

Contents

1	Info About Memory and Time Limits	2	7.1	Useful Data	16
2	C++ Cheat Sheet	2	7.2	Modular Arithmetic	16
2.1	Headers	2	7.2.1	Chinese Remainder Theorem	16
2.2	Cheat Sheet	2	7.2.2	Binomial Coefficients mod m	17
3	General Algorithms	7	7.3	Primality Checks	18
3.1	Search	7	7.3.1	Miller Rabin	18
3.1.1	Binary Search	7	7.3.2	Sieve of Eratosthenes	19
3.1.2	Ternary Search	8	7.3.3	trialDivision	19
3.2	Brute Force	8	7.4	Others	19
4	Data Structures	8	7.4.1	Polynomials	19
4.1	Segment Tree	8	7.4.2	Factorial Factorization	20
4.1.1	Lazy	8	8	Geometry	20
4.1.2	Iterative	9	8.1	Vectors/Points	20
4.2	Fenwick Tree/BIT	10	8.2	Calculate Areas	22
4.2.1	1D	10	8.2.1	Integration via Simpson's Method	22
4.2.2	2D	10	8.2.2	Green's Theorem	22
4.3	Wavelet Tree	10	8.3	Convex Hull	22
5	Dynamic Programming	11	8.4	Pick's Theorem	23
5.1	Knapsack	11	9	Strings	23
5.2	Matrix Chain Multiplication	12	9.1	KMP	23
5.3	Longest Increasing Subsequence	12	9.2	Rolling Hashing	24
6	Graphs	12	9.3	Trie	24
6.1	Graph Traversal	12	9.4	Suffix Tree	25
6.1.1	Breadth First Search	12			
6.1.2	Recursive Depth First Search	12			
6.1.3	Iterative Depth First Search	12			
6.2	Shortest Path Algorithms	13			
6.2.1	Dijkstra	13			
6.2.2	Bellman Ford	13			
6.2.3	Floyd Warshall	13			
6.3	Minimum Spanning Tree (MST)	13			
6.3.1	Kruskal	13			
6.4	Lowest Common Ancestor (LCA)	14			
6.5	Max Flow	14			
6.6	Others	15			
6.6.1	Diameter of a tree	15			
7	Mathematics	16			

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 verbose 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 #define umap unordered_map
35 #define uset unordered_set
36
37 //Debugs single variables (e.g. int, string)
38 #define debugx(x) cerr << #x << ": " << x << endl
39 //Debugs Iterables (e.g. vi, uset<int>)
40 #define debugv(v) \
41     cerr << #v << ":\n"; \
42     for (auto e : v) \
43     { \
44         cerr << " " << e; \
45     } \
46     cerr << endl
47 //Debugs Iterables of Iterables (e.g. graph, umap<int, umap<int, int>)
48 #define debugm(m) \
49     cerr << #m << endl; \
50     for (auto v : m) \
51     { \
52         for (auto e : v) \
53             cerr << " " << e; \
54         cerr << endl; \
55     }
56 #define print(x) copy(x.begin(), x.end(), ostream_iterator<int>(cout, "")), cout << endl
57
58 //Outputs generic pairs through streams (including cerr and cout)
59 template <typename T1, typename T2>
60 ostream &operator<<(ostream &os, const pair<T1, T2> &p)
61 {
62     os << '(' << p.first << ', ' << p.second << ')';
63     return os;
64 }
65
66 #endif

```

2.2 Cheat Sheet

```

1 /* ===== */
2 /* Reading from stdin */
3 /* ===== */
4 // With scanf
5 scanf("%d", &a); // int
6 scanf("%x", &a); // int in hexadecimal
7 scanf("%llx", &a); // long long in hexadecimal
8 scanf("%lld", &a); // long long int
9 scanf("%c", &c); // char
10 scanf("%s", buffer); // string without whitespaces
11 scanf("%f", &f); // float

```

```

12 scanf("%lf", &d);           // double
13 scanf("%d %s %d", &a, &b); // * = consume but skip
14
15 // read until EOL
16 // - EOL not included in buffer
17 // - EOL is not consumed
18 // - nothing is written into buffer if EOF is found
19 scanf(" %[^\n]", buffer);
20
21 // reading until EOL or EOF
22 // - EOL not included in buffer
23 // - EOL is consumed
24 // - works with EOF
25 char *output = gets(buffer);
26 if (feof(stind)) {
27 } // EOF file found
28 if (output == buffer) {
29 } // succesful read
30 if (output == NULL) {
31 } // EOF found without previous chars found
32 // example
33 while (gets(buffer) != NULL) {
34     puts(buffer);
35     if (feof(stdin)) {
36         break;
37     }
38 }
39
40 // read single char
41 getchar();
42 while (true) {
43     c = getchar();
44     if (c == EOF || c == '\n')
45         break;
46 }
47
48 /* ===== */
49 /* Printing to stdout */
50 /* ===== */
51 // With printf
52 printf("%d", a);           // int
53 printf("%u", a);           // unsigned int
54 printf("%lld", a);         // long long int
55 printf("%llu", a);         // unsigned long long int
56 printf("%c", c);           // char
57 printf("%s", buffer);      // string until \0
58 printf("%f", f);           // float
59 printf("%lf", d);          // double
60 printf("%0*.f", x, y, f);   // padding = 0, width = x, decimals = y
61 printf("%.5s)\n", buffer); // print at most the first five characters
                             // (safe to
                             // use on short strings)
62
63 // print at most first n characters (safe)
64 printf("%.s)\n", n,

```

```

66     buffer); // make sure that n is integer (with long long I had
              // problems)
67 // string + \n
68 puts(buffer);
69
70 /* ===== */
71 /* Reading from c string */
72 /* ===== */
73
74 // same as scanf but reading from s
75 int sscanf(const char *s, const char *format, ...);
76
77 /* ===== */
78 /* Printing to c string */
79 /* ===== */
80 // Same as printf but writing into str, the number of characters is
    // returned
81 // or negative if there is failure
82 int sprintf(char *str, const char *format, ...);
83 // example:
84 int n = sprintf(buffer, "%d plus %d is %d", a, b, a + b);
85 printf("[%s] is a string %d chars long\n", buffer, n);
86
87 /* ===== */
88 /* Peek last char of stdin */
89 /* ===== */
90 bool peekAndCheck(char c) {
91     char c2 = getchar();
92     ungetc(c2, stdin); // return char to stdin
93     return c == c2;
94 }
95
96 /* ===== */
97 /* Reading from cin */
98 /* ===== */
99 // reading a line of unknown length
100 string line;
101 getline(cin, line);
102 while (getline(cin, line)) {
103 }
104
105 // Optimizations with cin/cout
106 ios::sync_with_stdio(0);
107 cin.tie(0);
108 cout.tie(0);
109
110 // Fix precision on cout
111 cout.setf(ios::fixed);
112 cout.precision(4); // e.g. 1.000
113
114 /* ===== */
115 /* USING PAIRS AND TUPLES */
116 /* ===== */
117 // ii = pair<int,int>
118 ii p(5, 5);

```

```

119 ii p = make_pair(5, 5) ii p = {5, 5};
120 int x = p.first, y = p.second;
121 // iii = tuple<int,int,int>
122 iii t(5, 5, 5);
123 tie(x, y, z) = t;
124 tie(x, y, z) = make_tuple(5, 5, 5);
125 get<0>(t)++;
126 get<1>(t)--;
127
128 /* ===== */
129 /* CONVERTING FROM STRING TO NUMBERS */
130 /* ===== */
131 //-----
132 // string to int
133 // option #1:
134 int atoi(const char *str);
135 // option #2:
136 sscanf(string, "%d", &i);
137 //-----
138 // string to long int:
139 // option #1:
140 long int strtol(const char *str, char **endptr, int base);
141 // it only works skipping whitespaces, so make sure your numbers
142 // are surrounded by whitespaces only
143 // Example:
144 char szNumbers[] = "2001 60c0c0 -1101110100110100100000 0x6ffff";
145 char *pEnd;
146 long int li1, li2, li3, li4;
147 li1 = strtol(szNumbers, &pEnd, 10);
148 li2 = strtol(pEnd, &pEnd, 16);
149 li3 = strtol(pEnd, &pEnd, 2);
150 li4 = strtol(pEnd, NULL, 0);
151 printf("The decimal equivalents are: %ld, %ld, %ld and %ld.\n", li1, li2
    , li3,
    li4);
152 // option #2:
153 long int atol(const char *str);
154 // option #3:
155 sscanf(string, "%ld", &l);
156 //-----
157 // string to long long int:
158 // option #1:
159 long long int strtoll(const char *str, char **endptr, int base);
160 // option #2:
161 sscanf(string, "%lld", &l);
162 //-----
163 // string to double:
164 // option #1:
165 double strtod(const char *str, char **endptr); // similar to strtol
166 // option #2:
167 double atof(const char *str);
168 // option #3:
169 sscanf(string, "%lf", &d);
170
171
172 /* ===== */

```

