

# Índice

<b>1. C++ Cheat Sheet</b>	<b>2</b>	8.1. Suffix Array . . . . .	24
<b>2. Data Structures</b>	<b>6</b>	<b>9. Parsers</b>	<b>24</b>
2.1. Fenwick Tree . . . . .	6	9.1. Arithmetic Parser . . . . .	24
2.2. Fenwick Tree 2D . . . . .	6		
2.3. Segment Tree . . . . .	6		
2.4. Segment Tree Lazy . . . . .	7		
2.5. Wavelet Tree . . . . .	8		
2.6. Union-Find . . . . .	9		
<b>3. General Algorithms</b>	<b>9</b>		
3.1. Binary Search . . . . .	9		
3.2. Ternary Search . . . . .	11		
3.3. Brute Force . . . . .	11		
3.3.1. Generate all combinations . . . . .	11		
<b>4. Dynamic Programming</b>	<b>11</b>		
4.1. Knapsack . . . . .	11		
<b>5. Graphs</b>	<b>12</b>		
5.1. Breadth-First Search . . . . .	12		
5.2. Depth-First Search . . . . .	13		
5.3. Dijkstra . . . . .	14		
5.4. Max Flow : Dinic . . . . .	15		
5.5. Max Flow : EdmondsKarp . . . . .	16		
5.6. Minimum Spanning Tree : Kruskal . . . . .	16		
5.7. Lowest Common Ancestor . . . . .	17		
5.8. Level Ancestor . . . . .	18		
5.9. Centroid Decomposition . . . . .	19		
<b>6. Geometry</b>	<b>20</b>		
6.1. Convex Hull . . . . .	20		
6.2. Geometry 2D Utils . . . . .	20		
6.3. Point Inside Polygon . . . . .	21		
6.4. Polygon Area . . . . .	22		
<b>7. Mathematics</b>	<b>22</b>		
7.1. Modular Arithmetics . . . . .	22		
7.2. Modular Fibonacci . . . . .	23		
7.3. Prime Numbers . . . . .	23		
<b>8. Strings</b>	<b>24</b>		

# 1. C++ Cheat Sheet

```

1  /* ===== */
2  // Template //
3  /* ===== */
4  #include <bits/stdc++.h> // add almost everything in one shot
5  #include <tr1/unordered_map>
6  #include <tr1/unordered_set>
7  using namespace std;
8
9  // defines
10 #define rep(i,a,b) for(int i = a; i <= b; ++i)
11 #define invrep(i,b,a) for(int i = b; i >= a; --i)
12 #define umap tr1::unordered_map
13 #define uset tr1::unordered_set
14
15 // typedefs
16 typedef vector<int> vi;
17 typedef vector<vi> vii;
18 typedef long long int ll;
19 typedef pair<int,int> pii;
20
21 int main() {
22     setvbuf(stdout, NULL, _IONBF, 0); //debugging
23     return 0;
24 }
25
26 /* ===== */
27 /* Reading from stdin */
28 /* ===== */
29 scanf("%d",&a); //int
30 scanf("%x",&a); // int in hexadecimal
31 scanf("%llx",&a); // long long in hexadecimal
32 scanf("%lld",&a); // long long int
33 scanf("%c",&c); // char
34 scanf("%s",buffer); // string without whitespaces
35 scanf("%f",&f); // float
36 scanf("%lf",&d); // double
37 scanf("%d %s %d",&a,&b); // * = consume but skip
38
39 // read until EOL
40 // - EOL not included in buffer
41 // - EOL is not consumed
42 // - nothing is written into buffer if EOF is found
43 scanf(" %[^\n]",buffer);
44
45 //reading until EOL or EOF
46 // - EOL not included in buffer
47 // - EOL is consumed
48 // - works with EOF
49 char* output = gets(buffer);
50 if(feof(stdin)) {} // EOF file found
51 if(output == buffer) {} // succesful read

```

```

52 if(output == NULL) {} // EOF found without previous chars found
53 //example
54 while(gets(buffer) != NULL) {
55     puts(buffer);
56     if(feof(stdin)) {
57         break;
58     }
59 }
60
61 // read single char
62 getchar();
63 while(true) {c = getchar(); if (c == EOF || c== '\n') break;}
64
65 /* ===== */
66 /* Printing to stdout */
67 /* ===== */
68 printf("%d",a); // int
69 printf("%ld",a); // long long int
70 printf("%llu",a); // unsigned long long int
71 printf("%c",c); // char
72 printf("%s",buffer); // string until \0
73 printf("%f",f); // float
74 printf("%lf",d); // double
75 printf("%0*.xf",x,y,f); // padding = 0, width = x, decimals = y
76 printf("(%.5s)\n", buffer); // print at most the first five characters (safe to use on
    short strings)
77
78 // print at most first n characters (safe)
79 printf("(%.s)\n", n, buffer); // make sure that n is integer (with long long I had
    problems)
80 //string + \n
81 puts(buffer);
82
83 /* ===== */
84 /* Reading from c string */
85 /* ===== */
86
87 // same as scanf but reading from s
88 int sscanf ( const char * s, const char * format, ...);
89
90 /* ===== */
91 /* Printing to c string */
92 /* ===== */
93 // Same as printf but writing into str, the number of characters is returned
94 // or negative if there is failure
95 int sprintf ( char * str, const char * format, ... );
96 //example:
97 int n=sprintf (buffer, "%d plus %d is %d", a, b, a+b);
98 printf ("%s] is a string %d chars long\n",buffer,n);
99
100 /* ===== */
101 /* Peek last char of stdin */
102 /* ===== */
103 bool peekAndCheck(char c) {

```

```

104 char c2 = getchar();
105 ungetc(c2, stdin); // return char to stdin
106 return c == c2;
107 }
108
109 /* ===== */
110 /* Reading from cin */
111 /* ===== */
112 // reading a line of unknown length
113 string line;
114 getline (cine, name);
115
116 /* ===== */
117 /* CONVERTING FROM STRING TO NUMBERS */
118 /* ===== */
119 //-----
120 // string to int
121 // option #1:
122 int atoi (const char * str);
123 // option #2:
124 sscanf(string, "%i", &i);
125 //-----
126 // string to long int:
127 // option #1:
128 long int strtol (const char* str, char** endptr, int base);
129 // it only works skipping whitespaces, so make sure your numbers
130 // are surrounded by whitespaces only
131 // Example:
132 char szNumbers[] = "2001 60c0c0 -1101110100110100100000 0x6fffff";
133 char * pEnd;
134 long int li1, li2, li3, li4;
135 li1 = strtol (szNumbers, &pEnd, 10);
136 li2 = strtol (pEnd, &pEnd, 16);
137 li3 = strtol (pEnd, &pEnd, 2);
138 li4 = strtol (pEnd, NULL, 0);
139 printf ("The decimal equivalents are: %ld, %ld, %ld and %ld.\n", li1, li2, li3, li4);
140 // option #2:
141 long int atol ( const char * str );
142 // option #3:
143 sscanf(string, "%ld", &l);
144 //-----
145 // string to long long int:
146 // option #1:
147 long long int strtoll (const char* str, char** endptr, int base);
148 // option #2:
149 sscanf(string, "%lld", &l);
150 //-----
151 // string to double:
152 // option #1:
153 double strtod (const char* str, char** endptr); //similar to strtol
154 // option #2:
155 double atof (const char* str);
156 // option #3:
157 sscanf(string, "%lf", &d);

```

```

158 /* ===== */
159 /* C STRING UTILITY FUNCTIONS */
160 /* ===== */
161
162 int strcmp ( const char * str1, const char * str2 ); // (-1,0,1)
163 int memcmp ( const void * ptr1, const void * ptr2, size_t num ); // (-1,0,1)
164 void * memcpy ( void * destination, const void * source, size_t num );
165
166 /* ===== */
167 /* C++ STRING UTILITY FUNCTIONS */
168 /* ===== */
169 // read tokens from string
170 string s = "tok1 tok2 tok3";
171 string tok;
172 stringstream ss(s);
173 while (getline(ss, tok, ' ')) printf("tok = %s\n", tok.c_str());
174
175 // split a string by a single char delimiter
176 void split(const string &s, char delim, vector<string> &elems) {
177     stringstream ss(s);
178     string item;
179     while (getline(ss, item, delim))
180         elems.push_back(item);
181 }
182
183 // find index of string or char within string
184 string str = "random";
185 std::size_t pos = str.find("ra");
186 std::size_t pos = str.find('m');
187 if (pos == string::npos) // not found
188
189 // substrings
190 string subs = str.substr(pos, length);
191 string subs = str.substr(pos); // default: to the end of the string
192
193 // std::string from cstring's substring
194 const char* s = "bla1 bla2";
195 int offset = 5, len = 4;
196 string subs(s + offset, len); // bla2
197
198 // -----
199 // string comparisons
200 int compare (const string& str) const;
201 int compare (size_t pos, size_t len, const string& str) const;
202 int compare (size_t pos, size_t len, const string& str,
203             size_t subpos, size_t sublen) const;
204 int compare (const char* s) const;
205 int compare (size_t pos, size_t len, const char* s) const;
206
207 // examples
208 // 1) check string begins with another string
209 string prefix = "prefix";
210 string word = "prefix suffix";
211 word.compare(0, prefix.size(), prefix);

```

```

212
213 /* ===== */
214 /* OPERATOR OVERLOADING */
215 /* ===== */
216
217 //-----
218 // method #1: inside struct
219 struct Point {
220     int x, y;
221     bool operator<(const Point& p) const {
222         if (x != p.x) return x < p.x;
223         return y < p.y;
224     }
225     bool operator>(const Point& p) const {
226         if (x != p.x) return x > p.x;
227         return y > p.y;
228     }
229     bool operator==(const Point& p) const {
230         return x == p.x && y == p.y;
231     }
232 };
233
234 //-----
235 // method #2: outside struct
236 struct Point {int x, y; };
237 bool operator<(const Point& a, const Point& b) {
238     if (a.x != b.x) return a.x < b.x;
239     return a.y < b.y;
240 }
241 bool operator>(const Point& a, const Point& b) {
242     if (a.x != b.x) return a.x > b.x;
243     return a.y > b.y;
244 }
245 bool operator==(const Point& a, const Point& b) {
246     return a.x == b.x && a.y == b.y;
247 }
248
249 // Note: if you overload the < operator for a custom struct,
250 // then you can use that struct with any library function
251 // or data structure that requires the < operator
252 // Examples:
253 priority_queue<Point> pq;
254 vector<Point> pts;
255 sort(pts.begin(), pts.end());
256 lower_bound(pts.begin(), pts.end(), {1,2});
257 upper_bound(pts.begin(), pts.end(), {1,2});
258 set<Point> pt_set;
259 map<Point, int> pt_map;
260
261
262 /* ===== */
263 /* CUSTOM COMPARISONS */
264 /* ===== */
265 // method #1: operator overloading

