

Contents

1	Info About Memory and Time Limits	2
2	C++ Cheat Sheet	2
2.1	Headers	2
2.2	Cheat Sheet	2
3	General Algorithms	8
3.1	Search	8
3.1.1	Binary Search	8
3.1.2	Ternary Search	8
3.2	Brute Force	8
4	Data Structures	8
4.1	Segment Tree	8
4.1.1	Lazy	8
4.1.2	Iterative	10
4.2	Wavelet Tree	11
5	Dynamic Programming	12
5.1	Knapsack	12
5.2	Matrix Chain Multiplication	12
5.3	Longest Increasing Subsequence	13
6	Graphs	13
6.1	Graph Traversal	13
6.1.1	Breadth First Search	13
6.1.2	Recursive Depth First Search	13
6.1.3	Iterative Depth First Search	13
6.2	Shortest Path Algorithms	14
6.2.1	Dijkstra	14
6.2.2	Bellman Ford	14
6.2.3	Floyd Warshall	14
6.3	Minimum Spanning Tree (MST)	14
6.3.1	Kruskal	14
6.4	Lowest Common Ancestor (LCA)	15
6.5	Max Flow	16
6.6	Others	16
6.6.1	Diameter of a tree	16
7	Mathematics	17
7.1	Useful Data	17
7.2	Modular Arithmetic	17
7.2.1	Chinese Remainder Theorem	17

7.2.2	Binomial Coefficients mod m	18
7.3	Primality Checks	20
7.3.1	Miller Rabin	20
7.3.2	Sieve of Eratosthenes	20
7.3.3	trialDivision	21
7.4	Others	21
7.4.1	Polynomials	21
7.4.2	Factorial Factorization	22
8	Geometry	22
8.1	Vectors/Points	22
8.2	Calculate Areas	25
8.2.1	Integration via Simpson's Method	25
8.2.2	Green's Theorem	25
8.3	Pick's Theorem	25
9	Strings	25
9.1	KMP	25
9.2	Rolling Hashing	26
9.3	Trie	26
9.4	Suffix Tree	27

1 Info About Memory and Time Limits

$O(f(n))$	Limite
$O(n!)$	10, ..., 11
$O(2^n n^2)$	15, ..., 18
$O(2^n n)$	18, ..., 21
$O(n^4)$	100
$O(n^3)$	500 ¹
$O(n^2 \log^2 n)$	1000
$O(n^2 \log n)$	2000
$O(n^2)$	1e4 ²
$O(n \log^2 n)$	3e5
$O(n \log n)$	1e6
$O(n)$	1e8 ³

2 C++ Cheat Sheet

2.1 Headers

```

1 #pragma GCC optimize("Ofast")
2 #include <bits/stdc++.h> //Import all
3
4 using namespace std; //Less vebose code
5
6 typedef long long ll;
7 typedef unsigned long long ull;
8 typedef pair<int, int> ii;
9 typedef tuple<int, int, int> iii;
10 typedef vector<int> vi;
11 typedef vector<ll> vll;
12 typedef vector<ii> vii;
13
14 typedef vector<vi> graph;
15 typedef vector<vii> wgraph;
16
17 #ifndef declaraciones_h
18 #define declaraciones_h
19
20 // Reps are inclusive exclusive (i.e. range is [a,b))
21 #define rep(i, n) for (int i = 0; i < (int)n; i++)
22 #define repx(i, a, b) for (int i = a; i < (int)b; i++)
23 #define invrep(i, a, b) for (int i = b; i-- > (int)a;)
24
25 #define pb push_back
26 #define eb emplace_back

```

¹Este caso esta justo en el limite de tiempo, además en 256 MB cabe a los una matriz de 400³ ints

²En general solo funciona hasta 6e3

³En general solo funciona hasta 4e7

```

27 #define ppb pop_back
28
29 // Base two log for ints and for ll
30 #define lg(x) (31 - __builtin_clz(x))
31 #define lgg(x) (63 - __builtin_clzll(x))
32 #define gcd __gcd
33
34 // Or LLONG_MAX for ll
35 #define INF INT_MAX
36
37 #define umap unordered_map
38 #define uset unordered_set
39
40 //Debugs single variables (e.g. int, string)
41 #define debugx(x) cerr << #x << ": " << x << endl
42 //Debugs Iterables (e.g. vi, uset<int>)
43 #define debugv(v) \
44     cerr << #v << ":\n"; \
45     for (auto e : v) \
46     { \
47         cerr << " " << e; \
48     } \
49     cerr << endl
50 //Debugs Iterables of Iterables (e.g. graph, umap<int,umap<int, int>)
51 #define debugm(m) \
52     cerr << #m << endl; \
53     for (auto v : m) \
54     { \
55         for (auto e : v) \
56         { \
57             cerr << " " << e; \
58         } \
59     }
60 #define print(x) copy(x.begin(), x.end(), ostream_iterator<int>(cout, \
61     "")), cout << endl
62
63 //Outputs generic pairs through streams (including cerr and cout)
64 template <typename T1, typename T2>
65 ostream &operator<<(ostream &os, const pair<T1, T2> &p)
66 {
67     os << '(' << p.first << ',' << p.second << ')';
68     return os;
69 }
70 #endif

```

2.2 Cheat Sheet

```

1 // Note: This Cheat Sheet is by no means complete
2 // If you want a thorough documentation of the Standard C++ Library
3 // please refer to this link: http://www.cplusplus.com/reference/
4
5 /* ===== */
6 /* Reading from stdin */
7 /* ===== */
8 // With scanf

```

```

9  scanf("%d", &a);           //int
10 scanf("%x", &a);           // int in hexadecimal
11 scanf("%llx", &a);          // long long in hexadecimal
12 scanf("%lld", &a);          // long long int
13 scanf("%c", &c);            // char
14 scanf("%s", buffer);        // string without whitespaces
15 scanf("%f", &f);            // float
16 scanf("%lf", &d);           // double
17 scanf("%d %s %d", &a, &b); // * = consume but skip
18
19 // read until EOL
20 // - EOL not included in buffer
21 // - EOL is not consumed
22 // - nothing is written into buffer if EOF is found
23 scanf(" %[^\n]", buffer);
24
25 //reading until EOL or EOF
26 // - EOL not included in buffer
27 // - EOL is consumed
28 // - works with EOF
29 char *output = gets(buffer);
30 if (feof(stdin))
31 {
32 } // EOF file found
33 if (output == buffer)
34 {
35 } // succesful read
36 if (output == NULL)
37 {
38 } // EOF found without previous chars found
39 //example
40 while (gets(buffer) != NULL)
41 {
42     puts(buffer);
43     if (feof(stdin))
44     {
45         break;
46     }
47 }
48
49 // read single char
50 getchar();
51 while (true)
52 {
53     c = getchar();
54     if (c == EOF || c == '\n')
55         break;
56 }
57
58 /* ===== */
59 /* Printing to stdout */
60 /* ===== */
61 // With printf
62 printf("%d", a);             // int
63 printf("%u", a);             // unsigned int

```

```

64 printf("%lld", a);           // long long int
65 printf("%llu", a);           // unsigned long long int
66 printf("%c", c);             // char
67 printf("%s", buffer);        // string until \0
68 printf("%f", f);             // float
69 printf("%lf", d);            // double
70 printf("%0*.f", x, y, f);     // padding = 0, width = x, decimals = y
71 printf("(%.5s)\n", buffer); // print at most the first five characters
                                (safe to use on short strings)
72
73 // print at most first n characters (safe)
74 printf("(%.s)\n", n, buffer); // make sure that n is integer (with long
                                long I had problems)
75 //string + \n
76 puts(buffer);
77
78 /* ===== */
79 /* Reading from c string */
80 /* ===== */
81
82 // same as scanf but reading from s
83 int sscanf(const char *s, const char *format, ...);
84
85 /* ===== */
86 /* Printing to c string */
87 /* ===== */
88 // Same as printf but writing into str, the number of characters is
                                returned
89 // or negative if there is failure
90 int sprintf(char *str, const char *format, ...);
91 //example:
92 int n = sprintf(buffer, "%d plus %d is %d", a, b, a + b);
93 printf("[%s] is a string %d chars long\n", buffer, n);
94
95 /* ===== */
96 /* Peek last char of stdin */
97 /* ===== */
98 bool peekAndCheck(char c)
99 {
100     char c2 = getchar();
101     ungetc(c2, stdin); // return char to stdin
102     return c == c2;
103 }
104
105 /* ===== */
106 /* Reading from cin */
107 /* ===== */
108 // reading a line of unknown length
109 string line;
110 getline(cin, line);
111 while (getline(cin, line))
112 {
113 }
114
115 // Optimizations with cin/cout

```

```

116 ios::sync_with_stdio(0);
117 cin.tie(0);
118 cout.tie(0);
119
120 // Fix precision on cout
121 cout.setf(ios::fixed);
122 cout.precision(4); // e.g. 1.000
123
124 /* ===== */
125 /* USING PAIRS AND TUPLES */
126 /* ===== */
127 // ii = pair<int,int>
128 ii p(5, 5);
129 ii p = make_pair(5, 5)
130     ii p = {5, 5};
131 int x = p.first, y = p.second;
132 // iii = tuple<int,int,int>
133 iii t(5, 5, 5);
134 tie(x, y, z) = t;
135 tie(x, y, z) = make_tuple(5, 5, 5);
136 get<0>(t)++;
137 get<1>(t)--;
138
139 /* ===== */
140 /* CONVERTING FROM STRING TO NUMBERS */
141 /* ===== */
142 //-----
143 // string to int
144 // option #1:
145 int atoi(const char *str);
146 // option #2:
147 sscanf(string, "%d", &i);
148 //-----
149 // string to long int:
150 // option #1:
151 long int strtol(const char *str, char **endptr, int base);
152 // it only works skipping whitespaces, so make sure your numbers
153 // are surrounded by whitespaces only
154 // Example:
155 char szNumbers[] = "2001 60c0c0 -1101110100110100100000 0x6ffff";
156 char *pEnd;
157 long int li1, li2, li3, li4;
158 li1 = strtol(szNumbers, &pEnd, 10);
159 li2 = strtol(pEnd, &pEnd, 16);
160 li3 = strtol(pEnd, &pEnd, 2);
161 li4 = strtol(pEnd, NULL, 0);
162 printf("The decimal equivalents are: %ld, %ld, %ld and %ld.\n", li1, li2
    , li3, li4);
163
164 // option #2:
165 long int atol(const char *str);
166 // option #3:
167 sscanf(string, "%ld", &l);
168 //-----
169 // string to long long int:
170 // option #1:

