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, iomanip, 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: 12. oktober 2016

Kazalo

1 Grafi

1.1 Topološko sortiranje

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

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

Časovna zahtevnost: $O(V + E)$

Prostorska zahtevnost: $O(V + E)$

Testiranje na terenu: UVa 10305

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

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

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

        res.push_back(t);

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

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

1.2 Najdaljša pot v DAGu

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

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

Časovna zahtevnost: $O(V + E)$

Prostorska zahtevnost: $O(V + E)$

Testiranje na terenu: UVa 103

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

    max_dist[s] = 0;

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

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

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

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

1.3 Mostovi in prerezna vozlišča grafa

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

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

Časovna zahtevnost: $O(V+E)$

Prostorska zahtevnost: $O(V+E)$

Testiranje na terenu: UVa 315

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

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

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

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

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

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

```

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

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

1.4 Močno povezane komponente

Vhod: Seznam sosednosti s težami povezav.

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

Časovna zahtevnost: $O(V + E)$

Prostorska zahtevnost: $O(V + E)$

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

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

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

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

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

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

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

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

1.5 Najkrajša pot v grafu

1.5.1 Dijkstra

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

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

Časovna zahtevnost: $O(E \log(E))$

Prostorska zahtevnost: $O(V + E)$

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

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

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

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

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

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

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

1.5.2 Dijkstra (kvadratičen)

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

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

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

Prostorska zahtevnost: $O(V + E)$

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

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

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

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

1.5.3 Bellman-Ford

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

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

Časovna zahtevnost: $O(EV)$

Prostorska zahtevnost: $O(V + E)$

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

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

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

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

1.5.4 Floyd-Warhsall

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

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

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

Prostorska zahtevnost: $O(V^2)$

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

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

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

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

1.6 Minimalno vpeto drevo

1.6.1 Prim

Vhod: Neusmerjen povezan graf s poljubnimi cenami povezav.

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

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

Prostorska zahtevnost: $O(V + E)$

Testiranje na terenu: UVa 11631

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

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

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

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

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

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

1.6.2 Kruskal

Vhod: Neusmerjen povezan graf s poljubnimi cenami povezav.

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

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

Prostorska zahtevnost: $O(V + E)$
Testiranje na terenu: UVa 11631

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

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

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

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

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

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

1.7 Najnižji skupni prednik

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

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

Časovna zahtevnost: $O(\log(n))$ na query, s $O(n \log(n))$ predprocesiranja.
Prostorska zahtevnost: $O(n \log(n))$
Testiranje na terenu: `http://www.spoj.com/problems/LCA/`

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

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

```

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

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

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

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

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

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

      if (u == v) return u;

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

      return parent[u];
}
```

1.8 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`

```cpp
vector<vector<int>> all_pairs_widest_path(const vector<vector<pair<int, int>>>& graf_) {
    auto graf = graf_;
    int n = graf.size();
    // Prepiši vse povezave v en velik container
    vector<tuple<int, int, int>> edges;
    for (int i = 0; i < n; i++) {
        sort(graf[i].begin(), graf[i].end(), [](const pair<int, int>& p, const pair<int, int>& q) {
                if (p.second == q.second) return p.first > q.first;
                return p.second > q.second; });
        for (const auto& p : graf[i]) {
            edges.push_back(make_tuple(p.second, p.first, i));  // w, v, u (reversed for sorting)
        }
    }
    sort(edges.begin(), edges.end(), greater<tuple<int, int, int>>());

    vector<vector<int>> result(n, vector<int>(n, 0));  // note: morda ull
    vector<bool> visited;
    queue<int> q;
    vector<int> vertex_degree;  // koliko edgev, ki vodijo iz mesta si ze dodal
    int u, v, w;

    // start in each vertex
    for (int i = 0; i < n; ++i) {
```

```
        visited.assign(n, false);  // Reset these
        vertex_degree.assign(n, 0);
        visited[i] = true;  // I am visited

        // add edges one by one from bigger to smaller weight
        for (auto e : edges) {
            tie(w, v, u) = e;
            vertex_degree[u]++;  // Oznacimo, da smo dodali dodatno cesto

            if (visited[u] && !visited[v]) {  // Ce smo iz ze dosezenega mesta prisli do novega
                visited[v] = true;  // Dodaj se med visited in dodaj svojo sirino v result
                result[i][v] = w;
                if (vertex_degree[v] > 0) {  // Ce je treba: flood fill
                    q.push(v);
                    while (!q.empty()) {
                        int cur = q.front();
                        q.pop();
                        for (int k = 0; k < vertex_degree[cur]; ++k) {
                            auto sosed = graf[cur][k].first;  // samo prvih toliko je obiskanih
                            if (!visited[sosed]) {            // zato smo jih prej sortirali
                                visited[sosed] = true;
                                result[i][sosed] = w;
                                q.push(sosed);
                            }
                        }
                    }
                }
            }
        }
    }
    return result;
}
```

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 sourca 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 dajlša) 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

```
int dfs(int cur, int t, vector<bool>& visited, vector<vector<int>>& flow, int f = 20000) {
    visited[cur] = true;
    if (cur == t) return f;
    int n = flow.size();
    for (int y = 0; y < n; ++y) {
        if (flow[cur][y] > 0 && !visited[y]) {
            int df = dfs(y, t, visited, flow, min(f, flow[cur][y]));
            if (df > 0) {
                flow[cur][y] -= df;
                flow[y][cur] += df;
                return df;
            }
        }
    }
```

```
    return 0;
}

int ford_fulkerson_maximal_flow(const vector<vector<int>>& capacity, int s, int t) {
    vector<vector<int>> flow = capacity;
    int max_flow = 0, df;
    do {
        vector<bool> visited(flow.size(), false);
        df = dfs(s, t, visited, flow);
        max_flow += df;
    } while (df != 0);
    return max_flow;
}
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2 Podatkovne strukture

2.1 Statično binarno iskalno drevo

Operacije: Klasično uravnoteženo binarno iskalno drevo.

