



Notebook - Maratona de Programação

Cabo HDMI, VGA, USB

Contents

1 String	2	8.8 Bellman Ford	13
1.1 String hash	2	8.9 2SAT	13
1.2 Z function	2	9 Parser	14
1.3 Suffix Array	2	9.1 Parsing Functions	14
2 Math	3		
2.1 Fast Fourier Transform	3		
3 Segtree	4		
3.1 Set and update lazy seg	4		
3.2 Standard SegTree	4		
3.3 Lazy SegTree	5		
3.4 Persistent sum segment tree	5		
4 Algorithms	6		
4.1 Sparse table	6		
5 Set	6		
5.1 Ordered Set	6		
5.2 Multiset	6		
5.3 Set	7		
6 Misc	7		
6.1 Template	7		
7 Geometry	7		
7.1 Convex Hull	7		
7.2 Lattice Points	8		
7.3 Line Intersegment	8		
8 Graph	8		
8.1 Dominator tree	8		
8.2 Topological Sort	9		
8.3 Kth Ancestor	9		
8.4 Dfs tree	10		
8.5 Dkistra	10		
8.6 Dinic	10		
8.7 Dinic Min cost	11		

1 String

1.1 String hash

Complexity: $O(n)$ preprocessing, $O(1)$ query

Computes the hash of arbitrary substrings of a given string s .

```
1 // TITLE: String hash
2 // COMPLEXITY:  $O(n)$  preprocessing,  $O(1)$  query
3 // DESCRIPTION: Computes the hash of arbitrary
  substrings of a given string  $s$ .
4
5 struct hashes
6 {
7     string s;
8     int m1, m2, n, p;
9     vector<int> p1, p2, sum1, sum2;
10
11     hashes(string s) : s(s), n(s.size()), p1(n + 1),
12                       p2(n + 1), sum1(n + 1), sum2(n + 1)
13     {
14         srand(time(0));
15         p = 31;
16         m1 = rand() / 10 + 1e9; // 1000253887;
17         m2 = rand() / 10 + 1e9; // 1000546873;
18
19         p1[0] = p2[0] = 1;
20         rep(i, 1, n + 1)
21         {
22             p1[i] = (p * p1[i - 1]) % m1;
23             p2[i] = (p * p2[i - 1]) % m2;
24         }
25
26         sum1[0] = sum2[0] = 0;
27         rep(i, 1, n + 1)
28         {
29             sum1[i] = (sum1[i - 1] * p) % m1 + s[i -
30             1];
31             sum2[i] = (sum2[i - 1] * p) % m2 + s[i -
32             1];
33             sum1[i] %= m1;
34             sum2[i] %= m2;
35         }
36
37         // hash do intervalo [l, r)
38         int gethash(int l, int r)
39         {
40             int c1 = m1 - (sum1[l] * p1[r - 1]) % m1;
41             int c2 = m2 - (sum2[l] * p2[r - 1]) % m2;
42             int h1 = (sum1[r] + c1) % m1;
43             int h2 = (sum2[r] + c2) % m2;
44             return (h1 << 30) ^ h2;
45         }
46     };
47 }
```

1.2 Z function

Complexity: Z function complexity

z function

```
1 // TITLE: Z function
2 // COMPLEXITY: Z function complexity
3 // DESCRIPTION: z function
4
5 void z_function(string& s)
6 {
7     return;
8 }
```

1.3 Suffix Array

Complexity: $O(n \log(n))$, contains big constant (around 25).

Computes a sorted array of the suffixes of a string.

```
1 // TITLE: Suffix Array
2 // COMPLEXITY:  $O(n \log(n))$ , contains big constant (
  around 25).
3 // DESCRIPTION: Computes a sorted array of the
  suffixes of a string.
4
5 void countingsort(vi& p, vi& c) {
6     int n=p.size();
7     vi count(n,0);
8     rep(i,0,n) count[c[i]]++;
9
10    vi psum(n); psum[0]=0;
11    rep(i,1,n) psum[i]=psum[i-1]+count[i-1];
12
13    vi ans(n);
14    rep(i,0,n)
15        ans[psum[c[p[i]]]]+=p[i];
16
17    p = ans;
18 }
19
20 vi sfa(string s) {
21     s += "$";
22
23     int n=s.size();
24     vi p(n);
25     vi c(n);
26     {
27         vector<pair<char, int>> a(n);
28         rep(i,0,n) a[i]={s[i],i};
29         sort(all(a));
30
31         rep(i,0,n) p[i]=a[i].second;
32
33         c[p[0]]=0;
34         rep(i,1,n) {
35             if(s[p[i]] == s[p[i-1]]) {
36                 c[p[i]]=c[p[i-1]];
37             }
38             else c[p[i]]=c[p[i-1]]+1;
39         }
40     }
41
42     for(int k=0; (1<<k) < n; k++) {
43         rep(i, 0, n)
44             p[i] = (p[i] - (1<<k) + n) % n;
45
46         countingsort(p,c);
47
48         vi nc(n);
49         nc[p[0]]=0;
50         rep(i,1,n) {
51             pii prev = {c[p[i-1]], c[(p[i-1]+(1<<k))%
52             n]};
53             pii cur = {c[p[i]], c[(p[i]+(1<<k))%n]};
54
55             if (prev == cur)
56                 nc[p[i]]=nc[p[i-1]];
57             else nc[p[i]]=nc[p[i-1]]+1;
58         }
59         c=nc;
60     }
61     return p;
62 }
```

