

Columbia University: CU Later Team Reference Document

Kevin Yang, Innokentiy Kaurov, Eric Yuang Shao

May 21th 2024

Contents

Templates	2
Ken's template	2
Kevin's template	2
Kevin's Template Extended	2
Geometry	2
Point basics	2
Line basics	2
Line and segment intersections	2
Distances from a point to line and segment	3
Polygon area	3
Convex hull	3
Point location in a convex polygon	3
Point location in a simple polygon	3
Minkowski Sum	3
Half-plane intersection	4
Strings	4
Manacher's algorithm	4
Aho-Corasick Trie	5
Flows	5
$O(N^2M)$, on unit networks $O(N^{1/2}M)$	5
MCMF – maximize flow, then minimize its cost. $O(mn + Fm \log n)$	6
Graphs	7
Kuhn's algorithm for bipartite matching . . .	7
Hungarian algorithm for Assignment Problem	7
Dijkstra's Algorithm	7
Eulerian Cycle DFS	7
SCC and 2-SAT	7
Finding Bridges	8
Virtual Tree	8
HLD on Edges DFS	8
Centroid Decomposition	8

Math	9
Binary exponentiation	9
Matrix Exponentiation: $O(n^3 \log b)$	9
Extended Euclidean Algorithm	9
Linear Sieve	9
Gaussian Elimination	9
is_prime	10
Berlekamp-Massey	10
Calculating k-th term of a linear recurrence .	10
Partition Function	11
NTT	11
FFT	11
MIT's FFT/NTT, Polynomial mod/log/exp Template	11
Simplex method for linear programs	14
Data Structures	14
Fenwick Tree	14
Lazy Propagation SegTree	14
Sparse Table	15
Suffix Array and LCP array	15
Aho Corasick Trie	16
Convex Hull Trick	16
Li-Chao Segment Tree	16
Persistent Segment Tree	17
Miscellaneous	17
Ordered Set	17
Measuring Execution Time	17
Setting Fixed D.P. Precision	17
Common Bugs and General Advice	17
Dynamic Programming	17
Sum over Subset DP	17
Divide and Conquer DP	18

Templates

Ken's template

```
1 #include <bits/stdc++.h>
2 using namespace std;
3 #define all(v) (v).begin(), (v).end()
4 typedef long long ll;
5 typedef long double ld;
6 #define pb push_back
7 #define sz(x) (int)(x).size()
8 #define fi first
9 #define se second
10 #define endl '\n'
```

Kevin's template

```
1 // paste Kaurov's Template, minus last line
2 typedef vector<int> vi;
3 typedef vector<ll> vll;
4 typedef pair<int, int> pii;
5 typedef pair<ll, ll> pll;
6 const char nl = '\n';
7 #define forn(i, n) for (int i = 0; i < int(n); i++)
8 ll k, n, m, u, v, w, x, y, z;
9 string s;
10
11 bool multiTest = 1;
12 void solve(int tt){
13 }
14
15 int main(){
16     ios::sync_with_stdio(0);cin.tie(0);cout.tie(0);
17     cout<<fixed<< setprecision(14);
18
19     int t = 1;
20     if (multiTest) cin >> t;
21     forn(ii, t) solve(ii);
22 }
```

Kevin's Template Extended

- to type after the start of the contest

```
1 typedef pair<double, double> pdd;
2 const ld PI = acos(-1);
3 const ll mod7 = 1e9 + 7;
4 const ll mod9 = 998244353;
5 const ll INF = 2*1024*1024*1023;
6 #pragma GCC target("avx2,bmi,bmi2,lzcnt,popcnt")
7 #include <ext/pb_ds/assoc_container.hpp>
8 #include <ext/pb_ds/tree_policy.hpp>
9 using namespace __gnu_pbds;
10 template<class T> using ordered_set = tree<T, null_type,
11     < less<T>, rb_tree_tag,
12     < tree_order_statistics_node_update>;
13 vi d4x = {1, 0, -1, 0};
```

```
12 vi d4y = {0, 1, 0, -1};
13 vi d8x = {1, 0, -1, 0, 1, 1, -1, -1};
14 vi d8y = {0, 1, 0, -1, 1, -1, 1, -1};
15 mt19937
16     < rng(chrono::steady_clock::now().time_since_epoch().count())
```

Geometry

Point basics

```
1 const ld EPS = 1e-9;
2
3 struct point{
4     ld x, y;
5     point() : x(0), y(0) {}
6     point(ld x_, ld y_) : x(x_), y(y_) {}
7
8     point operator+ (point rhs) const{
9         return point(x + rhs.x, y + rhs.y);
10     }
11     point operator- (point rhs) const{
12         return point(x - rhs.x, y - rhs.y);
13     }
14     point operator* (ld rhs) const{
15         return point(x * rhs, y * rhs);
16     }
17     point operator/ (ld rhs) const{
18         return point(x / rhs, y / rhs);
19     }
20     point ort() const{
21         return point(-y, x);
22     }
23     ld abs2() const{
24         return x * x + y * y;
25     }
26     ld len() const{
27         return sqrt1(abs2());
28     }
29     point unit() const{
30         return point(x, y) / len();
31     }
32     point rotate(ld a) const{
33         return point(x * cosl(a) - y * sinl(a), x * sinl(a)
34             < + y * cosl(a));
35     }
36     friend ostream& operator<<(ostream& os, point p){
37         return os << "(" << p.x << ", " << p.y << ")";
38     }
39     bool operator< (point rhs) const{
40         return make_pair(x, y) < make_pair(rhs.x, rhs.y);
41     }
42     bool operator==(point rhs) const{
43         return abs(x - rhs.x) < EPS && abs(y - rhs.y) < EPS;
44     }
45 };
```

```
46
47 ld sq(ld a){
48     return a * a;
49 }
50 ld smul(point a, point b){
51     return a.x * b.x + a.y * b.y;
52 }
53 ld vmul(point a, point b){
54     return a.x * b.y - a.y * b.x;
55 }
56 ld dist(point a, point b){
57     return (a - b).len();
58 }
59 bool acw(point a, point b){
60     return vmul(a, b) > -EPS;
61 }
62 bool cw(point a, point b){
63     return vmul(a, b) < EPS;
64 }
65 int sgn(ld x){
66     return (x > EPS) - (x < EPS);
67 }
```

Line basics

```
1 struct line{
2     ld a, b, c;
3     line() : a(0), b(0), c(0) {}
4     line(ld a_, ld b_, ld c_) : a(a_), b(b_), c(c_) {}
5     line(point p1, point p2){
6         a = p1.y - p2.y;
7         b = p2.x - p1.x;
8         c = -a * p1.x - b * p1.y;
9     }
10 };
11
12 ld det(ld a11, ld a12, ld a21, ld a22){
13     return a11 * a22 - a12 * a21;
14 }
15 bool parallel(line l1, line l2){
16     return abs(vmul(point(l1.a, l1.b), point(l2.a, l2.b)))
17     < EPS;
18 }
19 bool operator==(line l1, line l2){
20     return parallel(l1, l2) &&
21     abs(det(l1.b, l1.c, l2.b, l2.c)) < EPS &&
22     abs(det(l1.a, l1.c, l2.a, l2.c)) < EPS;
23 }
```

Line and segment intersections

```
1 // {p, 0} - unique intersection, {p, 1} - infinite, {p,
2     < 2} - none
3 pair<point, int> line_inter(line l1, line l2){
4     if (parallel(l1, l2)){
```

```

4     return {point(), l1 == 12? 1 : 2};
5 }
6 return {point(
7     det(-l1.c, l1.b, -l2.c, l2.b) / det(l1.a, l1.b,
↪ 12.a, l2.b),
8     det(l1.a, -l1.c, l2.a, -l2.c) / det(l1.a, l1.b,
↪ 12.a, l2.b)
9     ), 0};
10 }
11
12 // Checks if p lies on ab
13 bool is_on_seg(point p, point a, point b){
14     return abs(vmul(p - a, p - b)) < EPS && smul(p - a, p
↪ - b) < EPS;
15 }
16
17 /*
18 If a unique intersection point between the line segments2
↪ going from a to b and from c to d exists then it is
↪ returned.
19 If no intersection point exists an empty vector is
↪ returned.
20 If infinitely many exist a vector with 2 elements is
↪ returned, containing the endpoints of the common
↪ line segment.
21 */
22 vector<point> segment_inter(point a, point b, point c,
↪ point d) {
23     auto oa = vmul(d - c, a - c), ob = vmul(d - c, b - c);
↪ oc = vmul(b - a, c - a), od = vmul(b - a, d - a);
24     if (sgn(oa) * sgn(ob) < 0 && sgn(oc) * sgn(od) < 0)
↪ return {(a * ob - b * oa) / (ob - oa)};
25     set<point> s;
26     if (is_on_seg(a, c, d)) s.insert(a);
27     if (is_on_seg(b, c, d)) s.insert(b);
28     if (is_on_seg(c, a, b)) s.insert(c);
29     if (is_on_seg(d, a, b)) s.insert(d);
30     return {all(s)};
31 }
32

```

Distances from a point to line and segment

```

1 // Distance from p to line ab
2 ld line_dist(point p, point a, point b){
3     return vmul(b - a, p - a) / (b - a).len();
4 }
5
6 // Distance from p to segment ab
7 ld segment_dist(point p, point a, point b){
8     if (a == b) return (p - a).len();
9     auto d = (a - b).abs2(), t = min(d, max((ld)0, smul(p
↪ - a, b - a)));
10     return ((p - a) * d - (b - a) * t).len() / d;
11 }

```

Polygon area

```

1 ld area(vector<point> pts){
2     int n = sz(pts);
3     ld ans = 0;
4     for (int i = 0; i < n; i++){
5         ans += vmul(pts[i], pts[(i + 1) % n]);
6     }
7     return abs(ans) / 2;
8 }

```

Convex hull

- Complexity: $O(n \log n)$.

```

vector<point> convex_hull(vector<point> pts){
    sort(all(pts));
    pts.erase(unique(all(pts)), pts.end());
    vector<point> up, down;
    for (auto p : pts){
        while (sz(up) > 1 && acw(up.end()[-1] -
↪ up.end()[-2], p - up.end()[-2])) up.pop_back();
        while (sz(down) > 1 && cw(down.end()[-1] -
↪ down.end()[-2], p - down.end()[-2]))
↪ down.pop_back();
        up.pb(p), down.pb(p);
    }
    for (int i = sz(up) - 2; i >= 1; i--) down.pb(up[i]);
    return down;
}

```

Point location in a convex polygon

- Complexity: $O(n)$ precalculation and $O(\log n)$ query.

```

1 void prep_convex_poly(vector<point>& pts){
2     rotate(pts.begin(), min_element(all(pts)), pts.end());
3 }
4
5 // 0 - Outside, 1 - Exclusively Inside, 2 - On the
↪ Border
6 int in_convex_poly(point p, vector<point>& pts){
7     int n = sz(pts);
8     if (!n) return 0;
9     if (n <= 2) return is_on_seg(p, pts[0], pts.back());
10    int l = 1, r = n - 1;
11    while (r - l > 1){
12        int mid = (l + r) / 2;
13        if (acw(pts[mid] - pts[0], p - pts[0])) l = mid;
14        else r = mid;
15    }
16    if (!in_triangle(p, pts[0], pts[l], pts[l + 1]))
↪ return 0;
17    if (is_on_seg(p, pts[l], pts[l + 1])) ||

```

```

18    is_on_seg(p, pts[0], pts.back()) ||
19    is_on_seg(p, pts[0], pts[l])
20    ) return 2;
21    return 1;
22 }