```

173 /* C STRING UTILITY FUNCTIONS */
174 /* ===== */
175 int strcmp(const char *str1, const char *str2); // (-1,0,1)
176 int memcmp(const void *ptr1, const void *ptr2, size_t num); // (-1,0,1)
177 void *memcpy(void *destination, const void *source, size_t num);
178
179 /* ===== */
180 /* C++ STRING UTILITY FUNCTIONS */
181 /* ===== */
182 // read tokens from string
183 string s = "tok1 tok2 tok3";
184 string tok;
185 stringstream ss(s);
186 while (getline(ss, tok, ' '))
187     printf("tok = %s\n", tok.c_str());
188
189 // split a string by a single char delimiter
190 void split(const string &s, char delim, vector<string> &elems) {
191     stringstream ss(s);
192     string item;
193     while (getline(ss, item, delim))
194         elems.push_back(item);
195 }
196
197 // find index of string or char within string
198 string str = "random";
199 std::size_t pos = str.find("ra");
200 std::size_t pos = str.find('m');
201 if (pos == string::npos) // not found
202
203     // substrings
204     string subs = str.substr(pos, length);
205 string subs = str.substr(pos); // default: to the end of the string
206
207 // std::string from cstring's substring
208 const char *s = "bla1 bla2";
209 int offset = 5, len = 4;
210 string subs(s + offset, len); // bla2
211
212 // -----
213 // string comparisons
214 int compare(const string &str) const;
215 int compare(size_t pos, size_t len, const string &str) const;
216 int compare(size_t pos, size_t len, const string &str, size_t subpos,
    size_t sublen) const;
217 int compare(const char *s) const;
218 int compare(size_t pos, size_t len, const char *s) const;
219
220
221 // examples
222 // 1) check string begins with another string
223 string prefix = "prefix";
224 string word = "prefix suffix";
225 word.compare(0, prefix.size(), prefix);
226
227 /* ===== */

```

```

228  /* OPERATOR OVERLOADING */
229  /* ===== */
230
231  //-----
232  // method #1: inside struct
233  struct Point {
234      int x, y;
235      bool operator<(const Point &p) const {
236          if (x != p.x)
237              return x < p.x;
238          return y < p.y;
239      }
240      bool operator>(const Point &p) const {
241          if (x != p.x)
242              return x > p.x;
243          return y > p.y;
244      }
245      bool operator==(const Point &p) const { return x == p.x && y == p.y; }
246  };
247
248  //-----
249  // method #2: outside struct
250  struct Point {
251      int x, y;
252  };
253  bool operator<(const Point &a, const Point &b) {
254      if (a.x != b.x)
255          return a.x < b.x;
256      return a.y < b.y;
257  }
258  bool operator>(const Point &a, const Point &b) {
259      if (a.x != b.x)
260          return a.x > b.x;
261      return a.y > b.y;
262  }
263  bool operator==(const Point &a, const Point &b) {
264      return a.x == b.x && a.y == b.y;
265  }
266
267  // Note: if you overload the < operator for a custom struct,
268  // then you can use that struct with any library function
269  // or data structure that requires the < operator
270  // Examples:
271  priority_queue<Point> pq;
272  vector<Point> pts;
273  sort(pts.begin(), pts.end());
274  lower_bound(pts.begin(), pts.end(), {1, 2});
275  upper_bound(pts.begin(), pts.end(), {1, 2});
276  set<Point> pt_set;
277  map<Point, int> pt_map;
278
279  /* ===== */
280  /* CUSTOM COMPARISONS */
281  /* ===== */
282  // method #1: operator overloading

```

```

283  // method #2: custom comparison function
284  bool cmp(const Point &a, const Point &b) {
285      if (a.x != b.x)
286          return a.x < b.x;
287      return a.y < b.y;
288  }
289  // method #3: functor
290  struct cmp {
291      bool operator()(const Point &a, const Point &b) {
292          if (a.x != b.x)
293              return a.x < b.x;
294          return a.y < b.y;
295      }
296  };
297  // without operator overloading, you would have to use
298  // an explicit comparison method when using library
299  // functions or data structures that require sorting
300  priority_queue<Point, vector<Point>, cmp> pq;
301  vector<Point> pts;
302  sort(pts.begin(), pts.end(), cmp);
303  lower_bound(pts.begin(), pts.end(), {1, 2}, cmp);
304  upper_bound(pts.begin(), pts.end(), {1, 2}, cmp);
305  set<Point, cmp> pt_set;
306  map<Point, int, cmp> pt_map;
307
308  /* ===== */
309  /* VECTOR UTILITY FUNCTIONS */
310  /* ===== */
311  vector<int> myvector;
312  myvector.push_back(100);
313  myvector.pop_back(); // remove last element
314  myvector.back();     // peek reference to last element
315  myvector.front();    // peek reference to first element
316  myvector.clear();    // remove all elements
317  // sorting a vector
318  vector<int> foo;
319  sort(foo.begin(), foo.end());
320  sort(foo.begin(), foo.end(), std::less<int>()); // increasing
321  sort(foo.begin(), foo.end(), std::greater<int>()); // decreasing
322
323  /* ===== */
324  /* SET UTILITY FUNCTIONS */
325  /* ===== */
326  set<int> myset;
327  myset.begin(); // iterator to first element
328  myset.end();   // iterator to after last element
329  myset.rbegin(); // iterator to last element
330  myset.rend();  // iterator to before first element
331  for (auto it = myset.begin(); it != myset.end(); ++it) {
332      do_something(*it);
333  } // left -> right
334  for (auto it = myset.rbegin(); it != myset.rend(); ++it) {
335      do_something(*it);
336  } // right -> left
337  for (auto &i : myset) {

```

```

338     do_something(i);
339 } // left->right shortcut
340 auto ret = myset.insert(
341     5); // ret.first = iterator, ret.second = boolean (inserted / not
        inserted)
342 int count = myset.erase(5); // count = how many items were erased
343 if (!myset.empty()) {
344 }
345 // custom comparator 1: functor
346 struct cmp {
347     bool operator()(int i, int j) { return i > j; }
348 };
349 set<int, cmp> myset;
350 // custom comparator 2: function
351 bool cmp(int i, int j) { return i > j; }
352 set<int, bool (*)(int, int)> myset(cmp);
353
354 /* ===== */
355 /* MAP UTILITY FUNCTIONS */
356 /* ===== */
357 struct Point {
358     int x, y;
359 };
360 bool operator<(const Point &a, const Point &b) {
361     return a.x < b.x || (a.x == b.x && a.y < b.y);
362 }
363 map<Point, int> ptcounts;
364
365 // -----
366 // inserting into map
367
368 // method #1: operator[]
369 // it overwrites the value if the key already exists
370 ptcounts[{1, 2}] = 1;
371
372 // method #2: .insert(pair<key, value>)
373 // it returns a pair { iterator(key, value) , bool }
374 // if the key already exists, it doesn't overwrite the value
375 void update_count(Point &p) {
376     auto ret = ptcounts.emplace(p, 1);
377     // auto ret = ptcounts.insert(make_pair(p, 1)); //
378     if (!ret.second)
379         ret.first->second++;
380 }
381
382 // -----
383 // generating ids with map
384 int get_id(string &name) {
385     static int id = 0;
386     static map<string, int> name2id;
387     auto it = name2id.find(name);
388     if (it == name2id.end())
389         return name2id[name] = id++;
390     return it->second;
391 }

```

