

1 Contest

2 Struktury danych

3 Grafy

4 Matma

5 Teksty

Contest (1)

```
sol.cpp

#include <bits/stdc++.h>
using namespace std;
using ll = long long;

#ifdef LOCAL
auto& operator<<(auto&, pair<auto, auto>);
template <typename T, typename = T::value_type>
auto& operator<<(auto& o, T x) {
    o << "{";
    string s;
    for (auto i : x) {
        o << s << i;
        s = ", ";
    }
    return o << "}";
}
auto& operator<<(auto& o, pair<auto, auto> x) {
    return o << "(" << x.first << ", " << x.second << ")";
}
#define debug(x...) \
    cerr << "[" #x "]:", \
    [] (auto... y) { ((cerr << " " << y), ...) << endl; } (x)
#else
#define debug(...) 2137
#endif
int main() {
    ios_base::sync_with_stdio(false);
    cin.tie(nullptr);
}
```

.vimrc

```
set autoindent expandtab tabstop=2 shiftwidth=2 nu ic
    timeoutlen=1000 ttimeoutlen=10 hidden smartindent
syntax enable
colorscheme retrobox
inoremap {<cr> {<cr>}<esc>O <bs>
nnoremap ; :
nnoremap : ;
nnoremap <space> `:b#<cr>
filetype indent on
```

Makefile

```
CXXFLAGS=-std=c++20 -Wall -Wextra -Wshadow

sol: sol.cpp
    g++ $(CXXFLAGS) -fsanitize=address,undefined -g -DLOCAL sol.
        cpp -o sol
```

```
1 fast: sol.cpp
    g++ $(CXXFLAGS) -O2 sol.cpp -o fast

1 test.sh

2 #!/bin/bash

3 for((i=1;i>0;i++)) do
    echo "$i"
    echo "$i" | ./gen > int
    diff -w <(. /sol < int) <(. /slow < int) || break
4 done
```

Struktury danych (2)

Drzewo falkowe

Opis: Obsługuje zapytania typu *podaj k-ty najmniejszy na przedziale* itp. na statycznej tablicy. Jeżeli czas albo pamięć są ciasne warto przeskalować liczby. Niszczy tablicę.

Czas: $\mathcal{O}(\log A)$

```
struct node {
    int lo, hi;
    vector<int> s;
    node* l = 0;
    node* r = 0;
    node(int _lo, int _hi, auto st, auto ed) {
        lo = _lo;
        hi = _hi;
        if (lo + 1 < hi) {
            int mid = (lo + hi) / 2;
            s.reserve(ed - st + 1);
            s.push_back(0);
            for (auto it = st; it != ed; it++) {
                s.push_back(s.back() + (*it < mid));
            }
            auto k = stable_partition(
                st, ed, [&](int x) { return x < mid; });
            if (k != st) l = new node(lo, mid, st, k);
            if (k != ed) r = new node(mid, hi, k, ed);
        }
    }
    int kth(int a, int b, int k) {
        if (lo + 1 == hi) return lo;
        int x = s[a];
        int y = s[b];
        return k < y - x ? l->kth(x, y, k)
            : r->kth(a - x, b - y, k - (y - x));
    }
    int count(int a, int b, int k) {
        if (lo >= k) return 0;
        if (hi <= k) return b - a;
        int x = s[a];
        int y = s[b];
        return (l ? l->count(x, y, k) : 0) +
            (r ? r->count(a - x, b - y, k) : 0);
    }
    int freq(int a, int b, int k) {
        if (k < lo || hi <= k) return 0;
        if (lo + 1 == hi) return b - a;
        int x = s[a];
        int y = s[b];
        return (l ? l->freq(x, y, k) : 0) +
            (r ? r->freq(a - x, b - y, k) : 0);
    }
};
```

Ordered set

Opis: Alternatywnie można użyć treapa albo trie.

Stosowanie: `s.find_by_order(k)` i `s.order_of_key(k)`.

Czas: $\mathcal{O}(\log n)$ z dużą stałą.

```
#include <bits/extc++.h>
using namespace __gnu_pbds;

template <typename T>
using ordered_set = tree<T, null_type, less<T>, rb_tree_tag,
    tree_order_statistics_node_update>;
```

Treap

Opis: Randomizowane drzewo binarne.