2 Math

2.1 Fast Fourier Transform

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

Multiply polynomials quickly

```
1 // TITLE: Fast Fourier Transform
2 // COMPLEXITY:  $O(n \log(n))$ 
3 // DESCRIPTION: Multiply polynomials quickly
4
5 typedef double ld;
6 typedef long long ll;
7
8 struct num{
9     ld x, y;
10    num() { x = y = 0; }
11    num(ld x, ld y) : x(x), y(y) {}
12 };
13
14 inline num operator+(num a, num b) { return num(a.x +
15     b.x, a.y + b.y); }
16 inline num operator-(num a, num b) { return num(a.x -
17     b.x, a.y - b.y); }
18 inline num operator*(num a, num b) { return num(a.x *
19     b.x - a.y * b.y, a.x * b.y + a.y * b.x); }
20 inline num conj(num a) { return num(a.x, -a.y); }
21
22 int base = 1;
23 vector<num> roots = {{0, 0}, {1, 0}};
24 vector<int> rev = {0, 1};
25 const ld PI = acos(-1);
26
27 void ensure_base(int nbase){
28     if(nbase <= base)
29         return;
30
31     rev.resize(1 << nbase);
32     for(int i = 0; i < (1 << nbase); i++)
33         rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (
34             nbase - 1));
35
36     roots.resize(1 << nbase);
37
38     while(base < nbase){
39         ld angle = 2*PI / (1 << (base + 1));
40         for(int i = 1 << (base - 1); i < (1 << base);
41             i++){
42             roots[i << 1] = roots[i];
43             ld angle_i = angle * (2 * i + 1 - (1 <<
44                 base));
45             roots[(i << 1) + 1] = num(cos(angle_i),
46                 sin(angle_i));
47         }
48         base++;
49     }
50 }
51
52 void fft(vector<num> &a, int n = -1){
53     if(n == -1)
54         n = a.size();
55
56     assert((n & (n-1)) == 0);
57     int zeros = __builtin_ctz(n);
58     ensure_base(zeros);
59     int shift = base - zeros;
60     for(int i = 0; i < n; i++)
61         if(i < (rev[i] >> shift))
62             swap(a[i], a[rev[i] >> shift]);
63
64     for(int k = 1; k < n; k <= 1)
65         for(int i = 0; i < n; i += 2 * k)
66             for(int j = 0; j < k; j++){
```

```
67         num z = a[i+j+k] * roots[j+k];
68         a[i+j+k] = a[i+j] - z;
69         a[i+j] = a[i+j] + z;
70     }
71 }
72
73 vector<num> fa, fb;
74 vector<ll> multiply(vector<ll> &a, vector<ll> &b){
75     int need = a.size() + b.size() - 1;
76     int nbase = 0;
77     while((1 << nbase) < need) nbase++;
78     ensure_base(nbase);
79     int sz = 1 << nbase;
80     if(sz > (int) fa.size())
81         fa.resize(sz);
82
83     for(int i = 0; i < sz; i++){
84         int x = (i < (int) a.size() ? a[i] : 0);
85         int y = (i < (int) b.size() ? b[i] : 0);
86         fa[i] = num(x, y);
87     }
88     fft(fa, sz);
89     num r(0, -0.25 / sz);
90     for(int i = 0; i <= (sz >> 1); i++){
91         int j = (sz - i) & (sz - 1);
92         num z = (fa[j] * fa[j] - conj(fa[i] * fa[i]))
93             * r;
94         if(i != j) {
95             fa[j] = (fa[i] * fa[i] - conj(fa[j] * fa[
96                 j])) * r;
97         }
98         fa[i] = z;
99     }
100    fft(fa, sz);
101    vector<ll> res(need);
102    for(int i = 0; i < need; i++)
103        res[i] = round(fa[i].x);
104
105    return res;
106 }
107
108 vector<ll> multiply_mod(vector<ll> &a, vector<ll> &b,
109     int m, int eq = 0){
110     int need = a.size() + b.size() - 1;
111     int nbase = 0;
112     while((1 << nbase) < need) nbase++;
113     ensure_base(nbase);
114     int sz = 1 << nbase;
115     if(sz > (int) fa.size())
116         fa.resize(sz);
117
118     for(int i=0;i<(int)a.size();i++){
119         int x = (a[i] % m + m) % m;
120         fa[i] = num(x & ((1 << 15) - 1), x >> 15);
121     }
122     fill(fa.begin() + a.size(), fa.begin() + sz, num
123         {0, 0});
124     fft(fa, sz);
125     if(sz > (int) fb.size())
126         fb.resize(sz);
127     if(eq)
128         copy(fa.begin(), fa.begin() + sz, fb.begin())
129         ;
130     else{
131         for(int i = 0; i < (int) b.size(); i++){
132             int x = (b[i] % m + m) % m;
133             fb[i] = num(x & ((1 << 15) - 1), x >> 15)
134             ;
135         }
136         fill(fb.begin() + b.size(), fb.begin() + sz,
137             num {0, 0});
138         fft(fb, sz);
```

```

126     }
127     ld ratio = 0.25 / sz;
128     num r2(0, -1);
129     num r3(ratio, 0);
130     num r4(0, -ratio);
131     num r5(0, 1);
132     for(int i=0;i<=(sz >> 1);i++) {
133         int j = (sz - i) & (sz - 1);
134         num a1 = (fa[i] + conj(fa[j]));
135         num a2 = (fa[i] - conj(fa[j])) * r2;
136         num b1 = (fb[i] + conj(fb[j])) * r3;
137         num b2 = (fb[i] - conj(fb[j])) * r4;
138         if(i != j){
139             num c1 = (fa[j] + conj(fa[i]));
140             num c2 = (fa[j] - conj(fa[i])) * r2;
141             num d1 = (fb[j] + conj(fb[i])) * r3;
142             num d2 = (fb[j] - conj(fb[i])) * r4;
143             fa[i] = c1 * d1 + c2 * d2 * r5;
144             fb[i] = c1 * d2 + c2 * d1;
145         }
146         fa[j] = a1 * b1 + a2 * b2 * r5;
147         fb[j] = a1 * b2 + a2 * b1;
148     }
149     fft(fa, sz);
150     fft(fb, sz);
151     vector<ll> res(need);
152     for(int i=0;i<need;i++){
153         ll aa = round(fa[i].x);
154         ll bb = round(fb[i].x);
155         ll cc = round(fa[i].y);
156         res[i] = (aa + ((bb % m) << 15) + ((cc % m)
157 << 30)) % m;
158     }
159     return res;
160 }

```

3 Segtree

3.1 Set and update lazy seg

Complexity: $O(\log(n))$ query and update
Sum segtree with set and update

```

1 // TITLE: Set and update lazy seg
2 // COMPLEXITY:  $O(\log(n))$  query and update
3 // DESCRIPTION: Sum segtree with set and update
4
5 vector<int> lazy, opvec;
6 vector<int> seg;
7
8 constexpr int SET = 30;
9 constexpr int ADD = 31;
10
11 int segsize;
12
13 void propagate(int no, int lx, int rx) {
14     if (lazy[no] == -1) return;
15
16     if (rx-lx == 1) {
17         if(opvec[no] == SET) seg[no] = lazy[no];
18         else seg[no] += lazy[no];
19
20         lazy[no]=-1;
21         opvec[no]=-1;
22         return;
23     }
24
25     if(opvec[no] == SET) {
26         seg[no] = (rx-lx) * lazy[no];
27         lazy[2*no+1] = lazy[no];