```

392
393 /* ===== */
394 /* BITSET UTILITY FUNCTIONS */
395 /* ===== */
396 bitset<4> foo; // 0000
397 foo.size();    // 4
398 foo.set();     // 1111
399 foo.set(1, 0); // 1011
400 foo.test(1);   // false
401 foo.set(1);    // 1111
402 foo.test(1);   // true
403
404 /* ===== */
405 /* RANDOM INTEGERS */
406 /* ===== */
407 #include <cstdlib>
408 #include <ctime>
409 srand(time(NULL));
410 int x = rand() % 100; // 0-99
411 int randBetween(int a, int b) { // a-b
412     return a + (rand() % (1 + b - a));
413 }
414
415 /* ===== */
416 /* CLIMITS */
417 /* ===== */
418 #include <climits>
419 INT_MIN
420 INT_MAX
421 UINT_MAX
422 LONG_MIN
423 LONG_MAX
424 ULONG_MAX
425 LLONG_MIN
426 LLONG_MAX
427 ULLONG_MAX
428
429 /* ===== */
430 /* Bitwise Tricks */
431 /* ===== */
432
433 // amount of one-bits in number
434 int __builtin_popcount(int x);
435 int __builtin_popcountl(long x);
436 int __builtin_popcountll(long long x);
437
438 // amount of leading zeros in number
439 int __builtin_clz(int x);
440 int __builtin_clzl(long x);
441 int __builtin_clzll(long long x);
442
443 // binary length of non-negative number
444 int bitlen(int x) { return sizeof(x) * 8 - __builtin_clz(x); }
445 int bitlen(ll x) { return sizeof(x) * 8 - __builtin_clzll(x); }
446

```

```

447 // index of most significant bit
448 int log2(int x) { return sizeof(x) * 8 - __builtin_clz(x) - 1; }
449 int log2(ll x) { return sizeof(x) * 8 - __builtin_clzll(x) - 1; }
450
451 // reverse the bits of an integer
452 int reverse_bits(int x) {
453     int v = 0;
454     while (x)
455         v <<= 1, v |= x & 1, x >>= 1;
456     return v;
457 }
458
459 // get string binary representation of an integer
460 string bitstring(int x) {
461     int len = sizeof(x) * 8 - __builtin_clz(x);
462     if (len == 0)
463         return "0";
464
465     char buff[len + 1];
466     buff[len] = '\0';
467     for (int i = len - 1; i >= 0; --i, x >>= 1)
468         buff[i] = (char)('0' + (x & 1));
469     return string(buff);
470 }
471
472 /* ===== */
473 /* Hexadecimal Tricks */
474 /* ===== */
475
476 // get string hex representation of an integer
477 string to_hex(int num) {
478     static char buff[100];
479     static const char *hexdigits = "0123456789abcdef";
480     buff[99] = '\0';
481     int i = 98;
482     do {
483         buff[i--] = hexdigits[num & 0xf];
484         num >>= 4;
485     } while (num);
486     return string(buff + i + 1);
487 }
488
489 // ['0'-'9' 'a'-'f'] -> [0 - 15]
490 int char_to_digit(char c) {
491     if ('0' <= c && c <= '9')
492         return c - '0';
493     return 10 + c - 'a';
494 }
495
496 /* ===== */
497 /* Other Tricks */
498 /* ===== */
499 // swap stuff
500 int x = 1, y = 2;
501 swap(x, y);

```

```

502 /* ===== */
503 /* TIPS */
504 /* ===== */
505 // 1) do not use .emplace(x, y) if your struct doesn't have an explicit
506 // constructor
507 // instead you can use .push({x, y})
508 // 2) be careful while mixing scanf() with getline(), scanf will not
509 // consume \n
510 // unless
511 // 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
10 [start, end]
11 {
12     double left = start, right = end;
13     double mid1, mid2;
14     int reps = 80;
15     rep(_, reps)
16     {
17         mid1 = left + (right - left) / 3, mid2 = right - (right - left)
18         / 3;
19         if (compare(mid1, mid2))
20             left = mid1;
21         else
22             right = mid2;
23     }
24     return (mid1 + mid2) / 2; // * Can return -0!
25     // Tends to the right
26 }
27
28 double minTer(double start, double end) // Searches minimum of f in range
29 [start, end]
30 {
31     double left = start, right = end;
32     double mid1, mid2;
33     int reps = 80;
34     rep(_, reps)
35     {
36         mid1 = left + (right - left) / 3, mid2 = right - (right - left)
37         / 3;
38         if (not compare(mid1, mid2))
39             left = mid1;
40         else
41             right = mid2;
42     }
43     return (mid1 + mid2) / 2;
44     // Tends to the left
45 }

```

3.2 Brute Force

4 Data Structures

4.1 Segment Tree

4.1.1 Lazy

```

1 struct RSQ // Range sum query
2 {
3     static ll const neutro = 0;
4     static ll op(ll x, ll y) { return x + y; }
5     static ll lazy_op(int i, int j, ll x) { return (j - i + 1) * x; }
6 };
7
8 struct RMinQ // Range minimum query
9 {
10     static ll const neutro = 1e18;
11     static ll op(ll x, ll y) { return min(x, y); }
12     static ll lazy_op(int i, int j, ll x) { return x; }
13 };
14
15 template <class t> class SegTreeLazy {
16     vector<ll> arr, st, lazy;
17     int n;
18
19     void build(int u, int i, int j) {
20         if (i == j) {
21             st[u] = arr[i];
22             return;
23         }
24         int m = (i + j) / 2, l = u * 2 + 1, r = u * 2 + 2;
25         build(l, i, m);
26         build(r, m + 1, j);
27         st[u] = t::op(st[l], st[r]);
28     }
29
30     void propagate(int u, int i, int j, ll x) {
31         // nota, las operaciones pueden ser un and, or, ..., etc.
32         st[u] += t::lazy_op(i, j, x); // incrementar el valor (+)
33         // st[u] = t::lazy_op(i, j, x); // setear el valor
34         if (i != j) {
35             // incrementar el valor
36             lazy[u * 2 + 1] += x;
37             lazy[u * 2 + 2] += x;
38             // setear el valor
39             // lazy[u * 2 + 1] = x;
40             // lazy[u * 2 + 2] = x;
41         }
42         lazy[u] = 0;
43     }
44
45     ll query(int a, int b, int u, int i, int j) {
46         if (j < a or b < i)

```



```

47     return t::neutro;
48     int m = (i + j) / 2, l = u * 2 + 1, r = u * 2 + 2;
49     if (lazy[u])
50         propagate(u, i, j, lazy[u]);
51     if (a <= i and j <= b)
52         return st[u];
53     ll x = query(a, b, l, i, m);
54     ll y = query(a, b, r, m + 1, j);
55     return t::op(x, y);
56 }
57
58 void update(int a, int b, ll value, int u, int i, int j) {
59     int m = (i + j) / 2, l = u * 2 + 1, r = u * 2 + 2;
60     if (lazy[u])
61         propagate(u, i, j, lazy[u]);
62     if (a <= i and j <= b)
63         propagate(u, i, j, value);
64     else if (j < a or b < i)
65         return;
66     else {
67         update(a, b, value, l, i, m);
68         update(a, b, value, r, m + 1, j);
69         st[u] = t::op(st[l], st[r]);
70     }
71 }
72
73 public:
74     SegTreeLazy(vector<ll> &v) {
75         arr = v;
76         n = v.size();
77         st.resize(n * 4 + 5);
78         lazy.assign(n * 4 + 5, 0);
79         build(0, 0, n - 1);
80     }
81
82     ll query(int a, int b) { return query(a, b, 0, 0, n - 1); }
83
84     void update(int a, int b, ll value) { update(a, b, value, 0, 0, n - 1); }
85 };

```

4.1.2 Iterative

```

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

```

```

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

```

```

65 ST(vector<node> &arr) {
66     n = arr.size();
67     t.resize(n * 2);
68     copy(arr.begin(), arr.end(), t.begin() + n);
69     for (int i = n - 1; i > 0; --i)
70         t[i] = node(t[i << 1], t[i << 1 | 1]);
71 }
72
73 // 0-indexed
74 void set_point(int p, const node &value) {
75     for (t[p += n] = value; p > 1; p >>= 1)
76         t[p >> 1] = node(t[p], t[p ^ 1]);
77 }
78
79 // inclusive exclusive, 0-indexed
80 node query(int l, int r) {
81     node ans1, ansr;
82     for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
83         if (l & 1)
84             ans1 = node(ans1, t[l++]);
85         if (r & 1)
86             ansr = node(t[--r], ansr);
87     }
88     return node(ans1, ansr);
89 }
90 };

```

4.2 Fenwick Tree/BIT

4.2.1 1D