```

```

170 long long int strtoll(const char *str, char **endptr, int base);
171 // option #2:
172 sscanf(string, "%lld", &l);
173 //-----
174 // string to double:
175 // option #1:
176 double strtod(const char *str, char **endptr); //similar to strtol
177 // option #2:
178 double atof(const char *str);
179 // option #3:
180 sscanf(string, "%lf", &d);
181
182 /* ===== */
183 /* C STRING UTILITY FUNCTIONS */
184 /* ===== */
185 int strcmp(const char *str1, const char *str2); // (-1,0,1)
186 int memcmp(const void *ptr1, const void *ptr2, size_t num); // (-1,0,1)
187 void *memcpy(void *destination, const void *source, size_t num);
188
189 /* ===== */
190 /* C++ STRING UTILITY FUNCTIONS */
191 /* ===== */
192 // read tokens from string
193 string s = "tok1 tok2 tok3";
194 string tok;
195 stringstream ss(s);
196 while (getline(ss, tok, ' '))
197     printf("tok = %s\n", tok.c_str());
198
199 // split a string by a single char delimiter
200 void split(const string &s, char delim, vector<string> &elems)
201 {
202     stringstream ss(s);
203     string item;
204     while (getline(ss, item, delim))
205         elems.push_back(item);
206 }
207
208 // find index of string or char within string
209 string str = "random";
210 std::size_t pos = str.find("ra");
211 std::size_t pos = str.find('m');
212 if (pos == string::npos) // not found
213
214     // substrings
215     string subs = str.substr(pos, length);
216 string subs = str.substr(pos); // default: to the end of the string
217
218 // std::string from cstring's substring
219 const char *s = "bla1 bla2";
220 int offset = 5, len = 4;
221 string subs(s + offset, len); // bla2
222
223 // -----
224 // string comparisons

```

```

225 int compare(const string &str) const;
226 int compare(size_t pos, size_t len, const string &str) const;
227 int compare(size_t pos, size_t len, const string &str,
228             size_t subpos, size_t sublen) const;
229 int compare(const char *s) const;
230 int compare(size_t pos, size_t len, const char *s) const;
231
232 // examples
233 // 1) check string begins with another string
234 string prefix = "prefix";
235 string word = "prefix suffix";
236 word.compare(0, prefix.size(), prefix);
237
238 /* ===== */
239 /* OPERATOR OVERLOADING */
240 /* ===== */
241
242 //-----
243 // method #1: inside struct
244 struct Point
245 {
246     int x, y;
247     bool operator<(const Point &p) const
248     {
249         if (x != p.x)
250             return x < p.x;
251         return y < p.y;
252     }
253     bool operator>(const Point &p) const
254     {
255         if (x != p.x)
256             return x > p.x;
257         return y > p.y;
258     }
259     bool operator==(const Point &p) const
260     {
261         return x == p.x && y == p.y;
262     }
263 };
264
265 //-----
266 // method #2: outside struct
267 struct Point
268 {
269     int x, y;
270 };
271 bool operator<(const Point &a, const Point &b)
272 {
273     if (a.x != b.x)
274         return a.x < b.x;
275     return a.y < b.y;
276 }
277 bool operator>(const Point &a, const Point &b)
278 {
279     if (a.x != b.x)

```

```

280         return a.x > b.x;
281     return a.y > b.y;
282 }
283 bool operator==(const Point &a, const Point &b)
284 {
285     return a.x == b.x && a.y == b.y;
286 }
287
288 // Note: if you overload the < operator for a custom struct,
289 // then you can use that struct with any library function
290 // or data structure that requires the < operator
291 // Examples:
292 priority_queue<Point> pq;
293 vector<Point> pts;
294 sort(pts.begin(), pts.end());
295 lower_bound(pts.begin(), pts.end(), {1, 2});
296 upper_bound(pts.begin(), pts.end(), {1, 2});
297 set<Point> pt_set;
298 map<Point, int> pt_map;
299
300 /* ===== */
301 /* CUSTOM COMPARISONS */
302 /* ===== */
303 // method #1: operator overloading
304 // method #2: custom comparison function
305 bool cmp(const Point &a, const Point &b)
306 {
307     if (a.x != b.x)
308         return a.x < b.x;
309     return a.y < b.y;
310 }
311 // method #3: functor
312 struct cmp
313 {
314     bool operator()(const Point &a, const Point &b)
315     {
316         if (a.x != b.x)
317             return a.x < b.x;
318         return a.y < b.y;
319     }
320 };
321 // without operator overloading, you would have to use
322 // an explicit comparison method when using library
323 // functions or data structures that require sorting
324 priority_queue<Point, vector<Point>, cmp> pq;
325 vector<Point> pts;
326 sort(pts.begin(), pts.end(), cmp);
327 lower_bound(pts.begin(), pts.end(), {1, 2}, cmp);
328 upper_bound(pts.begin(), pts.end(), {1, 2}, cmp);
329 set<Point, cmp> pt_set;
330 map<Point, int, cmp> pt_map;
331
332 /* ===== */
333 /* VECTOR UTILITY FUNCTIONS */
334 /* ===== */

```

```

335 vector<int> myvector;
336 myvector.push_back(100);
337 myvector.pop_back(); // remove last element
338 myvector.back();     // peek reference to last element
339 myvector.front();    // peek reference to first element
340 myvector.clear();    // remove all elements
341 // sorting a vector
342 vector<int> foo;
343 sort(foo.begin(), foo.end());
344 sort(foo.begin(), foo.end(), std::less<int>()); // increasing
345 sort(foo.begin(), foo.end(), std::greater<int>()); // decreasing
346
347 /* ===== */
348 /* SET UTILITY FUNCTIONS */
349 /* ===== */
350 set<int> myset;
351 myset.begin(); // iterator to first elemnt
352 myset.end();   // iterator to after last element
353 myset.rbegin(); // iterator to last element
354 myset.rend();  // iterator to before first element
355 for (auto it = myset.begin(); it != myset.end(); ++it)
356 {
357     do_something(*it);
358 } // left -> right
359 for (auto it = myset.rbegin(); it != myset.rend(); ++it)
360 {
361     do_something(*it);
362 } // right -> left
363 for (auto &i : myset)
364 {
365     do_something(i);
366 } // left->right shortcut
367 auto ret = myset.insert(5); // ret.first = iterator, ret.second =
    boolean (inserted / not inserted)
368 int count = myset.erase(5); // count = how many items were erased
369 if (!myset.empty())
370 {
371 }
372 // custom comparator 1: functor
373 struct cmp
374 {
375     bool operator()(int i, int j) { return i > j; }
376 };
377 set<int, cmp> myset;
378 // custom comparator 2: function
379 bool cmp(int i, int j) { return i > j; }
380 set<int, bool (*)(int, int)> myset(cmp);
381
382 /* ===== */
383 /* MAP UTILITY FUNCTIONS */
384 /* ===== */
385 struct Point
386 {
387     int x, y;
388 };

```

```

389 bool operator<(const Point &a, const Point &b)
390 {
391     return a.x < b.x || (a.x == b.x && a.y < b.y);
392 }
393 map<Point, int> ptcounts;
394
395 // -----
396 // inserting into map
397
398 // method #1: operator[]
399 // it overwrites the value if the key already exists
400 ptcounts[{1, 2}] = 1;
401
402 // method #2: .insert(pair<key, value>)
403 // it returns a pair { iterator(key, value) , bool }
404 // if the key already exists, it doesn't overwrite the value
405 void update_count(Point &p)
406 {
407     auto ret = ptcounts.emplace(p, 1);
408     // auto ret = ptcounts.insert(make_pair(p, 1)); //
409     if (!ret.second)
410         ret.first->second++;
411 }
412
413 // -----
414 // generating ids with map
415 int get_id(string &name)
416 {
417     static int id = 0;
418     static map<string, int> name2id;
419     auto it = name2id.find(name);
420     if (it == name2id.end())
421         return name2id[name] = id++;
422     return it->second;
423 }
424
425 /* ===== */
426 /* BITSET UTILITY FUNCTIONS */
427 /* ===== */
428 bitset<4> foo; // 0000
429 foo.size();    // 4
430 foo.set();     // 1111
431 foo.set(1, 0); // 1011
432 foo.test(1);   // false
433 foo.set(1);    // 1111
434 foo.test(1);   // true
435
436 /* ===== */
437 /* RANDOM INTEGERS */
438 /* ===== */
439 #include <cstdlib>
440 #include <ctime>
441 srand(time(NULL));
442 int x = rand() % 100; // 0-99
443 int randBetween(int a, int b)

```

```

444 { // a-b
445     return a + (rand() % (1 + b - a));
446 }
447
448 /* ===== */
449 /* CLIMITS */
450 /* ===== */
451 #include <climits>
452 INT_MIN
453 INT_MAX
454 UINT_MAX
455 LONG_MIN
456 LONG_MAX
457 ULONG_MAX
458 LLONG_MIN
459 LLONG_MAX
460 ULLONG_MAX
461
462 /* ===== */
463 /* Bitwise Tricks */
464 /* ===== */
465
466 // amount of one-bits in number
467 int __builtin_popcount(int x);
468 int __builtin_popcountl(long x);
469 int __builtin_popcountll(long long x);
470
471 // amount of leading zeros in number
472 int __builtin_clz(int x);
473 int __builtin_clzl(long x);
474 int __builtin_clzll(long long x);
475
476 // binary length of non-negative number
477 int bitlen(int x) { return sizeof(x) * 8 - __builtin_clz(x); }
478 int bitlen(long x) { return sizeof(x) * 8 - __builtin_clzl(x); }
479
480 // index of most significant bit
481 int log2(int x) { return sizeof(x) * 8 - __builtin_clz(x) - 1; }
482 int log2(long x) { return sizeof(x) * 8 - __builtin_clzl(x) - 1; }
483
484 // reverse the bits of an integer
485 int reverse_bits(int x)
486 {
487     int v = 0;
488     while (x)
489         v <<= 1, v |= x & 1, x >>= 1;
490     return v;
491 }
492
493 // get string binary representation of an integer
494 string bitstring(int x)
495 {
496     int len = sizeof(x) * 8 - __builtin_clz(x);
497     if (len == 0)
498         return "0";

```

```

499
500     char buff[len + 1];
501     buff[len] = '\0';
502     for (int i = len - 1; i >= 0; --i, x >>= 1)
503         buff[i] = (char)('0' + (x & 1));
504     return string(buff);
505 }
506
507 /* ===== */
508 /* Hexadecimal Tricks */
509 /* ===== */
510
511 // get string hex representation of an integer
512 string to_hex(int num)
513 {
514     static char buff[100];
515     static const char *hexdigits = "0123456789abcdef";
516     buff[99] = '\0';
517     int i = 98;
518     do
519     {
520         buff[i--] = hexdigits[num & 0xf];
521         num >>= 4;
522     } while (num);
523     return string(buff + i + 1);
524 }
525
526 // ['0'-'9' 'a'-'f'] -> [0 - 15]
527 int char_to_digit(char c)
528 {
529     if ('0' <= c && c <= '9')
530         return c - '0';
531     return 10 + c - 'a';
532 }
533
534 /* ===== */
535 /* Other Tricks */
536 /* ===== */
537 // swap stuff
538 int x = 1, y = 2;
539 swap(x, y);
540
541 /* ===== */
542 /* TIPS */
543 /* ===== */
544 // 1) do not use .emplace(x, y) if your struct doesn't have an explicit
545 //     constructor
546 //     instead you can use .push({x, y})
547 // 2) be careful while mixing scanf() with getline(), scanf will not
548 //     consume \n unless
549 //     you explicitly tell it to do so (e.g scanf("%d\n", &x)) )

