

Contents

1 Data Structures

1.1	Bit 2d
1.2	DSU - Disjoint Set Union
1.3	DSU - Binary Tree
1.4	Mo's Algorithm
1.5	Segment Tree
1.6	Sparse Table RMQ

2 Graphs

2.1	BFS 0-1
2.2	Dijkstra
2.3	Dinic - Flow/matchings
2.4	Floyd-Warshall
2.5	Hopcroft-Karp - Bipartite Matching
2.6	Hungarian
2.7	Kosaraju - SCCs
2.8	Kuhn - Bipartite Matching
2.9	Min cost flow
2.10	MST - Kruskal
2.11	MST - Prim

3 DP

3.1	Bin Packing
3.2	Broken Profile DP
3.3	Convex Hull Trick (CHT)
3.4	Edit Distance (Levenshtein)
3.5	Knapsack - 1D
3.6	Knapsack - 2D
3.7	LCS - Longest Common Subsequence
3.8	LiChao Tree
3.9	LIS - Longest Increasing Subsequence
3.10	SOSDP
3.11	Subset Sum

4 Trees

4.1	Sum of distances
4.2	Edge HLD
4.3	HLD - Heavy light decomposition
4.4	LCA - RMQ
4.5	LCA - binary lifting

5 Problemas clássicos

5.1	2SAT
5.2	Next Greater Element

6 Strings

6.1	Hashing
6.2	KMP
6.3	Suffix Array
6.4	Suffix Automaton
6.5	Z

7 Math

7.1	Combinatorics (Pascal's Triangle)
7.1.1	Combinatorial Analysis
7.2	Convolutions
7.2.1	AND convolution
7.2.2	GCD convolution
7.2.3	LCM convolution
7.2.4	OR convolution
7.2.5	XOR convolution
7.3	Extended Euclid
7.4	Factorization
7.5	FFT - Fast Fourier Transform
7.6	Inclusion-Exclusion Principle
7.7	Mint
7.8	Modular Inverse
7.9	Number Theoretic Transform (NTT)
7.9.1	NTT-Friendly Primes and Roots

7.10	Euler's Totient
------	---------------------------

8 Geometry

8.1	Convex hull - Graham Scan
8.2	Basic elements - geometry lib
8.2.1	Polygon Area
8.2.2	Point in polygon

1 Data Structures

1.1 Bit 2d

2D Sum BIT, update and sum. The problem must be 1-indexed.

Query/update time: $\mathcal{O}((\log n)^2)$

Construction time: $\mathcal{O}(n^2(\log n)^2)$

Space: $\mathcal{O}(n^2)$

```

1 #include <bits/stdc++.h>
2 using namespace std;
3
4 typedef long long ll;
5 #define MAX 1123
6

```

```

7 int bit[MAX][MAX], x, y;
8 void setbit(int i, int j, int delta) {
9     int j_;
10    while(i <= x) {
11        j_ = j;
12        while(j_ <= y) {
13            bit[i][j_] += delta;
14            j_ += j_ & -j_;
15        }
16        i += i & -i;
17    }
18 }
19 ll getbit(int i, int j) {
20     ll ans = 0;
21     int j_;
22     while(i) {
23        j_ = j;
24        while(j_) {
25            ans += bit[i][j_];
26            j_ -= j_ & -j_;
27        }
28        i -= i & -i;
29    }
30    return ans;
31 }
32
33 int main(void) {
34     int p;
35     while (scanf("%d %d %d", &x, &y, &p), x || y || p) {
36         for(int i = 0; i <= x; i++)
37             for(int j = 0; j <= y; j++)
38                 bit[i][j] = 0;
39         int q;
40         scanf("%d", &q);
41         while(q--) {
42             char c;
43             scanf(" %c", &c);
44             int n, xi, yi, zi, wi;
45             if(c == 'A') {
46                 scanf("%d %d %d", &n, &xi, &yi);
47                 xi++; yi++;
48                 setbit(xi, yi, n);
49             }
50             else {
51                 scanf("%d %d %d %d", &xi, &yi, &zi, &wi);
52                 xi++; yi++; zi++; wi++;
53                 if(xi > zi) swap(xi, zi);
54                 if(yi > wi) swap(yi, wi);
55                 ll ans = getbit(xi, wi) - getbit(xi, yi - 1)
56                 - getbit(xi - 1, wi) + getbit(xi - 1, yi - 1);
57                 printf("%lld\n", ans * (ll) p);
58             }
59         }
60         printf("\n");
61     }
62     return 0;
63 }

```

1.2 DSU - Disjoint Set Union

Query/update time: $\mathcal{O}(1)$

Construction time: $\mathcal{O}(n)$

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

```
1 struct DSU {
```

```

2     vi p, sz;
3     DSU(int n) {
4         p.resize(n);
5         iota(p.begin(), p.end(), 0);
6         sz.assign(n, 1);
7     }
8     int find(int i) {
9         if (p[i] == i) return i;
10        return p[i] = find(p[i]);
11    }
12    bool unite(int u, int v) {
13        u = find(u);
14        v = find(v);
15        if (u == v) return false;
16        if (sz[u] < sz[v]) swap(u, v);
17        p[v] = u;
18        sz[u] += sz[v];
19        return true;
20    }
21 }

```

1.3 DSU - Binary Tree

Specific code to find maximum path sums between pairs of vertices. Uses Kruskal-style MST. Query/update time: possibly $\mathcal{O}(n)$ Construction time: $\mathcal{O}(n)$ Space: $\mathcal{O}(n)$

```

1 vi d;
2 vi_ii e;
3 vi ans;
4
5 int merged;
6 vi _p, _leaf, _wei;
7 vvi adj;
8 int _find(int u) { return _p[u] == u ? u: _p[u] = _find(_p[u]); }
9 void _union(int u, int v, int w){
10    u = _find(u);
11    v = _find(v);
12    int merge_ind = merged+n;
13    _p[u] = merge_ind;
14    _p[v] = merge_ind;
15    _leaf[merge_ind] = _leaf[u] + _leaf[v];
16    _wei[merge_ind] = max(_wei[u], _wei[v]);
17    adj[u].push_back(merge_ind);
18    adj[merge_ind].push_back(u);
19    adj[v].push_back(merge_ind);
20    adj[merge_ind].push_back(v);
21    merged++;
22 }
23 void make(){
24    _p = vi(2*n);
25    for(int i = 0; i < 2*n; i++) _p[i] = i;
26    _leaf = vi(2*n, 1);
27    _wei = vi(2*n);
28    for(int i = 0; i < n; i++) _wei[i] = d[i];
29    merged = 0;
30    adj = vvi(2*n);
31 }
32
33 void dfs(int u, int p){
34    for(auto &v: adj[u]){
35        if(v == p) continue;
36        ans[v] = ans[u] + (_leaf[u] - _leaf[v])*_wei[u];
37        dfs(v, u);
38    }
39 }

```

1.4 Mo's Algorithm

A technique for solving offline range queries on static arrays by sorting queries to minimize total pointer movement. It processes intervals by incrementally updating the range via `add` and `remove` operations. With the optimal block size, the time complexity is $\mathcal{O}((N + Q)\sqrt{N})$ or $\mathcal{O}(N\sqrt{Q})$, depending on block size choice (\sqrt{N} or N/\sqrt{Q}). This example solves queries for distinct elements in range

```

1 struct Mo {
2     struct Query {
3         int l, r, idx, b;
4         bool operator<(const Query& o) const {
5             return b != o.b ? b < o.b :
6                 (b & 1 ? r > o.r : r < o.r);
7         }
8     };
9
10    int n, block_sz;
11
12    // custom stuff
13    vi freq, a;
14    int ans = 0;
15
16    vector<Query> queries;
17    Mo(int n) : n(n), block_sz(round(sqrt(n))) {}
18
19    // [l,r] indexed
20    void add_query(int l, int r, int i) {
21        queries.push_back({l,r,i,l/block_sz});
22    }
23    void add(int i) {
24        // add val at i
25        freq[a[i]]++;
26        if (freq[a[i]] == 1) ans++;
27    }
28    void remove(int i) {
29        // remove value at i
30        freq[a[i]]--;
31        if (freq[a[i]] == 0) ans--;
32    }
33    int get_ans() {
34        // compute current answer
35        return ans;
36    }
37
38    vi run() {
39        vi ans(queries.size());
40        sort(queries.begin(), queries.end());
41        int l = 0, r = -1;
42        for (auto& q : queries) {
43            while (l > q.l) add(--l);
44            while (r < q.r) add(++r);
45            while (l < q.l) remove(l++);
46            while (r > q.r) remove(r--);
47            ans[q.idx] = get_ans();
48        }
49        return ans;
50    }
51 }

```

1.5 Segment Tree

Segment tree with lazy propagation. Here the interval convention is $[l, r]$, with 0-based indexing. The example solves Kadane (max subarray sum) with point/range updates.