```

1
2 struct FenwickTree {
3     vector<int> FT;
4     FenwickTree(int N) { FT.resize(N + 1, 0); }
5
6     int query(int i) {
7         int ans = 0;
8         for (; i; i -= i & (-i))
9             ans += FT[i];
10        return ans;
11    }
12
13    int query(int i, int j) { return query(j) - query(i - 1); }
14
15    void update(int i, int v) {
16        int s = query(i, i); // Sets range to v?
17        for (; i < FT.size(); i += i & (-i))
18            FT[i] += v - s;
19    }
20
21    // Queries puntuales, Updates por rango
22    void update(int i, int j, int v) {
23        update(i, v);
24        update(j + 1, -v);

```

```

25 }
26 };

```

4.2.2 2D

4.3 Wavelet Tree

```

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

```

```

49     m = (l + r) / 2;
50     za = r0[u][a];
51     zb = r0[u][b];
52     u *= 2;
53     if (k <= zb - za)
54         a = za, b = zb, r = m;
55     else
56         k -= zb - za, a -= za, b -= zb, l = m + 1, ++u;
57 }
58 return r;
59 }
60
61 // counts numbers in [x,y] in positions [a,b]
62 int range(int x, int y, int a, int b) {
63     if (y < x or b <= a)
64         return 0;
65     q = x;
66     w = y;
67     return range(a, b, 0, s - 1, 1);
68 }
69
70 // count occurrences of x in positions [0,k]
71 int rank(int x, int k) {
72     int l = 0, r = s - 1, u = 1, m, z;
73     while (l != r) {
74         m = (l + r) / 2;
75         z = r0[u][k];
76         u *= 2;
77         if (x <= m)
78             k = z, r = m;
79         else
80             k -= z, l = m + 1, ++u;
81     }
82     return k;
83 }
84
85 // x in [0,sigma)
86 void push_back(int x) {
87     int l = 0, r = s - 1, u = 1, m, p;
88     ++n;
89     while (l != r) {
90         m = (l + r) / 2;
91         p = (x <= m);
92         r0[u].push_back(r0[u].back() + p);
93         u *= 2;
94         if (p)
95             r = m;
96         else
97             l = m + 1, ++u;
98     }
99 }
100
101 // doesn't check if empty
102 void pop_back() {
103     int l = 0, r = s - 1, u = 1, m, p, k;

```

```

104     --n;
105     while (l != r) {
106         m = (l + r) / 2;
107         k = r0[u].size();
108         p = r0[u][k - 1] - r0[u][k - 2];
109         r0[u].pop_back();
110         u *= 2;
111         if (p)
112             r = m;
113         else
114             l = m + 1, ++u;
115     }
116 }
117
118 // swap arr[i] with arr[i+1], i in [0,n-1)
119 void swap_adj(int i) {
120     int &x = arrCopy[i], &y = arrCopy[i + 1];
121     int l = 0, r = s - 1, u = 1;
122     while (l != r) {
123         int m = (l + r) / 2, p = (x <= m), q = (y <= m);
124         if (p != q) {
125             r0[u][i + 1] ^= r0[u][i] ^ r0[u][i + 2];
126             break;
127         }
128         u *= 2;
129         if (p)
130             r = m;
131         else
132             l = m + 1, ++u;
133     }
134     swap(x, y);
135 }
136 };

```

5 Dynamic Programming

5.1 Knapsack

```

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

```

5.2 Matrix Chain Multiplication

```

1 vector<vector<ii>> DP; // Pair value, op result
2 int n; // Size of DP (i.e. i,j<n)
3 ii op(ii a, ii b) {
4     return {
5         a.first + b.first + a.second * b.second,
6         (a.second + b.second) %
7         100}; // Second part MUST be associative, first part is cost
8         function
9     }
10 ii MCM(int i, int j) {
11     if (DP[i][j].first != -1)
12         return DP[i][j];
13     int ans = 1e9; // INF
14     int res;
15     repx(k, i + 1, j) {
16         ii temp = op(MCM(i, k), MCM(k, j));
17         ans = min(ans, temp.first);
18         res = temp.second;
19     }
20     return DP[i][j] = {ans, res};
21 }
22
23 void fill() {
24     DP.assign(n, vector<ii>(n, {-1, 0}));
25     rep(i, n - 1) {
26         DP[i][i + 1].first = 1;
27     } // Pair op identity, cost (cost must be from input)
28 }

```

5.3 Longest Increasing Subsequence

```

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

```

6 Graphs

6.1 Graph Traversal

6.1.1 Breadth First Search

```

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

```

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     cout << u << '\n';
4     visited[u] = 0;
5
6     for (int v : g[u])
7         if (visited[v])
8             dfs_r(g, visited, v);
9 }

```

6.1.3 Iterative Depth First Search

```

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

```

```

15     s.emplace(v);
16     visited[v] = 0;
17 }
18 }
19 }
20 }

```

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

```

6.2.2 Bellman Ford

Edges can be negative, and it detects negative cycles

```

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

```

```

12        int v = p.first, w = p.second;
13        if (dist[v] > dist[u] + w)
14            hayCicloNegativo = true;
15    }
16
17    return hayCicloNegativo;
18 }

```

6.2.3 Floyd Warshall

Shortest path from every node to every other node

```

1 /*
2 Floyd Warshall implemenation, note that g is using an adjacency matrix
3 and not
4 an adjacency list
5 */
6 static const int INF = 1e9;
7 graph floydWarshall(const graph g) {
8     int n = g.size();
9     graph dist(n, vi(n, -1));
10
11     rep(i, n) rep(j, n) dist[i][j] = g[i][j];
12
13     rep(k, n) rep(i, n) rep(j, n) if (dist[i][k] + dist[k][j] < dist[i][j]
14         &&
15         dist[i][k] != INF && dist[k][j] !=
16         INF)
17         dist[i][j] = dist[i][k] + dist[k][j];
18
19     return dist;
20 }

```

6.3 Minimum Spanning Tree (MST)

6.3.1 Kruskal

```

1 struct edge {
2     int u, v;
3     ll w;
4     edge(int u, int v, ll w) : u(u), v(v), w(w) {}
5
6     bool operator<(const edge &o) const { return w < o.w; }
7 };
8
9 class Kruskal {
10 private:
11     ll sum;
12     vi p, rank;
13
14 public:
15     // Amount of Nodes n, and unordered vector of Edges E
16     Kruskal(int n, vector<edge> E) {

```

```

17     sum = 0;
18     p.resize(n);
19     rank.assign(n, 0);
20     rep(i, n) p[i] = i;
21     sort(E.begin(), E.end());
22     for (auto &e : E)
23         UnionSet(e.u, e.v, e.w);
24 }
25 int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
26 bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
27 void UnionSet(int i, int j, ll w) {
28     if (not isSameSet(i, j)) {
29         int x = findSet(i), y = findSet(j);
30         if (rank[x] > rank[y])
31             p[y] = x;
32         else
33             p[x] = y;
34
35         if (rank[x] == rank[y])
36             rank[y]++;
37
38         sum += w;
39     }
40 }
41 ll mst_val() { return sum; }
42 };

```

6.4 Lowest Common Ancestor (LCA)

Supports multiple trees

```

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

```

```

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

```

6.5 Max Flow

```

1 class Dinic {
2     struct edge {
3         int to, rev;
4         ll f, cap;
5     };
6
7     vector<vector<edge>> > g;
8     vector<ll> dist;
9     vector<int> q, work;
10    int n, sink;
11
12    bool bfs(int start, int finish) {
13        dist.assign(n, -1);
14        dist[start] = 0;
15        int head = 0, tail = 0;
16        q[tail++] = start;
17        while (head < tail) {
18            int u = q[head++];
19            for (const edge &e : g[u]) {
20                int v = e.to;
21                if (dist[v] == -1 and e.f < e.cap) {
22                    dist[v] = dist[u] + 1;
23                    q[tail++] = v;

```

