**SCC (Tarjan):**

1. /// Cycle contains which scc node belongs too.
3. **struct** SCC{
4. **int** num[NODE], low[NODE], col[NODE], cycle[NODE], st[NODE];
5. **int** tail, cnt, cc;
6. vi adj[NODE];
8. SCC():tail(0),cnt(0),cc(0) {}
9. **void** clear () {
10. cc += 3;
11. FOR(i,0,NODE-1) adj[i].clear();
12. tail = 0;
13. }
14. **void** tarjan ( **int** s ) {
15. num[s] = low[s] = cnt++;
16. col[s] = cc + 1;
17. st[tail++] = s;
18. FOR(i,0,SZ(adj[s])-1) {
19. **int** t = adj[s][i];
20. **if** ( col[t] <= cc ) {
21. tarjan ( t );
22. low[s]=MIN(low[s],low[t]);
23. }
24. /\*Back edge\*/
25. **else** **if** (col[t]==cc+1)
26. low[s]=MIN(low[s],low[t]);
27. }
28. **if** ( low[s] == num[s] ) {
29. **while** ( 1 ) {
30. **int** temp=st[tail-1];
31. tail--;
32. col[temp] = cc + 2;
33. cycle[temp] = s;
34. **if** ( s == temp ) **break**;
35. }
36. }
37. }
38. **void** shrink( **int** n ) {
39. FOR(i,0,n){
40. FOR(j,0,SZ(adj[i])-1){
41. adj[i][j] = cycle[adj[i][j]]; ///Careful. This will create self-loop. Just ignore i->i edges when processing.
42. }
43. }
44. FOR(i,0,n){
45. **if** ( cycle[i] == i ) **continue**;
46. **int** u = cycle[i];
47. FOR(j,0,SZ(adj[i])-1){
48. **int** v = adj[i][j];
49. adj[u].pb ( v );
50. }
51. adj[i].clear();
52. }
53. FOR(i,0,n){ ///Not always necessary
54. sort ( ALL(adj[i]) );
55. UNIQUE(adj[i]);
56. }
57. }
59. **void** findSCC( **int** n ) {
60. FOR(i,0,n) {
61. **if** ( col[i] <= cc ) {
62. tarjan ( i );
63. }
64. }
65. }
66. };

**2SAT:**

1. /\*\*
2. 1. The nodes need to be split. So change convert() accordingly.
3. 2. Using clauses, populate scc edges.
4. 3. Call possible, to find if a valid solution is possible or not.
5. 4. Dont forget to keep space for !A variables
6. \*\*/
7. **struct** SAT2 {
8. SCC scc;
10. SAT2(): bfscc(1) {}
12. **void** clear(**int** n) {
13. scc.clear(**int** n);
14. }
16. **int** convert(**int** n) { ///Change here. Depends on how input is provided
17. **int** x = ABS(n);
18. x--;
19. x \*= 2;
20. **if** (n < 0) x ^= 1;
21. **return** x;
22. }
24. **void** mustTrue(**int** a) { ///A is True
25. scc.adj[a ^ 1].pb(a);
26. }
27. **void** orClause(**int** a, **int** b) { /// A || B clause
28. //!a->b !b->a
29. scc.adj[a ^ 1].pb(b);
30. scc.adj[b ^ 1].pb(a);
31. }
32. /// Out of all possible option, only one is true
33. **void** atMostOneClause(**int** a[], **int** n, **int** flag) {
34. **if** (flag == 0) { /// At most one can be false
35. FOR(i, 0, n) {
36. a[i] = a[i] ^ 1;
37. }
38. }
39. FOR(i, 0, n) {
40. FOR(j, i + 1, n) {
41. orClause(a[i] ^ 1, a[j] ^ 1); /// !a || !b both being true not allowed
42. }
43. }
44. }
46. ///Send n, total number of nodes, after expansion
47. **bool** possible(**int** n) {
48. scc.findSCC(n);
50. FOR(i, 0, n) {
51. **int** a = i, b = i ^ 1;
52. ///Falls on same cycle a and !a.
53. **if** (scc.cycle[a] == scc.cycle[b]) **return** **false**;
54. }
56. ///Valid solution exists
57. **return** **true**;
58. }
60. ///To determine if A can be true. It cannot be true, if a path exists from A to !A.
61. **int** vis[SAT2NODE], qqq[SAT2NODE], bfscc;
62. **void** bfs(**int** s) {
63. bfscc++;
64. **int** qs = 0, qt = 0;
65. vis[s] = bfscc;
66. qqq[qt++] = s;
67. **while** (qs < qt) {
68. s = qqq[qs++];
69. FOR(i, 0, SZ(scc.adj[s]) - 1) {
70. **int** t = scc.adj[s][i];
71. **if** (vis[t] != bfscc) {
72. vis[t] = bfscc;
73. qqq[qt++] = t;
74. }
75. }
76. }
77. }
79. }sat2;

