

Echo's notebook

```

1  FLAGS=-Wall -Wextra -Wshadow -Wno-unused-result -D_GLIBCXX_DEBUG -fsanitize=address -fsanitize=undefined
   ↪ -fno-sanitize-recover
2
3  @g++ A.cpp $(FLAGS) -DJUNCO_DEBUG && ./a.out < z.in

1  // Iterate over all submasks of a mask. CONSIDER SUBMASK = 0 APART.
2  for(submask = mask; submask > 0; submask = (submask-1)&mask) {}

```

DP

LCS

```

1  int LCS() { // Longest Common Subsequence.
2      int ns = s.length(), nt = t.length(), i, j;
3      vector<vi> dp(ns + 1, vi(nt + 1, 0)); // One empty row and column, dp is 1-index
4      for(i = 1; i <= ns; i++) {
5          for(j = 1; j <= nt; j++) {
6              if(s[i-1] == t[j-1]) dp[i][j] = dp[i-1][j-1] + 1;
7              else dp[i][j] = max(dp[i-1][j], dp[i][j-1]);
8          }
9      }
10     return dp[ns][nt]; // Length.
11 }

```

LIS

```

1  vll v_LIS(vll &v) {
2      int i, j, n = v.size();
3      vll lis, lis_time(n), ans;
4      if(!n) return ans;
5      lis.pb(v[0]); lis_time[0] = 1;
6      for(i = 1; i < n; i++) {
7          if(v[i] > lis.back()) {lis.pb(v[i]); lis_time[i] = lis.size(); continue;}
8          int pos = upper_bound(lis.begin(), lis.end(), v[i]) - lis.begin();
9          // if(pos > 0 && lis[pos-1] == v[i]) continue; // USE IF YOU WANT STRICTLY INCREASING.
10         lis[pos] = v[i];
11         lis_time[i] = pos+1;
12     }
13     j = lis.size();
14     for(i = n-1; i >= 0; i--) {
15         if(lis_time[i] == j && (ans.empty() || v[i] <= ans.back())) {ans.pb(v[i]); j--;} // <= or <.
16     }
17     reverse(ans.begin(), ans.end());
18     return ans;
19 }

```

IO

```

1  ios::sync_with_stdio(false); cin.tie(nullptr); cout.tie(nullptr);
2
3  stringstream ss;
4  ss << "Hello world";
5  ss.str("Hello world");
6  while(ss >> s) cout << s << endl;
7  ss.clear();