```

Point location in a simple polygon

- Complexity: $O(n)$.

```

1 // 0 - Outside, 1 - Exclusively Inside, 2 - On the
↪ Border
2 int in_simple_poly(point p, vector<point>& pts){
3     int n = sz(pts);
4     bool res = 0;
5     for (int i = 0; i < n; i++){
6         auto a = pts[i], b = pts[(i + 1) % n];
7         if (is_on_seg(p, a, b)) return 2;
8         if (((a.y > p.y) - (b.y > p.y)) * vmul(b - p, a - p)
↪ > EPS){
9             res ^= 1;
10        }
11    }
12    return res;
13 }

```

Minkowski Sum

- For two convex polygons P and Q , returns the set of points $(p + q)$, where $p \in P, q \in Q$.
- This set is also a convex polygon.
- Complexity: $O(n)$.

```

void minkowski_rotate(vector<point>& P){
    int pos = 0;
    for (int i = 1; i < sz(P); i++){
        if (abs(P[i].y - P[pos].y) <= EPS){
            if (P[i].x < P[pos].x) pos = i;
        }
        else if (P[i].y < P[pos].y) pos = i;
    }
    rotate(P.begin(), P.begin() + pos, P.end());
}
// P and Q are strictly convex, points given in
↪ counterclockwise order.
vector<point> minkowski_sum(vector<point> P,
↪ vector<point> Q){
    minkowski_rotate(P);
    minkowski_rotate(Q);
    P.pb(P[0]);
    Q.pb(Q[0]);
}

```

```

17 vector<point> ans;
18 int i = 0, j = 0;
19 while (i < sz(P) - 1 || j < sz(Q) - 1){
20     ans.pb(P[i] + Q[j]);
21     ld curmul;
22     if (i == sz(P) - 1) curmul = -1;
23     else if (j == sz(Q) - 1) curmul = +1;
24     else curmul = vmul(P[i + 1] - P[i], Q[j + 1] -
    ↪ Q[j]);
25     if (abs(curmul) < EPS || curmul > 0) i++;
26     if (abs(curmul) < EPS || curmul < 0) j++;
27 }
28 return ans;
29 }

```

Half-plane intersection

- Given N half-plane conditions in the form of a ray, computes the vertices of their intersection polygon.
- Complexity: $O(N \log N)$.
- A ray is defined by a point p and direction vector dp . The half-plane is to the **left** of the direction vector.

```

1 // Extra functions needed: point operations, smul, vmul
2 const ld EPS = 1e-9;
3
4 int sgn(ld a){
5     return (a > EPS) - (a < -EPS);
6 }
7 int half(point p){
8     return p.y != 0? sgn(p.y) : -sgn(p.x);
9 }
10 bool angle_comp(point a, point b){
11     int A = half(a), B = half(b);
12     return A == B? vmul(a, b) > 0 : A < B;
13 }
14 struct ray{
15     point p, dp; // origin, direction
16     ray(point p_, point dp_){
17         p = p_, dp = dp_;
18     }
19     point isect(ray l){
20         return p + dp * (vmul(l.dp, l.p - p) / vmul(l.dp,
    ↪ dp));
21     }
22     bool operator<(ray l){
23         return angle_comp(dp, l.dp);
24     }
25 };
26 vector<point> half_plane_isect(vector<ray> rays, ld DX =
    ↪ 1e9, ld DY = 1e9){
27     // constrain the area to [0, DX] x [0, DY]
28     rays.pb({point(0, 0), point(1, 0)});
29     rays.pb({point(DX, 0), point(0, 1)});

```

```

30     rays.pb({point(DX, DY), point(-1, 0)});
31     rays.pb({point(0, DY), point(0, -1)});
32     sort(all(rays));
33     {
34         vector<ray> nrays;
35         for (auto t : rays){
36             if (nrays.empty() || vmul(nrays.back().dp, t.dp)
    ↪ EPS){
37                 nrays.pb(t);
38                 continue;
39             }
40             if (vmul(t.dp, t.p - nrays.back().p) > 0)
    ↪ nrays.back() = t;
41         }
42         swap(rays, nrays);
43     }
44     auto bad = [&] (ray a, ray b, ray c){
45         point p1 = a.isect(b), p2 = b.isect(c);
46         if (smul(p2 - p1, b.dp) <= EPS){
47             if (vmul(a.dp, c.dp) <= 0) return 2;
48             return 1;
49         }
50         return 0;
51     };
52     #define reduce(t) \
53         while (sz(poly) > 1){ \
54             int b = bad(poly[sz(poly) - 2], poly.back()
    ↪ t); \
55             if (b == 2) return {}; \
56             if (b == 1) poly.pop_back(); \
57             else break; \
58         }
59     deque<ray> poly;
60     for (auto t : rays){
61         reduce(t);
62         poly.pb(t);
63     }
64     for (; poly.pop_front()){
65         reduce(poly[0]);
66         if (!bad(poly.back(), poly[0], poly[1])) break;
67     }
68     assert(sz(poly) >= 3); // expect nonzero area
69     vector<point> poly_points;
70     for (int i = 0; i < sz(poly); i++){
71         poly_points.pb(poly[i].isect(poly[(i + 1) %
    ↪ sz(poly)]));
72     }
73     return poly_points;
74 }

```

Strings

```

vector<int> prefix_function(string s){
2     int n = sz(s);
3     vector<int> pi(n);
4     for (int i = 1; i < n; i++){
5         int k = pi[i - 1];

```

```

6         while (k > 0 && s[i] != s[k]){
7             k = pi[k - 1];
8         }
9         pi[i] = k + (s[i] == s[k]);
10    }
11    return pi;
12}
13vector<int> kmp(string s, string k){
14    string st = k + "#" + s;
15    vector<int> res;
16    auto pi = prefix_function(st);
17    for (int i = 0; i < sz(st); i++){
18        if (pi[i] == sz(k)){
19            res.pb(i - 2 * sz(k));
20        }
21    }
22    return res;
23}
24vector<int> z_function(string s){
25    int n = sz(s);
26    vector<int> z(n);
27    int l = 0, r = 0;
28    for (int i = 1; i < n; i++){
29        if (r >= i) z[i] = min(z[i - l], r - i + 1);
30        while (i + z[i] < n && s[z[i]] == s[i + z[i]]){
31            z[i]++;
32        }
33        if (i + z[i] - 1 > r){
34            l = i, r = i + z[i] - 1;
35        }
36    }
37    return z;
38}

```

Manacher's algorithm

```

1 /*
2 Finds longest palindromes centered at each index
3 even[i] = d --> [i - d, i + d - 1] is a max-palindrome
4 odd[i] = d --> [i - d, i + d] is a max-palindrome
5 */
6 pair<vector<int>, vector<int>> manacher(string s) {
7     vector<char> t{'^', '#'};
8     for (char c : s) t.push_back(c), t.push_back('#');
9     t.push_back('$');
10    int n = t.size(), r = 0, c = 0;
11    vector<int> p(n, 0);
12    for (int i = 1; i < n - 1; i++) {
13        if (i < r + c) p[i] = min(p[2 * c - i], r + c - i);
14        while (t[i + p[i] + 1] == t[i - p[i] - 1]) p[i]++;
15        if (i + p[i] > r + c) r = p[i], c = i;
16    }
17    vector<int> even(sz(s)), odd(sz(s));
18    for (int i = 0; i < sz(s); i++){
19        even[i] = p[2 * i + 1] / 2, odd[i] = p[2 * i + 2] /
    ↪ 2;

```

```

20     }
21     return {even, odd};
22 }

```

Aho-Corasick Trie

```

1  const int S = 26;
2
3  // Function converting char to int.
4  int ctoi(char c){
5      return c - 'a';
6  }
7
8  // To add terminal links, use DFS
9  struct Node{
10     vector<int> nxt;
11     int link;
12     bool terminal;
13
14     Node() {
15         nxt.assign(S, -1), link = 0, terminal = 0;
16     }
17 };
18
19 vector<Node> trie(1);
20
21 // add_string returns the terminal vertex.
22 int add_string(string& s){
23     int v = 0;
24     for (auto c : s){
25         int cur = ctoi(c);
26         if (trie[v].nxt[cur] == -1){
27             trie[v].nxt[cur] = sz(trie);
28             trie.emplace_back();
29         }
30         v = trie[v].nxt[cur];
31     }
32     trie[v].terminal = 1;
33     return v;
34 }
35
36 /*
37 Suffix links are compressed.
38 This means that:
39     If vertex v has a child by letter x, then:
40         trie[v].nxt[x] points to that child.
41     If vertex v doesn't have such child, then:
42         trie[v].nxt[x] points to the suffix link of that
43     ↪ child
44         if we would actually have it.
45 */
46 void add_links(){
47     queue<int> q;
48     q.push(0);
49     while (!q.empty()){
50         auto v = q.front();
51         int u = trie[v].link;

```

```

52     for (int i = 0; i < S; i++){
53         int& ch = trie[v].nxt[i];
54         if (ch == -1){
55             ch = v? trie[u].nxt[i] : 0;
56         }
57         else{
58             trie[ch].link = v? trie[u].nxt[i] : 0;
59             q.push(ch);
60         }
61     }
62 }
63 }
64
65 bool is_terminal(int v){
66     return trie[v].terminal;
67 }
68
69 int get_link(int v){
70     return trie[v].link;
71 }
72
73 int go(int v, char c){
74     return trie[v].nxt[ctoi(c)];
75 }

```

Flows

$O(N^2M)$, on unit networks $O(N^{1/2}M)$

```

1  struct FlowEdge {
2      int from, to;
3      ll cap, flow = 0;
4      FlowEdge(int u, int v, ll cap) : from(u), to(v),
5      ↪ cap(cap) {}
6  };
7
8  struct Dinic {
9      const ll flow_inf = 1e18;
10     vector<FlowEdge> edges;
11     vector<vector<int>> adj;
12     int n, m = 0;
13     int s, t;
14     vector<int> level, ptr;
15     vector<bool> used;
16     queue<int> q;
17     Dinic(int n, int s, int t) : n(n), s(s), t(t) {
18         adj.resize(n);
19         level.resize(n);
20         ptr.resize(n);
21     }
22     void add_edge(int u, int v, ll cap) {
23         edges.emplace_back(u, v, cap);
24         edges.emplace_back(v, u, 0);
25         adj[u].push_back(m);
26         adj[v].push_back(m + 1);
27         m += 2;
28     }

```

```

27 bool bfs() {
28     while (!q.empty()) {
29         int v = q.front();
30         q.pop();
31         for (int id : adj[v]) {
32             if (edges[id].cap - edges[id].flow < 1)
33                 continue;
34             if (level[edges[id].to] != -1)
35                 continue;
36             level[edges[id].to] = level[v] + 1;
37             q.push(edges[id].to);
38         }
39     }
40     return level[t] != -1;
41 }
42 ll dfs(int v, ll pushed) {
43     if (pushed == 0)
44         return 0;
45     if (v == t)
46         return pushed;
47     for (int& cid = ptr[v]; cid <
48     ↪ (int)adj[v].size(); cid++) {
49         int id = adj[v][cid];
50         int u = edges[id].to;
51         if (level[v] + 1 != level[u] ||
52     ↪ edges[id].cap - edges[id].flow < 1)
53             continue;
54         ll tr = dfs(u, min(pushed, edges[id].cap -
55     ↪ edges[id].flow));
56         if (tr == 0)
57             continue;
58         edges[id].flow += tr;
59         edges[id ^ 1].flow -= tr;
60         return tr;
61     }
62     return 0;
63 }
64 ll flow() {
65     ll f = 0;
66     while (true) {
67         fill(level.begin(), level.end(), -1);
68         level[s] = 0;
69         q.push(s);
70         if (!bfs())
71             break;
72         fill(ptr.begin(), ptr.end(), 0);
73         while (ll pushed = dfs(s, flow_inf)) {
74             f += pushed;
75         }
76     }
77     return f;
78 }
79
80 void cut_dfs(int v){
81     used[v] = 1;
82     for (auto i : adj[v]){

```

```

80     if (edges[i].flow < edges[i].cap &&
    ↪ !used[edges[i].to]){
81         cut_dfs(edges[i].to);
82     }
83 }
84 }
85
86 // Assumes that max flow is already calculated
87 // true -> vertex is in S, false -> vertex is in T
88 vector<bool> min_cut(){
89     used = vector<bool>(n);
90     cut_dfs(s);
91     return used;
92 }
93 };
94 // To recover flow through original edges: iterate over
    ↪ even indices in edges.