**Aho Corasick:**

1. /// Aho Corasick
2. /// !!! Works with lowercase letters !!!
3. /// !!! If RTE, bring struct node outside !!!
5. **class** AhoCorasick
6. {
7. #define SIZE asdf
8. **struct** node
9. {
10. **int** val;
11. **int** child[26];
12. **void** clear()
13. {
14. memset(child,0,**sizeof** child);
15. }
16. } trie[SIZE];
18. **int** curNode, root, fail[SIZE], par[SIZE];
20. **public**:
21. **void** clear()
22. {
23. root = 0;
24. curNode = 0;
25. trie[root].clear();
26. }
28. **void** addTrie ( string s )
29. {
30. **int** len = s.size();
32. **int** cur = root;
33. **for** ( **int** i = 0; i < len; i++ )
34. {
35. **char** c = s[i] - 'a';
36. **if** ( trie[cur].child[c] == 0 )
37. {
38. curNode++;
39. trie[curNode].clear();
40. trie[curNode].val = c;
41. trie[cur].child[c] = curNode;
42. }
43. cur = trie[cur].child[c];
44. }
46. ///Mark the node cur as finishing point, if needed
47. }
49. **void** calcFail()
50. {
51. queue<**int**> q;
52. q.push ( 0 );
54. **while** ( !q.empty() )
55. {
56. **int** s = q.front();
57. q.pop();
59. ///Push all child to queue
60. **for** ( **int** i = 0; i < 26; i++ )
61. {
62. **int** t = trie[s].child[i];
63. **if** ( t != 0 )
64. {
65. q.push ( t );
66. par[t] = s;
67. }
68. }
69. ///Calculate failure of current node s
70. **if** ( s == 0 )   ///Special case
71. {
72. fail[s] = 0;
73. par[s] = 0;
74. **continue**;
75. }
77. **int** p = par[s];
78. **int** f = fail[p];
79. **int** v = trie[s].val;
80. **while** ( f != 0 && trie[f].child[v] == 0 ) f = fail[f]; ///Keep on jumping until found
82. fail[s] = trie[f].child[v];
83. **if** ( fail[s] == s ) fail[s] = 0; ///Can't be fail of itself
84. }
85. }
86. **int** goTo ( **int** state, **int** c )
87. {
88. **if** ( trie[state].child[c] != 0 )
89. {
90. **return** trie[state].child[c];
91. }
92. ///Fall back
93. **int** f = fail[state];
94. **while** ( f != 0 && trie[f].child[c] == 0 )
95. {
96. f = fail[f];
97. }
98. **return** trie[f].child[c];
99. }
100. } aho;