```

Geometry

```

1  template<typename T>
2  class Point {
3      public:
4          static const int LEFT_TURN = 1;
5          static const int RIGHT_TURN = -1;
6          T x = 0, y = 0;
7          Point() = default;
8          Point(T _x, T _y) {
9              x = _x;
10             y = _y;
11         }
12         friend ostream &operator << (ostream &os,
13             ↪ Point<T> &p) {
14             os << "(" << p.x << " " << p.y << ")";
15             return os;
16         }
17         bool operator == (const Point<T> other) const {
18             return x == other.x && y == other.y;
19         }
20         // Get the (1º) bottom (2º) left point.
21         bool operator < (const Point<T> other) const {
22             if(y != other.y) return y < other.y;
23             return x < other.x;
24         }
25         T euclidean_distance(Point<T> other) {
26             T dx = x - other.x;
27             T dy = y - other.y;
28             return sqrt(dx*dx + dy*dy);
29         }
30         T euclidean_distance_squared(Point<T> other) {
31             T dx = x - other.x;
32             T dy = y - other.y;
33             return dx*dx + dy*dy;
34         }
35     };
36
37     // True if a have less angle than b, if *this->a->b is a left turn.
38     bool angle_cmp(Point<T> a, Point<T> b) {
39         if(get_relative_quadrant(a) != get_relative_quadrant(b))
40             return get_relative_quadrant(a) < get_relative_quadrant(b);
41         int ori = get_orientation(a, b);
42         if(ori == 0) return euclidean_distance_squared(a) < euclidean_distance_squared(b);
43         return ori == LEFT_TURN;
44     }
45
46     // Anticlockwise sort starting at 1º quadrant, respect to *this point.
47     void polar_sort(vector<Point<T>> &v) {
48         sort(v.begin(), v.end(), [&](Point<T> a, Point<T> b) {return angle_cmp(a, b);});
49     }
50
51     // Convert v to its convex hull, Do a Graham Scan. O(n log n).
52     void convert_convex_hull(vector<Point<T>> &v) {
53         if(v.size() < 3) return;
54         Point<T> bottom_left = v[0], p2;
55         for(auto p : v) bottom_left = min(bottom_left, p);
56         bottom_left.polar_sort(v);
57         vector<Point<T>> v_input = v; v.clear();
58         for(auto p : v_input) {
59             while(v.size() >= 2) {
60                 p2 = v.back(); v.pop_back();
61                 if(v.back().get_orientation(p2, p) == LEFT_TURN) {
62                     v.pb(p2);
63                     break;
64                 }
65             }
66             v.pb(p);
67         }
68     }
69
70     T manhatan_distance(Point<T> other) {
71         return abs(other.x - x) + abs(other.y - y);
72     }
73
74     // Get the height of the triangle with base b1,
75     ↪ b2.
76     T height_triangle(Point<T> b1, Point<T> b2) {
77         if(b1 == b2 || *this == b1 || *this == b2)
78             ↪ return 0; // It's not a triangle.
79         T a = euclidean_distance(b1);
80         T b = b1.euclidean_distance(b2);
81         T c = euclidean_distance(b2);
82         T d = (c*c-b*b-a*a)/(2*b);
83         return sqrt(a*a - d*d);
84     }
85
86     int get_quadrant() {
87         if(x > 0 && y >= 0) return 1;
88         if(x <= 0 && y > 0) return 2;
89         if(x < 0 && y <= 0) return 3;
90         if(x >= 0 && y < 0) return 4;
91         return 0; // Point (0, 0).
92     }
93
94     // Relative quadrant respect the point other,
95     ↪ not the origin.
96     int get_relative_quadrant(Point<T> other) {
97         Point<T> p(other.x - x, other.y - y);
98         return p.get_quadrant();
99     }
100
101     // Orientation of points *this -> a -> b.
102     int get_orientation(Point<T> a, Point<T> b) {
103         T prod = (a.x - x)*(b.y - a.y) - (a.y -
104             ↪ y)*(b.x - a.x);
105         if(prod == 0) return 0;
106         return prod > 0? LEFT_TURN : RIGHT_TURN;
107     }
108
109     // True if a have less angle than b, if *this->a->b is a left turn.
110     bool angle_cmp(Point<T> a, Point<T> b) {
111         if(get_relative_quadrant(a) != get_relative_quadrant(b))
112             return get_relative_quadrant(a) < get_relative_quadrant(b);
113         int ori = get_orientation(a, b);
114         if(ori == 0) return euclidean_distance_squared(a) < euclidean_distance_squared(b);
115         return ori == LEFT_TURN;
116     }
117
118     // Anticlockwise sort starting at 1º quadrant, respect to *this point.
119     void polar_sort(vector<Point<T>> &v) {
120         sort(v.begin(), v.end(), [&](Point<T> a, Point<T> b) {return angle_cmp(a, b);});
121     }
122
123     // Convert v to its convex hull, Do a Graham Scan. O(n log n).
124     void convert_convex_hull(vector<Point<T>> &v) {
125         if(v.size() < 3) return;
126         Point<T> bottom_left = v[0], p2;
127         for(auto p : v) bottom_left = min(bottom_left, p);
128         bottom_left.polar_sort(v);
129         vector<Point<T>> v_input = v; v.clear();
130         for(auto p : v_input) {
131             while(v.size() >= 2) {
132                 p2 = v.back(); v.pop_back();
133                 if(v.back().get_orientation(p2, p) == LEFT_TURN) {
134                     v.pb(p2);
135                     break;
136                 }
137             }
138             v.pb(p);
139         }
140     }
141
142     T manhatan_distance(Point<T> other) {
143         return abs(other.x - x) + abs(other.y - y);
144     }
145
146     // Get the height of the triangle with base b1,
147     ↪ b2.
148     T height_triangle(Point<T> b1, Point<T> b2) {
149         if(b1 == b2 || *this == b1 || *this == b2)
150             ↪ return 0; // It's not a triangle.
151         T a = euclidean_distance(b1);
152         T b = b1.euclidean_distance(b2);
153         T c = euclidean_distance(b2);
154         T d = (c*c-b*b-a*a)/(2*b);
155         return sqrt(a*a - d*d);
156     }
157
158     int get_quadrant() {
159         if(x > 0 && y >= 0) return 1;
160         if(x <= 0 && y > 0) return 2;
161         if(x < 0 && y <= 0) return 3;
162         if(x >= 0 && y < 0) return 4;
163         return 0; // Point (0, 0).
164     }
165
166     // Relative quadrant respect the point other,
167     ↪ not the origin.
168     int get_relative_quadrant(Point<T> other) {
169         Point<T> p(other.x - x, other.y - y);
170         return p.get_quadrant();
171     }
172
173     // Orientation of points *this -> a -> b.
174     int get_orientation(Point<T> a, Point<T> b) {
175         T prod = (a.x - x)*(b.y - a.y) - (a.y -
176             ↪ y)*(b.x - a.x);
177         if(prod == 0) return 0;
178         return prod > 0? LEFT_TURN : RIGHT_TURN;
179     }
180
181     // True if a have less angle than b, if *this->a->b is a left turn.
182     bool angle_cmp(Point<T> a, Point<T> b) {
183         if(get_relative_quadrant(a) != get_relative_quadrant(b))
184             return get_relative_quadrant(a) < get_relative_quadrant(b);
185         int ori = get_orientation(a, b);
186         if(ori == 0) return euclidean_distance_squared(a) < euclidean_distance_squared(b);
187         return ori == LEFT_TURN;
188     }
189
190     // Anticlockwise sort starting at 1º quadrant, respect to *this point.
191     void polar_sort(vector<Point<T>> &v) {
192         sort(v.begin(), v.end(), [&](Point<T> a, Point<T> b) {return angle_cmp(a, b);});
193     }
194
195     // Convert v to its convex hull, Do a Graham Scan. O(n log n).
196     void convert_convex_hull(vector<Point<T>> &v) {
197         if(v.size() < 3) return;
198         Point<T> bottom_left = v[0], p2;
199         for(auto p : v) bottom_left = min(bottom_left, p);
200         bottom_left.polar_sort(v);
201         vector<Point<T>> v_input = v; v.clear();
202         for(auto p : v_input) {
203             while(v.size() >= 2) {
204                 p2 = v.back(); v.pop_back();
205                 if(v.back().get_orientation(p2, p) == LEFT_TURN) {
206                     v.pb(p2);
207                     break;
208                 }
209             }
210             v.pb(p);
211         }
212     }
213
214     T manhatan_distance(Point<T> other) {
215         return abs(other.x - x) + abs(other.y - y);
216     }
217
218     // Get the height of the triangle with base b1,
219     ↪ b2.
220     T height_triangle(Point<T> b1, Point<T> b2) {
221         if(b1 == b2 || *this == b1 || *this == b2)
222             ↪ return 0; // It's not a triangle.
223         T a = euclidean_distance(b1);
224         T b = b1.euclidean_distance(b2);
225         T c = euclidean_distance(b2);
226         T d = (c*c-b*b-a*a)/(2*b);
227         return sqrt(a*a - d*d);
228     }
229
230     int get_quadrant() {
231         if(x > 0 && y >= 0) return 1;
232         if(x <= 0 && y > 0) return 2;
233         if(x < 0 && y <= 0) return 3;
234         if(x >= 0 && y < 0) return 4;
235         return 0; // Point (0, 0).
236     }
237
238     // Relative quadrant respect the point other,
239     ↪ not the origin.
240     int get_relative_quadrant(Point<T> other) {
241         Point<T> p(other.x - x, other.y - y);
242         return p.get_quadrant();
243     }
244
245     // Orientation of points *this -> a -> b.
246     int get_orientation(Point<T> a, Point<T> b) {
247         T prod = (a.x - x)*(b.y - a.y) - (a.y -
248             ↪ y)*(b.x - a.x);
249         if(prod == 0) return 0;
250         return prod > 0? LEFT_TURN : RIGHT_TURN;
251     }
252
253     // True if a have less angle than b, if *this->a->b is a left turn.
254     bool angle_cmp(Point<T> a, Point<T> b) {
255         if(get_relative_quadrant(a) != get_relative_quadrant(b))
256             return get_relative_quadrant(a) < get_relative_quadrant(b);
257         int ori = get_orientation(a, b);
258         if(ori == 0) return euclidean_distance_squared(a) < euclidean_distance_squared(b);
259         return ori == LEFT_TURN;
260     }
261
262     // Anticlockwise sort starting at 1º quadrant, respect to *this point.
263     void polar_sort(vector<Point<T>> &v) {
264         sort(v.begin(), v.end(), [&](Point<T> a, Point<T> b) {return angle_cmp(a, b);});
265     }
266
