

Contents

1	Info About Memory and Time Limits	1
2	C++ Cheat Sheet	1
2.1	Headers	1
2.2	Cheat Sheet	2
3	General Algorithms	7
3.1	Search	7
3.2	Brute Force	7
4	Data Structures	7
5	Dynamic Programming	7
6	Graphs	7
7	Mathematics	7
7.1	Modular Arithmetic	7
7.2	Primality Checks	7
7.3	Others	7
8	Geometry	7
8.1	Vectors/Points	7
8.2	Calculate Areas	10
8.2.1	Integration via Simpson's Method	10
8.2.2	Green's Theorem	10
9	Strings	10
9.1	Trie	10
9.2	kmp	11

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 <sup>1</sup>
$O(n^2 \log^2 n)$	1000
$O(n^2 \log n)$	2000
$O(n^2)$	1e4 <sup>2</sup>
$O(n \log^2 n)$	3e5
$O(n \log n)$	1e6
$O(n)$	1e8 <sup>3</sup>

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
```

<sup>1</sup>Este caso esta justo en el limite de tiempo, además en 256 MB cabe a los una matriz de 400<sup>3</sup> ints  
<sup>2</sup>En general solo funciona hasta 6e3  
<sup>3</sup>En 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         \
60         rep(j, (int)m[i].size()) \
61         { \
62             \
63             cerr << " {" << m[i][j].first << "," << m[i][j].second << " } \
64             "; \
65         } \
66         \
67         cerr << endl; \
68     } \
69 #define print(x) copy(x.begin(), x.end(), ostream_iterator<int>(cout, \
70     "")), cout << endl
71
72 template <typename T1, typename T2>
73 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
                             // (safe to use on short strings)
74
75 // print at most first n characters (safe)
76 printf("%.s\n", n, buffer); // make sure that n is integer (with long
                             // long I had problems)
77 //string + \n
78 puts(buffer);
79
80 /* ===== */
81 /* Reading from c string */
82 /* ===== */
83
84 // same as scanf but reading from s
85 int sscanf(const char *s, const char *format, ...);
86
87 /* ===== */
88 /* Printing to c string */
89 /* ===== */
90 // Same as printf but writing into str, the number of characters is
   // returned
91 // or negative if there is failure
92 int sprintf(char *str, const char *format, ...);
93 //example:
94 int n = sprintf(buffer, "%d plus %d is %d", a, b, a + b);
95 printf("[%s] is a string %d chars long\n", buffer, n);
96
97 /* ===== */
98 /* 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
    , li3, li4);
165 // option #2:
166 long int atol(const char *str);
167 // option #3:
168 sscanf(string, "%ld", &l);
169 //-----
170 // string to long long int:
171 // option #1:
172 long long int strtoll(const char *str, char **endptr, int base);
173 // option #2:
174 sscanf(string, "%lld", &l);
175 //-----
176 // string to double:
177 // option #1:
178 double strtod(const char *str, char **endptr); //similar to strtol
179 // option #2:
180 double atof(const char *str);
181 // option #3:
182 sscanf(string, "%lf", &d);
183
184 /* ===== */
185 /* C STRING UTILITY FUNCTIONS */
186 /* ===== */
187 int strcmp(const char *str1, const char *str2); // (-1,0,1)
188 int memcmp(const void *ptr1, const void *ptr2, size_t num); // (-1,0,1)
189 void *memcpy(void *destination, const void *source, size_t num);
190
191 /* ===== */
192 /* C++ STRING UTILITY FUNCTIONS */
193 /* ===== */
194 // read tokens from string
195 string s = "tok1 tok2 tok3";
196 string tok;
197 stringstream ss(s);
198 while (getline(ss, tok, ' '))
199     printf("tok = %s\n", tok.c_str());
200
201 // split a string by a single char delimiter
202 void split(const string &s, char delim, vector<string> &elems)
203 {
204     stringstream ss(s);
205     string item;
206     while (getline(ss, item, delim))
207         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 element
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> ptcunts;
396
397 // -----
398 // inserting into map
399
400 // method #1: operator[]
401 // it overwrites the value if the key already exists
402 ptcunts[{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 = ptcunts.emplace(p, 1);
410     // auto ret = ptcunts.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

## 5 Dynamic Programming

## 6 Graphs

## 7 Mathematics

### 7.1 Modular Arithmetic

### 7.2 Primality Checks

### 7.3 Others

## 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
16   vector2D &operator-=(const vector2D &o)
17   {
18       this->x -= o.x;
19       this->y -= o.y;
20       return *this;
21   }
22
23   vector2D operator+(const vector2D &o)
24   {
25       return {x + o.x, y + o.y};
26   }
27
28   vector2D operator-(const vector2D &o)
29   {
30       return {x - o.x, y - o.y};
31   }
32
33   vector2D operator*(const double &o)
34   {
35       return {x * o, y * o};
36   }
37
38   bool operator==(const vector2D &o)
39   {
40       return x == o.x and y == o.y;
41   }
42
43   double norm2() { return x * x + y * y; }
44   double norm() { return sqrt(norm2()); }
45   double dot(const vector2D &o) { return x * o.x + y * o.y; }
46   double cross(const vector2D &o) { return x * o.y - y * o.x; }
47   double angle()
48   {
49       double angle = atan2(y, x);
50       if (angle < 0)
51           angle += 2 * PI;
52       return angle;
53   }
54
55   vector2D Unit()
56   {
57       return {x / norm(), y / norm()};
58   }
59 };
60
61 /* ===== */
62 /* Cross Product -> orientation of vector2D with respect to ray */
63 /* ===== */

