dvanajstera pot:

Razporejamo n predmetov v r predalov. Ali ločimo elemente, dopuščamo prazne predale, dopuščamo več kot en predmet v predalu? Glejmo $f: [n] \rightarrow [r]$.

predmeti/predali \ f	poljubna	injektivna	surjektivna
DA/DA	r^n	$r^{\underline{n}}$	$r! S(n, r)$
NE/DA	$\binom{r+n-1}{n}$	$\binom{r}{n}$	$\binom{n-1}{r-1}$
DA/NE	$\sum_{k=1}^r S(n, k)$	$n \leq r$	$S(n, r)$
NE/NE	$\sum_{k=1}^r p(n; k)$	$n \leq r$	$p(n; r)$

Binomska števila: $\binom{n}{k}$

$n \backslash k$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
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	1					
10	1	10	45	120	210	252	210	120	45	10	1				
11	1	11	55	165	330	462	462	330	165	55	11	1			
12	1	12	66	220	495	792	924	792	495	220	66	12	1		
13	1	13	78	286	715	1287	1716	1716	1287	715	286	78	13	1	
14	1	14	91	364	1001	2002	3003	3432	3003	2002	1001	364	91	14	1

Stirlingova števila 2. vrste: $S(n, k)$ in **Bellova števila** $B(n)$

$n \backslash k$	1	2	3	4	5	6	7	8	9	10	$B(n)$
1	1										1
2	1	1									2
3	1	3	1								5
4	1	7	6	1							15
5	1	15	25	10	1						52
6	1	31	90	65	15	1					203
7	1	63	301	350	140	21	1				877
8	1	127	966	1701	1050	266	28	1			4140
9	1	255	3025	7770	6951	2646	462	36	1		21147
10	1	511	9330	34105	42525	22827	5880	750	45	1	115975

Stirlingova števila 1. vrste: $s(n, k)$

$n \backslash k$	1	2	3	4	5	6	7	8	9	10
1	1									
2	1	1								
3	2	3	1							
4	6	11	6	1						
5	24	50	35	10	1					
6	120	274	225	85	15	1				
7	720	1764	1624	735	175	21	1			
8	5040	13068	13132	6769	1960	322	28	1		
9	40320	109584	118124	67284	22449	4536	546	36	1	
10	362880	1026576	1172700	723680	269325	63273	9450	870	45	1

Lahova števila: $L(n, k)$

$n \backslash k$	1	2	3	4	5	6	7	8	9	10
1	1									
2	1	1								
3	1	5	1							
4	1	26	11	1						
5	1	157	103	19	1					
6	1	1100	981	274	29	1				
7	1	8801	9929	3721	593	41	1			
8	1	79210	108091	50860	10837	1126	55	1		
9	1	792101	1268211	718411	191741	26601	1951	71	1	
10	1	8713112	16010633	10607554	3402785	590756	57817	3158	89	1

Particije števila: $p(n; k)$

$n \backslash k$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	1														
2	1	1													
3	1	1	1												
4	1	2	1	1											
5	1	2	2	1	1										
6	1	3	3	2	1	1									
7	1	3	4	3	2	1	1								
8	1	4	5	5	3	2	1	1							
9	1	4	7	6	5	3	2	1	1						
10	1	5	8	9	7	5	3	2	1	1					
11	1	5	10	11	10	7	5	3	2	1	1				
12	1	6	12	15	13	11	7	5	3	2	1	1			
13	1	6	14	18	18	14	11	7	5	3	2	1	1		
14	1	7	16	23	23	20	15	11	7	5	3	2	1	1	
15	1	7	19	27	30	26	21	15	11	7	5	3	2	1	1

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			2178309
64		19,26		89,10	Google		