267     // Convert v to its convex hull, Do a Graham Scan. O(n log n).
268     void convert_convex_hull(vector<Point<T>> &v) {
269         if(v.size() < 3) return;
270         Point<T> bottom_left = v[0], p2;
271         for(auto p : v) bottom_left = min(bottom_left, p);
272         bottom_left.polar_sort(v);
273         vector<Point<T>> v_input = v; v.clear();
274         for(auto p : v_input) {
275             while(v.size() >= 2) {
276                 p2 = v.back(); v.pop_back();
277                 if(v.back().get_orientation(p2, p) == LEFT_TURN) {
278                     v.pb(p2);
279                     break;
280                 }
281             }
282             v.pb(p);
283         }
284     }
285
286     T manhatan_distance(Point<T> other) {
287         return abs(other.x - x) + abs(other.y - y);
288     }
289
290     // Get the height of the triangle with base b1,
291     ↪ b2.
292     T height_triangle(Point<T> b1, Point<T> b2) {
293         if(b1 == b2 || *this == b1 || *this == b2)
294             ↪ return 0; // It's not a triangle.
295         T a = euclidean_distance(b1);
296         T b = b1.euclidean_distance(b2);
297         T c = euclidean_distance(b2);
298         T d = (c*c-b*b-a*a)/(2*b);
299         return sqrt(a*a - d*d);
300     }
301
302     int get_quadrant() {
303         if(x > 0 && y >= 0) return 1;
304         if(x <= 0 && y > 0) return 2;
305         if(x < 0 && y <= 0) return 3;
306         if(x >= 0 && y < 0) return 4;
307         return 0; // Point (0, 0).
308     }
309
310     // Relative quadrant respect the point other,
311     ↪ not the origin.
312     int get_relative_quadrant(Point<T> other) {
313         Point<T> p(other.x - x, other.y - y);
314         return p.get_quadrant();
315     }
316
317     // Orientation of points *this -> a -> b.
318     int get_orientation(Point<T> a, Point<T> b) {
319         T prod = (a.x - x)*(b.y - a.y) - (a.y -
320             ↪ y)*(b.x - a.x);
321         if(prod == 0) return 0;
322         return prod > 0? LEFT_TURN : RIGHT_TURN;
323     }
324
325     // True if a have less angle than b, if *this->a->b is a left turn.
326     bool angle_cmp(Point<T> a, Point<T> b) {
327         if(get_relative_quadrant(a) != get_relative_quadrant(b))
328             return get_relative_quadrant(a) < get_relative_quadrant(b);
329         int ori = get_orientation(a, b);
330         if(ori == 0) return euclidean_distance_squared(a) < euclidean_distance_squared(b);
331         return ori == LEFT_TURN;
332     }
333
334     // Anticlockwise sort starting at 1º quadrant, respect to *this point.
335     void polar_sort(vector<Point<T>> &v) {
336         sort(v.begin(), v.end(), [&](Point<T> a, Point<T> b) {return angle_cmp(a, b);});
337     }
338
339     // Convert v to its convex hull, Do a Graham Scan. O(n log n).
340     void convert_convex_hull(vector<Point<T>> &v) {
341         if(v.size() < 3) return;
342         Point<T> bottom_left = v[0], p2;
343         for(auto p : v) bottom_left = min(bottom_left, p);
344         bottom_left.polar_sort(v);
345         vector<Point<T>> v_input = v; v.clear();
346         for(auto p : v_input) {
347             while(v.size() >= 2) {
348                 p2 = v.back(); v.pop_back();
349                 if(v.back().get_orientation(p2, p) == LEFT_TURN) {
350                     v.pb(p2);
351                     break;
352                 }
353             }
354             v.pb(p);
355         }
356     }
357
358     T manhatan_distance(Point<T> other) {
359         return abs(other.x - x) + abs(other.y - y);
360     }
361
362     // Get the height of the triangle with base b1,
363     ↪ b2.
364     T height_triangle(Point<T> b1, Point<T> b2) {
365         if(b1 == b2 || *this == b1 || *this == b2)
366             ↪ return 0; // It's not a triangle.
367         T a = euclidean_distance(b1);
368         T b = b1.euclidean_distance(b2);
369         T c = euclidean_distance(b2);
370         T d = (c*c-b*b-a*a)/(2*b);
371         return sqrt(a*a - d*d);
372     }
373
374     int get_quadrant() {
375         if(x > 0 && y >= 0) return 1;
376         if(x <= 0 && y > 0) return 2;
377         if(x < 0 && y <= 0) return 3;
378         if(x >= 0 && y < 0) return 4;
379         return 0; // Point (0, 0).
380     }
381
382     // Relative quadrant respect the point other,
383     ↪ not the origin.
384     int get_relative_quadrant(Point<T> other) {
385         Point<T> p(other.x - x, other.y - y);
386         return p.get_quadrant();
387     }
388
389     // Orientation of points *this -> a -> b.
390     int get_orientation(Point<T> a, Point<T> b) {
391         T prod = (a.x - x)*(b.y - a.y) - (a.y -
392             ↪ y)*(b.x - a.x);
393         if(prod == 0) return 0;
394         return prod > 0? LEFT_TURN : RIGHT_TURN;
395     }
396
397     // True if a have less angle than b, if *this->a->b is a left turn.
398     bool angle_cmp(Point<T> a, Point<T> b) {
399         if(get_relative_quadrant(a) != get_relative_quadrant(b))
400             return get_relative_quadrant(a) < get_relative_quadrant(b);
401         int ori = get_orientation(a, b);
402         if(ori == 0) return euclidean_distance_squared(a) < euclidean_distance_squared(b);
403         return ori == LEFT_TURN;
404     }
405
406     // Anticlockwise sort starting at 1º quadrant, respect to *this point.
407     void polar_sort(vector<Point<T>> &v) {
408         sort(v.begin(), v.end(), [&](Point<T> a, Point<T> b) {return angle_cmp(a, b);});
409     }
410
411     // Convert v to its convex hull, Do a Graham Scan. O(n log n).
412     void convert_convex_hull(vector<Point<T>> &v) {
413         if(v.size() < 3) return;
414         Point<T> bottom_left = v[0], p2;
415         for(auto p : v) bottom_left = min(bottom_left, p);
416         bottom_left.polar_sort(v);
417         vector<Point<T>> v_input = v; v.clear();
418         for(auto p : v_input) {
419             while(v.size() >= 2) {
420                 p2 = v.back(); v.pop_back();
421                 if(v.back().get_orientation(p2, p) == LEFT_TURN) {
422                     v.pb(p2);
423                     break;
424                 }
425             }
426             v.pb(p);
427         }
428     }
429
430     T manhatan_distance(Point<T> other) {
431         return abs(other.x - x) + abs(other.y - y);
432     }
433
434     // Get the height of the triangle with base b1,
435     ↪ b2.
436     T height_triangle(Point<T> b1, Point<T> b2) {
437         if(b1 == b2 || *this == b1 || *this == b2)
438             ↪ return 0; // It's not a triangle.
439         T a = euclidean_distance(b1);
440         T b = b1.euclidean_distance(b2);
441         T c = euclidean_distance(b2);
442         T d = (c*c-b*b-a*a)/(2*b);
443         return sqrt(a*a - d*d);
444     }
445
446     int get_quadrant() {
447         if(x > 0 && y >= 0) return 1;
448         if(x <= 0 && y > 0) return 2;
449         if(x < 0 && y <= 0) return 3;
450         if(x >= 0 && y < 0) return 4;
451         return 0; // Point (0, 0).
452     }
453
454     // Relative quadrant respect the point other,
455     ↪ not the origin.
456     int get_relative_quadrant(Point<T> other) {
457         Point<T> p(other.x - x, other.y - y);
458         return p.get_quadrant();
459     }
460
461     // Orientation of points *this -> a -> b.
462     int get_orientation(Point<T> a, Point<T> b) {
463         T prod = (a.x - x)*(b.y - a.y) - (a.y -
464             ↪ y)*(b.x - a.x);
465         if(prod == 0) return 0;
466         return prod > 0? LEFT_TURN : RIGHT_TURN;
467     }
468
469     // True if a have less angle than b, if *this->a->b is a left turn.
470     bool angle_cmp(Point<T> a, Point<T> b) {
471         if(get_relative_quadrant(a) != get_relative_quadrant(b))
472             return get_relative_quadrant(a) < get_relative_quadrant(b);
473         int ori = get_orientation(a, b);
474         if(ori == 0) return euclidean_distance_squared(a) < euclidean_distance_squared(b);
475         return ori == LEFT_TURN;
476     }
477
478     // Anticlockwise sort starting at 1º quadrant, respect to *this point.
479     void polar_sort(vector<Point<T>> &v) {
480         sort(v.begin(), v.end(), [&](Point<T> a, Point<T> b) {return angle_cmp(a, b);});
481     }
482
483     // Convert v to its convex hull, Do a Graham Scan. O(n log n).
484     void convert_convex_hull(vector<Point<T>> &v) {
485         if(v.size() < 3) return;
486         Point<T> bottom_left = v[0], p2;
487         for(auto p : v) bottom_left = min(bottom_left, p);
488         bottom_left.polar_sort(v);
489         vector<Point<T>> v_input = v; v.clear();
490         for(auto p : v_input) {
491             while(v.size() >= 2) {
492                 p2 = v.back(); v.pop_back();
493                 if(v.back().get_orientation(p2, p) == LEFT_TURN) {
494                     v.pb(p2);
495                     break;
496                 }
497             }
498             v.pb(p);
499         }
500     }
501
502     T manhatan_distance(Point<T> other) {
503         return abs(other.x - x) + abs(other.y - y);
504     }
505
506     // Get the height of the triangle with base b1,
507     ↪ b2.
508     T height_triangle(Point<T> b1, Point<T> b2) {
509         if(b1 == b2 || *this == b1 || *this == b2)
510             ↪ return 0; // It's not a triangle.
511         T a = euclidean_distance(b1);
512         T b = b1.euclidean_distance(b2);
513         T c = euclidean_distance(b2);
514         T d = (c*c-b*b-a*a)/(2*b);
515         return sqrt(a*a - d*d);
516     }
517
518     int get_quadrant() {
519         if(x > 0 && y >= 0) return 1;
520         if(x <= 0 && y > 0) return 2;
521         if(x < 0 && y <= 0) return 3;
522         if(x >= 0 && y < 0) return 4;
523         return 0; // Point (0, 0).
524     }
525
526     // Relative quadrant respect the point other,
527     ↪ not the origin.
528     int get_relative_quadrant(Point<T> other) {
529         Point<T> p(other.x - x, other.y - y);
530         return p.get_quadrant();
531     }
532
533     // Orientation of points *this -> a -> b.
