

Contents

| | | | | | |
|----------|--|-----------|-------|-----------------------------------|----|
| 1 | Info About Memory and Time Limits | 2 | | | |
| 2 | C++ Cheat Sheet | 2 | | | |
| 2.1 | Headers | 2 | | | |
| 2.2 | Cheat Sheet | 3 | | | |
| 3 | General Algorithms | 8 | | | |
| 3.1 | 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 | 9 | | | |
| 5 | Dynamic Programming | 10 | | | |
| 6 | Graphs | 10 | | | |
| 6.1 | Graph Traversal | 10 | | | |
| 6.1.1 | Breadth First Search | 10 | | | |
| 6.1.2 | Recursive Depth First Search | 10 | | | |
| 6.1.3 | Iterative Depth First Search | 11 | | | |
| 6.2 | Shortest Path Algorithms | 11 | | | |
| 6.2.1 | Dijkstra | 11 | | | |
| 6.2.2 | Bellman Ford | 11 | | | |
| 6.2.3 | Floyd Warshall | 11 | | | |
| 6.3 | Minimum Spanning Tree (MST) | 12 | | | |
| 6.3.1 | Kruskal | 12 | | | |
| 6.4 | Lowest Common Ancestor (LCA) | 12 | | | |
| 6.5 | Max Flow | 13 | | | |
| 6.6 | Others | 14 | | | |
| 6.6.1 | Diameter of a tree | 14 | | | |
| 7 | Mathematics | 15 | | | |
| 7.1 | Useful Data | 15 | | | |
| 7.2 | Modular Arithmetic | 15 | | | |
| 7.2.1 | Chinese Remainder Theorem | 15 | | | |
| 7.3 | Primality Checks | 16 | | | |
| 7.3.1 | Miller Rabin | 16 | | | |
| 7.3.2 | Sieve of Eratosthenes | 16 | | | |
| 7.3.3 | trialDivision | 17 | | | |
| 7.4 | Others | 17 | | | |
| 7.4.1 | Polynomials | 17 | | | |
| | | | 7.4.2 | Factorial Factorization | 18 |
| 8 | Geometry | 18 | | | |
| 8.1 | Vectors/Points | 18 | | | |
| 8.2 | Calculate Areas | 21 | | | |
| 8.2.1 | Integration via Simpson's Method | 21 | | | |
| 8.2.2 | Green's Theorem | 21 | | | |
| 8.3 | Pick's Theorem | 21 | | | |
| 9 | Strings | 21 | | | |
| 9.1 | Trie | 21 | | | |
| 9.2 | KMP | 22 | | | |

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>
3
4  using namespace std;
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 #define rep(i, n) for (int i = 0; i < (int)n; i++)
21 #define repx(i, a, b) for (int i = a; i < (int)b; i++)
22 #define invrep(i, a, b) for (int i = b; i-- > (int)a;)
23
24 #define pb push_back
25 #define eb emplace_back
26 #define ppb pop_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
28 #define lg(x) (31 - __builtin_clz(x))
29 #define lgg(x) (63 - __builtin_clzll(x))
30 #define gcd __gcd
31
32 #define INF INT_MAX
33
34 #define umap unordered_map
35 #define uset unordered_set
36
37 #define debugx(x) cerr << #x << ": " << x << endl
38 #define debugv(v) \
39     cerr << #v << ":\n"; \
40     for (auto e : v) \
41     { \
42         cerr << " " << e; \
43     } \
44     cerr << endl
45 #define debugm(m) \
46     cerr << #m << endl; \
47     rep(i, (int)m.size()) \
48     { \
49         cerr << i << ":\n"; \
50         rep(j, (int)m[i].size()) cerr << " " << m[i][j]; \
51         cerr << endl; \
52     } \
53 #define debugmp(m) \
54     cerr << #m << endl; \
55     rep(i, (int)m.size()) \
56     { \
57         \
58         cerr << i << ":\n"; \
59         rep(j, (int)m[i].size()) \
60         { \
61             \
62             cerr << " {" << m[i][j].first << ", " << m[i][j].second << " } \
63             "; \
64         } \
65         \
66         cerr << endl; \
67     } \
68 #define print(x) copy(x.begin(), x.end(), ostream_iterator<int>(cout, \
69     "")), cout << endl
70
71 template <typename T1, typename T2>
72 ostream &operator<<(ostream &os, const pair<T1, T2> &p)

```

```

68 {
69     os << '(' << p.first << ',' << p.second << ')';
70     return os;
71 }
72
73 #endif

```

2.2 Cheat Sheet

```

1  #include "../headers/headers.h"
2
3  // Note: This Cheat Sheet is by no means complete
4  // If you want a thorough documentation of the Standard C++ Library
5  // please refer to this link: http://www.cplusplus.com/reference/
6
7  /* ===== */
8  /* Reading from stdin */
9  /* ===== */
10 // With scanf
11 scanf("%d", &a);           //int
12 scanf("%x", &a);           // int in hexadecimal
13 scanf("%llx", &a);          // long long in hexadecimal
14 scanf("%lld", &a);          // long long int
15 scanf("%c", &c);            // char
16 scanf("%s", buffer);        // string without whitespaces
17 scanf("%f", &f);            // float
18 scanf("%lf", &d);            // double
19 scanf("%d %s %d", &a, &b); // * = consume but skip
20
21 // read until EOL
22 // - EOL not included in buffer
23 // - EOL is not consumed
24 // - nothing is written into buffer if EOF is found
25 scanf(" %[^\n]", buffer);
26
27 //reading until EOL or EOF
28 // - EOL not included in buffer
29 // - EOL is consumed
30 // - works with EOF
31 char *output = gets(buffer);
32 if (feof(stdin))
33 {
34 } // EOF file found
35 if (output == buffer)
36 {
37 } // succesful read
38 if (output == NULL)
39 {
40 } // EOF found without previous chars found
41 //example
42 while (gets(buffer) != NULL)
43 {
44     puts(buffer);
45     if (feof(stdin))
46     {

```

```

47         break;
48     }
49 }
50
51 // read single char
52 getchar();
53 while (true)
54 {
55     c = getchar();
56     if (c == EOF || c == '\n')
57         break;
58 }
59
60 /* ===== */
61 /* Printing to stdout */
62 /* ===== */
63 // With printf
64 printf("%d", a);           // int
65 printf("%u", a);           // unsigned int
66 printf("%lld", a);          // long long int
67 printf("%llu", a);          // unsigned long long int
68 printf("%c", c);           // char
69 printf("%s", buffer);       // string until \0
70 printf("%f", f);            // float
71 printf("%lf", d);           // double
72 printf("%0*.f", x, y, f);    // padding = 0, width = x, decimals = y
73 printf("%.5s\n", buffer);   // print at most the first five characters
74                               // (safe to use on short strings)
75
76 // print at most first n characters (safe)
77 printf("%.s\n", n, buffer); // make sure that n is integer (with long
78                               // long I had problems)
79 //string + \n
80 puts(buffer);
81
82 /* ===== */
83 /* Reading from c string */
84 /* ===== */
85
86 // same as scanf but reading from s
87 int sscanf(const char *s, const char *format, ...);
88
89 /* ===== */
90 /* Printing to c string */
91 /* ===== */
92 // Same as printf but writing into str, the number of characters is
93 // 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 */

```