Query/update time: $\mathcal{O}(\log n)$

Construction time: $\mathcal{O}(n)$

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

```

1 struct segtree {
2     int size;
3     vector<node> nodes;
4     vector<bool> hasLazy;
5     vector<int> lazy;
6
7     struct node {
8         int seg, pre, suf, sum;
9     };
10
11    node NEUTRAL = {0,0,0,0};
12
13    void debug(){
14        if (nodes.empty() || size == 0) {
15            cout << "[Empty Tree]\n"; return;
16        }
17
18        string indent = "...";
19        function<void(int, int, int, string)> print_dfs;
20
21        print_dfs = [&](int x, int lx, int rx, string
22                         prefix) {
23            cout << prefix << "[" << lx << ", " << rx << "]"
24                         " ";
25
26            // debug node
27            node a = nodes[x];
28            cout << "{ ";
29            cout << "seg: " << a.seg << ", ";
30            cout << "pre: " << a.pre << ", ";
31            cout << "suf: " << a.suf << ", ";
32            cout << "sum: " << a.sum << ", ";
33            cout << "hasLazy: " << hasLazy[x] << ", ";
34            cout << "lazy: " << lazy[x] << ", ";
35            cout << "}";
36            cout << endl;
37
38            if (rx-lx <= 1) return;
39
40            int mx = (lx+rx)/2;
41            print_dfs(2*x+1, lx, mx, prefix + indent);
42            print_dfs(2*x+2, mx, rx, prefix + indent);
43        };
44        print_dfs(0, 0, size, "");
45
46        node single(int v){
47            return {v,v,v,v};
48        }
49
50        node merge(node a, node b){
51            return {
52                max(max(a.seg, b.seg), a.suf + b.pre),
53                max(a.pre, a.sum + b.pre),
54                max(b.suf, b.sum + a.suf),
55                a.sum+b.sum
56            };
57        }
58
59    };
60
61    void update(int l, int r, int val, int id) {
62        update(l, r, val, 0, 0, size);
63    }
64
65    int query(int l, int r) {
66        return query(l, r, 0, 0, size);
67    }
68
69    void update(int l, int r, int val, int id, int lx, int rx) {
70        if (lx > r || rx < l) return;
71        if (lx >= l & rx <= r) {
72            lazy[id] += val;
73            hasLazy[id] = true;
74            return;
75        }
76        if (lx < rx) {
77            int mx = (lx+rx)/2;
78            update(l, r, val, 2*id+1, lx, mx);
79            update(l, r, val, 2*id+2, mx, rx);
80        }
81    }
82
83    int query(int l, int r, int id, int lx, int rx) {
84        if (lx > r || rx < l) return 0;
85        if (lx >= l & rx <= r) {
86            return nodes[id].sum;
87        }
88        if (lx < rx) {
89            int mx = (lx+rx)/2;
90            return query(l, r, 2*id+1, lx, mx) +
91                   query(l, r, 2*id+2, mx, rx);
92        }
93    }
94
95    void print_dfs(int id, int lx, int rx, string prefix) {
96        cout << prefix << "[";
97        cout << "seg: " << nodes[id].seg << ", ";
98        cout << "pre: " << nodes[id].pre << ", ";
99        cout << "suf: " << nodes[id].suf << ", ";
100       cout << "sum: " << nodes[id].sum << ", ";
101       cout << "hasLazy: " << hasLazy[id] << ", ";
102       cout << "lazy: " << lazy[id] << ", ";
103       cout << "]";
104       cout << endl;
105   }
106
107   void print() {
108       print_dfs(0, 0, size, "");
109   }
110
111   void print() {
112       print_dfs(0, 0, size, "");
113   }
114
115   void print() {
116       print_dfs(0, 0, size, "");
117   }
118
119   void print() {
120       print_dfs(0, 0, size, "");
121   }
122
123   void print() {
124       print_dfs(0, 0, size, "");
125   }
126
127   void print() {
128       print_dfs(0, 0, size, "");
129   }
130
131   void print() {
132       print_dfs(0, 0, size, "");
133   }
134
135   void print() {
136       print_dfs(0, 0, size, "");
137   }
138
139   void print() {
140       print_dfs(0, 0, size, "");
141   }
142
143   void print() {
144       print_dfs(0, 0, size, "");
145   }
146
147   void print() {
148       print_dfs(0, 0, size, "");
149   }
150
151   void print() {
152       print_dfs(0, 0, size, "");
153   }
154
155   void print() {
156       print_dfs(0, 0, size, "");
157   }
158
159   void print() {
160       print_dfs(0, 0, size, "");
161   }
162
163   void print() {
164       print_dfs(0, 0, size, "");
165   }
166
167   void print() {
168       print_dfs(0, 0, size, "");
169   }
170
171   void print() {
172       print_dfs(0, 0, size, "");
173   }
174
175   void print() {
176       print_dfs(0, 0, size, "");
177   }
178
179   void print() {
180       print_dfs(0, 0, size, "");
181   }
182
183   void print() {
184       print_dfs(0, 0, size, "");
185   }
186
187   void print() {
188       print_dfs(0, 0, size, "");
189   }
190
191   void print() {
192       print_dfs(0, 0, size, "");
193   }
194
195   void print() {
196       print_dfs(0, 0, size, "");
197   }
198
199   void print() {
200       print_dfs(0, 0, size, "");
201   }
202
203   void print() {
204       print_dfs(0, 0, size, "");
205   }
206
207   void print() {
208       print_dfs(0, 0, size, "");
209   }
210
211   void print() {
212       print_dfs(0, 0, size, "");
213   }
214
215   void print() {
216       print_dfs(0, 0, size, "");
217   }
218
219   void print() {
220       print_dfs(0, 0, size, "");
221   }
222
223   void print() {
224       print_dfs(0, 0, size, "");
225   }
226
227   void print() {
228       print_dfs(0, 0, size, "");
229   }
230
231   void print() {
232       print_dfs(0, 0, size, "");
233   }
234
235   void print() {
236       print_dfs(0, 0, size, "");
237   }
238
239   void print() {
240       print_dfs(0, 0, size, "");
241   }
242
243   void print() {
244       print_dfs(0, 0, size, "");
245   }
246
247   void print() {
248       print_dfs(0, 0, size, "");
249   }
250
251   void print() {
252       print_dfs(0, 0, size, "");
253   }
254
255   void print() {
256       print_dfs(0, 0, size, "");
257   }
258
259   void print() {
260       print_dfs(0, 0, size, "");
261   }
262
263   void print() {
264       print_dfs(0, 0, size, "");
265   }
266
267   void print() {
268       print_dfs(0, 0, size, "");
269   }
270
271   void print() {
272       print_dfs(0, 0, size, "");
273   }
274
275   void print() {
276       print_dfs(0, 0, size, "");
277   }
278
279   void print() {
280       print_dfs(0, 0, size, "");
281   }
282
283   void print() {
284       print_dfs(0, 0, size, "");
285   }
286
287   void print() {
288       print_dfs(0, 0, size, "");
289   }
290
291   void print() {
292       print_dfs(0, 0, size, "");
293   }
294
295   void print() {
296       print_dfs(0, 0, size, "");
297   }
298
299   void print() {
300       print_dfs(0, 0, size, "");
301   }
302
303   void print() {
304       print_dfs(0, 0, size, "");
305   }
306
307   void print() {
308       print_dfs(0, 0, size, "");
309   }
310
311   void print() {
312       print_dfs(0, 0, size, "");
313   }
314
315   void print() {
316       print_dfs(0, 0, size, "");
317   }
318
319   void print() {
320       print_dfs(0, 0, size, "");
321   }
322
323   void print() {
324       print_dfs(0, 0, size, "");
325   }
326
327   void print() {
328       print_dfs(0, 0, size, "");
329   }
330
331   void print() {
332       print_dfs(0, 0, size, "");
333   }
334
335   void print() {
336       print_dfs(0, 0, size, "");
337   }
338
339   void print() {
340       print_dfs(0, 0, size, "");
341   }
342
343   void print() {
344       print_dfs(0, 0, size, "");
345   }
346
347   void print() {
348       print_dfs(0, 0, size, "");
349   }
350
351   void print() {
352       print_dfs(0, 0, size, "");
353   }
354
355   void print() {
356       print_dfs(0, 0, size, "");
357   }
358
359   void print() {
360       print_dfs(0, 0, size, "");
361   }
362
363   void print() {
364       print_dfs(0, 0, size, "");
365   }
366
367   void print() {
368       print_dfs(0, 0, size, "");
369   }
370
371   void print() {
372       print_dfs(0, 0, size, "");
373   }
374
375   void print() {
376       print_dfs(0, 0, size, "");
377   }
378
379   void print() {
380       print_dfs(0, 0, size, "");
381   }
382
383   void print() {
384       print_dfs(0, 0, size, "");
385   }
386
387   void print() {
388       print_dfs(0, 0, size, "");
389   }
390
391   void print() {
392       print_dfs(0, 0, size, "");
393   }
394
395   void print() {
396       print_dfs(0, 0, size, "");
397   }
398
399   void print() {
400       print_dfs(0, 0, size, "");
401   }
402
403   void print() {
404       print_dfs(0, 0, size, "");
405   }
406
407   void print() {
408       print_dfs(0, 0, size, "");
409   }
410
411   void print() {
412       print_dfs(0, 0, size, "");
413   }
414
415   void print() {
416       print_dfs(0, 0, size, "");
417   }
418
419   void print() {
420       print_dfs(0, 0, size, "");
421   }
422
423   void print() {
424       print_dfs(0, 0, size, "");
425   }
426
427   void print() {
428       print_dfs(0, 0, size, "");
429   }
430
431   void print() {
432       print_dfs(0, 0, size, "");
433   }
434
435   void print() {
436       print_dfs(0, 0, size, "");
437   }
438
439   void print() {
440       print_dfs(0, 0, size, "");
441   }
442
443   void print() {
444       print_dfs(0, 0, size, "");
445   }
446
447   void print() {
448       print_dfs(0, 0, size, "");
449   }
450
451   void print() {
452       print_dfs(0, 0, size, "");
453   }
454
455   void print() {
456       print_dfs(0, 0, size, "");
457   }
458
459   void print() {
460       print_dfs(0, 0, size, "");
461   }
462
463   void print() {
464       print_dfs(0, 0, size, "");
465   }
466
467   void print() {
468       print_dfs(0, 0, size, "");
469   }
470
471   void print() {
472       print_dfs(0, 0, size, "");
473   }
474
475   void print() {
476       print_dfs(0, 0, size, "");
477   }
478
479   void print() {
480       print_dfs(0, 0, size, "");
481   }
482
483   void print() {
484       print_dfs(0, 0, size, "");
485   }
486
487   void print() {
488       print_dfs(0, 0, size, "");
489   }
490
491   void print() {
492       print_dfs(0, 0, size, "");
493   }
494
495   void print() {
496       print_dfs(0, 0, size, "");
497   }
498
499   void print() {
500       print_dfs(0, 0, size, "");
501   }
502
503   void print() {
504       print_dfs(0, 0, size, "");
505   }
506
507   void print() {
508       print_dfs(0, 0, size, "");
509   }
510
511   void print() {
512       print_dfs(0, 0, size, "");
513   }
514
515   void print() {
516       print_dfs(0, 0, size, "");
517   }
518
519   void print() {
520       print_dfs(0, 0, size, "");
521   }
522
523   void print() {
524       print_dfs(0, 0, size, "");
525   }
526
527   void print() {
528       print_dfs(0, 0, size, "");
529   }
530
531   void print() {
532       print_dfs(0, 0, size, "");
533   }
534
535   void print() {
536       print_dfs(0, 0, size, "");
537   }
538
539   void print() {
540       print_dfs(0, 0, size, "");
541   }
542
543   void print() {
544       print_dfs(0, 0, size, "");
545   }
546
547   void print() {
548       print_dfs(0, 0, size, "");
549   }
550
551   void print() {
552       print_dfs(0, 0, size, "");
553   }
554
555   void print() {
556       print_dfs(0, 0, size, "");
557   }
558
559   void print() {
560       print_dfs(0, 0, size, "");
561   }
562
563   void print() {
564       print_dfs(0, 0, size, "");
565   }
566
567   void print() {
568       print_dfs(0, 0, size, "");
569   }
570
571   void print() {
572       print_dfs(0, 0, size, "");
573   }
574
575   void print() {
576       print_dfs(0, 0, size, "");
577   }
578
579   void print() {
580       print_dfs(0, 0, size, "");
581   }
582
583   void print() {
584       print_dfs(0, 0, size, "");
585   }
586
587   void print() {
588       print_dfs(0, 0, size, "");
589   }
590
591   void print() {
592       print_dfs(0, 0, size, "");
593   }
594
595   void print() {
596       print_dfs(0, 0, size, "");
597   }
598
599   void print() {
600       print_dfs(0, 0, size, "");
601   }
602
603   void print() {
604       print_dfs(0, 0, size, "");
605   }
606
607   void print() {
608       print_dfs(0, 0, size, "");
609   }
610
611   void print() {
612       print_dfs(0, 0, size, "");
613   }
614
615   void print() {
616       print_dfs(0, 0, size, "");
617   }
618
619   void print() {
620       print_dfs(0, 0, size, "");
621   }
622
623   void print() {
624       print_dfs(0, 0, size, "");
625   }
626
627   void print() {
628       print_dfs(0, 0, size, "");
629   }
630
631   void print() {
632       print_dfs(0, 0, size, "");
633   }
634
635   void print() {
636       print_dfs(0, 0, size, "");
637   }
638
639   void print() {
640       print_dfs(0, 0, size, "");
641   }
642
643   void print() {
644       print_dfs(0, 0, size, "");
645   }
646
647   void print() {
648       print_dfs(0, 0, size, "");
649   }
650
651   void print() {
652       print_dfs(0, 0, size, "");
653   }
654
655   void print() {
656       print_dfs(0, 0, size, "");
657   }
658
659   void print() {
660       print_dfs(0, 0, size, "");
661   }
662
663   void print() {
664       print_dfs(0, 0, size, "");
665   }
666
667   void print() {
668       print_dfs(0, 0, size, "");
669   }
670
671   void print() {
672       print_dfs(0, 0, size, "");
673   }
674
675   void print() {
676       print_dfs(0, 0, size, "");
677   }
678
679   void print() {
680       print_dfs(0, 0, size, "");
681   }
682
683   void print() {
684       print_dfs(0, 0, size, "");
685   }
686
687   void print() {
688       print_dfs(0, 0, size, "");
689   }
690
691   void print() {
692       print_dfs(0, 0, size, "");
693   }
694
695   void print() {
696       print_dfs(0, 0, size, "");
697   }
698
699   void print() {
700       print_dfs(0, 0, size, "");
701   }
702
703   void print() {
704       print_dfs(0, 0, size, "");
705   }
706
707   void print() {
708       print_dfs(0, 0, size, "");
709   }
710
711   void print() {
712       print_dfs(0, 0, size, "");
713   }
714
715   void print() {
716       print_dfs(0, 0, size, "");
717   }
718
719   void print() {
720       print_dfs(0, 0, size, "");
721   }
722
723   void print() {
724       print_dfs(0, 0, size, "");
725   }
726
727   void print() {
728       print_dfs(0, 0, size, "");
729   }
730
731   void print() {
732       print_dfs(0, 0, size, "");
733   }
734
735   void print() {
736       print_dfs(0, 0, size, "");
737   }
738
739   void print() {
740       print_dfs(0, 0, size, "");
741   }
742
743   void print() {
744       print_dfs(0, 0, size, "");
745   }
746
747   void print() {
748       print_dfs(0, 0, size, "");
749   }
750
751   void print() {
752       print_dfs(0, 0, size, "");
753   }
754
755   void print() {
756       print_dfs(0, 0, size, "");
757   }
758
759   void print() {
760       print_dfs(0, 0, size, "");
761   }
762
763   void print() {
764       print_dfs(0, 0, size, "");
765   }
766
767   void print() {
768       print_dfs(0, 0, size, "");
769   }
770
771   void print() {
772       print_dfs(0, 0, size, "");
773   }
774
775   void print() {
776       print_dfs(0, 0, size, "");
777   }
778
779   void print() {
780       print_dfs(0, 0, size, "");
781   }
782
783   void print() {
784       print_dfs(0, 0, size, "");
785   }
786
787   void print() {
788       print_dfs(0, 0, size, "");
789   }
790
791   void print() {
792       print_dfs(0, 0, size, "");
793   }
794
795   void print() {
796       print_dfs(0, 0, size, "");
797   }
798
799   void print() {
800       print_dfs(0, 0, size, "");
801   }
802
803   void print() {
804       print_dfs(0, 0, size, "");
805   }
806
807   void print() {
808       print_dfs(0, 0, size, "");
809   }
810
811   void print() {
812       print_dfs(0, 0, size, "");
813   }
814
815   void print() {
816       print_dfs(0, 0, size, "");
817   }
818
819   void print() {
820       print_dfs(0, 0, size, "");
821   }
822
823   void print() {
824       print_dfs(0, 0, size, "");
825   }
826
827   void print() {
828       print_dfs(0, 0, size, "");
829   }
830
831   void print() {
832       print_dfs(0, 0, size, "");
833   }
834
835   void print() {
836       print_dfs(0, 0, size, "");
837   }
838
839   void print() {
840       print_dfs(0, 0, size, "");
841   }
842
843   void print() {
844       print_dfs(0, 0, size, "");
845   }
846
847   void print() {
848       print_dfs(0, 0, size, "");
849   }
850
851   void print() {
852       print_dfs(0, 0, size, "");
853   }
854
855   void print() {
856       print_dfs(0, 0, size, "");
857   }
858
859   void print() {
860       print_dfs(0, 0, size, "");
861   }
862
863   void print() {
864       print_dfs(0, 0, size, "");
865   }
866
867   void print() {
868       print_dfs(0, 0, size, "");
869   }
870
871   void print() {
872       print_dfs(0, 0, size, "");
873   }
874
875   void print() {
876       print_dfs(0, 0, size, "");
877   }
878
879   void print() {
880       print_dfs(0, 0, size, "");
881   }
882
883   void print() {
884       print_dfs(0, 0, size, "");
885   }
886
887   void print() {
888       print_dfs(0, 0, size, "");
889   }
890
891   void print() {
892       print_dfs(0, 0, size, "");
893   }
894
895   void print() {
896       print_dfs(0, 0, size, "");
897   }
898
899   void print() {
900       print_dfs(0, 0, size, "");
901   }
902
903   void print() {
904       print_dfs(0, 0, size, "");
905   }
906
907   void print() {
908       print_dfs(0, 0, size, "");
909   }
910
911   void print() {
912       print_dfs(0, 0, size, "");
913   }
914
915   void print() {
916       print_dfs(0, 0, size, "");
917   }
918
919   void print() {
920       print_dfs(0, 0, size, "");
921   }
922
923   void print() {
924       print_dfs(0, 0, size, "");
925   }
926
927   void print() {
928       print_dfs(0, 0, size, "");
929   }
930
931   void print() {
932       print_dfs(0, 0, size, "");
933   }
934
935   void print() {
936       print_dfs(0, 0, size, "");
937   }
938
939   void print() {
940       print_dfs(0, 0, size, "");
941   }
942
943   void print() {
944       print_dfs(0, 0, size, "");
945   }
946
947   void print() {
948       print_dfs(0, 0, size, "");
949   }
950
951   void print() {
952       print_dfs(0, 0, size, "");
953   }
954
955   void print() {
956       print_dfs(0, 0, size, "");
957   }
958
959   void print() {
960       print_dfs(0, 0, size, "");
961   }
962
963   void print() {
964       print_dfs(0, 0, size, "");
965   }
966
967   void print() {
968       print_dfs(0, 0, size, "");
969   }
970
971   void print() {
972       print_dfs(0, 0, size, "");
973   }
974
975   void print() {
976       print_dfs(0, 0, size, "");
977   }
978
979   void print() {
980       print_dfs(0, 0, size, "");
981   }
982
983   void print() {
984       print_dfs(0, 0, size, "");
985   }
986
987   void print() {
988       print_dfs(0, 0, size, "");
989   }
990
991   void print() {
992       print_dfs(0, 0, size, "");
993   }
994
995   void print() {
996       print_dfs(0, 0, size, "");
997   }
998
999   void print() {
1000      print_dfs(0, 0, size, "");
1001  }
1002
1003  void print() {
1004      print_dfs(0, 0, size, "");
1005  }
1006
1007  void print() {
1008      print_dfs(0, 0, size, "");
1009  }
1010
1011  void print() {
1012      print_dfs(0, 0, size, "");
1013  }
1014
1015  void print() {
1016      print_dfs(0, 0, size, "");
1017  }
1018
1019  void print() {
1020      print_dfs(0, 0, size, "");
1021  }
1022
1023  void print() {
1024      print_dfs(0, 0, size, "");
1025  }
1026
1027  void print() {
1028      print_dfs(0, 0, size, "");
1029  }
1030
1031  void print() {
1032      print_dfs(0, 0, size, "");
1033  }
1034
1035  void print() {
1036      print_dfs(0, 0, size, "");
1037  }
1038
1039  void print() {
1040      print_dfs(0, 0, size, "");
1041  }
1042
1043  void print() {
1044      print_dfs(0, 0, size, "");
1045  }
1046
1047  void print() {
1048      print_dfs(0, 0, size, "");
1049  }
1050
1051  void print() {
1052      print_dfs(0, 0, size, "");
1053  }
1054
1055  void print() {
1056      print_dfs(0, 0, size, "");
1057  }
1058
1059  void print() {
1060      print_dfs(0, 0, size, "");
1061  }
1062
1063  void print() {
1064      print_dfs(0, 0, size, "");
1065  }
1066
1067  void print() {
1068      print_dfs(0
```

```

55     };
56 }
57 void init (vi &a){
58     int n = a.size();
59     size = 1;
60     while (size < n) size *= 2;
61     nodes.assign(2*size-1, NEUTRAL);
62     hasLazy.assign(2*size-1, false);
63     lazy.assign(2*size-1, 0);
64     build(0,0,size,a);
65 }
66 }
67 void build(int x, int lx, int rx, vi &a){
68     if (rx-lx == 1){
69         if (lx < a.size()) nodes[x] = single(a[lx]);
70         return;
71     }
72     int mx = (lx+rx)/2;
73     build(2*x+1,lx,mx,a);
74     build(2*x+2,mx,rx,a);
75     nodes[x] = merge(nodes[2*x+1], nodes[2*x+2]);
76 }
77 }
78 void set(int i, int v, int x, int lx, int rx){
79     if (rx-lx == 1){
80         nodes[x] = single(v);
81         return;
82     }
83     int mx = (lx+rx)/2;
84     if (i < mx) set(i, v, 2*x+1, lx, mx);
85     else set(i, v, 2*x+2, mx, rx);
86     nodes[x] = merge(nodes[2*x+1], nodes[2*x+2]);
87 }
88 }
89 void set(int i, int v){
90     set(i, v, 0, 0, size);
91 }
92 }
93 void rangeUpdate(int l, int r, int v){
94     rangeUpdate(l,r,v,0,0,size);
95 }
96 }
97 void rangeUpdate(int l, int r, int v, int x, int lx,
98     int rx){
99     unlazy(x, lx, rx);
100    if (rx-lx < 1 || rx <= l || lx >= r) return;
101    if (l <= lx && rx <= r) return propagate(x, lx, rx, v);
102    ;
103    int mx = (lx+rx)/2;
104    rangeUpdate(l,r,v,2*x+1,lx,mx);
105    rangeUpdate(l,r,v,2*x+2,mx,rx);
106    nodes[x] = merge(nodes[2*x+1], nodes[2*x+2]);
107 }
108 node query(int l, int r){
109     return query(l,r,0,0,size);
110 }
111 }
112 node query(int l, int r, int x, int lx, int rx){
113     unlazy(x, lx, rx);
114     if (rx-lx < 1 || rx <= l || lx >= r) return NEUTRAL;
115     ;
116     if (l <= lx && rx <= r) return nodes[x];
117     int mx = (lx+rx)/2;
118     node left = query(l,r,2*x+1,lx,mx);
119     node right = query(l,r,2*x+2,mx,rx);
120     return merge(left,right);
121 }