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

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

Prostorska zahtevnost: $O(n)$

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

```
struct BST {
    int depth, n;  // 1 << depth mora biti >= številu možnih elementov
    vector<int> tree;  // number of elements in the tree = tree[0]

    BST(int d) : depth(d), n(1 << d), tree(2*n, 0) {}

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

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

    bool remove(int idx) {
        if (tree[n-1 + idx] <= 0) return false;
        insert(idx, -1);
        return true;
    }
};
ostream& operator<<(ostream& os, const BST& t) {
    for (int i = 0; i <= t.depth; ++i) {
        os << string((1 << (t.depth - i)) - 1, ' ');
        for (int j = (1 << i) - 1; j < (1 << (i+1)) - 1; ++j) {
            os << t.tree[j] << string((1 << (t.depth - i + 1)) - 1, ' ');
        }
        os << '\n';
    }
    return os;
}
#endif  // IMPLEMENTACIJA_PS_STATIC_BINARY_SEARCH_TREE_H_
```

2.2 Statično drevo segmentov

Stuktura 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`

```
const int inf = 1e9;
struct RangeQuery {  // Change inf and min below to your operation and initial value
    int n;  // size of the tree
    vector<int> tree;  // 1-based
    RangeQuery(const vector<int>& a) :  // Constructs a queryable structure over you list
            n(1 << static_cast<int>(ceil(log2(a.size())))),
            tree(2*n, inf) {
        for (size_t i = 0; i < a.size(); ++i) tree[n+i] = a[i];
        for (int i = n - 1; i; --i) tree[i] = min(tree[2*i], tree[2*i+1]);
    }
    void set(int pos, int x) {
        for (int i = pos + n; i; i >>= 1) tree[i] = min(tree[i], x);
    }
    int get(int s, int e) const {  // [s, e) i.e. s <= i < e
        int res = inf;
        for (s += n, e += n; s < e; s >>= 1, e >>= 1) {
            if (s & 1) res = min(res, tree[s++]);
            if (e & 1) res = min(res, tree[--e]);
        }
        return res;
    }
};
ostream& operator<<(ostream& os, const RangeQuery& rq) {  // fancy print
    int depth = static_cast<int>(log2(rq.n));
    for (int i = 0; i <= depth; ++i) {
        os << string((1 << (depth - i)) - 1, ' ');
        for (int j = (1 << i); j < (1 << (i+1)); ++j) {
            os << rq.tree[j] << string((1 << (depth - i + 1)) - 1, ' ');
        }
        os << '\n';
    }
    return os;
}
#endif  // IMPLEMENTACIJA_PS_STATIC_SEGMENT_TREE_H_
```

2.3 Drevo segmentov

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

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

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

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

Prostorska zahtevnost: $O(n)$

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

```
struct Node {  // static division points, [l, r] intervals
    int l, m, r;
    Node* left, *right;
    int here_count, total_count;
    Node(int ll, int rr) : l(ll), m((ll+rr) / 2), r(rr),
```

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

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

    int count(int f, int t) {  // count on interval [f, t]
        if (f <= l && r <= t) return total_count;
        int sum = 0, lb = max(l, f), rb = min(r, t);
        if (f <= m && left != nullptr) sum += left->count(f, t);
        if (m + 1 <= t && right != nullptr) sum += right->count(f, t);
        return sum + (rb - lb + 1) * here_count;
    }

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

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

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

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

```

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

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

2.4 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 nekje globalen root in private metode.

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

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

```cpp
        return find(root, value);
    }
    template<typename U>
    friend ostream& operator<<(ostream& os, const AvlTree<U>& tree);

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

```
        if (node == nullptr) return nullptr;
        if (value < node->value) return find(node->left, value);
        else if (value == node->value) return node;
        else return find(node->right, value);
    }

    AvlNode<T>* root;
};

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

2.5 Fenwickovo drevo

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

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

Časovna zahtevnost: $O(k)$ na operacijo
Prostorska zahtevnost: $O(2^k)$
Testiranje na terenu: `http://putka.upm.si/competitions/upm2013-finale/safety`

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

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

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

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

2.6 Fenwickovo drevo (n-dim)

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

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

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

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

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

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

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

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

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

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

2.7 Trie

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

- 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

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

```cpp
    void add(const string& word, size_t idx = 0) {
        if (idx == word.size()) { words++; return; }
        int p = word[idx]-'a';
        if (nodes[p] == nullptr) nodes[p] = new Node();
        total++;
        nodes[p]->add(word, idx+1);
    }
    void print(int n, string& prefix) {
        if (n == 0) return;
        for (int j = 0; j < words; ++j)
            printf("%s\n", prefix.c_str());

        for (int i = 0; i < 26; ++i) {
            if (nodes[i] != nullptr) {
                prefix.push_back('a'+i);
                nodes[i]->print(n-1, prefix);
                prefix.pop_back();
            }
        }
    }
    int longest_common_prefix_length() {  // najdaljši skupen prefix med besedami v drevesu + 1
        int childc = 0;
        int maxl = 0;
        for (int i = 0; i < 26; ++i) {
            if (nodes[i] != nullptr) {
                childc++;
                maxl = max(maxl, nodes[i]->longest_common_prefix_length());
            }
        }
        if (maxl > 0) return maxl+1;  // ce je match v poddrevesu, imamo match+1
        return childc > 1 || (childc == 1 && words > 0) || (words > 1);
    }  // sicer imamo match ce sta dve isti ali imamo otroka in mi eno besedo ali dva otroka
    int common_prefix(const string& s, size_t x = 0) {  // koliko crk imata skupnih
        if (nodes[s[x]-'a'] == nullptr || x == s.size()) return x;
        return nodes[s[x]-'a']->common_prefix(s, x+1);
    }
    int size() { return total+words; }
    ~Node() {
        for (int i = 0; i < 26; ++i)
            delete nodes[i];
    }
};