```

3 General Algorithms

3.1 Search

3.1.1 Binary Search

```

1 // On iterables v use lower_bound(v.begin(),v.begin()+delta,key) and
  // upper_bound(v.begin(), v.begin()+delta,key)
2
3 int val;
4 vi vals;
5 bool discreteP(int x) { return x > val; }
6
7 int bin(int start, int end)
8 {
9     int left = start, right = end, mid;
10    while (left < right)
11    {
12        mid = (left + right) / 2;
13        if (discreteP(vals[mid]))
14            right = mid;
15        else
16            left = mid + 1;
17    }
18    return left;
19 }
20
21 double approx;
22 bool continuousP(double x) { return x > approx; }
23
24 double bin(double start, double end)
25 {
26     double left = start, right = end;
27     int reps = 80; //Safe numbers check if viable for problem
28     double mid;
29     rep(_, reps)
30     {
31         mid = (left + right) / 2;
32         if (continuousP(mid))
33             right = mid;
34         else
35             left = mid;
36     }
37     return mid;
38 }

```

3.1.2 Ternary Search

```

1
2 double f(double x)
3 {
4     return -x * x;
5 }
6
7 bool compare(double x, double y) { return f(x) < f(y); }

```

```

8
9 double maxTer(double start, double end) //Searches maximum of f in range
  [start, end]
10 {
11     double left = start, right = end;
12     double mid1, mid2;
13     int reps = 80;
14     rep(_, reps)
15     {
16         mid1 = left + (right - left) / 3, mid2 = right - (right - left)
          / 3;
17         if (compare(mid1, mid2))
18             left = mid1;
19         else
20             right = mid2;
21     }
22     return (mid1 + mid2) / 2; // * Can return -0!
23     // Tends to the right
24 }
25
26 double minTer(double start, double end) //Searches minimum of f in range
  [start,end]
27 {
28     double left = start, right = end;
29     double mid1, mid2;
30     int reps = 80;
31     rep(_, reps)
32     {
33         mid1 = left + (right - left) / 3, mid2 = right - (right - left)
          / 3;
34         if (not compare(mid1, mid2))
35             left = mid1;
36         else
37             right = mid2;
38     }
39     return (mid1 + mid2) / 2;
40     // Tends to the left
41 }

```

3.2 Brute Force

4 Data Structures

4.1 Segment Tree

4.1.1 Lazy

```

1
2 struct RSQ // Range sum query
3 {
4     static ll const neutro = 0;
5     static ll op(ll x, ll y)
6     {
7         return x + y;

```



```

8     }
9     static ll
10     lazy_op(int i, int j, ll x)
11     {
12         return (j - i + 1) * x;
13     }
14 };
15
16 struct RMinQ // Range minimum query
17 {
18     static ll const neutro = 1e18;
19     static ll op(ll x, ll y)
20     {
21         return min(x, y);
22     }
23     static ll
24     lazy_op(int i, int j, ll x)
25     {
26         return x;
27     }
28 };
29
30 template <class t>
31 class SegTreeLazy
32 {
33     vector<ll> arr, st, lazy;
34     int n;
35
36     void build(int u, int i, int j)
37     {
38         if (i == j)
39         {
40             st[u] = arr[i];
41             return;
42         }
43         int m = (i + j) / 2, l = u * 2 + 1, r = u * 2 + 2;
44         build(l, i, m);
45         build(r, m + 1, j);
46         st[u] = t::op(st[l], st[r]);
47     }
48
49     void propagate(int u, int i, int j, ll x)
50     {
51         // nota, las operaciones pueden ser un and, or, ..., etc.
52         st[u] += t::lazy_op(i, j, x); // incrementar el valor (+)
53         // st[u] = t::lazy_op(i, j, x); // setear el valor
54         if (i != j)
55         {
56             // incrementar el valor
57             lazy[u * 2 + 1] += x;
58             lazy[u * 2 + 2] += x;
59             // setear el valor
60             // lazy[u * 2 + 1] = x;
61             // lazy[u * 2 + 2] = x;
62         }

```

```

63         lazy[u] = 0;
64     }
65
66     ll query(int a, int b, int u, int i, int j)
67     {
68         if (j < a or b < i)
69             return t::neutro;
70         int m = (i + j) / 2, l = u * 2 + 1, r = u * 2 + 2;
71         if (lazy[u])
72             propagate(u, i, j, lazy[u]);
73         if (a <= i and j <= b)
74             return st[u];
75         ll x = query(a, b, l, i, m);
76         ll y = query(a, b, r, m + 1, j);
77         return t::op(x, y);
78     }
79
80     void update(int a, int b, ll value,
81                 int u, int i, int j)
82     {
83         int m = (i + j) / 2, l = u * 2 + 1, r = u * 2 + 2;
84         if (lazy[u])
85             propagate(u, i, j, lazy[u]);
86         if (a <= i and j <= b)
87             propagate(u, i, j, value);
88         else if (j < a or b < i)
89             return;
90         else
91         {
92             update(a, b, value, l, i, m);
93             update(a, b, value, r, m + 1, j);
94             st[u] = t::op(st[l], st[r]);
95         }
96     }
97
98     public:
99     SegTreeLazy(vector<ll> &v)
100     {
101         arr = v;
102         n = v.size();
103         st.resize(n * 4 + 5);
104         lazy.assign(n * 4 + 5, 0);
105         build(0, 0, n - 1);
106     }
107
108     ll query(int a, int b)
109     {
110         return query(a, b, 0, 0, n - 1);
111     }
112
113     void update(int a, int b, ll value)
114     {
115         update(a, b, value, 0, 0, n - 1);
116     }
117 };

```

4.1.2 Iterative

```

1 // It requires a struct for a node (e.g. prodsn)
2 // A node must have three constructors
3 // Arity 0: Constructs the identity of the operation (e.g. 1 for
4 //   prodsn)
5 // Arity 1: Constructs a leaf node from the input
6 // Arity 2: Constructs a node from its children
7 //
8 // Building the Segment Tree:
9 //   Create a vector of nodes (use constructor of arity 1).
10 //   ST<miStructNode> mySegmentTree(vectorOfNodes);
11 // Update:
12 //   mySegmentTree.set_points(index, myStructNode(input));
13 // Query:
14 //   mySegmentTree.query(l, r); (It searches on the range [l,r), and
15 //   returns a node.)
16
17 // Logic And Query
18 struct ANDQ
19 {
20     ll value;
21     ANDQ() { value = -1ll; }
22     ANDQ(ll x) { value = x; }
23     ANDQ(const ANDQ &a,
24           const ANDQ &b)
25     {
26         value = a.value & b.value;
27     }
28 };
29
30 // Interval Product (LiveArchive)
31 struct prodsn
32 {
33     int sgn;
34     prodsn() { sgn = 1; }
35     prodsn(int x)
36     {
37         sgn = (x > 0) - (x < 0);
38     }
39     prodsn(const prodsn &a,
40           const prodsn &b)
41     {
42         sgn = a.sgn * b.sgn;
43     }
44 };
45
46 // Maximum Sum (SPOJ)
47 struct maxsum
48 {
49     int first, second;
50     maxsum() { first = second = -1; }
51     maxsum(int x)
52     {

```

```

52         first = x;
53         second = -1;
54     }
55     maxsum(const maxsum &a,
56           const maxsum &b)
57     {
58         if (a.first > b.first)
59         {
60             first = a.first;
61             second = max(a.second,
62                         b.first);
63         }
64         else
65         {
66             first = b.first;
67             second = max(a.first,
68                         b.second);
69         }
70     }
71     int answer()
72     {
73         return first + second;
74     }
75 };
76
77 // Range Minimum Query
78 struct rminq
79 {
80     int value;
81     rminq() { value = INT_MAX; }
82     rminq(int x) { value = x; }
83     rminq(const rminq &a,
84           const rminq &b)
85     {
86         value = min(a.value,
87                     b.value);
88     }
89 };
90
91 template <class node>
92 class ST
93 {
94     vector<node> t;
95     int n;
96
97 public:
98     ST(vector<node> &arr)
99     {
100         n = arr.size();
101         t.resize(n * 2);
102         copy(arr.begin(), arr.end(), t.begin() + n);
103         for (int i = n - 1; i > 0; --i)
104             t[i] = node(t[i << 1], t[i << 1 | 1]);
105     }
106

```

```

107 // 0-indexed
108 void set_point(int p, const node &value)
109 {
110     for (t[p += n] = value; p > 1; p >>= 1)
111         t[p >> 1] = node(t[p], t[p ^ 1]);
112 }
113
114 // inclusive exclusive, 0-indexed
115 node query(int l, int r)
116 {
117     node ans_l, ans_r;
118     for (l += n, r += n; l < r; l >>= 1, r >>= 1)
119     {
120         if (l & 1)
121             ans_l = node(ans_l, t[l++]);
122         if (r & 1)
123             ans_r = node(t[--r], ans_r);
124     }
125     return node(ans_l, ans_r);
126 }
127 };

```

4.2 Wavelet Tree

```

1
2 class WaveTree
3 {
4     typedef vector<int>::iterator iter;
5     vector<vector<int>> r0;
6     int n, s;
7     vector<int> arrCopy;
8
9     void build(iter b, iter e, int l, int r, int u)
10    {
11        if (l == r)
12            return;
13        int m = (l + r) / 2;
14        r0[u].reserve(e - b + 1);
15        r0[u].push_back(0);
16        for (iter it = b; it != e; ++it)
17            r0[u].push_back(r0[u].back() + (*it <= m));
18        iter p = stable_partition(b, e, [=](int i) { return i <= m; });
19        build(b, p, l, m, u * 2);
20        build(p, e, m + 1, r, u * 2 + 1);
21    }
22
23    int q, w;
24    int range(int a, int b, int l, int r, int u)
25    {
26        if (r < q or w < l)
27            return 0;
28        if (q <= l and r <= w)
29            return b - a;
30        int m = (l + r) / 2, za = r0[u][a], zb = r0[u][b];
31        return range(za, zb, l, m, u * 2) +