```

```

122     void unlazy(int x, int lx, int rx){
123         if (hasLazy[x]){
124             propagate(x, lx, rx, lazy[x]);
125             hasLazy[x] = false;
126         }
127     }
128     void propagate(int x, int lx, int rx, int v){
129         nodes[x].sum = (rx-lx)*v;
130         nodes[x].seg = max((rx-lx)*v,0ll);
131         nodes[x].pre = max((rx-lx)*v,0ll);
132         nodes[x].suf = max((rx-lx)*v,0ll);
133         if (rx-lx > 1){
134             lazy[2*x+1] = v;
135             lazy[2*x+2] = v;
136             hasLazy[2*x+1] = true;
137             hasLazy[2*x+2] = true;
138         }
139     }
140 }
141 }

```

1.6 Sparse Table RMQ

Sparse table for RMQ in $\mathcal{O}(1)$, used in many problems, including $\mathcal{O}(1)$ LCA (Trees) and LCP (SuffixArray) queries.

```

1 struct SparseTable {
2     vector<vector<ii>> st;
3
4     void build(const vi &a) {
5         int n = a.size();
6         int max_log = __bit_width(n);
7         st.assign(max_log, vector<ii>(n));
8         for (int i = 0; i < n; i++) {
9             st[0][i] = {a[i], i};
10        }
11        for (int i = 1; i < max_log; i++) {
12            for (int j = 0; j + (1 << i) <= n; j++) {
13                // Combine the two halves
14                st[i][j] = std::min(st[i-1][j], st[i-1][j + (1
15                    << (i-1))]);
16            }
17        }
18    }
19
20    // Returns min value and index in range [l, r]
21    // inclusive
21    ii min(int l, int r) {
22        int len = r - l + 1;
23        int k = __bit_width(len) - 1;
24        return std::min(st[k][l], st[k][r - (1<<k) + 1]);
25    }
26 }

```

2 Graphs

2.1 BFS 0-1

Time: $\mathcal{O}(n + m)$

```

1 vi bfs01(int s){
2     vi d(n, INF);

```

```

3     d[s] = 0;
4     deque<int> q;
5     q.push_front(s);
6     while(!q.empty()){
7         int u = q.front(); q.pop_front();
8         for (auto [w,v] : adj[u]){
9             if (d[u]+w < d[v]){
10                 d[v] = d[u] + w;
11                 if (w == 1) q.push_back(v);
12                 else q.push_front(v);
13             }
14         }
15     }
16 }
17 }

```

2.2 Dijkstra

Time: $\mathcal{O}(m \log n)$

```

1 void dijkstra(int s){
2     int d, u, v;
3     dist = vi(n, INF);
4     dist[s] = 0;
5     priority_queue<ii, vi, greater<ii>> pq;
6     pq.emplace(0,s);
7     while(!pq.empty()){
8         auto [d,u] = pq.top(); pq.pop();
9         if (d > dist[u]) continue;
10        for (auto &[w,v] : adj[u]){
11            if (dist[v] > dist[u] + w){
12                dist[v] = dist[u] + w;
13                pq.emplace(dist[v], v);
14            }
15        }
16    }
17 }

```

2.3 Dinic - Flow/matchings

- General Network: $\mathcal{O}(VE \log U)$.
- Unit Capacity Network: $\mathcal{O}(\min(V^{2/3}, E^{1/2}) \cdot E)$. Often considered $\mathcal{O}(E\sqrt{V})$.
- Bipartite Matching: $\mathcal{O}(\min(V^{2/3}, E^{1/2}) \cdot E)$. Often considered $\mathcal{O}(E\sqrt{V})$.

```

1 struct Dinic {
2     struct Edge {
3         int u, v;
4         ll cap, flow = 0;
5         Edge(int u, int v, ll cap) : u(u), v(v), cap(cap)
6         {}
7     };
8     const ll flow_inf = 1e18;
9     vector<Edge> edges;
10    vvi adj;
11    int n, m = 0;
12    int s, t;
13    vi level, ptr;

```

```

queue<int> q;
Dinic(int n): n(n) {
    adj.resize(n);
    level.resize(n);
    ptr.resize(n);
}
void add_edge(int u, int v, ll cap) {
    edges.emplace_back(u,v,cap);
    edges.emplace_back(v,u,0);
    adj[u].push_back(m++);
    adj[v].push_back(m++);
}
bool bfs(ll delta){
    queue<int> q;
    q.push(s);
    while(!q.empty()){
        int u = q.front(); q.pop();
        for (int id : adj[u]){
            auto &e = edges[id];
            if (e.cap - e.flow < delta) continue;
            if (level[e.v] != -1) continue;
            level[e.v] = level[u]+1;
            q.push(e.v);
        }
    }
    return level[t] != -1;
}
ll dfs(int u, ll pushed) {
    if (pushed == 0) return 0;
    if (u == t) return pushed;
    for (int &cid = ptr[u]; cid < (int)adj[u].size(); cid++){
        int id = adj[u][cid];
        auto &e = edges[id];
        if (level[u]+1 != level[e.v]) continue;
        ll tr = dfs(e.v,min(pushed, e.cap - e.flow));
        if (tr == 0) continue;
        e.flow += tr;
        edges[id^1].flow -= tr;
        return tr;
    }
    return 0;
}
ll maxflow(int s, int t){
    this->s = s; this->t = t;
    ll max_c = 0;
    for (auto &e : edges) max_c = max(max_c, e.cap);
    ll delta = 1;
    while(delta <= max_c) delta <= 1;
    delta >= 1;
    ll f = 0;
    for (;delta > 0; delta >>= 1){
        while(true){
            fill(level.begin(), level.end(), -1);
            level[s] = 0;
            if (!bfs(delta)) break;
            fill(ptr.begin(), ptr.end(), 0);
            while(ll pushed = dfs(s,flow_inf)) f += pushed;
        }
    }
    return f;
}
// call constructor with (n1+n2+2) beforehand (don't
// add edges manually)
// assumes pairs are 1-indexed
vii maxmatchings(int n1, int n2, const vii& pairs){
    for (int i = 1; i <= n1; i++)
        add_edge(0,i,1);
    for (int i = 1; i <= n2; i++)
        add_edge(i+n1,n-1,1);
    for (auto &[u,v] : pairs)
        add_edge(u,v+n1,1);
    maxflow(0,n-1);
}
vii matchings;
for (auto &e : edges){
    if (e.u >= 1 && e.u <= n1 && e.flow == 1 && e.v <= n1){
        matchings.emplace_back(e.u,e.v-n1);
    }
}
return matchings;
}
vii mincut(int s, int t){
    maxflow(s,t);
    queue<int> q; q.push(s);
    vector<bool> reachable(n);
    reachable[s] = true;
    while(!q.empty()){
        int u = q.front(); q.pop();
        for (auto &id : adj[u]){
            int v = edges[id].v;
            if (edges[id].cap - edges[id].flow > 0 && !reachable[v]){
                reachable[v] = true;
                q.push(v);
            }
        }
    }
    vii minCutEdges;
    for (int i = 0; i < m; i += 2) {
        const Edge& edge = edges[i];
        if (reachable[edge.u] && !reachable[edge.v])
            minCutEdges.emplace_back(edge.u, edge.v);
    }
    return minCutEdges;
}

```

2.4 Floyd-Warshall

Time: $\mathcal{O}(n^3)$

```

1
2 vvi d(n, vi(n, INF));
3 void floyd_marshall(){
4     for (int k = 0; k < n; k++)
5         for (int i = 0; i < n; i++)
6             for (int j = 0; j < n; j++)
7                 d[i][j] = min(d[i][j], d[i][k]+d[k][j])
8 }
```

2.5 Hopcroft-Karp - Bipartite Matching

Bipartite matching such as Kuhn but faster. BFS until first layer missing match, DFS for the BFS graph to find pairings. Time: $\mathcal{O}(E\sqrt{V})$

```

1 int n, m, k;
2 vvi adj;
3 vi p, dist; /*p is in matching for [0, n[ and parent
   for [n, n+m[*/
4
5 int bfs(){
6     queue<int> q;
7     dist = vi(n+m, inf);
8     for(int i = 0; i < n; i++){
9         if(p[i] == -1) q.push(i), dist[i] = 0;
10    }
11    int min_dist_match = inf;
12    while(!q.empty()){
13        int u = q.front(); q.pop();
14        if(dist[u] > min_dist_match) continue;
15        for(auto v: adj[u]){
16            if(p[v] == -1) min_dist_match = dist[u];
17            else if(dist[p[v]] == inf){
18                dist[p[v]] = dist[u] + 1;
19                q.push(p[v]);
20            }
21        }
22    }
23    return min_dist_match != inf;
24 }
25
26 int dfs(int u){
27     for(auto v: adj[u]){
28         if(p[v] == -1 || (dist[u]+1 == dist[p[v]] && dfs(p[
29             v))){
30             p[v] = u;
31             p[u] = 1;
32             return true;
33         }
34     }
35     dist[u] = inf;
36     return false;
37 }
38
39 int hopkarp(){
40     p = vi(n+m, -1);
41     int matchings = 0;
42     while(bfs()){
43         for(int i = 0; i < n; i++){
44             if(p[i] == -1 && dfs(i)) matchings++;
45         }
46     }
47     return matchings;
48 }
49 void create(){
50     adj = vvi(n+m);
51     for(int i = 0; i < k; i++){
52         int u, v;
53         cin >> u >> v; u--; v--;
54         v += n;
55         adj[u].push_back(v);
56     }
57 }
```

2.6 Hungarian

Solves minimum cost assignment for n workers and m jobs.
Time: $\mathcal{O}((n+m)^3)$

```

1 // cost should be (cost[worker][job])
2 pair<int,vvi> hungarian(int n, int m, const vvi &cost)
3 {
4     if (n == 0) return {0,{}};
5     int N = max(n, m);
6
7     vi u(N+1), v(N+1), p(N+1), way(N+1);
8
9     const int INF = 1e9;
10    for (int i = 1; i <= n; ++i) {
11        p[0] = i;
12        int j0 = 0;
13        vi minv(N + 1, INF);
14        vector<bool> used(N + 1, false);
15
16        do {
17            used[j0] = true;
18            int i0 = p[j0], delta = INF, j1;
19
20            for (int j = 1; j <= N; ++j) {
21                if (!used[j]) {
22                    int cur = cost[i0-1][j-1] - u[i0] - v[j];
23                    if (cur < minv[j])
24                        minv[j] = cur;
25                    way[j] = j0;
26                }
27                if (minv[j] < delta) {
28                    delta = minv[j];
29                    j1 = j;
30                }
31            }
32
33            for (int j = 0; j <= N; ++j) {
34                if (used[j]) {
35                    u[p[j]] += delta;
36                    v[j] -= delta;
37                } else {
38                    minv[j] -= delta;
39                }
40            }
41            j0 = j1;
42        } while (p[j0] != 0);
43
44        do {
45            int j1 = way[j0];
46            p[j0] = p[j1];
47            j0 = j1;
48        } while (j0);
49    }
50
51    int total_cost = 0;
52    for (int j = 1; j <= m; ++j) {
53        if (p[j] != 0) {
54            total_cost += cost[p[j] - 1][j - 1];
55        }
56    }
57
58 // {worker, job}[] 0-indexed
59 vii matchings;
60 for (int j = 1; j <= m; ++j) {
61     if (p[j] != 0) {
62         matchings.push_back({p[j] - 1, j - 1});
63     }
64 }
```

2.7 Kosaraju - SCCs

Computes the strongly connected components of a graph.
Also computes the reverse topological order (if it exists).
Time: $\mathcal{O}(n+m)$

```

1 void dfs1(int u){
2     vis[u] = 1;
3     for (auto v : adj[u]){
4         if (!vis[v]) dfs1(v);
5     }
6
7     ts.push_back(u);
8 }
9
10 void dfs2(int u, int c){
11     scc[u] = c;
12     for (auto v : adjT[u]){
13         if (!scc[v]) dfs2(v,c);
14     }
15
16 // usage
17 for (int i = 0; i < n; i++)
18     if (!vis[i]) dfs1(i);
19
20 reverse(ts.begin(), ts.end());
21
22 int c = 1;
23 for (auto u : ts)
24     if (!scc[u]) dfs2(u,c++);
25 }
```