**Articulation Point:**

1. #define ARTNODE 10010
3. **class** ArticulationPoint {
4. **int** disc[ARTNODE]; ///Discovery time of nodes
5. **int** low[ARTNODE]; ///Lowest back edge extension
6. **int** col[ARTNODE]; ///Color for marking node
8. **int** cnt; ///Timer
9. **int** cc; ///Color
10. **int** root; ///Root of tree
12. **void** tarjan(**int** s, **int** p) {
13. disc[s] = low[s] = cnt++;
14. col[s] = cc + 1;
16. **int** child = 0; ///Needed for root only
17. **int** art = 0;
18. **for** (**int** i = 0; i < adj[s].size(); ++i) {
20. **int** t = adj[s][i];
21. **if** (t == p) **continue**; ///Don't go to parent
23. **if** (col[t] <= cc) { ///New node. Discovery.
24. child++;
26. tarjan(t, s);
27. low[s] = min(low[s], low[t]); ///Update back edge extension for S
29. **if** (low[t] >= disc[s]) { ///Back edge of T did not go above S
30. art++; ///S is articulation point for T
31. }
32. } **else** **if** (col[t] == cc + 1) { ///Back Edge
33. low[s] = min(low[s], disc[t]);
34. }
35. }
37. **if** ((s == root && child > 1) || (s != root && art)) {
38. ///Edit in this block
39. printf("This is a articulation point: %d\n", s);
40. }
41. }
43. **public**:
45. vector < **int** > adj[ARTNODE];
47. **void** clear(**int** n) {
48. cc += 2; ///cc is now 0. cc+1 is 1
49. **for** (**int** i = 0; i <= n; i++) {
50. adj[i].clear();
51. }
52. }
54. **void** findArt(**int** n, **int** start = 0) {
55. **for** (**int** i = start; i <= n; i++) {
56. **if** (col[i] <= cc) {
57. root = i;
58. tarjan(i, -1);
59. }
60. }
61. }
62. }art;

**BIT:**

1. /\*\*
2. Make sure BIT is 1-indexed
3. 1. Properly change update and read
4. 2. Call clear() and set row and col
5. \*\*/
6. **struct** BIT {
7. **int** arr2D[XXX][XXX], arr[XXX];
8. **int** row, col;
10. **void** clear( **int** r, **int** c ){
11. row = r;
12. col = c;
13. CLR(arr,0);
14. }
16. ///If I need sum[1-x], I will update from [x-n]. Opposite for reverse.
17. **void** update ( **int** x, **int** val ) {
18. **for** ( **int** i = x; i <= col; i += i & -i ) arr[i] += val;
19. }
20. **int** read ( **int** x ) {
21. **int** res = 0;
22. **for** ( **int** i = x; i > 0; i -= i & -i ) res += arr[i];
23. }
25. ///If I need sum[1 x], I will update from [x-n]. Opposite for reverse.
26. **void** update2D ( **int** x, **int** y, **int** val ) {
27. **for** ( **int** i = x; i > 0; i -= i & -i ) { ///Change Here. Both increment and loop condition
28. **for** ( **int** j = y; j <= col; j += j & -j ) { ///And here
29. arr[i][j] += val;
30. }
31. }
32. }
34. ///Just reverse direction of update
35. **int** read2D ( **int** x, **int** y ) {
36. **int** res = 0;
37. **for** ( **int** i = x; i <= row; i += i & -i ) { ///Change Here. Both increment and loop condition
38. **for** ( **int** j = y; j > 0; j -= j & -j ) { ///And here
39. res += arr[i][j];
40. }
41. }
42. **return** res;
43. }
44. }bit;