```

MCMF – maximize flow, then minimize its cost. $O(mn + Fm \log n)$.

```

1  #include <ext/pb_ds/priority_queue.hpp>
2  template <typename T, typename C>
3  class MCMF {
4  public:
5      static constexpr T eps = (T) 1e-9;
6
7      struct edge {
8          int from;
9          int to;
10         T c;
11         T f;
12         C cost;
13     };
14
15     int n;
16     vector<vector<int>> g;
17     vector<edge> edges;
18     vector<C> d;
19     vector<C> pot;
20     __gnu_pbds::priority_queue<pair<C, int>> q;
21     vector<typename decltype(q)::point_iterator> its;
22     vector<int> pe;
23     const C INF_C = numeric_limits<C>::max() / 2;
24
25     explicit MCMF(int n_) : n(n_), g(n), d(n), pot(n, 0),
    ↪ its(n), pe(n) {}
26
27     int add(int from, int to, T forward_cap, C edge_cost,
    ↪ T backward_cap = 0) {
28         assert(0 <= from && from < n && 0 <= to && to < n);
29         assert(forward_cap >= 0 && backward_cap >= 0);
30         int id = static_cast<int>(edges.size());
31         g[from].push_back(id);
32         edges.push_back({from, to, forward_cap, 0,
    ↪ edge_cost});
33         g[to].push_back(id + 1);

```

```

34         edges.push_back({to, from, backward_cap, 0,
    ↪ -edge_cost});
35         return id;
36     }
37
38     void expath(int st) {
39         fill(d.begin(), d.end(), INF_C);
40         q.clear();
41         fill(its.begin(), its.end(), q.end());
42         its[st] = q.push({pot[st], st});
43         d[st] = 0;
44         while (!q.empty()) {
45             int i = q.top().second;
46             q.pop();
47             its[i] = q.end();
48             for (int id : g[i]) {
49                 const edge &e = edges[id];
50                 int j = e.to;
51                 if (e.c - e.f > eps && d[i] + e.cost < d[j]) {
52                     d[j] = d[i] + e.cost;
53                     pe[j] = id;
54                     if (its[j] == q.end()) {
55                         its[j] = q.push({pot[j] - d[j], j});
56                     } else {
57                         q.modify(its[j], {pot[j] - d[j], j});
58                     }
59                 }
60             }
61         }
62         swap(d, pot);
63     }
64
65     pair<T, C> max_flow(int st, int fin) {
66         T flow = 0;
67         C cost = 0;
68         bool ok = true;
69         for (auto& e : edges) {
70             if (e.c - e.f > eps && e.cost + pot[e.from] -
    ↪ pot[e.to] < 0) {
71                 ok = false;
72                 break;
73             }
74         }
75         if (ok) {
76             expath(st);
77         } else {
78             vector<int> deg(n, 0);
79             for (int i = 0; i < n; i++) {
80                 for (int eid : g[i]) {
81                     auto& e = edges[eid];
82                     if (e.c - e.f > eps) {
83                         deg[e.to] += 1;
84                     }
85                 }
86             }
87             vector<int> que;
88             for (int i = 0; i < n; i++) {
89                 if (deg[i] == 0) {

```

```

90                 que.push_back(i);
91             }
92         }
93         for (int b = 0; b < (int) que.size(); b++) {
94             for (int eid : g[que[b]]) {
95                 auto& e = edges[eid];
96                 if (e.c - e.f > eps) {
97                     deg[e.to] -= 1;
98                     if (deg[e.to] == 0) {
99                         que.push_back(e.to);
100                     }
101                 }
102             }
103         }
104         fill(pot.begin(), pot.end(), INF_C);
105         pot[st] = 0;
106         if (static_cast<int>(que.size()) == n) {
107             for (int v : que) {
108                 if (pot[v] < INF_C) {
109                     for (int eid : g[v]) {
110                         auto& e = edges[eid];
111                         if (e.c - e.f > eps) {
112                             if (pot[v] + e.cost < pot[e.to]) {
113                                 pot[e.to] = pot[v] + e.cost;
114                                 pe[e.to] = eid;
115                             }
116                         }
117                     }
118                 }
119             }
120         } else {
121             que.assign(1, st);
122             vector<bool> in_queue(n, false);
123             in_queue[st] = true;
124             for (int b = 0; b < (int) que.size(); b++) {
125                 int i = que[b];
126                 in_queue[i] = false;
127                 for (int id : g[i]) {
128                     const edge &e = edges[id];
129                     if (e.c - e.f > eps && pot[i] + e.cost <
    ↪ pot[e.to]) {
130                         pot[e.to] = pot[i] + e.cost;
131                         pe[e.to] = id;
132                         if (!in_queue[e.to]) {
133                             que.push_back(e.to);
134                             in_queue[e.to] = true;
135                         }
136                     }
137                 }
138             }
139         }
140     }
141     while (pot[fin] < INF_C) {
142         T push = numeric_limits<T>::max();
143         int v = fin;
144         while (v != st) {
145             const edge &e = edges[pe[v]];

```



```

146     push = min(push, e.c - e.f);
147     v = e.from;
148 }
149 v = fin;
150 while (v != st) {
151     edge &e = edges[pe[v]];
152     e.f += push;
153     edge &back = edges[pe[v] ^ 1];
154     back.f -= push;
155     v = e.from;
156 }
157 flow += push;
158 cost += push * pot[fin];
159 expath(st);
160 }
161 return {flow, cost};
162 }
163 };
164
165 // Examples: MCMF<int, int> g(n); g.add(u,v,c,w,0);
166 // To recover flow through original edges: iterate over
    ↪ even indices in edges.

```

Graphs

Kuhn's algorithm for bipartite matching

```

1  /*
2  The graph is split into 2 halves of n1 and n2 vertices.
3  Complexity: O(n1 * m). Usually runs much faster. MUCH
    ↪ FASTER!!!
4  */
5  const int N = 305;
6
7  vector<int> g[N]; // Stores edges from left half to
    ↪ right.
8  bool used[N]; // Stores if vertex from left half is
    ↪ used.
9  int mt[N]; // For every vertex in right half, stores to
    ↪ which vertex in left half it's matched (-1 if not
    ↪ matched).
10
11 bool try_dfs(int v){
12     if (used[v]) return false;
13     used[v] = 1;
14     for (auto u : g[v]){
15         if (mt[u] == -1 || try_dfs(mt[u])){
16             mt[u] = v;
17             return true;
18         }
19     }
20     return false;
21 }
22
23 int main(){
24     // .....

```

```

25     for (int i = 1; i <= n2; i++) mt[i] = -1;
26     for (int i = 1; i <= n1; i++) used[i] = 0;
27     for (int i = 1; i <= n1; i++){
28         if (try_dfs(i)){
29             for (int j = 1; j <= n1; j++) used[j] = 0;
30         }
31     }
32     vector<pair<int, int>> ans;
33     for (int i = 1; i <= n2; i++){
34         if (mt[i] != -1) ans.pb({mt[i], i});
35     }
36 }
37
38 // Finding maximal independent set: size = # of nodes -
    ↪ # of edges in matching.
39 // To construct: launch Kuhn-like DFS from unmatched
    ↪ nodes in the left half.
40 // Independent set = visited nodes in left half +
    ↪ unvisited in right half.
41 // Finding minimal vertex cover: complement of maximal
    ↪ independent set.

```

Hungarian algorithm for Assignment Problem

- Given a 1-indexed $(n \times m)$ matrix A , select a number in each row such that each column has at most 1 number selected, and the sum of the selected numbers is minimized.

```

1  int INF = 1e9; // constant greater than any number in
    ↪ the matrix
2  vector<int> u(n+1), v(m+1), p(m+1), way(m+1);
3  for (int i=1; i<=n; ++i) {
4      p[0] = i;
5      int j0 = 0;
6      vector<int> minv (m+1, INF);
7      vector<bool> used (m+1, false);
8      do {
9          used[j0] = true;
10         int i0 = p[j0], delta = INF, j1;
11         for (int j=1; j<=m; ++j)
12             if (!used[j]) {
13                 int cur = A[i0][j]-u[i0]-v[j];
14                 if (cur < minv[j])
15                     minv[j] = cur, way[j] = j0;
16                 if (minv[j] < delta)
17                     delta = minv[j], j1 = j;
18             }
19         for (int j=0; j<=m; ++j)
20             if (used[j])
21                 u[p[j]] += delta, v[j] -= delta;
22             else
23                 minv[j] -= delta;
24         j0 = j1;
25     } while (p[j0] != 0);
26     do {

```

```

27         int j1 = way[j0];
28         p[j0] = p[j1];
29         j0 = j1;
30     } while (j0);
31 }
32 vector<int> ans (n+1); // ans[i] stores the column
    ↪ selected for row i
33 for (int j=1; j<=m; ++j)
34     ans[p[j]] = j;
35 int cost = -v[0]; // the total cost of the matching

```

Dijkstra's Algorithm

```

1  priority_queue<pair<ll, ll>, vector<pair<ll, ll>>,
    ↪ greater<pair<ll, ll>>> q;
2  dist[start] = 0;
3  q.push({0, start});
4  while (!q.empty()){
5      auto [d, v] = q.top();
6      q.pop();
7      if (d != dist[v]) continue;
8      for (auto [u, w] : g[v]){
9          if (dist[u] > dist[v] + w){
10             dist[u] = dist[v] + w;
11             q.push({dist[u], u});
12         }
13     }
14 }

```

Eulerian Cycle DFS

```

1  void dfs(int v){
2      while (!g[v].empty()){
3          int u = g[v].back();
4          g[v].pop_back();
5          dfs(u);
6          ans.pb(v);
7      }
8  }

```

SCC and 2-SAT

```

1  void scc(vector<vector<int>>& g, int* idx) {
2      int n = g.size(), ct = 0;
3      int out[n];
4      vector<int> ginv[n];
5      memset(out, -1, sizeof out);
6      memset(idx, -1, n * sizeof(int));
7      function<void(int)> dfs = [&](int cur) {
8          out[cur] = INT_MAX;
9          for(int v : g[cur]) {
10             ginv[v].push_back(cur);
11             if(out[v] == -1) dfs(v);
12         }
13         ct++; out[cur] = ct;

```



```

14     };
15     vector<int> order;
16     for(int i = 0; i < n; i++) {
17         order.push_back(i);
18         if(out[i] == -1) dfs(i);
19     }
20     sort(order.begin(), order.end(), [&](int& u, int& v) {
21         return out[u] > out[v];
22     });
23     ct = 0;
24     stack<int> s;
25     auto dfs2 = [&](int start) {
26         s.push(start);
27         while(!s.empty()) {
28             int cur = s.top();
29             s.pop();
30             idx[cur] = ct;
31             for(int v : ginv[cur])
32                 if(idx[v] == -1) s.push(v);
33         }
34     };
35     for(int v : order) {
36         if(idx[v] == -1) {
37             dfs2(v);
38             ct++;
39         }
40     }
41 }

42 // 0 => impossible, 1 => possible
43 pair<int, vector<int>> sat2(int n, vector<pair<int, int>>&
44     clauses) {
45     vector<int> ans(n);
46     vector<vector<int>> g(2*n + 1);
47     for(auto [x, y] : clauses) {
48         x = x < 0 ? -x + n : x;
49         y = y < 0 ? -y + n : y;
50         int nx = x <= n ? x + n : x - n;
51         int ny = y <= n ? y + n : y - n;
52         g[nx].push_back(y);
53         g[ny].push_back(x);
54     }
55     int idx[2*n + 1];
56     scc(g, idx);
57     for(int i = 1; i <= n; i++) {
58         if(idx[i] == idx[i + n]) return {0, {}};
59         ans[i - 1] = idx[i + n] < idx[i];
60     }
61     return {1, ans};
62 }

```

Finding Bridges

```

1  /*
2  Bridges.
3  Results are stored in a map "is_bridge".
4  For each connected component, call "dfs(starting vertex,
5  starting vertex)".