```

```

28         lazy[2*no+2] = lazy[no];
29
30         opvec[2*no+1] = SET;
31         opvec[2*no+2] = SET;
32
33         lazy[no] = -1;
34         opvec[no]=-1;
35         return;
36     }
37
38     seg[no] += (rx-lx) * lazy[no];
39     if (lazy[2*no+1] == -1) {
40         lazy[2*no+1] = 0;
41         opvec[2*no+1] = ADD;
42     }
43     if (lazy[2*no+2] == -1) {
44         lazy[2*no+2] = 0;
45         opvec[2*no+2] = ADD;
46     }
47     lazy[2*no+1] += lazy[no];
48     lazy[2*no+2] += lazy[no];
49
50     lazy[no] = -1;
51     opvec[no]=-1;
52 }
53
54 void update(int l, int r, int val, int op, int no=0,
55 int lx=0, int rx=segsize) {
56     propagate(no, lx, rx);
57     if (r <= lx or l >= rx) return;
58     if (lx >= l and rx <= r) {
59         lazy[no] = val;
60         opvec[no] = op;
61         propagate(no, lx, rx);
62         return;
63     }
64
65     int mid = (rx+lx)/2;
66     update(l, r, val, op, 2*no+1, lx, mid);
67     update(l, r, val, op, 2*no+2, mid, rx);
68     seg[no] = seg[2*no+1]+seg[2*no+2];
69
70 int query(int l, int r, int no=0, int lx=0, int rx=
71 segsize) {
72     propagate(no, lx, rx);
73     if (r <= lx or l >= rx) return 0;
74     if (lx >= l and rx <= r) return seg[no];
75
76     int mid = (rx+lx)/2;
77     return
78         query(l,r,2*no+1,lx,mid) +
79         query(l,r,2*no+2, mid, rx);

```

3.2 Standard SegTree

Complexity: $O(\log(n))$ query and update
Sum segment tree with point update.

```

1 // TITLE: Standard SegTree
2 // COMPLEXITY:  $O(\log(n))$  query and update
3 // DESCRIPTION: Sum segment tree with point update.
4
5 using type = int;
6
7 type iden = 0;
8 vector<type> seg;
9 int segsize;
10
11 type func(type a, type b)
12 {

```

```

13     return a + b;
14 }
15
16 // query do intervalo [l, r)
17 type query(int l, int r, int no = 0, int lx = 0, int
    rx = segsize)
18 {
19     // l lx rx r
20     if (r <= lx or rx <= l)
21         return iden;
22     if (l <= lx and rx <= r)
23         return seg[no];
24
25     int mid = lx + (rx - lx) / 2;
26     return func(query(l, r, 2 * no + 1, lx, mid),
27                 query(l, r, 2 * no + 2, mid, rx));
28 }
29
30 void update(int dest, type val, int no = 0, int lx =
    0, int rx = segsize)
31 {
32     if (dest < lx or dest >= rx)
33         return;
34     if (rx - lx == 1)
35     {
36         seg[no] = val;
37         return;
38     }
39
40     int mid = lx + (rx - lx) / 2;
41     update(dest, val, 2 * no + 1, lx, mid);
42     update(dest, val, 2 * no + 2, mid, rx);
43     seg[no] = func(seg[2 * no + 1], seg[2 * no + 2]);
44 }
45
46 signed main()
47 {
48     ios_base::sync_with_stdio(0);
49     cin.tie(0);
50     cout.tie(0);
51     int n;
52     cin >> n;
53     segsize = n;
54     if (__builtin_popcount(n) != 1)
55     {
56         segsize = 1 + (int)log2(segsz);
57         segsize = 1 << segsize;
58     }
59     seg.assign(2 * segsize - 1, iden);
60
61     rep(i, 0, n)
62     {
63         int x;
64         cin >> x;
65         update(i, x);
66     }
67 }

```

3.3 Lazy SegTree

Complexity: $O(\log(n))$ query and update
Sum segment tree with range sum update.

```

1 // TITLE: Lazy SegTree
2 // COMPLEXITY:  $O(\log(n))$  query and update
3 // DESCRIPTION: Sum segment tree with range sum
    update.
4 vector<int> seg, lazy;
5 int segsize;
6
7 // change 0s to -1s if update is
8 // set instead of add. also,

```

```

9 // remove the +=s
10 void prop(int no, int lx, int rx) {
11     if (lazy[no] == 0) return;
12
13     seg[no] += (rx - lx) * lazy[no];
14     if (rx - lx > 1) {
15         lazy[2 * no + 1] += lazy[no];
16         lazy[2 * no + 2] += lazy[no];
17     }
18
19     lazy[no] = 0;
20 }
21
22 void update(int l, int r, int val, int no = 0, int lx = 0,
    int rx = segsize) {
23     // l r lx rx
24     prop(no, lx, rx);
25     if (r <= lx or rx <= l) return;
26     if (l <= lx and rx <= r) {
27         lazy[no] = val;
28         prop(no, lx, rx);
29         return;
30     }
31
32     int mid = lx + (rx - lx) / 2;
33     update(l, r, val, 2 * no + 1, lx, mid);
34     update(l, r, val, 2 * no + 2, mid, rx);
35     seg[no] = seg[2 * no + 1] + seg[2 * no + 2];
36 }
37
38 int query(int l, int r, int no = 0, int lx = 0, int rx =
    segsize) {
39     prop(no, lx, rx);
40     if (r <= lx or rx <= l) return 0;
41     if (l <= lx and rx <= r) return seg[no];
42
43     int mid = lx + (rx - lx) / 2;
44     return query(l, r, 2 * no + 1, lx, mid) +
45         query(l, r, 2 * no + 2, mid, rx);
46 }
47
48 signed main() {
49     ios_base::sync_with_stdio(0); cin.tie(0); cout.tie
    (0);
50
51     int n; cin >> n;
52     segsize = n;
53     if (__builtin_popcount(n) != 1) {
54         segsize = 1 + (int)log2(segsz);
55         segsize = 1 << segsize;
56     }
57
58     seg.assign(2 * segsize - 1, 0);
59     // use -1 instead of 0 if
60     // update is set instead of add
61     lazy.assign(2 * segsize - 1, 0);
62 }

```

3.4 Persistent sum segment tree

Complexity: $O(\log(n))$ query and update, $O(k \log(n))$ memory,
 n = number of elements, k = number of operations
Sum segment tree which preserves its history.

```

1 // TITLE: Persistent sum segment tree
2 // COMPLEXITY:  $O(\log(n))$  query and update,  $O(k \log(n))$ 
    memory,  $n$  = number of elements,  $k$  = number of
    operations
3 // DESCRIPTION: Sum segment tree which preserves its
    history.
4

```

```

5 int segsize;
6
7 struct node {
8     int val;
9     int lx, rx;
10    node *l=0, *r=0;
11
12    node() {}
13    node(int val, int lx, int rx, node *l, node *r) :
14        val(val), lx(lx), rx(rx), l(l), r(r) {}
15 };
16
17 node* build(vi& arr, int lx=0, int rx=segsize) {
18     if (rx - lx == 1) {
19         if (lx < (int)arr.size()) {
20             return new node(arr[lx], lx, rx, 0, 0);
21         }
22
23         return new node(0, lx, rx, 0, 0);
24     }
25
26     int mid = (lx+rx)/2;
27     auto nol = build(arr, lx, mid);
28     auto nor = build(arr, mid, rx);
29     return new node(nol->val + nor->val, lx, rx, nol,
30                     nor);
31 }
32
33 node* update(int idx, int val, node *no) {
34     if (idx < no->lx or idx >= no->rx) return no;
35     if (no->rx - no->lx == 1) {
36         return new node(val+no->val, no->lx, no->rx,
37                         no->l, no->r);
38     }
39
40     auto nol = update(idx, val, no->l);
41     auto nor = update(idx, val, no->r);
42     return new node(nol->val + nor->val, no->lx, no->rx,
43                     nol, nor);
44 }
45
46 int query(int l, int r, node *no) {
47     if (r <= no->lx or no->rx <= l) return 0;
48     if (l <= no->lx and no->rx <= r) return no->val;
49
50     return query(l, r, no->l) + query(l, r, no->r);
51 }