```

```

266 // method #2: custom comparison function
267 bool cmp(const Point& a, const Point& b) {
268     if (a.x != b.x) return a.x < b.x;
269     return a.y < b.y;
270 }
271 // method #3: functor
272 struct cmp {
273     bool operator()(const Point& a, const Point& b) {
274         if (a.x != b.x) return a.x < b.x;
275         return a.y < b.y;
276     }
277 };
278 // without operator overloading, you would have to use
279 // an explicit comparison method when using library
280 // functions or data structures that require sorting
281 priority_queue<Point, vector<Point>, cmp> pq;
282 vector<Point> pts;
283 sort(pts.begin(), pts.end(), cmp);
284 lower_bound(pts.begin(), pts.end(), {1,2}, cmp);
285 upper_bound(pts.begin(), pts.end(), {1,2}, cmp);
286 set<Point, cmp> pt_set;
287 map<Point, int, cmp> pt_map;
288
289 /* ===== */
290 /* VECTOR UTILITY FUNCTIONS */
291 /* ===== */
292 std::vector<int> myvector;
293 myvector.push_back(100);
294 myvector.pop_back(); // remove last element
295 myvector.back(); // peek reference to last element
296 myvector.front(); // peek reference to first element
297 myvector.clear(); // remove all elements
298 // sorting a vector
299 vector<int> foo;
300 sort (foo.begin(), foo.end());
301 sort (foo.begin(), foo.end(), std::less<int>()); // increasing
302 sort (foo.begin(), foo.end(), std::greater<int>()); // decreasing
303
304 /* ===== */
305 /* MAP UTILITY FUNCTIONS */
306 /* ===== */
307 struct Point {int x, y; };
308 bool operator<(const Point& a, const Point& b) {
309     return a.x < b.x || (a.x == b.x && a.y < b.y);
310 }
311 map<Point, int> pt2id;
312 // -----
313 // inserting into map
314
315 // method #1: operator[]
316 // it overwrites the value if the key already exists
317 toId[{1, 2}] = 1;
318
319 // method #2: .insert(key, value)

```

```

320 // it returns a pair { iterator(key, value) , bool }
321 // if the key already exists, it doesn't overwrite the value
322 int tid = 0;
323 while (true) {
324     int x,y; scanf("%d %d",&x,&y);
325     auto res = pt2id.insert({x,y}, tid);
326     int id = res.first->second;
327     if (res->second) // insertion happened
328         tid++;
329 }
330 // -----
331 // generating ids with map
332 int get_id(string& name) {
333     static int id = 0;
334     static map<string,int> name2id;
335     auto it = name2id.find(name);
336     if (it == name2id.end())
337         return name2id[name] = id++;
338     return it->second;
339 }
340
341 /* ===== */
342 /* RANDOM INTEGERS */
343 /* ===== */
344 #include <cstdlib>
345 #include <ctime>
346 srand(time(NULL));
347 int x = rand() % 100; // 0-99
348 int randBetween(int a, int b) { // a-b
349     return a + (rand() % (1 + b - a));
350 }
351
352 /* ===== */
353 /* CLIMITS */
354 /* ===== */
355 #include <climits>
356 INT_MIN
357 INT_MAX
358 UINT_MAX
359 LONG_MIN
360 LONG_MAX
361 ULONG_MAX
362 LLONG_MIN
363 LLONG_MAX
364 ULLONG_MAX
365
366 /* ===== */
367 /* Bitwise Tricks */
368 /* ===== */
369
370 // amount of one-bits in number
371 int __builtin_popcount(int x);
372 int __builtin_popcountl(long x);
373 int __builtin_popcountll(long long x);

```

```

374
375 // amount of leading zeros in number
376 int __builtin_clz(int x);
377 int __builtin_clzl(long x);
378 int __builtin_clzll(long long x);
379
380 // binary length of non-negative number
381 int bitlen(int x) { return sizeof(x) * 8 - __builtin_clz(x); }
382 int bitlen(long x) { return sizeof(x) * 8 - __builtin_clzl(x); }
383
384 // index of most significant bit
385 int log2(int x) { return sizeof(x) * 8 - __builtin_clz(x) - 1; }
386 int log2(long x) { return sizeof(x) * 8 - __builtin_clzl(x) - 1; }
387
388 // reverse the bits of an integer
389 int reverse_bits(int x) {
390     int v = 0;
391     while (x) v <<= 1, v |= x&1, x >>= 1;
392     return v;
393 }
394
395 // get string binary representation of an integer
396 string bitstring(int x) {
397     int len = sizeof(x) * 8 - __builtin_clz(x);
398     if (len == 0) return "0";
399
400     char buff[len+1]; buff[len] = '\0';
401     for (int i = len-1; i >= 0; --i, x >>= 1)
402         buff[i] = (char)('0' + (x&1));
403     return string(buff);
404 }
405
406 /* ===== */
407 /* Hexadecimal Tricks */
408 /* ===== */
409
410 // get string hex representation of an integer
411 string to_hex(int num) {
412     static char buff[100];
413     static const char* hexdigits = "0123456789abcdef";
414     buff[99] = '\0';
415     int i = 98;
416     do {
417         buff[i--] = hexdigits[num & 0xf];
418         num >>= 4;
419     } while (num);
420     return string(buff+i+1);
421 }
422
423 // ['0'-'9' 'a'-'f'] -> [0 - 15]
424 int char_to_digit(char c) {
425     if ('0' <= c && c <= '9')
426         return c - '0';
427     return 10 + c - 'a';

```

```

428 }
429
430 /* ===== */
431 /* Other Tricks */
432 /* ===== */
433 // swap stuff
434 int x = 1, y = 2;
435 swap(x, y);
436
437 /* ===== */
438 /* TIPS */
439 /* ===== */
440 // 1) do not use .emplace(x, y) if your struct doesn't have an explicit constructor
441 // instead you can use .push({x, y})
442 // 2) be careful while mixing scanf() with getline(), scanf will not consume \n unless
443 // you explicitly tell it to do so (e.g scanf("%d\n", &x)) )

```

## 2. Data Structures

### 2.1. Fenwick Tree

```

1 #define LSOne(s) (s & (-s))
2 struct FenwickTree {
3     vector<int> ft;
4     FenwickTree(int n) { ft.assign(n+1, 0); }
5     int rsq(int b) {
6         int sum = 0;
7         for (; b; b -= LSOne(b)) sum += ft[b];
8         return sum;
9     }
10    int rsq(int a, int b) {
11        return rsq(b) - (a == 1 ? 0 : rsq(a-1));
12    }
13    void adjust(int k, int v) {
14        for (; k < ft.size(); k += LSOne(k)) ft[k] += v;
15    }
16    void range_adj(int i, int j, int v) {
17        adjust(i, v);
18        adjust(j+1, -v);
19    }
20 };

```

### 2.2. Fenwick Tree 2D

```

1 template<class T> class FenwickTree2D {
2     vector<vector<T> > t;
3     int n, m;
4
5 public:
6     FenwickTree2D() {}
7
8     FenwickTree2D(int n, int m) {

```

```

9         t.assign(n, vector<T>(m, 0));
10        this->n = n; this->m = m;
11    }
12
13    void add(int r, int c, T value) {
14        for (int i = r; i < n; i |= i + 1)
15            for (int j = c; j < m; j |= j + 1)
16                t[i][j] += value;
17    }
18
19    // sum[(0, 0), (r, c)]
20    T sum(int r, int c) {
21        T res = 0;
22        for (int i = r; i >= 0; i = (i & (i + 1)) - 1)
23            for (int j = c; j >= 0; j = (j & (j + 1)) - 1)
24                res += t[i][j];
25        return res;
26    }
27
28    // sum[(r1, c1), (r2, c2)]
29    T sum(int r1, int c1, int r2, int c2) {
30        return sum(r2, c2) - sum(r1 - 1, c2) - sum(r2, c1 - 1) + sum(r1 - 1, c1 - 1);
31    }
32
33    T get(int r, int c) {
34        return sum(r, c, r, c);
35    }
36
37    void set(int r, int c, T value) {
38        add(r, c, -get(r, c) + value);
39    }
40 };

```

### 2.3. Segment Tree

```

1 #include <bits/stdc++.h>
2 using namespace std;
3 typedef vector<int> vi;
4
5 struct SegmentTree {           // the segment tree is stored like a heap array
6     vi st, A;
7     int n;
8     int left (int p) { return p << 1; } // same as binary heap operations
9     int right(int p) { return (p << 1) + 1; }
10
11    void build(int p, int L, int R) { // O(n log n)
12        if (L == R) // as L == R, either one is fine
13            st[p] = L; // store the index
14        else { // recursively compute the values
15            build(left(p), L, (L + R) / 2);
16            build(right(p), (L + R) / 2 + 1, R);
17            int p1 = st[left(p)], p2 = st[right(p)];
18            st[p] = (A[p1] <= A[p2]) ? p1 : p2;
19        }

```

```

20 }
21
22 int rmq(int p, int L, int R, int i, int j) { // O(log n)
23     if (i > R || j < L) return -1; // current segment outside query range
24     if (L >= i && R <= j) return st[p]; // inside query range
25
26     // compute the min position in the left and right part of the interval
27     int p1 = rmq(left(p), L, (L+R) / 2, i, j);
28     int p2 = rmq(right(p), (L+R) / 2 + 1, R, i, j);
29
30     if (p1 == -1) return p2; // if we try to access segment outside query
31     if (p2 == -1) return p1; // same as above
32     return (A[p1] <= A[p2]) ? p1 : p2; // as as in build routine
33
34 int update_point(int p, int L, int R, int idx, int new_value) {
35     // this update code is still preliminary, i == j
36     // must be able to update range in the future!
37     int i = idx, j = idx;
38
39     // if the current interval does not intersect
40     // the update interval, return this st node value!
41     if (i > R || j < L)
42         return st[p];
43
44     // if the current interval is included in the update range,
45     // update that st[node]
46     if (L == i && R == j) {
47         A[i] = new_value; // update the underlying array
48         return st[p] = L; // this index
49     }
50
51     // compute the minimum position in the
52     // left and right part of the interval
53     int p1, p2;
54     p1 = update_point(left(p), L, (L + R) / 2, idx, new_value);
55     p2 = update_point(right(p), (L + R) / 2 + 1, R, idx, new_value);
56
57     // return the position where the overall minimum is
58     return st[p] = (A[p1] <= A[p2]) ? p1 : p2;
59 }
60
61 SegmentTree(const vi &_A) {
62     A = _A; n = (int)A.size(); // copy content for local usage
63     st.assign(4 * n, 0); // create large enough vector of zeroes
64     build(1, 0, n - 1); // recursive build
65 }
66
67 int rmq(int i, int j) { return rmq(1, 0, n - 1, i, j); } // overloading
68
69 int update_point(int idx, int new_value) {
70     return update_point(1, 0, n - 1, idx, new_value); }
71 };
72
73 // usage

```

```

74 int main() {
75     vi A = { 18, 17, 13, 19, 15, 11, 20 };
76     SegmentTree st(A);
77     st.rmq(1,3);
78     st.update_point(5, 100);
79     return 0;
80 }

```

## 2.4. Segment Tree Lazy

```

1  #include <bits/stdc++.h>
2  using namespace std;
3  typedef vector<int> vi;
4
5  struct SegmentTreeLazy {
6      vi arr, tree, lazy;
7      int n;
8      inline int left (int p) { return p << 1; }
9      inline int right(int p) { return (p << 1) + 1; }
10
11     // build the tree
12     void build(int node, int a, int b) {
13         if(a > b) return; // out of range
14         if(a == b) { // leaf node
15             tree[node] = arr[a]; // init value
16             return;
17         }
18         int lnode = left(node), rnode = right(node);
19         build(lnode, a, (a+b)/2); // init left child
20         build(rnode, (a+b)/2 + 1, b); // init right child
21         tree[node] = max(tree[lnode], tree[rnode]); // init root value
22     }
23
24     // increment elements within range [i, j] with value
25     void range_update(int node, int a, int b, int i, int j, int value) {
26         if(lazy[node] != 0) { // this node needs to be updated
27             tree[node] += lazy[node]; // update it
28             if(a != b) {
29                 lazy[left(node)] += lazy[node]; // mark left child as lazy
30                 lazy[right(node)] += lazy[node]; // mark right child as lazy
31             }
32             lazy[node] = 0; // Reset it
33         }
34
35         if(a > b || a > j || b < i) // current segment is not within range [i, j]
36             return;
37
38         if(a >= i && b <= j) { // segment is fully within range
39             tree[node] += value;
40             if(a != b) { // not leaf node
41                 lazy[left(node)] += value;
42                 lazy[right(node)] += value;
43             }
44             return;