**Bridge:**

1. #define BRIDGENODE 10010
3. **class** BridgeFinding {
4. **int** disc[BRIDGENODE]; ///Discovery time of nodes
5. **int** low[BRIDGENODE]; ///Lowest back edge extension
6. **int** col[BRIDGENODE]; ///Color for marking node
8. **int** cnt; ///Timer
9. **int** cc; ///Color
11. **void** tarjan ( **int** s, **int** parentEdge ) {
12. disc[s] = low[s] = cnt++;
13. col[s] = cc + 1;
15. **for** ( **int** i = 0; i < adj[s].size(); ++i ) {
17. **int** t = adj[s][i].ff;
18. **int** edgeNumber = adj[s][i].ss;
19. **if** ( edgeNumber == parentEdge ) **continue**; ///Don't use the same edge from parent
21. **if** ( col[t] <= cc ) { ///New node. Discovery.
23. tarjan ( t, edgeNumber );
24. low[s]=min(low[s],low[t]); ///Update back edge extension for S
26. **if** ( low[t] > disc[s] ) { ///Back edge of T did not go above S
27. ///This edge is Bridge
28. }
29. }
30. **else** **if** ( col[t] == cc + 1 ) { ///Back Edge
31. low[s]=min(low[s],disc[t]);
32. }
33. }
34. }
36. **public**:
38. vector< pair<**int**,**int**> > adj[BRIDGENODE]; ///Enter target and edge number as pair
40. **void** clear ( **int** n ) {
41. cc += 2; ///cc is now 0. cc+1 is 1
42. **for** (**int** i = 0; i <= n; i++ ) {
43. adj[i].clear();
44. }
45. }
47. **void** findBridge( **int** n, **int** start = 0 ) {
48. **for** ( **int** i = start; i <= n; i++ ) {
49. **if** ( col[i] <= cc ) {
50. tarjan ( i, -1 );
51. }
52. }
53. }
54. }bridge;

**Calender:**

1. **struct** CALENDAR {
2. **int** month[13];
3. **int** d, m, y, daysInYear;
4. **int** day;
6. CALENDER() {
7. month[1] = 31; month[2] = 28; month[3] = 31;
8. month[4] = 30; month[5] = 31; month[6] = 30;
9. month[7] = 31; month[8] = 31; month[9] = 30;
10. month[10] = 31; month[11] = 30; month[12] = 31;
11. daysInYear = 365;
12. }
13. **void** assign(**int** a, **int** b, **int** c, **int** \_day = 0) {
14. d = a;
15. m = b;
16. y = c;
17. day = \_day;
18. updateFeb();
19. }
21. **bool** isLeapYear(**int** n) {
22. **if** (n % 400 == 0 || (n % 4 == 0 && n % 100 != 0)) **return** **true**;
23. **else** **return** **false**;
24. }
25. //Call this whenever year changes
26. **void** updateFeb() {
27. **if** (isLeapYear(y)) {
28. month[2] = 29;
29. daysInYear = 366;
30. } **else** {
31. month[2] = 28;
32. daysInYear = 365;
33. }
34. }
35. **bool** operator < (**const** CALENDER & b) **const** {
36. **if** (y < b.y) **return** **true**;
37. **else** **if** (y == b.y && m < b.m) **return** **true**;
38. **else** **if** (y == b.y && m == b.m && d < b.d) **return** **true**;
39. **else** **return** **false**;
40. }
42. //Increase 1 day
43. **void** increment() {
44. d++;
45. day = (day + 1) % 7;
47. **if** (d > month[m]) {
48. d = 1;
49. m++;
50. **if** (m > 12) {
51. m = 1;
52. y++;
53. updateFeb();
54. }
55. }
56. }
58. //Number of days between 1/1/a to 31/12/b inclusive
59. **int** daysBetweenYear(**int** a, **int** b) {
60. **int** p = b - a + 1;
61. **int** res = p \* 365;
62. //Add 1 day for each leap year between [a,b].
64. //Leap year between [0,b] - [0,a-1]
65. **int** leap = b / 4 - b / 100 + b / 400;
66. **if** (a) {
67. a--;
68. leap -= a / 4 - a / 100 + a / 400;
69. }
71. **return** res + leap;
72. }
74. //Return number of days between two dates. Go from b to \*this.
75. **int** diff(CALENDER b) {
76. **if** (y == b.y && m == b.m && d == b.d) **return** 0;
77. **if** ( \* **this** < b) **return** b.diff( \* **this**);
79. **int** res = 0;
80. //The year is not same yet
81. **if** (b.y < y) {
82. //So we can move to next year, one day at a time.
83. **while** (b.d != 1 || b.m != 1) {
84. b.increment();
85. res++;
86. }
87. res += daysBetweenYear(b.y, y - 1);
88. b.y = y;
89. b.updateFeb();
90. }
92. //Same year
93. **while** (b.d != d || b.m != m) {
94. b.increment();
95. res++;
96. }
98. **return** res;
99. }
101. //Add n days to current date
102. CALENDER add(**int** n) {
103. CALENDER res = \* **this**;
105. **if** (n < 800) {
106. **while** (n) {
107. res.increment();
108. n--;
109. }
110. **return** res;
111. }
113. **while** (res.d != 1 || res.m != 1) {
114. n--;
115. res.increment();
116. }
118. **while** (n > 366) {
119. **int** jump = n / 366;
120. **if** (jump) {
121. **int** jd = daysBetweenYear(res.y, res.y + jump - 1);
122. n -= jd;
123. res.day = (res.day + jd) % 7;
124. res.y = res.y + jump;
125. res.updateFeb();
126. }
127. }
129. **while** (n) {
130. res.increment();
131. n--;
132. }
134. **return** res;
135. }
136. };