```

4 Algorithms

4.1 Sparse table

Complexity: $O(n \log(n))$ preprocessing, $O(1)$ query
 Computes the minimum of a half open interval.

```

1 // TITLE: Sparse table
2 // COMPLEXITY:  $O(n \log(n))$  preprocessing,  $O(1)$  query
3 // DESCRIPTION: Computes the minimum of a half open
4     interval.
5
6 struct sptable {
7     vector<vi> table;
8
9     int ilog(int x) {
10         return (__builtin_clzll(1ll) -
11                __builtin_clzll(x));
12     }
13
14     sptable(vi& vals) {
15         int n = vals.size();

```

```

14         int ln= ilog(n)+1;
15         table.assign(ln, vi(n));
16
17         rep(i,0,n) table[0][i]=vals[i];
18
19         rep(k, 1, ln) {
20             rep(i,0,n) {
21                 table[k][i] = min(table[k-1][i],
22                                   table[k-1][min(i + (1<<(k-1)), n-1)]);
23             }
24         }
25
26         // returns minimum of vals in range [a, b)
27         int getmin(int a, int b) {
28             int k = ilog(b-a);
29             return min(table[k][a], table[k][b-(1<<k)]);
30         }
31     };
32 };

```

5 Set

5.1 Ordered Set

Complexity: $\log n$
 Worst set with additional operations

```

1 // TITLE: Ordered Set
2 // COMPLEXITY:  $\log n$ 
3 // DESCRIPTION: Worst set with additional operations
4
5
6 #include <bits/extc++.h>
7 using namespace __gnu_pbds; // or pb_ds;
8 template<typename T, typename B = null_type>
9 using ordered_set = tree<T, B, less<T>, rb_tree_tag,
10                        tree_order_statistics_node_update>;
11
12 int32_t main() {
13     ordered_set<int> oset;
14
15     oset.insert(5);
16     oset.insert(1);
17     oset.insert(2);
18     // o_set = {1, 2, 5}
19     5 == *(oset.find_by_order(2)); // Like an array
20     index
21     2 == oset.order_of_key(4); // How many elements
22     are strictly less than 4
23 }

```

5.2 Multiset

Complexity: $O(\log(n))$
 Same as set but you can have multiple elements with same values

```

1 // TITLE: Multiset
2 // COMPLEXITY:  $O(\log(n))$ 
3 // DESCRIPTION: Same as set but you can have multiple
4     elements with same values
5
6 int main() {
7     multiset<int> set1;
8 }

```

5.3 Set

Complexity: Insertion $\log(n)$

Keeps elements sorted, remove duplicates, upper_bound, lower_bound, find, count

```
1 // TITLE: Set
2 // COMPLEXITY: Insertion Log(n)
3 // Description: Keeps elements sorted, remove
  duplicates, upper_bound, lower_bound, find, count
4
5 int main() {
6     set<int> set1;
7
8     set1.insert(1);      // O(log(n))
9     set1.erase(1);      // O(log(n))
10
11     set1.upper_bound(1); // O(log(n))
12     set1.lower_bound(1); // O(log(n))
13     set1.find(1);        // O(log(n))
14     set1.count(1);       // O(log(n))
15
16     set1.size();          // O(1)
17     set1.empty();        // O(1)
18
19     set1.clear()         // O(1)
20     return 0;
21 }
```

6 Misc

6.1 Template

Complexity: $O(1)$

Standard template for competitions

```
1 // TITLE: Template
2 // COMPLEXITY: O(1)
3 // DESCRIPTION: Standard template for competitions
4
5 #include <bits/stdc++.h>
6
7 #define int long long
8 #define endl '\n'
9 #define pb push_back
10 #define eb emplace_back
11 #define all(x) (x).begin(), (x).end()
12 #define rep(i, a, b) for(int i=(int)(a); i < (int)(b); i++)
13 #define debug(var) cout << #var << ": " << var << endl
14 #define pii pair<int, int>
15 #define vi vector<int>
16
17 int MAX = 2e5;
18 int MOD = 1e9 + 7;
19 int oo = 0x3f3f3f3f3f3f3f3f;
20
21 using namespace std;
22
23 void solve()
24 {
25
26 }
27
28 signed main()
29 {
30     ios_base::sync_with_stdio(0); cin.tie(0); cout.tie(0);
31     int t = 1;
```

```
32     // cin >> t;
33     while(t--) solve();
34 }
```

7 Geometry

7.1 Convex Hull

Complexity: N

Gives you the convex hull of a set of points

```
1 // TITLE: Convex Hull
2 // COMPLEXITY: N
3 // DESCRIPTION: Gives you the convex hull of a set of
  points
4
5 struct Point
6 {
7     int x, y;
8
9     void read()
10     {
11         cin >> x >> y;
12     }
13
14     Point operator- (const Point & b) const
15     {
16         Point p;
17         p.x = x - b.x;
18         p.y = y - b.y;
19         return p;
20     }
21
22     void operator-= (const Point & b)
23     {
24         x -= b.x;
25         y -= b.y;
26     }
27
28     int operator* (const Point & b) const
29     {
30         return x * b.y - b.x * y;
31     }
32
33     bool operator< (const Point & b) const
34     {
35         return make_pair(x, y) < make_pair(b.x, b.y);
36     }
37 };
38
39 int triangle(const Point & a, const Point & b, const
  Point & c)
40 {
41     return (b - a) * (c - a);
42 }
43
44 vector<Point> convex_hull(vector<Point> points)
45 {
46     vector<Point> hull;
47     sort(all(points));
48
49     for (int z = 0; z < 2; z++) {
50         int s = hull.size();
51         for (int i = 0; i < points.size(); i++) {
52             while(hull.size() >= s + 2) {
53                 auto a = hull.end()[-2];
54                 auto b = hull.end()[-1];
55                 if (triangle(a, b, points[i]) <= 0) {
56                     hull.erase(hull.end() - 1);
57                     hull.erase(hull.end() - 2);
58                 }
59             }
60             hull.push_back(points[i]);
61         }
62     }
63 }
```

```

58         break;
59     }
60     hull.pop_back();
61 }
62 hull.push_back(points[i]);
63 }
64 hull.pop_back();
65 reverse(all(points));
66 }
67 return hull;
68 }