```

24     }
25   }
26   }
27   return dist[finish] != -1;
28 }
29
30 ll dfs(int u, ll f) {
31   if (u == sink)
32     return f;
33   for (int &i = work[u]; i < (int)g[u].size(); ++i) {
34     edge &e = g[u][i];
35     int v = e.to;
36     if (e.cap <= e.f or dist[v] != dist[u] + 1)
37       continue;
38     ll df = dfs(v, min(f, e.cap - e.f));
39     if (df > 0) {
40       e.f += df;
41       g[v][e.rev].f -= df;
42       return df;
43     }
44   }
45   return 0;
46 }
47
48 public:
49   Dinic(int n) {
50     this->n = n;
51     g.resize(n);
52     dist.resize(n);
53     q.resize(n);
54   }
55
56   void add_edge(int u, int v, ll cap) {
57     edge a = {v, (int)g[v].size(), 0, cap};
58     edge b = {u, (int)g[u].size(), 0,
59               0}; // Poner cap en vez de 0 si la arista es bidireccional
60     g[u].pb(a);
61     g[v].pb(b);
62   }
63
64   ll max_flow(int source, int dest) {
65     sink = dest;
66     ll ans = 0;
67     while (bfs(source, dest)) {
68       work.assign(n, 0);
69       while (ll delta = dfs(source, LLONG_MAX))
70         ans += delta;
71     }
72     return ans;
73   }
74 };

```

6.6 Others

6.6.1 Diameter of a tree

```

1  graph Tree;
2  vi dist;
3
4  // Finds a diameter node
5  int bfs1() {
6    int n = Tree.size();
7    queue<int> q;
8
9    q.emplace(0);
10   dist[0] = 0;
11   int u;
12   while (not q.empty()) {
13     u = q.front();
14     q.pop();
15
16     for (int v : Tree[u]) {
17       if (dist[v] == -1) {
18         q.emplace(v);
19         dist[v] = dist[u] + 1;
20       }
21     }
22   }
23   return u;
24 }
25
26 // Fills the distances from one diameter node and finds another diameter
27   node
28   int bfs2() {
29     int n = Tree.size();
30     vi visited(n, 1);
31     queue<int> q;
32     int start = bfs1();
33     q.emplace(start);
34     visited[start] = 0;
35     int u;
36     while (not q.empty()) {
37       u = q.front();
38       q.pop();
39
40       for (int v : Tree[u]) {
41         if (visited[v]) {
42           q.emplace(v);
43           visited[v] = 0;
44           dist[v] = max(dist[v], dist[u] + 1);
45         }
46       }
47     }
48     return u;
49   }
50
51 // Finds the diameter
52 int bfs3() {
53   int n = Tree.size();
54   vi visited(n, 1);
55   queue<int> q;

```

```

55 int start = bfs2();
56 q.emplace(start);
57 visited[start] = 0;
58 int u;
59 while (not q.empty()) {
60     u = q.front();
61     q.pop();
62
63     for (int v : Tree[u]) {
64         if (visited[v]) {
65             q.emplace(v);
66             visited[v] = 0;
67             dist[v] = max(dist[v], dist[u] + 1);
68         }
69     }
70 }
71 return dist[u];
72 }

```

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 ll inline mod(ll x, ll m) { return ((x % m) < 0) ? x + m : x; }
3 ll inline mul(ll x, ll y, ll m) { return (x * y) % m; }
4 ll inline add(ll x, ll y, ll 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 ll gcdext(ll a, ll b, ll &x, ll &y) {
10     ll r2, x2, y2, r1, x1, y1, r0, x0, y0, q;
11     r2 = a, x2 = 1, y2 = 0;
12     r1 = b, x1 = 0, y1 = 1;
13     while (r1) {
14         q = r2 / r1;

```

```

15     r0 = r2 % r1;
16     x0 = x2 - q * x1;
17     y0 = y2 - q * y1;
18     r2 = r1, x2 = x1, y2 = y1;
19     r1 = r0, x1 = x0, y1 = y0;
20 }
21 ll g = r2;
22 x = x2, y = y2;
23 if (g < 0)
24     g = -g, x = -x, y = -y; // make sure g > 0
25 // for debugging (in case you think you might have bugs)
26 // assert (g == a * x + b * y);
27 // assert (g == __gcd(abs(a),abs(b)));
28 return g;
29 }
30
31 // =====
32 // CRT for a system of 2 modular linear equations
33 // =====
34 // We want to find X such that:
35 // 1) x = r1 (mod m1)
36 // 2) x = r2 (mod m2)
37 // The solution is given by:
38 // sol = r1 + m1 * (r2-r1)/g * x' (mod LCM(m1,m2))
39 // where x' comes from
40 // m1 * x' + m2 * y' = g = GCD(m1,m2)
41 // where x' and y' are the values found by extended euclidean
42 // algorithm
43 // (gcdext)
44 // Useful references:
45 // https://codeforces.com/blog/entry/61290
46 // https://forthright48.com/chinese-remainder-theorem-part-1-coprime-
47 // moduli
48 // https://forthright48.com/chinese-remainder-theorem-part-2-non-
49 // coprime-moduli
50 // ** Note: this solution works if lcm(m1,m2) fits in a long long (64
51 // bits)
52 pair<ll, ll> CRT(ll r1, ll m1, ll r2, ll m2) {
53     ll g, x, y;
54     g = gcdext(m1, m2, x, y);
55     if ((r1 - r2) % g != 0)
56         return {-1, -1}; // no solution
57     ll z = m2 / g;
58     ll lcm = m1 * z;
59     ll sol =
60         add(mod(r1, lcm), m1 * mul(mod(x, z), mod((r2 - r1) / g, z), z),
61             lcm);
62     // for debugging (in case you think you might have bugs)
63     // assert (0 <= sol and sol < lcm);
64     // assert (sol % m1 == r1 % m1);
65     // assert (sol % m2 == r2 % m2);
66     return {sol, lcm}; // solution + lcm(m1,m2)
67 }
68 // =====

```



```

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

```

7.2.2 Binomial Coefficients mod m

```

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

```

```

14 void preCompute() {
15     // Factor mod->MOD
16     vi primes = sieve(num);
17     ll m = num;
18     for (auto p : primes) {
19         if (p * p > m)
20             break;
21         while (m % p == 0) {
22             MOD[p]++;
23             if ((m /= p) == 1)
24                 goto next;
25         }
26     }
27     if (m > 1)
28         MOD[m] = 1;
29 next:
30     // Compute FMOD and invFMOD
31     for (auto p : MOD) {
32         int m = pow(p.first, p.second); // p^V_p(p)
33         FMOD[p.first].assign(m, 1);
34         invFMOD[p.first].assign(m, 1);
35         rep(i, 2, FMOD[p.first].size()) {
36             if (i % p.first == 0 and p.second > 1)
37                 FMOD[p.first][i] = FMOD[p.first][i - 1];
38             else
39                 FMOD[p.first][i] = mul(FMOD[p.first][i - 1], i, FMOD[p.first].
40                    size());
41
42             // Compute using Euler's theorem i.e. a^phi(m)=1 mod m with (a,m)
43             // =1
44             invFMOD[p.first][i] = modularInverse(FMOD[p.first][i], m);
45         }
46     }
47
48     // Compute nCr using Granville's theorem (prime powers)
49     // Auxiliary functions
50
51     // V_p(n!) using Legendre's theorem
52     int V(ll n, int p) {
53         int e = 0;
54         while ((n /= p) > 0)
55             e += n;
56         return e;
57     }
58
59     // f(ll n, ll p) {
60         ll m = pow(p, MOD[p]);
61         int e = n / m;
62         return mul(fastPow(FMOD[p][m - 1], e, m), FMOD[p][n % m], m);
63     }
64
65     // F(ll n, ll p) {
66         ll m = pow(p, MOD[p]);
67         ll ans = 1;