```

```

45     }
46
47     range_update(left(node), a, (a+b)/2, i, j, value); // updating left child
48     range_update(right(node), 1+(a+b)/2, b, i, j, value); // updating right child
49     tree[node] = max(tree[left(node)], tree[right(node)]); // Updating root with max
        value
50 }
51
52 // query tree to get max element value within range [i, j]
53 int range_query(int node, int a, int b, int i, int j) {
54     if(a > b || a > j || b < i) return INT_MIN; // out of range
55     if(lazy[node] != 0) { // this node needs to be updated
56         tree[node] += lazy[node]; // update it
57         if(a != b) {
58             lazy[left(node)] += lazy[node]; // mark child as lazy
59             lazy[right(node)] += lazy[node]; // mark child as lazy
60         }
61         lazy[node] = 0; // reset it
62     }
63     if(a >= i && b <= j) // current segment is totally within range [i, j]
64         return tree[node];
65     int q1 = range_query(left(node), a, (a+b)/2, i, j); // Query left child
66     int q2 = range_query(right(node), 1+(a+b)/2, b, i, j); // Query right child
67     return max(q1, q2); // Return final result
68 }
69
70 SegmentTree(const vi& A) {
71     arr = A; n = (int)A.size(); // copy content for local usage
72     tree.assign(4 * n, 0); // create large enough vector of zeroes
73     lazy.assign(4 * n, 0);
74     build(1, 0, n - 1); // recursive build
75 }
76 // overloading
77 int range_update(int i, int j, int value) { return range_update(1, 0, n - 1, i, j,
        value); }
78 int range_query(int i, int j) { return range_query(1, 0, n - 1, i, j); }
79
80 };
81
82 // usage
83 int main() {
84     vi A = { 18, 17, 13, 19, 15, 11, 20 };
85     SegmentTreeLazy stl(A);
86     stl.range_update(1, 5, 100);
87     stl.range_query(1, 3);
88     return 0;
89 }

```

## 2.5. Wavelet Tree

```

1 #include <bits/stdc++.h>
2 using namespace std;
3 typedef vector<int>::iterator iter;
4

```

```

5 struct WaveTree {
6     vector<vector<int>> r0; int n, s;
7     vector<int> arrCopy;
8
9     void build(iter b, iter e, int l, int r, int u) {
10         if (l == r)
11             return;
12         int m = (l+r)/2;
13         r0[u].reserve(e-b+1); r0[u].push_back(0);
14         for (iter it = b; it != e; ++it)
15             r0[u].push_back(r0[u].back() + (*it<=m));
16         iter p = stable_partition(b, e, [=](int i){
17             return i<=m;});
18         build(b, p, l, m, u*2);
19         build(p, e, m+1, r, u*2+1);
20     }
21
22     int q, w;
23     int range(int a, int b, int l, int r, int u) {
24         if (r < q or w < l)
25             return 0;
26         if (q <= l and r <= w)
27             return b-a;
28         int m = (l+r)/2, za = r0[u][a], zb = r0[u][b];
29         return range(za, zb, l, m, u*2) +
30             range(a-za, b-zb, m+1, r, u*2+1);
31     }
32
33     // arr[i] in [0,sigma)
34     WaveTree(vector<int> arr, int sigma) {
35         n = arr.size(); s = sigma;
36         r0.resize(s*2); arrCopy = arr;
37         build(arr.begin(), arr.end(), 0, s-1, 1);
38     }
39
40     // k in [1,n], [a,b] is 0-indexed, -1 if error
41     int quantile(int k, int a, int b) {
42         //extra conditions disabled
43         if (/*a < 0 or b > n or*/ k < 1 or k > b-a)
44             return -1;
45         int l = 0, r = s-1, u = 1, m, za, zb;
46         while (l != r) {
47             m = (l+r)/2;
48             za = r0[u][a]; zb = r0[u][b]; u*=2;
49             if (k <= zb-za)
50                 a = za, b = zb, r = m;
51             else
52                 k -= zb-za, a -= za, b -= zb,
53                 l = m+1, ++u;
54         }
55         return r;
56     }
57
58     // counts numbers in [x,y] in positions [a,b)

```



```

59 int range(int x, int y, int a, int b) {
60     if (y < x or b <= a)
61         return 0;
62     q = x; w = y;
63     return range(a, b, 0, s-1, 1);
64 }
65
66 // count occurrences of x in positions [0,k)
67 int rank(int x, int k) {
68     int l = 0, r = s-1, u = 1, m, z;
69     while (l != r) {
70         m = (l+r)/2;
71         z = r0[u][k]; u*=2;
72         if (x <= m)
73             k = z, r = m;
74         else
75             k -= z, l = m+1, ++u;
76     }
77     return k;
78 }
79
80 // x in [0,sigma)
81 void push_back(int x) {
82     int l = 0, r = s-1, u = 1, m, p; ++n;
83     while (l != r) {
84         m = (l+r)/2;
85         p = (x<=m);
86         r0[u].push_back(r0[u].back() + p);
87         u*=2; if (p) r = m; else l = m+1, ++u;
88     }
89 }
90
91 // doesn't check if empty
92 void pop_back() {
93     int l = 0, r = s-1, u = 1, m, p, k; --n;
94     while (l != r) {
95         m = (l+r)/2; k = r0[u].size();
96         p = r0[u][k-1] - r0[u][k-2];
97         r0[u].pop_back();
98         u*=2; if (p) r = m; else l = m+1, ++u;
99     }
100 }
101
102 //swap arr[i] with arr[i+1], i in [0,n-1)
103 void swap_adj(int i) {
104     int &x = arrCopy[i], &y = arrCopy[i+1];
105     int l = 0, r = s-1, u = 1;
106     while (l != r) {
107         int m = (l+r)/2, p = (x<=m), q = (y<=m);
108         if (p != q) {
109             r0[u][i+1] ^= r0[u][i] ^ r0[u][i+2];
110             break;
111         }
112         u*=2; if (p) r = m; else l = m+1, ++u;

```

```

113     }
114     swap(x, y);
115 }
116 };

```

## 2.6. Union-Find

```

1 #include <bits/stdc++.h>
2 using namespace std;
3 typedef vector<int> vi;
4
5 struct UnionFind {
6     vi p, rank, setSize;
7     int numSets;
8     UnionFind(int n) {
9         numSets = n; setSize.assign(n, 1); rank.assign(n, 0); p.resize(n);
10        rep(i,0,n-1) p[i] = i;
11    }
12    int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
13    bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
14    void unionSet(int i, int j) {
15        if (!isSameSet(i, j)) {
16            numSets--;
17            int x = findSet(i), y = findSet(j);
18            // rank is used to keep the tree short
19            if (rank[x] > rank[y]) {
20                p[y] = x; setSize[x] += setSize[y];
21            } else {
22                p[x] = y; setSize[y] += setSize[x];
23                if (rank[x] == rank[y]) rank[y]++;
24            }
25        }
26    }
27    int numDisjointSets() { return numSets; }
28    int sizeOfSet(int i) { return setSize[findSet(i)]; }
29 };

```

## 3. General Algorithms

### 3.1. Binary Search

```

1 // Find the index of the first item that satisfies a predicate.
2 // If no such index exists, returns -1
3 // Pseudo-code:
4 function binsearch(array, i, j) {
5     while (i < j) {
6         m = (i+j)/2
7         if (predicate(array[m]))
8             j = m
9         else
10            i = m + 1
11    }

```

```

12     return (predicate(array[i]) ? i : -1)
13 }
14
15 // -----
16 // EXAMPLE 1: Integer Lowerbound
17 // predicate(a, i, key) = (a[i] >= key)
18 // i.e. "first element >= key"
19 int lowerbound(vector<int> a, int key, int i, int j) {
20     while (i < j) {
21         int m = (i + j) / 2;
22         if (a[m] >= key)
23             j = m;
24         else
25             i = m + 1;
26     }
27     return a[i] >= key ? i : -1;
28 }
29
30 // -----
31 // EXAMPLE 2: Integer Upperbound
32 // predicate(a, i, key) = (a[i] > key)
33 // i.e. "first element > key"
34 int upperbound(vector<int> a, int key, int i, int j) {
35     while (i < j) {
36         int m = (i + j) / 2;
37         if (a[m] > key)
38             j = m;
39         else
40             i = m + 1;
41     }
42     return a[i] > key ? i : -1;
43 }
44
45 /* ===== */
46 /* upper_bound(), lower_bound() */
47 /* ===== */
48
49 // search between [first, last)
50 // if no value is >= key (lb) / > key (ub), return last
51
52 #include <algorithm>
53 #include <iostream> // std::cout
54 #include <algorithm> // std::lower_bound, std::upper_bound, std::sort
55 #include <vector> // std::vector
56
57 int main () {
58     int myints[] = {10,20,30,30,20,10,10,20};
59     std::vector<int> v(myints,myints+8); // 10 20 30 30 20 10 10 20
60
61     std::sort (v.begin(), v.end()); // 10 10 10 20 20 20 30 30
62
63     std::vector<int>::iterator low,up;
64     low=std::lower_bound (v.begin(), v.end(), 20); //
65     up= std::upper_bound (v.begin(), v.end(), 20); //

```

```

66
67     std::cout << "lower_bound at position " << (low- v.begin()) << '\n';
68     std::cout << "upper_bound at position " << (up - v.begin()) << '\n';
69
70     return 0;
71 }
72
73 // -----
74 // Query: how many items are LESS THAN (<) value x
75
76 lower_bound(v.begin(), v.end(), x) - v.begin();
77
78 // -----
79 // Query: how many items are GREATER THAN (>) value x
80
81 v.end() - upper_bound(v.begin(), v.end(), x);
82
83 //=====
84 // binary_search()
85 //=====
86 bool myfunction (int i,int j) { return (i<j); }
87 int myints[] = {1,2,3,4,5,4,3,2,1};
88 std::vector<int> v(myints,myints+9);
89 bool found = std::binary_search (v.begin(), v.end(), 6, myfunction)
90
91
92 /* ===== */
93 /* Discrete Ternary Search */
94 /* ===== */
95
96 int min_search(int i, int j) {
97     while (i < j) {
98         int m = (i+j)/2;
99         int slope = eval(m+1) - eval(m);
100         if (slope >= 0)
101             j = m;
102         else
103             i = m+1;
104     }
105     return eval(i);
106 }
107
108 int max_search(int i, int j) {
109     while (i < j) {
110         int m = (i+j)/2;
111         int slope = eval(m+1) - eval(m);
112         if (slope <= 0)
113             j = m;
114         else
115             i = m+1;
116     }
117     return eval(i);
118 }