```

7.2 Lattice Points

Complexity: N

Points with integer coordinate

```

1 // TITLE: Lattice Points
2 // COMPLEXITY: N
3 // DESCRIPTION: Points with integer coordinate
4
5 // Picks theorem
6 // A = area
7 // i = points_inside
8 // b = points in boundary including vertices
9 // A = i + b/2 - 1
10
11 void solve()
12 {
13     int n; cin >> n;
14     vector<Point> points(n);
15     for (int i = 0; i < n; i++) {
16         points[i].read();
17     }
18
19     // Calculatting points on boundary
20     int B = 0;
21     for (int i = 0; i < n; i++) {
22         int j = (i + 1) % n;
23         Point p = points[j] - points[i];
24         B += __gcd(abs(p.x), abs(p.y)); // Unsafe for 0
25     }
26     // Calculating Area
27     int a2 = 0;
28     for (int i = 0; i < n; i++) {
29         int j = (i + 1) % n;
30         a2 += points[i] * points[j];
31     }
32     a2 = abs(a2);
33     // Picks theorem
34     int I = (a2 - B + 2)/2;
35     cout << I << " " << B << endl;
36 }

```

7.3 Line Intersegment

Complexity: $O(1)$

Check if two half segments intersect with which other

```

1 // TITLE: Line Intersegment
2 // COMPLEXITY: O(1)
3 // DESCRIPTION: Check if two half segments intersect
4 // with which other
5
6 struct Point
7 {
8     int x, y;
9
10    void read()
11    {

```

```

11        cin >> x >> y;
12    }
13
14    Point operator- (const Point & b) const
15    {
16        Point p;
17        p.x = x - b.x;
18        p.y = y - b.y;
19        return p;
20    }
21
22    void operator-= (const Point & b)
23    {
24        x -= b.x;
25        y -= b.y;
26    }
27
28    int operator* (const Point & b) const
29    {
30        return x * b.y - b.x * y;
31    }
32
33 };
34
35 int triangle(const Point & a, const Point & b, const
36             Point & c)
37 {
38     return (b - a) * (c - a);
39 }
40
41 bool intersect(const Point & p1, const Point & p2,
42               const Point & p3, const Point & p4) {
43     bool ans = true;
44     int s1 = triangle(p1, p2, p3);
45     int s2 = triangle(p1, p2, p4);
46
47     if (s1 == 0 && s2 == 0) {
48         int a_min_x = min(p1.x, p2.x);
49         int a_max_x = max(p1.x, p2.x);
50         int a_min_y = min(p1.y, p2.y);
51         int a_max_y = max(p1.y, p2.y);
52
53         int b_min_x = min(p3.x, p4.x);
54         int b_max_x = max(p3.x, p4.x);
55         int b_min_y = min(p3.y, p4.y);
56         int b_max_y = max(p3.y, p4.y);
57         if (a_min_x > b_max_x || a_min_y > b_max_y) {
58             ans = false;
59         }
60         if (b_min_x > a_max_x || b_min_y > a_max_y) {
61             ans = false;
62         }
63         return ans;
64     }
65     int s3 = triangle(p3, p4, p1);
66     int s4 = triangle(p3, p4, p2);
67
68     if ((s1 < 0) && (s2 < 0)) ans = false;
69     if ((s1 > 0) && (s2 > 0)) ans = false;
70     if ((s3 < 0) && (s4 < 0)) ans = false;
71     if ((s3 > 0) && (s4 > 0)) ans = false;
72     return ans;
73 }

```

8 Graph

8.1 Dominator tree

Complexity: $O(E + V)$


```

1 // TITLE: Dominator tree
2 // COMPLEXITY: O(E + V)
3 // DESCRIPTION: Builds dominator tree
4
5 vector<int> g[mxN];
6 vector<int> S, gt[mxN], T[mxN];
7 int dsu[mxN], label[mxN];
8 int sdom[mxN], idom[mxN], id[mxN];
9 int dfs_time = 0;
10
11 vector<int> bucket[mxN];
12 vector<int> down[mxN];
13
14 void prep(int a)
15 {
16     S.pb(a);
17     id[a] = ++dfs_time;
18     label[a] = sdom[a] = dsu[a] = a;
19
20     for (auto b: g[a]) {
21         if (!id[b]) {
22             prep(b);
23             down[a].pb(b);
24         }
25         gt[b].pb(a);
26     }
27 }
28
29 int fnd(int a, int flag = 0)
30 {
31     if (a == dsu[a]) return a;
32     int p = fnd(dsu[a], 1);
33     int b = label[ dsu[a] ];
34     if (id [ sdom[b] ] < id[ sdom[ label[a] ] ]) {
35         label[a] = b;
36     }
37     dsu[a] = p;
38     return (flag ? p: label[a]);
39 }
40
41 void build_dominator_tree(int root)
42 {
43     prep(root);
44     reverse(all(S));
45
46     int w;
47     for (int a: S) {
48         for (int b: gt[a]) {
49             w = fnd(b);
50             if (id[ sdom[w] ] < id[ sdom[a] ]) {
51                 sdom[a] = sdom[w];
52             }
53         }
54         gt[a].clear();
55         if (a != root) {
56             bucket[ sdom[a] ].pb(a);
57         }
58         for (int b: bucket[a]) {
59             w = fnd(b);
60             if (sdom[w] == sdom[b]) {
61                 idom[b] = sdom[b];
62             }
63             else {
64                 idom[b] = w;
65             }
66         }
67         bucket[a].clear();
68         for (int b: down[a]) {
69             dsu[b] = a;
70         }
71         down[a].clear();
72     }
73     reverse(all(S));

```

```

74     for (int a: S) {
75         if (a != root) {
76             if (idom[a] != sdom[a]) {
77                 idom[a] = idom[ idom[a] ];
78             }
79             T[ idom[a] ].pb(a);
80         }
81     }
82     S.clear();
83 }

```

8.2 Topological Sort

Complexity: $O(N + M)$, N: Vertices, M: Arestas

Retorna no do grapho em ordem topologica, se a quantidade de nos retornada nao for igual a quantidade de nos e impossivel

```

1 // TITLE: Topological Sort
2 // COMPLEXITY: O(N + M), N: Vertices, M: Arestas
3 // DESCRIPTION: Retorna no do grapho em ordem
4 // topologica, se a quantidade de nos retornada nao
5 // for igual a quantidade de nos e impossivel
6
7 typedef vector<vector<int>> Adj_List;
8 typedef vector<int> Indegree_List; // How many nodes
9 // depend on him
10 typedef vector<int> Order_List; // The order in
11 // which the nodes appears
12
13 Order_List kahn(Adj_List adj, Indegree_List indegree)
14 {
15     queue<int> q;
16     // priority_queue<int> q; // If you want in
17     // lexicografic order
18     for (int i = 0; i < indegree.size(); i++) {
19         if (indegree[i] == 0)
20             q.push(i);
21     }
22     vector<int> order;
23
24     while (not q.empty()) {
25         auto a = q.front();
26         q.pop();
27
28         order.push_back(a);
29         for (auto b: adj[a]) {
30             indegree[b]--;
31             if (indegree[b] == 0)
32                 q.push(b);
33         }
34     }
35     return order;
36 }
37
38 int32_t main()
39 {
40     Order_List = kahn(adj, indegree);
41     if (Order_List.size() != N) {
42         cout << "IMPOSSIBLE" << endl;
43     }
44     return 0;
45 }