```

```

5  */
6  const int N = 2e5 + 10; // Careful with the constant!
7
8  vector<int> g[N];
9  int tin[N], fup[N], timer;
10 map<pair<int, int>, bool> is_bridge;
11
12 void dfs(int v, int p){
13     tin[v] = ++timer;
14     fup[v] = tin[v];
15     for (auto u : g[v]){
16         if (!tin[u]){
17             dfs(u, v);
18             if (fup[u] > tin[v]){
19                 is_bridge[{v, u}] = is_bridge[{u, v}] = true;
20             }
21             fup[v] = min(fup[v], fup[u]);
22         }
23         else{
24             if (u != p) fup[v] = min(fup[v], tin[u]);
25         }
26     }
27 }

```

Virtual Tree

```

1  // order stores the nodes in the queried set
2  sort(all(order), [&](int u, int v){return tin[u] <
3  tin[v]});
4  int m = sz(order);
5  for (int i = 1; i < m; i++){
6      order.pb(lca(order[i], order[i - 1]));
7  }
8  sort(all(order), [&](int u, int v){return tin[u] <
9  tin[v]});
10 order.erase(unique(all(order)), order.end());
11 vector<int> stk{order[0]};
12 for (int i = 1; i < sz(order); i++){
13     int v = order[i];
14     while (tout[stk.back()] < tout[v]) stk.pop_back();
15     int u = stk.back();
16     vg[u].pb({v, dep[v] - dep[u]});
17     stk.pb(v);
18 }

```

HLD on Edges DFS

```

1  void dfs1(int v, int p, int d){
2      par[v] = p;
3      for (auto e : g[v]){
4          if (e.fi == p){
5              g[v].erase(find(all(g[v]), e));
6              break;
7          }
8      }
9      dep[v] = d;
10     sz[v] = 1;

```

```

11     for (auto [u, c] : g[v]){
12         dfs1(u, v, d + 1);
13         sz[v] += sz[u];
14     }
15     if (!g[v].empty()) iter_swap(g[v].begin(),
16     max_element(all(g[v]), comp));
17 }
18 void dfs2(int v, int rt, int c){
19     pos[v] = sz(a);
20     a.pb(c);
21     root[v] = rt;
22     for (int i = 0; i < sz(g[v]); i++){
23         auto [u, c] = g[v][i];
24         if (!i) dfs2(u, rt, c);
25         else dfs2(u, u, c);
26     }
27 }
28 int getans(int u, int v){
29     int res = 0;
30     for (; root[u] != root[v]; v = par[root[v]]){
31         if (dep[root[u]] > dep[root[v]]) swap(u, v);
32         res = max(res, rmq(0, 0, n - 1, pos[root[v]],
33         pos[v]));
34     }
35     if (pos[u] > pos[v]) swap(u, v);
36     return max(res, rmq(0, 0, n - 1, pos[u] + 1, pos[v]));
37 }

```

Centroid Decomposition

```

1  vector<char> res(n), seen(n), sz(n);
2  function<int(int, int)> get_size = [&](int node, int fa)
3  {
4      sz[node] = 1;
5      for (auto& ne : g[node]) {
6          if (ne == fa || seen[ne]) continue;
7          sz[node] += get_size(ne, node);
8      }
9      return sz[node];
10 }
11 function<int(int, int, int)> find_centroid = [&](int
12 node, int fa, int t) {
13     for (auto& ne : g[node])
14         if (ne != fa && !seen[ne] && sz[ne] > t / 2) return
15         find_centroid(ne, node, t);
16     return node;
17 }
18 function<void(int, char)> solve = [&](int node, char
19 cur) {
20     get_size(node, -1); auto c = find_centroid(node, -1,
21     sz[node]);
22     seen[c] = 1, res[c] = cur;
23     for (auto& ne : g[c]) {
24         if (seen[ne]) continue;
25         solve(ne, char(cur + 1)); // we can pass c here to
26         build tree

```

```

21 }
22 };

```

Math

Binary exponentiation

```

1 ll power(ll a, ll b){
2     ll res = 1;
3     for (; b; a = a * a % MOD, b >= 1){
4         if (b & 1) res = res * a % MOD;
5     }
6     return res;
7 }

```

Matrix Exponentiation: $O(n^3 \log b)$

```

1 const int N = 100, MOD = 1e9 + 7;
2
3 struct matrix{
4     ll m[N][N];
5     int n;
6     matrix(){
7         n = N;
8         memset(m, 0, sizeof(m));
9     };
10    matrix(int n){
11        n = n;
12        memset(m, 0, sizeof(m));
13    };
14    matrix(int n, ll val){
15        n = n;
16        memset(m, 0, sizeof(m));
17        for (int i = 0; i < n; i++) m[i][i] = val;
18    };
19
20    matrix operator* (matrix oth){
21        matrix res(n);
22        for (int i = 0; i < n; i++){
23            for (int j = 0; j < n; j++){
24                for (int k = 0; k < n; k++){
25                    res.m[i][j] = (res.m[i][j] + m[i][k] *
26                    oth.m[k][j]) % MOD;
27                }
28            }
29        }
30        return res;
31    };
32
33    matrix power(matrix a, ll b){
34        matrix res(a.n, 1);
35        for (; b; a = a * a, b >= 1){
36            if (b & 1) res = res * a;
37        }

```

```

38     return res;
39 }

```

Extended Euclidean Algorithm

```

1 // gives (x, y) for ax + by = g
2 // solutions given (x0, y0): a(x0 + kb/g) + b(y0 - ka/g)
3 // = g
4 int gcd(int a, int b, int& x, int& y) {
5     x = 1, y = 0; int sum1 = a;
6     int x2 = 0, y2 = 1, sum2 = b;
7     while (sum2) {
8         int q = sum1 / sum2;
9         tie(x, x2) = make_tuple(x2, x - q * x2);
10        tie(y, y2) = make_tuple(y2, y - q * y2);
11        tie(sum1, sum2) = make_tuple(sum2, sum1 - q * sum2);
12    }
13    return sum1;

```

Linear Sieve

- Mobius Function

```

1 vector<int> prime;
2 bool is_composite[MAX_N];
3 int mu[MAX_N];
4
5 void sieve(int n){
6     fill(is_composite, is_composite + n, 0);
7     mu[1] = 1;
8     for (int i = 2; i < n; i++){
9         if (!is_composite[i]){
10            prime.push_back(i);
11            mu[i] = -1; //i is prime
12        }
13        for (int j = 0; j < prime.size() && i * prime[j] < n;
14        j++){
15            is_composite[i * prime[j]] = true;
16            if (i % prime[j] == 0){
17                mu[i * prime[j]] = 0; //prime[j] divides i
18                break;
19            } else {
20                mu[i * prime[j]] = -mu[i]; //prime[j] does not
21                divide i
22            }
23        }

```

- Euler's Totient Function

```

1 vector<int> prime;
2 bool is_composite[MAX_N];
3 int phi[MAX_N];
4
5 void sieve(int n){
6     fill(is_composite, is_composite + n, 0);

```

```

7     phi[1] = 1;
8     for (int i = 2; i < n; i++){
9         if (!is_composite[i]){
10            prime.push_back(i);
11            phi[i] = i - 1; //i is prime
12        }
13        for (int j = 0; j < prime.size() && i * prime[j] < n;
14        j++){
15            is_composite[i * prime[j]] = true;
16            if (i % prime[j] == 0){
17                phi[i * prime[j]] = phi[i] * prime[j]; //prime[j]
18                divides i
19                break;
20            } else {
21                phi[i * prime[j]] = phi[i] * phi[prime[j]];
22                //prime[j] does not divide i
23            }

```

Gaussian Elimination

```

1 bool is_0(Z v) { return v.x == 0; }
2 Z abs(Z v) { return v; }
3 bool is_0(double v) { return abs(v) < 1e-9; }
4
5 // 1 => unique solution, 0 => no solution, -1 =>
6 // multiple solutions
7 template <typename T>
8 int gaussian_elimination(vector<vector<T>> &a, int
9 limit) {
10    if (a.empty() || a[0].empty()) return -1;
11    int h = (int)a.size(), w = (int)a[0].size(), r = 0;
12    for (int c = 0; c < limit; c++) {
13        int id = -1;
14        for (int i = r; i < h; i++) {
15            if (!is_0(a[i][c]) && (id == -1 || abs(a[id][c]) <
16            abs(a[i][c]))) {
17                id = i;
18            }
19        }
20        if (id == -1) continue;
21        if (id > r) {
22            swap(a[r], a[id]);
23            for (int j = c; j < w; j++) a[id][j] = -a[id][j];
24        }
25        vector<int> nonzero;
26        for (int j = c; j < w; j++) {
27            if (!is_0(a[r][j])) nonzero.push_back(j);
28        }
29        T inv_a = 1 / a[r][c];
30        for (int i = r + 1; i < h; i++) {
31            if (is_0(a[i][c])) continue;
32            T coeff = -a[i][c] * inv_a;
33            for (int j : nonzero) a[i][j] += coeff * a[r][j];

```

```

32     ++r;
33 }
34 for (int row = h - 1; row >= 0; row--) {
35     for (int c = 0; c < limit; c++) {
36         if (!is_0(a[row][c])) {
37             T inv_a = 1 / a[row][c];
38             for (int i = row - 1; i >= 0; i--) {
39                 if (is_0(a[i][c])) continue;
40                 T coeff = -a[i][c] * inv_a;
41                 for (int j = c; j < w; j++) a[i][j] += coeff;
42             }
43             break;
44         }
45     }
46 } // not-free variables: only it on its line
47 for (int i = r; i < h; i++) if (!is_0(a[i][limit]))
48     return 0;
49 return (r == limit) ? 1 : -1;
50 }
51 template <typename T>
52 pair<int, vector<T>> solve_linear(vector<vector<T>> a,
53     const vector<T> &b, int w) {
54     int h = (int)a.size();
55     for (int i = 0; i < h; i++) a[i].push_back(b[i]);
56     int sol = gaussian_elimination(a, w);
57     if (!sol) return {0, vector<T>()};
58     vector<T> x(w, 0);
59     for (int i = 0; i < h; i++) {
60         for (int j = 0; j < w; j++) {
61             if (!is_0(a[i][j])) {
62                 x[j] = a[i][w] / a[i][j];
63                 break;
64             }
65         }
66     }
67     return {sol, x};

