



# Notebook - Maratona de Programação

Cabo HDMI, VGA, USB

## Contents

<b>1 String</b>	<b>2</b>	8.8 Bellman Ford . . . . .	13
1.1 String hash . . . . .	2	8.9 2SAT . . . . .	13
1.2 Z function . . . . .	2	<b>9 Parser</b>	<b>14</b>
1.3 Suffix Array . . . . .	2	9.1 Parsing Functions . . . . .	14
<b>2 Math</b>	<b>3</b>		
2.1 Fast Fourier Transform . . . . .	3		
<b>3 Segtree</b>	<b>4</b>		
3.1 Set and update lazy seg . . . . .	4		
3.2 Standard SegTree . . . . .	4		
3.3 Lazy SegTree . . . . .	5		
3.4 Persistent sum segment tree . . . . .	6		
<b>4 Algorithms</b>	<b>6</b>		
4.1 Sparse table . . . . .	6		
<b>5 Set</b>	<b>6</b>		
5.1 Ordered Set . . . . .	6		
5.2 Multiset . . . . .	6		
5.3 Set . . . . .	7		
<b>6 Misc</b>	<b>7</b>		
6.1 Template . . . . .	7		
<b>7 Geometry</b>	<b>7</b>		
7.1 Convex Hull . . . . .	7		
7.2 Lattice Points . . . . .	8		
7.3 Line Intersegment . . . . .	8		
<b>8 Graph</b>	<b>9</b>		
8.1 Dominator tree . . . . .	9		
8.2 Topological Sort . . . . .	9		
8.3 Kth Ancestor . . . . .	10		
8.4 Dfs tree . . . . .	10		
8.5 Dkistra . . . . .	10		
8.6 Dinic . . . . .	10		
8.7 Dinic Min cost . . . . .	12		

# 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 - l]) % m1;
41             int c2 = m2 - (sum2[l] * p2[r - l]) % 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:  $O(n)$

$z[i]$  = largest  $m$  such that  $s[0..m] = s[i..i+m]$

```
1 // TITLE: Z function
2 // COMPLEXITY:  $O(n)$ 
3 // DESCRIPTION:  $z[i]$  = largest  $m$  such that  $s[0..m] = s[
  i..i+m]$ 
4
5 vector<int> Z(string s) {
6     int n = s.size();
7     vector<int> z(n);
8     int x = 0, y = 0;
9     for (int i = 1; i < n; i++) {
```

```
10         z[i] = max(0, min(z[i - x], y - i + 1));
11         while (i + z[i] < n and s[z[i]] == s[i + z[i]
12         ]) {
13             x = i; y = i + z[i]; z[i]++;
14         }
15         return z;
16     }
```

## 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             if (prev == cur)
55                 nc[p[i]]=nc[p[i-1]];
```

```

55         else nc[p[i]]=nc[p[i-1]]+1;
56     }
57     c=nc;
58 }
59
60 return p;
61 }

```

## 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
37     roots.resize(1 << nbase);
38
39     while(base < nbase){
40         ld angle = 2*PI / (1 << (base + 1));
41         for(int i = 1 << (base - 1); i < (1 << base);
42             i++){
43             roots[i << 1] = roots[i];
44             ld angle_i = angle * (2 * i + 1 - (1 <<
45                 base));
46             roots[(i << 1) + 1] = num(cos(angle_i),
47                 sin(angle_i));
48             base++;
49         }
50     }
51
52 void fft(vector<num> &a, int n = -1){
53     if(n == -1)
54         n = a.size();
55
56