534     int get_orientation(Point<T> a, Point<T> b) {
535         T prod = (a.x - x)*(b.y - a.y) - (a.y -
536             ↪ y)*(b.x - a.x);
537         if(prod == 0) return 0;
538         return prod > 0? LEFT_TURN : RIGHT_TURN;
539     }
540
541     // True if a have less angle than b, if *this->a->b is a left turn.
542     bool angle_cmp(Point<T> a, Point<T> b) {
543         if(get_relative_quadrant(a) != get_relative_quadrant(b))
544             return get_relative_quadrant(a) < get_relative_quadrant(b);
545         int ori = get_orientation(a, b);
546         if(ori == 0) return euclidean_distance_squared(a) < euclidean_distance_squared(b);
547         return ori == LEFT_TURN;
548     }
549
550     // Anticlockwise sort starting at 1º quadrant, respect to *this point.
551     void polar_sort(vector<Point<T>> &v) {
552         sort(v.begin(), v.end(), [&](Point<T> a, Point<T> b) {return angle_cmp(a, b);});
553     }
554
555     // Convert v to its convex hull, Do a Graham Scan. O(n log n).
556     void convert_convex_hull(vector<Point<T>> &v) {
557         if(v.size() < 3) return;
558         Point<T> bottom_left = v[0], p2;
559         for(auto p : v) bottom_left = min(bottom_left, p);
560         bottom_left.polar_sort(v);
561         vector<Point<T>> v_input = v; v.clear();
562         for(auto p : v_input) {
563             while(v.size() >= 2) {
564                 p2 = v.back(); v.pop_back();
565                 if(v.back().get_orientation(p2, p) == LEFT_TURN) {
566                     v.pb(p2);
567                     break;
568                 }
569             }
570             v.pb(p);
571         }
572     }
573
574     T manhatan_distance(Point<T> other) {
575         return abs(other.x - x) + abs(other.y - y);
576     }
577
578     // Get the height of the triangle with base b1,
579     ↪ b2.
580     T height_triangle(Point<T> b1, Point<T> b2) {
581         if(b1 == b2 || *this == b1 || *this == b2)
582             ↪ return 0; // It's not a triangle.
583         T a = euclidean_distance(b1);
584         T b = b1.euclidean_distance(b2);
585         T c = euclidean_distance(b2);
586         T d = (c*c-b*b-a*a)/(2*b);
587         return sqrt(a*a - d*d);
588     }
589
590     int get_quadrant() {
591         if(x > 0 && y >= 0) return 1;
592         if(x <= 0 && y > 0) return 2;
593         if(x < 0 && y <= 0) return 3;
594         if(x >= 0 && y < 0) return 4;
595         return 0; // Point (0, 0).
596     }
597
598     // Relative quadrant respect the point other,
599     ↪ not the origin.
600     int get_relative_quadrant(Point<T> other) {
601         Point<T> p(other.x - x, other.y - y);
602         return p.get_quadrant();
603     }
604
605     // Orientation of points *this -> a -> b.
606     int get_orientation(Point<T> a, Point<T> b) {
607         T prod = (a.x - x)*(b.y - a.y) - (a.y -
608             ↪ y)*(b.x - a.x);
609         if(prod == 0) return 0;
610         return prod > 0? LEFT_TURN : RIGHT_TURN;
611     }
612
613     // True if a have less angle than b, if *this->a->b is a left turn.
614     bool angle_cmp(Point<T> a, Point<T> b) {
615         if(get_relative_quadrant(a) != get_relative_quadrant(b))
616             return get_relative_quadrant(a) < get_relative_quadrant(b);
617         int ori = get_orientation(a, b);
618         if(ori == 0) return euclidean_distance_squared(a) < euclidean_distance_squared(b);
619         return ori == LEFT_TURN;
620     }
621
622     // Anticlockwise sort starting at 1º quadrant, respect to *this point.
623     void polar_sort(vector<Point<T>> &v) {
624         sort(v.begin(), v.end(), [&](Point<T> a, Point<T> b) {return angle_cmp(a, b);});
625     }
626
627     // Convert v to its convex hull, Do a Graham Scan. O(n log n).
628     void convert_convex_hull(vector<Point<T>> &v) {
629         if(v.size() < 3) return;
630         Point<T> bottom_left = v[0], p2;
631         for(auto p : v) bottom_left = min(bottom_left, p);
632         bottom_left.polar_sort(v);
633         vector<Point<T>> v_input = v; v.clear();
634         for(auto p : v_input) {
635             while(v.size() >= 2) {
636                 p2 = v.back(); v.pop_back();
637                 if(v.back().get_orientation(p2, p) == LEFT_TURN) {
638                     v.pb(p2);
639                     break;
640                 }
641             }
642             v.pb(p);
643         }
644     }
645
646     T manhatan_distance(Point<T> other) {
647         return abs(other.x - x) + abs(other.y - y);
648     }
649
650     // Get the height of the triangle with base b1,
651     ↪ b2.
652     T height_triangle(Point<T> b1, Point<T> b2) {
653         if(b1 == b2 || *this == b1 || *this == b2)
654             ↪ return 0; // It's not a triangle.
655         T a = euclidean_distance(b1);
656         T b = b1.euclidean_distance(b2);
657         T c = euclidean_distance(b2);
658         T d = (c*c-b*b-a*a)/(2*b);
659         return sqrt(a*a - d*d);
660     }
661
662     int get_quadrant() {
663         if(x > 0 && y >= 0) return 1;
664         if(x <= 0 && y > 0) return 2;
665         if(x < 0 && y <= 0) return 3;
666         if(x >= 0 && y < 0) return 4;
667         return 0; // Point (0, 0).
668     }
669
670     // Relative quadrant respect the point other,
671     ↪ not the origin.
672     int get_relative_quadrant(Point<T> other) {
673         Point<T> p(other.x - x, other.y - y);
674         return p.get_quadrant();
675     }
676
677     // Orientation of points *this -> a -> b.
678     int get_orientation(Point<T> a, Point<T> b) {
679         T prod = (a.x - x)*(b.y - a.y) - (a.y -
680             ↪ y)*(b.x - a.x);
681         if(prod == 0) return 0;
682         return prod > 0? LEFT_TURN : RIGHT_TURN;
683     }
684
685     // True if a have less angle than b, if *this->a->b is a left turn.
686     bool angle_cmp(Point<T> a, Point<T> b) {
687         if(get_relative_quadrant(a) != get_relative_quadrant(b))
688             return get_relative_quadrant(a) < get_relative_quadrant(b);
689         int ori = get_orientation(a, b);
690         if(ori == 0) return euclidean_distance_squared(a) < euclidean_distance_squared(b);
691         return ori == LEFT_TURN;
692     }
693
694     // Anticlockwise sort starting at 1º quadrant, respect to *this point.
695     void polar_sort(vector<Point<T>> &v) {
696         sort(v.begin(), v.end(), [&](Point<T> a, Point<T> b) {return angle_cmp(a, b);});
697     }
698
699     // Convert v to its convex hull, Do a Graham Scan. O(n log n).
700     void convert_convex_hull(vector<Point<T>> &v) {
701         if(v.size() < 3) return;
702         Point<T> bottom_left = v[0], p2;
703         for(auto p : v) bottom_left = min(bottom_left, p);
704         bottom_left.polar_sort(v);
705         vector<Point<T>> v_input = v; v.clear();
706         for(auto p : v_input) {
707             while(v.size() >= 2) {
708                 p2 = v.back(); v.pop_back();
709                 if(v.back().get_orientation(p2, p) == LEFT_TURN) {
710                     v.pb(p2);
711                     break;
712                 }
713             }
714             v.pb(p);
715         }
716     }
717
718     T manhatan_distance(Point<T> other) {
719         return abs(other.x - x) + abs(other.y - y);
720     }
721
722     // Get the height of the triangle with base b1,
723     ↪ b2.
724     T height_triangle(Point<T> b1, Point<T> b2) {
725         if(b1 == b2 || *this == b1 || *this == b2)
726             ↪ return 0; // It's not a triangle.
727         T a = euclidean_distance(b1);
728         T b = b1.euclidean_distance(b2);
729         T c = euclidean_distance(b2);
730         T d = (c*c-b*b-a*a)/(2*b);
731         return sqrt(a*a - d*d);
732     }
733
734     int get_quadrant() {
735         if(x > 0 && y >= 0) return 1;
736         if(x <= 0 && y > 0) return 2;
737         if(x < 0 && y <= 0) return 3;
738         if(x >= 0 && y < 0) return 4;
739         return 0; // Point (0, 0).
740     }
741
742     // Relative quadrant respect the point other,
743     ↪ not the origin.
744     int get_relative_quadrant(Point<T> other) {
745         Point<T> p(other.x - x, other.y - y);
746         return p.get_quadrant();
747     }
748
749     // Orientation of points *this -> a -> b.
750     int get_orientation(Point<T> a, Point<T> b) {
751         T prod = (a.x - x)*(b.y - a.y) - (a.y -
752             ↪ y)*(b.x - a.x);
753         if(prod == 0) return 0;
754         return prod > 0? LEFT_TURN : RIGHT_TURN;
755     }
756
757     // True if a have less angle than b, if *this->a->b is a left turn.
758     bool angle_cmp(Point<T> a, Point<T> b) {
759         if(get_relative_quadrant(a) != get_relative_quadrant(b))
760             return get_relative_quadrant(a) < get_relative_quadrant(b);
761         int ori = get_orientation(a, b);
762         if(ori == 0) return euclidean_distance_squared(a) < euclidean_distance_squared(b);
763         return ori == LEFT_TURN;
764     }
765
766     // Anticlockwise sort starting at 1º quadrant, respect to *this point.
767     void polar_sort(vector<Point<T>> &v) {
768         sort(v.begin(), v.end(), [&](Point<T> a, Point<T> b) {return angle_cmp(a, b);});
769     }
770
771     // Convert v to its convex hull, Do a Graham Scan. O(n log n).
772     void convert_convex_hull(vector<Point<T>> &v) {
773         if(v.size() < 3) return;
774         Point<T> bottom_left = v[0], p2;
775         for(auto p : v) bottom_left = min(bottom_left, p);
776         bottom_left.polar_sort(v);
777         vector<Point<T>> v_input = v; v.clear();
778         for(auto p : v_input) {
779             while(v.size() >= 2) {
780                 p2 = v.back(); v.pop_back();
781                 if(v.back().get_orientation(p2, p) == LEFT_TURN) {
782                     v.pb(p2);
783                     break;
784                 }
785             }
786             v.pb(p);
787         }
788     }
789
790     T manhatan_distance(Point<T> other) {
791         return abs(other.x - x) + abs(other.y - y);
792     }
793
794     // Get the height of the triangle with base b1,