Czas: $\mathcal{O}(\log n)$

```
mt19937_64 rng(2137);
struct treap {
    struct node {
        int val, sz;
        uint64_t pr;
        int l = -1, r = -1;
    };
    vector<node> t;
    int make(int val) {
        int a = ssize(t);
        node& x = t.emplace_back();
        x.val = val;
        x.pr = rng();
        pull(x);
        return a;
    }
    int size(int x) { return x != -1 ? t[x].sz : 0; }
    void pull(int x) {
        if (x != -1) t[x].sz = 1 + size(t[x].l) + size(t[x].r);
    }
    int merge(int x, int y) {
        if (x == -1 || y == -1) return x != -1 ? x : y;
        int a = -1;
        if (t[x].pr > t[y].pr) {
            t[x].r = merge(t[x].r, y);
            a = x;
        } else {
            t[y].l = merge(x, t[y].l);
            a = y;
        }
        pull(a);
        return a;
    }
    pair<int, int> split(int x, int k) {
        if (x == -1) return {-1, -1};
        auto a = pair(-1, -1);
        if (k <= size(t[x].l)) {
            auto [aa, bb] = split(t[x].l, k);
            t[x].l = bb;
            a = {aa, x};
        } else {
            auto [aa, bb] = split(t[x].r, k - size(t[x].l) - 1);
            t[x].r = aa;
            a = {x, bb};
        }
        pull(a.first);
        pull(a.second);
        return a;
    }
};
```

Grafy (3)

3.1 Przepływy

Dinic

Opis: Znajduje największy przepływ. Jeżeli nie wchodzi czasowo można spróbować usunąć skalowanie krawędzi.

Czas: $\mathcal{O}(nm \log U)$

```
struct dinic {
    struct edge {
        int to, rev;
        int cap;
    };
    int n;
    vector<vector<edge>> adj;
    vector<int> q, lvl, it;
    dinic(int _n) {
        n = _n;
        adj.resize(n);
        q.reserve(n);
        lvl.resize(n);
        it.resize(n);
    }
    void add_edge(int u, int v, int cap) {
        int i = ssize(adj[u]);
        int j = ssize(adj[v]) + (u == v);
        adj[u].push_back({v, j, cap});
        adj[v].push_back({u, i, 0});
    }
    bool bfs(int s, int t, int r) {
        q.clear();
        lvl.assign(n, -1);
        lvl[s] = 0;
        q.push_back(s);
        for (int i = 0; i < ssize(q); i++) {
            int u = q[i];
            for (edge& e : adj[u]) {
                if (e.cap >= r && lvl[e.to] == -1) {
                    lvl[e.to] = lvl[u] + 1;
                    q.push_back(e.to);
                    if (e.to == t) return true;
                }
            }
        }
        return false;
    }
    ll dfs(int u, int t, ll cap) {
        if (u == t) return cap;
        ll f = 0;
        for (int& i = it[u]; i < ssize(adj[u]); i++) {
            edge& e = adj[u][i];
            if (e.cap > 0 && lvl[u] + 1 == lvl[e.to]) {
                ll add = dfs(e.to, t, min(cap - f, (ll)e.cap));
                e.cap -= add;
                adj[e.to][e.rev].cap += add;
                f += add;
            }
            if (f == cap) return f;
        }
        lvl[u] = -1;
        return f;
    }
    ll flow(int s, int t, ll cap) {
        ll f = 0;
        for (int i = 29; i >= 0; i--) {
            while (f < cap && bfs(s, t, 1 << i)) {
```

```
                it.assign(n, 0);
                f += dfs(s, t, cap - f);
            }
        }
        return f;
    }
};
```

MCMF

Opis: Znajduje największy przepływ o najmniejszym koszcie.

Stosowanie: Jeżeli są ujemne krawędzie, przed puszczeniem flow w dst trzeba policzyć najkrótsze ścieżki z s i puścić reduce(t).