```

```

57     assert((n & (n-1)) == 0);
58     int zeros = __builtin_ctz(n);
59     ensure_base(zeros);
60     int shift = base - zeros;
61     for(int i = 0; i < n; i++){
62         if(i < (rev[i] >> shift))
63             swap(a[i], a[rev[i] >> shift]);
64     }
65
66     for(int k = 1; k < n; k <= 1)
67         for(int i = 0; i < n; i += 2 * k)
68             for(int j = 0; j < k; j++){
69                 num z = a[i+j+k] * roots[j+k];
70                 a[i+j+k] = a[i+j] - z;
71                 a[i+j] = a[i+j] + z;
72             }
73     }
74
75 vector<num> fa, fb;
76 vector<ll> multiply(vector<ll> &a, vector<ll> &b){
77     int need = a.size() + b.size() - 1;
78     int nbase = 0;
79     while((1 << nbase) < need) nbase++;
80     ensure_base(nbase);
81     int sz = 1 << nbase;
82     if(sz > (int) fa.size())
83         fa.resize(sz);
84
85     for(int i = 0; i < sz; i++){
86         int x = (i < (int) a.size() ? a[i] : 0);
87         int y = (i < (int) b.size() ? b[i] : 0);
88         fa[i] = num(x, y);
89     }
90     fft(fa, sz);
91     num r(0, -0.25 / sz);
92     for(int i = 0; i <= (sz >> 1); i++){
93         int j = (sz - i) & (sz - 1);
94         num z = (fa[j] * fa[j] - conj(fa[i] * fa[i]))
95             * r;
96         if(i != j) {
97             fa[j] = (fa[i] * fa[i] - conj(fa[j] * fa[
98                 j])) * r;
99         }
100         fa[i] = z;
101     }
102     fft(fa, sz);
103     vector<ll> res(need);
104     for(int i = 0; i < need; i++)
105         res[i] = round(fa[i].x);
106
107     return res;
108 }
109
110 vector<ll> multiply_mod(vector<ll> &a, vector<ll> &b,
111     int m, int eq = 0){
112     int need = a.size() + b.size() - 1;
113     int nbase = 0;
114     while((1 << nbase) < need) nbase++;
115     ensure_base(nbase);
116     int sz = 1 << nbase;
117     if(sz > (int) fa.size())
118         fa.resize(sz);
119
120     for(int i=0;i<(int)a.size();i++){
121         int x = (a[i] % m + m) % m;
122         fa[i] = num(x & ((1 << 15) - 1), x >> 15);
123     }
124     fill(fa.begin() + a.size(), fa.begin() + sz, num
125         {0, 0});
126     fft(fa, sz);
127     if(sz > (int) fb.size())
128         fb.resize(sz);
129     if(eq)

```

```

118     copy(fa.begin(), fa.begin() + sz, fb.begin()) 17
119 ; 18
120 else{ 19
121     for(int i = 0; i < (int) b.size(); i++){ 20
122         int x = (b[i] % m + m) % m; 21
123         fb[i] = num(x & ((1 << 15) - 1), x >> 15) 22
124 ; 23
125 } 24
126 fill(fb.begin() + b.size(), fb.begin() + sz, 25
num {0, 0});
127 fft(fb, sz); 26
128 } 27
129 ld ratio = 0.25 / sz; 28
130 num r2(0, -1); 29
131 num r3(ratio, 0); 30
132 num r4(0, -ratio); 31
133 num r5(0, 1); 32
134 for(int i=0;i<=(sz >> 1);i++) { 33
135     int j = (sz - i) & (sz - 1); 34
136     num a1 = (fa[i] + conj(fa[j])); 35
137     num a2 = (fa[i] - conj(fa[j])) * r2; 36
138     num b1 = (fb[i] + conj(fb[j])) * r3; 37
139     num b2 = (fb[i] - conj(fb[j])) * r4; 38
140     if(i != j){ 39
141         num c1 = (fa[j] + conj(fa[i])); 40
142         num c2 = (fa[j] - conj(fa[i])) * r2; 41
143         num d1 = (fb[j] + conj(fb[i])) * r3; 42
144         num d2 = (fb[j] - conj(fb[i])) * r4; 43
145         fa[i] = c1 * d1 + c2 * d2 * r5; 44
146         fb[i] = c1 * d2 + c2 * d1; 45
147     } 46
148     fa[j] = a1 * b1 + a2 * b2 * r5; 47
149     fb[j] = a1 * b2 + a2 * b1; 48
150 } 49
151 fft(fa, sz); 50
152 fft(fb, sz); 51
153 vector<ll> res(need); 52
154 for(int i=0;i<need;i++){ 53
155     ll aa = round(fa[i].x); 54
156     ll bb = round(fb[i].x); 55
157     ll cc = round(fa[i].y); 56
158     res[i] = (aa + ((bb % m) << 15) + ((cc % m) 57
<< 30)) % m;
159 } 58
160 } 59