**Convex Hull Trick:**

1. /\* Instructions
2. 1. Sort lines based on decreasing M and in case of tie, increasing B.
3. 2. Sort query points according to increasing X.
4. 3. Clear the class, add all lines and then query.
5. \*/
7. **class** ConvexHullTrick {
8. **int** pointer; //Keeps track of the best line from previous query
9. vector<**long** **long**> M; //Holds the slopes of the lines in the envelope
10. vector<**long** **long**> B; //Holds the y-intercepts of the lines in the envelope
12. //Returns true if line l3 is always better than line l2
13. **bool** bad(**int** l1,**int** l2,**int** l3){
14. /\*
15. intersection(l1,l2) has x-coordinate (b1-b2)/(m2-m1)
16. intersection(l1,l3) has x-coordinate (b1-b3)/(m3-m1)
17. set the former greater than the latter, and cross-multiply to
18. eliminate division
19. \*/
20. **return** (B[l3]-B[l1])\*(M[l1]-M[l2])<(B[l2]-B[l1])\*(M[l1]-M[l3]);
21. }
23. **public**:
25. **void** clear() {
26. pointer = 0;
27. M.clear();
28. B.clear();
29. }
31. //Adds a new line (with lowest slope) to the structure
32. **void** add(**long** **long** m,**long** **long** b){
33. **if** ( M.size() > 0 && M.back() == m ) **return**; ///Same Gradient. Don't add.
35. //First, let's add it to the end
36. M.push\_back(m);
37. B.push\_back(b);
38. //If the penultimate is now made irrelevant between the antepenultimate
39. //and the ultimate, remove it. Repeat as many times as necessary
40. **while** (M.size()>=3&&bad(M.size()-3,M.size()-2,M.size()-1)){
41. M.erase(M.end()-2);
42. B.erase(B.end()-2);
43. }
44. }
46. //Returns the minimum y-coordinate of any intersection between a given vertical
47. //line and the lower envelope
48. **long** **long** query(**long** **long** x){
49. //Any better line must be to the right, since query values are
50. //non-decreasing
51. **while** (pointer<M.size()-1&&
52. M[pointer+1]\*x+B[pointer+1]<M[pointer]\*x+B[pointer])
53. pointer++;
54. **return** M[pointer]\*x+B[pointer];
55. }
56. };