Czas: $\mathcal{O}(Fm \log n)$

```
#include <bits/extc++.h>

ll INF64 = 2e18;
struct MCMF {
    struct edge {
        int to, rev;
        int cap, cost;
    };
    struct cmp {
        bool operator() (const auto& l, const auto& r) const {
            return l.second > r.second;
        }
    };
    int n;
    vector<vector<edge>> adj;
    vector<ll> dst;
    ll c = 0;
    __gnu_pbds::priority_queue<pair<ll, int>, cmp> q;
    vector<decltype(q)::point_iterator> its;
    vector<int> id;
    MCMF(int _n) {
        n = _n;
        adj.resize(n);
        id.resize(n);
    }
    void add_edge(int u, int v, int cap, int cost) {
        int i = ssize(adj[u]);
        int j = ssize(adj[v]) + (u == v);
        adj[u].push_back({v, j, cap, cost});
        adj[v].push_back({u, i, 0, -cost});
    }
    void reduce(int t) {
        for (int i = 0; i < n; i++) {
            for (edge& e : adj[i]) {
                if (dst[i] != INF64 && dst[e.to] != INF64) {
                    e.cost += dst[i] - dst[e.to];
                }
            }
        }
        c += dst[t];
    }
    bool dijkstra(int s, int t) {
        dst.assign(n, INF64);
        its.assign(n, q.end());
        dst[s] = 0;
        q.push({s, 0});
        while (!q.empty()) {
            int u = q.top().first;
            q.pop();
            for (edge& e : adj[u]) {
                if (e.cap > 0) {
                    ll d = dst[u] + e.cost;
                    if (d < dst[e.to]) {
                        dst[e.to] = d;
                        if (its[e.to] == q.end()) {
```

```
                            its[e.to] = q.push({e.to, dst[e.to]});
                    } else {
                        q.modify(its[e.to], {e.to, dst[e.to]});
                    }
                    id[e.to] = e.rev;
                }
            }
        }
        reduce(t);
        return dst[t] != INF64;
    }
    pair<ll, ll> flow(int s, int t, ll cap) {
        ll ff = 0;
        ll cc = 0;
        while (ff < cap && dijkstra(s, t)) {
            ll f = cap - ff;
            for (int i = t; i != s; i) {
                edge& e = adj[i][id[i]];
                f = min(f, (ll)adj[e.to][e.rev].cap);
                i = e.to;
            }
            for (int i = t; i != s; i) {
                edge& e = adj[i][id[i]];
                e.cap += f;
                adj[e.to][e.rev].cap -= f;
                i = e.to;
            }
            ff += f;
            cc += f * c;
        }
        return {ff, cc};
    }
};
```

3.1.1 Przepływy z wymaganiami

Szukamy przepływu $\leq F$ takiego, że $f_i \geq d_i$ dla każdej krawędzi. Tworzymy nowe źródło s' i ujście t' . Następnie dodajemy krawędzie

- (u_i, t', d_i) , (s', v_i, d_i) , $(u_i, v_i, c_i - d_i)$ zamiast (u_i, v_i, c_i, d_i)
- (t, s, F)

Przepływ spełnia wymagania jeżeli maksymalnie wypełnia wszystkie krawędzie s' .

3.2 Grafy dwudzielne

Matching

Opis: Dinic uproszczony do szukania największego skojarzenia.

Czas: $\mathcal{O}(m\sqrt{n})$

```
struct matching {
    int n, m;
    vector<vector<int>> adj;
    vector<int> pb, pa;
    vector<int> lvl, it;
    matching(int _n, int _m) {
        n = _n;
        m = _m;
        adj.resize(n);
        pb.resize(n, -1);
        pa.resize(m, -1);
        it.resize(n);
    }
};
```

```
void add_edge(int u, int v) { adj[u].push_back(v); }
bool bfs() {
    bool res = false;
    lvl.assign(n, -1);
    queue<int> q;
    for (int i = 0; i < n; i++) {
        if (pb[i] == -1) {
            q.push(i);
            lvl[i] = 0;
        }
    }
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int j : adj[u]) {
            if (pa[j] == -1) {
                res = true;
            } else if (lvl[pa[j]] == -1) {
                lvl[pa[j]] = lvl[u] + 1;
                q.push(pa[j]);
            }
        }
    }
    return res;
}
bool dfs(int u) {
    for (auto& i = it[u]; i < ssize(adj[u]); i++) {
        int v = adj[u][i];
        if (pa[v] == -1 ||
            (lvl[pa[v]] == lvl[u] + 1 && dfs(pa[v]))) {
            pb[u] = v;
            pa[v] = u;
            return true;
        }
    }
    return false;
}
int match() {
    int ans = 0;
    while (bfs()) {
        it.assign(n, 0);
        for (int i = 0; i < n; i++) {
            if (pb[i] == -1 && dfs(i)) ans++;
        }
    }
    return ans;
}
};
```