2.8 Kuhn - Bipartite Matching

Bipartite matching. Time: $\mathcal{O}(VE)$

```

1 int matchings;
2 vi p, vis;
3 vii match;
4
5 int dfs(int u){
6     if(vis[u]) return 0;
7     vis[u] = 1;
8     for(auto v: adj[u]){
9         if(p[v] == -1 || dfs(p[v])){
10             p[v] = u;
11             return 1;
12         }
13     }
14     return 0;
15 }
16
17 void kuhn(){
18     matchings = 0;
19     p = vi(n+m, -1);
20     for(int i = 0; i < n; i++){
21         vis = vi(n, 0);
22         matchings += dfs(i);
23     }
24     for(int i = n; i < n+m; i++){
25         if(p[i] != -1) match.push_back(ii(p[i], i));
26     }
27 }
28
29 void create(){
30     adj = vvi(n+m);
31     for(int i = 0; i < k; i++){
32         int u, v;
33         cin >> u >> v; u--; v--;
34         adj[u].push_back(v+n);
35     }
36 }
```

```

25     if(p[i] != -1) match.push_back(ii(p[i], i));
26 }
27 }
28
29 void create(){
30     adj = vvi(n+m);
31     for(int i = 0; i < k; i++){
32         int u, v;
33         cin >> u >> v; u--; v--;
34         adj[u].push_back(v+n);
35     }
36 }
```

2.9 Min cost flow

Time: $\mathcal{O}(FE \log V)$

If negative costs are needed (maximize cost), need to run SPFA once at the start, making the solution $\mathcal{O}(EV + FE \log V)$.

```

1 struct MinCostFlow {
2     struct Edge {
3         int to, capacity, rev;
4         ll cost;
5     };
6     int n;
7     vector<vector<Edge>> adj;
8
9     MinCostFlow(int _n) : n(_n), adj(_n) {}
10
11     void add_edge(int from, int to, int cap, ll cost){
12         adj[from].push_back({to,cap,(int)adj[to].size(),
13                             cost});
14         adj[to].push_back({from,0,(int)adj[from].size()-1,
15                           -cost});
15     }
16
17 // O(FE log(V))
18 lli min_cost_flow(int s, int t, int targetFlow) {
19     int flow = 0;
20     ll total_cost = 0;
21     vll dist, h(n);
22     vi pv, pe;
23
24     // needed only if negative costs exists
25     spfa(s, h, pv, pe);
26
27     while (flow < targetFlow) {
28         dijkstra(s, h, dist, pv, pe);
29
30         if (dist[t] == INF) break;
31
32         for (int i = 0; i < n; i++) {
33             if (dist[i] < INF) {
34                 h[i] += dist[i];
35             }
36         }
37
38         int f = targetFlow - flow;
39         int cur = t;
40
41         while (cur != s) {
42             f = min(f, adj[pv[cur]][pe[cur]].capacity);
43             cur = pv[cur];
44         }
45     }
46 }
```

```

45     flow += f;
46     total_cost += f * h[t];
47     cur = t;
48     while (cur != s) {
49       Edge &e = adj[pv[cur]][pe[cur]];
50       e.capacity -= f;
51       adj[e.to][e.rev].capacity += f;
52       cur = pv[cur];
53     }
54
55   return {total_cost, flow};
56 }
57
58 // needed only if negative costs exists
59 void spfa(int s, vll &dist, vi &pv, vi &pe) {
60   dist.assign(n, INF);
61   pv.assign(n, -1);
62   pe.assign(n, -1);
63   vector<bool> inq(n, false);
64   queue<int> q;
65
66   dist[s] = 0;
67   q.push(s);
68   inq[s] = true;
69
70   while (!q.empty()) {
71     int u = q.front(); q.pop();
72     inq[u] = false;
73     for (int i = 0; i < adj[u].size(); i++) {
74       Edge &e = adj[u][i];
75       int v = e.to;
76       if (e.capacity > 0 && dist[v] > dist[u] + e.cost) {
77         dist[v] = dist[u] + e.cost;
78         pv[v] = u;
79         pe[v] = i;
80         if (!inq[v]) {
81           inq[v] = true;
82           q.push(v);
83         }
84       }
85     }
86   }
87
88 }
89
90 void dijkstra(int s, vll &h, vll &dist, vi &pv, vi &
91   pe) {
92   dist.assign(n, INF);
93   pv.assign(n, -1);
94   pe.assign(n, -1);
95   dist[s] = 0;
96
97   priority_queue<lli, vector<lli>, greater<lli>> pq;
98   pq.emplace(0, s);
99
100  while (!pq.empty()) {
101    auto [d, u] = pq.top(); pq.pop();
102    if (d > dist[u]) continue;
103
104    for (int i = 0; i < adj[u].size(); i++) {
105      Edge &e = adj[u][i];
106      if (e.capacity <= 0) continue;
107      int v = e.to;
108
109      ll reduced_cost = e.cost + h[u] - h[v];
110      if (dist[u] != INF && dist[v] > dist[u] +
111          reduced_cost) {
112        dist[v] = dist[u] + reduced_cost;
113        pq.push({dist[v], v});
114      }
115    }
116  }
117 }
118 }
119 }
120
121 // usage
122 int nodes = 302; // amount of nodes in the network
123 MinCostFlow mcf(nodes);
124
125 for (int i = 0; i < 150; i++){
126   mcf.add_edge(0, i+1, 1, 0); // source to node
127   mcf.add_edge(i+151, nodes-1, 1, 0); // node to sink
128 }
129
130 for (int i = 0; i < n; i++){
131   int a, b, c; cin >> a >> b >> c;
132   mcf.add_edge(a, b+150, 1, -c); // edges in between (-c to maximize the cost)
133 }
134
135 // final max cost is -cost
136 auto [cost, flow] = mcf.min_cost_flow(0,nodes-1,150);

```

2.10 MST - Kruskal

Time: $\mathcal{O}(m \log m)$

```

1 vector<pair<int,ii>> edges; // [weight, (u,v)]
2 int kruskal(int n){
3   int cost = 0;
4   DSU dsu(n); // n is the num of vertices
5   sort(edges.begin(), edges.end());
6   for (auto &[w,uv] : edges){
7     auto [u,v] = uv;
8     if (dsu.unite(u,v)) cost += w;
9   }
10  return cost;
11 }

```

2.11 MST - Prim

Time: $\mathcal{O}(m \log n)$

```

1 vvii adj, mst;
2 vi taken;
3
4 int prim(){
5   priority_queue<iii, vector<iii>, greater<iii>> pq;
6   taken[0] = 1;
7   for (auto [w,v] : adj[0]){
8     if (!taken[v]) pq.push({w, {0,v}});
9   }
10
11   int cost = 0;
12   while (!pq.empty()){
13     auto [w,pv] = pq.top(); pq.pop();
14     auto [p,u] = pv;
15     if (!taken[u]){
16       cost += w;
17       mst[p].emplace_back(w,u);
18       mst[u].emplace_back(w,p);
19     }
20   }
21 }

```

```

19
20   taken[u] = 1;
21   for (auto [w,v] : adj[u]){
22     if (!taken[v]) {
23       pq.push({w,{u,v}});
24     }
25   }
26 }
27
28 } return cost;
29

```

3 DP

3.1 Bin Packing

Time: $\mathcal{O}(n \cdot 2^n)$ Space: $\mathcal{O}(2^n)$

```

1 vi w(n);
2
3 vector<ii> dp(1<<n, ii(INF,0));
4 // dp[i] = for the subset i(bitmap) (A,B) is the pair
5 // where
6 // A - the min number of knapsacks to store this subset
7 // B - the min size of a used knapsack
8
9 dp[0] = ii(0,INF);
10 for (int subset = 1; subset < (1<<n); subset++){
11   for (int item = 0; item < n; item++){
12     if (!((subset>>item)&1)) continue;
13     int prevsubset = subset - (1<<item);
14     ii prev = dp[prevsubset];
15
16     if (prev.second + w[item] <= x) {
17       // can fill the knapsack, fill it
18       dp[subset] = min(dp[subset], ii(prev.first, prev.
19         second+w[item]));
20     } else {
21       // cant fill the knapsack, create a new one
22       dp[subset] = min(dp[subset], ii(prev.first+1, w[
23         item]));
24     }
25   }
26 }
27
28 cout << dp[(1<<n)-1].first << endl;

```

3.2 Broken Profile DP

Solves the problem of counting how many ways to fill an $n \times m$ grid using 1×2 tiles. This technique can be used whenever the state dependence is only on the previous state (column). Time: $\mathcal{O}(mn2^n)$ Space: $\mathcal{O}(mn2^n)$

```

1 int dp[1002][12][1024];
2 dp[0][0][0] = 1;
3
4 for (int i = 0; i < m; i++){
5   for (int j = 0; j < n; j++){
6     for (int mask = 0; mask < (1<<n); mask++){
7       if (mask & (1<<j)){
8         int nxt_mask = mask - (1<<j);
8         dp[i][j+1][nxt_mask] += dp[i][j][mask];
9       }
9     }
9   }
9 }

```

```

10     dp[i][j+1][nxt_mask] %= M;
11 } else {
12     int q = mask + (1 << j);
13     dp[i][j+1][q] += dp[i][j][mask];
14     dp[i][j+1][q] %= M;
15     if (j < n-1 && (mask & (1<<(j+1)))==0) {
16         q = mask + (1 << (j+1));
17         dp[i][j+1][q] += dp[i][j][mask];
18         dp[i][j+1][q] %= M;
19     }
20 }
21 }
22 }
23 for (int p = 0; p < (1<<n); p++){
24     dp[i+1][0][p] = dp[i][n][p];
25 }
26 }
27 }
```

3.3 Convex Hull Trick (CHT)

- Recurrence form:

TODO formulas

- **Slope monotonicity:** If coefficients a_j (slopes) are inserted in strictly decreasing (or increasing) order as j grows, and
- **Query monotonicity:** Values x_i for query come in non-decreasing (min) or increasing (max) order consistent with slope order,

- Complexity:

- Insertion + amortized query in $\mathcal{O}(1)$ per operation (pointer walk) under monotonicity.
- Non-monotonic case, generic CHT via binary search: $\mathcal{O}(\log n)$ per query.
- General alternative: Li Chao Tree for insertions/queries in arbitrary order, $\mathcal{O}(\log M)$ per operation (where M is the domain of x).

- Constraints:

- If it cannot be written in linear form, CHT does not apply.
- If there is no monotonicity of slopes or queries, consider Li Chao Tree or CHT variant with binary search.

The example below solves the dp where the recurrence is:

TODO formulas

```

1 struct CHT {
2     struct Line { // y = mx + c
3         int m, c;
```

```

4     Line(int m, int c) : m(m), c(c) {}
5     int val(int x){ return m*x + c; }
6 }
7
8 int floorDiv(int num, int den) {
9     if (den < 0) num = -num, den = -den;
10    if (num >= 0) return num / den;
11    else return -((-num + den - 1) / den);
12 }
13 int ceilDiv(int num, int den) {
14     if (den < 0) num = -num, den = -den;
15     if (num >= 0) return (num + den - 1) / den;
16     else return -((-num) / den);
17 }
18 int intersect(Line l){
19     // m1x + c1 = m2x + c2
20     // x = (c2 - c1)/(m1 - m2)
21     // if slopes are increasing, use floor div
22     return ceilDiv(l.c - c, m - l.m);
23 }
24 }
25
26 deque<pair<Line, int>> dq;
27
28 void insert(int m, int c){
29     Line newLine(m, c);
30     if (!dq.empty() && newLine.m == dq.back().first.m) {
31         // If slopes increasing, change to <=
32         if (newLine.c >= dq.back().first.c) return;
33         else dq.pop_back();
34     }
35     // if slopes increasing, change to <=
36     while (dq.size() > 1 && dq.back().second >= dq.back()
37         .first.intersect(newLine)){
38         dq.pop_back();
39     }
40     if (dq.empty()){
41         // assuming queries are positive numbers, may
42         // change to -INF or +INF if needed
43         dq.emplace_back(newLine, 0);
44         return;
45     }
46     dq.emplace_back(newLine, dq.back().first.intersect(
47         newLine));
48 }
49
50 // dont use query and queryNonMonotonicValues in the
51 // same problem
52 int query(int x){
53     while (dq.size() > 1){
54         // if slopes increasing, change to >=
55         if (dq[1].second <= x) dq.pop_front();
56         else break;
57     }
58     return dq[0].first.val(x);
59 }
60
61 int queryNonMonotonicValues(int x){
62     int l=0, r=dq.size()-1, ans=0;
63     while (l <= r) {
64         int mid = (l+r)>>1;
65         if (dq[mid].second <= x) {
66             ans = mid;
67             l = mid + 1;
68         } else {
69             r = mid - 1;
70         }
71     }
72 }
```

```

69     return dq[ans].first.val(x);
70 }
71 }
72
73 void solve(){
74     int n, c; cin >> n >> c;
75     vi h(n);
76     for (auto &x : h) cin >> x;
77
78     vi dp(n);
79     dp[0] = 0;
80     CHT cht;
81     cht.insert(-2*h[0], h[0]*h[0]);
82     for (int i = 1; i < n; i++){
83         dp[i] = cht.query(h[i]) + c + h[i]*h[i];
84         cht.insert(-2*h[i], h[i]*h[i] + dp[i]);
85     }
86     cout << dp[n-1] << endl;
87 }
```

3.4 Edit Distance (Levenshtein)

Very similar to LCS, in the sense that it considers prefixes already computed. Time: $\mathcal{O}(mn)$ Space: $\mathcal{O}(mn)$

```

1 vvi dp(n+1, vi(m+1));
2 for (int i = 0; i <= n; i++) dp[i][0] = i;
3 for (int i = 0; i <= m; i++) dp[0][i] = i;
4 for (int i = 1; i <= n; i++){
5     for (int j = 1; j <= m; j++){
6         dp[i][j] = min(
7             min(dp[i][j-1]+1, dp[i-1][j]+1),
8             dp[i-1][j-1]+(s[i-1]!=t[j-1]));
9     }
10 }
```

3.5 Knapsack - 1D

The spirit here is the same as the 2D version, but here it iterates on the knapsack capacity backwards, to ensure that the value of $dp[j-w[i]]$ is not considering the item i . Time: $\mathcal{O}(nW)$ Space: $\mathcal{O}(W)$

```

1 vi dp(W+1);
2 for (int i = 0; i < n; i++){
3     for (int j = W; j >= w[i]; j--){
4         dp[j] = max(dp[j], v[i] + dp[j-w[i]]);
5     }
6 }
```

3.6 Knapsack - 2D

Time: $\mathcal{O}(nW)$ Space: $\mathcal{O}(nW)$

```

1 vvi dp(n+1, vi(W+1));
2 for (int c = 1; c <= W; c++){
3     for (int i = 1; i <= n; i++){
4         dp[i][c] = dp[i-1][w];
5         if (c-w[i] >= 0) {
```

```

6     dp[i][c] = max(dp[i][c], dp[i-1][c-w[i-1]] + v[i
7   }
8 }
9 }
```