```

is_prime

- (Miller–Rabin primality test)

```

1 typedef __int128_t i128;
2
3 i128 power(i128 a, i128 b, i128 MOD = 1, i128 res = 1) {
4     for (; b /= 2, (a *= a) %= MOD)
5         if (b & 1) (res *= a) %= MOD;
6     return res;
7 }
8
9 bool is_prime(ll n) {
10     if (n < 2) return false;
11     static constexpr int A[] = {2, 3, 5, 7, 11, 13, 17,
12     19, 23};
13     int s = __builtin_ctzll(n - 1);
14     ll d = (n - 1) >> s;

```

```

14 for (auto a : A) {
15     if (a == n) return true;
16     ll x = (ll)power(a, d, n);
17     if (x == 1 || x == n - 1) continue;
18     bool ok = false;
19     for (int i = 0; i < s - 1; ++i) {
20         x = ll((i128)x * x % n); // potential overflow!
21         if (x == n - 1) {
22             ok = true;
23             break;
24         }
25     }
26     if (!ok) return false;
27 }
28 return true;
29 }
30
31 typedef __int128_t i128;
32
33 ll pollard_rho(ll x) {
34     ll s = 0, t = 0, c = rng() % (x - 1) + 1;
35     ll stp = 0, goal = 1, val = 1;
36     for (goal = 1; goal *= 2, s = t, val = 1) {
37         for (stp = 1; stp <= goal; ++stp) {
38             t = ll(((i128)t * t + c) % x);
39             val = ll((i128)val * abs(t - s) % x);
40             if ((stp % 127) == 0) {
41                 ll d = gcd(val, x);
42                 if (d > 1) return d;
43             }
44         }
45         ll d = gcd(val, x);
46         if (d > 1) return d;
47     }
48 }
49
50 ll get_max_factor(ll x) {
51     ll max_factor = 0;
52     function<void(ll)> fac = [&](ll x) {
53         if (x <= max_factor || x < 2) return;
54         if (is_prime(x)) {
55             max_factor = max_factor > x ? max_factor : x;
56             return;
57         }
58         ll p = x;
59         while (p >= x) p = pollard_rho(x);
60         while ((x % p) == 0) x /= p;
61         fac(x), fac(p);
62     };
63     fac(x);
64     return max_factor;
65 }

```

Berlekamp-Massey

- Recovers any n -order linear recurrence relation from the first $2n$ terms of the sequence.

- Input s is the sequence to be analyzed.
- Output c is the shortest sequence c_1, \dots, c_n , such that

$$s_m = \sum_{i=1}^n c_i \cdot s_{m-i}, \text{ for all } m \geq n.$$

- Be careful since c is returned in 0-based indexation.
- Complexity: $O(N^2)$

```

1 vector<ll> berlekamp_massey(vector<ll> s) {
2     int n = sz(s), l = 0, m = 1;
3     vector<ll> b(n), c(n);
4     ll ldd = b[0] = c[0] = 1;
5     for (int i = 0; i < n; i++, m++) {
6         ll d = s[i];
7         for (int j = 1; j <= l; j++) d = (d + c[j] * s[i -
8     j]) % MOD;
9         if (d == 0) continue;
10        vector<ll> temp = c;
11        ll coef = d * power(ldd, MOD - 2) % MOD;
12        for (int j = m; j < n; j++){
13            c[j] = (c[j] + MOD - coef * b[j - m]) % MOD;
14            if (c[j] < 0) c[j] += MOD;
15        }
16        if (2 * l <= i) {
17            l = i + 1 - l;
18            b = temp;
19            ldd = d;
20            m = 0;
21        }
22    }
23    c.resize(l + 1);
24    c.erase(c.begin());
25    for (ll &x : c)
26        x = (MOD - x) % MOD;
27    return c;
}

```

Calculating k-th term of a linear recurrence

- Given the first n terms s_0, s_1, \dots, s_{n-1} and the sequence c_1, c_2, \dots, c_n such that

$$s_m = \sum_{i=1}^n c_i \cdot s_{m-i}, \text{ for all } m \geq n,$$

the function `calc_kth` computes s_k .

- Complexity: $O(n^2 \log k)$

```

1 vector<ll> poly_mult_mod(vector<ll> p, vector<ll> q,
  ↪ vector<ll>& c){
2     vector<ll> ans(sz(p) + sz(q) - 1);
3     for (int i = 0; i < sz(p); i++){
4         for (int j = 0; j < sz(q); j++){
5             ans[i + j] = (ans[i + j] + p[i] * q[j]) % MOD;
6         }
7     }
8     int n = sz(ans), m = sz(c);
9     for (int i = n - 1; i >= m; i--){
10         for (int j = 0; j < m; j++){
11             ans[i - 1 - j] = (ans[i - 1 - j] + c[j] * ans[i]) % MOD;
12         }
13     }
14     ans.resize(m);
15     return ans;
16 }

17 ll calc_kth(vector<ll> s, vector<ll> c, ll k){
18     assert(sz(s) >= sz(c)); // size of s can be greater
19     ↪ than c, but not less
20     if (k < sz(s)) return s[k];
21     vector<ll> res{1};
22     for (vector<ll> poly = {0, 1}; k; poly =
  ↪ poly_mult_mod(poly, poly, c), k >= 1){
23         if (k & 1) res = poly_mult_mod(res, poly, c);
24     }
25     ll ans = 0;
26     for (int i = 0; i < min(sz(res), sz(c)); i++) ans =
  ↪ (ans + s[i] * res[i]) % MOD;
27     return ans;
28 }

```

Partition Function

- Returns number of partitions of n in $O(n^{1.5})$

```

1 int partition(int n) {
2     int dp[n + 1];
3     dp[0] = 1;
4     for (int i = 1; i <= n; i++) {
5         dp[i] = 0;
6         for (int j = 1, r = 1; i - (3 * j * j - j) / 2 >= 0;
  ↪ ++j, r += -1) {
7             dp[i] += dp[i - (3 * j * j - j) / 2] * r;
8             if (i - (3 * j * j + j) / 2 >= 0) dp[i] += dp[i -
  ↪ (3 * j * j + j) / 2] * r;
9         }
10    }
11    return dp[n];
12 }

```

NTT

```

1 void ntt(vector<ll>& a, int f) {
2     int n = int(a.size());

```

```

3     vector<ll> w(n);
4     vector<int> rev(n);
5     for (int i = 0; i < n; i++) rev[i] = (rev[i / 2] / 2) |
  ↪ ((i & 1) * (n / 2));
6     for (int i = 0; i < n; i++) {
7         if (i < rev[i]) swap(a[i], a[rev[i]]);
8     }
9     ll wn = power(f ? (MOD + 1) / 3 : 3, (MOD - 1) / n);
10    w[0] = 1;
11    for (int i = 1; i < n; i++) w[i] = w[i - 1] * wn % MOD;
12    for (int mid = 1; mid < n; mid *= 2) {
13        for (int i = 0; i < n; i += 2 * mid) {
14            for (int j = 0; j < mid; j++) {
15                ll x = a[i + j], y = a[i + j + mid] * w[n / (2 *
  ↪ mid) * j] % MOD;
16                a[i + j] = (x + y) % MOD, a[i + j + mid] = (x -
  ↪ y) % MOD;
17            }
18        }
19    }
20    if (f) {
21        ll iv = power(n, MOD - 2);
22        for (auto& x : a) x = x * iv % MOD;
23    }
24 }

25 vector<ll> mul(vector<ll> a, vector<ll> b) {
26     int n = 1, m = (int)a.size() + (int)b.size() - 1;
27     while (n < m) n *= 2;
28     a.resize(n), b.resize(n);
29     ntt(a, 0), ntt(b, 0); // if squaring, you can save one
  ↪ NTT here
30     for (int i = 0; i < n; i++) a[i] = a[i] * b[i] % MOD;
31     ntt(a, 1);
32     a.resize(m);
33     return a;
34 }

```

FFT

```

1 const ld PI = acos(-1);
2 auto mul = [&](const vector<ld>& aa, const vector<ld>&
  ↪ bb) {
3     int n = (int)aa.size(), m = (int)bb.size(), bit = 1;
4     while ((1 << bit) < n + m - 1) bit++;
5     int len = 1 << bit;
6     vector<complex<ld>> a(len), b(len);
7     vector<int> rev(len);
8     for (int i = 0; i < n; i++) a[i].real(aa[i]);
9     for (int i = 0; i < m; i++) b[i].real(bb[i]);
10    for (int i = 0; i < len; i++) rev[i] = (rev[i >> 1] >
  ↪ 1) | ((i & 1) << (bit - 1));
11    auto fft = [&](vector<complex<ld>>& p, int inv) {
12        for (int i = 0; i < len; i++)
13            if (i < rev[i]) swap(p[i], p[rev[i]]);
14        for (int mid = 1; mid < len; mid *= 2) {
15            auto w1 = complex<ld>(cos(PI / mid), (inv ? -1 :
  ↪ 1) * sin(PI / mid));

```

```

        for (int i = 0; i < len; i += mid * 2) {
            auto wk = complex<ld>(1, 0);
            for (int j = 0; j < mid; j++, wk = wk * w1) {
                auto x = p[i + j], y = wk * p[i + j + mid];
                p[i + j] = x + y, p[i + j + mid] = x - y;
            }
        }
        if (inv == 1) {
            for (int i = 0; i < len; i++)
                ↪ p[i].real(p[i].real() / len);
        }
    };
    fft(a, 0), fft(b, 0);
    for (int i = 0; i < len; i++) a[i] = a[i] * b[i];
    fft(a, 1);
    a.resize(n + m - 1);
    vector<ld> res(n + m - 1);
    for (int i = 0; i < n + m - 1; i++) res[i] =
  ↪ a[i].real();
    return res;
};

```

MIT's FFT/NTT, Polynomial mod/log/exp Template

- For integers rounding works if $(|a| + |b|) \max(a, b) < \sim 10^9$, or in theory maybe 10^6
- $\frac{1}{P(x)}$ in $O(n \log n)$, $e^{P(x)}$ in $O(n \log n)$, $\ln(P(x))$ in $O(n \log n)$, $P(x)^k$ in $O(n \log n)$, Evaluates $P(x_1), \dots, P(x_n)$ in $O(n \log^2 n)$, Lagrange Interpolation in $O(n \log^2 n)$

```

1 // use #define FFT 1 to use FFT instead of NTT (default)
2 // Examples:
3 // poly a(n+1); // constructs degree n poly
4 // a[0].v = 10; // assigns constant term a_0 = 10
5 // poly b = exp(a);
6 // poly is vector<num>
7 // for NTT, num stores just one int named v
8 // for FFT, num stores two doubles named x (real), y
  ↪ (imag)

9
10 #define sz(x) ((int)x.size())
11 #define rep(i, j, k) for (int i = int(j); i < int(k);
  ↪ i++)
12 #define trav(a, x) for (auto &a : x)
13 #define per(i, a, b) for (int i = (b)-1; i >= (a); --i)
14 using ll = long long;
15 using vi = vector<int>;

16 namespace fft {
17     #if FFT
18     // FFT