```

```

32         range(a - za, b - zb, m + 1, r, u * 2 + 1);
33     }
34
35 public:
36     //arr[i] in [0,sigma)
37     WaveTree(vector<int> arr, int sigma)
38     {
39         n = arr.size();
40         s = sigma;
41         r0.resize(s * 2);
42         arrCopy = arr;
43         build(arr.begin(), arr.end(), 0, s - 1, 1);
44     }
45
46     //k in [1,n], [a,b] is 0-indexed, -1 if error
47     int quantile(int k, int a, int b)
48     {
49         //extra conditions disabled
50         if (/*a < 0 or b > n or*/ k < 1 or k > b - a)
51             return -1;
52         int l = 0, r = s - 1, u = 1, m, za, zb;
53         while (l != r)
54         {
55             m = (l + r) / 2;
56             za = r0[u][a];
57             zb = r0[u][b];
58             u *= 2;
59             if (k <= zb - za)
60                 a = za, b = zb, r = m;
61             else
62                 k -= zb - za, a -= za, b -= zb,
63                 l = m + 1, ++u;
64         }
65         return r;
66     }
67
68     //counts numbers in [x,y] in positions [a,b]
69     int range(int x, int y, int a, int b)
70     {
71         if (y < x or b <= a)
72             return 0;
73         q = x;
74         w = y;
75         return range(a, b, 0, s - 1, 1);
76     }
77
78     //count occurrences of x in positions [0,k]
79     int rank(int x, int k)
80     {
81         int l = 0, r = s - 1, u = 1, m, z;
82         while (l != r)
83         {
84             m = (l + r) / 2;
85             z = r0[u][k];
86             u *= 2;

```

```

87         if (x <= m)
88             k = z, r = m;
89         else
90             k -= z, l = m + 1, ++u;
91     }
92     return k;
93 }
94
95 //x in [0,sigma)
96 void push_back(int x)
97 {
98     int l = 0, r = s - 1, u = 1, m, p;
99     ++n;
100    while (l != r)
101    {
102        m = (l + r) / 2;
103        p = (x <= m);
104        r0[u].push_back(r0[u].back() + p);
105        u *= 2;
106        if (p)
107            r = m;
108        else
109            l = m + 1, ++u;
110    }
111 }
112
113 //doesn't check if empty
114 void pop_back()
115 {
116     int l = 0, r = s - 1, u = 1, m, p, k;
117     --n;
118     while (l != r)
119     {
120         m = (l + r) / 2;
121         k = r0[u].size();
122         p = r0[u][k - 1] - r0[u][k - 2];
123         r0[u].pop_back();
124         u *= 2;
125         if (p)
126             r = m;
127         else
128             l = m + 1, ++u;
129     }
130 }
131
132 //swap arr[i] with arr[i+1], i in [0,n-1)
133 void swap_adj(int i)
134 {
135     int &x = arrCopy[i], &y = arrCopy[i + 1];
136     int l = 0, r = s - 1, u = 1;
137     while (l != r)
138     {
139         int m = (l + r) / 2, p = (x <= m), q = (y <= m);
140         if (p != q)
141         {

```

```

142             r0[u][i + 1] ^= r0[u][i] ^ r0[u][i + 2];
143             break;
144         }
145         u *= 2;
146         if (p)
147             r = m;
148         else
149             l = m + 1, ++u;
150     }
151     swap(x, y);
152 }
153 };

```

5 Dynamic Programming

5.1 Knapsack

```

1
2 vector<vector<ll>>> DP;
3 vector<ll> Weights;
4 vector<ll> Values;
5
6 ll Knapsack(int w, int i)
7 {
8     if (w == 0 or i == -1)
9         return 0;
10    if (DP[w][i] != -1)
11        return DP[w][i];
12    if (Weights[i] > w)
13        return DP[w][i] = Knapsack(w, i - 1);
14    return DP[w][i] = max(Values[i] + Knapsack(w - Weights[i], i - 1),
15                          Knapsack(w, i - 1));
16 }

```

5.2 Matrix Chain Multiplication

```

1
2 vector<vector<ii>>> DP; //Pair value, op result
3 int n; //Size of DP (i.e. i,j<n)
4 ii op(ii a, ii b)
5 {
6     return {a.first + b.first + a.second * b.second, (a.second + b.
7             second) % 100}; //Second part MUST be associative, first part
8             is cost function
9 }
10
11 ii MCM(int i, int j)
12 {
13     if (DP[i][j].first != -1)
14         return DP[i][j];
15     int ans = 1e9; //INF
16     int res;
17     repx(k, i + 1, j)
18     {

```

```

17         ii temp = op(MCM(i, k), MCM(k, j));
18         ans = min(ans, temp.first);
19         res = temp.second;
20     }
21     return DP[i][j] = {ans, res};
22 }
23
24 void fill()
25 {
26     DP.assign(n, vector<ii>(n, {-1, 0}));
27     rep(i, n - 1) { DP[i][i + 1].first = 1; } // Pair op identity, cost
28                                         (cost must be from input)
29 }

```

5.3 Longest Increasing Subsequence

```

1
2 vi L;
3 vi vals;
4
5 int maxl = 1;
6
7 //Bottom up approach O(nlogn)
8 int lis(int n)
9 {
10     L.assign(n, -1);
11     L[0] = vals[0];
12     repx(i, 1, n)
13     {
14         auto it = lower_bound(L.begin(), L.begin() + maxl, vals[i]);
15         if (it == L.begin() + maxl)
16         {
17             L[maxl] = vals[i];
18             maxl++;
19         }
20         else
21             *it = vals[i];
22     }
23     return maxl;
24 }

```

6 Graphs

6.1 Graph Traversal

6.1.1 Breadth First Search

```

1
2 void bfs(graph &g, int start)
3 {
4     int n = g.size();
5     vi visited(n, 1);
6     queue<int> q;
7

```

```

8     q.emplace(start);
9     visited[start] = 0;
10    while (not q.empty())
11    {
12        int u = q.front();
13        q.pop();
14
15        for (int v : g[u])
16        {
17            if (visited[v])
18            {
19                q.emplace(v);
20                visited[v] = 0;
21            }
22        }
23    }
24 }

```

6.1.2 Recursive Depth First Search

```

1 //Recursive (create visited filled with 1s)
2 void dfs_r(graph &g, vi &visited, int u)
3 {
4     cout << u << '\n';
5     visited[u] = 0;
6
7     for (int v : g[u])
8         if (visited[v])
9             dfs_r(g, visited, v);
10 }

```

6.1.3 Iterative Depth First Search

```

1 //Iterative
2 void dfs_i(graph &g, int start)
3 {
4     int n = g.size();
5     vi visited(n, 1);
6     stack<int> s;
7
8     s.emplace(start);
9     visited[start] = 0;
10    while (not s.empty())
11    {
12        int u = s.top();
13        s.pop();
14
15        for (int v : g[u])
16        {
17            if (visited[v])
18            {
19                s.emplace(v);
20                visited[v] = 0;
21            }
22        }
23    }
24 }

```

```

23     }
24 }

```

6.2 Shortest Path Algorithms

6.2.1 Dijkstra

All edges have non-negative values

```

1 //g has vectors of pairs of the form (w, index)
2 int dijkstra(wgraph g, int start, int end)
3 {
4     int n = g.size();
5     vi cost(n, 1e9); //~INT_MAX/2
6     priority_queue<ii, greater<ii>> q;
7
8     q.emplace(0, start);
9     cost[start] = 0;
10    while (not q.empty())
11    {
12        int u = q.top().second, w = q.top().first;
13        q.pop();
14
15        // we skip all nodes in the q that we have discovered before at
16        // a lower cost
17        if (cost[u] < w) continue;
18
19        for (auto v : g[u])
20        {
21            if (cost[v.second] > v.first + w)
22            {
23                cost[v.second] = v.first + w;
24                q.emplace(cost[v.second], v.second);
25            }
26        }
27
28        return cost[end];
29    }

```

6.2.2 Bellman Ford

Edges can be negative, and it detects negative cycles

```

1 bool bellman_ford(wgraph &g, int start)
2 {
3     int n = g.size();
4     vector<int> dist(n, 1e9); //~INT_MAX/2
5     dist[start] = 0;
6     rep(i, n - 1) rep(u, n) for (ii p : g[u])
7     {
8         int v = p.first, w = p.second;
9         dist[v] = min(dist[v], dist[u] + w);
10    }
11

```

```

12    bool hayCicloNegativo = false;
13    rep(u, n) for (ii p : g[u])
14    {
15        int v = p.first, w = p.second;
16        if (dist[v] > dist[u] + w)
17            hayCicloNegativo = true;
18    }
19
20    return hayCicloNegativo;
21 }

```

6.2.3 Floyd Warshall

Shortest path from every node to every other node

```

1
2 /*
3 Floyd Warshall implemenation, note that g is using an adjacency matrix
4 and not an
5 adjacency list
6 */
7 graph floydWarshall (const graph g)
8 {
9     int n = g.size();
10    graph dist(n, vi(n, -1));
11
12    rep(i, n)
13        rep(j, n)
14            dist[i][j] = g[i][j];
15
16    rep(k, n)
17        rep(i, n)
18            rep(j, n)
19                if (dist[i][k] + dist[k][j] < dist[i][j] &&
20                    dist[i][k] != INF &&
21                    dist[k][j] != INF)
22                    dist[i][j] = dist[i][k] + dist[k][j];
23
24    return dist;
25 }

```

6.3 Minimum Spanning Tree (MST)

6.3.1 Kruskal

```

1 struct edge
2 {
3     int u, v;
4     ll w;
5     edge(int u, int v, ll w) : u(u), v(v), w(w) {}
6
7     bool operator<(const edge &o) const
8     {

```

```

9     return w < o.w;
10 }
11 };
12
13 class Kruskal
14 {
15     private:
16         ll sum;
17         vi p, rank;
18
19     public:
20         //Amount of Nodes n, and unordered vector of Edges E
21         Kruskal(int n, vector<edge> E)
22         {
23             sum = 0;
24             p.resize(n);
25             rank.assign(n, 0);
26             rep(i, n) p[i] = i;
27             sort(E.begin(), E.end());
28             for (auto &e : E)
29                 UnionSet(e.u, e.v, e.w);
30         }
31         int findSet(int i)
32         {
33             return (p[i] == i) ? i : (p[i] = findSet(p[i]));
34         }
35         bool isSameSet(int i, int j)
36         {
37             return findSet(i) == findSet(j);
38         }
39         void UnionSet(int i, int j, ll w)
40         {
41             if (not isSameSet(i, j))
42             {
43                 int x = findSet(i), y = findSet(j);
44                 if (rank[x] > rank[y])
45                     p[y] = x;
46                 else
47                     p[x] = y;
48
49                 if (rank[x] == rank[y])
50                     rank[y]++;
51
52                 sum += w;
53             }
54         }
55         ll mst_val()
56         {
57             return sum;
58         }
59 };