```

## 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) {

```

```

if(opvec[no] == SET) seg[no] = lazy[no];
else seg[no] += lazy[no];

```

```

lazy[no]=-1;
opvec[no]=-1;
return;

```

```

}

```

```

if(opvec[no] == SET) {
    seg[no] = (rx-lx) * lazy[no];
    lazy[2*no+1] = lazy[no];
    lazy[2*no+2] = lazy[no];

```

```

    opvec[2*no+1] = SET;
    opvec[2*no+2] = SET;

```

```

    lazy[no] = -1;
    opvec[no]=-1;
    return;

```

```

}

```

```

seg[no] += (rx-lx) * lazy[no];
if (lazy[2*no+1] == -1) {
    lazy[2*no+1] = 0;
    opvec[2*no+1] = ADD;

```

```

}

```

```

if (lazy[2*no+2] == -1) {
    lazy[2*no+2] = 0;
    opvec[2*no+2] = ADD;

```

```

}

```

```

lazy[2*no+1] += lazy[no];
lazy[2*no+2] += lazy[no];

```

```

lazy[no] = -1;
opvec[no]=-1;

```

```

}

```

```

54 void update(int l, int r, int val, int op, int no=0,
int lx=0, int rx=segsize) {
55     propagate(no, lx, rx);
56     if (r <= lx or l >= rx) return;
57     if (lx >= l and rx <= r) {
58         lazy[no] = val;
59         opvec[no] = op;
60         propagate(no, lx, rx);
61         return;
62     }
63
64     int mid = (rx+lx)/2;
65     update(l, r, val, op, 2*no+1, lx, mid);
66     update(l, r, val, op, 2*no+2, mid, rx);
67     seg[no] = seg[2*no+1]+seg[2*no+2];
68 }
69
70 int query(int l, int r, int no=0, int lx=0, int rx=
segsize) {
71     propagate(no, lx, rx);
72     if (r <= lx or l >= rx) return 0;
73     if (lx >= l and rx <= r) return seg[no];
74
75     int mid = (rx+lx)/2;
76     return
77         query(l,r,2*no+1,lx,mid) +
78         query(l,r,2*no+2, mid, rx);
79 }

```

### 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(segsize);
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(segsize);
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         return new node(0, lx, rx, 0, 0);
23     }
24
25     int mid = (lx+rx)/2;
26     auto nol = build(arr, lx, mid);
27     auto nor = build(arr, mid, rx);
28     return new node(nol->val + nor->val, lx, rx, nol,
29                     nor);
30 }
31
32 node* update(int idx, int val, node *no) {
33     if (idx < no->lx or idx >= no->rx) return no;
34     if (no->rx - no->lx == 1) {
35         return new node(val+no->val, no->lx, no->rx,
36                         no->l, no->r);
37     }
38
39     auto nol = update(idx, val, no->l);
40     auto nor = update(idx, val, no->r);
41     return new node(nol->val + nor->val, no->lx, no->
42                     rx, nol, nor);
43 }
44
45 int query(int l, int r, node *no) {
46     if (r <= no->lx or no->rx <= l) return 0;
47     if (l <= no->lx and no->rx <= r) return no->val;
48
49     return query(l, r, no->l) + query(l, r, no->r);
50 }
```

## 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
  interval.
4
5 struct sptable {
6     vector<vi> table;
7
8     int ilog(int x) {
9         return (__builtin_clzll(1ll) -
10             __builtin_clzll(x));
11     }
12
13     sptable(vi& vals) {
14         int n = vals.size();
15         int ln = ilog(n)+1;
16         table.assign(ln, vi(n));
17
18         rep(i, 0, n) table[0][i] = vals[i];
19
20         rep(k, 1, ln) {
21             rep(i, 0, n) {
22                 table[k][i] = min(table[k-1][i],
23                                     table[k-1][min(i + (1<<(k-1)), n-1)]);
24             }
25         }
26
27         // returns minimum of vals in range [a, b)
28         int getmin(int a, int b) {
29             int k = ilog(b-a);
30             return min(table[k][a], table[k][b-(1<<k)]);
31         }
32     };
33 }
```

## 5 Set

### 5.1 Ordered Set

Complexity:  $\log n$

Worst set with adtional operations