```

```

20 using dbl = double;
21 struct num {
22     dbl x, y;
23     num(dbl x_ = 0, dbl y_ = 0): x(x_), y(y_) {}
24 };
25 inline num operator+(num a, num b) {
26     return num(a.x + b.x, a.y + b.y);
27 }
28 inline num operator-(num a, num b) {
29     return num(a.x - b.x, a.y - b.y);
30 }
31 inline num operator*(num a, num b) {
32     return num(a.x * b.x - a.y * b.y, a.x * b.y + a.y *
    ↪ b.x);
33 }
34 inline num conj(num a) { return num(a.x, -a.y); }
35 inline num inv(num a) {
36     dbl n = (a.x * a.x + a.y * a.y);
37     return num(a.x / n, -a.y / n);
38 }
39
40 #else
41 // NTT
42 const int mod = 998244353, g = 3;
43 // For  $p < 2^{30}$  there is also  $(5 \ll 25, 3)$ ,  $(7 \ll 26,$ 
    ↪  $3)$ ,
44 //  $(479 \ll 21, 3)$  and  $(483 \ll 21, 5)$ . Last two are  $>$ 
    ↪  $10^9$ .
45 struct num {
46     int v;
47     num(ll v_ = 0): v(int(v_ % mod)) {
48         if (v < 0) v += mod;
49     }
50     explicit operator int() const { return v; }
51 };
52 inline num operator+(num a, num b) { return num(a.v +
    ↪ b.v); }
53 inline num operator-(num a, num b) {
54     return num(a.v + mod - b.v);
55 }
56 inline num operator*(num a, num b) {
57     return num(1ll * a.v * b.v);
58 }
59 inline num pow(num a, int b) {
60     num r = 1;
61     do {
62         if (b & 1) r = r * a;
63         a = a * a;
64     } while (b >= 1);
65     return r;
66 }
67 inline num inv(num a) { return pow(a, mod - 2); }
68
69 #endif
70 using vn = vector<num>;
71 vi rev({0, 1});
72 vn rt(2, num(1)), fa, fb;
73 inline void init(int n) {
74     if (n <= sz(rt)) return;
75     rev.resize(n);
76     rep(i, 0, n) rev[i] = (rev[i >> 1] | ((i & 1) * n))
    ↪ 1;
77     rt.reserve(n);
78     for (int k = sz(rt); k < n; k *= 2) {
79         rt.resize(2 * k);
80         #if FFT
81             double a = M_PI / k;
82             num z(cos(a), sin(a)); // FFT
83         #else
84             num z = pow(num(g), (mod - 1) / (2 * k)); // NTT
85         #endif
86         rep(i, k / 2, k) rt[2 * i] = rt[i],
87             rt[2 * i + 1] = rt[i] * z;
88     }
89 }
90 inline void fft(vector<num>& a, int n) {
91     init(n);
92     int s = __builtin_ctz(sz(rev)) / n;
93     rep(i, 0, n) if (i < rev[i] >> s) swap(a[i], a[rev[i]
    ↪ >> s]);
94     for (int k = 1; k < n; k *= 2)
95         for (int i = 0; i < n; i += 2 * k) rep(j, 0, k) {
96             num t = rt[j + k] * a[i + j + k];
97             a[i + j + k] = a[i + j] - t;
98             a[i + j] = a[i + j] + t;
99         }
100 }
101 // Complex/NTT
102 vn multiply(vn a, vn b) {
103     int s = sz(a) + sz(b) - 1;
104     if (s <= 0) return {};
105     int L = s > 1 ? 32 - __builtin_clz(s - 1) : 0, n =
    ↪ << L;
106     a.resize(n), b.resize(n);
107     fft(a, n);
108     fft(b, n);
109     num d = inv(num(n));
110     rep(i, 0, n) a[i] = a[i] * b[i] * d;
111     reverse(a.begin() + 1, a.end());
112     fft(a, n);
113     a.resize(s);
114     return a;
115 }
116 // Complex/NTT power-series inverse
117 // Doubles  $b$  as  $b[:n] = (2 - a[:n] * b[:n/2]) * b[:n/2]$ 
118 vn inverse(const vn& a) {
119     if (a.empty()) return {};
120     vn b({inv(a[0])});
121     b.reserve(2 * a.size());
122     while (sz(b) < sz(a)) {
123         int n = 2 * sz(b);
124         b.resize(2 * n, 0);
125         if (sz(fa) < 2 * n) fa.resize(2 * n);
126         fill(fa.begin(), fa.begin() + 2 * n, 0);
127         copy(a.begin(), a.begin() + min(n, sz(a)),
    ↪ fa.begin());
128         fft(b, 2 * n);
129         num d = inv(num(2 * n));
130         rep(i, 0, 2 * n) b[i] = b[i] * (2 - fa[i] * b[i]) *
    ↪ d;
131         b.resize(n);
132     }
133     #if FFT
134         double a = M_PI / k;
135         num z(cos(a), sin(a)); // FFT
136     #else
137         num z = pow(num(g), (mod - 1) / (2 * k)); // NTT
138     #endif
139     rep(i, k / 2, k) rt[2 * i] = rt[i],
140         rt[2 * i + 1] = rt[i] * z;
141 }
142
143 // Double multiply (num = complex)
144 using vd = vector<double>;
145 vd multiply(const vd& a, const vd& b) {
146     int s = sz(a) + sz(b) - 1;
147     if (s <= 0) return {};
148     int L = s > 1 ? 32 - __builtin_clz(s - 1) : 0, n =
    ↪ << L;
149     if (sz(fa) < n) fa.resize(n);
150     if (sz(fb) < n) fb.resize(n);
151     fill(fa.begin(), fa.begin() + n, 0);
152     rep(i, 0, sz(a)) fa[i].x = a[i];
153     rep(i, 0, sz(b)) fa[i].y = b[i];
154     fft(fa, n);
155     trav(x, fa) x = x * x;
156     rep(i, 0, n) fb[i] = fa[(n - i) & (n - 1)] -
    ↪ conj(fa[i]);
157     fft(fb, n);
158     vd r(s);
159     rep(i, 0, s) r[i] = fb[i].y / (4 * n);
160     return r;
161 }
162 // Integer multiply mod m (num = complex)
163 vi multiply_mod(const vi& a, const vi& b, int m) {
164     int s = sz(a) + sz(b) - 1;
165     if (s <= 0) return {};
166     int L = s > 1 ? 32 - __builtin_clz(s - 1) : 0, n =
    ↪ << L;
167     if (sz(fa) < n) fa.resize(n);
168     if (sz(fb) < n) fb.resize(n);
169     rep(i, 0, sz(a)) fa[i] =
170         num(a[i] & ((1 << 15) - 1), a[i] >> 15);
171     fill(fa.begin() + sz(a), fa.begin() + n, 0);
172     rep(i, 0, sz(b)) fb[i] =
173         num(b[i] & ((1 << 15) - 1), b[i] >> 15);
174     fill(fb.begin() + sz(b), fb.begin() + n, 0);
175     fft(fa, n);
176     fft(fb, n);
177     double r0 = 0.5 / n; // 1/2n
178     rep(i, 0, n / 2 + 1) {
179         int j = (n - i) & (n - 1);
180         num g0 = (fb[i] + conj(fb[j])) * r0;
181         num g1 = (fb[i] - conj(fb[j])) * r0;
182         swap(g1.x, g1.y);
183         g1.y *= -1;

```

```

181     if (j != i) {
182         swap(fa[j], fa[i]);
183         fb[j] = fa[j] * g1;
184         fa[j] = fa[j] * g0;
185     }
186     fb[i] = fa[i] * conj(g1);
187     fa[i] = fa[i] * conj(g0);
188 }
189 fft(fa, n);
190 fft(fb, n);
191 vi r(s);
192 rep(i, 0, s) r[i] =
193     int((ll(fa[i].x + 0.5) + (ll(fa[i].y + 0.5) % m < 15) +
194         (ll(fb[i].x + 0.5) % m < 15) +
195         (ll(fb[i].y + 0.5) % m < 30)) %
196         m);
197     return r;
198 }
199 #endif
200 } // namespace fft
201 // For multiply_mod, use num = modnum, poly =
202     vector<num>
203 using fft::num;
204 using poly = fft::vn;
205 using fft::multiply;
206 using fft::inverse;
207
208 poly& operator+=(poly& a, const poly& b) {
209     if (sz(a) < sz(b)) a.resize(b.size());
210     rep(i, 0, sz(b)) a[i] = a[i] + b[i];
211     return a;
212 }
213
214 poly operator+(const poly& a, const poly& b) {
215     poly r = a;
216     r += b;
217     return r;
218 }
219
220 poly& operator-=(poly& a, const poly& b) {
221     if (sz(a) < sz(b)) a.resize(b.size());
222     rep(i, 0, sz(b)) a[i] = a[i] - b[i];
223     return a;
224 }
225
226 poly operator-(const poly& a, const poly& b) {
227     poly r = a;
228     r -= b;
229     return r;
230 }
231
232 poly& operator*(const poly& a, const poly& b) {
233     return multiply(a, b);
234 }
235
236 poly& operator==(poly& a, const poly& b) { return a =
237     a * b; }
238
239 poly& operator==(poly& a, const num& b) { // Optional
240     trav(x, a) x = x * b;
241     return a;
242 }
243
244 poly operator*(const poly& a, const num& b) {
245     poly r = a;
246     r *= b;
247     return r;
248 }
249
250 // Polynomial floor division; no leading 0's please
251 poly operator/(poly a, poly b) {
252     if (sz(a) < sz(b)) return {};
253     int s = sz(a) - sz(b) + 1;
254     reverse(a.begin(), a.end());
255     reverse(b.begin(), b.end());
256     a.resize(s);
257     b.resize(s);
258     a = a * inverse(move(b));
259     a.resize(s);
260     reverse(a.begin(), a.end());
261     return a;
262 }
263
264 poly& operator/=(poly& a, const poly& b) { return a =
265     a / b; }
266
267 poly& operator%=(poly& a, const poly& b) {
268     if (sz(a) >= sz(b)) {
269         poly c = (a / b) * b;
270         a.resize(sz(b) - 1);
271         rep(i, 0, sz(a)) a[i] = a[i] - c[i];
272     }
273     return a;
274 }
275
276 poly operator%(const poly& a, const poly& b) {
277     poly r = a;
278     r %= b;
279     return r;
280 }
281
282 // Log/exp/pow
283 poly deriv(const poly& a) {
284     if (a.empty()) return {};
285     poly b(sz(a) - 1);
286     rep(i, 1, sz(a)) b[i - 1] = a[i] * i;
287     return b;
288 }
289
290 poly integ(const poly& a) {
291     poly b(sz(a) + 1);
292     b[1] = 1; // mod p
293     rep(i, 2, sz(b)) b[i] =
294         b[fft::mod % i] * (-fft::mod / i); // mod p
295     rep(i, 1, sz(b)) b[i] = a[i - 1] * b[i]; // mod p
296     //rep(i, 1, sz(b)) b[i] = a[i - 1] * inv(num(i)); // else
297     return b;
298 }
299
300 poly log(const poly& a) { // MUST have a[0] == 1
301     poly b = integ(deriv(a) * inverse(a));
302     b.resize(a.size());
303     return b;
304 }
305
306 poly exp(const poly& a) { // MUST have a[0] == 0
307     poly b(1, num(1));
308     if (a.empty()) return b;
309     while (sz(b) < sz(a)) {
310         int n = min(sz(b) * 2, sz(a));
311         b.resize(n);
312         poly v = poly(a.begin(), a.begin() + n) - log(b);
313         v[0] = v[0] + num(1);
314         b *= v;
315         b.resize(n);
316     }
317     return b;
318 }
319
320 poly pow(const poly& a, int m) { // m >= 0
321     poly b(a.size());
322     if (!m) {
323         b[0] = 1;
324         return b;
325     }
326     int p = 0;
327     while (p < sz(a) && a[p].v == 0) ++p;
328     if (1ll * m * p >= sz(a)) return b;
329     num mu = pow(a[p], m), di = inv(a[p]);
330     poly c(sz(a) - m * p);
331     rep(i, 0, sz(c)) c[i] = a[i + p] * di;
332     c = log(c);
333     trav(v, c) v = v * m;
334     c = exp(c);
335     rep(i, 0, sz(c)) b[i + m * p] = c[i] * mu;
336     return b;
337 }
338
339 // Multipoint evaluation/interpolation
340
341 vector<num> eval(const poly& a, const vector<num>& x) {
342     int n = sz(x);
343     if (!n) return {};
344     vector<poly> up(2 * n);
345     rep(i, 0, n) up[i + n] = poly({0 - x[i], 1});
346     per(i, 1, n) up[i] = up[2 * i] * up[2 * i + 1];
347     vector<poly> down(2 * n);
348     down[1] = a % up[1];
349     rep(i, 2, 2 * n) down[i] = down[i / 2] % up[i];
350     vector<num> y(n);
351     rep(i, 0, n) y[i] = down[i + n][0];
352     return y;
353 }
354
355 poly interp(const vector<num>& x, const vector<num>& y)
356     {
357     {
358         int n = sz(x);
359         assert(n);
360         vector<poly> up(n * 2);
361         rep(i, 0, n) up[i + n] = poly({0 - x[i], 1});
362         per(i, 1, n) up[i] = up[2 * i] * up[2 * i + 1];
363         vector<num> a = eval(deriv(up[1]), x);
364         vector<poly> down(2 * n);
365         rep(i, 0, n) down[i + n] = poly({y[i] * inv(a[i])});
366         per(i, 1, n) down[i] =
367             down[i * 2] * up[i * 2 + 1] + down[i * 2 + 1] * up[i
368             * 2];
369         return down[1];
370     }
371 }