3.2.1 Rozszerzone twierdzenie Königa

W grafie dwudzielnym zachodzi

- $n_k = p_w$
- $n_k + p_k = n$
- $p_w + n_w = n$

oraz

- p_w to zbiór wierzchołków na brzegu min-cut
- n_w to dopełnienie p_w
- p_k to n_k z dodanymi pojedynczymi krawędziami każdego nieskojarzonego wierzchołka

3.3 Grafy skierowane

SCC

Opis: Znajduje silne spójne składowe w kolejności topologicznej.
Czas: $\mathcal{O}(n + m)$

```
struct SCC {
    int n, cnt = 0;
    vector<vector<int>> adj;
    vector<int> p, low, in;
    stack<int> st;
    int tour = 0;
    SCC(int _n) {
        n = _n;
        adj.resize(n);
        p.resize(n, -1);
        low.resize(n);
        in.resize(n, -1);
    }
    void add_edge(int u, int v) { adj[u].push_back(v); }
    void dfs(int u) {
        low[u] = in[u] = tour++;
        st.push(u);
        for (int v : adj[u]) {
            if (in[v] == -1) {
                dfs(v);
                low[u] = min(low[u], low[v]);
            } else {
                low[u] = min(low[u], in[v]);
            }
        }
        if (low[u] == in[u]) {
            int v = -1;
            do {
                v = st.top();
                st.pop();
                in[v] = n;
                p[v] = cnt;
            } while (v != u);
            cnt++;
        }
    }
    void build() {
        for (int i = 0; i < n; ++i) {
            if (in[i] == -1) dfs(i);
        }
        for (int i = 0; i < n; i++) p[i] = cnt - 1 - p[i];
    }
    vector<vector<int>> groups() {
        vector<vector<int>> res(cnt);
        for (int i = 0; i < n; i++) res[p[i]].push_back(i);
        return res;
    }
};
```

Matma (4)

4.1 Wielomiany

FFT

Opis: Mnoży dwa wielomiany o sumarycznej długości 2^{23} modulo 998244353.
Czas: $\mathcal{O}((n + m) \log(n + m))$

```
struct FFT {
    int N;
```

```
vector<int> rev;
vector<mint> w;
FFT(int k) {
    N = 1 << k;
    rev.resize(N);
    for (int i = 1; i < N; i++) {
        rev[i] = (rev[i >> 1] >> 1) | ((i & 1) << (k - 1));
    }
#warning MOD = 998244353
    mint W = mint(3).pow(119 * ((1 << 23) / N));
    w.resize(N);
    mint ww = 1;
    for (int i = 0; i < N / 2; i++) {
        w[i + N / 2] = ww;
        ww *= W;
    }
    for (int i = N / 2 - 1; i > 0; i--) w[i] = w[2 * i];
}
void fft(vector<mint>& a) {
    int n = ssize(a);
    int s = __lg(N / n);
    for (int i = 0; i < n; i++) {
        int r = rev[i] >> s;
        if (i < r) swap(a[i], a[r]);
    }
    for (int i = 1; i < n; i *= 2) {
        for (int j = 0; j < n; j += 2 * i) {
            for (int k = 0; k < i; k++) {
                mint z = w[i + k] * a[j + k + i];
                a[j + k + i] = a[j + k] - z;
                a[j + k] += z;
            }
        }
    }
}
vector<mint> conv(vector<mint> a, vector<mint> b) {
    int n = ssize(a);
    int m = ssize(b);
    int k = 1;
    while (k < n + m - 1) k *= 2;
    a.resize(k);
    b.resize(k);
    fft(a);
    fft(b);
    for (int i = 0; i < k; i++) a[i] *= b[i];
    fft(a);
    reverse(a.begin() + 1, a.end());
    a.resize(n + m - 1);
    mint inv = mint(k).inv();
    for (int i = 0; i < n + m - 1; i++) a[i] *= inv;
    return a;
}
};
```

4.2 Mnożniki Lagrange’a

Jeżeli optymalizujemy $f(x_1, \dots, x_n)$ przy ograniczeniach typu $g_k(x_1, \dots, x_n) = 0$ to x_1, \dots, x_n jest ekstremum lokalnym tylko jeżeli gradient $\nabla f(x_1, \dots, x_n)$ jest kombinacją liniową gradientów $\nabla g_k(x_1, \dots, x_n)$.

Teksty (5)

KMP

Opis: Znajduje funkcje prefiksową. Można z niej skonstruować automat w czasie $\mathcal{O}(nA)$.