```
1 // TITLE: Ordered Set
2 // COMPLEXITY:  $\log n$ 
3 // DESCRIPTION: Worst set with adtional 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 val-

ues

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

## 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 signed main()
28 {
29     ios_base::sync_with_stdio(0); cin.tie(0); cout.tie(0);
30     int t=1;
31     // cin>>t;
32     while(t--) solve();
33 }
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
6 struct Point
7 {
8     int x, y;
9
10     void read()
11     {
12         cin >> x >> y;
13     }
14
15     Point operator- (const Point & b) const
16     {
17         Point p;
18         p.x = x - b.x;
19         p.y = y - b.y;
20         return p;
21     }
22
23     void operator-= (const Point & b)
24     {
25         x -= b.x;
26         y -= b.y;
27     }
28
29     int operator* (const Point & b) const
30     {
31         return x * b.y - b.x * y;
32     }
33
34     bool operator< (const Point & b) const
35     {
36         return make_pair(x, y) < make_pair(b.x, b.y);
37     }
38 };
39
40 int triangle(const Point & a, const Point & b, const
  Point & c)
41 {
42     return (b - a) * (c - a);
43 }
44
45
```

```

46 vector<Point> convex_hull(vector<Point> points)
47 {
48     vector<Point> hull;
49     sort(all(points));
50
51     for (int z = 0; z < 2; z++) {
52         int s = hull.size();
53         for (int i = 0; i < points.size(); i++) {
54             while(hull.size() >= s + 2) {
55                 auto a = hull.end()[-2];
56                 auto b = hull.end()[-1];
57                 if (triangle(a, b, points[i]) <= 0) {
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
27     // Calculating Area
28     int a2 = 0;
29     for (int i = 0; i < n; i++) {
30         int j = (i + 1) % n;
31         a2 += points[i] * points[j];
32     }
33     a2 = abs(a2);
34     // Picks theorem
35     int I = (a2 - B + 2)/2;
36     cout << I << " " << B << endl;
37 }

```

## 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    {
12        cin >> x >> y;
13    }
14
15    Point operator- (const Point & b) const
16    {
17        Point p;
18        p.x = x - b.x;
19        p.y = y - b.y;
20        return p;
21    }
22
23    void operator-= (const Point & b)
24    {
25        x -= b.x;
26        y -= b.y;
27    }
28
29    int operator* (const Point & b) const
30    {
31        return x * b.y - b.x * y;
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
66     int s3 = triangle(p3, p4, p1);
67     int s4 = triangle(p3, p4, p2);
68
69     if ((s1 < 0) && (s2 < 0)) ans = false;
70     if ((s1 > 0) && (s2 > 0)) ans = false;
71     if ((s3 < 0) && (s4 < 0)) ans = false;
72     if ((s3 > 0) && (s4 > 0)) ans = false;
73     return ans;
74 }

```



```
71 }
```

## 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) {
```

```

38     cout << "IMPOSSIBLE" << endl;
39 }
40 return 0;
41 }

```

## 8.3 Kth Ancestor

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

Preprocess, then find in  $\log n$