```

```

64 // cross product (b - a) x (c - a)
65 ll cross(vector2D &a, vector2D &b, vector2D &c)
66 {
67     ll dx0 = b.x - a.x, dy0 = b.y - a.y;
68     ll dx1 = c.x - a.x, dy1 = c.y - a.y;
69     return dx0 * dy1 - dx1 * dy0;
70     // return (b - a).cross(c - a); // alternatively, using struct
71     // function
72 }
73
74 // calculates the cross product (b - a) x (c - a)
75 // and returns orientation:
76 // LEFT (1):      c is to the left of ray (a -> b)
77 // RIGHT (-1):    c is to the right of ray (a -> b)
78 // COLLINEAR (0): c is collinear to ray (a -> b)
79 // inspired by: https://www.geeksforgeeks.org/orientation-3-ordered-
80 // points/
81 int orientation(vector2D &a, vector2D &b, vector2D &c)
82 {
83     ll tmp = cross(a, b, c);
84     return tmp < 0 ? -1 : tmp == 0 ? 0 : 1; // sign
85 }
86
87 /* ===== */
88 /* Check if a segment is below another segment (wrt a ray) */
89 /* ===== */
90 // i.e: check if a segment is intersected by the ray first
91 // Assumptions:
92 // 1) for each segment:
93 // p1 should be LEFT (or COLLINEAR) and p2 should be RIGHT (or
94 // COLLINEAR) wrt ray
95 // 2) segments do not intersect each other
96 // 3) segments are not collinear to the ray
97 // 4) the ray intersects all segments
98 struct Segment
99 {
100     vector2D p1, p2;
101 };
102 #define MAXN (int)1e6 //Example
103 Segment segments[MAXN]; // array of line segments
104 bool is_si_below_sj(int i, int j)
105 { // custom comparator based on cross product
106     Segment &si = segments[i];
107     Segment &sj = segments[j];
108     return (si.p1.x >= sj.p1.x) ? cross(si.p1, sj.p2, sj.p1) > 0 : cross
109         (sj.p1, si.p1, si.p2) > 0;
110 }
111 // this can be used to keep a set of segments ordered by order of
112 // intersection
113 // by the ray, for example, active segments during a SWEEP LINE
114 set<int, bool> (*)(int, int)> active_segments(is_si_below_sj); // ordered
115 // set
116
117 /* ===== */
118 /* Rectangle Intersection */

```



```

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

```

```

163  {
164      double dx = c1.x - c2.x;
165      double dy = c1.y - c2.y;
166      double d_sqr = dx * dx + dy * dy;
167      if (is_fully_inside(c1.r, c2.r, d_sqr))
168          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 ->
    b
191  double point_segment_dist(vector2D &p, vector2D &a, vector2D &b)
192  {
193      if (a == b)
194          return (p - a).norm(); // segment is a single vector2D
195      vector2D d = b - a; // direction
196      double t = d.dot(p - a) / d.norm2();
197      if (t <= 0)
198          return (p - a).norm(); // truncate left
199      if (t >= 1)
200          return (p - b).norm(); // truncate right
201      return (a + d * t - p).norm();
202  }
203
204  /* ===== */
205  /* Straight Line Hashing (integer coords) */
206  /* ===== */
207  // task: given 2 points p1, p2 with integer coordinates, output a unique
208  // representation {a,b,c} such that a*x + b*y + c = 0 is the equation
209  // of the straight line defined by p1, p2. This representation must be
210  // unique for each straight line, no matter which p1 and p2 are sampled.
211  struct Line
212  {
213      int a, b, c;
214  };
215  int gcd(int a, int b)
216  { // greatest common divisor

```

```

217     a = abs(a);
218     b = abs(b);
219     while (b)
220     {
221         int c = a;
222         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 }

```

## 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  */
32 TrieNode *trie_find(TrieNode *root, const string &str)
33 {
34     TrieNode *pNode = root;
35     for (string::const_iterator key = str.begin(); key != str.end(); key
36         ++)
```

```

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 /**
40  * Complexity: O(|key| * log(k))
41  */
42 void trie_insert(TrieNode *root, const string &str)
43 {
44     TrieNode *pNode = root;
45     root->count += 1;
46     for (string::const_iterator key = str.begin(); key != str.end(); key
47         ++){
48         if (pNode->child.find(*key) == pNode->child.end())
49             pNode->child[*key] = new TrieNode(pNode, pNode->height + 1);
50         pNode = pNode->child[*key];
51         pNode->count += 1;
52     }
53     pNode->leaf += 1;
54 }
55
56 /**
57  * Complexity: O(|key| * log(k))
58  */
59 void trie_erase(TrieNode *root, const string &str)
60 {
61     TrieNode *pNode = root;
62     string::const_iterator key = str.begin();
63     for (; key != str.end(); key++)
64     {
65         if (pNode->child.find(*key) == pNode->child.end())
66             return;
67         pNode = pNode->child[*key];
68     }
69     pNode->leaf -= 1;
70     pNode->count -= 1;
71     while (pNode->parent != NULL)
72     {
73         if (pNode->child.size() > 0 || pNode->leaf)
74             break;
75         pNode = pNode->parent; key--;
76         pNode->child.erase(*key);
77         pNode->count -= 1;
78     }
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

```