3.7 LCS - Longest Common Subsequence

Subsequence generation included here. Time: $\mathcal{O}(mn)$
Space: $\mathcal{O}(mn)$

```

1 vvi dp(n+1,vi(m+1));
2 vvi p(n+1,vii(m+1));
3
4 for (int i = 1; i <= n; i++) {
5   for (int j = 1; j <= m; j++) {
6     if (a[i-1] == b[j-1]) {
7       dp[i][j] = dp[i-1][j-1]+1;
8       p[i][j] = {i-1,j-1};
9     } else if (dp[i][j-1] > dp[i-1][j]) {
10      dp[i][j] = dp[i][j-1];
11      p[i][j] = {i,j-1};
12    } else {
13      dp[i][j] = dp[i-1][j];
14      p[i][j] = {i-1,j};
15    }
16  }
17 }
18 int pos = ii(n,m);
19 stack<int> st;
20 while(pos != ii(0,0)){
21   auto [i,j] = pos;
22   if (p[i][j] == ii(i-1,j-1)) st.push(a[i-1]);
23   pos = p[i][j];
24 }
25 cout << st.size() << endl;
26 while (!st.empty()) {
27   cout << st.top() << ' ';
28   st.pop();
29 }
30 cout << endl;
```

3.8 LiChao Tree

Generalization of CHT for linear functions that do not need to be sorted. Inspired by segtree. Queries and insertions are all $\mathcal{O}(\log M)$. Where M is the size of the query interval the tree receives.

```

1 // Li Chao tree for minimum (or maximum) over domain [L
2   , R].
3 // T should support +, *, comparisons.
4 // For integer x use eps = 0 and discrete mid+1
5   splitting;
6 // For floating use eps > 0 and continuous splitting
7   without +1.
8 template<typename T>
9 struct lichao_tree {
7   // if max lichao, change to ::min()
8   static const T identity = numeric_limits<T>::max();
9
10  struct Line {
```

```

11    T m, c;
12    Line() {
13      m = 0;
14      c = identity;
15    }
16    Line(T m, T c) : m(m), c(c) {}
17    T val(T x) { return m * x + c; }
18  };
19  struct Node {
20    Line line;
21    Node *lc, *rc;
22    Node() : lc(0), rc(0) {}
23  };
24
25  T L, R, eps;
26  deque<Node> buffer;
27  Node* root;
28
29  Node* new_node() {
30    buffer.emplace_back();
31    return &buffer.back();
32  }
33
34  lichao_tree() {}
35
36  lichao_tree(T _L, T _R, T _eps) {
37    init(_L, _R, _eps);
38  }
39
40  void clear() {
41    buffer.clear();
42    root = nullptr;
43  }
44
45  void init(T _L, T _R, T _eps) {
46    clear();
47    L = _L;
48    R = _R;
49    eps = _eps;
50    root = new_node();
51  }
52
53  void insert(Node* &cur, T l, T r, Line line) {
54    if (!cur) {
55      cur = new_node();
56      cur->line = line;
57      return;
58    }
59
60    T mid = l + (r - 1) / 2;
61    if (r - 1 <= eps) return;
62
63    // if max lichao, change to >
64    if (line.val(mid) < cur->line.val(mid))
65      swap(line, cur->line);
66
67    // if max lichao, change to >
68    if (line.val(l) < cur->line.val(l)) insert(cur->lc,
69      l, mid, line);
70    else insert(cur->rc, mid + 1, r, line);
71
72  }
73  T query(Node* &cur, T l, T r, T x) {
74    if (!cur) return identity;
75
76    T mid = l + (r - 1) / 2;
77    T res = cur->line.val(x);
78    if (r - 1 <= eps) return res;
79
80    // if max lichao, change min to max
81    if (x <= mid) return min(res, query(cur->lc, l,
82      mid, x));
83    else return min(res, query(cur->rc, mid + 1, r, x));
84  }
85  void insert(T m, T c) { insert(root, L, R, Line(m, c));
86  }
87  T query(T x) { return query(root, L, R, x); }
88 }
```

3.9 LIS - Longest Increasing Subsequence

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

```

1 int lis(vi &a){
2   int n = a.size();
3   vi len(n+1, INF);
4   len[0] = -INF;
5   for (int i = 0; i < n; i++) {
6     int l = upper_bound(len.begin(), len.end(), a[i]
7       ) - len.begin();
8     if (len[l-1] < a[i] && a[i] < len[l]) len[l] = a
9       [i];
10  }
11  int ans = 0;
12  for (int i = 0; i <= n; i++) {
13    if (len[i] < INF) ans = i;
14  }
15  return ans;
16 }
```

3.10 SOSDP

```

1 int k; // amount of bits
2 vi a(1<<k);
3 // sosdp
4 for (int bit = 0; bit < k; bit++) {
5   for (int mask = 0; mask < (1<<k); mask++) {
6     if ((1<<bit) & mask) {
7       a[mask] += a[mask ^ (1<<bit)];
8     }
9   }
10 }
11 // do stuff (such as multiplication for OR convolution)
13
14 // sosdp inverse
15 for (int bit = 0; bit < k; bit++) {
16   for (int mask = 0; mask < (1<<k); mask++) {
17     if ((1<<bit) & mask) {
18       a[mask] -= a[mask ^ (1<<bit)];
19     }
20 }
```

3.11 Subset Sum

Almost identical to Knapsack, this code contains the subset reconstruction. Time: $\mathcal{O}(nS)$ Space: $\mathcal{O}(nS)$

```

1 vvi dp(n+1,vi(sum+1));
2 vvi p(n+1,vi(sum+1));
3
4 dp[0][0] = 1;
5
6 for (int i = 1; i <= n; i++){
7     for (int s = 1; s <= sum; s++) {
8         if (s-a[i-1] >= 0 && dp[i-1][s-a[i-1]]) {
9             // sum is possible taking item i
10            p[i][s] = {i-1,s-a[i-1]};
11            dp[i][s] = 1;
12        } else if (dp[i-1][s]) {
13            // sum not possible taking item i
14            // but still possible with other items (<i)
15            p[i][s] = {i-1,s};
16            dp[i][s] = 1;
17        }
18    }
19 }
20
21 if (!dp[n][target]) {
22     cout << -1 << endl;
23     return;
24 }
25
26 vi subset;
27 ii pos = {n,target};
28 while(pos != ii(0,0)){
29     auto [i,s] = pos;
30     if (p[i][s].second != s) subset.push_back(a[i-1]);
31     pos = p[i][s];
32 }

```

4 Trees

4.1 Sum of distances

Given a tree, $f(u, v) :=$ distance from u to v in the tree, compute

$$\sum_{u,v} f(u, v)$$

. Time: $\mathcal{O}(n)$

```

1 vvi adj;
2 vi sum_going_down, sum_going_up, sz;
3
4 void dfs(int u, int p){
5     for (auto v : adj[u]){
6         if (v == p) continue;
7         dfs(v,u);
8         sz[u] += sz[v];
9         sum_going_down[u] += sum_going_down[v];
10    }
11    sum_going_down[u] += sz[u];
12 }
13
14 void dfs2(int u, int p, int par_ans){
15     int up_amount = sz[0] - sz[u];
16     sum_going_up[u] += par_ans + up_amount;
17     int sum = sum_going_down[u];
18     for (auto v : adj[u]){
19         if (v == p) continue;
20         int par_amount = sz[0] - sz[v];
21         dfs2(v,u, par_ans + par_amount + sum - (
22             sum_going_down[v]+sz[v]));
23     }
24 }
25 void solve(){
26     int n; cin >> n;
27     adj = vvi(n);
28     sum_going_down = sum_going_up = vi(n);
29     sz = vi(n,1);
30
31     for (int i = 1; i < n; i++){
32         int a, b; cin >> a >> b;
33         a--; b--;
34         adj[a].push_back(b);
35         adj[b].push_back(a);
36     }
37
38     dfs(0,0);
39     dfs2(0,0,0);
40
41     for (int i = 0; i < n; i++){
42         cout << sum_going_down[i]+sum_going_up[i] << ','
43     }
44     cout << endl;
45 }
```

4.2 Edge HLD

Sometimes the value is on the edges, for this few things need to change, but here is a template. Pre-computation: $\mathcal{O}(n)$ Queries: $\mathcal{O}(\log^2 n)$

```

1 struct EdgeHLD {
2     int n, timer = 0;
3     vvi adj;
4     vi parent, depth, size, heavy, head, value;
5     vi tin, tout;
6     segtree seg;
7
8     void init(int _n, vvii& _adj) {
9         n = _n;
10        adj = _adj;
11        value.assign(n, 0);
12        parent.assign(n, -1);
13        depth.assign(n, 0);
14        size.assign(n, 0);
15        heavy.assign(n, -1);
16        head.assign(n, 0);
17        tin.assign(n, 0);
18        tout.assign(n, 0);
19        timer = 0;
20
21        // edgeWeight[v] = weight of edge (parent[v], v),
22        // for v>0
23        // root (0) has no parent, so its value is dummy
24        // (0)
25        dfs1(0,0,0);
26        dfs2(0, 0);
27
28        vi linear(n);
29        for (int u = 0; u < n; u++)
30            linear[tin[u]] = value[u]; // position stores
31            edge weight
32
33        seg.init(linear);
34    }
35
36    int dfs1(int u, int p, int w) {
37        size[u] = 1;
38        parent[u] = p;
39        value[u] = w;
40        int max_sz = 0;
41        for (auto[v,w] : adj[u]) {
42            if (v == p) continue;
43            depth[v] = depth[u] + 1;
44            int sz = dfs1(v, u, w);
45            size[u] += sz;
46            if (sz > max_sz) {
47                max_sz = sz;
48                heavy[u] = v;
49            }
50        }
51        return size[u];
52    }
53
54    void dfs2(int u, int h) {
55        tin[u] = timer++;
56        head[u] = h;
57        if (heavy[u] != -1)
58            dfs2(heavy[u], h);
59        for (auto [v,w] : adj[u]) {
60            if (v != parent[u] && v != heavy[u])
61                dfs2(v, v);
62        }
63        tout[u] = timer;
64    }
65
66    // u deve ser o filho
67    void update_edge(int u, int val) {
68        seg.set(tin[u], val);
69    }
70
71    void rangeUpdate(int u, int v, int x) {
72        while (head[u] != head[v]) {
73            if (depth[head[u]] < depth[head[v]]) swap(u, v);
74            seg.rangeUpdate(tin[head[u]], tin[u] + 1, x);
75            u = parent[head[u]];
76        }
77        if (depth[u] > depth[v]) swap(u, v);
78        seg.rangeUpdate(tin[u] + 1, tin[v] + 1, x); // +1
79        to skip LCA's edge
80    }
81
82    void update_subtree(int u, int x) {
83        // updates all edges in subtree of u (skip incoming
84        // edge to u)
85        seg.rangeUpdate(tin[u] + 1, tout[u], x);
86    }
87
88    segtree::node query(int u, int v) {
89        segtree::node res = seg.NEUTRAL;
90        while (head[u] != head[v]) {
91            if (depth[head[u]] < depth[head[v]]) swap(u, v);
92            res = seg.merge(res, seg.query(tin[head[u]], tin[
93                u] + 1));
94            u = parent[head[u]];
95        }
96        if (depth[u] > depth[v]) swap(u, v);
97        res = seg.merge(res, seg.query(tin[u] + 1, tin[v] +
98            1)); // skip LCA's edge
99        return res;
100 }
```

```
96     }
97
98     segtree::node query_subtree(int u) {
99         // query all edges in subtree of u
100        return seg.query(tin[u] + 1, tout[u])
101    }
102};
```

4.3 HLD - Heavy light decomposition

If you need to compute a function on a path in a tree and need to support value updates on nodes, HLD is the way.
 Pre-computation: $\mathcal{O}(n)$ Queries: $\mathcal{O}(\log^2 n)$

OBS: this implementation uses the same segtree as this notebook, with 0-indexing and open-closed interval convention. Ideally, just change the segtree to change the computed function, the HLD struct remains the same. OBS2: this template also supports mass updates (path/subtree) and subtree queries.

```

44 }
45 }
46 return size[u];
47 }

48 void dfs2(int u, int h) {
49     tin[u] = timer++;
50     head[u] = h;
51     if (heavy[u] != -1)
52         dfs2(heavy[u], h);
53     for (int v : adj[u]) {
54         if (v != parent[u] && v != heavy[u])
55             dfs2(v, v);
56     }
57     tout[u] = timer;
58 }
59

60 void update(int u, int val) {
61     seg.set(tin[u], val);
62 }
63

64 void rangeUpdate(int u, int v, int x) {
65     while (head[u] != head[v]) {
66         if (depth[head[u]] < depth[head[v]]) swap(u, v);
67         seg.rangeUpdate(tin[head[u]], tin[u] + 1, x);
68         u = parent[head[u]];
69     }
70     if (depth[u] > depth[v]) swap(u, v);
71     seg.rangeUpdate(tin[u], tin[v] + 1, x);
72 }
73

74 void update_subtree(int u, int x) {
75     seg.rangeUpdate(tin[u], tout[u], x);
76 }
77

78 segtree::node query(int u, int v) {
79     segtree::node res = seg.NEUTRAL;
80     while (head[u] != head[v]) {
81         if (depth[head[u]] < depth[head[v]])
82             swap(u, v);
83         res = seg.merge(res, seg.query(tin[head[u]], ti
84         u]+1));
85         u = parent[head[u]];
86     }
87     if (depth[u] > depth[v]) swap(u, v);
88     res = seg.merge(res, seg.query(tin[u], tin[v]+1));
89     return res;
90 }

91 segtree::node query_subtree(int u){
92     return seg.query(tin[u], tout[u]);
93 }
94
95 };
```

4.4 LCA - RMQ

Pre-computation: $\mathcal{O}(n \log n)$ Queries: $\mathcal{O}(1)$ OBS: call first `dfs(root)` and then `buildSparseTable()` before making queries. Also remember to call `eulertournodes.reserve(2 · n)` and `eulertourdepths.reserve(2 · n)` to optimize memory allocation time of `pushback`.

```
1 int n, timer = 0;
2 vi tin, dep, et_nodes, et_depths;
3 vvi ch;
4 vvii sparse_table;
```

```

5
6 void dfs(int u) {
7     et_nodes.push_back(u);
8     et_depths.push_back(dep[u]);
9     tin[u] = timer++;
10
11    for (int v : ch[u]) {
12        dep[v] = dep[u] + 1;
13        dfs(v);
14        et_nodes.push_back(u);
15        et_depths.push_back(dep[u]);
16    }
17
18    timer++;
19}
20
21 void buildSparseTable() {
22     int m = et_depths.size();
23     sparse_table.assign(LOGN, vii(m));
24
25     for (int i = 0; i < m; i++) {
26         sparse_table[0][i] = {et_depths[i], i};
27     }
28
29     for (int i = 1; (1 << i) <= m; i++) {
30         int len = 1<<i;
31         for (int time = 0; time+len <= m; time++) {
32             ii ans1 = sparse_table[i-1][time];
33             ii ans2 = sparse_table[i-1][time + len/2];
34             sparse_table[i][time] = min(ans1, ans2);
35         }
36     }
37 }
38
39 // TODO: change to struct sparse table for RMQ
40
41 int lca(int u, int v) {
42     int tu = tin[u];
43     int tv = tin[v];
44     if (tu > tv) swap(tu, tv);
45
46     int k = __bit_width((tv - tu + 1)) - 1;
47
48     ii ans1 = sparse_table[k][tu];
49     ii ans2 = sparse_table[k][tv - (1 << k) + 1];
50
51     if (ans1.first <= ans2.first) {
52         return et_nodes[ans1.second];
53     }
54     return et_nodes[ans2.second];
55}