```

99  /* ===== */
100 bool peekAndCheck(char c)
101 {
102     char c2 = getchar();
103     ungetc(c2, stdin); // return char to stdin
104     return c == c2;
105 }
106
107 /* ===== */
108 /* Reading from cin */
109 /* ===== */
110 // reading a line of unknown length
111 string line;
112 getline(cin, line);
113 while (getline(cin, line))
114 {
115 }
116
117 // Optimizations with cin/cout
118 ios::sync_with_stdio(0);
119 cin.tie(0);
120 cout.tie(0);
121
122 // Fix precision on cout
123 cout.setf(ios::fixed);
124 cout.precision(4); // e.g. 1.000
125
126 /* ===== */
127 /* USING PAIRS AND TUPLES */
128 /* ===== */
129 // ii = pair<int,int>
130 ii p(5, 5);
131 ii p = make_pair(5, 5)
132     ii p = {5, 5};
133 int x = p.first, y = p.second;
134 // iii = tuple<int,int,int>
135 iii t(5, 5, 5);
136 tie(x, y, z) = t;
137 tie(x, y, z) = make_tuple(5, 5, 5);
138 get<0>(t)++;
139 get<1>(t)--;
140
141 /* ===== */
142 /* CONVERTING FROM STRING TO NUMBERS */
143 /* ===== */
144 //-----
145 // string to int
146 // option #1:
147 int atoi(const char *str);
148 // option #2:
149 sscanf(string, "%d", &i);
150 //-----
151 // string to long int:
152 // option #1:
153 long int strtol(const char *str, char **endptr, int base);

```

```

154 // it only works skipping whitespaces, so make sure your numbers
155 // are surrounded by whitespaces only
156 // Example:
157 char szNumbers[] = "2001 60c0c0 -1101110100110100100000 0x6fffff";
158 char *pEnd;
159 long int li1, li2, li3, li4;
160 li1 = strtol(szNumbers, &pEnd, 10);
161 li2 = strtol(pEnd, &pEnd, 16);
162 li3 = strtol(pEnd, &pEnd, 2);
163 li4 = strtol(pEnd, NULL, 0);
164 printf("The decimal equivalents are: %ld, %ld, %ld and %ld.\n", li1, li2
165     , li3, li4);
166 // option #2:
167 long int atol(const char *str);
168 // option #3:
169 sscanf(string, "%ld", &l);
170 //-----
171 // string to long long int:
172 // option #1:
173 long long int strtoll(const char *str, char **endptr, int base);
174 // option #2:
175 sscanf(string, "%lld", &l);
176 //-----
177 // string to double:
178 // option #1:
179 double strtod(const char *str, char **endptr); //similar to strtol
180 // option #2:
181 double atof(const char *str);
182 // option #3:
183 sscanf(string, "%lf", &d);
184
185 /* ===== */
186 /* C STRING UTILITY FUNCTIONS */
187 /* ===== */
188 int strcmp(const char *str1, const char *str2); // (-1,0,1)
189 int memcmp(const void *ptr1, const void *ptr2, size_t num); // (-1,0,1)
190 void *memcpy(void *destination, const void *source, size_t num);
191
192 /* ===== */
193 /* C++ STRING UTILITY FUNCTIONS */
194 /* ===== */
195 // read tokens from string
196 string s = "tok1 tok2 tok3";
197 string tok;
198 stringstream ss(s);
199 while (getline(ss, tok, ' '))
200     printf("tok = %s\n", tok.c_str());
201
202 // split a string by a single char delimiter
203 void split(const string &s, char delim, vector<string> &elems)
204 {
205     stringstream ss(s);
206     string item;
207     while (getline(ss, item, delim))
208         elems.push_back(item);

```

```

208 }
209
210 // find index of string or char within string
211 string str = "random";
212 std::size_t pos = str.find("ra");
213 std::size_t pos = str.find('m');
214 if (pos == string::npos) // not found
215
216     // substrings
217     string subs = str.substr(pos, length);
218 string subs = str.substr(pos); // default: to the end of the string
219
220 // std::string from cstring's substring
221 const char *s = "bla1 bla2";
222 int offset = 5, len = 4;
223 string subs(s + offset, len); // bla2
224
225 // -----
226 // string comparisons
227 int compare(const string &str) const;
228 int compare(size_t pos, size_t len, const string &str) const;
229 int compare(size_t pos, size_t len, const string &str,
230             size_t subpos, size_t sublen) const;
231 int compare(const char *s) const;
232 int compare(size_t pos, size_t len, const char *s) const;
233
234 // examples
235 // 1) check string begins with another string
236 string prefix = "prefix";
237 string word = "prefix suffix";
238 word.compare(0, prefix.size(), prefix);
239
240 /* ===== */
241 /* OPERATOR OVERLOADING */
242 /* ===== */
243
244 //-----
245 // method #1: inside struct
246 struct Point
247 {
248     int x, y;
249     bool operator<(const Point &p) const
250     {
251         if (x != p.x)
252             return x < p.x;
253         return y < p.y;
254     }
255     bool operator>(const Point &p) const
256     {
257         if (x != p.x)
258             return x > p.x;
259         return y > p.y;
260     }
261     bool operator==(const Point &p) const
262     {

```

```

263         return x == p.x && y == p.y;
264     }
265 };
266
267 //-----
268 // method #2: outside struct
269 struct Point
270 {
271     int x, y;
272 };
273 bool operator<(const Point &a, const Point &b)
274 {
275     if (a.x != b.x)
276         return a.x < b.x;
277     return a.y < b.y;
278 }
279 bool operator>(const Point &a, const Point &b)
280 {
281     if (a.x != b.x)
282         return a.x > b.x;
283     return a.y > b.y;
284 }
285 bool operator==(const Point &a, const Point &b)
286 {
287     return a.x == b.x && a.y == b.y;
288 }
289
290 // Note: if you overload the < operator for a custom struct,
291 // then you can use that struct with any library function
292 // or data structure that requires the < operator
293 // Examples:
294 priority_queue<Point> pq;
295 vector<Point> pts;
296 sort(pts.begin(), pts.end());
297 lower_bound(pts.begin(), pts.end(), {1, 2});
298 upper_bound(pts.begin(), pts.end(), {1, 2});
299 set<Point> pt_set;
300 map<Point, int> pt_map;
301
302 /* ===== */
303 /* CUSTOM COMPARISONS */
304 /* ===== */
305 // method #1: operator overloading
306 // method #2: custom comparison function
307 bool cmp(const Point &a, const Point &b)
308 {
309     if (a.x != b.x)
310         return a.x < b.x;
311     return a.y < b.y;
312 }
313 // method #3: functor
314 struct cmp
315 {
316     bool operator()(const Point &a, const Point &b)
317     {

```

```

318     if (a.x != b.x)
319         return a.x < b.x;
320     return a.y < b.y;
321 }
322 };
323 // without operator overloading, you would have to use
324 // an explicit comparison method when using library
325 // functions or data structures that require sorting
326 priority_queue<Point, vector<Point>, cmp> pq;
327 vector<Point> pts;
328 sort(pts.begin(), pts.end(), cmp);
329 lower_bound(pts.begin(), pts.end(), {1, 2}, cmp);
330 upper_bound(pts.begin(), pts.end(), {1, 2}, cmp);
331 set<Point, cmp> pt_set;
332 map<Point, int, cmp> pt_map;
333
334 /* ===== */
335 /* VECTOR UTILITY FUNCTIONS */
336 /* ===== */
337 vector<int> myvector;
338 myvector.push_back(100);
339 myvector.pop_back(); // remove last element
340 myvector.back();     // peek reference to last element
341 myvector.front();    // peek reference to first element
342 myvector.clear();    // remove all elements
343 // sorting a vector
344 vector<int> foo;
345 sort(foo.begin(), foo.end());
346 sort(foo.begin(), foo.end(), std::less<int>()); // increasing
347 sort(foo.begin(), foo.end(), std::greater<int>()); // decreasing
348
349 /* ===== */
350 /* SET UTILITY FUNCTIONS */
351 /* ===== */
352 set<int> myset;
353 myset.begin(); // iterator to first elemnt
354 myset.end();   // iterator to after last element
355 myset.rbegin(); // iterator to last element
356 myset.rend();  // iterator to before first element
357 for (auto it = myset.begin(); it != myset.end(); ++it)
358 {
359     do_something(*it);
360 } // left -> right
361 for (auto it = myset.rbegin(); it != myset.rend(); ++it)
362 {
363     do_something(*it);
364 } // right -> left
365 for (auto &i : myset)
366 {
367     do_something(i);
368 } // left->right shortcut
369 auto ret = myset.insert(5); // ret.first = iterator, ret.second =
    boolean (inserted / not inserted)
370 int count = myset.erase(5); // count = how many items were erased
371 if (!myset.empty())

```