#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

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

```
            sequence.push_back(a[i-1]);
            i--; j--;
        } else if (c[i][j-1] > c[i-1][j]) {
            j--;
        } else {
            i--;
        }
    }
    reverse(sequence.begin(), sequence.end());
    return sequence;
}

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

3.2 Najdaljše naraščajoče podzaporedje

Vhod: Zaporedje elementov na katerih imamo linearno urejenost.

Izhod: Najdaljše naraščajoče podzaporedje.

Časovna zahtevnost: $O(n \log(n))$ in $O(n^2)$

Prostorska zahtevnost: $O(n)$

Testiranje na terenu: UVa 103

Opomba: Za hitro verzijo je zaradi bisekcije potrebna linearna urejenost elementov. Pri n^2 verziji je dovolj delna urejenost. V tem primeru je elemente morda treba urediti, tako da je potem potrebno za urejanje izbrati neko linearno razširitev dane delne urejenosti. Pri obeh verzijah elementi niso omejeni na števila, vendar pri prvi ne moremo samo zamenjati tipa, ki ga funkcija vrača, lažje je spremeniti, da vrača indekse elementov namesto dejanskega zaporedja.

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

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

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

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

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

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

```
}

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

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

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

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

3.3 Najdaljši strnjen palindrom

Vhod: Niz s dolžine n.

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

Časovna zahtevnost: $O(n)$

Prostorska zahtevnost: $O(n)$

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

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

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

        if (i + L[i] > R) {  // If palindrome centered at currentRightPosition i
```

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

3.4 Podseznam z največjo vsoto

Vhod: Zaporedje elementov a_i dolžine n.

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

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

Prostorska zahtevnost: $O(n)$

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

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

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

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

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

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

```
    }
    return v;
}
```

3.5 Leksikografsko minimalna rotacija

Vhod: Niz znakov s dolžine n.

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

Časovna zahtevnost: $O(n)$

Prostorska zahtevnost: $O(n)$

Testiranje na terenu: UVa 719

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

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

3.6 BigInt in Karatsuba

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

Operacije: Seštevanje, odštevanje, množenje, primerjanje.

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

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

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

Prostorska zahtevnost: $O(n)$

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

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

    Number() {}  // default constructor, positive zero
    Number(const deque<T>& a, bool s = 1) : sign(s), data(a) { clear_zeros(); }
    Number(deque<T>&& a, bool s = 1) : sign(s), data(a) { clear_zeros(); }
    Number(const string& s) { from_string(s); }
    void from_string(const string& s) {
```

```
        if (s.size() == 0) data = {0};
        int i = 0;
        if (s[0] == '+') sign = 1, i = 1;
        if (s[0] == '-') sign = 0, i = 1;
        int l = s.size(); data.resize(l-i);
        for (; i < l; ++i) {
            if (!('0' <= s[i] && s[i] <= '9')) return;  // silent quit po prvi ne stevilki
            data[l-i-1] = s[i] - '0';
        }
        clear_zeros();
    }
    string to_string() const {
        if (data.empty()) return "0";
        string s = (sign) ? "" : "-";
        for (int i = data.size() - 1; i >= 0; --i)
            s.push_back('0' + data[i]);
        return s;
    }
    Number operator+(const Number& o) const {  // remove signs if using for positive only
        if (sign == 0) return -((-*this) + (-o));
        if (sign == 1 && o.sign == 0) return (*this < -o) ? -((-o) - *this) : *this - (-o);
        bool carry = false;
        int i = 0, j = 0, n = data.size(), m = o.data.size();
        deque<T> r;
        while (i < n || j < m) {
            T c = ((i < n) ? data[i++] : 0) + ((j < m) ? o.data[j++] : 0) + carry;
            carry = (c >= base);
            r.push_back(c % base);
        }
        if (carry) r.push_back(1);
        return Number(move(r));
    }
    Number operator-() const { return Number(data, !sign); }
    bool operator==(const Number& o) const { return sign == o.sign && data == o.data; }
    bool operator<(const Number& o) const {
        if (sign == o.sign) {
            if (sign == 0) return -o < -*this;
            if (data.size() == o.data.size()) {
                for (int i = data.size() - 1; i >= 0; --i)
                    if (data[i] == o.data[i]) continue;
                    else return data[i] < o.data[i];
            }
            return data.size() < o.data.size();
        }
        return sign < o.sign;
    }
    Number& operator+=(const Number& o) {  // lahko tudi s +. Samo za pozitivne.
        bool carry = false;
        int i = 0, j = 0, n = data.size(), m = o.data.size();
        while (i < n || j < m) {
            if (i < n && j < m) data[i] += o.data[j++] + carry;
            else if (i < n)     data[i] += carry;
            else                data.push_back(o.data[j++] + carry);
            carry = data[i] / base;
            data[i++] %= base;
        }
        if (carry) data.push_back(1);
        clear_zeros();
        return *this;
    }
    Number operator*(const T& o) const {  // z eno števko
        deque<T> r;
        int carry = 0, n = data.size();
        for (int i = 0; i < n; ++i) {
            T c = data[i]*o + carry;
            carry = c / base;
            c %= base;
            r.push_back(c);
        }
        if (carry) r.push_back(carry);
        return Number(move(r), sign);
    }
    Number operator<<(int n) const {  // na zacetek dodamo n ničel
        deque<T> r(n, 0);
        r.insert(r.end(), data.begin(), data.end());
        return Number(move(r));
    }
    Number operator*(const Number<T>& o) const {
        if (sign == 0 && o.sign == 0) return ((-*this) * (-o));
        if (sign == 0 && o.sign == 1) return -((-*this) * o);
        if (sign == 1 && o.sign == 0) return -(*this * (-o));
```

```
        Number r;
        int m = o.data.size();
        for (int i = 0; i < m; ++i)
            r += (*this*o.data[i] << i);
        return r;
    }
    Number operator-(const Number& o) const {
        deque<T> r;
        bool carry = false;
        int i = 0, j = 0, n = data.size(), m = o.data.size();
        while (i < n || j < m) {
            T c = data[i++] + base - ((j < m) ? o.data[j++] : 0) - carry;
            carry = 1 - c / base;
            c %= base;
            r.push_back(c);
        }
        return Number(move(r));
    }

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

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

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

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

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

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

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

3.7 2-SAT

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

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

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