**Dice:**

1. //The dice is sitting on its bottom and I can see the front.
2. **struct** DICE {
3. **int** bottom = 0, top = 1, front = 2, right = 3, back = 4, left = 5; //Names of Faces
4. **int** face[6]; //Value of faces
6. **void** assign(**int** \_bot, **int** \_top, **int** \_front, **int** \_right, **int** \_back, **int** \_left) {
7. face[0] = \_bot; face[1] = \_top;
8. face[2] = \_front; face[3] = \_right;
9. face[4] = \_back; face[5] = \_left;
10. }
12. //All rotation anti-clockwise
13. DICE rotateZ() {
14. DICE res;
15. res.assign(face[0], face[1], face[5], face[2], face[3], face[4]);
16. **return** res;
17. }
18. DICE rotateX() {
19. DICE res;
20. res.assign(face[back], face[front], face[bottom], face[right], face[top], face[left]);
21. **return** res;
22. }
23. DICE rotateY() {
24. DICE res;
25. res.assign(face[left], face[right], face[front], face[bottom], face[back], face[top]);
26. **return** res;
27. }
29. //Comparer
30. **bool** operator == (**const** DICE & b) **const** {
31. FOR(i, 0, 5) {
32. **if** (face[i] != b.face[i]) **return** **false**;
33. }
34. **return** **true**;
35. }
36. **bool** operator < (**const** DICE & b) **const** {
37. FOR(i, 0, 5) {
38. **if** (face[i] < b.face[i]) **return** **true**;
39. **else** **if** (face[i] > b.face[i]) **return** **false**;
40. }
41. **return** **false**;
42. }
44. //Find the original Scheme of the DICE
45. DICE diceScheme() {
46. DICE res = \* **this**;
47. DICE temp = \* **this**;
49. FOR(i, 0, 3) {
50. temp = temp.rotateX();
51. FOR(j, 0, 3) {
52. temp = temp.rotateZ();
53. **if** (temp < res) res = temp;
54. }
55. }
56. FOR(i, 0, 3) {
57. temp = temp.rotateY();
58. FOR(j, 0, 3) {
59. temp = temp.rotateZ();
60. **if** (temp < res) res = temp;
61. }
62. }
63. **return** res;
64. }
66. **void** print() {
67. FOR(i, 0, 5) printf("%d ", face[i]);
68. nl;
69. }
70. };

**FFT:**

1. **typedef** complex<**double**> base;
3. **void** fft (vector<base> & a, **bool** invert) {
4. **int** n = (**int**) a.size();
5. **if** (n == 1)  **return**;
7. vector<base> a0 (n/2), a1 (n/2);
8. **for** (**int** i=0, j=0; i<n; i+=2, ++j) {
9. a0[j] = a[i];
10. a1[j] = a[i+1];
11. }
12. fft (a0, invert);
13. fft (a1, invert);
15. **double** ang = 2\*PI/n \* (invert ? -1 : 1);
16. base w (1), wn (cos(ang), sin(ang));
17. **for** (**int** i=0; i<n/2; ++i) {
18. a[i] = a0[i] + w \* a1[i];
19. a[i+n/2] = a0[i] - w \* a1[i];
20. **if** (invert)
21. a[i] /= 2, a[i+n/2] /= 2;
22. w \*= wn;
23. }
24. }

27. /\*
28. 1. Build two different vectors with coefficient of the equation ( 3 + x + 2x^2 will be [3,1,2] )
29. 2. Send them to multiply ( arr, brr, res )
30. 3. res vector contains result of multiplication.
32. WARNING: FFT causes precision error. multiply() rounds them to integer.
33. \*/
35. **void** multiply (**const** vector<**int**> & a, **const** vector<**int**> & b, vector<**int**> & res) {
36. vector<base> fa (a.begin(), a.end()), fb (b.begin(), b.end());
37. **size\_t** n = 1;
38. **while** (n < max (a.size(), b.size())) n <<= 1;
39. n <<= 1;
40. fa.resize (n), fb.resize (n);
42. fft (fa, **false**), fft (fb, **false**);
43. **for** (**size\_t** i=0; i<n; ++i)
44. fa[i] \*= fb[i];
45. fft (fa, **true**);
47. res.resize (n);
48. **for** (**size\_t** i=0; i<n; ++i)
49. res[i] = **int** (fa[i].real() + 0.5);
50. }