```

### 3.2. Ternary Search

```

1 | int times = 100;
2 | double left = 0.0;
3 | double right = 1000.0;
4 | double ans, m1, m2, v1, v2, third;
5 |
6 | while (times--> 0) {
7 |     third = (right + left) / 3.0;
8 |     m1 = left + third;
9 |     m2 = right - third;
10 |    v1 = eval(m1);
11 |    v2 = eval(m2);
12 |    if (v1 < v2)
13 |        left = m1;
14 |    else if (v2 < v1)
15 |        right = m2;
16 |    else
17 |        left = m1, right = m2;
18 | }
19 |
20 | ans = (v1 + v2) * 0.5;

```

### 3.3. Brute Force

#### 3.3.1. Generate all combinations

```

1 | /* ===== */
2 | /* Try all 2^n combinations */
3 | /* ===== */
4 |
5 | void all_combs(vector<int> items) {
6 |     int n = items.size();
7 |     int times = (1 << n);
8 |     vector<int> comb(n, 0);
9 |
10 |    while(times--> 0) {
11 |
12 |        do_something(comb)
13 |
14 |        // generate next combination
15 |        int i = 0, carry = 1;
16 |        while (i < n) {
17 |            in[i] += carry;
18 |            if (in[i] >= 1)
19 |                carry = 0;
20 |            else
21 |                in[i] = 0;
22 |            // do something with i'th item
23 |            i++;
24 |        }
25 |    }
26 | }

```

## 4. Dynamic Programming

### 4.1. Knapsack

```

1 | /* ===== */
2 | /* Knapsack problem : DP */
3 | /* ===== */
4 |
5 | // -----
6 | // VARIANT 1: without reposition of items
7 | // -----
8 |
9 | // -----
10 | // TOP-DOWN RECURSION (pseudo-code)
11 |
12 | function DP(i, c)
13 |     if i == first
14 |         if c >= weight[i] && value[i] > 0 // enough space and worth it
15 |             return value[i]
16 |         else
17 |             return 0
18 |     else
19 |         ans = DP(i-1, c)
20 |         if c >= weight[i] && value[i] > 0 // enough space and worth it
21 |             ans = max(ans, value[i] + DP(i-1, c - weight[i]))
22 |         return ans
23 |
24 | // -----
25 | // BOTTOM-UP
26 |
27 | #define MAXN 1000 // max num items
28 | #define MAXC 500 // max capacity
29 | int value[MAXN];
30 | int weight[MAXN];
31 | int memo[MAXC+1]; // 0 ... MAXC
32 | int N, C;
33 |
34 | int dp() {
35 |     // first item (i = 0)
36 |     memset(memo, 0, sizeof(memo[0]) * (C+1));
37 |     if (value[0] > 0) { // worth it
38 |         rep (c, weight[0], C) {
39 |             memo[c] = value[0];
40 |         }
41 |     }
42 |     // other items (i = 1 .. N-1)
43 |     rep (i, 1, N-1) {
44 |         if (value[i] > 0) { // worth it
45 |             invrep(c, C, weight[i]) { // <--- REVERSE ORDER !!
46 |                 memo[c] = max(memo[c], value[i] + memo[c - weight[i]]);
47 |             }
48 |         }
49 |     }

```

```

50     return memo[C];
51 }
52
53 // -----
54 // VARIANT 2: with reposition of items
55 // -----
56
57 // -----
58 // TOP-DOWN RECURSION (pseudo-code)
59
60 function DP(i, c)
61     if i == first
62         if c >= weight[i] && value[i] > 0 // enough space and worth it
63             return value[i]
64         else
65             return 0
66     else
67         ans = DP(i-1, c)
68         if c >= weight[i] && value[i] > 0 // enough space and worth it
69             ans = max(ans, value[i] + DP(i, c - weight[i])) // << i instead of i-1
70         return ans
71
72 // -----
73 // BOTTOM-UP
74
75 #define MAXN 1000 // max num items
76 #define MAXC 500 // max capacity
77 int value[MAXN];
78 int weight[MAXN];
79 int memo[2][MAXC + 1]; // 0 .. MAXC
80 int N, C;
81
82 int dp() {
83     // first item (i = 0)
84     memset(memo, 0, sizeof(memo[0]) * (C+1));
85     if (value[0] > 0) { // worth it
86         rep (c, 0, C) {
87             memo[0][c] = value[0] * (c / weight[0]); // collect it as many times as you can
88         }
89     }
90     // other items (i = 1 .. N-1)
91     int prev = 0, curr = 1;
92     rep (i, 1, N-1) {
93         rep(c, 0, C) { // <--- INCREASING ORDER !!
94             if (c >= weight[i] && value[i] > 0) { // if fits in && worth it
95                 memo[curr][c] = max(
96                     memo[prev][c], // option 1: don't take it
97                     value[i] + memo[curr][c - weight[i]] // option 2: take it
98                 );
99             } else {
100                 memo[curr][c] = memo[prev][c]; // only option is to skip it
101             }
102         }
103         // update prev, curr

```

```

104     prev = curr;
105     curr = 1-curr;
106 }
107 return memo[(N-1)&1][C]; // last item + full capacity
108 }

```

## 5. Graphs

### 5.1. Breadth-First Search

```

1  /* ===== */
2  /* BFS (Breadth First Search) */
3  /* ===== */
4
5  #include <queue>
6  #include <stack>
7  #include <vector>
8  #define MAXN 1000
9
10 typedef vector<int> vi;
11 vector<vi> g; // graph
12 vi depth; // bfs depth per node
13 int N; // num of nodes
14
15 void bfs(int s) {
16     queue<int> q; q.push(s);
17     depth.assign(N,-1);
18     depth[s] = 0;
19     while (!q.empty()) {
20         int u = q.front(); q.pop();
21         for (int v : g[u]) {
22             if (depth[v] == -1) {
23                 depth[v] = depth[u] + 1;
24                 q.push(v);
25             }
26         }
27     }
28 }
29
30
31 // =====
32 // Find Tree's Diameter Ends
33 // =====
34
35 #include <cstring>
36 #include <queue>
37 #include <vector>
38 using namespace std;
39
40 int dist[MAXN];
41 vector<vi> g;
42
43 int farthestFrom(int s) {

```

```

44 int farthest = s;
45 int maxd = 0;
46 memset(dist, -1, sizeof(dist[0]) * n);
47 queue<int> q; q.push(s);
48 dist[s] = 0;
49 while (!q.empty()) {
50     int u = q.front(); q.pop();
51     for (int v : g[u]) {
52         if (dist[v] == -1) {
53             dist[v] = dist[u] + 1;
54             q.push(v);
55             if (dist[v] > maxd) {
56                 maxd = dist[v];
57                 farthest = v;
58             }
59         }
60     }
61 }
62 return farthest;
63 }
64
65 void findDiameter(int& e1, int& e2) {
66     e1 = farthestFrom(0);
67     e2 = farthestFrom(e1);
68 }

```

## 5.2. Depth-First Search

```

1  #include <queue>
2  #include <stack>
3  #include <vector>
4  #define MAXN 1000
5  vector<int> adjList[MAXN];
6  bool visited[MAXN];
7
8
9  //iterative
10 void dfs(int root) {
11     stack<int> s;
12     s.push(root);
13     visited[root] = true;
14     while (!s.empty()) {
15         int u = s.top();
16         s.pop();
17         for (int i = 0; i < adjList[u].size(); ++i) {
18             int v = adjList[u][i];
19             if (visited[v])
20                 continue;
21             visited[u] = true;
22             s.push(v);
23         }
24     }
25 }
26

```

```

27 //recursive
28 void dfs(int u) {
29     visited[u] = true;
30     for (int i = 0; i < adjList[i].size(); ++i) {
31         int v = adjList[u][i];
32         if (!visited[v])
33             dfs(v);
34     }
35 }
36
37 //-----
38 // Finding connected components
39 //-----
40 int numCC = 0;
41 memset(visited, false, sizeof visited);
42 for (int i = 0; i < V; i++)
43     if (!visited[i])
44         printf("Component %d:", ++numCC), dfs(i), printf("\n"); // 3 lines here!
45 printf("There are %d connected components\n", numCC);
46
47 //-----
48 // Flood Fill
49 //-----
50
51 //explicit graph
52 #define DFS_WHITE (-1)
53 vector<int> dfs_num(DFS_WHITE, n);
54 void floodfill(int u, int color) {
55     dfs_num[u] = color; // not just a generic DFS_BLACK
56     for (int j = 0; j < (int)AdjList[u].size(); j++) {
57         int v = AdjList[u][j];
58         if (dfs_num[v] == DFS_WHITE)
59             floodfill(v, color);
60     }
61 }
62
63 //implicit graph
64 int dr[] = {1, 1, 0, -1, -1, -1, 0, 1}; // trick to explore an implicit 2D grid
65 int dc[] = {0, 1, 1, 1, 0, -1, -1, -1}; // S, SE, E, NE, N, NW, W, SW neighbors
66 int floodfill(int r, int c, char c1, char c2) { // returns the size of CC
67     if (r < 0 || r >= R || c < 0 || c >= C) return 0; // outside grid
68     if (grid[r][c] != c1) return 0; // does not have color c1
69     int ans = 1; // adds 1 to ans because vertex (r, c) has c1 as its color
70     grid[r][c] = c2; // now recolors vertex (r, c) to c2 to avoid cycling!
71     for (int d = 0; d < 8; d++)
72         ans += floodfill(r + dr[d], c + dc[d], c1, c2);
73     return ans; // the code is neat due to dr[] and dc[]
74 }
75
76
77 //-----
78 // Topo Sort
79 //-----
80

```

```

81 //option 1: tarjan's algorithm
82
83 vector<int> topoSort;
84 void dfs2(int u) {
85     visited[u] = true;
86     for (int j = 0; j < (int)AdjList[u].size(); j++) {
87         int v = AdjList[u][j];
88         if (!visited[v])
89             dfs2(v);
90     }
91     topoSort.push_back(u); //only change with respect to dfs()
92 }
93 //in main
94 topoSort.clear();
95 memset(visited, false, sizeof visited);
96 for (int i = 0; i < V; i++) // this part is the same as finding CCs
97     if (!visited[i])
98         dfs2(i);
99 for (int i = topoSort.size()-1; i >= 0; i--) // we need to print in reverse order
100     printf(" %d", topoSort[i]);
101
102 //option 2: Kahn's algorithm
103
104 //pseudo-code
105 // L <- Empty list that will contain the sorted elements
106 // S <- Set of all nodes with no incoming edges
107 // while S is non-empty do
108 //     remove a node n from S
109 //     add n to tail of L
110 //     for each node m with an edge e from n to m do
111 //         remove edge e from the graph
112 //         if m has no other incoming edges then
113 //             insert m into S
114 // if graph has edges then
115 //     return error (graph has at least one cycle)
116 // else
117 //     return L (a topologically sorted order)
118
119 //c++ version
120 //Input : adj_list ->Adjacency list; indegree : indegrees of all nodes .....
121 void topoSort(vii & adj_list, vi & indegree) {
122
123     queue<int> tsort_queue;
124     vector<int> sorted;
125
126     for(int i = 0; i < (signed)indegree.size(); i++)
127         if(indegree[i] == 0)
128             tsort_queue.push(i);
129
130     while(!tsort_queue.empty()) {
131         int u = tsort_queue.front();
132         tsort_queue.pop();
133         sorted.push_back(u);
134