```

6.4 Lowest Common Ancestor (LCA)

Supports multiple trees

```

1 class LcaForest
2 {
3     int n;
4     vi parent;
5     vi level;
6     vi root;
7     graph P;
8
9     public:
10        LcaForest(int n)
11        {
12            this->n = n;
13            parent.assign(n, -1);
14            level.assign(n, -1);
15            P.assign(n, vi(lg(n) + 1, -1));
16            root.assign(n, -1);
17        }
18        void addLeaf(int index, int par)
19        {
20            parent[index] = par;
21            level[index] = level[par] + 1;
22            P[index][0] = par;
23            root[index] = root[par];
24            for (int j = 1; (1 << j) < n; ++j)
25            {
26                if (P[index][j - 1] != -1)
27                    P[index][j] = P[P[index][j - 1]][j - 1];
28            }
29        }
30        void addRoot(int index)
31        {
32            parent[index] = index;
33            level[index] = 0;
34            root[index] = index;
35        }
36        int lca(int u, int v)
37        {
38            if (root[u] != root[v] || root[u] == -1)
39                return -1;
40            if (level[u] < level[v])
41                swap(u, v);
42            int dist = level[u] - level[v];
43            while (dist != 0)
44            {
45                int raise = lg(dist);
46                u = P[u][raise];
47                dist -= (1 << raise);
48            }
49            if (u == v)
50                return u;
51            for (int j = lg(n); j >= 0; --j)
52            {
53                if (P[u][j] != -1 && P[u][j] != P[v][j])
54                {
55                    u = P[u][j];

```

```

56         v = P[v][j];
57     }
58 }
59 return parent[u];
60 }
61 };

```

6.5 Max Flow

```

1  class Dinic
2  {
3      struct edge
4      {
5          int to, rev;
6          ll f, cap;
7      };
8
9      vector<vector<edge>> g;
10     vector<ll> dist;
11     vector<int> q, work;
12     int n, sink;
13
14     bool bfs(int start, int finish)
15     {
16         dist.assign(n, -1);
17         dist[start] = 0;
18         int head = 0, tail = 0;
19         q[tail++] = start;
20         while (head < tail)
21         {
22             int u = q[head++];
23             for (const edge &e : g[u])
24             {
25                 int v = e.to;
26                 if (dist[v] == -1 and e.f < e.cap)
27                 {
28                     dist[v] = dist[u] + 1;
29                     q[tail++] = v;
30                 }
31             }
32         }
33         return dist[finish] != -1;
34     }
35
36     ll dfs(int u, ll f)
37     {
38         if (u == sink)
39             return f;
40         for (int &i = work[u]; i < (int)g[u].size(); ++i)
41         {
42             edge &e = g[u][i];
43             int v = e.to;
44             if (e.cap <= e.f or dist[v] != dist[u] + 1)
45                 continue;

```

```

46         ll df = dfs(v, min(f, e.cap - e.f));
47         if (df > 0)
48         {
49             e.f += df;
50             g[v][e.rev].f -= df;
51             return df;
52         }
53     }
54     return 0;
55 }
56
57 public:
58     Dinic(int n)
59     {
60         this->n = n;
61         g.resize(n);
62         dist.resize(n);
63         q.resize(n);
64     }
65
66     void add_edge(int u, int v, ll cap)
67     {
68         edge a = {v, (int)g[v].size(), 0, cap};
69         edge b = {u, (int)g[u].size(), 0, 0}; //Poner cap en vez de 0 si
69             la arista es bidireccional
70         g[u].pb(a);
71         g[v].pb(b);
72     }
73
74     ll max_flow(int source, int dest)
75     {
76         sink = dest;
77         ll ans = 0;
78         while (bfs(source, dest))
79         {
80             work.assign(n, 0);
81             while (ll delta = dfs(source, LLONG_MAX))
82                 ans += delta;
83         }
84         return ans;
85     }
86 };

```

6.6 Others

6.6.1 Diameter of a tree

```

1  graph Tree;
2  vi dist;
3
4
5  // Finds a diameter node
6  int bfs1()
7  {
8      int n = Tree.size();

```



```

9   queue<int> q;
10
11   q.emplace(0);
12   dist[0] = 0;
13   int u;
14   while (not q.empty())
15   {
16       u = q.front();
17       q.pop();
18
19       for (int v : Tree[u])
20       {
21           if (dist[v] == -1)
22           {
23               q.emplace(v);
24               dist[v] = dist[u] + 1;
25           }
26       }
27   }
28   return u;
29 }
30
31 // Fills the distances from one diameter node and finds another diameter
   node
32 int bfs2()
33 {
34     int n = Tree.size();
35     vi visited(n, 1);
36     queue<int> q;
37     int start = bfs1();
38     q.emplace(start);
39     visited[start] = 0;
40     int u;
41     while (not q.empty())
42     {
43         u = q.front();
44         q.pop();
45
46         for (int v : Tree[u])
47         {
48             if (visited[v])
49             {
50                 q.emplace(v);
51                 visited[v] = 0;
52                 dist[v] = max(dist[v], dist[u] + 1);
53             }
54         }
55     }
56     return u;
57 }
58
59 // Finds the diameter
60 int bfs3()
61 {
62     int n = Tree.size();

```

```

63     vi visited(n, 1);
64     queue<int> q;
65     int start = bfs2();
66     q.emplace(start);
67     visited[start] = 0;
68     int u;
69     while (not q.empty())
70     {
71         u = q.front();
72         q.pop();
73
74         for (int v : Tree[u])
75         {
76             if (visited[v])
77             {
78                 q.emplace(v);
79                 visited[v] = 0;
80                 dist[v] = max(dist[v], dist[u] + 1);
81             }
82         }
83     }
84     return dist[u];
85 }

```

7 Mathematics

7.1 Useful Data

n	Primes less than n	Maximal Prime Gap	$\max_{0 < i < n}(d(i))$
1e2	25	8	12
1e3	168	20	32
1e4	1229	36	64
1e5	9592	72	128
1e6	78.498	114	240
1e7	664.579	154	448
1e8	5.761.455	220	768
1e9	50.487.534	282	1344

7.2 Modular Arithmetic

7.2.1 Chinese Remainder Theorem

```

1
2 11 inline mod(11 x, 11 m) { return ((x % m) < 0) ? x + m : x; }
3 11 inline mul(11 x, 11 y, 11 m) { return (x * y) % m; }
4 11 inline add(11 x, 11 y, 11 m) { return (x + y) % m; }
5
6 // extended euclidean algorithm
7 // finds g, x, y such that
8 //   a * x + b * y = g = GCD(a,b)
9 11 gcdext(11 a, 11 b, 11 &x, 11 &y)

```

```

10 {
11     ll r2, x2, y2, r1, x1, y1, r0, x0, y0, q;
12     r2 = a, x2 = 1, y2 = 0;
13     r1 = b, x1 = 0, y1 = 1;
14     while (r1)
15     {
16         q = r2 / r1;
17         r0 = r2 % r1;
18         x0 = x2 - q * x1;
19         y0 = y2 - q * y1;
20         r2 = r1, x2 = x1, y2 = y1;
21         r1 = r0, x1 = x0, y1 = y0;
22     }
23     ll g = r2;
24     x = x2, y = y2;
25     if (g < 0)
26         g = -g, x = -x, y = -y; // make sure g > 0
27     // for debugging (in case you think you might have bugs)
28     // assert (g == a * x + b * y);
29     // assert (g == __gcd(abs(a),abs(b)));
30     return g;
31 }
32
33 // =====
34 // CRT for a system of 2 modular linear equations
35 // =====
36 // We want to find X such that:
37 // 1) x = r1 (mod m1)
38 // 2) x = r2 (mod m2)
39 // The solution is given by:
40 // sol = r1 + m1 * (r2-r1)/g * x' (mod LCM(m1,m2))
41 // where x' comes from
42 // m1 * x' + m2 * y' = g = GCD(m1,m2)
43 // where x' and y' are the values found by extended euclidean
44 // algorithm (gcdext)
45 // Useful references:
46 // https://codeforces.com/blog/entry/61290
47 // https://forthright48.com/chinese-remainder-theorem-part-1-coprime-
48 // moduli
49 // https://forthright48.com/chinese-remainder-theorem-part-2-non-
50 // coprime-moduli
51 // ** Note: this solution works if lcm(m1,m2) fits in a long long (64
52 // bits)
53 pair<ll, ll> CRT(ll r1, ll m1, ll r2, ll m2)
54 {
55     ll g, x, y;
56     g = gcdext(m1, m2, x, y);
57     if ((r1 - r2) % g != 0)
58         return {-1, -1}; // no solution
59     ll z = m2 / g;
60     ll lcm = m1 * z;
61     ll sol = add(mod(r1, lcm), m1 * mul(mod(x, z), mod((r2 - r1) / g, z), z), lcm);
62     // for debugging (in case you think you might have bugs)
63     // assert (0 <= sol and sol < lcm);

```

```

60 // assert (sol % m1 == r1 % m1);
61 // assert (sol % m2 == r2 % m2);
62 return {sol, lcm}; // solution + lcm(m1,m2)
63 }
64
65 // =====
66 // CRT for a system of N modular linear equations
67 // =====
68 // Args:
69 // r = array of remainders
70 // m = array of modules
71 // n = length of both arrays
72 // Output:
73 // a pair {X, lcm} where X is the solution of the sytemm
74 // X = r[i] (mod m[i]) for i = 0 ... n-1
75 // and lcm = LCM(m[0], m[1], ..., m[n-1])
76 // if there is no solution, the output is {-1, -1}
77 // ** Note: this solution works if LCM(m[0],...,m[n-1]) fits in a long
78 // long (64 bits)
79 pair<ll, ll> CRT(ll *r, ll *m, int n)
80 {
81     ll r1 = r[0], m1 = m[0];
82     rep(x, 1, n)
83     {
84         ll r2 = r[i], m2 = m[i];
85         ll g, x, y;
86         g = gcdext(m1, m2, x, y);
87         if ((r1 - r2) % g != 0)
88             return {-1, -1}; // no solution
89         ll z = m2 / g;
90         ll lcm = m1 * z;
91         ll sol = add(mod(r1, lcm), m1 * mul(mod(x, z), mod((r2 - r1) / g, z), z), lcm);
92         r1 = sol;
93         m1 = lcm;
94     }
95     // for debugging (in case you think you might have bugs)
96     // assert (0 <= r1 and r1 < m1);
97     // rep(i, n) assert (r1 % m[i] == r[i]);
98     return {r1, m1};
99 }