**Flow:**

1. **struct** node {
2. **int** x, y, next, cap, cost;
3. };
5. /\*
6. 1. Clear graph
7. 2. Add edge
8. 3. Assign source and sink
9. 4. Pass highestNumberOfNode to maxFlow() or minCostMaxFlow()
10. \*/
11. **struct** FLOW {
12. **int** source, sink;
14. **int** head[NODE];
15. **void** clear() {
16. e = 0;
17. CLR(head,-1);
18. }
20. node edge[EDGE]; **int** e;
21. **void** addEdge ( **int** u, **int** v, **int** cap, **int** cap2, **int** cost ) {
22. edge[e].x = u; edge[e].y = v; edge[e].cap = cap; edge[e].cost = cost;
23. edge[e].next = head[u]; head[u] = e; e++;
24. edge[e].x = v; edge[e].y = u; edge[e].cap = cap2; edge[e].cost = -cost;
25. edge[e].next = head[v]; head[v] = e; e++;
26. }
28. **int** vis[NODE], q[NODE], now[NODE];
29. **bool** bfs ( ) {
30. memset ( vis, -1, **sizeof** vis );
31. vis[source] = 0;
32. **int** ini = 0, qend = 0;
33. q[qend++] = source;
35. **while** ( ini < qend && vis[sink] == -1 ) {
36. **int** s = q[ini++];
37. **int** i;
38. **for** (i=head[s];i!=-1;i= edge[i].next){
39. **int** t = edge[i].y;
40. **if** ( vis[t] == -1 && edge[i].cap){
41. vis[t] = vis[s] + 1;
42. q[qend++] = t;
43. }
44. }
45. }
46. **if** ( vis[sink] != -1 ) **return** **true**;
47. **else** **return** **false**;
48. }
49. **int** dfs ( **int** s, **int** f ) {
50. **if** ( f == 0 ) **return** 0;
51. **if** ( s == sink ) **return** f;
52. **for** ( **int** &i=now[s];i!=-1;i=edge[i].next){
53. **int** t = edge[i].y;
54. **if** ( vis[s] + 1 != vis[t] ) **continue**;
55. **int** pushed=dfs(t,MIN(f,edge[i].cap));
56. **if** ( pushed ) {
57. edge[i].cap -= pushed;
58. edge[i^1].cap += pushed;
59. **return** pushed;
60. }
61. }
62. **return** 0;
63. }
65. **int** maxFlow ( **int** highestNumberOfNode, **int** flow ) {
66. **int** res = 0;
67. **while** ( 1 ) {
68. **if** ( flow == 0 ) **break**;
69. **if** ( bfs () == **false** ) **break**;
70. **int** i;
71. **for** ( i=0;i<=highestNumberOfNode;i++)now[i]= head[i];
72. **while** (**int** pushed=dfs(source,flow ) ) {
73. res += pushed; ///Can overflow depending on Max Flow
74. flow -= pushed;
75. }
76. }
77. **return** res;
78. }
80. **int** inq[NODE], par[NODE], record[NODE];
81. **int** minCostFlow ( **int** highestNumberOfNode ) {
82. **int** res = 0, i, j, k, fl = 0;
83. **while** ( 1 ) {
84. **for** ( i = 0; i <= highestNumberOfNode; i++ ) vis[i] = inf;
85. vis[source] = 0;
86. deque < **int** > dq;
87. dq.pb ( source );
88. **while** ( !dq.empty() ) {
89. **int** s = dq.front(); dq.pop\_front();
90. inq[s] = 0;
91. **for** ( i = head[s]; i != -1; i = edge[i].next ) {
92. **int** t = edge[i].y;
93. **if** ( edge[i].cap ) {
94. **if** ( vis[s] + edge[i].cost < vis[t] ) {
95. vis[t] = vis[s] + edge[i].cost;
96. par[t] = s;
97. record[t] = i;
98. **if** ( inq[t] == 0 ) {
99. inq[t] = 1;
100. **if** ( dq.empty() == **false** && vis[dq.front()] > vis[t] )
101. dq.push\_front( t );
102. **else** dq.pb ( t );
103. }