```

4.5 LCA - binary lifting

Pre-computation: $\mathcal{O}(n \log n)$ Queries: $\mathcal{O}(\log n)$ OBS: just call `dfs(root)` before starting queries.

```
1 vvi adj, up;
2 vi tin, tout;
3 int timer = 0;
4
5 void dfs(int u, int p){
6     tin[u] = timer++;
7     for (auto v : adj[u]){
8         if (v == p) continue;
9         up[v][0] = u;
10        for (int dist = 1; dist < LOGN; dist++){
11            up[v][dist] = up[up[v][dist-1]][dist-1];
```

```

12         }
13     dfs(v);
14 }
15 tout[u] = timer++;
16 }

17 int isAncestor(int u, int v){
18     return tin[u] <= tin[v] && tout[v] <= tout[u];
19 }
20 }

21 int lca(int u, int v){
22     if (isAncestor(u,v)) return u;
23     if (isAncestor(v,u)) return v;
24     for (int dist = LOGN-1; dist >= 0; dist--){
25         if(!isAncestor(up[u][dist],v)) u = up[u][dist];
26     }
27 }
28 return up[u][0];
29 }

```

5 Problemas clássicos

5.1 2SAT

Struct for solving 2SAT problems that supports many types of boolean expressions. To add a negated literal use u

```

1 // para adicionar negacao usar ~u
2 // Ex: a clausula (a v !b) se traduz para add_or(a, ~b)
3 struct TwoSatSolver {
4     int n;
5     vvi adj, adjT;
6     vector<bool> vis, assignment;
7     vi topo, scc;
8
9     void build(int _n){
10        n = 2*_n;
11        adj.assign(n, vi());
12        adjT.assign(n, vi());
13    }
14
15    int get(int u){
16        if (u < 0) return 2*(~u)+1;
17        else return 2*u;
18    }
19
20    // u -> v
21    void add_impl(int u, int v){
22        u = get(u), v = get(v);
23        adj[u].push_back(v);
24        adjT[v].push_back(u);
25        adj[v^1].push_back(u^1);
26        adjT[u^1].push_back(v^1);
27    }
28
29    // u || v
30    void add_or(int u, int v){
31        add_impl(~u, v);
32    }
33
34    // u && v
35    void add_and(int u, int v){
36        add_or(u,u); add_or(v,v);
37    }
38
39    // u ^ v (equiv of x != v)
40    void add_xor(int u, int v){

```

```

41        add_impl(u, ~v);
42        add_impl(~u, v);
43    }
44
45    // u == v
46    void add_equals(int u, int v){
47        add_impl(u, v);
48        add_impl(v, u);
49    }
50
51    void toposort(int u){
52        vis[u] = true;
53        for (int v : adj[u])
54            if (!vis[v]) toposort(v);
55        topo.push_back(u);
56    }
57
58    void dfs(int u, int c){
59        scc[u] = c;
60        for (int v : adjT[u])
61            if (!scc[v]) dfs(v,c);
62    }
63
64    pair<bool, vector<bool>> solve(){
65        topo.clear();
66        vis.assign(n, false);
67
68        for (int i = 0; i < n; i++)
69            if (!vis[i]) toposort(i);
70
71        reverse(topo.begin(), topo.end());
72
73        scc.assign(n, 0);
74        int c = 0;
75        for (int u : topo)
76            if (!scc[u]) dfs(u,++c);
77
78        assignment.assign(n/2, false);
79        for (int i = 0; i < n; i += 2){
80            if (scc[i] == scc[i+1]) return {false, {}};
81            assignment[i/2] = scc[i] > scc[i+1];
82        }
83
84        return {true, assignment};
85    }
86 }

```

5.2 Next Greater Element

One of the classic stack applications. Easy to translate to lower, leq or geq, just change the comparator of the while.

```

1 vi next_greater_elem(n, n);
2
3 stack<ii> st;
4 for (int i = 0; i < n; i++){
5     while (!st.empty() && st.top().first < h[i]){
6         next_greater_elem[st.top().second] = i;
7         st.pop();
8     }
9     st.emplace(h[i], i);
10 }

```

6 Strings

6.1 Hashing

Creation time: $\mathcal{O}(n)$ Access time: $\mathcal{O}(1)$ Space: $\mathcal{O}(n)$

```

1 class Hashing{
2     const int mod0 = 1e9+7;
3     vi pmod0;
4     vull pmod1;
5
6     public:
7     void CalcP(int mn, int n){
8         random_device rd;
9         uniform_int_distribution<int> dist(mn+2, mod0
10             -1);
11         int p = dist(rd);
12         if(p % 2 == 0) p--;
13         pmod0 = vi(n);
14         pmod1 = vull(n);
15         pmod0[0] = pmod1[0] = 1;
16         for(int i = 1; i < n; i++){
17             pmod0[i] = (pmod0[i-1] * p) % mod0;
18             pmod1[i] = (pmod1[i-1] * p);
19         }
20
21     viull DistinctSubstrHashes(string base, int
22         offsetVal){
23         int n = base.size();
24         viull ans;
25         for(int i = 0; i < n; i++){
26             int h0 = 0;
27             ull h1 = 0;
28             for(int j = i; j < n; j++){
29                 h0 = (h0 + (base[j]-offsetVal)*pmod0[j-
30                     i]) % mod0;
31                 h1 = (h1 + (base[j]-offsetVal)*pmod1[j-
32                     i]);
33                 ans.push_back(iull(h0, h1));
34             }
35         }
36         sort(ans.begin(), ans.end());
37         auto last = unique(ans.begin(), ans.end());
38         ans.erase(last, ans.end());
39         return ans;
40     }
41
42     viull WindowHash(string data, int offsetVal, int
43         lenWindow){
44         int n = data.size();
45         int h0 = 0;
46         ull h1 = 0;
47         viull ans;
48         for(int i = 0; i < lenWindow; i++){
49             h0 = (h0 + (data[i]+offsetVal)*pmod0[i]) %
50                 mod0;
51             h1 = (h1 + (data[i]+offsetVal)*pmod1[i]);
52         }
53         ans.push_back(iull((h0*pmod0[n-1])%mod0, h1*
54             pmod1[n-1]));
55         for(int i = lenWindow; i < n; i++){
56             h0 = (h0 - (data[i-lenWindow]+offsetVal)*
57                 pmod0[i-lenWindow]) % mod0;
58             h0 = (h0 + mod0) % mod0;
59             h0 = (h0 + (data[i]+offsetVal)*pmod0[i]) %
60                 mod0;
61             h1 = (h1 - (data[i-lenWindow]+offsetVal)*
62                 pmod1[i-lenWindow]));
63         }
64     }
65 }

```

```

54         h1 = (h1 + (data[i]+offsetVal)*pmod1[i]);    58     return ans;
55         ans.push_back(iull((h0*pmod0[n-1-(i-      59 })
56             lenWindow+1)])%mod0, h1*pmod1[n-1-(i-      59 });
57     }                                              59 );
58 }                                              59 };
59 };                                              59 };



## 6.2 KMP


1 vi compute_lps(const string &pat){
2     int m = pat.length();
3     vi lps(m);
4     int len = 0;
5     for (int i = 1; i < m; i++) {
6         while(len > 0 && pat[i] != pat[len])
7             len = lps[len-1];
8         if (pat[i] == pat[len]) len++;
9         lps[i] = len;
10    }
11    return lps;
12 }
13
14 // find all occurrences
15 vi kmp_search(const string &txt, const string &pat){
16     int n = txt.length();
17     int m = pat.length();
18     if (m == 0) return {};
19     vi lps = compute_lps(pat);
20     vi occurrences;
21     int j = 0;
22     for (int i = 0; i < n; i++) {
23         while (j > 0 && txt[i] != pat[j])
24             j = lps[j-1];
25         if (txt[i] == pat[j]) j++;
26
27         if (j == m) {
28             occurrences.push_back(i-m+1);
29             j = lps[j-1];
30         }
31     }
32     return occurrences;
33 }
34
35 // find all occurrences (simpler version)
36 vi kmp_search(const string &txt, const string &pat){
37     int n = txt.length(), m = pat.length();
38     vi lps = compute_lps(pat + '#' + txt);
39     vi occurrences;
40     for (int i = 0; i < n+m+1; i++) {
41         if (lps[i] == pat.length())
42             occurrences.push_back(i-m*2);
43     }
44     return occurrences;
45 }
46
47 // borda sao os prefixos que tambem sao sufixos
48 vi find_borders(const string &s){
49     vi lps = compute_lps(s);
50     int i = s.length()-1;
51
52     vi ans;
53     while (lps[i] > 0){
54         ans.push_back(lps[i]);
55         i = lps[i]-1;
56     }
57     reverse(ans.begin(), ans.end());

```

6.3 Suffix Array

Time: $\mathcal{O}(n \log n)$ Space: $\mathcal{O}(n)$

```

1 struct SuffixArray {
2     int sz;
3     vi suff_ind, lcp;
4     viii suffs;
5
6     void radix_sort() {
7         if (sz <= 1) return;
8         viii suffs_new(sz);
9         vi cnt(sz + 1, 0); /*rever esse tamanho*/
10
11        for (auto& item : suffs) cnt[item.first.second]++;
12        for (int i = 1; i <= sz; ++i) cnt[i] += cnt[i - 1];
13        for (int i = sz - 1; i >= 0; --i) suffs_new[--cnt[ suffs[i].first.second]] = suffs[i];
14
15        cnt.assign(sz + 1, 0);
16        for (auto& item : suffs_new) cnt[item.first.first] +=;
17        for (int i = 1; i <= sz; ++i) cnt[i] += cnt[i - 1];
18        for (int i = sz - 1; i >= 0; --i) suffs[--cnt[ suffs_new[i].first.first]] = suffs_new[i];
19    }
20
21    void build_lcp(vi& a) {
22        lcp.assign(sz, 0);
23        vi rank(sz);
24        for (int i = 0; i < sz; ++i) rank[suff_ind[i]] = i;
25
26        int h = 0;
27        for (int i = 0; i < sz; ++i) {
28            if (rank[i] == sz - 1) { h = 0; continue; }
29            if (h > 0) h--;
30            int j = suff_ind[rank[i] + 1];
31            while (i + h < sz && j + h < sz && a[i + h] == a[j + h]) h++;
32            lcp[rank[i] + 1] = h;
33        }
34    }
35
36    void build(vi& a) {
37        a.push_back(0);
38        sz = a.size();
39        suffs.resize(sz);
40        suff_ind.resize(sz);
41        vi equiv(sz);
42
43        for (int i = 0; i < sz; ++i) suffs[i] = iii(ii(a[i]
44                ], a[i]), i);
45        radix_sort();
46        for (int i = 1; i < sz; ++i) {
47            auto [c, ci] = suffs[i];
48            auto [p, pi] = suffs[i-1];
49            equiv[ci] = equiv[pi] + (c > p);
50        }
51
52        for (int suflen = 1; suflen < sz; suflen *= 2) {
53            for (int i = 0; i < sz; ++i) {
54                suffs[i] = {{equiv[i], equiv[(i + suflen) % sz
55                                ]}, i};
56            }
57            radix_sort();

```

```

57
58        for (int i = 1; i < sz; ++i) {
59            auto [c, ci] = suffs[i];
60            auto [p, pi] = suffs[i-1];
61            equiv[ci] = equiv[pi] + (c > p);
62        }
63
64        for(int i = 0; i < sz; ++i) suff_ind[i] = suffs[i].second;
65        build_lcp(a);
66
67        a.pop_back();
68        sz--;
69        suff_ind.erase(suff_ind.begin());
70        lcp.erase(lcp.begin());
71    }
72 };



## 6.4 Suffix Automaton


1 struct SAM {
2     struct State {
3         int len_link;
4         ll cnt = 0;
5         int first_occ=-1;
6         map<char,int> next;
7     };
8
9     vector<State> st;
10    int last;
11
12    SAM(string s){
13        st.push_back({0,-1,0,-1});
14        last = 0;
15        for (int i = 0; i < s.length(); i++) {
16            extend(s[i], i);
17        }
18        calc_cnt();
19    }
20
21    void extend(char c, int id){
22        int cur = st.size();
23        st.push_back({st[last].len+1,0,1,id});
24        int p = last;
25        while(p!= -1 && st[p].next[c]==0){
26            st[p].next[c] = cur;
27            p = st[p].link;
28        }
29        if (p == -1){
30            st[cur].link = 0;
31            last = cur;
32            return;
33        }
34
35        int q = st[p].next[c];
36        if (st[p].len+1 == st[q].len) {
37            st[cur].link = q;
38            last = cur;
39            return;
40        }
41
42        int clone = st.size();
43        st.push_back({
44            st[p].len+1,
45            st[q].link,
46            0,
47            st[q].first_occ,
48            st[q].next
49        });
50    }

```

```

49     while (p != -1 && st[p].next[c] == q){
50         st[p].next[c] = clone;
51         p = st[p].link;
52     }
53     st[q].link = st[cur].link = clone;
54     last = cur;
55 }
56
57 void calc_cnt(){
58     vi nodes(st.size());
59     iota(nodes.begin(),nodes.end(),0);
60     sort(nodes.begin(),nodes.end(),[&](int a, int b){
61         return st[a].len > st[b].len;
62     });
63
64     for (int u : nodes){
65         if (st[u].link != -1){
66             st[st[u].link].cnt += st[u].cnt;
67         }
68     }
69 }
70
71 int count_occurrences(string t){
72     int cur = 0;
73     for (char c : t){
74         if (st[cur].next.count(c) == 0) return 0;
75         cur = st[cur].next[c];
76     }
77     return st[cur].cnt;
78 }
79
80 int first_occurrence(string t){
81     int cur = 0;
82     for (char c : t){
83         if (!st[cur].next.count(c)) return -2;
84         cur = st[cur].next[c];
85     }
86     return st[cur].first_occ-t.length()+1;
87 }
88
89 int distinct_substrings(){
90     int ans = 0;
91     for (int i = 1; i < st.size(); i++){
92         ans += st[i].len - st[st[i].link].len;
93     }
94     return ans;
95 }
96
97 vi distinct_substrings_perlen(int n){
98     vi diff(n+2);
99     for (int i = 1; i < st.size(); i++){
100        int l = st[st[i].link].len+1;
101        int r = st[i].len;
102        diff[l]++;
103        diff[r+1]--;
104    }
105    vi ans(n+1);
106    ans[0] = diff[0];
107    for (int i = 1; i <= n; i++){
108        ans[i] = ans[i-1]+diff[i];
109    }
110
111    return ans;
112 }
113
114 vi dp;
115 void calc_paths(int u){
116     if (dp[u] != -1) return;
117     dp[u]=1;
118     for (auto [c,v] : st[u].next){
119         dp[v] += dp[u];
120     }
121 }
122
123 string find_kth(int k){
124     dp.assign(st.size(), -1);
125     calc_paths(0);
126     int u = 0;
127     string ans = "";
128     while(k>0){
129         for (auto [c,v] : st[u].next){
130             bool ok = false;
131             if (k <= dp[v]){
132                 ans += c;
133                 u = v;
134                 k--;
135                 ok = true;
136                 break;
137             }
138         }
139     }
140     if (!ok) k-=dp[u];
141     return ans;
142 }
143
144 void calc_paths_with_repetitions(int u){
145     if (dp[u] != -1) return;
146     dp[u]=st[u].cnt;
147     for (auto [c,v] : st[u].next){
148         calc_paths_with_repetitions(v);
149         dp[u] += dp[v];
150     }
151 }
152
153 string find_kth_with_repetitions(int k){
154     dp.assign(st.size(), -1);
155     calc_paths_with_repetitions(0);
156     int u = 0;
157     string ans = "";
158     while(k>0){
159         for (auto [c,v] : st[u].next){
160             bool ok = false;
161             if (k <= dp[v]){
162                 ans += c;
163                 k-=st[v].cnt;
164                 u = v;
165                 ok = true;
166                 break;
167             }
168         }
169     }
170     if (!ok) k-=dp[u];
171 }
172 }

6.5 Z

z[i] := max(k)|s[0..k-1] = s[i..i+k-1]

Time:  $\mathcal{O}(n+m)$  Space:  $\mathcal{O}(n+m)$ 

1 vi compute_z(const string &s) {
2     int n = s.length();
3     vi z(n);
4     int l = 0, r = 0;
5
6     for (int i = 1; i < n; i++) {
7         if (i <= r)
8             z[i] = min(r-i+1, z[i]);
9         else
10            z[i] = 0;
11
12         while (i+l < n && s[i+l] == s[i+r])
13             l++;
14         r = i + l - 1;
15         z[i] = r - i + 1;
16     }
17 }
```

```

8     z[i] = min(r - i + 1, z[i-1]);
9
10    while (i + z[i] < n && s[z[i]] == s[i + z[i]]) {
11        z[i]++;
12        if (i + z[i] - 1 > r) {
13            l = i;
14            r = i + z[i] - 1;
15        }
16    }
17
18    return z;
19}
20
21 vi find_occurrences(const string &txt, const string &
22 pat){
23 vi occurences;
24 vi z = compute_z(pat + '#' + txt);
25 int n = txt.length(), m = pat.length();
26 for (int i = 0; i < n+m+1; i++) {
27     if (z[i] == m) occurences.push_back(i-m-1);
28 }
29 return occurences;
}