```

8.3 Kth Ancestor

Complexity: $O(n * \log(n))$

Preprocess, then find in $\log n$

```

1 // TITLE: Kth Ancestor
2 // COMPLEXITY: O(n * log(n))
3 // DESCRIPTION: Preprocess, then find in log n
4
5 const int LOG_N = 30;
6 int get_kth_ancestor(vector<vector<int>> & up, int v,
    int k)
7 {
8     for (int j = 0; j < LOG_N; j++) {
9         if (k & ((int)1 << j)) {
10             v = up[v][j];
11         }
12     }
13     return v;
14 }
15
16 void solve()
17 {
18     vector<vector<int>> up(n, vector<int>(LOG_N));
19
20     for (int i = 0; i < n; i++) {
21         up[i][0] = parents[i];
22         for (int j = 1; j < LOG_N; j++) {
23             up[i][j] = up[up[i][j-1]][j-1];
24         }
25     }
26     cout << get_kth_ancestor(up, x, k) << endl;
27 }
28 }

```

8.4 Dfs tree

Complexity: $O(E + V)$

```

1 // TITLE: Dfs tree
2 // COMPLEXITY: O(E + V)
3 // DESCRIPTION: Create dfs tree from graph
4
5 int desce[mxN], sobe[mxN];
6 int backedges[mxN], vis[mxN];
7 int pai[mxN], h[mxN];
8
9 void dfs(int a, int p) {
10     if(vis[a]) return;
11     pai[a] = p;
12     h[a] = h[p]+1;
13     vis[a] = 1;
14
15     for(auto b : g[a]) {
16         if (p == b) continue;
17         if (vis[b]) continue;
18         dfs(b, a);
19         backedges[a] += backedges[b];
20     }
21     for(auto b : g[a]) {
22         if(h[b] > h[a]+1)
23             desce[a]++;
24         else if(h[b] < h[a]-1)
25             sobe[a]++;
26     }
27     backedges[a] += sobe[a] - desce[a];
28 }

```

8.5 Dkistra

Complexity: $O(E + V \cdot \log(v))$

```

1 // TITLE: Dkistra

```

```

2 // COMPLEXITY: O(E + V.log(v))
3 // DESCRIPTION: Finds to shortest path from start
4
5 int dist[mxN];
6 bool vis[mxN];
7 vector<pair<int, int>> g[mxN];
8
9 void dikstra(int start)
10 {
11     fill(dist, dist + mxN, oo);
12     fill(vis, vis + mxN, 0);
13     priority_queue<pair<int, int>> q;
14     dist[start] = 0;
15     q.push({0, start});
16
17     while(!q.empty()) {
18         auto [d, a] = q.top();
19         q.pop();
20         if (vis[a]) continue;
21         vis[a] = true;
22         for (auto [b, w]: g[a]) {
23             if (dist[a] + w < dist[b]) {
24                 dist[b] = dist[a] + w;
25                 q.push({-dist[b], b});
26             }
27         }
28     }
29 }

```

8.6 Dinic

Complexity: $O(V \cdot V \cdot E)$, Bipartite is $O(\sqrt{V} \cdot E)$

Dinic

```

1 // TITLE: Dinic
2 // COMPLEXITY: O(V*V*E), Bipartite is O(sqrt(V) E)
3 // DESCRIPTION: Dinic
4
5 const int oo = 0x3f3f3f3f3f3f3f3f;
6 // Edge structure
7 struct Edge
8 {
9     int from, to;
10    int flow, capacity;
11
12    Edge(int from_, int to_, int flow_, int capacity_)
13        : from(from_), to(to_), flow(flow_), capacity
14          (capacity_)
15    {}
16 };
17 struct Dinic
18 {
19     vector<vector<int>> graph;
20     vector<Edge> edges;
21     vector<int> level;
22     int size;
23
24     Dinic(int n)
25     {
26         graph.resize(n);
27         level.resize(n);
28         size = n;
29         edges.clear();
30     }
31
32     void add_edge(int from, int to, int capacity)
33     {
34         edges.emplace_back(from, to, 0, capacity);
35         graph[from].push_back(edges.size() - 1);
36     }

```

```

37     edges.emplace_back(to, from, 0, 0);
38     graph[to].push_back(edges.size() - 1);
39 }
40
41 int get_max_flow(int source, int sink)
42 {
43     int max_flow = 0;
44     vector<int> next(size);
45     while(bfs(source, sink)) {
46         next.assign(size, 0);
47         for (int f = dfs(source, sink, next, oo); f != 0; f = dfs(source, sink, next, oo)) {
48             max_flow += f;
49         }
50     }
51     return max_flow;
52 }
53
54 bool bfs(int source, int sink)
55 {
56     level.assign(size, -1);
57     queue<int> q;
58     q.push(source);
59     level[source] = 0;
60
61     while(!q.empty()) {
62         int a = q.front();
63         q.pop();
64
65         for (int & b: graph[a]) {
66             auto edge = edges[b];
67             int cap = edge.capacity - edge.flow;
68             if (cap > 0 && level[edge.to] == -1) {
69                 level[edge.to] = level[a] + 1;
70                 q.push(edge.to);
71             }
72         }
73     }
74     return level[sink] != -1;
75 }
76
77 int dfs(int curr, int sink, vector<int> & next, int flow)
78 {
79     if (curr == sink) return flow;
80     int num_edges = graph[curr].size();
81
82     for (; next[curr] < num_edges; next[curr]++) {
83         int b = graph[curr][next[curr]];
84         auto & edge = edges[b];
85         auto & rev_edge = edges[b^1];
86
87         int cap = edge.capacity - edge.flow;
88         if (cap > 0 && (level[curr] + 1 == level[edge.to])) {
89             int bottle_neck = dfs(edge.to, sink, next, min(flow, cap));
90             if (bottle_neck > 0) {
91                 edge.flow += bottle_neck;
92                 rev_edge.flow -= bottle_neck;
93                 return bottle_neck;
94             }
95         }
96     }
97     return 0;
98 }
99
100 vector<pair<int, int>> mincut(int source, int sink)
101 {
102     vector<pair<int, int>> cut;
103     bfs(source, sink);
104     for (auto & e: edges) {
105         if (e.flow == e.capacity && level[e.from] != -1 && level[e.to] == -1 && e.capacity > 0) {
106             cut.emplace_back(e.from, e.to);
107         }
108     }
109     return cut;
110 }
111
112 // Example on how to use
113 void solve()
114 {
115     int n, m;
116     cin >> n >> m;
117     int N = n + m + 2;
118
119     int source = N - 2;
120     int sink = N - 1;
121
122     Dinic flow(N);
123
124     for (int i = 0; i < n; i++) {
125         int q; cin >> q;
126         while(q--) {
127             int b; cin >> b;
128             flow.add_edge(i, n + b - 1, 1);
129         }
130     }
131
132     for (int i = 0; i < n; i++) {
133         flow.add_edge(source, i, 1);
134     }
135
136     for (int i = 0; i < m; i++) {
137         flow.add_edge(i + n, sink, 1);
138     }
139
140     cout << m - flow.get_max_flow(source, sink) << endl;
141
142     // Getting participant edges
143     for (auto & edge: flow.edges) {
144         if (edge.capacity == 0) continue; // This means is a reverse edge
145         if (edge.from == source || edge.to == source) continue;
146         if (edge.from == sink || edge.to == sink) continue;
147         if (edge.flow == 0) continue; // Is not participant
148
149         cout << edge.from + 1 << " " << edge.to - n + 1 << endl;
150     }
151 }