```

Graphs

Articulation points and bridges

```

1 vector<vi> adyList; // Graph
2 vi num, low; // num and low for DFS
3 int cnt; // Counter for DFS
4 int root, rchild; // Root and number of (DFS)
  ↳ children
5 vi artic; // Contains the articulation
  ↳ points at the end
6 set<pii> bridges; // Contains the bridges at the
  ↳ end
7
8 void dfs(int nparent, int nnode) {
9     num[nnode] = low[nnode] = cnt++;
10    rchild += (nparent == root);
11
12    for (auto a : adyList[nnode]) {
13        if (num[a] == -1) { // Tree edge
14            dfs(nnode, a);
15            low[nnode] = min(low[nnode], low[a]);
16            if (low[a] >= num[nnode]) {
17                artic[nnode] = true;
18            }
19            if (low[a] > num[nnode]) {
20                bridges.insert((nnode < a) ?
  ↳ mp(nnode, a) : mp(a, nnode));
21            }
22        } else if (a != nparent) { // Back edge
23            low[nnode] = min(low[nnode], num[a]);
24        }
25    }
26 }
27 void findArticulations(int n) {
28     cnt = 0;
29     low = num = vi(n, -1);
30     artic = vi(n, 0);
31     bridges.clear();
32
33     for (int i = 0; i < n; ++i) {
34         if (num[i] != -1) {
35             continue;
36         }
37         root = i;
38         rchild = 0;
39         dfs(-1, i);
40         artic[root] = rchild > 1; //Special case
41     }
42 }

```

Bellman Ford's

```

1 for(i = 0; i < n - 1; i++) { // Iterate n - 1 times.
2     for(auto e : edge) {
3         if(dist[e.fi.fi] != inf)
4             dist[e.fi.se] = min(dist[e.fi.se], dist[e.fi.fi] + e.se);
5     }
6 }

```

Floyd cycle detection

```

1 void floyd_detection() {
2     ll pslow = f(F_0), pfast = f(f(F_0)), iteration = 0;
3     while(pslow != pfast) pslow = f(pslow), pfast = f(f(pfast));
4     pslow = F_0;
5     while(pslow != pfast) pslow = f(pslow), pfast = f(pfast), iteration++;
6     cout << "In " << iteration << " coincide with value: " << pslow << endl;
7     pfast = f(pfast), iteration++;
8     while(pslow != pfast) pfast = f(pfast), iteration++;
9     cout << "In " << iteration << " coincide with value: " << pfast << endl;
10 }

```

Max Flow: Edmond Karp's

```

1 vector<vector<ll>> adjList;
2 vector<vector<ll>> adjMat;
3
4 void initialize(int n) {
5     adjList = decltype(adjList)(n);
6     adjMat = decltype(adjMat)(n, vector<ll>(n, 0));
7 }
8
9 map<int, int> p;
10 bool bfs(int source, int sink) {
11     queue<int> q;
12     vi visited(adjList.size(), 0);
13     q.push(source);
14     visited[source] = 1;
15     while (!q.empty()) {
16         int u = q.front();
17         q.pop();
18         if (u == sink)
19             return true;
20         for (auto v : adjList[u]) {
21             if (adjMat[u][v] > 0 && !visited[v]) {
22                 visited[v] = true;
23                 q.push(v);
24                 p[v] = u;
25             }
26         }
27     }
28     return false;
29 }
30 int max_flow(int source, int sink) {
31     ll max_flow = 0;
32     while (bfs(source, sink)) {

```

```

33         ll flow = inf;
34         for (int v = sink; v != source; v = p[v]) {
35             flow = min(flow, adjMat[p[v]][v]);
36         }
37         for (int v = sink; v != source; v = p[v]) {
38             adjMat[p[v]][v] -= flow; // Decrease
39             ↪ capacity forward edge
40             adjMat[v][p[v]] += flow; // Increase
41             ↪ capacity backward edge
42         }
43         max_flow += flow;
44     }
45     return max_flow;
46 }
47 void addedgeUni(int orig, int dest, ll flow) {
48     adjList[orig].pb(dest);
49     adjMat[orig][dest] = flow;
50     adjList[dest].pb(orig); //Add edge for residual
51     ↪ flow
52 }
53 void addEdgeBi(int orig, int dest, ll flow) {
54     adjList[orig].pb(dest);
55     adjList[dest].pb(orig);
56     adjMat[orig][dest] = flow;
57     adjMat[dest][orig] = flow;
58 }