```

372 {
373 }
374 // custom comparator 1: functor
375 struct cmp
376 {
377     bool operator()(int i, int j) { return i > j; }
378 };
379 set<int, cmp> myset;
380 // custom comparator 2: function
381 bool cmp(int i, int j) { return i > j; }
382 set<int, bool (*)(int, int)> myset(cmp);
383
384 /* ===== */
385 /* MAP UTILITY FUNCTIONS */
386 /* ===== */
387 struct Point
388 {
389     int x, y;
390 };
391 bool operator<(const Point &a, const Point &b)
392 {
393     return a.x < b.x || (a.x == b.x && a.y < b.y);
394 }
395 map<Point, int> ptcounts;
396
397 // -----
398 // inserting into map
399
400 // method #1: operator[]
401 // it overwrites the value if the key already exists
402 ptcounts[{1, 2}] = 1;
403
404 // method #2: .insert(pair<key, value>)
405 // it returns a pair { iterator(key, value) , bool }
406 // if the key already exists, it doesn't overwrite the value
407 void update_count(Point &p)
408 {
409     auto ret = ptcounts.emplace(p, 1);
410     // auto ret = ptcounts.insert(make_pair(p, 1)); //
411     if (!ret.second)
412         ret.first->second++;
413 }
414
415 // -----
416 // generating ids with map
417 int get_id(string &name)
418 {
419     static int id = 0;
420     static map<string, int> name2id;
421     auto it = name2id.find(name);
422     if (it == name2id.end())
423         return name2id[name] = id++;
424     return it->second;
425 }
426

```

```

427 /* ===== */
428 /* BITSET UTILITY FUNCTIONS */
429 /* ===== */
430 bitset<4> foo; // 0000
431 foo.size(); // 4
432 foo.set(); // 1111
433 foo.set(1, 0); // 1011
434 foo.test(1); // false
435 foo.set(1); // 1111
436 foo.test(1); // true
437
438 /* ===== */
439 /* RANDOM INTEGERS */
440 /* ===== */
441 #include <cstdlib>
442 #include <ctime>
443 srand(time(NULL));
444 int x = rand() % 100; // 0-99
445 int randBetween(int a, int b)
446 { // a-b
447     return a + (rand() % (1 + b - a));
448 }
449
450 /* ===== */
451 /* CLIMITS */
452 /* ===== */
453 #include <climits>
454 INT_MIN
455 INT_MAX
456 UINT_MAX
457 LONG_MIN
458 LONG_MAX
459 ULONG_MAX
460 LLONG_MIN
461 LLONG_MAX
462 ULLONG_MAX
463
464 /* ===== */
465 /* Bitwise Tricks */
466 /* ===== */
467
468 // amount of one-bits in number
469 int __builtin_popcount(int x);
470 int __builtin_popcountl(long x);
471 int __builtin_popcountll(long long x);
472
473 // amount of leading zeros in number
474 int __builtin_clz(int x);
475 int __builtin_clzl(long x);
476 int __builtin_clzll(long long x);
477
478 // binary length of non-negative number
479 int bitlen(int x) { return sizeof(x) * 8 - __builtin_clz(x); }
480 int bitlen(ll x) { return sizeof(x) * 8 - __builtin_clzll(x); }
481

```

```

482 // index of most significant bit
483 int log2(int x) { return sizeof(x) * 8 - __builtin_clz(x) - 1; }
484 int log2(ll x) { return sizeof(x) * 8 - __builtin_clzll(x) - 1; }
485
486 // reverse the bits of an integer
487 int reverse_bits(int x)
488 {
489     int v = 0;
490     while (x)
491         v <<= 1, v |= x & 1, x >>= 1;
492     return v;
493 }
494
495 // get string binary representation of an integer
496 string bitstring(int x)
497 {
498     int len = sizeof(x) * 8 - __builtin_clz(x);
499     if (len == 0)
500         return "0";
501
502     char buff[len + 1];
503     buff[len] = '\0';
504     for (int i = len - 1; i >= 0; --i, x >>= 1)
505         buff[i] = (char)('0' + (x & 1));
506     return string(buff);
507 }
508
509 /* ===== */
510 /* Hexadecimal Tricks */
511 /* ===== */
512
513 // get string hex representation of an integer
514 string to_hex(int num)
515 {
516     static char buff[100];
517     static const char *hexdigits = "0123456789abcdef";
518     buff[99] = '\0';
519     int i = 98;
520     do
521     {
522         buff[i--] = hexdigits[num & 0xf];
523         num >>= 4;
524     } while (num);
525     return string(buff + i + 1);
526 }
527
528 // ['0'-'9' 'a'-'f'] -> [0 - 15]
529 int char_to_digit(char c)
530 {
531     if ('0' <= c && c <= '9')
532         return c - '0';
533     return 10 + c - 'a';
534 }
535
536 /* ===== */

```

```

537 /* Other Tricks */
538 /* ===== */
539 // swap stuff
540 int x = 1, y = 2;
541 swap(x, y);
542
543 /* ===== */
544 /* TIPS */
545 /* ===== */
546 // 1) do not use .emplace(x, y) if your struct doesn't have an explicit
    constructor
547 // instead you can use .push({x, y})
548 // 2) be careful while mixing scanf() with getline(), scanf will not
    consume \n unless
549 // you explicitly tell it to do so (e.g scanf("%d\n", &x)) )

```

3 General Algorithms

3.1 Search

3.2 Brute Force

4 Data Structures

4.1 Segment Tree

4.1.1 Lazy

```

1 #include "../headers/headers.h"
2
3 struct RSQ // Range sum query
4 {
5     static ll const neutro = 0;
6     static ll op(ll x, ll y)
7     {
8         return x + y;
9     }
10    static ll
11    lazy_op(int i, int j, ll x)
12    {
13        return (j - i + 1) * x;
14    }
15 };
16
17 struct RMinQ // Range minimum query
18 {
19     static ll const neutro = 1e18;
20     static ll op(ll x, ll y)
21     {
22         return min(x, y);
23     }
24    static ll
25    lazy_op(int i, int j, ll x)

```