```

```

135         for(int i = 0; i < adj_list[u].size(); ++i) {
136             int v = adj_list[u][i];
137             if(--indegree[v] == 0)
138                 tsort_queue.push(v);
139         }
140     }
141
142     printf("Top Sorted Order : ");
143     for(int i = 0; i < (signed)sorted.size(); i++)
144         printf(" %d ", sorted[i]);
145     printf("\n");
146 }
147
148 /* ===== */
149 /* Articulation Points & Cut Edges */
150 /* ===== */
151 vi depth(N,-1);
152 vi low(N);
153 vii graph(N,vi());
154 int rootChildren = 0;
155
156 void dfs(int u, int p, int d) { // (node, parent, depth)
157     depth[u] = d;
158     low[u] = d;
159     for(int v : graph[u]) {
160         if (v == p) continue; // direct edge to parent is not back edge
161         if (depth[v] == -1) {
162             if (p == -1 && ++rootChildren > 1) // root
163                 printf("root = %d is articulation point\n", root);
164
165             dfs(v, u, d + 1);
166
167             if (low[v] >= depth[u] && p != -1)
168                 printf("u = %d is articulation point\n", u);
169
170             if (low[v] > depth[u])
171                 printf("(u,v) = (%d, %d) is cut edge\n", u, v);
172
173             if (low[v] < low[u]) low[u] = low[v];
174
175         } else if (depth[v] < low[u]) {
176             low[u] = depth[v];
177         }
178     }
179 }

```

### 5.3. Dijkstra

```

1 // complexity: (|E| + |V|) * log |V|
2 #include <bits/stdc++.h>
3 using namespace std;
4 typedef pair<int, int> pii; // (weight, node), in that order
5
6 vector<vector<pii>> g; // graph

```

```

7 int N; // number of nodes
8 vector<int> mindist; // min distance from source to each node
9 vector<int> parent; // parent of each node in shortest path from source
10
11 void dijkstra(int source) {
12     parent.assign(N, -1);
13     mindist.assign(N, INT_MAX);
14     mindist[source] = 0;
15     priority_queue<pii, vector<pii>, greater<pii>> q;
16     q.push(pii(0, source));
17     while (!q.empty()) {
18         pii p = q.front(); q.pop();
19         int u = p.second, dist = p.first;
20         if (mindist[u] < dist) continue; // skip outdated improvements
21         for (pii& e : g[u]) {
22             int v = e.second, w = e.first;
23             if (mindist[v] > dist + w) {
24                 mindist[v] = dist + w;
25                 parent[v] = u;
26                 q.push(v);
27             }
28         }
29     }
30 }

```

#### 5.4. Max Flow : Dinic

```

1 /* ===== */
2 /* DINIC : Max Flow */
3 /* ===== */
4 // Time Complexity:
5 // - general worst case:  $O(|E| * |V|^2)$ 
6 // - unit capacities:  $O(\min(V^{2/3}, \sqrt{E}))$ 
7 // - Bipartite graph (unit capacities) + source & sink (any capacities):  $O(E \sqrt{V})$ 
8
9 #include <bits/stdc++.h>
10 using namespace std;
11 typedef long long int ll;
12
13 struct Dinic {
14     struct edge {
15         int to, rev;
16         ll f, cap;
17     };
18
19     vector<vector<edge>> g;
20     vector<ll> dist;
21     vector<int> q, work;
22     int n, sink;
23
24     bool bfs(int start, int finish) {
25         dist.assign(n, -1);
26         dist[start] = 0;
27         int head = 0, tail = 0;

```

```

28         q[tail++] = start;
29         while (head < tail) {
30             int u = q[head++];
31             for (const edge &e : g[u]) {
32                 int v = e.to;
33                 if (dist[v] == -1 and e.f < e.cap) {
34                     dist[v] = dist[u] + 1;
35                     q[tail++] = v;
36                 }
37             }
38         }
39         return dist[finish] != -1;
40     }
41
42     ll dfs(int u, ll f) {
43         if (u == sink)
44             return f;
45         for (int &i = work[u]; i < (int)g[u].size(); ++i) {
46             edge &e = g[u][i];
47             int v = e.to;
48             if (e.cap <= e.f or dist[v] != dist[u] + 1)
49                 continue;
50             ll df = dfs(v, min(f, e.cap - e.f));
51             if (df > 0) {
52                 e.f += df;
53                 g[v][e.rev].f -= df;
54                 return df;
55             }
56         }
57         return 0;
58     }
59
60     Dinic(int n) {
61         this->n = n;
62         g.resize(n);
63         dist.resize(n);
64         q.resize(n);
65     }
66
67     void add_edge(int u, int v, ll cap) {
68         edge a = {v, (int)g[v].size(), 0, cap};
69         edge b = {u, (int)g[u].size(), 0, cap};
70         g[u].push_back(a);
71         g[v].push_back(b);
72     }
73
74     ll max_flow(int source, int dest) {
75         sink = dest;
76         ll ans = 0;
77         while (bfs(source, dest)) {
78             work.assign(n, 0);
79             while (ll delta = dfs(source, LLONG_MAX))
80                 ans += delta;
81         }

```

```

82     return ans;
83 }
84 };
85
86 // usage
87 int main() {
88     Dinic din(2);
89     din.add_edge(0,1,10);
90     ll mf = din.max_flow(0,1);
91 }

```

## 5.5. Max Flow : EdmondsKarp

```

1  /* ===== */
2  /* Edmonds Karp */
3  /* ===== */
4  // complexity: |V| * |E|^2
5  #include <bits/stdc++.h>
6  using namespace std;
7  typedef vector<int> vi;
8
9  #define INF 1000000000
10 #define MAX_V 40
11
12 int res[MAX_V][MAX_V]; //residual capacities
13 int mf, f, s, t;
14 vi p;
15 vector<vi> AdjList;
16
17 void augment(int v, int minEdge) { // traverse BFS spanning tree from s to t
18     if (v == s) { f = minEdge; return; } // record minEdge in a global variable f
19     else if (p[v] != -1) { augment(p[v], min(minEdge, res[p[v]][v])); // recursive
20         res[p[v]][v] -= f; res[v][p[v]] += f; } // update
21 }
22
23 int main() {
24     int V, k, vertex, weight;
25     scanf("%d %d %d", &V, &s, &t);
26
27     //initialize AdjList and res
28     AdjList.assign(V, vi());
29     for (int i = 0; i < V; i++) {
30         scanf("%d", &k);
31         for (int j = 0; j < k; j++) {
32             scanf("%d %d", &vertex, &weight);
33             //forward residual capacity
34             AdjList[i].push_back(vertex);
35             res[i][vertex] = weight;
36             //backward residual capacity
37             AdjList[vertex].push_back(i);
38             res[vertex][i] = 0;
39         }
40     }
41 }

```

```

42 mf = 0;
43 while (1) {
44
45     //run BFS to find augmenting path
46     f = 0;
47     bitset<MAX_V> vis; vis[s] = true;
48     queue<int> q; q.push(s);
49     p.assign(MAX_V, -1); //reset parents
50     while (!q.empty()) {
51         int u = q.front(); q.pop();
52         //if (u == t) break; //not necessary, check goto below
53         for (int j = 0; j < (int)AdjList[u].size(); j++) { // we use AdjList here!
54             int v = AdjList[u][j];
55             if (res[u][v] > 0 && !vis[v]) {
56                 vis[v] = true, q.push(v), p[v] = u;
57                 if(v == t) { //target found!!
58                     goto end_bfs;
59                 }
60             }
61         }
62     }
63     end_bfs:
64     augment(t, INF);
65     if (f == 0) break;
66     mf += f;
67 }
68
69 printf("%d\n", mf); // this is the max flow value
70
71 return 0;
72 }

```

## 5.6. Minimum Spanning Tree : Kruskal

```

1  /* ===== */
2  /* KRUSKAL ALGORITHM : Minimum Spanning Tree */
3  /* ===== */
4
5  typedef pair<int,int> pii;
6
7  // edge list
8  vector<pair<int,pii>> edge_list; // (weight, (u, v))
9  // num of nodes
10 int N;
11
12 struct UnionFind {
13     vi p, rank;
14     int numSets;
15     UnionFind(int n) {
16         numSets = n;
17         rank.assign(n,0);
18         p.resize(n);
19         rep(i,0,n-1) p[i] = i;
20     }

```



```

21 int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
22 bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
23 void unionSet(int i, int j) {
24     if (!isSameSet(i, j)) {
25         numSets--;
26         int x = findSet(i), y = findSet(j);
27         if (rank[x] > rank[y]) {
28             p[y] = x;
29         } else {
30             p[x] = y;
31             if (rank[x] == rank[y]) rank[y]++;
32         }
33     }
34 }
35 };
36
37 int mst_cost() {
38     sort(edge_list.begin(), edge_list.end());
39     UnionFind uf(N);
40     int cost = 0;
41     for (auto& edge : edge_list) {
42         int w = edge.first;
43         int u = edge.second.first;
44         int v = edge.second.second;
45         if (!uf.isSameSet(u, v)) {
46             cost += w;
47             uf.unionSet(u, v);
48         }
49     }
50     return cost;
51 }

```

## 5.7. Lowest Common Ancestor

```

1  /* ===== */
2  /* LCA (Lowest Common Ancestor) */
3  /* ===== */
4  #include <bits/stdc++.h>
5  using namespace std;
6  typedef vector<int> vi;
7  #define rep(i,a,b) for (int i=a; i<=b; ++i)
8  #define invrep(i,b,a) for (int i=b; i>=a; --i)
9
10 // -----
11 // METHOD 1: SPARSE TABLE - EULER TOUR
12 // -----
13 // construction:  $O(2|V| \log 2|V|) = O(|V| \log |V|)$ 
14 // query:  $O(1)$ 
15 // cannot be updated :(
16
17
18 #define MAXN 10000
19 #define MAXLOG 14
20

```

```

21 int E[2 * MAXN]; // records sequence of visited nodes
22 int L[2 * MAXN]; // records level of each visited node
23 int H[MAXN]; // records index of first occurrence of node u in E
24 int idx; // tracks node occurrences
25 int rmq[2 * MAXN][MAXLOG + 1];
26
27 int N; // number of nodes
28 vector<vi> g; // tree graph
29
30 // get highest exponent e such that  $2^e \leq x$ 
31 inline int log2(int x) { return sizeof(x) * 8 - __builtin_clz(x) - 1; }
32
33 void dfs(int u, int depth) {
34     H[u] = idx; // index of first u's occurrence
35     E[idx] = u; // record node occurrence
36     L[idx++] = depth; // record depth
37     for (int v : g[u])
38         if (H[v] == -1) {
39             dfs(v, depth + 1); // backtrack
40             E[idx] = u; // new occurrence of u
41             L[idx++] = depth;
42         }
43 }
44
45 void lca_init() {
46     idx = 0;
47     memset(H, -1, sizeof(H[0]) * N);
48     dfs(0, 0); // euler tour to initialize H, E, L
49     int nn = idx; // <-- make sure you use the correct number
50     int m = log2(nn);
51
52     rep(i, 0, nn - 1)
53         rmq[i][0] = i; // base case
54     rep(j, 1, m) {
55         rep(i, 0, nn - (1 << j)) {
56             //  $i \dots i + 2^{(j-1)} - 1$ 
57             int i1 = rmq[i][j-1];
58             //  $i + 2^{(j-1)} \dots i + 2^j - 1$ 
59             int i2 = rmq[i + (1 << (j-1))][j-1];
60             // choose index with minimum level
61             rmq[i][j] = (L[i1] < L[i2] ? i1 : i2);
62         }
63     }
64 }
65
66 int LCA(int u, int v) {
67     // get occurrence indexes in increasing order
68     int l, r;
69     if (H[u] < H[v])
70         l = H[u], r = H[v];
71     else
72         l = H[v], r = H[u];
73     // get node with minimum level within [l .. r] in  $O(1)$ 
74     int len = r - l + 1;