```

Max Flow: Dinic's

```

1 // O(V^2*E) max flow algorithm. For bipartite
2 ↪ matching O(sqrt(V)*E), always faster than
3 ↪ Edmond-Karp.
4 // Creates layer's graph with a BFS and then it
5 ↪ tries all possibles DFS, branching while the
6 ↪ path doesn't reach the sink
7 struct EdgeFlow {
8     ll u, v;
9     ll cap, flow = 0; //capacity and current flow
10     EdgeFlow(ll _u, ll _v, ll _cap) : u(_u), v(_v),
11     ↪ cap(_cap) { }
12 };
13 struct Dinic {
14     vector<EdgeFlow> edge; //keep the edges
15     vector<vll> graph; //graph[u] is the list of
16     ↪ their edges
17     ll n, n_edges = 0;
18     ll source, sink, inf_flow = inf;
19     vll lvl; //lvl of the node to the source
20     vll ptr; //ptr[u] is the next edge you have to
21     ↪ take in order to branch the DFS
22     queue<ll> q;
23
24     Dinic(ll _n, ll _source, ll _sink) : n(_n),
25     ↪ source(_source), sink(_sink) { //n nodes
26         graph.assign(_n, vll());
27     }
28
29     void add_edge(ll u, ll v, ll flow) { //u->v
30     ↪ with cost x
31         EdgeFlow uv(u, v, flow), vu(v, u, 0);
32         edge.pb(uv);
33         edge.pb(vu);
34         graph[u].pb(n_edges);
35         graph[v].pb(n_edges+1);
36         n_edges += 2;
37     }
38
39     bool BFS() {
40         ll u;
41         while(q.empty() == false) {
42             u = q.front(); q.pop();
43             for(auto e1 : graph[u]) {
44                 if(lvl[edge[e1].v] != -1) {
45                     continue;
46                 }
47                 if(edge[e1].cap - edge[e1].flow <=
48                 ↪ 0) {
49                     continue;
50                 }
51                 lvl[edge[e1].v] = lvl[edge[e1].u] +
52                 ↪ 1;
53                 q.push(edge[e1].v);
54             }
55         }
56         return lvl[sink] != -1;
57     }
58
59     ll dfs(ll u, ll min_flow) {
60         if(u == sink) return min_flow;
61         ll pushed, e1;
62         for(; ptr[u] < (int)graph[u].size();
63         ↪ ptr[u]++) { //if you can pick ok, else
64         ↪ you crop that edge for the current bfs
65         ↪ layer
66             e1 = graph[u][ptr[u]];
67             if(lvl[edge[e1].v] != lvl[edge[e1].u] +
68             ↪ 1 || edge[e1].cap - edge[e1].flow
69             ↪ <= 0) {
70                 continue;
71             }
72             pushed = dfs(edge[e1].v, min(min_flow,
73             ↪ edge[e1].cap - edge[e1].flow));
74             if(pushed > 0) {
75                 edge[e1].flow += pushed;
76                 edge[e1^1].flow -= pushed;
77                 return pushed;
78             }
79         }
80         return 0;
81     }
82
83     ll max_flow() {
84         ll flow = 0, pushed;
85         while(true) {
86             lvl.assign(n, -1);
87             lvl[source] = 0;
88             q.push(source);
89             if(!BFS()) {
90                 break;
91             }
92             ptr.assign(n, 0);
93             while(true) {
94                 pushed = dfs(source, inf_flow);
95                 if(!pushed) break;
96                 flow += pushed;
97             }
98         }
99         return flow;
100     }
101 };

```

Hungarian Algorithm

```

1  const int MAX_N1 = 1002; //number of workers
2  const int MAX_N2 = 2002; //number of items
3  int cost[MAX_N1][MAX_N2]; //cost matrix, entries >= 0
4  int u[MAX_N1+1], v[MAX_N2+1]; //potentials, always cost[i][j] >= u[i] + v[j]
5  int slack[MAX_N2+1]; //cost[i][j] - u[i] - v[j], always >= 0
6  int prevy[MAX_N2+1]; //edges of the current path: prev[j0] -> yx[prev[j0]] -> j0. Dont need to reset
7  bool used[MAX_N2+1]; //visited array
8  int yx[MAX_N2+1]; //match of the j column
9  int n1=, n2=, INF = INT_MAX-1; //actual size of workers and items.
10
11 //http://e-maxx.ru/algo/assignment_hungary
12 //Solves MINIMUM Assignment. For maximum change cost[i][j] to Max_entry - cost[i][j] and resize the answer.
13 //There are 1..n1 rows and 1..n2 columns, ALWAYS n1 <= n2. Complexity(n1 * n1*n2)
14 //The function use 1-index for variables because it creates a virtual vertex 0
15 int Hungarian() {
16     int i, i0, j, j0, min_j, delta, ans;
17     fill(u, u+n1+1, 0);
18     fill(v, v+n2+1, 0);
19     fill(yx, yx+n2+1, 0);
20     for(i = 1; i <= n1; i++) { //Add row by row to the current matching
21         yx[0] = i; //connect 0 of set 2 with vertex i
22         j0 = 0; //i0 and j0 are the current selected row and column, i and j are just iterators
23         fill(slack, slack+n2 + 1, INF);
24         fill(used, used + n2 + 1, false);
25         do { //while the alternating path is not augmenting path
26             used[j0] = true;
27             delta = INF;
28             i0 = yx[j0];
29             for(j = 1; j <= n2; j++) { //get the delta among all columns not used
30                 if(!used[j]) {
31                     int cur = cost[i0-1][j-1] - u[i0] - v[j];
32                     if(cur < slack[j]) {
33                         slack[j] = cur, prevy[j] = j0;
34                     }
35                     if(slack[j] < delta) { //try if delta == 0 break
36                         delta = slack[j], min_j = j;
37                     }
38                 }
39             }
40             for(j = 0; j <= n2; j++) { //add delta in set 1, subtract delta in set 2
41                 if(used[j]) u[yx[j]] += delta, v[j] -= delta;
42                 else slack[j] -= delta;
43             }
44             j0 = min_j;
45         } while(yx[j0] != 0);
46         do { //invert the augmenting path
47             yx[j0] = yx[prevy[j0]];
48             j0 = prevy[j0];
49         } while(j0);
50     }
51     ans = 0;
52     for(j = 1; j <= n2; j++) { //recover solution. The matched edges are yx[j]-1 -> j-1
53         if(yx[j])
54             ans += cost[yx[j]-1][j-1];
55     }
56     return ans;
57 }
58
59 // THE ANS IS n1*M_factor - Hungarian().
60 int M_factor; // Change problem finding the minimum cost to maximum cost, that can be solved by Hungarian
61 void min_to_max() { //min in cost[i][j] = max in M - cost[i][j].
62     int i, j;
63     M_factor = 0;
64     for(i = 0; i < n1; i++) {
65         M_factor = max(M_factor, *max_element(cost[i], cost[i]+n2));
66     }
67     for(i = 0; i < n1; i++) {

```

```

68         for(j = 0; j < n2; j++) {
69             cost[i][j] = M_factor - cost[i][j];
70         }
71     }
72 }

```

Floyd - Warshall: k->i->j LCA tree

```

1  const ll LOG_N = 20; //log2(MAX_N) + 4
2  const ll MAX_N = 1e5 + 3; //1e5, example
3  vector<vector<ll>> graph2, graph; //graph2 is the
   ↳ bidirectional and graph is the one you ask LCA
4  //----- LCA in a tree rooted at 0
   ↳ -----
5  int parent[LOG_N][MAX_N]; //parent[i][j] is the
   ↳ ancestor 2^i of node j. Is a sparse table
6  int level[MAX_N]; //depth of the node in the tree
7
8  //call dfs0(0, 0);
9  void dfs0(int u, int p) {
10     parent[0][u] = p;
11     for(auto v : graph[u]) {
12         if(v == p) continue;
13         level[v] = level[u] + 1;
14         dfs0(v, u);
15     }
16 }
17
18 //O(n log n)
19 void preprocess() {
20     int i, j;
21     dfs0(0, 0);
22     for(i = 1; i < LOG_N; ++i) {
23         for(j = 0; j < MAX_N; ++j) {
24             parent[i][j] = parent[i - 1][parent[i - 1][j]];
25         }
26     }
27 }
28
29 //rise b to the same level as a and continue moving
   ↳ up. O(log n)
30 int lca(int a, int b) {
31     int i;
32
33     if(level[a] > level[b]) swap(a, b);
34     int d = level[b] - level[a];
35
36     for(i = 0; i < LOG_N; ++i) {
37         if((d >> i) & 1) b = parent[i][b];
38     }
39     if(a == b) return a;
40
41     for(i = LOG_N - 1; i >= 0; --i) {
42         if(parent[i][a] != parent[i][b])
43             a = parent[i][a], b = parent[i][b];
44     }
45     return parent[0][a];
46 }
47
48 //distance between two nodes in a tree
49 int dist(int u, int v) {
50     return level[u] + level[v] - 2 * level[lca(u,
51     ↳ v)];
52 }
53
54 //call dfs(0, -1) to root a tree at 0. the graph
   ↳ had to be bidirectional
55 vector<bool> visited;
56 void dfs(int x, int p) {
57     if(visited[x]) return;
58     visited[x] = true;
59     if(p != -1) graph[p].pb(x);
60     for(auto el : graph2[x]) {
61         if(el == p) continue;
62         dfs(el, x);
63     }
64 }