```

26    {
27        return x;
28    }
29 };
30
31 template <class t>
32 class SegTreeLazy
33 {
34     vector<ll> arr, st, lazy;
35     int n;
36
37     void build(int u, int i, int j)
38     {
39         if (i == j)
40         {
41             st[u] = arr[i];
42             return;
43         }
44         int m = (i + j) / 2, l = u * 2 + 1, r = u * 2 + 2;
45         build(l, i, m);
46         build(r, m + 1, j);
47         st[u] = t::op(st[l], st[r]);
48     }
49
50     void propagate(int u, int i, int j, ll x)
51     {
52         // nota, las operaciones pueden ser un and, or, ..., etc.
53         st[u] += t::lazy_op(i, j, x); // incrementar el valor (+)
54         // st[u] = t::lazy_op(i, j, x); // setear el valor
55         if (i != j)
56         {
57             // incrementar el valor
58             lazy[u * 2 + 1] += x;
59             lazy[u * 2 + 2] += x;
60             // setear el valor
61             //lazy[u * 2 + 1] = x;
62             //lazy[u * 2 + 2] = x;
63         }
64         lazy[u] = 0;
65     }
66
67     ll query(int a, int b, int u, int i, int j)
68     {
69         if (j < a or b < i)
70             return t::neutro;
71         int m = (i + j) / 2, l = u * 2 + 1, r = u * 2 + 2;
72         if (lazy[u])
73             propagate(u, i, j, lazy[u]);
74         if (a <= i and j <= b)
75             return st[u];
76         ll x = query(a, b, l, i, m);
77         ll y = query(a, b, r, m + 1, j);
78         return t::op(x, y);
79     }
80 }

```



```

81 void update(int a, int b, ll value,
82            int u, int i, int j)
83 {
84     int m = (i + j) / 2, l = u * 2 + 1, r = u * 2 + 2;
85     if (lazy[u])
86         propagate(u, i, j, lazy[u]);
87     if (a <= i and j <= b)
88         propagate(u, i, j, value);
89     else if (j < a or b < i)
90         return;
91     else
92     {
93         update(a, b, value, l, i, m);
94         update(a, b, value, r, m + 1, j);
95         st[u] = t::op(st[l], st[r]);
96     }
97 }
98
99 public:
100 SegTreeLazy(vector<ll> &v)
101 {
102     arr = v;
103     n = v.size();
104     st.resize(n * 4 + 5);
105     lazy.assign(n * 4 + 5, 0);
106     build(0, 0, n - 1);
107 }
108
109 ll query(int a, int b)
110 {
111     return query(a, b, 0, 0, n - 1);
112 }
113
114 void update(int a, int b, ll value)
115 {
116     update(a, b, value, 0, 0, n - 1);
117 }
118 };

```

4.1.2 Iterative

```

1 #include "../headers/headers.h"
2
3 // It requires a struct for a node (e.g. prodsgn)
4 // A node must have three constructors
5 //     Arity 0: Constructs the identity of the operation (e.g. 1 for
6 //     prodsgn)
7 //     Arity 1: Constructs a leaf node from the input
8 //     Arity 2: Constructs a node from its children
9 // Building the Segment Tree:
10 //     Create a vector of nodes (use constructor of arity 1).
11 //     ST<miStructNode> mySegmentTree(vectorOfNodes);
12 // Update:
13 //     mySegmentTree.set_points(index, myStructNode(input));

```

```

14 // Query:
15 //     mySegmentTree.query(l, r); (It searches on the range [l,r), and
16 //     returns a node.)
17
18 // Logic And Query
19 struct ANDQ
20 {
21     ll value;
22     ANDQ() { value = -1ll; }
23     ANDQ(ll x) { value = x; }
24     ANDQ(const ANDQ &a,
25           const ANDQ &b)
26     {
27         value = a.value & b.value;
28     }
29 };
30
31 // Interval Product (LiveArchive)
32 struct prodsgn
33 {
34     int sgn;
35     prodsgn() { sgn = 1; }
36     prodsgn(int x)
37     {
38         sgn = (x > 0) - (x < 0);
39     }
40     prodsgn(const prodsgn &a,
41             const prodsgn &b)
42     {
43         sgn = a.sgn * b.sgn;
44     }
45 };
46
47 // Maximum Sum (SPOJ)
48 struct maxsum
49 {
50     int first, second;
51     maxsum() { first = second = -1; }
52     maxsum(int x)
53     {
54         first = x;
55         second = -1;
56     }
57     maxsum(const maxsum &a,
58           const maxsum &b)
59     {
60         if (a.first > b.first)
61         {
62             first = a.first;
63             second = max(a.second,
64                         b.first);
65         }
66         else
67         {
68             first = b.first;

```

```

68         second = max(a.first,
69                       b.second);
70     }
71 }
72 int answer()
73 {
74     return first + second;
75 }
76 };
77
78 // Range Minimum Query
79 struct rminq
80 {
81     int value;
82     rminq() { value = INT_MAX; }
83     rminq(int x) { value = x; }
84     rminq(const rminq &a,
85           const rminq &b)
86     {
87         value = min(a.value,
88                     b.value);
89     }
90 };
91
92 template <class node>
93 class ST
94 {
95     vector<node> t;
96     int n;
97
98 public:
99     ST(vector<node> &arr)
100     {
101         n = arr.size();
102         t.resize(n * 2);
103         copy(arr.begin(), arr.end(), t.begin() + n);
104         for (int i = n - 1; i > 0; --i)
105             t[i] = node(t[i << 1], t[i << 1 | 1]);
106     }
107
108     // 0-indexed
109     void set_point(int p, const node &value)
110     {
111         for (t[p += n] = value; p > 1; p >>= 1)
112             t[p >> 1] = node(t[p], t[p ^ 1]);
113     }
114
115     // inclusive exclusive, 0-indexed
116     node query(int l, int r)
117     {
118         node ans1, ansr;
119         for (l += n, r += n; l < r; l >>= 1, r >>= 1)
120         {
121             if (l & 1)
122                 ans1 = node(ans1, t[l++]);

```

```

123             if (r & 1)
124                 ansr = node(t[--r], ansr);
125             }
126             return node(ans1, ansr);
127         }
128     };

```

5 Dynamic Programming

6 Graphs

6.1 Graph Traversal

6.1.1 Breadth First Search

```

1 #include "../headers/headers.h"
2
3 void bfs(graph &g, int start)
4 {
5     int n = g.size();
6     vi visited(n, 1);
7     queue<int> q;
8
9     q.emplace(start);
10    visited[start] = 0;
11    while (not q.empty())
12    {
13        int u = q.front();
14        q.pop();
15
16        for (int v : g[u])
17        {
18            if (visited[v])
19            {
20                q.emplace(v);
21                visited[v] = 0;
22            }
23        }
24    }
25 }

```

6.1.2 Recursive Depth First Search

```

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

```

```
11 | }
```

6.1.3 Iterative Depth First Search

```
1 | #include "../headers/headers.h"
2 | //Iterative
3 | void dfs_i(graph &g, int start)
4 | {
5 |     int n = g.size();
6 |     vi visited(n, 1);
7 |     stack<int> s;
8 |
9 |     s.emplace(start);
10 |    visited[start] = 0;
11 |
12 |    while (not s.empty())
13 |    {
14 |        int u = s.top();
15 |        s.pop();
16 |
17 |        cout << u << '\n';
18 |
19 |        for (int v : g[u])
20 |            if (visited[v])
21 |            {
22 |                s.emplace(v);
23 |                visited[v] = 0;
24 |            }
25 |    }
26 | }
```

6.2 Shortest Path Algorithms

6.2.1 Dijkstra

All edges have non-negative values

```
1 | #include "../headers/headers.h"
2 | //g has vectors of pairs of the form (w, index)
3 | int dijkstra(wgraph g, int start, int end)
4 | {
5 |     int n = g.size();
6 |     vi cost(n, 1e9); //~INT_MAX/2
7 |     priority_queue<ii, greater<ii>> q;
8 |
9 |     q.emplace(0, start);
10 |    cost[start] = 0;
11 |    while (not q.empty())
12 |    {
13 |        int u = q.top().second, w = q.top().first;
14 |        q.pop();
15 |
16 |        // we skip all nodes in the q that we have discovered before at
17 |        // a lower cost
18 |        if (cost[u] < w) continue;
```