```
int var2node(int var, int n) {  // sprem. od ±1 do ±n spremeni v vozlišče od 0 do 2n − 1
    return (var > 0) ? var - 1 : -var - 1 + n;
}

namespace {
int dfscount;
vector<int> postorder;
vector<int> comp;  // comp[i] pove v kateri componenti je i. Componente so top. urejene.
vector<bool> visited;
}

void get_postorder(const vector<vector<int>>& G, int v) {  // izračuna čase odhodov iz vozlišč
    visited[v] = true;
    for (int u : G[v])
        if (!visited[u])
            get_postorder(G, u);
    postorder[dfscount--] = v;
}

void mark_comp(const vector<vector<int>>& G, int v, int scccount) {  // najde povezane komponente
    comp[v] = scccount;
    for (int u : G[v])
        if (comp[u] == -1)
            mark_comp(G, u, scccount);
}

vector<int> solve_2sat(const vector<pair<int, int>>& formula, int n) {
    vector<vector<int>> G(2*n), GR(2*n);
    int x, y;
    for (const auto& term : formula) {  // Imamo stavek x v y.
        tie(x, y) = term;  // Naredimo dva sklepa: -x => y, -y => x
        int tx = var2node(x, n), ty = var2node(y, n);
        int nx = var2node(-x, n), ny = var2node(-y, n);
        G[nx].push_back(ty); G[ny].push_back(tx);   // naredimo graf implikacij
        GR[ty].push_back(nx); GR[tx].push_back(ny);  // in obraten graf
    }

    dfscount = 2*n-1;
    postorder.resize(2*n);
    visited.assign(2*n, false);
    for (int i = 0; i < 2*n; ++i)
        if (!visited[i])
            get_postorder(G, i);

    comp.assign(2*n, -1);
    int scccount = 0;
    for (int v : postorder)
        if (comp[v] == -1)
            mark_comp(GR, v, scccount++);

    for (int i = 0; i < n; ++i) {
        if (comp[i] == comp[n+i])
            return vector<int>(n, -1);  // ali false
    }  // ce ne rabis resitve lahko das tukaj return true;

    vector<int> solution(n);
    for (int i = 0; i < n; ++i)
        solution[i] = comp[i] > comp[n+i];
    return solution;
}

bool evaluate(const vector<pair<int, int>>& formula, const vector<int>& values) {
    int x, y;  // izračuna vrednost formule pri danem naboru vrednosti spremenljivk
    for (const auto& p : formula) {  // pricakujemo da so v formuli stevilke od \pm 1 do \pm n
        tie(x, y) = p;               // in da imamo n vrednosti, ki so 0 ali 1
        bool vx = values[abs(x)-1] ^ (x < 0);
        bool vy = values[abs(y)-1] ^ (y < 0);
        if (!(vx || vy)) return false;
    }
    return true;
}
```

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 nekje naprej.

Izhod: Najmanjši indeks $0 \leqslant i < n$, 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 $\texttt{ff}[i-1]$ 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/`

```
vector<int> compute_failure_function(const string& p) {
    int m = p.size();
    vector<int> ff(m, 0);
    for (int k = 0, i = 1; i < m; ++i) {
        while (k > 0 && p[i] != p[k]) k = ff[k-1];
        if (p[i] == p[k]) k++;
        ff[i] = k;
    }
    return ff;
}

int knuth_morris_pratt(const string& s, const string& p, const vector<int>& ff, int start) {
    int k = 0, n = s.size(), m = p.size();
    for (int i = start; i < n; i++) {
        while (k > 0 && p[k] != s[i]) k = ff[k-1];
        if (s[i] == p[k]) k++;
        if (k == m) return i - k + 1;
    }
    return -1;
}

int knuth_morris_pratt(const string& s, const string& p) {
    vector<int> ff = compute_failure_function(p);
    return knuth_morris_pratt(s, p, ff, 0);
}

vector<int> find_all_occurences(const string& s, const string& p) {
    vector<int> ff = compute_failure_function(p), result;
    int i = -1;
    while ((i = knuth_morris_pratt(s, p, ff, i + 1)) != -1) {
        result.push_back(i);  // or do something else
    }
    return result;
}
vector<int> find_non_overlaping_occurences(const string& s, const string& p) {
    vector<int> ff = compute_failure_function(p), result;
    int i = -p.size();
    while ((i = knuth_morris_pratt(s, p, ff, i + p.size())) != -1) {
        result.push_back(i);  // or do something else
    }
    return result;
}
int minimal_period(const string& s) {
    int n = s.size();
    vector<int> ff = compute_failure_function(s);
    int candidate = n - ff.back();
    if (n % candidate == 0) return candidate;
    return n;
}
```

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 $s =$ "*aaaaa*" je $z = [0,4,3,2,1]$, za $s =$ "*aaabaab*" je $z = [0,2,1,0,2,1,0]$, za $s =$ "*abacaba*" je $z = [0,0,1,0,3,0,1]$.

Časovna zahtevnost: $O(n)$

Prostorska zahtevnost: $O(n)$

Testiranje na terenu: TODO