```

Kosaraju

```

1  vector<vi> adyList; // Graph
2  vector<int> visited; // Visited for DFS
3  vector<vi> sccs; // Contains the SCCs at the
   ↳ end
4
5  void dfs(int nnode, vector<int> &v, vector<vi>
   ↳ &adyList) {
6     if (visited[nnode]) {
7         return;
8     }
9     visited[nnode] = true;
10     for (auto a : adyList[nnode]) {
11         dfs(a, v, adyList);
12     }
13     v.push_back(nnode);
14 }
15
16 void Kosaraju(int n) {
17     visited = vi(n, 0);
18     stack<int> s = stack<int>();
19
20     sccs = vector<vi>();
21
22     vector<int> postorder;
23     for (int i = 0; i < n; ++i) {
24         dfs(i, postorder, adyList);
25     }
26     reverse(all(postorder));
27
28     vector<vi> rAdyList = vector<vi>(n, vi());
29     for (int i = 0; i < n; ++i) {
30         for (auto v : adyList[i]) {
31             rAdyList[v].push_back(i);
32         }
33     }
34
35     visited = vi(n, 0);
36     vi data;
37     for (auto a : postorder) {
38         if (!visited[a]) {
39             data = vi();

```

```

39         dfs(a, data, rAdyList);
40         if (!data.empty())
41             sccs.pb(data);

```

```

42     }
43 }
44 }

```

Mathematics

Binary operations

```

1  ll elevate(ll a, ll b) { // b >= 0.
2      ll ans = 1;
3      while(b) {
4          if(b & 1) ans = ans * a % mod;
5          b >>= 1;
6          a = a * a % mod;
7      }
8      return ans;
9  }
10 // a^(mod - 1) = 1, Euler.
11 ll inv(ll a) {
12     return elevate(((a%mod) + mod)%mod, mod - 2);
13 }
14

```

```

15 ll mul(ll a, ll b) {
16     ll ans = 0, neg = (a < 0) ^ (b < 0);
17     a = abs(a); b = abs(b);
18     while(b) {
19         if(b & 1) ans = (ans + a) % mod;
20         b >>= 1;
21         a = (a + a) % mod;
22     }
23     if(neg) return -ans;
24     return ans;
25 }

```

Catalan numbers: $C_n = \frac{1}{n+1} \binom{2n}{n}$

Combinatoric numbers

```

1  const int MAX_C = 1+66; // 66 is the for long
   ↪ long, C(66, x)
2  ll Comb[MAX_C][MAX_C];
3
4  void calc() {
5      int i, j;
6      for(i = 0; i < MAX_C; i++) {
7          Comb[i][0] = 1;
8          Comb[0][i] = 1;
9      }
10     for(i = 1; i < MAX_C; i++) {

```

```

11         for(j = 1; j < MAX_C; j++) {
12             if(i+j >= MAX_C) continue;
13             Comb[i][j] = Comb[i-1][j] +
   ↪ Comb[i][j-1];
14         }
15     }
16 }
17 ll C(ll i, ll j) {
18     return Comb[i-j][j];
19 }

```

Chinese Remainder

```

1  const ll MAX = 10;
2  ll a[MAX], p[MAX], n;
3  // Given n x == a[i] mod p[i], find x, or -1 if it doesn't exist.
4  // Let q[i] = (\prod_{i=0}^{n-1} p[j])/p[i].
5  // x will be = \sum_{i=0}^{n-1} a[i]*q[i]*inv(q[i], mod p[i])
6  ll chinese_remainder() {
7      ll i, j, g, ans = 0, inv1, inv2;
8      mod = 1;
9      for(i = 0; i < n; i++) { // If the p[i] are not coprimes, do them coprimes.
10         a[i] %= p[i]; a[i] += p[i]; a[i] %= p[i];
11         for(j = 0; j < i; j++) {
12             g = __gcd(p[i], p[j]);
13             if((a[i]%g + g)%g != (a[j]%g + g)%g) return -1;
14             // Delete the repeated factor at the correct side.
15             if(__gcd(p[i]/g, p[j]) == 1) {p[i] /= g; a[i] %= p[i];}
16             else {p[j] /= g; a[j] %= p[j];}
17         }
18     }
19     // If you have a supermod, take P = min(P, supermod);
20     for(i = 0; i < n; i++) {
21         mod *= p[i];
22     }
23     for(i = 0; i < n; i++) {
24         gcdEx(mod/p[i], p[i], &inv1, &inv2);
25         ans += mul(a[i], mul(mod/p[i], inv1));
26         ans %= mod;
27     }
28     return (ans%mod + mod) % mod;
29 }

```

Euclides

```

1 ll gcdEx(ll a, ll b, ll *x1, ll *y1) {
2     if(a == 0) {
3         *x1 = 0;
4         *y1 = 1;
5         return b;
6     }
7     ll x0, y0, g;
8     g = gcdEx(b%a, a, &x0, &y0);
9     *x1 = y0 - (b/a)*x0;
10    *y1 = x0;
11    return g;
12 }

```

Linear Sieve

```

1 const int MAX_PRIME = 1e6+5;
2 bool num[MAX_PRIME]; // If num[i] = false => i is prime.
3 int num_div[MAX_PRIME]; // Number of divisors of i.
4 int min_div[MAX_PRIME]; // The smallest prime that divide i.
5 vector<int> prime;
6
7 void linear_sieve(){
8     int i, j, prime_size = 0;
9     min_div[1] = 1;
10    for(i = 2; i < MAX_PRIME; ++i){
11        if(num[i] == false) {prime.push_back(i); ++prime_size; num_div[i] = 1; min_div[i] = i;}
12
13        for(j = 0; j < prime_size && i * prime[j] < MAX_PRIME; ++j){
14            num[i * prime[j]] = true;
15            num_div[i * prime[j]] = num_div[i] + 1;
16            min_div[i * prime[j]] = min(min_div[i], prime[j]);
17            if(i % prime[j] == 0) break;
18        }
19    }
20    bool is_prime(ll n) {
21        for(auto el : prime) {
22            if(n == el) return true;
23            if(n%el == 0) return false;
24        }
25        return true;
26    }
27
28    vll fact, nfact; // The factors of n and their
29    ↪ exponent.
30    void factorize(int n) { // Up to
31    ↪ MAX_PRIME*MAX_PRIME.
32        ll cont, prev_p;
33        fact.clear(); nfact.clear();
34        for(auto p : prime) {
35            if(n < MAX_PRIME) break;
36            if(n%p == 0) {
37                fact.pb(p);
38                cont = 0;
39                while(n%p == 0) n /= p, cont++;
40                nfact.pb(cont);
41            }
42            if(n >= MAX_PRIME) {
43                fact.pb(n);
44                nfact.pb(1);
45                return;
46            }
47            while(n != 1) { // When n < MAX_PRIME,
48            ↪ factorization in almost O(1).
49                prev_p = min_div[n];
50                cont = 0;
51                while(n%prev_p == 0) n /= prev_p, cont++;
52                fact.pb(prev_p);
53                nfact.pb(cont);
54            }
55        }
56    }