```

7 Math

7.1 Combinatorics (Pascal's Triangle)

Computes " n choose k ". Requires factorials to be pre-computed. Time: $\mathcal{O}(\log ZAP)$

7.1.1 Combinatorial Analysis

Fundamental Counting Principles

- **Permutations:** The number of ways to arrange k items from a set of n distinct items.

$$P(n, k) = \frac{n!}{(n - k)!}$$

- **Combinations (Binomial Coefficient):** The number of ways to choose k items from a set of n distinct items, regardless of order.

$$\binom{n}{k} = C(n, k) = \frac{n!}{k!(n-k)!}$$

- **Combinations with Repetition (Stars and Bars):** The number of ways to choose k items of n types, allowing repetitions. Equivalently, the number of ways to distribute k identical balls into n distinct urns.

$$\binom{k+n-1}{n-1} = \binom{k+n-1}{k}$$

6.5 Z

$$z[i] := \max(k) | s[0..k-1] = s[i..i+k-1]$$

Time: $\mathcal{O}(n + m)$ Space: $\mathcal{O}(n + m)$

```
1 vi compute_z(const string &s) {
2     int n = s.length();
3     vi z(n);
4     int l = 0, r = 0;
5
6     for (int i = 1; i < n; i++) {
7         if (i <= r)
```

Binomial Coefficient Properties and Pascal's Triangle

- Pascal's Triangle

$$[n=0 : \binom{0}{0} n=1 : \binom{1}{0} \quad \binom{1}{1} n=2 : \binom{2}{0} \quad \binom{2}{1} \quad \binom{2}{2} n=3 : \binom{3}{0} \quad \binom{3}{1} \quad \binom{3}{2} \quad \binom{3}{3} n=4 : \binom{4}{0} \quad \binom{4}{1} \quad \binom{4}{2} \quad \binom{4}{3} \quad \binom{4}{4}]$$

Stifel's Relation: Each element in Pascal's Triangle is the sum of the two elements immediately above it.

$$\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$$

Symmetry: Elements of a row are symmetric with respect to the center. Choosing k elements is the same as choosing the $n-k$ elements to be left behind.

$$\binom{n}{k} = \binom{n}{n-k}$$

Row Sum: The sum of all elements in row n of Pascal's Triangle (where the first row is $n=0$) is equal to 2^n .

$$\sum_{k=0}^n \binom{n}{k} = 2^n$$

Hockey Stick Identity: The sum of elements in a diagonal, starting at

$$\binom{r}{r}$$

and ending at

$$\binom{n}{r}$$

, is equal to the element in the next row and next column,

$$\binom{n+1}{r+1}$$

$$\sum_{i=r}^n \binom{i}{r} = \binom{n+1}{r+1}$$

- Binomial Theorem:

$$(x+y)^n = \sum_{k=0}^n \binom{n}{k} x^{n-k} y^k$$

- Vandermonde's Identity:

$$\sum_{j=0}^k \binom{m}{j} \binom{n}{k-j} = \binom{m+n}{k}$$

The easiest way to understand the identity is through a counting problem. Imagine you have a committee with m men and n women. How many ways can you form a subcommittee of k people?

Way 1 Direct Counting

You have a total of $m+n$ people and need to choose k of them. The number of ways to do this is simply:

$$\binom{m+n}{k}$$

Way 2 Counting by Cases

We can divide the problem into cases, based on how many men (j) are chosen for the subcommittee.

Case 0: Choose 0 men and k women. The number of ways is

$$\binom{m}{0} \binom{n}{k}$$

Case 1: Choose 1 man and $k-1$ women. The number of ways is

$$\binom{m}{1} \binom{n}{k-1}$$

Case j : Choose j men and $k-j$ women. The number of ways is

$$\binom{m}{j} \binom{n}{k-j}$$

Other Important Concepts

- Catalan Numbers:** A sequence of natural numbers that occurs in various counting problems (e.g., number of binary trees, balanced parenthesis expressions). $C_n = \frac{1}{n+1} \binom{2n}{n} = \frac{\binom{5}{0}}{\binom{5}{1}} \binom{5}{2} = \frac{\binom{5}{0}}{\binom{5}{1}} \binom{5}{2} = \binom{5}{2}$

A commonly used combinatorial proof for the Catalan numbers involves counting the number of lattice (grid) paths from $(0,0)$ to (n,n) that do not cross above the diagonal $y=x$. Each such path consists of n rightward steps and n upward steps, and the Catalan number counts the number of these "Dyck paths" that never go above the diagonal.

- Stirling Numbers of the Second Kind:** The number of ways to partition a set of n labeled objects into k non-empty unlabeled subsets. Denoted by $S(n, k)$ or

$$\{n \ k\}$$

$$S(n, k) = \frac{1}{k!} \sum_{j=0}^k (-1)^{k-j} \binom{k}{j} j^n$$

The Stirling numbers of the second kind can also be computed recursively:

$$S(n, k) = k \cdot S(n-1, k) + S(n-1, k-1)$$

with the boundary conditions:

$$S(0, 0) = 1; \quad S(n, 0) = 0 \text{ for } n > 0; \quad S(0, k) = 0 \text{ for } k > 0$$

- Bell Number:** The Bell number B^n counts the total number of ways to partition a set of n labeled elements into any number (from 1 up to n) of non-empty, unlabeled subsets. It can also be written as a recurrence relation

$$B^n = \sum_{k=0}^n S(n, k)$$

- Pigeonhole Principle:** If n items are put into m boxes, with $n > m$, then at least one box must contain more than one item.

```

1 // n escolhe k
2 // linha n, coluna k no triangulo (indexadas em 0)
3 int pascal(int n, int k){
4     int num = fat[n];
5     int den = (fat[k]*fat[n-k])%ZAP;
6     return (num*expbin(den, ZAP-2))%ZAP;
7 }

```

7.2 Convolutions

7.2.1 AND convolution

$$c[k] = \sum_{i \& j=k} a[i] \cdot b[j]$$

```

1 vector<mint<MOD>> and_conv(vector<mint<MOD>> a, vector<
2     mint<MOD>> b){
3     int n = a.size(); // must be pow of 2
4     for (int j = 1; j < n; j <= 1) {
5         for (int i = 0; i < n; i++) {
6             if (i&j) {
7                 a[i^j] += a[i];
8                 b[i^j] += b[i];
9             }
10        }
11    for (int i = 0; i < n; i++) a[i] *= b[i];
12    for (int j = 1; j < n; j <= 1) {
13        for (int i = 0; i < n; i++) {
14            if (i&j) a[i^j] -= a[i];
15        }
16    }
17    return a;
21 }

```

7.2.2 GCD convolution

$$c[k] = \sum_{\gcd(i,j)=k} a[i] \cdot b[j]$$

```

1 vector<mint<MOD>> gcd_conv(vector<mint<MOD>> a, vector<
2     mint<MOD>> b){
3     int n = (int)max(a.size(), b.size());
4     a.resize(n);
5     b.resize(n);
6     vector<mint<MOD>> c(n);
7     for (int i = 1; i < n; i++) {
8         mint<MOD> x = 0;
9         mint<MOD> y = 0;
10        for (int j = i; j < n; j += i) {
11            x += a[j];
12            y += b[j];
13        }
14        c[i] = x*y;
15    }
16    for (int i = n-1; i >= 1; i--)
17        for (int j = 2 * i; j < n; j += i)
18            c[i] -= c[j];
19    return c;
20 }

```

7.2.3 LCM convolution

$$c[k] = \sum_{\text{lcm}(i,j)=k} a[i] \cdot b[j]$$

```

1 vector<mint<MOD>> lcm_conv(vector<mint<MOD>> a, vector<
2     mint<MOD>> b){
3     int n = (int)max(a.size(), b.size());
4     a.resize(n);
5     b.resize(n);
6     vector<mint<MOD>> c(n), x(n), y(n);
7     for (int i = 1; i < n; i++) {
8         for (int j = i; j < n; j += i) {
9             x[j] += a[i];
10            y[j] += b[i];
11        }
12        c[i] = x[i]*y[i];
13    }
14    for (int i = 1; i < n; i++) {
15        for (int j = 2 * i; j < n; j += i)
16            c[j] -= c[i];
17    }
18    return c;
19 }

```

```

6                         auto u = a[j];
7                         auto v = a[j+step];
8                         a[j] = u+v;
9                         a[j+step] = u-v;
10                    }
11                }
12            }
13        }
14    }
15    if (inv) for (auto &x : a) x /= n;
16
16 vector<mint<MOD>> xor_conv(vector<mint<MOD>> a, vector<
17     mint<MOD>> b){
18     int n = a.size();
19     fwht(a, 0), fwht(b, 0);
20     for (int i = 0; i < n; i++) a[i] *= b[i];
21     fwht(a, 1);
22    return a;
22 }

```

7.3 Extended Euclid

Time: $\mathcal{O}(\log n)$.

```

1 int extended_gcd(int a, int b, int &x, int &y) {
2     x = 1, y = 0;
3     int x1 = 0, y1 = 1;
4     while (b) {
5         int q = a / b;
6         tie(x, x1) = make_tuple(x1, x - q * x1);
7         tie(y, y1) = make_tuple(y1, y - q * y1);
8         tie(a, b) = make_tuple(b, a - q * b);
9     }
10    return a;
11 }

```

7.4 Factorization

Time: $\mathcal{O}(\sqrt{n})$

```

1 // OBS: tem outras variantes mais rápidas no caderno da
2 // UDESC
3 // O(sqrt(n)) fatores repetidos
4 vi fatora(int n) {
5     vi factors;
6     for (int x = 2; x * x <= n; x++) {
7         while (n % x == 0) {
8             factors.push_back(x);
9             n /= x;
10        }
11    }
12    if (n > 1) factors.push_back(n);
13    return factors;
14 }
15
16 // O(sqrt(n))
17 // Calcula a quantidade de divisores de um numero n.
18 int qtdDivisores(int n) {
19     int ans = 1;
20     for (int i = 2; i * i <= n; i += 2) {
21         int exp = 0;
22         while (n % i == 0) {
23             n /= i; exp++;
24         }
25         if (exp > 0) ans *= (exp + 1);
26         if (i == 2) i--;
26 }

```

7.2.4 OR convolution

$$c[k] = \sum_{i|j=k} a[i] \cdot b[j]$$

```

1 vector<mint<MOD>> or_conv(vector<mint<MOD>> a, vector<
2     mint<MOD>> b){
3     int n = a.size(); // must be pow of 2
4     for (int j = 1; j < n; j <= 1) {
5         for (int i = 0; i < n; i++) {
6             if (i&j) {
7                 a[i] += a[i^j];
8                 b[i] += b[i^j];
9             }
10        }
11    for (int i = 0; i < n; i++) a[i] *= b[i];
12    for (int j = 1; j < n; j <= 1) {
13        for (int i = 0; i < n; i++) {
14            if (i&j) a[i] -= a[i^j];
15        }
16    }
17    return a;
21 }

```

7.2.5 XOR convolution

$$c[k] = \sum_{i \oplus j=k} a[i] \cdot b[j]$$

```

1 void fwht(vector<mint<MOD>> &a, bool inv){
2     int n = a.size(); // must be pow of 2
3     for (int step = 1; step < n; step <= 1) {
4         for (int i = 0; i < n; i += 2*step) {
5             for (int j = i; j < i+step; j++)
5 }

```

```

27     }
28     if (n > 1) ans *= 2;
29     return ans;
30 }
31
32 // O(sqrt(n))
33 // Calcula a soma de todos os divisores de um numero n.
34 ll somaDivisores(int n) {
35     ll ans = 1;
36     for (int i = 2; i * i <= n; i += 2) {
37         if (n % i == 0) {
38             int exp = 0;
39             while (n % i == 0) {
40                 n /= i; exp++;
41             }
42
43             ll aux = expbin(i, exp + 1);
44             ans *= ((aux - 1) / (i - 1));
45         }
46         if (i == 2) i--;
47     }
48
49     if (n > 1) ans *= (n + 1);
50     return ans;
51 }
```

7.5 FFT - Fast Fourier Transform

Divide and conquer algorithm used for convolutions and polynomial multiplication. Vector size a is a power of 2.
Time: $\mathcal{O}(n \log n)$ Space: $\mathcal{O}(n)$

```

1 void fft(vector<cd> &a, bool invert){
2     int len = a.size();
3     for(int i = 1, j = 0; i < len; i++){
4         int bit = len >> 1;
5         while(bit & j){
6             j ^= bit;
7             bit >>= 1;
8         }
9         j ^= bit;
10        if(i < j) swap(a[i], a[j]);
11    }
12    for(int l = 2; l <= len; l <<= 1){
13        double ang = 2*PI/l * (invert ? -1: 1);
14        cd wd(cos(ang), sin(ang));
15        for(int i = 0; i < len; i += l){
16            cd w(1);
17            for(int j = 0; j < l/2; j++){
18                cd u = a[i+j], v = a[i+j+1/2];
19                a[i+j] = u+w*v;
20                a[i+j+1/2] = u-w*v;
21                w *= wd;
22            }
23        }
24        if(invert){
25            for(int i = 0; i < len; i++)
26                a[i] /= len;
27        }
28    }
29 }
```