```
vector<int> z_function(const string& s) {
    int n = s.length();
    vector<int> z(n);
    for (int i = 1, l = 0, r = 0; i < n; ++i) {
        if (i <= r) z[i] = min(r - i + 1, z[i - l]);
        while (i + z[i] < n && s[z[i]] == s[i + z[i]]) ++z[i];
        if (i + z[i] - 1 > r) l = i, r = i + z[i] - 1;
    }
    return z;
}
int match(const string& s, const string& p) {
    string t = p + '#' + s;  // # does not appear in s or p
    vector<int> z = z_function(t);
    int n = s.size(), m = p.size();
    for (int i = 0; i < n; ++i)
        if (z[i+m+1] == m)
            return i;
    return -1;
}
int minimal_period_zf(const string& s) {
    int n = s.size();
    vector<int> z = z_function(s);
    for (int i = 1; i < n; ++i)
        if (n % i == 0 && i + z[i] == n)  // ce ni pogoja n % i == 0 najdemo tudi necele periode
            return i;  // če najdemo vse take i, najdemo vse periode
    return n;
}
```

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 $s =$ "*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 $a+2^r \leqslant b$ 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 (dolg 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 ali pa po modulu p, kjer je potrebno deljenja zamenjati z inverzi.

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

Prostorska zahtevnost: $O(nm)$

```
namespace {
const double eps = 1e-9;
}

int gauss(vector<vector<double>> A, vector<double>& ans) {
    int n = A.size(), m = A[0].size() - 1;

    vector<int> where(m, -1);
    for (int col = 0, row = 0; col < m && row < n; ++col) {
        int sel = row;
        for (int i = row; i < n; ++i)
            if (abs(A[i][col]) > abs(A[sel][col])) sel = i;
        if (abs(A[sel][col]) < eps) continue;
        for (int i = col; i <= m; ++i) swap(A[sel][i], A[row][i]);
        where[col] = row;

        for (int i = 0; i < n; ++i)
            if (i != row) {
                double c = A[i][col] / A[row][col];
                for (int j = col; j <= m; ++j) A[i][j] -= A[row][j] * c;
            }
        ++row;
    }  // tukaj je v A spravljen  LU razcep PA = LU.

    ans.assign(m, 0);
    for (int i = 0; i < m; ++i)
        if (where[i] != -1)
            ans[i] = A[where[i]][m] / A[where[i]][i];

    // Preverjamo rank in to, da rešitev res reši sistem
    for (int i = 0; i < n; ++i) {
        double sum = 0;
        for (int j = 0; j < m; ++j) sum += ans[j] * A[i][j];
        if (abs(sum - A[i][m]) > eps) return -1;
    }

    return count(where.begin(), where.end(), -1);
}

double det(vector<vector<double>> a) {
    double det = 1;
    int n = a.size();
    for (int i = 0; i < n; ++i) {
        int k = i;
        for (int j = i + 1; j < n; ++j)
            if (abs(a[j][i]) > abs (a[k][i]))
                k = j;
        if (abs(a[k][i]) < eps) {
            det = 0;
            break;
        }
        swap(a[i], a[k]);
        if (i != k) det = -det;
        det *= a[i][i];
        for (int j = i + 1; j < n; ++j) {
            a[i][j] /= a[i][i];
        }
        for (int j = 0; j < n; ++j) {
            if (j != i && abs(a[j][i]) > eps) {
```

```
                for (int k = i + 1; k < n; ++k) {
                    a[j][k] -= a[i][k] * a[j][i];
                }
            }
        }
    }
    return det;
}
```

4.2 Tangentna metoda

Vhod: Funkciji f in f' ter približek za ničlo x_0.

Izhod: Približek x_n po n korakih metode $x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$ ali ko je zaporedni premik manjši od ε.

Časovna zahtevnost: Konvergenca je kvadratična, torej $O(\log \frac{1}{\sqrt{\varepsilon}})$.

Prostorska zahtevnost: $O(1)$

```
double tangent_method(function<double(double)> f, function<double(double)> df, double x0,
                      double eps, int max_iterations) {
    double xn = x0;
    for (int i = 0; i < max_iterations; ++i) {
        x0 = xn;
        xn = x0 - f(x0) / df(x0);
        if (abs(x0-xn) < eps * abs(xn)) break;
    }
    return xn;
}
```

4.3 Bisekcija

Vhod: Funkcija f in krajišči intervala $[a, b]$ na katerem leži ničla in ima funkcija v krajiščih različen predznak.

Izhod: Približek za ničlo, ko je širina intervala enaka $\varepsilon|b - a|$ ali približek po n korakih.

Časovna zahtevnost: Konvergenca je linearna, torej $O(\log \frac{1}{\varepsilon})$.

Prostorska zahtevnost: $O(1)$

```
int sgn(double x) {
    return (x > 0.0) ? 1 : -1;
}

double bisection(function<double(double)> f, double a, double b, double eps) {
    double fa = f(a);
    double fb = f(b);
    if (b < a || sgn(fa) == sgn(fb)) return numeric_limits<double>::signaling_NaN();
    double d = b - a, c = 0.0, fc = 0.0;
    double prec = eps * d;
    while (d > prec) {
        d /= 2.0;
        c = a + d;
        fc = f(c);
        if (sgn(fa) == sgn(fc)) {
            a = c;
            fa = fc;
        }
    }
    return c;
}
```

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: $O(1)$

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

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 Z_b^*.

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

Prostorska zahtevnost: $O(1)$

Testiranje na terenu: UVa 756

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

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

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

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

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

Prostorska zahtevnost: $O(n)$

Potrebuje: Evklidov algoritem (str. 33)

Testiranje na terenu: UVa 756

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