```

8.7 Dinic Min cost

Complexity: $O(V \cdot V \cdot E)$, Bipartite is $O(\sqrt{V} \cdot E)$

Gives you the max_flow with the min cost

```

1 // TITLE: Dinic Min cost
2 // COMPLEXITY:  $O(V \cdot V \cdot E)$ , Bipartite is  $O(\sqrt{V} \cdot E)$ 
3 // DESCRIPTION: Gives you the max_flow with the min cost
4
5 // Edge structure
6 struct Edge
7 {
8     int from, to;
9     int flow, capacity;
10    int cost;

```

```

11 Edge(int from_, int to_, int flow_, int capacity_ 77
12 , int cost_) 78
13 : from(from_), to(to_), flow(flow_), capacity 79
14 (capacity_), cost(cost_) 80
15 {} 81
16 }; 82
17 struct Dinic 83
18 { 84
19     vector<vector<int>> graph; 85
20     vector<Edge> edges; 86
21     vector<int> dist; 87
22     vector<bool> inqueue; 88
23     int size; 89
24     int cost = 0; 90
25 91
26 Dinic(int n) 92
27 { 93
28     graph.resize(n); 94
29     dist.resize(n); 95
30     inqueue.resize(n); 96
31     size = n; 97
32     edges.clear(); 98
33 99
34 void add_edge(int from, int to, int capacity, int 100
35 cost) 101
36 { 102
37     edges.emplace_back(from, to, 0, capacity, 103
38 cost); 104
39     graph[from].push_back(edges.size() - 1); 105
40 106
41     edges.emplace_back(to, from, 0, 0, -cost); 107
42     graph[to].push_back(edges.size() - 1); 108
43 109
44 int get_max_flow(int source, int sink) 110
45 { 111
46     int max_flow = 0; 112
47     vector<int> next(size); 113
48     while(spfa(source, sink)) { 114
49         next.assign(size, 0); 115
50         for (int f = dfs(source, sink, next, oo); 116
51             f != 0; f = dfs(source, sink, next, oo)) { 117
52             max_flow += f; 118
53         } 119
54     } 120
55     return max_flow; 121
56 } 122
57 123
58 bool spfa(int source, int sink) 124
59 { 125
60     dist.assign(size, oo); 126
61     inqueue.assign(size, false); 127
62     queue<int> q; 128
63     q.push(source); 129
64     dist[source] = 0; 130
65     inqueue[source] = true; 131
66 132
67     while(!q.empty()) { 133
68         int a = q.front(); 134
69         q.pop(); 135
70         inqueue[a] = false; 136
71 137
72         for (int & b: graph[a]) { 138
73             auto edge = edges[b]; 139
74             int cap = edge.capacity - edge.flow; 140
75             if (cap > 0 && dist[edge.to] > dist[ 141
76 edge.from] + edge.cost) { 142
77                 dist[edge.to] = dist[edge.from] + 143
78                 edge.cost; 144
79                 if (not inqueue[edge.to]) { 145
80                     q.push(edge.to); 146
81                     inqueue[edge.to] = true; 147
82                 } 148
83             } 149
84         } 150
85     } 151
86     return dist[sink] != oo; 152
87 } 153
88 154
89 int dfs(int curr, int sink, vector<int> & next, 155
90 int flow) 156
91 { 157
92     if (curr == sink) return flow; 158
93     int num_edges = graph[curr].size(); 159
94 160
95     for (; next[curr] < num_edges; next[curr]++) 161
96     { 162
97         int b = graph[curr][next[curr]]; 163
98         auto & edge = edges[b]; 164
99         auto & rev_edge = edges[b^1]; 165
100 166
101         int cap = edge.capacity - edge.flow; 167
102         if (cap > 0 && (dist[edge.from] + edge. 168
103 cost == dist[edge.to])) { 169
104             int bottle_neck = dfs(edge.to, sink, 170
105 next, min(flow, cap)); 171
106             if (bottle_neck > 0) { 172
107                 edge.flow += bottle_neck; 173
108                 rev_edge.flow -= bottle_neck; 174
109                 cost += edge.cost * bottle_neck; 175
110                 return bottle_neck; 176
111             } 177
112         } 178
113     } 179
114     return 0; 180
115 } 181
116 182
117 vector<pair<int, int>> mincut(int source, int 183
118 sink) 184
119 { 185
120     vector<pair<int, int>> cut; 186
121     spfa(source, sink); 187
122     for (auto & e: edges) { 188
123         if (e.flow == e.capacity && dist[e.from] 189
124 != oo && level[e.to] == oo && e.capacity > 0) { 190
125             cut.emplace_back(e.from, e.to); 191
126         } 192
127     } 193
128     return cut; 194
129 } 195
130 196
131 }; 197
132 198
133 // Example on how to use 199
134 void solve() 200
135 { 201
136     int N = 10; 202
137 203
138     int source = 8; 204
139     int sink = 9; 205
140 206
141     Dinic flow(N); 207
142     flow.add_edge(8, 0, 4, 0); 208
143     flow.add_edge(8, 1, 3, 0); 209
144     flow.add_edge(8, 2, 2, 0); 210
145     flow.add_edge(8, 3, 1, 0); 211
146 212
147     flow.add_edge(0, 6, oo, 3); 213
148     flow.add_edge(0, 7, oo, 2); 214
149     flow.add_edge(0, 5, oo, 0); 215
150 216
151     flow.add_edge(1, 4, oo, 0); 217
152 } 218