7.6 Inclusion-Exclusion Principle

TODO: rewrite math statement

```

1 // Exemplo:
2 // Contar numeros de 1 a n divisiveis por uma lista de
3 // primos.
4 int n;
5 vi primes;
6 int factors = primes.size();
7 int total_divisible = 0;
8
9 // Itera pelas bitmaskas nao vazias de 'primes',
10 for (int i = 1; i < (1 << factors); i++) {
11     int current_lcm = 1;
12     int subset_size = 0;
13
14     // calcula lcm do subconjunto
15     for (int j = 0; j < factors; j++) {
16         if (i & (1<<j)) {
17             subset_size++;
18             current_lcm = lcm(current_lcm, primes[j]);
19             if (current_lcm > n) break;
20         }
21     }
22
23     if (current_lcm > n) {
24         continue;
25     }
26
27     int count = n / current_lcm;
28
29     // Aplica o Principio da Inclusao-Exclusao:
30     // Se o tamanho do subconjunto eh impar, adiciona.
31     // Se o tamanho do subconjunto eh par, subtrai.
32     if (subset_size & 1) {
33         total_divisible += count;
34     } else {
35         total_divisible -= count;
36     }
37 }
```

7.7 Mint

```

1 template<ll MOD>
2 struct mint {
3     ll val;
4     mint(ll v = 0) {
5         if (v < 0) v = v % MOD + MOD;
6         if (v >= MOD) v %= MOD;
7         val = v;
8     }
9     mint& operator+=(const mint& other) {
10         val += other.val;
11         if (val >= MOD) val -= MOD;
12         return *this;
13     }
14     mint& operator-=(const mint& other) {
15         val -= other.val;
16         if (val < 0) val += MOD;
17         return *this;
18     }
19     mint& operator*=(const mint& other) {
20         val = (val * other.val) % MOD;
21         return *this;
22     }
23     mint& operator/=(const mint& other) {
24         val = (val * inv(other).val) % MOD;
25     }
26 }
```

```

25         return *this;
26     }
27     friend mint operator+(mint a, const mint& b) {
28         return a += b;
29     }
30     friend mint operator-(mint a, const mint& b) {
31         return a -= b;
32     }
33     friend mint operator*(mint a, const mint& b) {
34         return a *= b;
35     }
36     friend mint operator/(mint a, const mint& b) {
37         return a /= b;
38     }
39     static mint power(mint b, ll e) {
40         mint ans = 1;
41         while (e > 0) {
42             if (e & 1) ans *= b;
43             b *= b;
44             e /= 2;
45         }
46         return ans;
47     }
48     static mint inv(mint n) { return power(n, MOD - 2); }
49 }
```

7.8 Modular Inverse

If m is prime, can use binary exponentiation to compute a^{p-2} (Fermat's Little Theorem).

This code works for non-prime m , as long as it is coprime to a .

Time: $\mathcal{O}(\log m)$

```

1 int modInverse(int a, int m) {
2     int x, y;
3     int g = extendedGcd(a, m, x, y);
4     if (g != 1) return -1;
5     return (x % m + m) % m;
6 }
```

7.9 Number Theoretic Transform (NTT)

NTT is a fast algorithm (analogous to FFT) for polynomial multiplication modulo a special prime. It requires a prime modulus $p = c \cdot 2^k + 1$ (a "primitive root prime") and a primitive 2^k -th root of unity modulo p .

- Prime Choices:** To use NTT, pick a modulus and a matching primitive root (see table below). For arbitrary moduli (e.g., $10^9 + 7$), multiply with several NTT-friendly primes and reconstruct with CRT (see `crt_multiply`).
- Time Complexity:** $\mathcal{O}(n \log n)$ for polynomial multiplication.

7.9.1 NTT-Friendly Primes and Roots

NTT-friendly primes and their primitive roots:

- Modulus: 998244353, Primitive Root: 3, Maximum N: 2^{23}
- Modulus: 734003201, Primitive Root: 3, Maximum N: 2^{20}
- Modulus: 167772161, Primitive Root: 3, Maximum N: 2^{25}
- Modulus: 469762049, Primitive Root: 3, Maximum N: 2^{26}

Use the modulus as MOD and the root as ROOT when instantiating the NTT.

- For large/concrete moduli, see crt_multiply in the code for a multi-modulus solution with Chinese Remainder Theorem (CRT).

```

1  template<typename T, ll MOD, ll ROOT>
2  void transform(vector<T>& a, bool invert) {
3      int n = a.size();
4
5      for (int i = 1, j = 0; i < n; i++) {
6          int bit = n >> 1;
7          for (; j & bit; bit >>= 1)
8              j ^= bit;
9          if (i < j) swap(a[i], a[j]);
10     }
11
12     for (int len = 2; len <= n; len <= 1) {
13         T wlen = T::power(ROOT, (MOD - 1) / len);
14         if (invert) wlen = T::inv(wlen);
15         for (int i = 0; i < n; i += len) {
16             T w = 1;
17             for (int j = 0; j < len / 2; j++) {
18                 T u = a[i + j], v = a[i + j + len / 2];
19                 a[i + j] = u + v;
20                 a[i + j + len / 2] = u - v;
21                 w *= wlen;
22             }
23         }
24
25         if (invert) {
26             T n_inv = T::inv(n);
27             for (T& x : a)
28                 x *= n_inv;
29         }
30     }
31
32     template<typename T, ll MOD, ll ROOT>
33     vector<ll> multiply(const vector<ll>& a, const
34     vector<ll>& b) {
35         vector<T> fa(a.begin(), a.end()), fb(b.begin(),
36                     b.end());
37         int n = 1;
38         while (n < a.size() + b.size()) n <= 1;
39         fa.resize(n);
40         fb.resize(n);
41
42         transform<T, MOD, ROOT>(fa, false);
43         transform<T, MOD, ROOT>(fb, false);
44
45         for (int i = 0; i < n; i++) fa[i] *= fb[i];
46
47         transform<T, MOD, ROOT>(fa, true);
48
49         vector<ll> result(n);
50         for (int i = 0; i < n; i++) result[i] = fa[i].val;
51         return result;
52     }
53
54     vector<ll> crt_multiply(const vector<ll>& a, const
55     vector<ll>& b) {
56         const ll mod1 = 998244353;
57         const ll root1 = 3;
58         using mint1 = mint<mod1>;
59         vector<ll> ans1 = NTT::multiply<mint1, mod1,
60                         root1>(a, b);
61
62         const ll mod2 = 1004535809;
63         const ll root2 = 3;
64         using mint2 = mint<mod2>;
65         vector<ll> ans2 = NTT::multiply<mint2, mod2,
66                         root2>(a, b);
67
68         int ans_size = a.size() + b.size() - 2;
69         ll M1_inv_M2 = mint<mod2>::inv(mod1).val;
70
71         vector<ll> final_result(ans_size + 1);
72         for (int i = 0; i <= ans_size; ++i) {
73             ll v1 = ans1[i];
74             ll v2 = ans2[i];
75             ll k = ((v2 - v1 + mod2) % mod2 * M1_inv_M2
76                      ) % mod2;
77             final_result[i] = v1 + k * mod1;
78         }
79     }
80
81     return final_result;
82 }
```

7.10 Euler's Totient

Returns the amount of numbers smaller than n that are coprime to n . Time: $\mathcal{O}(\sqrt{n})$

```

1  int phi(int n){
2      int ans = n;
3      for (int i = 2; i*i <= n; i++) {
4          if (n%i == 0) {
5              while (n%i == 0) n /= i;
6              ans -= ans/i;
7          }
8      }
9      if (n>1) ans -= ans/n;
10     return ans;
11 }
```

8 Geometry

8.1 Convex hull - Graham Scan

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

```

1 #define CLOCKWISE -1
2 #define COUNTERCLOCKWISE 1
3 #define INCLUDE_COLLINEAR 0 // pode mudar
```

```

4  struct Point {
5      ll x, y;
6      bool operator==(Point const& t) const {
7          return x == t.x && y == t.y;
8      }
9  };
10
11 struct Vec {
12     int x, y, z;
13 };
14
15 Vec cross(Vec v1, Vec v2){
16     int x = v1.y*v2.z - v1.z*v2.y;
17     int y = -v1.x*v2.z + v1.z*v2.x;
18     int z = v1.x*v2.y - v1.y*v2.x;
19     return {x,y,z};
20 }
21
22 ll dist2(Point p1, Point p2){
23     int dx = p1.x-p2.x;
24     int dy = p1.y-p2.y;
25     return dx*dx+dy*dy;
26 }
27
28 ll orientation(Point pivot, Point a, Point b){
29     Vec va = {a.x-pivot.x, a.y-pivot.y, 0};
30     Vec vb = {b.x-pivot.x, b.y-pivot.y, 0};
31     Vec v = cross(va,vb);
32     if (v.z < 0) return CLOCKWISE;
33     if (v.z > 0) return COUNTERCLOCKWISE;
34     return 0;
35 }
36
37 bool clock_wise(Point pivot, Point a, Point b) {
38     int o = orientation(pivot, a, b);
39     return o < 0 || (INCLUDE_COLLINEAR && o == 0);
40 }
41
42 bool collinear(Point a, Point b, Point c) { return
43     orientation(a, b, c) == 0; }
44
45 vector<Point> convex_hull(vector<Point> &points, bool
46     counterClockwise) {
47     int n = points.size();
48     Point pivot = *min_element(points.begin(), points.
49     end(), [] (Point a, Point b) {
50         return ii(a.y, a.x) < ii(b.y, b.x);
51     });
52     sort(points.begin(), points.end(), [&] (Point a,
53         Point b) {
54         int o = orientation(pivot, a, b);
55         if (o == 0) return dist2(pivot, a) < dist2(
56             pivot, b);
57         return o == COUNTERCLOCKWISE;
58     });
59
60     if (INCLUDE_COLLINEAR) {
61         int i = n-1;
62         while (i >= 0 && collinear(pivot, points[i],
63             points.back())) i--;
64         reverse(points.begin() + i + 1, points.end());
65     }
66
67     vector<Point> hull;
68     for (auto p : points) {
69         while (hull.size() > 1 && !clock_wise(hull[hull.
70             size() - 2], hull.back(), p))
71             hull.pop_back();
72     }
73 }
```

```

67     hull.push_back(p);
68 }
69 if (!INCLUDE_COLLINEAR && hull.size() == 2 && hull
70 [0] == hull[1])
71     hull.pop_back();
72 if (counterClockwise && hull.size() > 1) {
73     vector<Point> reversed_hull = hull;
74     reverse(reversed_hull.begin() + 1,
75             reversed_hull.end());
76     return reversed_hull;
77 }
78 }
```

8.2 Basic elements - geometry lib

- Basic elements for using the geometry lib, contains points, vector operations and distances between points, distance between point and segment, distance between segments, segment intersection check, orientation check (ccw).
- Always use long double for floating point. Only use floating point if indispensable.
- For $a == b$, use $|a - b| < \text{eps}!!!!$

Time: $\mathcal{O}(1)$

8.2.1 Polygon Area

- Heron's Formula for triangle area:

$$A = \sqrt{s(s-a)(s-b)(s-c)}$$

, where a , b , and c are the triangle sides and $s = (a+b+c)/2$

TODO Shoelace

- Pick's Theorem for polygon area with integer coordinates:

$$A = a + b/2 - 1$$

, where a is the number of integer coordinates inside the polygon and b is the number of integer coordinates on the polygon boundary. b can be calculated for each edge as

$$b = \gcd(x_i + 1 - x_i, y_i + 1 - y_i) + 1$$

Polygon Area Time: $\mathcal{O}(n)$

8.2.2 Point in polygon

Sum of edge angles relative to the point must sum to 2π
Time: $\mathcal{O}(n \log n)$

```

1 #include<bits/stdc++.h>
2 using namespace std;
3 typedef long double ld;
4 #define eps 1e-9
5 #define pi 3.141592653589
6 #define int long long int
7
8 struct pt {
9     int x, y;
10    int operator==(pt b) {
11        return x == b.x && y == b.y;
12    }
13    int operator<(pt b) {
14        if(x == b.x) return y < b.y;
15        return x < b.x;
16    }
17    pt operator-(pt b) {
18        return {x - b.x, y - b.y};
19    }
20    pt operator+(pt b) {
21        return {x+b.x, y + b.y};
22    }
23 };
24 };
25 int cross(pt u, pt v) {
26     return u.x * v.y - u.y * v.x;
27 }
28 int dot(pt u, pt v) {
29     return u.x * v.x + u.y * v.y;
30 }
31 ld norm(pt u) {
32     return sqrt(dot(u, u));
33 }
34 ld dist(pt u, pt v) {
35     return norm(u - v);
36 }
37 int ccw(pt u, pt v) { // cuidado com colineares!!!!
38     return (cross(u, v) > eps)?1:((fabs(cross(u, v)) <
39         eps)?0:-1);
40 }
41 int pointInSegment(pt a, pt u, pt v) { // checks if a
42     lies in uv
43     if(ccw(v - u, a - u)) return 0;
44     vector<pt> pts = {a, u, v};
45     sort(pts.begin(), pts.end());
46     return pts[1] == a;
47 }
48 ld angle(pt u, pt v) { // angle between two vectors
49     ld c = cross(u, v);
50     ld d = dot(u, v);
```

```

51     return atan2l(c, d);
52 }
53 int intersect(pt sa, pt sb, pt ra, pt rb) { // not sure
54     if it works when one of the segments is a point
55     pt s = sb - sa, r = rb - ra;
56     if(pointInSegment(sa, ra, rb) || pointInSegment(sb,
57         ra, rb) || pointInSegment(ra, sa, sb) ||
58         pointInSegment(rb, sa, sb)) return 1;
59     return !(ccw(s, ra - sa) == ccw(s, rb - sa) || ccw(
60         r, sa - ra) == ccw(r, sb - ra));
61 }
62 ld polygonArea(vector<pt>& p) { // not signed (for
63     signed area remove the absolute value at the end)
64     ld area = 0;
65     int n = p.size() - 1; // p[n] = p[0]
66     for(int i = 0; i < n; i++) {
67         area += cross(p[i], p[i + 1]);
68     }
69     return fabs(area)/2;
70 }
71 int pointInPolygon(pt a, vector<pt>& p) { // returns 0
72     for point in BOUNDARY, 1 for point in polygon and
73     -1 for outside
74     ld total = 0;
75     int n = p.size() - 1;
76     for(int i = 0; i < n; i++) {
77         pt u = p[i] - a;
78         pt v = p[i + 1] - a;
79         if(fabs(dist(p[i], a) + dist(p[i + 1], a) -
80             dist(p[i], p[i + 1])) < eps) {
81             return 0;
82         }
83         total += angle(u, v);
84     }
85     return (fabs(fabs(total) - 2 * pi) < eps)?1:-1;
86 }
87 signed main() {
88     int n, m; scanf("%lld %lld", &n, &m);
89     vector<pt> p(n + 1);
90     for(int i = 0; i < n; i++) {
91         scanf("%lld %lld", &p[i].x, &p[i].y);
92     }
93     p[n] = p[0];
94     while(m--) {
95         pt a; scanf("%lld %lld", &a.x, &a.y);
96         int ans = pointInPolygon(a, p);
97         printf("%s\n", (ans > 0)?"INSIDE":(ans?"OUTSIDE":
98                     "BOUNDARY"));
99     }
100 }
```