```
int mul_inverse(int a, int m) {
    int x, y;
```

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

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

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`

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

5.5 Številski sestavi

Vhod: Število $n \in \mathbb{N}_0$ ali $\frac{p}{q} \in Q$ ter $b \in [2, \infty) \cap \mathbb{N}$.
Izhod: Število n ali $\frac{p}{q}$ predstavljeno v izbranem sestavu z izbranimi števkami in označeno periodo.
Časovna zahtevnost: $O(\log(n))$ ali $O(q \log(q))$
Prostorska zahtevnost: $O(n)$ ali $O(q)$
Testiranje na terenu: `http://putka.upm.si/tasks/2010/2010_finale/ulomki`
Opomba: Zgornja meja za bazo b je dolžina niza `STEVILSKI_SESTAVI_ZNAKI`.

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

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

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

    string decimalke;  // decimalni del
```

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

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:

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

Časovna zahtevnost: $O(\sqrt{n})$

Prostorska zahtevnost: $O(1)$

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

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

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 praštevilski delitelj. To se lahko uporablja za faktorizacijo števil in testiranje praštevilskosti.

Časovna zahtevnost: $O(n \log(n))$

Prostorska zahtevnost: $O(n)$

Testiranje na terenu: UVa 10394

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

```
        while (j < primes.size() && primes[j] <= is_prime[i] && i * primes[j] <= n) {
            is_prime[i * primes[j]] = primes[j];
            j++;
        }
    }
}
```

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`

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

5.9 Binomski koeficienti

Vhod: Števili $n, k \in \mathbb{Z}$.

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

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`

```
int binomial(int n, int k) {
    if (k < 0 || k > n) return 0;
    if (k == 0 || k == n) return 1;
    k = min(k, n - k);
    int r = 1;
    for (int i = 0; i < k; ++i) {
        r *= n - i;
        r /= i + 1;  // deljenje se vedno izide
    }
    return r;
}

double binomial_approx(double n, double k) {
    return exp(lgamma(n+1) - lgamma(k+1) - lgamma(n-k+1));
}

```

```
vector<vector<int>> pascal_triangle(int n) {
    vector<vector<int>> ret(n, vector<int>(n, 0));
    ret[0][0] = 1;
    for (int i = 1; i < n; ++i) {
        ret[i][0] = 1;
        for (int j = 1; j <= i; ++j) {
            ret[i][j] = ret[i-1][j] + ret[i-1][j-1];
        }
    }
    return ret;
}
```

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:

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

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. 34)

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

```
int binomial_modp(int n, int k, int p) {
    if (k < 0 || k > n) return 0;
    if (k == 0 || k == n) return 1;
    int r = 1;
    while (n > 0) {
        int ni = n % p, ki = k % p;
        r *= binomial(ni, ki);  // če imaš zračunano vnaprej, uporabi tisto
        r %= p;  // sicer pa delaj modulo tudi znotraj binomial
        n /= p;
        k /= p;
    }
    return r;
}

// funkcija je tu bolj za demonstracijo, v praksi racunaj vse rezultate po modulih
int binomial_mod(int n, int k, int m) {  // prafaktorjev in sele na koncu rekunstruiraj
    if (m == 1) return 0;
    if (k < 0 || k > n) return 0;
    if (k == 0 || k == n) return 1;
    vector<pair<int, int>> mods;
    int i = 2;
    while (i * i <= m) {
        int p = 0;
        while (m % i == 0) {  // i je prafaktor n, s potenco p
            m /= i;
            p++;
        }
        assert(p <= 1 && "Invalid number, numbers containing prime powers not supported.");
        if (p == 1) mods.emplace_back(binomial_modp(n, k, i), i);

        i++;
    }  // kar ostane mora biti praštevilo ali 1
    if (m != 1) mods.emplace_back(binomial_modp(n, k, m), m);
```

```cpp

    return chinese_reminder_theorem(mods);
}
```

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…

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

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

template<typename T>
struct line_t {  // premica, dana z enačbo ax + by = c ali z dvema točkama
    double a, b, c;  // lahko tudi int
    line_t() : a(0), b(0), c(0) {}
    line_t(int A, int B, int C) {
        if (A < 0 || (A == 0 && B < 0)) a = -A, b = -B, c = -C;
        else a = A, b = B, c = C;
        int d = gcd(gcd(abs(a), abs(b)), abs(c));  // same sign as A, if nonzero, else B, else C
        if (d == 0) d = 1;                          // in case of 0 0 0 input
        a /= d;
        b /= d;
        c /= d;
    }
    line_t(T A, T B, T C) {
        if (A < 0 || (A == 0 && B < 0)) a = -A, b = -B, c = -C;
        else a = A, b = B, c = C;
```

```
    }
    line_t(const P& p, const P& q) : line_t(imag(q-p), real(p-q), cross(p, q)) {}
    P normal() const { return {a, b}; }
    double value(const P& p) const { return dot(normal(), p) - c; }
    bool operator<(const line_t<T>& line) const {  // da jih lahko vržemo v set, če T = int
        if (a == line.a) {
            if (b == line.b) return c < line.c;
            return b < line.b;
        }
        return a < line.a;
    }
    bool operator==(const line_t<T>& line) const {
        return cross(normal(), line.normal()) < eps && c*line.b == b*line.c;
    }
};
template<typename T>
ostream& operator<<(ostream& os, const line_t<T>& line) {
    os << line.a << "x + " << line.b << "y == " << line.c; return os;
}

typedef line_t<double> L;