```

1 // TITLE: Kth Ancestor
2 // COMPLEXITY:  $O(n \cdot \log(n))$ 
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,
7     int k)
8 {
9     for (int j = 0; j < LOG_N; j++) {
10         if (k & ((int)1 << j)) {
11             v = up[v][j];
12         }
13     }
14     return v;
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:  $O(E + V \cdot \log(v))$ 
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:  $O(V \cdot V \cdot E)$ , Bipartite is  $O(\sqrt{V} \cdot E)$ 
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     for (; next[curr] < num_edges; next[curr]++) {
82         int b = graph[curr][next[curr]];
83         auto & edge = edges[b];
84         auto & rev_edge = edges[b^1];
85
86         int cap = edge.capacity - edge.flow;
87         if (cap > 0 && (level[curr] + 1 == level[
88             edge.to])) {
89             int bottle_neck = dfs(edge.to, sink,
90                 next, min(flow, cap));
91             if (bottle_neck > 0) {
92                 edge.flow += bottle_neck;
93                 rev_edge.flow -= bottle_neck;
94                 return bottle_neck;
95             }
96         }
97         return 0;
98     }
99 }
100
101 vector<pair<int, int>> mincut(int source, int sink)
102 {
103     vector<pair<int, int>> cut;
104     bfs(source, sink);
105     for (auto & e: edges) {
106         if (e.flow == e.capacity && level[e.from]
107             != -1 && level[e.to] == -1 && e.capacity > 0) {
108             cut.emplace_back(e.from, e.to);
109         }
110     }
111     return cut;
112 }
113
114 // Example on how to use
115 void solve()
116 {
117     int n, m;
118     cin >> n >> m;
119     int N = n + m + 2;
120
121     int source = N - 2;
122     int sink = N - 1;
123
124     Dinic flow(N);
125
126     for (int i = 0; i < n; i++) {
127         int q; cin >> q;
128         while(q--) {
129             int b; cin >> b;
130             flow.add_edge(i, n + b - 1, 1);
131         }
132     }
133     for (int i = 0; i < n; i++) {
134         flow.add_edge(source, i, 1);
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) <<
141     endl;
142
143     // Getting participant edges
144     for (auto & edge: flow.edges) {
145         if (edge.capacity == 0) continue; // This
146         // means is a reverse edge
147         if (edge.from == source || edge.to == source)
148             continue;
149         if (edge.from == sink || edge.to == sink)
150             continue;
151         if (edge.flow == 0) continue; // Is not
152         // participant
153         cout << edge.from + 1 << " " << edge.to - n +
154         1 << endl;
155     }
156 }

```

## 8.7 Dinic Min cost

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

Gives you the max\_flow with the min cost

```

1 // TITLE: Dinic Min cost
2 // COMPLEXITY:  $O(V*V*E)$ , Bipartite is  $O(\sqrt{V} E)$ 
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
12    Edge(int from_, int to_, int flow_, int capacity_
13        , int cost_)
14        : from(from_), to(to_), flow(flow_), capacity
15        (capacity_), cost(cost_)
16    {}
17 };
18
19 struct Dinic
20 {
21     vector<vector<int>> graph;
22     vector<Edge> edges;
23     vector<int> dist;
24     vector<bool> inqueue;
25     int size;
26     int cost = 0;
27
28     Dinic(int n)
29     {
30         graph.resize(n);
31         dist.resize(n);
32         inqueue.resize(n);
33         size = n;
34         edges.clear();
35
36     void add_edge(int from, int to, int capacity, int
37         cost)
38     {
39         edges.emplace_back(from, to, 0, capacity,
40         cost);
41         graph[from].push_back(edges.size() - 1);
42         edges.emplace_back(to, from, 0, 0, -cost);
43         graph[to].push_back(edges.size() - 1);
44     }
45
46     int get_max_flow(int source, int sink)
47     {
48         int max_flow = 0;
49         vector<int> next(size);
50         while(spfa(source, sink)) {
51             next.assign(size, 0);
52             for (int f = dfs(source, sink, next, oo);
53                 f != 0; f = dfs(source, sink, next, oo)) {
54                 max_flow += f;
55             }
56         }
57         return max_flow;
58     }
59
60     bool spfa(int source, int sink)
61     {
62         dist.assign(size, oo);
63         inqueue.assign(size, false);
64         queue<int> q;
65         q.push(source);
66         dist[source] = 0;
67         inqueue[source] = true;
68
69         while(!q.empty()) {
70             int a = q.front();
71             q.pop();
72             inqueue[a] = false;
73
74             for (int & b: graph[a]) {
75                 auto edge = edges[b];
76                 int cap = edge.capacity - edge.flow;
77                 if (cap > 0 && dist[edge.to] > dist[
78                     edge.from] + edge.cost) {
79                     dist[edge.to] = dist[edge.from] +
80                     edge.cost;
81                     if (not inqueue[edge.to]) {
82                         q.push(edge.to);
83                         inqueue[edge.to] = true;
84                     }
85                 }
86             }
87         }
88         return dist[sink] != oo;
89     }
90
91     int dfs(int curr, int sink, vector<int> & next,
92         int flow)
93     {
94         if (curr == sink) return flow;
95         int num_edges = graph[curr].size();
96         for (; next[curr] < num_edges; next[curr]++)
97         {
98             int b = graph[curr][next[curr]];
99             auto & edge = edges[b];
100             auto & rev_edge = edges[b^1];
101
102             int cap = edge.capacity - edge.flow;
103             if (cap > 0 && (dist[edge.from] + edge.
104                 cost == dist[edge.to])) {
105                 int bottle_neck = dfs(edge.to, sink,
106                     next, min(flow, cap));
107                 if (bottle_neck > 0) {
108                     edge.flow += bottle_neck;
109                     rev_edge.flow -= bottle_neck;
110                     cost += edge.cost * bottle_neck;
111                     return bottle_neck;
112                 }
113             }
114         }
115         return 0;
116     }
117
118     vector<pair<int, int>> mincut(int source, int
119         sink)
120     {
121         vector<pair<int, int>> cut;
122         spfa(source, sink);
123         for (auto & e: edges) {
124             if (e.flow == e.capacity && dist[e.from]
125                 != oo && level[e.to] == oo && e.capacity > 0) {
126                 cut.emplace_back(e.from, e.to);
127             }
128         }
129         return cut;
130     }
131 };
132
133 // Example on how to use
134 void solve()
135 {
136     int N = 10;

```