```
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 |
29 |     return cost[end];
30 | }
```

6.2.2 Bellman Ford

Edges can be negative, and it detects negative cycles

```
1 | #include "../headers/headers.h"
2 | bool bellman_ford(wgraph &g, int start)
3 | {
4 |     int n = g.size();
5 |     vector<int> dist(n, 1e9); //~INT_MAX/2
6 |     dist[start] = 0;
7 |     rep(i, n - 1) rep(u, n) for (ii p : g[u])
8 |     {
9 |         int v = p.first, w = p.second;
10 |        dist[v] = min(dist[v], dist[u] + w);
11 |    }
12 |
13 |    bool hayCicloNegativo = false;
14 |    rep(u, n) for (ii p : g[u])
15 |    {
16 |        int v = p.first, w = p.second;
17 |        if (dist[v] > dist[u] + w)
18 |            hayCicloNegativo = true;
19 |    }
20 |
21 |    return hayCicloNegativo;
22 | }
```

6.2.3 Floyd Warshall

Shortest path from every node to every other node

```
1 | #include "../headers/headers.h"
2 |
3 | /*
4 | Floyd Warshall implemenation, note that g is using an adjacency matrix
5 | and not an
6 | adjacency list
7 | */
8 | graph floydWarshall (const graph g)
9 | {
10 |     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 #include "../headers/headers.h"
2 struct edge
3 {
4     int u, v;
5     ll w;
6     edge(int u, int v, ll w) : u(u), v(v), w(w) {}
7
8     bool operator<(const edge &o) const
9     {
10         return w < o.w;
11     }
12 };
13
14 class Kruskal
15 {
16 private:
17     ll sum;
18     vi p, rank;
19
20 public:
21     //Amount of Nodes n, and unordered vector of Edges E
22     Kruskal(int n, vector<edge> E)
23     {
24         sum = 0;
25         p.resize(n);
26         rank.assign(n, 0);
27         rep(i, n) p[i] = i;
28         sort(E.begin(), E.end());
29         for (auto &e : E)
30             UnionSet(e.u, e.v, e.w);
31     }
32     int findSet(int i)
33     {

```

```

34         return (p[i] == i) ? i : (p[i] = findSet(p[i]));
35     }
36     bool isSameSet(int i, int j)
37     {
38         return findSet(i) == findSet(j);
39     }
40     void UnionSet(int i, int j, ll w)
41     {
42         if (not isSameSet(i, j))
43         {
44             int x = findSet(i), y = findSet(j);
45             if (rank[x] > rank[y])
46                 p[y] = x;
47             else
48                 p[x] = y;
49
50             if (rank[x] == rank[y])
51                 rank[y]++;
52
53             sum += w;
54         }
55     }
56     ll mst_val()
57     {
58         return sum;
59     }
60 };

```

6.4 Lowest Common Ancestor (LCA)

Supports multiple trees

```

1 #include "../headers/headers.h"
2 class LcaForest
3 {
4     int n;
5     vi parent;
6     vi level;
7     vi root;
8     graph P;
9
10 public:
11     LcaForest(int n)
12     {
13         this->n = n;
14         parent.assign(n, -1);
15         level.assign(n, -1);
16         P.assign(n, vi(lg(n) + 1, -1));
17         root.assign(n, -1);
18     }
19     void addLeaf(int index, int par)
20     {
21         parent[index] = par;
22         level[index] = level[par] + 1;
23         P[index][0] = par;
24         root[index] = root[par];

```

```

25     for (int j = 1; (1 << j) < n; ++j)
26     {
27         if (P[index][j - 1] != -1)
28             P[index][j] = P[P[index][j - 1]][j - 1];
29     }
30 }
31 void addRoot(int index)
32 {
33     parent[index] = index;
34     level[index] = 0;
35     root[index] = index;
36 }
37 int lca(int u, int v)
38 {
39     if (root[u] != root[v] || root[u] == -1)
40         return -1;
41     if (level[u] < level[v])
42         swap(u, v);
43     int dist = level[u] - level[v];
44     while (dist != 0)
45     {
46         int raise = lg(dist);
47         u = P[u][raise];
48         dist -= (1 << raise);
49     }
50     if (u == v)
51         return u;
52     for (int j = lg(n); j >= 0; --j)
53     {
54         if (P[u][j] != -1 && P[u][j] != P[v][j])
55         {
56             u = P[u][j];
57             v = P[v][j];
58         }
59     }
60     return parent[u];
61 }
62 };

```

6.5 Max Flow

```

1  #include "../headers/headers.h"
2  class Dinic
3  {
4      struct edge
5      {
6          int to, rev;
7          ll f, cap;
8      };
9
10     vector<vector<edge>> g;
11     vector<ll> dist;
12     vector<int> q, work;
13     int n, sink;

```

```

14
15     bool bfs(int start, int finish)
16     {
17         dist.assign(n, -1);
18         dist[start] = 0;
19         int head = 0, tail = 0;
20         q[tail++] = start;
21         while (head < tail)
22         {
23             int u = q[head++];
24             for (const edge &e : g[u])
25             {
26                 int v = e.to;
27                 if (dist[v] == -1 and e.f < e.cap)
28                 {
29                     dist[v] = dist[u] + 1;
30                     q[tail++] = v;
31                 }
32             }
33         }
34         return dist[finish] != -1;
35     }
36
37     ll dfs(int u, ll f)
38     {
39         if (u == sink)
40             return f;
41         for (int &i = work[u]; i < (int)g[u].size(); ++i)
42         {
43             edge &e = g[u][i];
44             int v = e.to;
45             if (e.cap <= e.f or dist[v] != dist[u] + 1)
46                 continue;
47             ll df = dfs(v, min(f, e.cap - e.f));
48             if (df > 0)
49             {
50                 e.f += df;
51                 g[v][e.rev].f -= df;
52                 return df;
53             }
54         }
55         return 0;
56     }
57
58     public:
59     Dinic(int n)
60     {
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     {

```

```

69     edge a = {v, (int)g[v].size(), 0, cap};
70     edge b = {u, (int)g[u].size(), 0, 0}; //Poner cap en vez de 0 si
        la arista es bidireccional
71     g[u].pb(a);
72     g[v].pb(b);
73 }
74
75 ll max_flow(int source, int dest)
76 {
77     sink = dest;
78     ll ans = 0;
79     while (bfs(source, dest))
80     {
81         work.assign(n, 0);
82         while (ll delta = dfs(source, LLONG_MAX))
83             ans += delta;
84     }
85     return ans;
86 }
87 };

```

6.6 Others

6.6.1 Diameter of a tree

```

1  #include "../headers/headers.h"
2
3  graph Tree;
4  vi dist;
5
6  // Finds a diameter node
7  int bfs1()
8  {
9      int n = Tree.size();
10     queue<int> q;
11
12     q.emplace(0);
13     dist[0] = 0;
14     int u;
15     while (not q.empty())
16     {
17         u = q.front();
18         q.pop();
19
20         for (int v : Tree[u])
21         {
22             if (dist[v] == -1)
23             {
24                 q.emplace(v);
25                 dist[v] = dist[u] + 1;
26             }
27         }
28     }
29     return u;
30 }

```