```

Hash Set

```

1 const int MAX = 2*1e5+5;
2 ll val[MAX]; // For random numbers and not index
3 ↪ use f with random xor.
4 void ini() { // CALL ME ONCE.
5     srand(time(0));
6     for(int i = 0; i < MAX; i++) val[i] = rand();
7 }
8 // Hash_set contains a set of indices [0..MAX-1]
9 ↪ with duplicates.
10 // a[i] = sum_x{val_x} % mod p[i].
11 class Hash_set {
12 public:
13     vll p = {1237273, 1806803, 3279209}; // Prime
14     ↪ numbers.
15     vll a = {0, 0, 0};
16     int n = 3; // n = p.size();
17
18     void insert(int x) { // Insert index x.
19         for(int i = 0; i < n; i++) a[i] = (a[i] +
20             ↪ val[x]) % p[i];
21     }
22     // Insert all the elements of hs.
23     void insert (Hash_set hs) {
24         for(int i = 0; i < n; i++) a[i] = (a[i] +
25             ↪ hs.a[i]) % p[i];
26     }
27
28     bool operator == (Hash_set hs) {
29         for(int i = 0; i < n; i++) if(a[i] !=
30             ↪ hs.a[i]) return false;
31         return true;
32     }
33 };

```


BIT Fenwick tree

```

1  template<typename T>
2  class BIT{
3      vector<T> bit;
4      int n;
5      public:
6      BIT(int _n) {
7          n = _n;
8          bit.assign(n+1, 0);
9      }
10     BIT(vector<T> v) {
11         n = v.size();
12         bit.assign(n+1, 0);
13         for(int i = 0; i < n; i++) update(i, v[i]);
14     }
15     // Point update.
16     void update(int i, T dx) {
17         for(i++; i < n+1; i += LSB(i)) bit[i] +=
18             ⇨ dx;
19     }
20     // query [0, r].
21     T query(int r) {
22         T ans = 0;
23         for(r++; r > 0; r -= LSB(r)) ans += bit[r];
24         return ans;
25     }
26     // query [l, r].
27     T query(int l, int r) {
28         return query(r) - query(l-1);
29     }
30     // k-th smallest element inserted.
31     int k_element(int k) { // k > 0 (1-indexed).
32         int l = 0, r = n+1, mid;
33         if(query(0) >= k) return 0;
34         while(l + 1 < r) {
35             mid = (l + r)/2;
36             if(query(mid) >= k) r = mid;
37             else l = mid;
38         }
39         return r;
40     }
41 };

```

Strings

KMP

```

1  // Knuth-Morris-Pratt. Search the occurrences of t
2  ⇨ (pattern to search) in s (the text).
3  // O(n). It increases j at most n times and
4  ⇨ decreases at most n times.
5  void KMP(string &s, string &t) {
6      int n = s.length(), m = t.length(), i, j, len =
7      ⇨ 0;
8      // Longest proper prefix that is also suffix.
9      // s[0..lps[i]-1] == s[i-lps[i]+1..i].
10     vi lps(m, 0);
11     for(i = 1; i < m; i++) {
12         if(t[i] == t[len]) {
13             len++;
14             lps[i] = len;
15         } else if(len > 0) {
16             len = lps[len - 1];
17         }
18     }
19     i--;
20 }

```

Longest Palindromic Substring

```

1  // LPS Longest Palindromic Substring, O(n).
2  void Manacher(string &str) {
3      char ch = '#'; // '#' a char not contained in str.
4      string s(1, ch), ans;
5      for(auto c : str) {s += c; s += ch;}
6      int i, n = s.length(), c = 0, r = 0;
7      vi lps(n, 0);
8      for(i = 1; i < n; i++) {
9          // lps[i] >= it's mirror, but falling in the interval [L..R]. L = c - (R - c).
10         if(i < r) lps[i] = min(r - i, lps[c - (i - c)]);
11         // Try to increase.
12         while(i-lps[i]-1 >= 0 && i+lps[i]+1 < n && s[i-lps[i]-1] == s[i+lps[i]+1]) lps[i]++;
13         // Update the interval [L..R].
14         if(i + lps[i] > r) c = i, r = i + lps[i];
15     }
16     // Get the longest palindrome in ans.
17     int pos = max_element(lps.begin(), lps.end()) - lps.begin();
18     for(i = pos - lps[pos]; i <= pos + lps[pos]; i++) {
19         if(s[i] != ch) ans += s[i];
20     }
21     //cout << ans.size() << "\n";
22 }

```

Z-algorithm

```

1 // Search the occurrences of t (pattern to search) in s (the text).
2 // O(n + m). It increases R at most 2n times and decreases at most n times.
3 // z[i] is the longest string s[i..i+z[i]-1] that is a prefix = s[0..z[i]-1].
4 void z_algorithm(string &s, string &t) {
5     s = t + "$" + s; // "$" is a char not present in s nor t.
6     int n = s.length(), m = t.length(), i, L = 0, R = 0;
7     vi z(n, 0);
8     // s[L..R] = s[0..R-L], [L, R] is the current window.
9     for(i = 1; i < n; i++) {
10         if(i > R) { // Old window, recalculate.
11             L = R = i;
12             while(R < n && s[R] == s[R-L]) R++;
13             R--;
14             z[i] = R - L + 1;
15         } else {
16             if(z[i-L] < R - i) z[i] = z[i-L]; // z[i] will fall in the window.
17             else { // z[i] can fall outside the window, try to increase the window.
18                 L = i;
19                 while(R < n && s[R] == s[R-L]) R++;
20                 R--;
21                 z[i] = R - L + 1;
22             }
23         }
24         if(z[i] == m) { // Match found.
25             //echo("Pattern found at: ", i-m-1);
26         }
27     }
28 }

```

Ad-hoc

```

1 int is_leap_year(int y) {
2     if(y%4 || (y%100==0 && y%400)) return 0;
3     return 1;
4 }
5 int days_month[12] = {31, 28, 31, 30, 31, 30, 31,
6     ↪ 31, 30, 31, 30, 31};
7 int days_month_accumulate[12] = {31, 59, 90, 120,
8     ↪ 151, 181, 212, 243, 273, 304, 334, 365};
9 // d 1-index, m 1-index.
10 int date_to_num(int d, int m, int y) {
11     m -= 2;
12     int sum = d;
13     if(m >= 1) sum += is_leap_year(y);
14     y--;
15     if(m >= 0) sum += days_month_accumulate[m];
16     if(y >= 0) {
17         sum += y/4 - y/100 + y/400;
18     }
19     return sum;
20 }
21 int nd, nm, ny; // Tiny optimization, binary search
22     ↪ the year, month and day.
23 void num_to_date(int num) {
24     nd = 1; nm = 1; ny = 2020; // The date searched
25     ↪ is >= this date.
26     while(date_to_num(nd, nm, ny) <= num) ny++;
27     ny--;
28     while(nm < 12 && date_to_num(nd, nm, ny) <=
29     ↪ num) nm++;
30     nm--;
31     while(date_to_num(nd, nm, ny) <= num) nd++;
32     nd--;
33 }

```

UFDS

```

1 void initialize(int n) {
2     for (int i = 0; i < n; ++i) {
3         rankk[i] = 0;
4         parent[i] = i;
5     }
6 }
7 int find(int x) {
8     if (parent[x] == x)
9         return x;
10    else
11        return parent[x] =
12        ↪ find(parent[x]);
13    }
14 void Union(int a, int b) {
15     int pa = find(a);
16     int pb = find(b);
17     if (pa == pb) {
18         return;
19     }
20     if (rankk[pa] > rankk[pb]) {
21         parent[pb] = pa;
22     } else if (rankk[pa] <
23     ↪ rankk[pb]) {
24         parent[pa] = pb;
25     } else {
26         parent[pa] = pb;
27         rankk[pb]++;
28     }
29 }

```