#endif  // IMPLEMENTACIJA_GEOM_BASICS_H_
```

```
double dot(const P& p, const P& q) {
    return p.real() * q.real() + p.imag() * q.imag();
}
double cross(const P& p, const P& q) {
    return p.real() * q.imag() - p.imag() * q.real();
}
double cross(const P& p, const P& q, const P& r) {
    return cross(q - p, r - q);  // > 0 levo, < 0 desno, = 0 naravnost
}
// true is p->q->r is a left turn, straight line is not, if so, change to -eps
bool left_turn(const P& p, const P& q, const P& r) {
    return cross(q-p, r-q) > eps;
}
P perp(const P& p) {  // get left perpendicular vector
    return P(-p.imag(), p.real());
}
int sign(double x) {
    if (x < -eps) return -1;
    if (x > eps) return 1;
    return 0;
}
double polar_angle(const P& p) {  // phi in [0, 2pi) or -1 for (0,0)
    if (p == P(0, 0)) return -1;
    double a = arg(p);
    if (a < 0) a += 2*pi;
    return a;
}
double area(const P& a, const P& b, const P& c) {  // signed
    return 0.5 * cross(a, b, c);
}
double area(const vector<P>& poly) {  // signed
    double A = 0;
    int n = poly.size();
    for (int i = 0; i < n; ++i) {
        int j = (i+1) % n;
        A += cross(poly[i], poly[j]);
    }
    return A/2;
}
double dist_to_line(const P& p, const L& line) {
    return abs(line.value(p)) / abs(line.normal());
}
double dist_to_line(const P& t, const P& p1, const P& p2) {  // t do premice p1p2
    return abs(cross(p2-p1, t-p1)) / abs(p2-p1);
}
double dist_to_segment(const P& t, const P& p1, const P& p2) {  // t do daljice p1p2
    P s = p2 - p1;
    P w = t - p1;
    double c1 = dot(s, w);
    if (c1 <= 0) return abs(w);
    double c2 = norm(s);
    if (c2 <= c1) return abs(t-p2);
    return dist_to_line(t, p1, p2);
}
double great_circle_dist(const P& a, const P& b) {  // pairs of (latitude, longitude) in radians
    double R = 6371.0;                               // compute great circle distance
    double u[3] = { cos(a.real()) * sin(a.imag()), cos(a.real()) * cos(a.imag()), sin(a.real()) };
```

```
    double v[3] = { cos(b.real()) * sin(b.imag()), cos(b.real()) * cos(b.imag()), sin(b.real()) };
    double dot = u[0]*v[0] + u[1]*v[1] + u[2]*v[2];
    bool flip = false;
    if (dot < 0.0) {
        flip = true;
        for (int i = 0; i < 3; i++) v[i] = -v[i];
    }
    double cr[3] = { u[1]*v[2] - u[2]*v[1], u[2]*v[0] - u[0]*v[2], u[0]*v[1] - u[1]*v[0] };
    double theta = asin(sqrt(cr[0]*cr[0] + cr[1]*cr[1] + cr[2]*cr[2]));
    double len = theta * R;
    if (flip) len = pi * R - len;
    return len;
}
bool point_in_rect(const P& t, const P& p1, const P& p2) {  // ali je t v pravokotniku p1p2
    return min(p1.real(), p2.real()) <= t.real() && t.real() <= max(p1.real(), p2.real()) &&
           min(p1.imag(), p2.imag()) <= t.imag() && t.imag() <= max(p1.imag(), p2.imag());
}
bool point_in_triangle(const P& t, const P& a, const P& b, const P& c) {  // orientation independant
    return abs(abs(area(a, b, t)) + abs(area(a, c, t)) + abs(area(b, c, t))  // edge inclusive
                - abs(area(a, b, c))) < eps;
}
pair<ITYPE, P> line_line_intersection(const L& p, const L& q) {
    double det = cross(p.normal(), q.normal());  // če imata odvisni normali (ali smerna vektorja)
    if (abs(det) < eps) {  // paralel
        if (abs(p.b*q.c - p.c*q.b) < eps && abs(p.a*q.c - p.c*q.a) < eps) {
            return {EQ, P()};  // razmerja koeficientov se ujemajo
        } else {
            return {NO, P()};
        }
    } else {
        return {OK, P(q.b*p.c - p.b*q.c, p.a*q.c - q.a*p.c) / det};
    }
}
pair<ITYPE, P> line_segment_intersection(const L& p, const P& u, const P& v) {
    double u_on = p.value(u);
    double v_on = p.value(v);
    if (abs(u_on) < eps && abs(v_on) < eps) return {EQ, u};
    if (abs(u_on) < eps) return {OK, u};
    if (abs(v_on) < eps) return {OK, v};
    if ((u_on > eps && v_on < -eps) || (u_on < -eps && v_on > eps)) {
        return line_line_intersection(p, L(u, v));
    }
    return {NO, P()};
}
pair<ITYPE, P> segment_segment_intersection(const P& p1, const P& p2, const P& q1, const P& q2) {
    int o1 = sign(cross(p1, p2, q1));  // daljico p1p1 sekamo z q1q2
    int o2 = sign(cross(p1, p2, q2));
    int o3 = sign(cross(q1, q2, p1));
    int o4 = sign(cross(q1, q2, p2));

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

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

    return {NO, P()};
}
ITYPE point_in_poly(const P& t, const vector<P>& poly) {
    int n = poly.size();
    int cnt = 0;
    double x2 = rand() % 100;
    double y2 = rand() % 100;
    P dalec(x2, y2);
    for (int i = 0; i < n; ++i) {
        int j = (i+1) % n;
        if (dist_to_segment(t, poly[i], poly[j]) < eps) return EQ;  // boundary
        ITYPE tip = segment_segment_intersection(poly[i], poly[j], t, dalec).first;
        if (tip != NO) cnt++;  // ne testiramo, ali smo zadeli oglisce, upamo da nismo
    }
    if (cnt % 2 == 0) return NO;
    else return OK;
}
pair<P, double> get_circle(const P& p, const P& q, const P& r) {  // circle through 3 points
    P v = q-p;
    P w = q-r;
```