```

31 // Fills the distances from one diameter node and finds another diameter
32 // node
33 int bfs2()
34 {
35     int n = Tree.size();
36     vi visited(n, 1);
37     queue<int> q;
38     int start = bfs1();
39     q.emplace(start);
40     visited[start] = 0;
41     int u;
42     while (not q.empty())
43     {
44         u = q.front();
45         q.pop();
46
47         for (int v : Tree[u])
48         {
49             if (visited[v])
50             {
51                 q.emplace(v);
52                 visited[v] = 0;
53                 dist[v] = max(dist[v], dist[u] + 1);
54             }
55         }
56     }
57     return u;
58 }
59
60 // Finds the diameter
61 int bfs3()
62 {
63     int n = Tree.size();
64     vi visited(n, 1);
65     queue<int> q;
66     int start = bfs2();
67     q.emplace(start);
68     visited[start] = 0;
69     int u;
70     while (not q.empty())
71     {
72         u = q.front();
73         q.pop();
74
75         for (int v : Tree[u])
76         {
77             if (visited[v])
78             {
79                 q.emplace(v);
80                 visited[v] = 0;
81                 dist[v] = max(dist[v], dist[u] + 1);
82             }
83         }
84     }

```

```

85     return dist[u];
86 }

```

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  #include "../headers/headers.h"
2
3  ll inline mod(ll x, ll m) { return ((x % m) < 0) ? x + m : x; }
4  ll inline mul(ll x, ll y, ll m) { return (x * y) % m; }
5  ll inline add(ll x, ll y, ll m) { return (x + y) % m; }
6
7  // extended euclidean algorithm
8  // finds g, x, y such that
9  //   a * x + b * y = g = GCD(a,b)
10 ll gcdext(ll a, ll b, ll &x, ll &y)
11 {
12     ll r2, x2, y2, r1, x1, y1, r0, x0, y0, q;
13     r2 = a, x2 = 1, y2 = 0;
14     r1 = b, x1 = 0, y1 = 1;
15     while (r1)
16     {
17         q = r2 / r1;
18         r0 = r2 % r1;
19         x0 = x2 - q * x1;
20         y0 = y2 - q * y1;
21         r2 = r1, x2 = x1, y2 = y1;
22         r1 = r0, x1 = x0, y1 = y0;
23     }
24     ll g = r2;
25     x = x2, y = y2;
26     if (g < 0)
27         g = -g, x = -x, y = -y; // make sure g > 0
28     // for debugging (in case you think you might have bugs)
29     // assert (g == a * x + b * y);
30     // assert (g == __gcd(abs(a),abs(b)));

```

```

31     return g;
32 }
33
34 // =====
35 // CRT for a system of 2 modular linear equations
36 // =====
37 // We want to find X such that:
38 //   1) x = r1 (mod m1)
39 //   2) x = r2 (mod m2)
40 // The solution is given by:
41 //   sol = r1 + m1 * (r2-r1)/g * x' (mod LCM(m1,m2))
42 // where x' comes from
43 //   m1 * x' + m2 * y' = g = GCD(m1,m2)
44 // where x' and y' are the values found by extended euclidean
45 //   algorithm (gcdext)
46 // Useful references:
47 //   https://codeforces.com/blog/entry/61290
48 //   https://forthright48.com/chinese-remainder-theorem-part-1-coprime-
49 //     moduli
50 //   https://forthright48.com/chinese-remainder-theorem-part-2-non-
51 //     coprime-moduli
52 // ** Note: this solution works if lcm(m1,m2) fits in a long long (64
53 //     bits)
54 pair<ll, ll> CRT(ll r1, ll m1, ll r2, ll m2)
55 {
56     ll g, x, y;
57     g = gcdext(m1, m2, x, y);
58     if ((r1 - r2) % g != 0)
59         return {-1, -1}; // no solution
60     ll z = m2 / g;
61     ll lcm = m1 * z;
62     ll sol = add(mod(r1, lcm), m1 * mul(mod(x, z), mod((r2 - r1) / g, z),
63         , z), lcm);
64     // for debugging (in case you think you might have bugs)
65     // assert (0 <= sol and sol < lcm);
66     // assert (sol % m1 == r1 % m1);
67     // assert (sol % m2 == r2 % m2);
68     return {sol, lcm}; // solution + lcm(m1,m2)
69 }
70
71 // =====
72 // CRT for a system of N modular linear equations
73 // =====
74 // Args:
75 //   r = array of remainders
76 //   m = array of modules
77 //   n = length of both arrays
78 // Output:
79 //   a pair {X, lcm} where X is the solution of the sytemm
80 //   X = r[i] (mod m[i]) for i = 0 ... n-1
81 //   and lcm = LCM(m[0], m[1], ..., m[n-1])
82 //   if there is no solution, the output is {-1, -1}
83 // ** Note: this solution works if LCM(m[0],...,m[n-1]) fits in a long
84 //     long (64 bits)
85 pair<ll, ll> CRT(ll *r, ll *m, int n)

```

```

80 {
81     ll r1 = r[0], m1 = m[0];
82     repx(i, 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
92             , z), z), lcm);
93         r1 = sol;
94         m1 = lcm;
95     }
96     // for debugging (in case you think you might have bugs)
97     // assert (0 <= r1 and r1 < m1);
98     // rep(i, n) assert (r1 % m[i] == r[i]);
99     return {r1, m1};

```

7.3 Primality Checks

7.3.1 Miller Rabin

```

1  #include ".././././headers/headers.h"
2
3  ll mulmod(ull a, ull b, ull c)
4  {
5      ull x = 0, y = a % c;
6      while (b)
7      {
8          if (b & 1)
9              x = (x + y) % c;
10             y = (y << 1) % c;
11             b >>= 1;
12         }
13         return x % c;
14     }
15
16     ll fastPow(ll x, ll n, ll MOD)
17     {
18         ll ret = 1;
19         while (n)
20         {
21             if (n & 1)
22                 ret = mulmod(ret, x, MOD);
23             x = mulmod(x, x, MOD);
24             n >>= 1;
25         }
26         return ret;
27     }
28
29     bool isPrime(ll n)

```

```

30 {
31     vi a = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
32
33     if (binary_search(a.begin(), a.end(), n))
34         return true;
35
36     if ((n & 1) == 0)
37         return false;
38
39     int s = 0;
40     for (ll m = n - 1; !(m & 1); ++s, m >>= 1)
41         ;
42
43     int d = (n - 1) / (1 << s);
44
45     for (int i = 0; i < 7; i++)
46     {
47         ll fp = fastPow(a[i], d, n);
48         bool comp = (fp != 1);
49         if (comp)
50             for (int j = 0; j < s; j++)
51             {
52                 if (fp == n - 1)
53                 {
54                     comp = false;
55                     break;
56                 }
57             }
58             fp = mulmod(fp, fp, n);
59         }
60         if (comp)
61             return false;
62     }
63     return true;
64 }

```

7.3.2 Sieve of Eratosthenes

```

1  #include ".././././headers/headers.h"
2
3  // O(n log log n)
4  vi sieve(int n)
5  {
6      vi primes;
7
8      vector<bool> is_prime(n + 1, true);
9      int limit = (int)floor(sqrt(n));
10     repx(i, 2, limit + 1) if (is_prime[i]) for (int j = i * i; j <= n; j
11         += i)
12         is_prime[j] = false;
13
14     repx(i, 2, n + 1) if (is_prime[i]) primes.eb(i);
15
16     return primes;

```


7.3.3 trialDivision

```

1  #include "../headers/headers.h"
2
3  // O(sqrt(n)/log(sqrt(n))+log(n))
4  vi trialDivision(int n, vi &primes)
5  {
6      vi factors;
7      for (auto p : primes)
8      {
9          if (p * p > n)
10             break;
11             while (n % p == 0)
12             {
13                 primes.pb(p);
14                 if ((n /= p) == 1)
15                     return factors;
16             }
17     }
18     if (n > 1)
19         factors.pb(n);
20
21     return factors;
22 }