```



```

348 }
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000

```

Simplex method for linear programs

- Maximize $c^T x$ subject to $Ax \leq b$, $x \geq 0$.
- Returns $-\infty$ if there is no solution, $+\infty$ if there are arbitrarily good solutions, or the maximum value of $c^T x$ otherwise. The (arbitrary) input vector is set to an optimal x (or in the unbounded case, an arbitrary solution fulfilling the constraints). Numerical stability is not guaranteed. For better performance, define variables such that $x = 0$ is viable.
- Complexity: $O(NM \cdot pivots)$. $O(2^n)$ in general (very hard to achieve).

```

35 int r = -1;
36 rep(i,0,m) {
37     if (D[i][s] <= eps) continue;
38     if (r == -1 || MP(D[i][n+1] / D[i][s], B[i]) <
39         MP(D[r][n+1] / D[r][s], B[r])) r = i;
40 }
41 if (r == -1) return false;
42 pivot(r, s);
43 }
44 T solve(vd &x){
45     int r = 0;
46     rep(i,1,m) if (D[i][n+1] < D[r][n+1]) r = i;
47     if (D[r][n+1] < -eps) {
48         pivot(r, n);
49         if (!simplex(2) || D[m+1][n+1] < -eps) return
50         -inf;
51     }
52     rep(i,0,m) if (B[i] == -1) {
53         int s = 0;
54         rep(j,1,n+1) ltj(D[i]);
55         pivot(i, s);
56     }
57 }
58 bool ok = simplex(1); x = vd(n);
59 rep(i,0,m) if (B[i] < n) x[B[i]] = D[i][n+1];
60 return ok ? D[m][n+1] : inf;
61 }
62 };

```

Data Structures

Fenwick Tree

```

11 sum(int r) {
12     ll ret = 0;
13     for (; r >= 0; r = (r & r + 1) - 1) ret += bit[r];
14     return ret;
15 }
16 void add(int idx, ll delta) {
17     for (; idx < n; idx |= idx + 1) bit[idx] += delta;
18 }

```

Lazy Propagation SegTree

```

1 // Clear: clear() or build()
2 const int N = 2e5 + 10; // Change the constant!
3 template<typename T>
4 struct LazySegTree{
5     T t[4 * N];
6     T lazy[4 * N];
7     int n;
8
9     // Change these functions, default return, and lazy
10    mark.
11    T default_return = 0, lazy_mark =
12    numeric_limits<T>::min();

```

```

13 // Lazy mark is how the algorithm will identify that
14 // no propagation is needed.
15 function<T(T, T)> f = [&] (T a, T b){
16     return a + b;
17 };
18 // f_on_seg calculates the function f, knowing the
19 // lazy value on segment,
20 // segment's size and the previous value.
21 // The default is segment modification for RSQ. For
22 // increments change to:
23 // return cur_seg_val + seg_size * lazy_val;
24 // For RMQ. Modification: return lazy_val;
25 // Increments: return cur_seg_val + lazy_val;
26 function<T(T, int, T)> f_on_seg = [&] (T cur_seg_val,
27     int seg_size, T lazy_val){
28     return seg_size * lazy_val;
29 };
30 // upd_lazy updates the value to be propagated to
31 // child segments.
32 // Default: modification. For increments change to:
33 // lazy[v] = (lazy[v] == lazy_mark? val : lazy[v]
34 // + val);
35 function<void(int, T)> upd_lazy = [&] (int v, T val){
36     lazy[v] = val;
37 };
38 // Tip: for "get element on single index" queries, use
39 // max() on segment: no overflows.
40
41 LazySegTree(int n_) : n(n_) {
42     clear(n);
43 }
44
45 void build(int v, int tl, int tr, vector<T>& a){
46     if (tl == tr) {
47         t[v] = a[tl];
48         return;
49     }
50     int tm = (tl + tr) / 2;
51     // left child: [tl, tm]
52     // right child: [tm + 1, tr]
53     build(2 * v + 1, tl, tm, a);
54     build(2 * v + 2, tm + 1, tr, a);
55     t[v] = f(t[2 * v + 1], t[2 * v + 2]);
56 }
57
58 LazySegTree(vector<T>& a){
59     build(a);
60 }
61
62 void push(int v, int tl, int tr){
63     if (lazy[v] == lazy_mark) return;
64     int tm = (tl + tr) / 2;
65     t[2 * v + 1] = f_on_seg(t[2 * v + 1], tm - tl + 1,
66     lazy[v]);
67     t[2 * v + 2] = f_on_seg(t[2 * v + 2], tr - tm,
68     lazy[v]);

```



```

57     upd_lazy(2 * v + 1, lazy[v]), upd_lazy(2 * v + 2,
↪ lazy[v]);
58     lazy[v] = lazy_mark;
59 }
60
61 void modify(int v, int tl, int tr, int l, int r, T
↪ val){
62     if (l > r) return;
63     if (tl == l && tr == r){
64         t[v] = f_on_seg(t[v], tr - tl + 1, val);
65         upd_lazy(v, val);
66         return;
67     }
68     push(v, tl, tr);
69     int tm = (tl + tr) / 2;
70     modify(2 * v + 1, tl, tm, l, min(r, tm), val);
71     modify(2 * v + 2, tm + 1, tr, max(l, tm + 1), r,
↪ val);
72     t[v] = f(t[2 * v + 1], t[2 * v + 2]);
73 }
74
75 T query(int v, int tl, int tr, int l, int r) {
76     if (l > r) return default_return;
77     if (tl == l && tr == r) return t[v];
78     push(v, tl, tr);
79     int tm = (tl + tr) / 2;
80     return f(
81         query(2 * v + 1, tl, tm, l, min(r, tm)),
82         query(2 * v + 2, tm + 1, tr, max(l, tm + 1), r)
83     );
84 }
85
86 void modify(int l, int r, T val){
87     modify(0, 0, n - 1, l, r, val);
88 }
89
90 T query(int l, int r){
91     return query(0, 0, n - 1, l, r);
92 }
93
94 T get(int pos){
95     return query(pos, pos);
96 }
97
98 // Change clear() function to t.clear() if using
↪ unordered_map for SegTree!!!
99 void clear(int n_){
100     n = n_;
101     for (int i = 0; i < 4 * n; i++) t[i] = 0, lazy[i] = 0;
↪ lazy_mark;
102 }
103
104 void build(vector<T>& a){
105     n = sz(a);
106     clear(n);
107     build(0, 0, n - 1, a);
108 }
109 };

```

Sparse Table

```

1  const int N = 2e5 + 10, LOG = 20; // Change the
↪ constant!
2  template<typename T>
3  struct SparseTable{
4      int lg[N];
5      T st[N][LOG];
6      int n;
7
8      // Change this function
9      function<T(T, T)> f = [&] (T a, T b){
10         return min(a, b);
11     };
12
13     void build(vector<T>& a){
14         n = sz(a);
15         lg[1] = 0;
16         for (int i = 2; i <= n; i++) lg[i] = lg[i / 2] + 1;
17
18         for (int k = 0; k < LOG; k++){
19             for (int i = 0; i < n; i++){
20                 if (!k) st[i][k] = a[i];
21                 else st[i][k] = f(st[i][k - 1], st[min(n - 1, i +
↪ (1 << (k - 1))))[k - 1]);
22             }
23         }
24     }
25
26     T query(int l, int r){
27         int sz = r - l + 1;
28         return f(st[l][lg[sz]], st[r - (1 << lg[sz]) +
↪ 1][lg[sz]]);
29     }
30 };

```

Suffix Array and LCP array

- (uses SparseTable above)

```

1  struct SuffixArray{
2      vector<int> p, c, h;
3      SparseTable<int> st;
4      /*
5       In the end, array c gives the position of each suffix
↪ in p
6       using 1-based indexation!
7       */
8
9      SuffixArray() {}
10
11     SuffixArray(string s){
12         buildArray(s);
13         buildLCP(s);
14         buildSparse();
15     }
16
17     void buildArray(string s){

```

```

18     int n = sz(s) + 1;
19     p.resize(n), c.resize(n);
20     for (int i = 0; i < n; i++) p[i] = i;
21     sort(all(p), [&] (int a, int b){return s[a] <
↪ s[b];});
22     c[p[0]] = 0;
23     for (int i = 1; i < n; i++){
24         c[p[i]] = c[p[i - 1]] + (s[p[i]] != s[p[i - 1]]);
25     }
26     vector<int> p2(n), c2(n);
27     // w is half-length of each string.
28     for (int w = 1; w < n; w <= 1){
29         for (int i = 0; i < n; i++){
30             p2[i] = (p[i] - w + n) % n;
31         }
32         vector<int> cnt(n);
33         for (auto i : c) cnt[i]++;
34         for (int i = 1; i < n; i++) cnt[i] += cnt[i - 1];
35         for (int i = n - 1; i >= 0; i--){
36             p[--cnt[c[p2[i]]]] = p2[i];
37         }
38         c2[p[0]] = 0;
39         for (int i = 1; i < n; i++){
40             c2[p[i]] = c2[p[i - 1]] +
41                 (c[p[i]] != c[p[i - 1]] ||
42                  c[(p[i] + w) % n] != c[(p[i - 1] + w) % n]);
43         }
44         c.swap(c2);
45     }
46     p.erase(p.begin());
47 }
48
49 void buildLCP(string s){
50     // The algorithm assumes that suffix array is
↪ already built on the same string.
51     int n = sz(s);
52     h.resize(n - 1);
53     int k = 0;
54     for (int i = 0; i < n; i++){
55         if (c[i] == n){
56             k = 0;
57             continue;
58         }
59         int j = p[c[i]];
60         while (i + k < n && j + k < n && s[i + k] == s[j +
↪ k]) k++;
61         h[c[i] - 1] = k;
62         if (k) k--;
63     }
64     /*
65     Then an RMQ Sparse Table can be built on array h
66     to calculate LCP of 2 non-consecutive suffixes.
67     */
68 }