```
    if (abs(cross(v, w)) < eps) return {P(), 0};
    P x = (p+q)/2.0, y = (q+r)/2.0;
    ITYPE tip;
    P intersection;
    tie(tip, intersection) = line_line_intersection(L(x, x+perp(v)), L(y, y+perp(w)));
    return {intersection, abs(intersection-p)};
}
// circle through 2 points with given r, to the left of pq
P get_circle(const P& p, const P& q, double r) {
    double d = norm(p-q);
    double h = r*r / d - 0.25;
    if (h < 0) return P(inf, inf);
    h = sqrt(h);
    return (p+q) / 2.0 + h * perp(q-p);
}
```

6.2 Konveksna ovojnica

Vhod: Seznam n točk.

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

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

Prostorska zahtevnost: $O(n)$

Potrebuje: Vektorski produkt, str. 39.

Testiranje na terenu: UVa 681

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

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

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

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

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

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

    return hull;
}
```

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`

```cpp
typedef complex<int> P;

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

vector<int> points;  // y-coordinates of rect sides (can be double)

struct Node {  // segment tree, s in e sta zacetek in konec intervala, torej en node pokrije [s, e]
    int s, e, m, c, a;  // start, end, middle, count, area
    Node *left, *right;
    Node(int s_, int e_) : s(s_), e(e_), m((s+e)/2), c(0), a(0), left(nullptr), right(nullptr) {
        if (e-s == 1) return;
        left = new Node(s, m);
        right = new Node(m, e);
    }
    int add(int f, int t) {  // returns area
        if (f <= s && e <= t) {
            c++;
            return a = points[e] - points[s];
        }
        if (f < m) left->add(f, t);
        if (t > m) right->add(f, t);
        if (c == 0) a = left->a + right->a;  // če nimam lastnega intervala, izračunaj
        return a;
    }
    int remove(int f, int t) {  // returns area
        if (f <= s && e <= t) {
            c--;
            if (c == 0) {  // če nima lastnega intervala
                if (left == nullptr) a = 0;  // če je list je area 0
                else a = left->a + right->a;  // če ne je vsota otrok
            }
            return a;
        }
        if (f < m) left->remove(f, t);
        if (t > m) right->remove(f, t);
        if (c == 0) a = left->a + right->a;
        return a;
    }
};

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

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

    P levo_spodaj, desno_zgoraj;  // pravokotniki so dani tako, ce v nalogi niso, zamenjaj x1 <-> x2
    for (int i = 0; i < n; ++i) {
        tie(levo_spodaj, desno_zgoraj) = rects[i];
        int a = levo_spodaj.real();   //         +----------+ (c, d)
        int c = desno_zgoraj.real();  //         |          |
        int b = levo_spodaj.imag();   //         |          |
        int d = desno_zgoraj.imag();  //  (a, b) +----------+
        verts.push_back(vert(a, b, d, true));
        verts.push_back(vert(c, b, d, false));
        points[2*i] = b;
        points[2*i+1] = d;
    }

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

    Node * sl = new Node(0, points.size() - 1);  // sweepline segment tree, po celem seznamu

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

```

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

    return area;
}
```

6.4 Najbližji par točk v ravnini

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

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

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

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

Testiranje na terenu: UVa 10245

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

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

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

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

7 Matematika

Vrste:

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

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

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):

$$p = \operatorname{abs}\left(\frac{1}{2}\det\begin{pmatrix} x_2-x_1 & y_2-y_1 \\ x_3-x_1 & y_3-y_1 \end{pmatrix}\right) = \frac{(B-A)\times(B-C)}{2} = \frac{c\cdot v_c}{2} = \frac{a\cdot b\cdot \sin\gamma}{2} =$$
$$= \sqrt{s(s-a)(s-b)(s-c)} = rs = \frac{abc}{4R}$$

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:

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

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

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

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

Pravilo vključitev in izključitev: $|A_1 \cup \cdots \cup A_n| = \alpha_1 - \alpha_2 + \cdots + (-1)^{n+1}\alpha_n$
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 $n \geqslant 1$.
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 linearno urejenih nepraznih kosov.
Definiramo $L(0,0)=1$ in $L(n,0)=0$ za $n \geqslant 1$.
Rekurzivna zveza: $L(n,k)=L(n-1,k-1)+(n+k-1)\cdot L(n-1,k)$
Eksplicitna 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 $n \geqslant 1$.
Rekurzivna zveza: $s(n,k)=s(n-1,k-1)+(n-1)\cdot s(n-1,k)$
Velja: $x^{\overline{n}}=\sum_{k=1}^{n} s(n,k)x^k$
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 $n=\lambda_1+\cdots+\lambda_k$, kjer velja $0<\lambda_1\leqslant\lambda_2\leqslant\cdots\leqslant\lambda_k$. λ_i so kosi.
Rekurzivna zveza: $p(n;k)=p(n-1;k-1)+p(n-k;k)$, št. particij n na k kosov.
$p(n;k)=p(n-k;\leqslant k)=\sum_{i=1}^{n-k}p(n-k;i)$
Dvanajstera pot:
Razporejamo n predmetov v r predalov. Ali ločimo elemente, dopuščamo prazne predale, dopuščamo več kot en predmet v predalu? Glejmo $f\colon[n]\to[r]$.

predmeti/predali $\setminus f$	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	$\sum_{k=1}^{r} S(n,k)$	$n\leqslant r$	$S(n,r)$
NE/NE	$\sum_{k=1}^{r} p(n;k)$	$n\leqslant r$	$p(n;r)$

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

Številske 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		