```

```

67 do {
68     ans = mul(ans, f(n, p), m);
69 } while ((n /= p) > 0);
70 return ans;
71 }
72 // Granville theorem
73 ll granville(ll n, ll r, int p) {
74     int e = V(n, p) - V(n - r, p) - V(r, p);
75     ll m = pow(p, MOD[p]);
76     if (e >= MOD[p])
77         return 0;
78     ll ans = fastPow(p, e, m);
79     ans = mul(ans, F(n, p), m);
80     ans = mul(ans, modularInverse(F(r, p), m), m);
81     ans = mul(ans, modularInverse(F(n - r, p), m), m);
82     return ans;
83 }
84
85 // Compute nCr using Lucas theorem (primes)
86 ll lucas(ll n, ll r, int p) {
87     // Trivial cases
88     if (r > n or r < 0)
89         return 0;
90     if (r == 0 or n == r)
91         return 1;
92     if (r == 1 or r == n - 1)
93         return n % p;
94     // Base case
95     if (n < p and r < p) {
96         ll ans = mul(invFMOD[p][r], invFMOD[p][n - r], p); // 1/(r!(n-r)!)
97         ans = mul(ans, FMOD[p][n], p); // n!/(r!(n-r)!)
98         return ans;
99     }
100     ll ans = lucas(n / p, r / p, p); // Recursion
101     ans = mul(ans, lucas(n % p, r % p, p), p); // False recursion
102     return ans;
103 }
104
105 // Given the prime decomposition of mod;
106 ll nCr(ll n, ll r) {
107     // Trivial cases
108     if (n < r or r < 0)
109         return 0;
110     if (r == 0 or r == n)
111         return 1;
112     if (r == 1 or r == n - 1)
113         return (n % num);
114     // Non-trivial cases
115     ll ans = 0;
116     ll mod = 1;
117     for (auto p : MOD) {
118         ll temp = pow(p.first, p.second);
119         if (p.second > 1) {

```

```

120         ans = CRT(ans, mod, granville(n, r, p.first), temp).first;
121     } else {
122         ans = CRT(ans, mod, lucas(n, r, p.first), temp).first;
123     }
124     mod *= temp;
125 }
126 return ans;
127 }

```

7.3 Primality Checks

7.3.1 Miller Rabin

```

1
2 ll mulmod(ull a, ull b, ull c) {
3     ull x = 0, y = a % c;
4     while (b) {
5         if (b & 1)
6             x = (x + y) % c;
7         y = (y << 1) % c;
8         b >>= 1;
9     }
10    return x % c;
11 }
12
13 ll fastPow(ll x, ll n, ll MOD) {
14     ll ret = 1;
15     while (n) {
16         if (n & 1)
17             ret = mulmod(ret, x, MOD);
18         x = mulmod(x, x, MOD);
19         n >>= 1;
20     }
21     return ret;
22 }
23
24 bool isPrime(ll n) {
25     vi a = {2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37};
26
27     if (binary_search(a.begin(), a.end(), n))
28         return true;
29
30     if ((n & 1) == 0)
31         return false;
32
33     int s = 0;
34     for (ll m = n - 1; !(m & 1); ++s, m >>= 1)
35         ;
36
37     int d = (n - 1) / (1 << s);
38
39     for (int i = 0; i < 7; i++) {
40         ll fp = fastPow(a[i], d, n);
41         bool comp = (fp != 1);
42         if (comp)

```

```

43     for (int j = 0; j < s; j++) {
44         if (fp == n - 1) {
45             comp = false;
46             break;
47         }
48
49         fp = mulmod(fp, fp, n);
50     }
51     if (comp)
52         return false;
53 }
54 return true;
55 }

```

7.3.2 Sieve of Eratosthenes

```

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

```

7.3.3 trialDivision

```

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

```

7.4 Others

7.4.1 Polynomials

```

1
2 template <class T> class Pol {
3 private:
4     vector<T> coefs;
5     int n;
6
7 public:
8     Pol(vector<T> coefs) : coefs(coefs) { this->n = coefs.size() - 1; }
9
10    Pol<T> operator+(const Pol<T> &o) {
11        vector<T> n_coefs;
12        if (n > o.n) {
13            n_coefs = coefs;
14            rep(i, o.n + 1) { n_coefs[i] += o.coefs[i]; }
15        } else {
16            n_coefs = o.coefs;
17            rep(i, n + 1) { n_coefs[i] += coefs[i]; }
18        }
19        return Pol(n_coefs);
20    }
21
22    Pol<T> operator-(const Pol<T> &o) {
23        vector<T> n_coefs;
24        if (n > o.n) {
25            n_coefs = coefs;
26            rep(i, o.n + 1) { n_coefs[i] -= o.coefs[i]; }
27        } else {
28            n_coefs = o.coefs;
29            rep(i, n + 1) {
30                n_coefs[i] *= -1;
31                n_coefs[i] += coefs[i];
32            }
33        }
34        return Pol(n_coefs);
35    }
36
37    Pol<T>
38    operator*(const Pol<T> &o) // Use Fast Fourier Transform when we
39                                // implement it
40    {
41        vector<T> n_coefs(n + o.n + 1);
42        rep(i, n + 1) {
43            rep(j, o.n + 1) { n_coefs[i + j] += coefs[i] * o.coefs[j]; }
44        }
45        return Pol(n_coefs);
46    }
47
48    Pol<T> operator*(const T &o) {
49        vector<T> n_coefs = coefs;
50        for (auto &cof : n_coefs) {
51            cof *= o;
52        }
53    }
54 }

```

```

51     }
52     return Pol(n_cofs);
53 }
54
55 double operator()(double x) {
56     double ans = 0;
57     double temp = 1;
58     for (auto cof : cofs) {
59         ans += (double)cof * temp;
60         temp *= x;
61     }
62     return ans;
63 }
64
65 Pol<T> integrate() {
66     vector<T> n_cofs(n + 2);
67     repx(i, 1, n_cofs.size()) { n_cofs[i] = cofs[i - 1] / T(i); }
68     return Pol<T>(n_cofs);
69 }
70
71 double integrate(T a, T b) {
72     Pol<T> temp = integrate();
73     return temp(b) - temp(a);
74 }
75
76 friend ostream &operator<<(ostream &str, const Pol &a);
77 };
78
79 ostream &operator<<(ostream &strm, const Pol<double> &a) {
80     bool flag = false;
81     rep(i, a.n + 1) {
82         if (a.cofs[i] == 0)
83             continue;
84
85         if (flag)
86             if (a.cofs[i] > 0)
87                 strm << " + ";
88             else
89                 strm << " - ";
90         else
91             flag = true;
92         if (i > 1) {
93             if (abs(a.cofs[i]) != 1)
94                 strm << abs(a.cofs[i]);
95             strm << "x^" << i;
96         } else if (i == 1) {
97             if (abs(a.cofs[i]) != 1)
98                 strm << abs(a.cofs[i]);
99             strm << "x";
100         } else {
101             strm << a.cofs[i];
102         }
103     }
104     return strm;
105 }

```

7.4.2 Factorial Factorization

```

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

```

8 Geometry

8.1 Vectors/Points

```

1 const double PI = acos(-1);
2
3 struct Point {
4     double x, y;
5
6     Point &operator+=(const Point &o) {
7         this->x += o.x;
8         this->y += o.y;
9         return *this;
10    }
11    Point &operator-=(const Point &o) {
12        this->x -= o.x;
13        this->y -= o.y;
14        return *this;
15    }
16    Point operator+(const Point &o) { return {x + o.x, y + o.y}; }
17    Point operator-(const Point &o) { return {x - o.x, y - o.y}; }
18    Point operator*(const double &o) { return {x * o, y * o}; }
19    bool operator==(const Point &o) { return x == o.x and y == o.y; }
20    double norm2() { return x * x + y * y; }
21    double norm() { return sqrt(norm2()); }
22    double dot(const Point &o) { return x * o.x + y * o.y; }
23    double cross(const Point &o) { return x * o.y - y * o.x; }
24    double angle() {
25        double angle = atan2(y, x);
26        if (angle < 0)
27            angle += 2 * PI;
28        return angle;
29    }
30
31    Point Unit() { return {x / norm(), y / norm()}; }