Czas: $\mathcal{O}(n)$

```
vector<int> kmp(const string& s) {
    int n = ssize(s);
    vector<int> p(n);
    for (int i = 1; i < n; i++) {
        int j = p[i - 1];
        while (j > 0 && s[i] != s[j]) j = p[j - 1];
        if (s[i] == s[j]) j++;
        p[i] = j;
    }
    return p;
}
```

Manacher

Opis: Znajduje najdłuższy promień palindromiczny w każdym środku.
Stosowanie: $p[2 * i]$ – środek w i , $p[2 * i + 1]$ – środek między i a $i + 1$.

Czas: $\mathcal{O}(n)$

```
vector<int> manacher(const string& s) {
    int n = ssize(s);
    string t(2 * n, '.');
    for (int i = 0; i < n; i++) {
        t[2 * i] = s[i];
        t[2 * i + 1] = '#';
    }
    vector<int> p(2 * n - 1);
    for (int i = 0, l = -1, r = -1; i < 2 * n - 1; i++) {
        if (i <= r) p[i] = min(r - i + 1, p[l + r - i]);
        while (p[i] < min(i + 1, 2 * n - 1 - i) &&
            t[i - p[i]] == t[i + p[i]]) {
            p[i]++;
        }
        if (i + p[i] - 1 > r) {
            l = i - p[i] + 1;
            r = i + p[i] - 1;
        }
    }
    for (int i = 0; i < 2 * n - 1; i++) {
        if (t[i - p[i] + 1] == '#') p[i]--;
        p[i] = (p[i] + (1 - i % 2)) / 2;
    }
    return p;
}
```

Tablica sufiksowa

Opis: Sortuje leksykograficznie wszystkie sufiksy słowa. Tablic rnk można użyć do porównywania leksykograficznie podstłów.

Czas: $\mathcal{O}(n \log n)$

```
vector<int> suffix_array(const string& s) {
    int n = ssize(s);
    vector<int> p(n), cnt(26);
    for (int i = 0; i < n; i++) cnt[s[i] - 'a']++;
    for (int i = 1; i < 26; i++) cnt[i] += cnt[i - 1];
    for (int i = 0; i < n; i++) p[--cnt[s[i] - 'a']] = i;
    vector<int> rnk(n);
    for (int i = 1; i < n; i++) {
        rnk[p[i]] = s[p[i]] == s[p[i - 1]] ? rnk[p[i - 1]] : i;
    }
    cnt.resize(n);
    vector<int> np(n), nrnk(n);
```

```
    for (int len = 1; len < n; len *= 2) {
        iota(cnt.begin(), cnt.end(), 0);
        for (int i = n - len; i < n; i++) {
            np[cnt[rnk[i]]++] = i;
        }
        for (int i = 0; i < n; i++) {
            if (p[i] - len >= 0) {
                np[cnt[rnk[p[i] - len]]++] = p[i] - len;
            }
        }
        nrnk[np[0]] = 0;
        for (int i = 1; i < n; i++) {
            int a = np[i - 1];
            int b = np[i];
            if (max(a, b) + len < n && rnk[a] == rnk[b] &&
                rnk[a + len] == rnk[b + len]) {
                nrnk[b] = nrnk[a];
            } else {
                nrnk[b] = i;
            }
        }
        swap(p, np);
        swap(rnk, nrnk);
    }
    return p;
};

vector<int> build_lcp(const string& s, const vector<int>& sa) {
    int n = ssize(s);
    vector<int> pos(n);
    for (int i = 0; i < n; i++) pos[sa[i]] = i;
    vector<int> lcp(n - 1);
    int k = 0;
    for (int i = 0; i < n; i++) {
        if (pos[i] == 0) continue;
        while (i + k < n && s[i + k] == s[sa[pos[i] - 1] + k]) {
            k++;
        }
        lcp[pos[i] - 1] = k;
        k = max(0, k - 1);
    }
    return lcp;
}
```

Z

Opis: Znajduje funkcję Z.

Czas: $\mathcal{O}(n)$

```
vector<int> z(const string& s) {
    int n = ssize(s);
    vector<int> f(n);
    for (int i = 1, l = 0, r = 0; i < n; i++) {
        if (i <= r) f[i] = min(r - i + 1, f[i - 1]);
        while (f[i] < n - i && s[i + f[i]] == s[f[i]]) f[i]++;
        if (i + f[i] - 1 > r) {
            l = i;
            r = i + f[i] - 1;
        }
    }
    return f;
}
```