```

7.4 Others

7.4.1 Polynomials

```

1  #include "../headers/headers.h"
2
3  template <class T>
4  class Pol
5  {
6  private:
7      vector<T> cofs;
8      int n;
9
10 public:
11     Pol(vector<T> cofs) : cofs(cofs)
12     {
13         this->n = cofs.size() - 1;
14     }
15
16     Pol<T> operator+(const Pol<T> &o)
17     {
18         vector<T> n_cofs;
19         if (n > o.n)
20         {
21             n_cofs = cofs;
22             rep(i, o.n + 1)
23             {
24                 n_cofs[i] += o.cofs[i];
25             }
26         }

```

```

27     else
28     {
29         n_cofs = o.cofs;
30         rep(i, n + 1)
31         {
32             n_cofs[i] += cofs[i];
33         }
34     }
35     return Pol(n_cofs);
36 }
37
38 Pol<T> operator-(const Pol<T> &o)
39 {
40     vector<T> n_cofs;
41     if (n > o.n)
42     {
43         n_cofs = cofs;
44         rep(i, o.n + 1)
45         {
46             n_cofs[i] -= o.cofs[i];
47         }
48     }
49     else
50     {
51         n_cofs = o.cofs;
52         rep(i, n + 1)
53         {
54             n_cofs[i] *= -1;
55             n_cofs[i] += cofs[i];
56         }
57     }
58     return Pol(n_cofs);
59 }
60
61 Pol<T> operator*(const Pol<T> &o) //Use Fast Fourier Transform when
62     we implement it
63 {
64     vector<T> n_cofs(n + o.n + 1);
65     rep(i, n + 1)
66     {
67         rep(j, o.n + 1)
68         {
69             n_cofs[i + j] += cofs[i] * o.cofs[j];
70         }
71     }
72     return Pol(n_cofs);
73 }
74
75 Pol<T> operator*(const T &o)
76 {
77     vector<T> n_cofs = cofs;
78     for (auto &cof : n_cofs)
79     {
80         cof *= o;
81     }

```

```

81     return Pol(n_cofs);
82 }
83
84 double operator()(double x)
85 {
86     double ans = 0;
87     double temp = 1;
88     for (auto cof : cofs)
89     {
90         ans += (double)cof * temp;
91         temp *= x;
92     }
93     return ans;
94 }
95
96 Pol<T> integrate()
97 {
98     vector<T> n_cofs(n + 2);
99     rep(x(i, 1, n_cofs.size())
100     {
101         n_cofs[i] = cofs[i - 1] / T(i);
102     }
103     return Pol<T>(n_cofs);
104 }
105
106 double integrate(T a, T b)
107 {
108     Pol<T> temp = integrate();
109     return temp(b) - temp(a);
110 }
111
112 friend ostream &operator<<(ostream &str, const Pol &a);
113 };
114
115 ostream &operator<<(ostream &strm, const Pol<double> &a)
116 {
117     bool flag = false;
118     rep(i, a.n + 1)
119     {
120         if (a.cofs[i] == 0)
121             continue;
122
123         if (flag)
124             if (a.cofs[i] > 0)
125                 strm << " + ";
126             else
127                 strm << " - ";
128         else
129             flag = true;
130         if (i > 1)
131         {
132             if (abs(a.cofs[i]) != 1)
133                 strm << abs(a.cofs[i]);
134             strm << "x" << i;
135         }

```

```

136     else if (i == 1)
137     {
138         if (abs(a.cofs[i]) != 1)
139             strm << abs(a.cofs[i]);
140         strm << "x";
141     }
142     else
143     {
144         strm << a.cofs[i];
145     }
146 }
147 return strm;
148 }

```

7.4.2 Factorial Factorization

```

1 #include "../headers/headers.h"
2
3 // O(n)
4 umap<int, int> factorialFactorization(int n, vi &primes)
5 {
6     umap<int, int> p2e;
7     for (auto p : primes)
8     {
9         if (p > n)
10             break;
11         int e = 0;
12         int tmp = n;
13         while ((tmp /= p) > 0)
14             e += tmp;
15         if (e > 0)
16             p2e[p] = e;
17     }
18     return p2e;
19 }

```

8 Geometry

8.1 Vectors/Points

```

1 #include "../headers/headers.h"
2
3 const double PI = acos(-1);
4
5 struct vector2D
6 {
7     double x, y;
8
9     vector2D &operator+=(const vector2D &o)
10     {
11         this->x += o.x;
12         this->y += o.y;
13         return *this;
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 }

```

```

70 // return (b - a).cross(c - a); // alternatively, using struct
    function
71 }
72
73 // calculates the cross product (b - a) x (c - a)
74 // and returns orientation:
75 // LEFT (1):      c is to the left of ray (a -> b)
76 // RIGHT (-1):    c is to the right of ray (a -> b)
77 // COLLINEAR (0): c is collinear to ray (a -> b)
78 // inspired by: https://www.geeksforgeeks.org/orientation-3-ordered-
    points/
79 int orientation(vector2D &a, vector2D &b, vector2D &c)
80 {
81     ll tmp = cross(a, b, c);
82     return tmp < 0 ? -1 : tmp == 0 ? 0 : 1; // sign
83 }
84
85 /* ===== */
86 /* Check if a segment is below another segment (wrt a ray) */
87 /* ===== */
88 // i.e: check if a segment is intersected by the ray first
89 // Assumptions:
90 // 1) for each segment:
91 // p1 should be LEFT (or COLLINEAR) and p2 should be RIGHT (or
    COLLINEAR) wrt ray
92 // 2) segments do not intersect each other
93 // 3) segments are not collinear to the ray
94 // 4) the ray intersects all segments
95 struct Segment
96 {
97     vector2D p1, p2;
98 };
99 #define MAXN (int)1e6 //Example
100 Segment segments[MAXN]; // array of line segments
101 bool is_si_below_sj(int i, int j)
102 { // custom comparator based on cross product
103     Segment &si = segments[i];
104     Segment &sj = segments[j];
105     return (si.p1.x >= sj.p1.x) ? cross(si.p1, sj.p2, sj.p1) > 0 : cross
        (sj.p1, si.p1, si.p2) > 0;
106 }
107 // this can be used to keep a set of segments ordered by order of
    intersection
108 // by the ray, for example, active segments during a SWEEP LINE
109 set<int, bool (*) (int, int)> active_segments(is_si_below_sj); // ordered
    set
110
111 /* ===== */
112 /* Rectangle Intersection */
113 /* ===== */
114 bool do_rectangles_intersect(vector2D &d1l, vector2D &ur1, vector2D &d1l2
    , vector2D &ur2)
115 {
116     return max(d1l.x, d1l2.x) <= min(ur1.x, ur2.x) && max(d1l.y, d1l2.y)
        <= min(ur1.y, ur2.y);