```

7.2.2 Binomial Coefficients mod m

```

1 #include "../CRT/CRT.cpp"
2 #include "../primalityChecks/millerRabin/millerRabin.cpp"
3 #include "../primalityChecks/sieveEratosthenes/sieve.cpp"
4
5 // Modular computation of nCr using lucas theorem, granville theorem and
6 // CRT
7
8 ll num; //Set num to the corresponding mod for the
9         nCr calculations
10 umap<ll, int> MOD; //MOD[P]=V_p(mod)
11 umap<ll, vector<ll>> FMOD; //n! mod p if MOD[p]=1 else the product of

```

```

10     all i mod P^MOD[P], where 1<=i<=n and (i,p)=1
    umap<ll, vector<ll>> invFMOD; //the inverse of FMOD[n] in the
        corresponding MOD
11
12 void preCompute()
13 {
14     // Factor mod->MOD
15     vi primes = sieve(num);
16     ll m = num;
17     for (auto p : primes)
18     {
19         if (p * p > m)
20             break;
21         while (m % p == 0)
22         {
23             MOD[p]++;
24             if ((m /= p) == 1)
25                 goto next;
26         }
27     }
28     if (m > 1)
29         MOD[m] = 1;
30 next:
31     // Compute FMOD and invFMOD
32     for (auto p : MOD)
33     {
34         int m = pow(p.first, p.second); //p^V_p(n)
35         FMOD[p.first].assign(m, 1);
36         invFMOD[p.first].assign(m, 1);
37         repx(i, 2, FMOD[p.first].size())
38         {
39             if (i % p.first == 0 and p.second > 1)
40                 FMOD[p.first][i] = FMOD[p.first][i - 1];
41             else
42                 FMOD[p.first][i] = mul(FMOD[p.first][i - 1], i, FMOD[p.
43                     first].size());
44
45             //Compute using Euler's theorem i.e. a^phi(m)=1 mod m with (
46                 a,m)=1
47             invFMOD[p.first][i] = fastPow(FMOD[p.first][i], m / p.first
48                 * (p.first - 1) - 1, m);
49         }
50     }
51
52     // Compute nCr using Granville's theorem (prime powers)
53     // Auxiliary functions
54
55     // V_p(n!) using Legendre's theorem
56     int V(ll n, int p)
57     {
58         int e = 0;
59         while ((n /= p) > 0)
60             e += n;
61         return e;

```

```

60     }
61
62     //
63     ll f(ll n, ll p)
64     {
65         ll m = pow(p, MOD[p]);
66         int e = n / m;
67         return mul(fastPow(FMOD[p][m - 1], e, m), FMOD[p][n % m], m);
68     }
69     ll F(ll n, ll p)
70     {
71         ll m = pow(p, MOD[p]);
72         ll ans = 1;
73         do
74         {
75             ans = mul(ans, f(n, p), m);
76         } while ((n /= p) > 0);
77         return ans;
78     }
79     // Granville theorem
80     ll granville(ll n, ll r, int p)
81     {
82         int e = V(n, p) - V(n - r, p) - V(r, p);
83         ll m = pow(p, MOD[p]);
84         if (e >= MOD[p])
85             return 0;
86         ll ans = fastPow(p, e, m);
87         ans = mul(ans, F(n, p), m);
88         ans = mul(ans, fastPow(F(r, p), pow(p, MOD[p] - 1) * (p - 1) - 1, m)
89             , m);
90         ans = mul(ans, fastPow(F(n - r, p), pow(p, MOD[p] - 1) * (p - 1) -
91             1, m), m);
92         return ans;
93     }
94
95     // Compute nCr using Lucas theorem (primes)
96     ll lucas(ll n, ll r, int p)
97     {
98         // Trivial cases
99         if (r > n or r < 0)
100             return 0;
101         if (r == 0 or n == r)
102             return 1;
103         if (r == 1 or r == n - 1)
104             return n % p;
105         // Base case
106         if (n < p and r < p)
107         {
108             ll ans = mul(invFMOD[p][r], invFMOD[p][n - r], p); // 1/(r!(n-r
109                 !) mod p
110             ans = mul(ans, FMOD[p][n], p); // n!/(r!(n-r
111                 !)) mod p
112             return ans;
113         }
114         ll ans = lucas(n / p, r / p, p); //Recursion

```

```

111     ans = mul(ans, lucas(n % p, r % p, p), p); //False recursion
112     return ans;
113 }
114
115 // Given the prime decomposition of mod;
116 ll nCr(ll n, ll r)
117 {
118     // Trivial cases
119     if (n < r or r < 0)
120         return 0;
121     if (r == 0 or r == n)
122         return 1;
123     if (r == 1 or r == n - 1)
124         return (n % num);
125     // Non-trivial cases
126     ll ans = 0;
127     ll mod = 1;
128     for (auto p : MOD)
129     {
130         ll temp = pow(p.first, p.second);
131         if (p.second > 1)
132         {
133             ans = CRT(ans, mod, granville(n, r, p.first), temp).first;
134         }
135         else
136         {
137             ans = CRT(ans, mod, lucas(n, r, p.first), temp).first;
138         }
139         mod *= temp;
140     }
141     return ans;
142 }

```

7.3 Primality Checks

7.3.1 Miller Rabin

```

1
2 ll mulmod(ull a, ull b, ull c)
3 {
4     ull x = 0, y = a % c;
5     while (b)
6     {
7         if (b & 1)
8             x = (x + y) % c;
9         y = (y << 1) % c;
10        b >>= 1;
11    }
12    return x % c;
13 }
14
15 ll fastPow(ll x, ll n, ll MOD)
16 {
17     ll ret = 1;
18     while (n)

```

```

19 {
20     if (n & 1)
21         ret = mulmod(ret, x, MOD);
22     x = mulmod(x, x, MOD);
23     n >>= 1;
24 }
25 return ret;
26 }
27
28 bool isPrime(ll n)
29 {
30     vi a = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
31
32     if (binary_search(a.begin(), a.end(), n))
33         return true;
34
35     if ((n & 1) == 0)
36         return false;
37
38     int s = 0;
39     for (ll m = n - 1; !(m & 1); ++s, m >>= 1)
40         ;
41
42     int d = (n - 1) / (1 << s);
43
44     for (int i = 0; i < 7; i++)
45     {
46         ll fp = fastPow(a[i], d, n);
47         bool comp = (fp != 1);
48         if (comp)
49             for (int j = 0; j < s; j++)
50             {
51                 if (fp == n - 1)
52                 {
53                     comp = false;
54                     break;
55                 }
56
57                 fp = mulmod(fp, fp, n);
58             }
59         if (comp)
60             return false;
61     }
62     return true;
63 }

```

7.3.2 Sieve of Eratosthenes

```

1
2 // O(n log log n)
3 vi sieve(int n)
4 {
5     vi primes;
6
7     vector<bool> is_prime(n + 1, true);

```

```

8   int limit = (int)floor(sqrt(n));
9   repx(i, 2, limit + 1) if (is_prime[i]) for (int j = i * i; j <= n; j
      += i)
10      is_prime[j] = false;
11
12   repx(i, 2, n + 1) if (is_prime[i]) primes.eb(i);
13
14   return primes;
15 }

```

7.3.3 trialDivision

```

1  // O(sqrt(n)/log(sqrt(n))+log(n))
2  vi trialDivision(int n, vi &primes)
3  {
4      vi factors;
5      for (auto p : primes)
6      {
7          if (p * p > n)
8              break;
9          while (n % p == 0)
10             {
11                 primes.pb(p);
12                 if ((n /= p) == 1)
13                     return factors;
14             }
15     }
16     if (n > 1)
17         factors.pb(n);
18
19     return factors;
20 }

```

7.4 Others

7.4.1 Polynomials

```

1  template <class T>
2  class Pol
3  {
4  private:
5      vector<T> coefs;
6      int n;
7
8  public:
9      Pol(vector<T> coefs) : coefs(coefs)
10     {
11         this->n = coefs.size() - 1;
12     }
13
14     Pol<T> operator+(const Pol<T> &o)
15     {
16         vector<T> n_coefs;

```

```

18     if (n > o.n)
19     {
20         n_coefs = coefs;
21         rep(i, o.n + 1)
22         {
23             n_coefs[i] += o.coefs[i];
24         }
25     }
26     else
27     {
28         n_coefs = o.coefs;
29         rep(i, n + 1)
30         {
31             n_coefs[i] += coefs[i];
32         }
33     }
34     return Pol(n_coefs);
35 }
36
37 Pol<T> operator-(const Pol<T> &o)
38 {
39     vector<T> n_coefs;
40     if (n > o.n)
41     {
42         n_coefs = coefs;
43         rep(i, o.n + 1)
44         {
45             n_coefs[i] -= o.coefs[i];
46         }
47     }
48     else
49     {
50         n_coefs = o.coefs;
51         rep(i, n + 1)
52         {
53             n_coefs[i] *= -1;
54             n_coefs[i] += coefs[i];
55         }
56     }
57     return Pol(n_coefs);
58 }
59
60 Pol<T> operator*(const Pol<T> &o) //Use Fast Fourier Transform when
61     we implement it
62 {
63     vector<T> n_coefs(n + o.n + 1);
64     rep(i, n + 1)
65     {
66         rep(j, o.n + 1)
67         {
68             n_coefs[i + j] += coefs[i] * o.coefs[j];
69         }
70     }
71     return Pol(n_coefs);

```

```

72 |
73 | Pol<T> operator*(const T &o)
74 | {
75 |     vector<T> n_cofs = cofs;
76 |     for (auto &cof : n_cofs)
77 |     {
78 |         cof *= o;
79 |     }
80 |     return Pol(n_cofs);
81 | }
82 |
83 | double operator()(double x)
84 | {
85 |     double ans = 0;
86 |     double temp = 1;
87 |     for (auto cof : cofs)
88 |     {
89 |         ans += (double)cof * temp;
90 |         temp *= x;
91 |     }
92 |     return ans;
93 | }
94 |
95 | Pol<T> integrate()
96 | {
97 |     vector<T> n_cofs(n + 2);
98 |     repx(i, 1, n_cofs.size())
99 |     {
100 |         n_cofs[i] = cofs[i - 1] / T(i);
101 |     }
102 |     return Pol<T>(n_cofs);
103 | }
104 |
105 | double integrate(T a, T b)
106 | {
107 |     Pol<T> temp = integrate();
108 |     return temp(b) - temp(a);
109 | }
110 |
111 | friend ostream &operator<<(ostream &str, const Pol &a);
112 | };
113 |
114 | ostream &operator<<(ostream &strm, const Pol<double> &a)
115 | {
116 |     bool flag = false;
117 |     rep(i, a.n + 1)
118 |     {
119 |         if (a.cofs[i] == 0)
120 |             continue;
121 |
122 |         if (flag)
123 |             if (a.cofs[i] > 0)
124 |                 strm << " + ";
125 |             else
126 |                 strm << " - ";