```

```

32 };
33 /* ===== */
34 /* Cross Product -> orientation of Point with respect to ray */
35 /* ===== */
36 // cross product (b - a) x (c - a)
37 ll cross(Point &a, Point &b, Point &c) {
38     ll dx0 = b.x - a.x, dy0 = b.y - a.y;
39     ll dx1 = c.x - a.x, dy1 = c.y - a.y;
40     return dx0 * dy1 - dx1 * dy0;
41     // return (b - a).cross(c - a); // alternatively, using struct
    function
42 }
43 // calculates the cross product (b - a) x (c - a)
44 // and returns orientation:
45 // LEFT (1):      c is to the left of ray (a -> b)
46 // RIGHT (-1):    c is to the right of ray (a -> b)
47 // COLLINEAR (0): c is collinear to ray (a -> b)
48 // inspired by: https://www.geeksforgeeks.org/orientation-3-ordered-
    points/
49 int orientation(Point &a, Point &b, Point &c) {
50     ll tmp = cross(a, b, c);
51     return tmp < 0 ? -1 : tmp == 0 ? 0 : 1; // sign
52 }
53 /* ===== */
54 /* Check if a segment is below another segment (wrt a ray) */
55 /* ===== */
56 // i.e: check if a segment is intersected by the ray first
57 // Assumptions:
58 // 1) for each segment:
59 // p1 should be LEFT (or COLLINEAR) and p2 should be RIGHT (or
    COLLINEAR) wrt
60 // ray
61 // 2) segments do not intersect each other
62 // 3) segments are not collinear to the ray
63 // 4) the ray intersects all segments
64 struct Segment {
65     Point p1, p2;
66 };
67 #define MAXN (int)1e6 // Example
68 Segment segments[MAXN]; // array of line segments
69 bool is_si_below_sj(int i, int j) { // custom comparator based on cross
    product
70     Segment &si = segments[i];
71     Segment &sj = segments[j];
72     return (si.p1.x >= sj.p1.x) ? cross(si.p1, sj.p2, sj.p1) > 0
73         : cross(sj.p1, si.p1, si.p2) > 0;
74 }
75 // this can be used to keep a set of segments ordered by order of
    intersection
76 // by the ray, for example, active segments during a SWEEP LINE
77 set<int, bool (*)>(int, int)> active_segments(is_si_below_sj); // ordered
    set
78 /* ===== */
79 /* Rectangle Intersection */
80 /* ===== */

```

```

81 bool do_rectangles_intersect(Point &dl1, Point &ur1, Point &dl2,
82     Point &ur2) {
83     return max(dl1.x, dl2.x) <= min(ur1.x, ur2.x) &&
84         max(dl1.y, dl2.y) <= min(ur1.y, ur2.y);
85 }
86 /* ===== */
87 /* Line Segment Intersection */
88 /* ===== */
89 // returns whether segments p1q1 and p2q2 intersect, inspired by:
90 // https://www.geeksforgeeks.org/check-if-two-given-line-segments-
    intersect/
91 bool do_segments_intersect(Point &p1, Point &q1, Point &p2,
92     Point &q2) {
93     int o11 = orientation(p1, q1, p2);
94     int o12 = orientation(p1, q1, q2);
95     int o21 = orientation(p2, q2, p1);
96     int o22 = orientation(p2, q2, q1);
97     if (o11 != o12 and o21 != o22) // general case -> non-collinear
        intersection
98         return true;
99     if (o11 == o12 and o11 == 0) { // particular case -> segments are
        collinear
100         Point dl1 = {min(p1.x, q1.x), min(p1.y, q1.y)};
101         Point ur1 = {max(p1.x, q1.x), max(p1.y, q1.y)};
102         Point dl2 = {min(p2.x, q2.x), min(p2.y, q2.y)};
103         Point ur2 = {max(p2.x, q2.x), max(p2.y, q2.y)};
104         return do_rectangles_intersect(dl1, ur1, dl2, ur2);
105     }
106     return false;
107 }
108 /* ===== */
109 /* Circle Intersection */
110 /* ===== */
111 struct Circle {
112     double x, y, r;
113 };
114 bool is_fully_outside(double r1, double r2, double d_sqr) {
115     double tmp = r1 + r2;
116     return d_sqr > tmp * tmp;
117 }
118 bool is_fully_inside(double r1, double r2, double d_sqr) {
119     if (r1 > r2)
120         return false;
121     double tmp = r2 - r1;
122     return d_sqr < tmp * tmp;
123 }
124 bool do_circles_intersect(Circle &c1, Circle &c2) {
125     double dx = c1.x - c2.x;
126     double dy = c1.y - c2.y;
127     double d_sqr = dx * dx + dy * dy;
128     if (is_fully_inside(c1.r, c2.r, d_sqr))
129         return false;
130     if (is_fully_inside(c2.r, c1.r, d_sqr))
131         return false;
132     if (is_fully_outside(c1.r, c2.r, d_sqr))

```

```

133     return false;
134     return true;
135 }
136 /* ===== */
137 /* Point - Line distance */
138 /* ===== */
139 // get distance between p and projection of p on line <- a - b ->
140 double point_line_dist(Point &p, Point &a, Point &b) {
141     Point d = b - a;
142     double t = d.dot(p - a) / d.norm2();
143     return (a + d * t - p).norm();
144 }
145 /* ===== */
146 /* Point - Segment distance */
147 /* ===== */
148 // get distance between p and truncated projection of p on segment a ->
149 // b
150 double point_segment_dist(Point &p, Point &a, Point &b) {
151     if (a == b)
152         return (p - a).norm(); // segment is a single Point
153     Point d = b - a; // direction
154     double t = d.dot(p - a) / d.norm2();
155     if (t <= 0)
156         return (p - a).norm(); // truncate left
157     if (t >= 1)
158         return (p - b).norm(); // truncate right
159     return (a + d * t - p).norm();
160 }
161 /* ===== */
162 /* Straight Line Hashing (integer coords) */
163 /* ===== */
164 // task: given 2 points p1, p2 with integer coordinates, output a unique
165 // representation {a,b,c} such that a*x + b*y + c = 0 is the equation
166 // of the straight line defined by p1, p2. This representation must be
167 // unique for each straight line, no matter which p1 and p2 are sampled.
168 struct Line {
169     int a, b, c;
170 };
171 int gcd(int a, int b) { // greatest common divisor
172     a = abs(a);
173     b = abs(b);
174     while (b) {
175         int c = a;
176         a = b;
177         b = c % b;
178     }
179     return a;
180 }
181 Line getLine(Point p1, Point p2) {
182     int a = p1.y - p2.y;
183     int b = p2.x - p1.x;
184     int c = p1.x * (p2.y - p1.y) - p1.y * (p2.x - p1.x);
185     int sgn = (a < 0 || (a == 0 && b < 0)) ? -1 : 1;
186     int f = gcd(a, gcd(b, c)) * sgn;
187     a /= f;

```

```

187     b /= f;
188     c /= f;
189     return {a, b, c};
190 }

```

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

```

8.2.2 Green's Theorem

```

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

```

8.3 Convex Hull

```

1 // =====
2 // Convex Hull: Andrew's Montone Chain Algorithm
3 // =====
4 struct Point {
5     ll x, y;
6     bool operator<(const Point &p) const {
7         return x < p.x || (x == p.x && y < p.y);
8     }
9 };

```

```

10
11 ll cross(Point &a, Point &b, Point &c) {
12     ll dx0 = b.x - a.x, dy0 = b.y - a.y;
13     ll dx1 = c.x - a.x, dy1 = c.y - a.y;
14     return dx0 * dy1 - dx1 * dy0;
15 }
16
17 vector<Point> upper_hull(vector<Point> &P) {
18     // sort points lexicographically
19     int n = P.size(), k = 0;
20     sort(P.begin(), P.end());
21     // build upper hull
22     vector<Point> uh(n);
23     invrep(i, n, 0) {
24         while (k >= 2 && cross(uh[k - 2], uh[k - 1], P[i]) <= 0)
25             k--;
26         uh[k++] = P[i];
27     }
28     uh.resize(k);
29     return uh;
30 }
31
32 vector<Point> lower_hull(vector<Point> &P) {
33     // sort points lexicographically
34     int n = P.size(), k = 0;
35     sort(P.begin(), P.end());
36     // collect lower hull
37     vector<Point> lh(n);
38     rep(i, n) {
39         while (k >= 2 && cross(lh[k - 2], lh[k - 1], P[i]) <= 0)
40             k--;
41         lh[k++] = P[i];
42     }
43     lh.resize(k);
44     return lh;
45 }
46
47 vector<Point> convex_hull(vector<Point> &P) {
48     int n = P.size(), k = 0;
49     // set initial capacity
50     vector<Point> H(2 * n);
51     // sort points lexicographically
52     sort(P.begin(), P.end());
53     // build lower hull
54     for (int i = 0; i < n; ++i) {
55         while (k >= 2 && cross(H[k - 2], H[k - 1], P[i]) <= 0)
56             k--;
57         H[k++] = P[i];
58     }
59     // build upper hull
60     for (int i = n - 2, t = k + 1; i >= 0; i--) {
61         while (k >= t && cross(H[k - 2], H[k - 1], P[i]) <= 0)
62             k--;
63         H[k++] = P[i];
64     }

```