```

```

144     flow.add_edge(4, 9, oo, 0);
145     flow.add_edge(5, 9, oo, 0);
146     flow.add_edge(6, 9, oo, 0);
147     flow.add_edge(7, 9, oo, 0);
148
149     int ans = flow.get_max_flow(source, sink);
150     debug(ans);
151     debug(flow.cost);
152 }
153
154 int32_t main()
155 {
156     solve();
157 }

```

8.8 Bellman Ford

Complexity: $O(n * m)$ | $n = |\text{nodes}|$, $m = |\text{edges}|$
 Finds shortest paths from a starting node to all nodes of the graph. Detects negative cycles, if they exist.

```

1 // TITLE: Bellman Ford
2 // COMPLEXITY:  $O(n * m)$  |  $n = |\text{nodes}|$ ,  $m = |\text{edges}|$ 
3 // DESCRIPTION: Finds shortest paths from a starting
  node to all nodes of the graph. Detects negative
  cycles, if they exist.
4
5 // a and b vertices, c cost
6 // [{a, b, c}, {a, b, c}]
7 vector<tuple<int, int, int>> edges;
8 int N;
9
10 void bellman_ford(int x){
11     for (int i = 0; i < N; i++){
12         dist[i] = oo;
13     }
14     dist[x] = 0;
15
16     for (int i = 0; i < N - 1; i++){
17         for (auto [a, b, c]: edges){
18             if (dist[a] == oo) continue;
19             dist[b] = min(dist[b], dist[a] + w);
20         }
21     }
22 }
23 // return true if has cycle
24 bool check_negative_cycle(int x){
25     for (int i = 0; i < N; i++){
26         dist[i] = oo;
27     }
28     dist[x] = 0;
29
30     for (int i = 0; i < N - 1; i++){
31         for (auto [a, b, c]: edges){
32             if (dist[a] == oo) continue;
33             dist[b] = min(dist[b], dist[a] + w);
34         }
35     }
36
37     for (auto [a, b, c]: edges){
38         if (dist[a] == oo) continue;
39         if (dist[a] + w < dist[b]){
40             return true;
41         }
42     }
43     return false;
44 }
45

```

8.9 2SAT

Complexity: $O(n+m)$, $n = \text{number of variables}$, $m = \text{number of conjunctions (ands)}$.

Finds an assignment that makes a certain boolean formula true, or determines that such an assignment does not exist.

```

1 // TITLE: 2SAT
2 // COMPLEXITY:  $O(n+m)$ ,  $n = \text{number of variables}$ ,  $m =$ 
  number of conjunctions (ands).
3 // DESCRIPTION: Finds an assignment that makes a
  certain boolean formula true, or determines that
  such an assignment does not exist.
4
5 struct twosat {
6     vi vis, begin;
7     stack<int> tout;
8     vector<vi> g, gi, con, sccg;
9     vi repr, conv;
10    int gsize;
11    void dfs1(int a) {
12        if (vis[a]) return;
13        vis[a]=true;
14
15        for(auto& b : g[a]) {
16            dfs1(b);
17        }
18
19        tout.push(a);
20    }
21
22    void dfs2(int a, int orig) {
23        if (vis[a]) return;
24        vis[a]=true;
25
26        repr[a]=orig;
27        sccg[orig].pb(a);
28        for(auto& b : gi[a]) {
29            if (vis[b]) {
30                if (repr[b] != orig) {
31                    con[repr[b]].pb(orig);
32                    begin[orig]++;
33                }
34                continue;
35            }
36            dfs2(b, orig);
37        }
38    }
39 }
40 // if s1 = 1 and s2 = 1 this adds a \ / b to the
  graph
41 void addedge(int a, int s1,
42              int b, int s2) {
43     g[2*a+(!s1)].pb(2*b+s2);
44     gi[2*b+s2].pb(2*a+(!s1));
45
46     g[2*b+(!s2)].pb(2*a+s1);
47     gi[2*a+s1].pb(2*b+(!s2));
48 }
49
50
51 twosat(int nvars) {
52     gsize=2*nvars;
53     g.assign(gsize, vi());
54     gi.assign(gsize, vi());
55     con.assign(gsize, vi());
56     sccg.assign(gsize, vi());
57     repr.assign(gsize, -1);
58     vis.assign(gsize, 0);
59     begin.assign(gsize, 0);
60 }
61

```

```

62 // returns empty vector if the formula is not
63 satisfiable.
64 vi run() {
65     vi vals(gsize/2, -1);
66     rep(i,0,gsize) dfs1(i);
67     vis.assign(gsize,0);
68     while(!tout.empty()) {
69         int cur = tout.top(); tout.pop();
70         if (vis[cur]) continue;
71         dfs2(cur, cur);
72         conv.pb(cur);
73     }
74     rep(i, 0, gsize/2) {
75         if (repr[2*i] == repr[2*i+1]) {
76             return {};
77         }
78     }
79     queue<int> q;
80     for(auto& v : conv) {
81         if (degin[v] == 0) q.push(v);
82     }
83     while(!q.empty()) {
84         int cur=q.front(); q.pop();
85         for(auto guy : sccg[cur]) {
86             int s = guy%2;
87             int idx = guy/2;
88             if (vals[idx] != -1) continue;
89             if (s) {
90                 vals[idx] = false;
91             } else {
92                 vals[idx]=true;
93             }
94         }
95         for (auto& b : con[cur]) {
96             if(--degin[b] == 0) q.push(b);
97         }
98     }
99     return vals;
100 }
101 }
102 }
103 }
104 };

```

9 Parser

9.1 Parsing Functions

Complexity:

```
1 // TITLE: Parsing Functions
```

```

2
3 vector<string> split_string(const string & s, const
4     string & sep = " ") {
5     int w = sep.size();
6     vector<string> ans;
7     string curr;
8     auto add = [&](string a) {
9         if (a.size() > 0) {
10             ans.push_back(a);
11         }
12     };
13
14     for (int i = 0; i + w < s.size(); i++) {
15         if (s.substr(i, w) == sep) {
16             i += w-1;
17             add(curr);
18             curr.clear();
19             continue;
20         }
21         curr.push_back(s[i]);
22     }
23     add(curr);
24     return ans;
25 }
26
27 vector<int> parse_vector_int(string & s)
28 {
29     vector<int> nums;
30     for (string x: split_string(s)) {
31         nums.push_back(stoi(x));
32     }
33     return nums;
34 }
35
36 vector<float> parse_vector_float(string & s)
37 {
38     vector<float> nums;
39     for (string x: split_string(s)) {
40         nums.push_back(stof(x));
41     }
42     return nums;
43 }
44
45 void solve()
46 {
47     cin.ignore();
48     string s;
49     getline(cin, s);
50
51     auto nums = parse_vector_float(s);
52     for (auto x: nums) {
53         cout << x << endl;
54     }
55 }

```