```

128
129     int source = 8;
130     int sink = 9;
131
132     Dinic flow(N);
133     flow.add_edge(8, 0, 4, 0);
134     flow.add_edge(8, 1, 3, 0);
135     flow.add_edge(8, 2, 2, 0);
136     flow.add_edge(8, 3, 1, 0);
137
138     flow.add_edge(0, 6, oo, 3);
139     flow.add_edge(0, 7, oo, 2);
140     flow.add_edge(0, 5, oo, 0);
141
142     flow.add_edge(1, 4, oo, 0);
143
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, degin;
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                    degin[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     degin.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
80     queue<int> q;
81     for(auto& v : conv) {
82         if (degin[v] == 0) q.push(v);
83     }
84
85     while(!q.empty()) {
86         int cur=q.front(); q.pop();
87         for(auto guy : sccg[cur]) {
88             int s = guy%2;
89             int idx = guy/2;
90             if (vals[idx] != -1) continue;
91             if (s) {
92                 vals[idx] = false;
93             } else {
94                 vals[idx]=true;
95             }
96         }
97         for (auto& b : con[cur]) {
98             if(--degin[b] == 0) q.push(b);
99         }
100     }
101
102     return vals;
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
9     auto add = [&](string a) {
10         if (a.size() > 0) {
11             ans.push_back(a);
12         }
13     };
14
15     for (int i = 0; i + w < s.size(); i++) {
16         if (s.substr(i, w) == sep) {
17             i += w-1;
18             add(curr);
19             curr.clear();
20             continue;
21         }
22         curr.push_back(s[i]);
23     }
24     add(curr);
25     return ans;
26 }
27
28 vector<int> parse_vector_int(string & s)
29 {
30     vector<int> nums;
31     for (string x: split_string(s)) {
32         nums.push_back(stoi(x));
33     }
34     return nums;
35 }
36
37 vector<float> parse_vector_float(string & s)
38 {
39     vector<float> nums;
40     for (string x: split_string(s)) {
41         nums.push_back(stof(x));
42     }
43     return nums;
44 }
45
46 void solve()
47 {
48     cin.ignore();
49     string s;
50     getline(cin, s);
51
52     auto nums = parse_vector_float(s);
53     for (auto x: nums) {
54         cout << x << endl;
55     }
56 }

```