```

65     // remove extra space
66     H.resize(k - 1);
67     return H;
68 }

```

8.4 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.eb(i - m + 1); // store the position of those
           matches
38         // return; we can return on the first match if needed
39         // this must stay the same
40         j = pi[j - 1];
41     }
42 }
43 return matches; // can be modified to return number of matches or
           location
44 }

```

9.2 Rolling Hashing

```

1  const int MAXLEN = 1e6;
2
3
4  class rollingHashing {
5      static const ull base = 127;
6      static const vector<ull> primes;
7      static vector<vector<ull>> POW;
8
9      static ull add(ull x, ull y, int a) { return (x + y) % primes[a]; }
10     static ull mul(ull x, ull y, int a) { return (x * y) % primes[a]; }
11
12     static void init(int a) {
13         if (POW.size() <= a + 1) {
14             POW.eb(MAXLEN, 1);
15         }
16         repx(i, 1, MAXLEN) POW[a][i] = mul(POW[a][i], base, a);
17     }
18
19     static void init() { rep(i, primes.size()) init(i); }
20
21     vector<vector<ull>> h;
22     int len;
23     rollingHashing(string &s) {
24         len = s.size();
25         h.assign(primes.size(), vector<ull>(len, 0));
26         rep(a, primes.size()) {
27             h[a][0] = s[0] - 'a'; // Assuming alphabetic alphabet
28             repx(i, 1, len) h[a][i] = add(s[i] - 'a', mul(h[a][i - 1], base, a), a);
29         }
30     }
31
32     ull hash(int i, int j, int a) // Inclusive-Exclusive [i,i)?
33     {
34         if (i == 0)
35             return h[a][j - 1];
36         return add(h[a][j - 1], primes[a] - mul(h[a][i - 1], POW[a][j - i], a), a);
37     }
38
39     ull hash(int i, int j) // Supports at most two primes

```

```

40     {
41         return hash(i, j, 1) << 32 |
42             hash(i, j, 0); // Using that 1e18 < __LONG_LONG_MAX__
43     }
44
45     ull hash() { return hash(0, len); } // Also supports at most two
           primes
46 };
47
48 const vector<ull> rollingHashing ::primes({(ull)1e9 + 7,
49                                             (ull)1e9 + 9}); // Add more
           if needed

```

9.3 Trie

```

1
2  /* Implementation from: https://pastebin.com/fyqsH65k */
3  struct TrieNode {
4      int leaf; // number of words that end on a TrieNode (allows for
           duplicate
           // words)
5      int height; // height of a TrieNode, root starts at height = 1, can be
           changed
           // with the default value of constructor
6      // number of words that pass through this node,
7      // ask root node for this count to find the number of entries on the
           whole
8      // Trie all nodes have 1 as they count the words than end on
           themselves (ie
           // leaf nodes count themselves)
9      int count;
10     TrieNode *parent; // pointer to parent TrieNode, used on erasing
           entries
11     map<char, TrieNode *> child;
12     TrieNode(TrieNode *parent = NULL, int height = 1)
13         : parent(parent), leaf(0), height(height),
14           count(0), // change to -1 if leaf nodes are to have count 0
           instead of 1
15           child() {}
16 };
17
18 /**
19  * Complexity: O(|key| * log(k))
20  */
21
22 TrieNode *trie_find(TrieNode *root, const string &str) {
23     TrieNode *pNode = root;
24     for (string::const_iterator key = str.begin(); key != str.end(); key
25         ++) {
26         if (pNode->child.find(*key) == pNode->child.end())
27             return NULL;
28         pNode = pNode->child[*key];
29     }
30     return (pNode->leaf) ? pNode : NULL; // returns only whole word
31     // return pNode; // allows to search for a suffix
32 }
33

```



```

34
35 /**
36  * Complexity:  $O(|key| * \log(k))$ 
37  */
38 void trie_insert(TrieNode *root, const string &str) {
39     TrieNode *pNode = root;
40     root->count += 1;
41     for (string::const_iterator key = str.begin(); key != str.end(); key
42         ++) {
43         if (pNode->child.find(*key) == pNode->child.end())
44             pNode->child[*key] = new TrieNode(pNode, pNode->height + 1);
45         pNode = pNode->child[*key];
46         pNode->count += 1;
47     }
48     pNode->leaf += 1;
49 }
50 /**
51  * Complexity:  $O(|key| * \log(k))$ 
52  */
53 void trie_erase(TrieNode *root, const string &str) {
54     TrieNode *pNode = root;
55     string::const_iterator key = str.begin();
56     for (; key != str.end(); key++) {
57         if (pNode->child.find(*key) == pNode->child.end())
58             return;
59         pNode = pNode->child[*key];
60     }
61     pNode->leaf -= 1;
62     pNode->count -= 1;
63     while (pNode->parent != NULL) {
64         if (pNode->child.size() > 0 || pNode->leaf)
65             break;
66         pNode = pNode->parent, key--;
67         pNode->child.erase(*key);
68         pNode->count -= 1;
69     }
70 }

```

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.size();
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                 } else {
51                     int next = tree[active_node].children[s[active_edge]];
52                     if (active_length >= length(next)) {
53                         active_edge += length(next);
54                         active_length -= length(next);
55                         active_node = next;
56                         continue;
57                     }
58                     if (s[tree[next].start + active_length] == s[i]) {
59                         if (last_new != -1 and active_node != 0) {
60                             tree[last_new].suffix_link = active_node;
61                         }
62                         active_length++;
63                         break;
64                     }
65                     int split_end = tree[next].start + active_length - 1;
66                     int split = tree.size();
67                     tree.emplace_back(tree[next].start, split_end);
68                     tree[active_node].children[s[active_edge]] = split;
69                     int new_leaf = tree.size();

```

```

70     tree.emplace_back(i, -1);
71     tree[split].children[s[i]] = new_leaf;
72     tree[next].start += active_length;
73     tree[split].children[s[tree[next].start]] = next;
74     if (last_new != -1) {
75         tree[last_new].suffix_link = split;
76     }
77     last_new = split;
78 }
79 remaining_suffix--;
80 if (active_node == 0 and active_length > 0) {
81     active_length--;
82     active_edge = i - remaining_suffix + 1;
83 } else if (active_node != 0) {
84     active_node = tree[active_node].suffix_link;
85 }
86 }
87 }
88 i = size - 1;
89 }
90 vector<int> lcp;
91 // last for lcp
92 void dfs(int node, int &index, int depth, int min_depth) {
93     if (tree[node].end == -1 and node != 0) {
94         suffix_array[index] = size - depth;
95         if (index != 0) {
96             lcp[index - 1] = min_depth;
97         }
98         index++;
99     }
100     for (auto it : tree[node].children) {
101         // if(i.second != -1){
102         //     dfs(i.second, index, depth + length(i.second));
103         //     min_depth = depth;
104         // }
105         if (it != -1) {
106             dfs(it, index, depth + length(it), min_depth);
107             min_depth = depth;
108         }
109     }
110 }
111 void build_suffix_array() {
112     suffix_array.resize(size, 0);
113     lcp.resize(size, 0);
114     int index = 0;
115     int depth = 0;
116     dfs(0, index, 0, 0);
117 }
118
119 // pensado para map<int,int>, pero puede modificarse para vector<int>
120 bool match(string &a, string &base) {
121     int active_node = 0;
122     int active_length = 0;
123     int active_char = -1;
124     for (int i = 0; i < a.size(); ) {

```

```

125     if (active_length == 0) {
126         if (!tree[active_node].has_child(a[i]))
127             return false;
128         active_char = a[i];
129         active_length++;
130         i++;
131         continue;
132     }
133     int next = tree[active_node].children[active_char];
134     if (active_length == length(next)) {
135         active_node = next;
136         active_length = 0;
137         active_char = -1;
138         continue;
139     }
140     if ((base)[tree[next].start + active_length] != a[i])
141         return false;
142     active_length++;
143     i++;
144 }
145 return true;
146 }
147 };

```