```

```

75     int m = log2(len);
76     int i1 = rmq[1][m];
77     int i2 = rmq[r - ((1 << m) - 1)][m];
78     return L[i1] < L[i2] ? E[i1] : E[i2];
79 }
80
81 inline int dist(int u, int v) {
82     // make sure you use H to retrieve the indexes of
83     // u and v within the Euler Tour sequence before
84     // using L
85     return L[H[u]] + L[H[v]] - 2 * L[H[LCA(u,v)]];
86 }
87
88 // -----
89 // METHOD 2: SPARSE TABLE - JUMP POINTERS
90 // -----
91 // construction: O(|V| log |V|)
92 // query: O(log|V|)
93 // can be updated: tree can receive new nodes :)
94
95 #define MAXN 1000000
96 #define MAXLOG 20
97
98 int P[MAXN][MAXLOG+1]; // level ancestor table
99 int L[MAXN]; // levels
100 int N; // num of nodes
101 vector<vi> g; // tree graph
102 int root; // root of the tree
103
104 // dfs to record direct parents and levels
105 void dfs(int u, int p, int l) {
106     P[u][0] = p;
107     L[u] = l;
108     for (int v : g[u])
109         if (L[v] == -1)
110             dfs(v, u, l + 1);
111 }
112
113 void init() {
114     memset(P, -1, sizeof P);
115     memset(L, -1, sizeof L);
116     dfs(root, -1, 0);
117     rep(j, 1, MAXLOG) {
118         rep(i, 0, N-1) {
119             // i's 2^j th ancestor is
120             // i's 2^(j-1) th ancestor's 2^(j-1) th ancestor
121             int p = P[i][j-1];
122             if (p != -1) P[i][j] = P[p][j-1];
123         }
124     }
125 }
126
127 int log2(int x) {
128     int i = 0;

```

```

129     while (x >>= 1, i++);
130     return i-1;
131 }
132
133 int LCA(int u, int v) {
134     if (level[u] < level[v]) swap(u, v);
135     // raise lowest to same level
136     int diff = level[u] - level[v];
137     while (diff) {
138         int j = log2(diff);
139         u = P[u][j];
140         diff -= (1 << j);
141     }
142     if (u == v) return u; // same node, we are done
143     // raise u and v to their highest ancestors below
144     // the LCA
145     invrep(j, MAXLOG, 0)
146         // if there are 2^j th ancestors for u and v
147         // and they are not the same,
148         // then they can be raised and still be below the LCA
149         if (P[u][j] != -1 && P[v][j] != P[u][j])
150             u = P[u][j], v = P[v][j];
151     // the direct parent of u (or v) is lca(u,v)
152     return P[u][0];
153 }
154
155 int dist(int u, int v) {
156     return L[u] + L[v] - 2 * L[LCA(u,v)];
157 }
158
159 int add_child(int u, int v) {
160     // add to graph
161     g[u].push_back(v);
162     // update level
163     L[v] = L[u] + 1;
164     // update ancestors
165     P[v][0] = u;
166     rep(j, 1, MAXLOG){
167         P[v][j] = P[P[v][j-1]][j-1];
168         if (P[v][j] == -1) break;
169     }
170 }

```

## 5.8. Level Ancestor

```

1  /* ===== */
2  /* LA (Level Ancestor Problem) */
3  /* ===== */
4  #include <vector>
5  using namespace std;
6  typedef vector<int> vi;
7
8  #define rep(i,a,b) for(int i=a; i<=b; ++i)
9  #define invrep(i,b,a) for(int i=b; i>=a; --i)

```

```

10
11 #define MAXN 10000
12 #define MAXLOG 16
13
14 int P[MAXN][MAXLOG + 1]; // level ancestor table
15 int L[MAXN]; // level array
16 vector<vi> g; // tree graph
17 int root; // root of the tree
18
19 // dfs to record direct parents and levels
20 void dfs(int u, int p, int l) {
21     P[u][0] = p;
22     L[u] = l;
23     for (int v : g[u])
24         if (L[v] == -1)
25             dfs(v, u, l + 1);
26 }
27
28 inline int log2(int x) { return sizeof(x) * 8 - __builtin_clz(x) - 1; }
29
30 void init_la(int n) {
31     memset(P, -1, sizeof P);
32     memset(L, -1, sizeof L);
33     dfs(root, -1, 0);
34     // fill sparse table
35     int m = log2(n);
36     rep(j, 1, m) {
37         rep(i, 0, n - 1) {
38             // 2^j th ancestor of i
39             // = 2^(j-1) th ancestor of 2^(j-1) th ancestor of i
40             int p = P[i][j-1];
41             if (p != -1) P[i][j] = P[p][j-1];
42         }
43     }
44 }
45
46 int lev_anc(int u, int k) {
47     if (k == 0) return u; // trivial case
48     if (L[u] < k) return -1; // check ancestor exists
49     invrep(j, log2(k), 0) {
50         if (k >= (1 << j)) {
51             u = P[u][j]; // u = 2^j th ancestor of u
52             k -= (1 << j); // only k - 2^j steps left
53             if (k == 0) break; // target reached
54         }
55     }
56     return u;
57 }

```

## 5.9. Centroid Decomposition

```

1 /* ===== */
2 /* Centroid Decomposition */
3 /* ===== */

```

```

4
5 // construction: O(n log n)
6 // query: O(log n)
7
8 #include <vector>
9 #include <queue>
10 #include <cstring>
11 using namespace std;
12
13 #define MAXN 100000
14 typedef vector<int> vi;
15
16 vector<vi> g; // graph
17 vector<vi> cg; // centroid graph
18 int N; // num of nodes
19 bool removed[MAXN]; // nodes removed from tree
20 int desc[MAXN]; // num of descendants
21 int cpar[MAXN]; // centroid parent
22
23 // count descendants
24 int dfs_count(int u, int p) {
25     int count = 1;
26     for (int v : g[u])
27         if (v != p && !removed[v])
28             count += dfs_count(v, u);
29     return desc[u] = count;
30 }
31
32 // recursive search of centroid
33 int dfs_cent(int u, int p, int lim) {
34     for (int v : g[u])
35         if (v != p && !removed[v] && desc[v] > lim)
36             return dfs_cent(v, u, lim);
37     return u;
38 }
39
40 // find centroid of u's subtree
41 int centroid(int u) {
42     dfs_count(u, -1);
43     return dfs_cent(u, -1, desc[u] / 2);
44 }
45
46 // perform centroid decomposition
47 void decomp() {
48     memset(removed, 0, sizeof(removed[0]) * N);
49     cg.assign(N, vi());
50     int c = centroid(0);
51     cpar[c] = -1;
52     removed[c] = true;
53     queue<int> q; q.push(c);
54     while (!q.empty()) {
55         int u = q.front(); q.pop();
56         for (int v : g[u]) {
57             if (!removed[v]) {

```

```

58     c = centroid(v);
59     cpar[c] = u; // set parent of c to u
60     cg[u].push_back(c); // add edge (u -> c)
61     removed[c] = true;
62     q.push(c);
63 }
64 }
65 }
66 }

```

## 6. Geometry

### 6.1. Convex Hull

```

1  struct Point {
2      double x, y;
3      bool operator<(const Point& p) {
4          return x < p.x || (x == p.x && y < p.y);
5      }
6  };
7
8  double isLeft(Point o, Point a, Point b) {
9      return (a.x - o.x) * (b.y - o.y) - (a.y - o.y) * (b.x - o.x);
10 }
11
12 vector<Point> upper_hull(vector<Point>& P) {
13     // sort points lexicographically
14     int n = P.size(), k = 0;
15     sort(P.begin(), P.end());
16
17     // build upper hull
18     vector<Point> uh(n);
19     invrep (i, n-1, 0) {
20         while (k >= 2 && isLeft(uh[k-2], uh[k-1], P[i]) <= 0) k--;
21         uh[k++] = P[i];
22     }
23     uh.resize(k);
24     return uh;
25 }
26
27 vector<Point> lower_hull(vector<Point>& P) {
28     // sort points lexicographically
29     int n = P.size(), k = 0;
30     sort(P.begin(), P.end());
31
32     // collect lower hull
33     vector<Point> lh(n);
34     rep (i, 0, n-1) {
35         while (k >= 2 && isLeft(lh[k-2], lh[k-1], P[i]) <= 0) k--;
36         lh[k++] = P[i];
37     }
38     lh.resize(k);
39     return lh;

```

```

40 }
41
42 vector<Point> convex_hull(vector<Point>& P) {
43     int n = P.size(), k = 0;
44
45     // set initial capacity
46     vector<Point> H(2*n);
47
48     // Sort points lexicographically
49     sort(P.begin(), P.end());
50
51     // Build lower hull
52     for (int i = 0; i < n; ++i) {
53         while (k >= 2 && isLeft(H[k-2], H[k-1], P[i]) <= 0) k--;
54         H[k++] = P[i];
55     }
56
57     // Build upper hull
58     for (int i = n-2, t = k+1; i >= 0; i--) {
59         while (k >= t && isLeft(H[k-2], H[k-1], P[i]) <= 0) k--;
60         H[k++] = P[i];
61     }
62
63     // remove extra space
64     H.resize(k-1);
65     return H;
66 }

```

### 6.2. Geometry 2D Utils

```

1  /* ===== */
2  /* Example of Point Definition */
3  /* ===== */
4
5  struct Point {
6      double x, y;
7      bool operator==(const Point& p) const { return x==p.x && y == p.y; }
8      Point operator+(const Point& p) const { return {x+p.x, y+p.y}; }
9      Point operator-(const Point& p) const { return {x-p.x, y-p.y}; }
10     Point operator*(double d) const { return {x*d, y*d}; }
11     double norm2() const { return x*x + y*y; }
12     double norm() const { return sqrt(norm2()); }
13     double dot(const Point& p) const { return x*p.x + y*p.y; }
14     Point unit() const {
15         double d = norm();
16         return {x/d, y/d};
17     }
18 };
19
20 /* ===== */
21 /* Angle Comparison */
22 /* ===== */
23
24 // -----

```

```

25 // method 1: atan2()
26 #include <cmath>
27 const double PI = atan(1) * 4;
28 double angle(double x, double y) {
29     double a = atan2(y, x);
30     return (a < 0) ? (a + 2 * PI) : a;
31 }
32 int cmpAngles(double x1, double y1, double x2, double y2) {
33     double a1 = angle(x1,y1);
34     double a2 = angle(x2,y2);
35     return (a1 < a2) ? -1 : (a1 == a2) ? 0 : 1;
36 }
37
38 // -----
39 // method 2: quadrants + slopes
40 // this is the preferred method when coordinates
41 // are given as integers
42 #include <cmath>
43 enum Quadrant { UpRight, Up, UpLeft, DownLeft, Down, DownRight };
44 int getQuadrant(int x, int y) {
45     if (x > 0) return (y >= 0) ? UpRight : DownRight;
46     if (x < 0) return (y >= 0) ? UpLeft : DownLeft;
47     return (y >= 0) ? Up : Down;
48 }
49 int cmpAngles(int x1, int y1, int x2, int y2) {
50     int q1 = getQuadrant(x1,y1);
51     int q2 = getQuadrant(x2,y2);
52     if (q1 > q2) return 1;
53     if (q1 < q2) return -1;
54     int m1 = abs(y1 * x2);
55     int m2 = abs(y2 * x1);
56     switch (q1) {
57         case UpRight:
58         case DownLeft:
59             return (m1 > m2) ? 1 : (m1 < m2) ? -1 : 0;
60         case UpLeft:
61         case DownRight:
62             return (m1 > m2) ? -1 : (m1 < m2) ? 1 : 0;
63         default: return 0;
64     }
65 }
66
67 /* ===== */
68 /* Straight Line Hashing (integer coords) */
69 /* ===== */
70
71 struct Point {int x, y; };
72 struct Line { int a, b, c; };
73
74 int gcd(int a, int b) {
75     a = abs(a);
76     b = abs(b);
77     while(b) {
78         int c = a;