```

```

70 void buildSparse(){
71     st.build(h);
72 }
73
74 // l and r must be in 0-BASED INDEXATION
75 int lcp(int l, int r){
76     l = c[l] - 1, r = c[r] - 1;
77     if (l > r) swap(l, r);
78     return st.query(l, r - 1);
79 }
80 };

```

Aho Corasick Trie

- For each node in the trie, the suffix link points to the longest proper suffix of the represented string. The terminal-link tree has square-root height (can be constructed by DFS).

```

1  const int S = 26;
2
3  // Function converting char to int.
4  int ctoi(char c){
5      return c - 'a';
6  }
7
8  // To add terminal links, use DFS
9  struct Node{
10     vector<int> nxt;
11     int link;
12     bool terminal;
13
14     Node() {
15         nxt.assign(S, -1), link = 0, terminal = 0;
16     }
17 };
18
19 vector<Node> trie(1);
20
21 // add_string returns the terminal vertex.
22 int add_string(string& s){
23     int v = 0;
24     for (auto c : s){
25         int cur = ctoi(c);
26         if (trie[v].nxt[cur] == -1){
27             trie[v].nxt[cur] = sz(trie);
28             trie.emplace_back();
29         }
30         v = trie[v].nxt[cur];
31     }
32     trie[v].terminal = 1;
33     return v;
34 }
35
36 /*
37 Suffix links are compressed.
38 This means that:

```

```

39 If vertex v has a child by letter x, then:
40     trie[v].nxt[x] points to that child.
41 If vertex v doesn't have such child, then:
42     trie[v].nxt[x] points to the suffix link of that
    ↪ child
43     if we would actually have it.
44 */
45 void add_links(){
46     queue<int> q;
47     q.push(0);
48     while (!q.empty()){
49         auto v = q.front();
50         int u = trie[v].link;
51         q.pop();
52         for (int i = 0; i < S; i++){
53             int& ch = trie[v].nxt[i];
54             if (ch == -1){
55                 ch = v? trie[u].nxt[i] : 0;
56             }
57             else{
58                 trie[ch].link = v? trie[u].nxt[i] : 0;
59                 q.push(ch);
60             }
61         }
62     }
63 }
64
65 bool is_terminal(int v){
66     return trie[v].terminal;
67 }
68
69 int get_link(int v){
70     return trie[v].link;
71 }
72
73 int go(int v, char c){
74     return trie[v].nxt[ctoi(c)];
75 }

```

Convex Hull Trick

- Allows to insert a linear function to the hull in (1) and get the minimum/maximum value of the stored function at a point in $O(\log n)$.
- NOTE: The lines must be added in the order of decreasing/increasing gradients. CAREFULLY CHECK THE SETUP BEFORE USING!
- IMPORTANT: THE DEFAULT VERSION SURELY WORKS. IF MODIFIED VERSIONS DON'T WORK, TRY TRANSFORMING THEM TO THE DEFAULT ONE BY CHANGING SIGNS.

```

1 struct line{
2     ll k, b;

```

```

3     ll f(ll x){
4         return k * x + b;
5     };
6 };
7
8 vector<line> hull;
9
10 void add_line(line nl){
11     if (!hull.empty() && hull.back().k == nl.k){
12         nl.b = min(nl.b, hull.back().b); // Default:
    ↪ minimum. For maximum change "min" to "max".
13         hull.pop_back();
14     }
15     while (sz(hull) > 1){
16         auto& l1 = hull.end()[-2], l2 = hull.back();
17         if ((l1.b - l2.b) * (l2.k - nl.k) >= (nl.b - l2.b) *
    ↪ (l1.k - nl.k)) hull.pop_back(); // Default:
    ↪ decreasing gradient k. For increasing k change the
    ↪ sign to <=.
18         else break;
19     }
20     hull.pb(nl);
21 }
22
23 ll get(ll x){
24     int l = 0, r = sz(hull);
25     while (r - l > 1){
26         int mid = (l + r) / 2;
27         if (hull[mid - 1].f(x) >= hull[mid].f(x)) l = mid;
    ↪ // Default: minimum. For maximum change the sign to
    ↪ <=.
28         else r = mid;
29     }
30     return hull[l].f(x);
31 }

```

Li-Chao Segment Tree

- allows to add linear functions in any order and query minimum/maximum value of those at a point, all in $O(\log n)$.
- Clear: clear()

```

const ll INF = 1e18; // Change the constant!
struct LiChaoTree{
    struct line{
        ll k, b;
        line(){
            k = b = 0;
        };
        line(ll k_, ll b_){
            k = k_, b = b_;
        };
        ll f(ll x){
            return k * x + b;
        };
    };

```

```

14   };
15   int n;
16   bool minimum, on_points;
17   vector<ll> pts;
18   vector<line> t;
19
20   void clear(){
21       for (auto& l : t) l.k = 0, l.b = minimum? INF :
↳ -INF;
22   }
23
24   LiChaoTree(int n_, bool min_){ // This is a default
↳ constructor for numbers in range [0, n - 1].
25       n = n_, minimum = min_, on_points = false;
26       t.resize(4 * n);
27       clear();
28   };
29
30   LiChaoTree(vector<ll> pts_, bool min_){ // This
↳ constructor will build LCT on the set of points you
↳ pass. The points may be in any order and contain
↳ duplicates.
31       pts = pts_, minimum = min_;
32       sort(all(pts));
33       pts.erase(unique(all(pts)), pts.end());
34       on_points = true;
35       n = sz(pts);
36       t.resize(4 * n);
37       clear();
38   };
39
40   void add_line(int v, int l, int r, line nl){
41       // Adding on segment [l, r]
42       int m = (l + r) / 2;
43       ll lval = on_points? pts[l] : l, mval = on_points?
↳ pts[m] : m;
44       if ((minimum && nl.f(mval) < t[v].f(mval)) ||
↳ (!minimum && nl.f(mval) > t[v].f(mval))) swap(t[v]
↳ nl);
45       if (r - l == 1) return;
46       if ((minimum && nl.f(lval) < t[v].f(lval)) ||
↳ (!minimum && nl.f(lval) > t[v].f(lval))) add_line(2
↳ * v + 1, l, m, nl);
47       else add_line(2 * v + 2, m, r, nl);
48   }
49
50   ll get(int v, int l, int r, int x){
51       int m = (l + r) / 2;
52       if (r - l == 1) return t[v].f(on_points? pts[x] :
↳ x);
53       else{
54           if (minimum) return min(t[v].f(on_points? pts[x] :
↳ x), x < m? get(2 * v + 1, l, m, x) : get(2 * v + 2,
↳ m, r, x));
55           else return max(t[v].f(on_points? pts[x] : x), x <
↳ m? get(2 * v + 1, l, m, x) : get(2 * v + 2, m, r,
↳ x));
56       }

```

```

57   }
58
59   void add_line(ll k, ll b){
60       add_line(0, 0, n, line(k, b));
61   }
62
63   ll get(ll x){
64       return get(0, 0, n, on_points? lower_bound(all(pts),
↳ x) - pts.begin() : x);
65   }; // Always pass the actual value of x, even if LCT
↳ is on points.
66   };

```

Persistent Segment Tree

- for RSQ

```

1   struct Node {
2       ll val;
3       Node *l, *r;
4
5       Node(ll x) : val(x), l(nullptr), r(nullptr) {}
6       Node(Node *ll, Node *rr) {
7           l = ll, r = rr;
8           val = 0;
9           if (l) val += l->val;
10          if (r) val += r->val;
11      }
12      Node(Node *cp) : val(cp->val), l(cp->l), r(cp->r) {}
13  };
14  const int N = 2e5 + 20;
15  ll a[N];
16  Node *roots[N];
17  int n, cnt = 1;
18  Node *build(int l = 1, int r = n) {
19      if (l == r) return new Node(a[l]);
20      int mid = (l + r) / 2;
21      return new Node(build(l, mid), build(mid + 1, r));
22  }
23  Node *update(Node *node, int val, int pos, int l = 1,
↳ int r = n) {
24      if (l == r) return new Node(val);
25      int mid = (l + r) / 2;
26      if (pos > mid)
27          return new Node(node->l, update(node->r, val,
↳ pos, mid + 1, r));
28      else return new Node(update(node->l, val, pos, l,
↳ mid), node->r);
29  }
30  ll query(Node *node, int a, int b, int l = 1, int r = n)
↳ {
31      if (l > b || r < a) return 0;
32      if (l >= a && r <= b) return node->val;
33      int mid = (l + r) / 2;
34      return query(node->l, a, b, l, mid) + query(node->r,
↳ a, b, mid + 1, r);
35  }

```

Miscellaneous

Ordered Set

```

1   #include <ext/pb_ds/assoc_container.hpp>
2   #include <ext/pb_ds/tree_policy.hpp>
3   using namespace __gnu_pbds;
4   typedef tree<int, null_type, less<int>, rb_tree_tag,
↳ tree_order_statistics_node_update> ordered_set;

```

Measuring Execution Time

```

1   ld tic = clock();
2   // execute algo...
3   ld tac = clock();
4   // Time in milliseconds
5   cerr << (tac - tic) / CLOCKS_PER_SEC * 1000 << endl;
6   // No need to comment out the print because it's done to
↳ cerr.

```

Setting Fixed D.P. Precision

```

1   cout << setprecision(d) << fixed;
2   // Each number is rounded to d digits after the decimal
↳ point, and truncated.

```

Common Bugs and General Advice

- Check overflow, array bounds
- Check variable overloading
- Check special cases (n=1?)
- Do something instead of nothing, stay organized
- Write stuff down!
- Don't get stuck on one approach!

Dynamic Programming

Sum over Subset DP

- Computes $f[A] = \sum_{B \subseteq A} a[B]$.
- Complexity: $O(2^n \cdot n)$.

```

1   for (int i = 0; i < (1 << n); i++) f[i] = a[i];
2   for (int i = 0; i < n; i++) for (int mask = 0; mask < (1
↳ << n); mask++) if ((mask >> i) & 1){
3       f[mask] += f[mask ^ (1 << i)];
4   }

```

Divide and Conquer DP

- Helps to compute 2D DP of the form:
- $dp[i][j] = \min_{0 \leq k \leq j-1} (dp[i-1][k] + cost(k+1, j))$
- **Necessary condition:** let $opt(i, j)$ be the optimal k for the state (i, j) . Then, $opt(i, j) \leq opt(i, j+1)$.
- Complexity: $O(M \cdot N \cdot \log N)$ for computing $dp[M][N]$.

```
1 vector<ll> dp_old(N), dp_new(N);
2
3 void rec(int l, int r, int optl, int optr){
4     if (l > r) return;
5     int mid = (l + r) / 2;
6     pair<ll, int> best = {INF, optl};
7     for (int i = optl; i <= min(mid - 1, optr); i++){ //
8         ↪ If k can be j, change to "i <= min(mid, optr)".
9         ll cur = dp_old[i] + cost(i + 1, mid);
10        if (cur < best.fi) best = {cur, i};
11    }
12    dp_new[mid] = best.fi;
13
14    rec(l, mid - 1, optl, best.se);
15    rec(mid + 1, r, best.se, optr);
16 }
17 // Computes the DP "by layers"
18 fill(all(dp_old), INF);
19 dp_old[0] = 0;
20 while (layers--){
21     rec(0, n, 0, n);
22     dp_old = dp_new;
23 }
```