```

```

127 |     else
128 |         flag = true;
129 |     if (i > 1)
130 |     {
131 |         if (abs(a.cofs[i]) != 1)
132 |             strm << abs(a.cofs[i]);
133 |         strm << "x^" << i;
134 |     }
135 |     else if (i == 1)
136 |     {
137 |         if (abs(a.cofs[i]) != 1)
138 |             strm << abs(a.cofs[i]);
139 |         strm << "x";
140 |     }
141 |     else
142 |     {
143 |         strm << a.cofs[i];
144 |     }
145 | }
146 | return strm;
147 | }

```

7.4.2 Factorial Factorization

```

1 |
2 | // O(n)
3 | umap<ll, int> factorialFactorization(int n, vi &primes)
4 | {
5 |     umap<ll, int> p2e;
6 |     for (auto p : primes)
7 |     {
8 |         if (p > n)
9 |             break;
10 |         int e = 0;
11 |         ll tmp = n;
12 |         while ((tmp /= p) > 0)
13 |             e += tmp;
14 |         if (e > 0)
15 |             p2e[p] = e;
16 |     }
17 |     return p2e;
18 | }

```

8 Geometry

8.1 Vectors/Points

```

1 |
2 | const double PI = acos(-1);
3 |
4 | struct vector2D
5 | {
6 |     double x, y;
7 |

```

```

8   vector2D &operator+=(const vector2D &o)
9   {
10      this->x += o.x;
11      this->y += o.y;
12      return *this;
13  }
14
15  vector2D &operator-=(const vector2D &o)
16  {
17      this->x -= o.x;
18      this->y -= o.y;
19      return *this;
20  }
21
22  vector2D operator+(const vector2D &o)
23  {
24      return {x + o.x, y + o.y};
25  }
26
27  vector2D operator-(const vector2D &o)
28  {
29      return {x - o.x, y - o.y};
30  }
31
32  vector2D operator*(const double &o)
33  {
34      return {x * o, y * o};
35  }
36
37  bool operator==(const vector2D &o)
38  {
39      return x == o.x and y == o.y;
40  }
41
42  double norm2() { return x * x + y * y; }
43  double norm() { return sqrt(norm2()); }
44  double dot(const vector2D &o) { return x * o.x + y * o.y; }
45  double cross(const vector2D &o) { return x * o.y - y * o.x; }
46  double angle()
47  {
48      double angle = atan2(y, x);
49      if (angle < 0)
50          angle += 2 * PI;
51      return angle;
52  }
53
54  vector2D Unit()
55  {
56      return {x / norm(), y / norm()};
57  }
58 };
59
60 /* ===== */
61 /* Cross Product -> orientation of vector2D with respect to ray */
62 /* ===== */

```

```

63 // cross product (b - a) x (c - a)
64 ll cross(vector2D &a, vector2D &b, vector2D &c)
65 {
66     ll dx0 = b.x - a.x, dy0 = b.y - a.y;
67     ll dx1 = c.x - a.x, dy1 = c.y - a.y;
68     return dx0 * dy1 - dx1 * dy0;
69     // return (b - a).cross(c - a); // alternatively, using struct
        function
70 }
71
72 // calculates the cross product (b - a) x (c - a)
73 // and returns orientation:
74 // LEFT (1):      c is to the left of ray (a -> b)
75 // RIGHT (-1):    c is to the right of ray (a -> b)
76 // COLLINEAR (0): c is collinear to ray (a -> b)
77 // inspired by: https://www.geeksforgeeks.org/orientation-3-ordered-
        points/
78 int orientation(vector2D &a, vector2D &b, vector2D &c)
79 {
80     ll tmp = cross(a, b, c);
81     return tmp < 0 ? -1 : tmp == 0 ? 0 : 1; // sign
82 }
83
84 /* ===== */
85 /* Check if a segment is below another segment (wrt a ray) */
86 /* ===== */
87 // i.e: check if a segment is intersected by the ray first
88 // Assumptions:
89 // 1) for each segment:
90 // p1 should be LEFT (or COLLINEAR) and p2 should be RIGHT (or
        COLLINEAR) wrt ray
91 // 2) segments do not intersect each other
92 // 3) segments are not collinear to the ray
93 // 4) the ray intersects all segments
94 struct Segment
95 {
96     vector2D p1, p2;
97 };
98 #define MAXN (int)1e6 //Example
99 Segment segments[MAXN]; // array of line segments
100 bool is_si_below_sj(int i, int j)
101 { // custom comparator based on cross product
102     Segment &si = segments[i];
103     Segment &sj = segments[j];
104     return (si.p1.x >= sj.p1.x) ? cross(si.p1, sj.p2, sj.p1) > 0 : cross
        (sj.p1, si.p1, si.p2) > 0;
105 }
106 // this can be used to keep a set of segments ordered by order of
        intersection
107 // by the ray, for example, active segments during a SWEEP LINE
108 set<int, bool> (*)(int, int)> active_segments(is_si_below_sj); // ordered
        set
109
110 /* ===== */
111 /* Rectangle Intersection */

```

```

112 /* ===== */
113 bool do_rectangles_intersect(vector2D &d11, vector2D &ur1, vector2D &dl2
    , vector2D &ur2)
114 {
115     return max(d11.x, dl2.x) <= min(ur1.x, ur2.x) && max(d11.y, dl2.y)
        <= min(ur1.y, ur2.y);
116 }
117
118 /* ===== */
119 /* Line Segment Intersection */
120 /* ===== */
121 // returns whether segments p1q1 and p2q2 intersect, inspired by:
122 // https://www.geeksforgeeks.org/check-if-two-given-line-segments-
    intersect/
123 bool do_segments_intersect(vector2D &p1, vector2D &q1, vector2D &p2,
    vector2D &q2)
124 {
125     int o11 = orientation(p1, q1, p2);
126     int o12 = orientation(p1, q1, q2);
127     int o21 = orientation(p2, q2, p1);
128     int o22 = orientation(p2, q2, q1);
129     if (o11 != o12 and o21 != o22) // general case -> non-collinear
        intersection
        return true;
130
131     if (o11 == o12 and o11 == 0)
132     { // particular case -> segments are collinear
133         vector2D dl1 = {min(p1.x, q1.x), min(p1.y, q1.y)};
134         vector2D ur1 = {max(p1.x, q1.x), max(p1.y, q1.y)};
135         vector2D dl2 = {min(p2.x, q2.x), min(p2.y, q2.y)};
136         vector2D ur2 = {max(p2.x, q2.x), max(p2.y, q2.y)};
137         return do_rectangles_intersect(dl1, ur1, dl2, ur2);
138     }
139     return false;
140 }
141
142 /* ===== */
143 /* Circle Intersection */
144 /* ===== */
145 struct Circle
146 {
147     double x, y, r;
148 };
149 bool is_fully_outside(double r1, double r2, double d_sqr)
150 {
151     double tmp = r1 + r2;
152     return d_sqr > tmp * tmp;
153 }
154 bool is_fully_inside(double r1, double r2, double d_sqr)
155 {
156     if (r1 > r2)
157         return false;
158     double tmp = r2 - r1;
159     return d_sqr < tmp * tmp;
160 }
161 bool do_circles_intersect(Circle &c1, Circle &c2)

```

```

162 {
163     double dx = c1.x - c2.x;
164     double dy = c1.y - c2.y;
165     double d_sqr = dx * dx + dy * dy;
166     if (is_fully_inside(c1.r, c2.r, d_sqr))
167         return false;
168     if (is_fully_outside(c1.r, c2.r, d_sqr))
169         return false;
170     if (is_fully_outside(c1.r, c2.r, d_sqr))
171         return false;
172     return true;
173 }
174
175 /* ===== */
176 /* vector2D - Line distance */
177 /* ===== */
178 // get distance between p and projection of p on line <- a - b ->
179 double point_line_dist(vector2D &p, vector2D &a, vector2D &b)
180 {
181     vector2D d = b - a;
182     double t = d.dot(p - a) / d.norm2();
183     return (a + d * t - p).norm();
184 }
185
186 /* ===== */
187 /* vector2D - Segment distance */
188 /* ===== */
189 // get distance between p and truncated projection of p on segment a ->
    b
190 double point_segment_dist(vector2D &p, vector2D &a, vector2D &b)
191 {
192     if (a == b)
193         return (p - a).norm(); // segment is a single vector2D
194     vector2D d = b - a; // direction
195     double t = d.dot(p - a) / d.norm2();
196     if (t <= 0)
197         return (p - a).norm(); // truncate left
198     if (t >= 1)
199         return (p - b).norm(); // truncate right
200     return (a + d * t - p).norm();
201 }
202
203 /* ===== */
204 /* Straight Line Hashing (integer coords) */
205 /* ===== */
206 // task: given 2 points p1, p2 with integer coordinates, output a unique
207 // representation {a,b,c} such that a*x + b*y + c = 0 is the equation
208 // of the straight line defined by p1, p2. This representation must be
209 // unique for each straight line, no matter which p1 and p2 are sampled.
210 struct Line
211 {
212     int a, b, c;
213 };
214 int gcd(int a, int b)
215 { // greatest common divisor

```



```

216     a = abs(a);
217     b = abs(b);
218     while (b)
219     {
220         int c = a;
221         a = b;
222         b = c % b;
223     }
224     return a;
225 }
226 Line getLine(vector2D p1, vector2D p2)
227 {
228     int a = p1.y - p2.y;
229     int b = p2.x - p1.x;
230     int c = p1.x * (p2.y - p1.y) - p1.y * (p2.x - p1.x);
231     int sgn = (a < 0 || (a == 0 && b < 0)) ? -1 : 1;
232     int f = gcd(a, gcd(b, c)) * sgn;
233     a /= f;
234     b /= f;
235     c /= f;
236     return {a, b, c};
237 }

```

8.2 Calculate Areas

8.2.1 Integration via Simpson's Method

```

1 //0(Evaluate f)=g(f)
2 //Numerical Integration of f in interval [a,b]
3 double simpsons_rule(function<double(double)> f, double a, double b)
4 {
5     double c = (a + b) / 2;
6     double h3 = abs(b - a) / 6;
7     return h3 * (f(a) + 4 * f(c) + f(b));
8 }
9
10 //0(n g(f))
11 //Integrate f between a and b, using intervals of length (b-a)/n
12 double simpsons_rule(function<double(double)> f, double a, double b, int
13     n)
14 {
15     //n sets the precision for the result
16     double ans = 0;
17     double step = 0, h = (b - a) / n;
18     rep(i, n)
19     {
20         ans += simpsons_rule(f, step, step + h);
21         step += h;
22     }
23     return ans;
24 }