```

```

117 }
118
119 /* ===== */
120 /* Line Segment Intersection */
121 /* ===== */
122 // returns whether segments p1q1 and p2q2 intersect, inspired by:
123 // https://www.geeksforgeeks.org/check-if-two-given-line-segments-
124 // intersect/
125 bool do_segments_intersect(vector2D &p1, vector2D &q1, vector2D &p2,
126 vector2D &q2)
127 {
128     int o11 = orientation(p1, q1, p2);
129     int o12 = orientation(p1, q1, q2);
130     int o21 = orientation(p2, q2, p1);
131     int o22 = orientation(p2, q2, q1);
132     if (o11 != o12 and o21 != o22) // general case -> non-collinear
133         intersection
134         return true;
135     if (o11 == o12 and o11 == 0)
136     { // particular case -> segments are collinear
137         vector2D dl1 = {min(p1.x, q1.x), min(p1.y, q1.y)};
138         vector2D ur1 = {max(p1.x, q1.x), max(p1.y, q1.y)};
139         vector2D dl2 = {min(p2.x, q2.x), min(p2.y, q2.y)};
140         vector2D ur2 = {max(p2.x, q2.x), max(p2.y, q2.y)};
141         return do_rectangles_intersect(dl1, ur1, dl2, ur2);
142     }
143     return false;
144 }
145
146 /* ===== */
147 /* Circle Intersection */
148 /* ===== */
149 struct Circle
150 {
151     double x, y, r;
152 };
153 bool is_fully_outside(double r1, double r2, double d_sqr)
154 {
155     double tmp = r1 + r2;
156     return d_sqr > tmp * tmp;
157 }
158 bool is_fully_inside(double r1, double r2, double d_sqr)
159 {
160     if (r1 > r2)
161         return false;
162     double tmp = r2 - r1;
163     return d_sqr < tmp * tmp;
164 }
165 bool do_circles_intersect(Circle &c1, Circle &c2)
166 {
167     double dx = c1.x - c2.x;
168     double dy = c1.y - c2.y;
169     double d_sqr = dx * dx + dy * dy;
170     if (is_fully_inside(c1.r, c2.r, d_sqr))
171         return false;

```

```

169     if (is_fully_inside(c2.r, c1.r, d_sqr))
170         return false;
171     if (is_fully_outside(c1.r, c2.r, d_sqr))
172         return false;
173     return true;
174 }
175
176 /* ===== */
177 /* vector2D - Line distance */
178 /* ===== */
179 // get distance between p and projection of p on line <- a - b ->
180 double point_line_dist(vector2D &p, vector2D &a, vector2D &b)
181 {
182     vector2D d = b - a;
183     double t = d.dot(p - a) / d.norm2();
184     return (a + d * t - p).norm();
185 }
186
187 /* ===== */
188 /* vector2D - Segment distance */
189 /* ===== */
190 // get distance between p and truncated projection of p on segment a ->
191 // b
192 double point_segment_dist(vector2D &p, vector2D &a, vector2D &b)
193 {
194     if (a == b)
195         return (p - a).norm(); // segment is a single vector2D
196     vector2D d = b - a; // direction
197     double t = d.dot(p - a) / d.norm2();
198     if (t <= 0)
199         return (p - a).norm(); // truncate left
200     if (t >= 1)
201         return (p - b).norm(); // truncate right
202     return (a + d * t - p).norm();
203 }
204
205 /* ===== */
206 /* Straight Line Hashing (integer coords) */
207 /* ===== */
208 // task: given 2 points p1, p2 with integer coordinates, output a unique
209 // representation {a,b,c} such that a*x + b*y + c = 0 is the equation
210 // of the straight line defined by p1, p2. This representation must be
211 // unique for each straight line, no matter which p1 and p2 are sampled.
212 struct Line
213 {
214     int a, b, c;
215 };
216 int gcd(int a, int b)
217 { // greatest common divisor
218     a = abs(a);
219     b = abs(b);
220     while (b)
221     {
222         int c = a;
223         a = b;

```

```

223     b = c % b;
224 }
225 return a;
226 }
227 Line getLine(vector2D p1, vector2D p2)
228 {
229     int a = p1.y - p2.y;
230     int b = p2.x - p1.x;
231     int c = p1.x * (p2.y - p1.y) - p1.y * (p2.x - p1.x);
232     int sgn = (a < 0 || (a == 0 && b < 0)) ? -1 : 1;
233     int f = gcd(a, gcd(b, c)) * sgn;
234     a /= f;
235     b /= f;
236     c /= f;
237     return {a, b, c};
238 }

```

8.2 Calculate Areas

8.2.1 Integration via Simpson's Method

```

1  #include "../headers/headers.h"
2
3  //O(Evaluate f)=g(f)
4  //Numerical Integration of f in interval [a,b]
5  double simpsons_rule(function<double(double)> f, double a, double b)
6  {
7      double c = (a + b) / 2;
8      double h3 = abs(b - a) / 6;
9      return h3 * (f(a) + 4 * f(c) + f(b));
10 }
11
12 //O(n g(f))
13 //Integrate f between a and b, using intervals of length (b-a)/n
14 double simpsons_rule(function<double(double)> f, double a, double b, int
15     n)
16 {
17     //n sets the precision for the result
18     double ans = 0;
19     double step = 0, h = (b - a) / n;
20     rep(i, n)
21     {
22         ans += simpsons_rule(f, step, step + h);
23         step += h;
24     }
25     return ans;
26 }

```

8.2.2 Green's Theorem

```

1  #include "../headers/headers.h"
2
3  // O(1)
4  // Circle Arc
5  double arc(double theta, double phi)

```

```

6  {
7  }
8
9  // O(1)
10 // Line
11 double line(double x1, double y1, double x2, double y2)
12 {
13 }

```

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 Trie

```

1  #include "../headers/headers.h"
2
3  /* Implementation from: https://pastebin.com/fyqsH65k */
4  struct TrieNode
5  {
6      int leaf; // number of words that end on a TrieNode (allows for
7          duplicate words)
8      int height; // height of a TrieNode, root starts at height = 1, can
9          be changed with the default value of constructor
10     // number of words that pass through this node,
11     // ask root node for this count to find the number of entries on the
12     // whole Trie
13     // all nodes have 1 as they count the words than end on themselves (
14     // ie leaf nodes count themselves)
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         child()
26     {}
27 };
28
29 /**
30  * Complexity: O(|key| * log(k))
31  */

```

```

26 TrieNode *trie_find(TrieNode *root, const string &str)
27 {
28     TrieNode *pNode = root;
29     for (string::const_iterator key = str.begin(); key != str.end(); key
        ++))
30     {
31         if (pNode->child.find(*key) == pNode->child.end())
32             return NULL;
33         pNode = pNode->child[*key];
34     }
35     return (pNode->leaf) ? pNode : NULL; // returns only whole word
36     // return pNode; // allows to search for a suffix
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  * Complexity: O(|key| * log(k))
56  */
57 void trie_erase(TrieNode *root, const string &str)
58 {
59     TrieNode *pNode = root;
60     string::const_iterator key = str.begin();
61     for (; key != str.end(); key++)
62     {
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     {
71         if (pNode->child.size() > 0 || pNode->leaf)
72             break;
73         pNode = pNode->parent, key--;
74         pNode->child.erase(*key);
75         pNode->count -= 1;
76     }
77 }

```

```

79 | }

```

9.2 KMP

```

1  #include "../headers/headers.h"
2
3  vi prefix(string &S)
4  {
5      vector<int> p(S.size());
6      p[0] = 0;
7      for (int i = 1; i < S.size(); ++i)
8      {
9          p[i] = p[i - 1];
10         while (p[i] > 0 && S[p[i]] != S[i])
11             p[i] = p[p[i] - 1];
12         if (S[p[i]] == S[i])
13             p[i]++;
14     }
15     return p;
16 }
17
18 vi KMP(string &P, string &S)
19 {
20     vector<int> pi = prefix(P);
21     vi matches;
22     int n = S.length(), m = P.length();
23     int j = 0, ans = 0;
24     for (int i = 0; i < n; ++i)
25     {
26         while (j > 0 && S[i] != P[j])
27             j = pi[j - 1];
28         if (S[i] == P[j])
29             ++j;
30
31         if (j == P.length())
32         {
33             /* This is where KMP found a match
34              * we can calculate its position on S by using i - m + 1
35              * or we can simply count it
36              */
37             ans += 1; // count the number of matches
38             matches.pb(i - m + 1); // store the position of those
39                                 // matches
40             // return; we can return on the first match if needed
41             // this must stay the same
42             j = pi[j - 1];
43         }
44     }
45     return matches; // can be modified to return number of matches or
                     // location

```