```

```

79     a = b;
80     b = c % b;
81     }
82     return a;
83 }
84
85 // Line = {a,b,c} such that a*x + b*y + c = 0
86 Line getLine(Point p1, Point p2) {
87     int a = p1.y - p2.y;
88     int b = p2.x - p1.x;
89     int c = p1.x * (p2.y - p1.y) - p1.y * (p2.x - p1.x);
90     int sgn = (a < 0 || (a == 0 && b < 0)) ? -1 : 1;
91     int f = gcd(a, gcd(b, c)) * sgn;
92     a /= f;
93     b /= f;
94     c /= f;
95     return {a, b, c};
96 }
97
98 /* ===== */
99 /* Point - Segment distance */
100 /* ===== */
101
102 // get distance between p and truncated projection of p on segment s -> e
103 double point_segment_dist(const Point& p, const Point& s, const Point& e) {
104     if (s==e) return (p-s).norm(); // segment is a single point
105     double t = min(1.0, max(0.0, (p-s).dot(e-s) / (e-s).norm2()));
106     return (s+(e-s)*t-p).norm();
107 }
108
109 /* ===== */
110 /* Point - Line distance */
111 /* ===== */
112
113 // get distance between p and projection of p on line <- a - b ->
114 double point_line_dist(const Point& p, const Point& a, const Point& b) {
115     double t = (p-a).dot(b-a) / (b-a).norm2();
116     return (a+(b-a)*t-p).norm();
117 }

```

### 6.3. Point Inside Polygon

```

1  /* ===== */
2  /* Point in Polygon */
3  /* ===== */
4
5  #include <vector>
6  struct Point { float x, y; };
7
8  /* signed area of p0 with respect to (p1 -> p2) */
9  float isLeft(Point p0, Point p1, Point p2) {
10     return (p1.x - p0.x) * (p2.y - p0.y)
11         - (p2.x - p0.x) * (p1.y - p0.y);
12 }

```

```

13
14 // -----
15 // General methods: for complex / simple polygons
16
17 /* Nonzero Rule (winding number) */
18 bool inPolygon_nonzero(Point p, vector<Point>& pts) {
19     int wn = 0; // winding number
20     Point prev = pts.back();
21     rep (i, 0, (int)pts.size() - 1) {
22         Point curr = pts[i];
23         if (prev.y <= p.y) {
24             if (p.y < curr.y && isLeft(p, prev, curr) > 0)
25                 ++wn; // upward & left
26         } else {
27             if (p.y >= curr.y && isLeft(p, prev, curr) < 0)
28                 --wn; // downward & right
29         }
30         prev = curr;
31     }
32     return wn != 0; // non-zero :)
33 }
34
35 /* EvenOdd Rule (ray casting - crossing number) */
36 bool inPolygon_evenodd(Point p, vector<Point>& pts) {
37     int cn = 0; // crossing number
38     Point prev = pts.back();
39     rep (i, 0, (int)pts.size() - 1) {
40         Point curr = pts[i];
41         if (((prev.y <= p.y) && (p.y < curr.y)) // upward crossing
42             || ((prev.y > p.y) && (p.y >= curr.y))) { // downward crossing
43             // check intersect's x-coordinate to the right of p
44             float t = (p.y - prev.y) / (curr.y - prev.y);
45             if (p.x < prev.x + t * (curr.x - prev.x))
46                 ++cn;
47         }
48         prev = curr;
49     }
50     return (cn & 1); // odd -> in, even -> out
51 }
52
53 // -----
54 // Convex Polygon method: check orientation changes
55 bool inConvexPolygon(Point p, vector<Point>& pts) {
56     Point prev_p = pts.back();
57     Point curr_p;
58     float prev_orient = 0;
59     float curr_orient;
60     rep (i, 0, (int)pts.size() - 1) {
61         curr_p = pts[i];
62         curr_orient = isLeft(p, prev_p, curr_p);
63         if ((prev_orient < 0 && curr_orient > 0)
64             || (prev_orient > 0 && curr_orient < 0))
65             return false;
66         prev_p = curr_p;

```

```

67     prev_orient = curr_orient;
68 }
69 return true;
70 }

```

## 6.4. Polygon Area

```

1 /* ===== */
2 /* Area of 2D non self intersecting Polygon */
3 /* ===== */
4 //based on Green Theorem
5
6 #include <bits/stdc++.h>
7 int N = 1000;
8 struct Point { int x, y; };
9 Point P[N];
10
11 double area() {
12     int A = 0;
13     for (int i = N-1, j = 0; j < N; i=j++)
14         A += (P[i].x + P[j].x) * (P[j].y - P[i].y);
15     return fabs(A * 0.5);
16 }

```

## 7. Mathematics

### 7.1. Modular Arithmetics

```

1 /* ===== */
2 /* Binary Modular Exponentiation */
3 /* ===== */
4 int mod_pow(int b, int e, int m) {
5     if (e == 1)
6         return b % m;
7     int he = e / 2;
8     int x = mod_pow(b, he, m);
9     x = (x * x) % m;
10    if (e % 2 == 1)
11        x = (x * b) % m;
12    return x;
13 }
14
15 /* ===== */
16 /* GCD (greatest common divisor) */
17 /* ===== */
18 // euclid algorithm
19 int gcd (int a, int b) {
20     while (b) {
21         int aux = a;
22         a = b;
23         b = aux % b;
24     }

```

```

25     return a;
26 }
27
28 /* ===== */
29 /* GCD extended */
30 /* ===== */
31 // extended euclid algorithm
32 // a * x + b * y = d = gcd(a, b)
33 // x = x0 + n * (b/d)
34 // y = y0 - n * (a/d)
35 void gcdext(int a, int b, int& d, int& x, int& y) {
36     if (b == 0) { x = 1; y = 0; d = a; return; }
37     gcdext(b, a % b, d, x, y);
38     int x1 = y;
39     int y1 = x - y * (a / b);
40     x = x1;
41     y = y1;
42 }
43
44 /* ===== */
45 /* Integer Root Square */
46 /* ===== */
47
48 // using sqrt()
49 bool perfect_square(ll x, ll& root) {
50     if (x < 0) return false;
51     root = (ll)sqrt(x);
52     return (root * root == x || ++root * root == x);
53 }
54
55 // Newton's method
56 ll isqrt(ll x) {
57     ll y0 = x;
58     while (true) {
59         ll y1 = (y0 + x / y0) / 2;
60         if (y1 == y0) break;
61         y0 = y1;
62     }
63     return y0;
64 }
65 bool isPerfectSquare(ll x, ll& root) {
66     root = isqrt(x);
67     return root * root == x;
68 }

```

## 7.2. Modular Fibonacci

```

1 //=====
2 // Modular Fibonacci with (Modular) Matrix Exponentiation
3 //=====
4 //source: http://mathoverflow.net/questions/40816/fibonacci-series-mod-a-number
5 #include <cstdio>
6 #include <vector>
7 using namespace std;

```

```

8 typedef unsigned long long ull;
9 const ull MOD = 1000000000;
10
11 vector<ull> mult(const vector<ull>& A, const vector<ull>& B) {
12     vector<ull> res
13     { ((A[0] * B[0]) % MOD) + ((A[1] * B[2]) % MOD) % MOD, //m11
14       ((A[0] * B[1]) % MOD) + ((A[1] * B[3]) % MOD) % MOD, //m12
15       ((A[2] * B[0]) % MOD) + ((A[3] * B[2]) % MOD) % MOD, //m21
16       ((A[2] * B[1]) % MOD) + ((A[3] * B[3]) % MOD) % MOD //m22
17     };
18     return res;
19 }
20
21 vector<ull> raise(const vector<ull>& matrix, ull exp) {
22     if (exp == 1)
23         return matrix;
24     ull m = exp / 2;
25     vector<ull> A = raise(matrix, m);
26     if (exp % 2 == 0)
27         return mult(A, A);
28     else
29         return mult(mult(A, A), matrix);
30 }
31
32 int main() {
33     int P;
34     int k;
35     ull y;
36     scanf("%i", &P);
37     vector<ull> fib_matrix { 1, 1, 1, 0 }; //starting fibonacci matrix [f2, f1, f1, f0]
38     while (P-- > 0) {
39         scanf("%i %llu", &k, &y);
40         vector<ull> ansm = raise(fib_matrix, y);
41         ull ans = ansm[1];
42         printf("%i %llu\n", k, ans);
43     }
44     return 0;
45 }

```

## 7.3. Prime Numbers

```

1 //=====
2 // Sieve of Eratosthenes (all primes up to N)
3 //=====
4 void collect_primes_up_to(int n, vector<int>& primes) {
5     vector<bool> isPrime(n + 1, true);
6     int limit = (int) floor(sqrt(n));
7     for (int i = 2; i <= limit; ++i)
8         if (isPrime[i])
9             for (int j = i * i; j <= n; j += i)
10                 isPrime[j] = false;
11     for (int i = 2; i <= n; ++i)
12         if (isPrime[i])
13             primes.push_back(i);

```

```

14 }
15
16
17 //=====
18 // Prime Factorization of Factorials
19 //=====
20 // source: http://mathforum.org/library/drmath/view/67291.html
21 int N = 9999;
22 int pcount[N];
23 vector<int> primes;
24 collect_primes_up_to(N,primes);
25 int number = 12312; //the number we want the prime factorization of
26 for (int i = 0; i < (int)primes.size() && primes[i] <= N; ++i) {
27     int p = primes[i];
28     pcount[p] = 0;
29     int n = number;
30     while ((n /= p) > 0)
31         pcount[p] += n;
32 }

```

## 8. Strings

### 8.1. Suffix Array

```

1 #include <algorithm>
2 #include <cstdio>
3 #include <cstring>
4 using namespace std;
5
6 typedef pair<int, int> ii;
7
8 #define MAX_N 100010 // second approach: O(n log n)
9 char T[MAX_N]; // the input string, up to 100K characters
10 int n; // the length of input string
11 int RA[MAX_N], tempRA[MAX_N]; // rank array and temporary rank array
12 int SA[MAX_N], tempSA[MAX_N]; // suffix array and temporary suffix array
13 int c[MAX_N]; // for counting/radix sort
14
15 bool cmp(int a, int b) { return strcmp(T + a, T + b) < 0; } // compare
16
17 void constructSA_slow() { // cannot go beyond 1000 characters
18     for (int i = 0; i < n; i++) SA[i] = i; // initial SA: {0, 1, 2, ..., n-1}
19     sort(SA, SA + n, cmp); // sort: O(n log n) * compare: O(n) = O(n^2 log n)
20 }
21
22 void countingSort(int k) { // O(n)
23     int i, sum, maxi = max(300, n); // up to 255 ASCII chars or length of n
24     memset(c, 0, sizeof c); // clear frequency table
25     for (i = 0; i < n; i++) // count the frequency of each integer rank
26         c[i + k < n ? RA[i + k] : 0]++;
27     for (i = sum = 0; i < maxi; i++) {
28         int t = c[i]; c[i] = sum; sum += t;
29     }