```

8.2.2 Green's Theorem

```

1 // Line integrals for calculating areas with green's theorem
2 struct Point { double x, y; };
3
4 double arc_integral(double x, double r, double a, double b)
5 {
6     return x * r * (sin(b) - sin(a)) + r * r * 0.5 * (0.5 * (sin(2 * b)
7         - sin(2 * a)) + b - a);
8 }
9 double segment_integral(Point &a, Point &b)
10 {
11     return 0.5 * (a.x + b.x) * (b.y - a.y);
12 }

```

8.3 Pick's Theorem

Given a simple polygon (no self intersections) in a lattice such that all vertices are grid points. Pick's theorem relates the Area A , points inside of the polygon i and the points of the border of the polygon b , in the following way:

$$A = i + \frac{b}{2} - 1$$

9 Strings

9.1 KMP

```

1
2 vi prefix(string &S)
3 {
4     vector<int> p(S.size());
5     p[0] = 0;
6     for (int i = 1; i < S.size(); ++i)
7     {
8         p[i] = p[i - 1];
9         while (p[i] > 0 && S[p[i]] != S[i])
10             p[i] = p[p[i] - 1];
11         if (S[p[i]] == S[i])
12             p[i]++;
13     }
14     return p;
15 }
16
17 vi KMP(string &P, string &S)
18 {
19     vector<int> pi = prefix(P);
20     vi matches;
21     int n = S.length(), m = P.length();
22     int j = 0, ans = 0;
23     for (int i = 0; i < n; ++i)
24     {
25         while (j > 0 && S[i] != P[j])
26             j = pi[j - 1];

```

```

27     if (S[i] == P[j])
28         ++j;
29
30     if (j == P.length())
31     {
32         /* This is where KMP found a match
33          * we can calculate its position on S by using i - m + 1
34          * or we can simply count it
35          */
36         ans += 1; // count the number of matches
37         matches.pb(i - m + 1); // store the position of those
38                             matches
39         // return; we can return on the first match if needed
40         // this must stay the same
41         j = pi[j - 1];
42     }
43     return matches; // can be modified to return number of matches or
44                     // location

```

9.2 Rolling Hashing

```

1  const int MAXLEN = 1e6;
2
3  class rollingHashing
4  {
5
6      static const ull base = 127;
7      static const vector<ull> primes;
8      static vector<vector<ull>> POW;
9
10     static ull add(ull x, ull y, int a) { return (x + y) % primes[a]; }
11     static ull mul(ull x, ull y, int a) { return (x * y) % primes[a]; }
12
13     static void init(int a)
14     {
15         if (POW.size() <= a + 1)
16         {
17             POW.pb(MAXLEN, 1);
18         }
19         repx(i, 1, MAXLEN) POW[a][i] = mul(POW[a][i], base, a);
20     }
21
22     static void init()
23     {
24         rep(i, primes.size()) init(i);
25     }
26
27     vector<vector<ull>> h;
28     int len;
29     rollingHashing(string &s)
30     {
31         len = s.size();
32         h.assign(primes.size(), vector<ull>(len, 0));

```

```

33         rep(a, primes.size())
34         {
35             h[a][0] = s[0] - 'a'; //Assuming alphabetic alphabet
36             repx(i, 1, len) h[a][i] = add(s[i] - 'a', mul(h[a][i - 1],
37                 base, a), a);
38         }
39
40     ull hash(int i, int j, int a) //Inclusive-Exclusive [i,i)?
41     {
42         if (i == 0)
43             return h[a][j - 1];
44         return add(h[a][j - 1], primes[a] - mul(h[a][i - 1], POW[a][j -
45             i], a), a);
46     }
47
48     ull hash(int i, int j) //Supports at most two primes
49     {
50         return hash(i, j, 1) << 32 | hash(i, j, 0); //Using that 1e18<
51             __LONG_LONG_MAX__
52     }
53
54     ull hash() { return hash(0, len); } //Also supports at most two
55     primes

```

9.3 Trie

```

1  /* Implementation from: https://pastebin.com/fyqsH65k */
2
3  struct TrieNode
4  {
5      int leaf; // number of words that end on a TrieNode (allows for
6          duplicate words)
7      int height; // height of a TrieNode, root starts at height = 1, can
8          be changed with the default value of constructor
9      // number of words that pass through this node,
10     // ask root node for this count to find the number of entries on the
11     // whole Trie
12     // all nodes have 1 as they count the words than end on themselves (
13     // ie leaf nodes count themselves)
14
15     int count;
16     TrieNode *parent; // pointer to parent TrieNode, used on erasing
17     // entries
18     map<char, TrieNode *> child;
19     TrieNode(TrieNode *parent = NULL, int height = 1):
20         parent(parent),
21         leaf(0),
22         height(height),
23         count(0), // change to -1 if leaf nodes are to have count 0
24             instead of 1
25     {
26         child()

```

```

19     {}
20 };
21
22 /**
23  * Complexity:  $O(|key| * \log(k))$ 
24  */
25 TrieNode *trie_find(TrieNode *root, const string &str)
26 {
27     TrieNode *pNode = root;
28     for (string::const_iterator key = str.begin(); key != str.end(); key
29         ++){
30         if (pNode->child.find(*key) == pNode->child.end())
31             return NULL;
32         pNode = pNode->child[*key];
33     }
34     return (pNode->leaf) ? pNode : NULL; // returns only whole word
35     // return pNode; // allows to search for a suffix
36 }
37
38 /**
39  * Complexity:  $O(|key| * \log(k))$ 
40  */
41 void trie_insert(TrieNode *root, const string &str)
42 {
43     TrieNode *pNode = root;
44     root -> count += 1;
45     for (string::const_iterator key = str.begin(); key != str.end(); key
46         ++){
47         if (pNode->child.find(*key) == pNode->child.end())
48             pNode->child[*key] = new TrieNode(pNode, pNode->height + 1);
49         pNode = pNode->child[*key];
50         pNode -> count += 1;
51     }
52     pNode->leaf += 1;
53 }
54
55 /**
56  * Complexity:  $O(|key| * \log(k))$ 
57  */
58 void trie_erase(TrieNode *root, const string &str)
59 {
60     TrieNode *pNode = root;
61     string::const_iterator key = str.begin();
62     for (; key != str.end(); key++){
63         if (pNode->child.find(*key) == pNode->child.end())
64             return;
65         pNode = pNode->child[*key];
66     }
67     pNode->leaf -= 1;
68     pNode->count -= 1;
69     while (pNode->parent != NULL)
70     {

```

```

72         if (pNode->child.size() > 0 || pNode->leaf)
73             break;
74         pNode = pNode->parent, key--;
75         pNode->child.erase(*key);
76         pNode->count -= 1;
77     }
78 }

```

9.4 Suffix Tree

```

1
2 struct Node{
3     //map<int,int> children;
4     vector<int> children;
5     int suffix_link;
6     int start;
7     int end;
8
9     Node(int start, int end):start(start),end(end){
10         children.resize(27,-1);
11         suffix_link = 0;
12     }
13     inline bool has_child(int i){
14         //return children.find(i) != children.end();
15         return children[i] != -1;
16     }
17 };
18
19 struct SuffixTree{
20     int size;
21     int i;
22     vector<int> suffix_array;
23     vector<Node> tree;
24     inline int length(int index){
25         if (tree[index].end == -1)
26             return i - tree[index].start + 1;
27         return tree[index].end - tree[index].start + 1;
28     }
29     //se puede usar string& s
30     SuffixTree(vector<int>& s){
31         size = s.size();
32         tree.emplace_back(-1,-1);
33         int remaining_suffix = 0;
34         int active_node = 0;
35         int active_edge = -1;
36         int active_length = 0;
37         for(i = 0; i < size; ++i){
38             int last_new = -1;
39             remaining_suffix++;
40             while(remaining_suffix > 0){
41                 if(active_length == 0)
42                     active_edge = i;
43                 if(!tree[active_node].has_child(s[active_edge])){
44                     tree[active_node].children[s[active_edge]] = tree.

```

```

45         tree.emplace_back(i,-1);
46         if(last_new != -1){
47             tree[last_new].suffix_link = active_node;
48             last_new = -1;
49         }
50     }
51     else{
52         int next = tree[active_node].children[s[active_edge
53             ]];
54         if(active_length >= length(next)){
55             active_edge += length(next);
56             active_length -= length(next);
57             active_node = next;
58             continue;
59         }
60         if(s[tree[next].start + active_length] == s[i]){
61             if(last_new != -1 and active_node != 0){
62                 tree[last_new].suffix_link = active_node;
63             }
64             active_length++;
65             break;
66         }
67         int split_end = tree[next].start + active_length -
68             1;
69         int split = tree.size();
70         tree.emplace_back(tree[next].start,split_end);
71         tree[active_node].children[s[active_edge]] = split;
72         int new_leaf = tree.size();
73         tree.emplace_back(i,-1);
74         tree[split].children[s[i]] = new_leaf;
75         tree[next].start += active_length;
76         tree[split].children[s[tree[next].start]] = next;
77         if(last_new != -1){
78             tree[last_new].suffix_link = split;
79         }
80         last_new = split;
81     }
82     remaining_suffix--;
83     if(active_node == 0 and active_length > 0){
84         active_length--;
85         active_edge = i - remaining_suffix + 1;
86     }
87     else if(active_node != 0){
88         active_node = tree[active_node].suffix_link;
89     }
90 }
91 i = size - 1;
92 }
93 vector<int> lcp;
94 //last for lcp
95 void dfs(int node, int& index, int depth,int min_depth){
96     if(tree[node].end == -1 and node != 0){
97         suffix_array[index] = size - depth;
98         if(index != 0){

```

```

98             lcp[index-1] = min_depth;
99         }
100         index++;
101     }
102     for(auto it: tree[node].children){
103         //if(i.second != -1){
104             //    dfs(i.second,index,depth + length(i.second));
105             //    min_depth = depth;
106             //}
107         if(it != -1){
108             dfs(it,index,depth + length(it),min_depth);
109             min_depth = depth;
110         }
111     }
112 }
113 void build_suffix_array(){
114     suffix_array.resize(size,0);
115     lcp.resize(size,0);
116     int index = 0;
117     int depth = 0;
118     dfs(0,index,0,0);
119 }
120
121 // pensado para map<int,int>, pero puede modificarse para vector<int
122 >
123 bool match(string& a, string& base){
124     int active_node = 0;
125     int active_length = 0;
126     int active_char = -1;
127     for(int i = 0; i < a.size();){
128         if(active_length == 0){
129             if(!tree[active_node].has_child(a[i]))
130                 return false;
131             active_char = a[i];
132             active_length++;
133             i++;
134             continue;
135         }
136         int next = tree[active_node].children[active_char];
137         if(active_length == length(next)){
138             active_node = next;
139             active_length = 0;
140             active_char = -1;
141             continue;
142         }
143         if((base)[tree[next].start + active_length] != a[i])
144             return false;
145         active_length++;
146         i++;
147     }
148     return true;
149 };

```