```

```

30     for (i = 0; i < n; i++) // shuffle the suffix array if necessary
31         tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] = SA[i];
32     for (i = 0; i < n; i++) // update the suffix array SA
33         SA[i] = tempSA[i];
34 }
35
36 void constructSA() { // this version can go up to 100000 characters
37     int i, k, r;
38     for (i = 0; i < n; i++) RA[i] = T[i]; // initial rankings
39     for (i = 0; i < n; i++) SA[i] = i; // initial SA: {0, 1, 2, ..., n-1}
40     for (k = 1; k < n; k <= 1) { // repeat sorting process log n times
41         countingSort(k); // actually radix sort: sort based on the second item
42         countingSort(0); // then (stable) sort based on the first item
43         tempRA[SA[0]] = r = 0; // re-ranking; start from rank r = 0
44         for (i = 1; i < n; i++) // compare adjacent suffixes
45             tempRA[SA[i]] = // if same pair => same rank r; otherwise, increase r
46                 (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k]) ? r : ++r;
47         for (i = 0; i < n; i++) // update the rank array RA
48             RA[i] = tempRA[i];
49         if (RA[SA[n-1]] == n-1) break; // nice optimization trick
50     } }

```

## 9. Parsers

### 9.1. Arithmetic Parser

```

1 /**
2  * Implementation of LL(1), recursive-descent Parser
3  * for Arithmetic Expressions
4  */
5 #include <cstdio>
6 #include <iostream>
7 #include <string>
8 #include <stack>
9 #include <vector>
10 #include <cstdlib>
11 #include <stdexcept>
12 #include <cmath>
13 using namespace std;
14
15 #define rep(i,a,b) for(int i=a; i<=b; ++i)
16
17 char errorBuffer[200];
18
19 enum Terminal { NUMBER, MINUS, PLUS, DIV, MULT, SQRT, OPEN_PAREN, CLOSE_PAREN, END };
20
21 const char * terminal2String(Terminal t) {
22     switch (t) {
23         case NUMBER: return "NUMBER";
24         case MINUS: return "MINUS";
25         case PLUS: return "PLUS";
26         case DIV: return "DIV";
27         case MULT: return "MULT";

```



```

28 case Sqrt: return "Sqrt";
29 case OPEN_PAREN: return "OPEN_PAREN";
30 case CLOSE_PAREN: return "CLOSE_PAREN";
31 default: return "END";
32 }
33 }
34
35 struct Token {
36     Terminal terminal;
37     Token(Terminal t) :
38         terminal(t) {
39     }
40 };
41
42 struct NumberToken: Token {
43     double value;
44     NumberToken(double value) :
45         Token(NUMBER), value(value) {
46     }
47 };
48
49 typedef vector<Token*> vtp;
50
51 void skipWhitespace(const char* buffer, int& offset) {
52     while (true) {
53         char c = buffer[offset];
54         if (c == ' ' || c == '\t') offset++;
55         else break;
56     }
57 }
58
59 bool isDigit(char c) {
60     return '0' <= c && c <= '9';
61 }
62
63 Token* getNextToken(const char* buffer, int& offset) {
64     skipWhitespace(buffer, offset);
65     char c = buffer[offset];
66     switch (c) {
67     case '(':
68         offset++;
69         return new Token(OPEN_PAREN);
70     case ')':
71         offset++;
72         return new Token(CLOSE_PAREN);
73     case '*':
74         offset++;
75         return new Token(MULT);
76     case '/':
77         offset++;
78         return new Token(DIV);
79     case '+':
80         offset++;

```

```

82     return new Token(PLUS);
83 case '-':
84     offset++;
85     return new Token(MINUS);
86 case '\0':
87     return new Token(END);
88 case 's':{
89     rep(i,0,3) if (buffer[offset + i] != "sqrt"[i]) {
90         sprintf(errorBuffer, "unexpected char '%c' at position %d\n", buffer[offset + i],
91             offset + i);
92         throw std::runtime_error(errorBuffer);
93     }
94     offset += 4;
95     return new Token(Sqrt);
96 }
97 default:
98     if (isDigit(c)) {
99         char* endp;
100         double num = strtod(buffer + offset, &endp);
101         offset = endp - buffer;
102         return new NumberToken(num);
103     }
104     sprintf(errorBuffer, "unexpected char '%c' at position %d\n", c, offset);
105     throw std::runtime_error(errorBuffer);
106 }
107
108 struct Node {
109     virtual ~Node() {};
110     virtual double eval() = 0;
111 };
112 struct DoubleOpNode: Node {
113     Node* left;
114     Node* right;
115     DoubleOpNode(Node* left, Node* right): left(left), right(right) {}
116     ~DoubleOpNode() { delete left; delete right; }
117 };
118 struct SingleOpNode: Node {
119     Node* child;
120     SingleOpNode(Node* child): child(child) {}
121     ~SingleOpNode() { delete child; }
122 };
123 struct AddNode : DoubleOpNode {
124     AddNode(Node* left, Node* right) : DoubleOpNode(left, right) {}
125     double eval() { return left->eval() + right->eval(); }
126 };
127 struct SubNode : DoubleOpNode {
128     SubNode(Node* left, Node* right) : DoubleOpNode(left, right) {}
129     double eval() { return left->eval() - right->eval(); }
130 };
131 struct MultNode : DoubleOpNode {
132     MultNode(Node* left, Node* right) : DoubleOpNode(left, right) {}
133     double eval() { return left->eval() * right->eval(); }
134 };

```

```

135 struct DivNode : DoubleOpNode {
136     DivNode(Node* left, Node* right) : DoubleOpNode(left, right) {}
137     double eval() { return left->eval() / right->eval(); }
138 };
139 struct NegNode : SingleOpNode {
140     NegNode(Node* child) : SingleOpNode(child) {}
141     double eval() { return -child->eval(); }
142 };
143 struct SqrtNode : SingleOpNode {
144     SqrtNode(Node* child) : SingleOpNode(child) {}
145     double eval() { return sqrt(child->eval()); }
146 };
147 struct IntegerNode : Node {
148     double value;
149     IntegerNode(double value) : value(value) {}
150     double eval() { return value; }
151 };
152
153 /**
154  * Context Free Grammar:
155  * Root    -> AddSum1 END
156  * AddSum1  -> MultDiv1 AddSum2
157  * AddSum2  -> + MultDiv1 AddSum2 | - MultDiv1 AddSum2 | epsilon
158  * MultDiv1 -> Term MultDiv2
159  * MultDiv2 -> * Term MultDiv2 | / Term MultDiv2 | epsilon
160  * Term     -> - Term | (AddSum1) | SQRT(AddSum1) | NUMBER
161  */
162
163 vector<Token*> tokens;
164 int offset;
165 stack<Node*> nodes;
166
167 void throwUnexpectedTerminalException(Terminal terminal, int offset);
168 void matchAndConsume(Terminal terminal);
169 void parseTerm();
170 void parseMultDiv1();
171 void parseMultDiv2();
172 void parseAddSub1();
173 void parseAddSub2();
174 void parseRoot();
175
176 template<typename T>
177 void swap2for1() {
178     Node* r = nodes.top(); nodes.pop();
179     Node* l = nodes.top(); nodes.pop();
180     nodes.push(new T(l,r));
181 }
182
183 template<typename T>
184 void swap1for1() {
185     Node* n = nodes.top(); nodes.pop();
186     nodes.push(new T(n));
187 }
188

```

```

189 void throwUnexpectedTerminalException(Terminal terminal, int offset) {
190     sprintf(errorBuffer, "unexpected terminal %s at position %d\n", terminal2String(
191         terminal), offset);
192     throw std::runtime_error(errorBuffer);
193 }
194 void matchAndConsume(Terminal terminal) {
195     if (tokens[offset]->terminal != terminal) {
196         sprintf(errorBuffer, "expected terminal %s but found %s\n",
197             terminal2String(terminal),
198             terminal2String(tokens[offset]->terminal));
199         throw std::runtime_error(errorBuffer);
200     }
201     offset++;
202 }
203 void parseTerm() {
204     Token* t = tokens[offset];
205     switch (t->terminal) {
206         case MINUS: {
207             offset++;
208             parseTerm();
209             // generate node
210             swap1for1<NegNode>();
211             break;
212         }
213         case OPEN_PAREN: {
214             offset++;
215             parseAddSub1();
216             matchAndConsume(CLOSE_PAREN);
217             break;
218         }
219         case Sqrt: {
220             offset++;
221             matchAndConsume(OPEN_PAREN);
222             parseAddSub1();
223             matchAndConsume(CLOSE_PAREN);
224             swap1for1<SqrtNode>();
225             break;
226         }
227         case NUMBER: {
228             offset++;
229             // generate node
230             double value = static_cast<NumberToken*>(t)->value;
231             nodes.push(new IntegerNode(value));
232             break;
233         }
234         default:
235             throwUnexpectedTerminalException(t->terminal, offset);
236             break;
237     }
238 }
239 void parseMultDiv1() {
240     parseTerm();
241     parseMultDiv2();
242 }

```

```
242 void parseMultDiv2() {
243     Token* t = tokens[offset];
244     switch (t->terminal) {
245         case MULT: {
246             offset++;
247             parseTerm();
248             // generate node
249             swap2for1<MultNode>();
250             // resume parsing
251             parseMultDiv2();
252             break;
253         }
254         case DIV: {
255             offset++;
256             parseTerm();
257             // generate node
258             swap2for1<DivNode>();
259             // resume parsing
260             parseMultDiv2();
261             break;
262         }
263         // follow set
264         case PLUS: case MINUS: case END: case CLOSE_PAREN:
265             break;
266         default:
267             throwUnexpectedTerminalException(t->terminal, offset);
268             break;
269     }
270 }
271 void parseAddSub2() {
272     Token* t = tokens[offset];
273     switch (t->terminal) {
274         case PLUS: {
275             offset++;
276             parseMultDiv1();
277             // generate node
278             swap2for1<AddNode>();
279             // resume parsing
280             parseAddSub2();
281             break;
282         }
283         case MINUS: {
284             offset++;
285             parseMultDiv1();
286             // generate node
287             swap2for1<SubNode>();
288             // resume parsing
289             parseAddSub2();
290             break;
291         }
292         // follow set
293         case END: case CLOSE_PAREN:
294             break;
295         default:
```

```
296             throwUnexpectedTerminalException(t->terminal, offset);
297             break;
298     }
299 }
300 void parseAddSub1() {
301     parseMultDiv1();
302     parseAddSub2();
303 }
304 void parseRoot() {
305     parseAddSub1();
306     matchAndConsume(END);
307 }
308
309
310 int main() {
311     string line;
312     while (true) {
313         /* read input */
314         getline(cin, line);
315         if (line == "exit") break;
316
317         /* get tokens */
318         int index = 0;
319         while(true) {
320             Token* t = getNextToken(line.c_str(), index);
321             tokens.push_back(t);
322             if (t->terminal == END) break;
323         }
324
325         /* parse tokens to generate AST */
326         parseRoot();
327         Node* root = nodes.top();
328
329         /* print result */
330         printf("==> %lf\n", root->eval());
331
332         /* clean memory */
333         delete root;
334         for (int i = 0, l = tokens.size(); i < l; ++i) delete tokens[i];
335         tokens.clear();
336         nodes.pop();
337         offset = 0;
338     }
339     return 0;
340 }
```