104. }
105. }
106. }
107. }
109. //if ( vis[sink] > 0 ) break; ///Cost Getting minimized, Change Here
110. **if** ( vis[sink] == inf ) **break**; //Flow getting maximized. Either this, or the one above
112. fl++; ///Total flow
113. res += vis[sink]; ///Cost of Flow
114. **for** ( i = sink; i != source; i = par[i] ) { //Travel from sink to source
115. **int** t = record[i]; //Record which edge was used to travel to t
116. edge[t].cap--;
117. edge[t^1].cap++;
118. }
119. }
121. **return** res; ///Min cost of max flow
122. }
124. }graph;
126. //Kuhn's Algo for finding Maximum matching
127. **struct** KUHN{
128. **int** left[NODE], right[NODE], vis[2\*NODE], cc; ///Node double in vis for vertex print
129. vector<**int**> adj[NODE];
131. KUHN() : cc(1) {}
133. **void** clear( **int** n ) {
134. FOR(i,0,n) adj[i].clear();
135. }
137. **bool** tryK ( **int** v ) {
138. **if** ( vis[v] == cc ) **return** **false**;
139. vis[v] = cc;
140. **for** ( **int** i = 0; i < SZ(adj[v]); i++ ) {
141. **int** t = adj[v][i];
142. **if** ( right[t] == -1 ) {
143. right[t] = v; left[v] = t;
144. **return** **true**;
145. }
146. }
147. **for** ( **int** i = 0; i < SZ(adj[v]); i++ ) {
148. **int** t = adj[v][i];
149. **if** ( tryK ( right[t] ) ) {
150. right[t] = v; left[v] = t;
151. **return** **true**;
152. }
153. }
154. **return** **false**;
155. }
157. **int** match(**int** n) {
158. **int** res = 0;
159. **bool** done;
160. CLR(left,-1); CLR(right,-1);
161. **do** {
162. done = **true**; cc++;
163. FOR(i,0,n) {
164. **if**(left[i]==-1 && tryK(i)) {
165. done = **false**;
166. }
167. }
168. } **while**(!done);
169. FOR(i,0,n) res += (left[i]!=-1);
170. **return** res;
171. }
173. ///Careful. Loop runs from 0 to n-1
174. ///Make sure match() has been run
175. **int** lcover[NODE], rcover[NODE];
176. **void** findVertexCover ( **int** n ) {
177. queue<**int**> q;
178. cc++;
179. FOR(i,0,n-1){
180. **if** ( left[i] == -1 ) {
181. q.push ( i );
182. vis[i] = cc;
183. }
184. };
186. **while** ( !q.empty() ) {
187. **int** s = q.front(); q.pop();
188. FOR(i,0,SZ(adj[s])-1){
189. **int** t = adj[s][i];
190. **if** ( t == left[s] ) **continue**;
192. **int** xt = t + n;
193. **if** ( vis[xt] == cc ) **continue**;
195. vis[xt] = cc;
196. xt = right[t];
198. **if** ( xt != -1 && vis[xt] != cc ) {
199. vis[xt] = cc;
200. q.push ( xt );
201. }
203. };
204. }
206. FOR(i,0,n-1){
207. **if** ( vis[i] != cc ) lcover[i] = 1;
208. **else** lcover[i] = 0;
209. };
210. FOR(i,0,n-1){
211. **if** ( vis[i+n] == cc ) rcover[i] = 1;
212. **else** rcover[i] = 0;
213. };
215. }
216. }kuhn;

**Fraction Class:**

1. **class** Fraction {
2. vlong num, denom;
4. **void** simplify() {
5. vlong g = gcd ( num, denom );
6. num /= g;
7. denom /= g;
8. **if** ( denom < 0 ) {
9. num \*= -1;
10. denom \*= -1;
11. }
12. }
14. **public**:
16. ///First deal with initiation of the class
17. Fraction () {
18. num = 0;
19. denom = 1;
20. }
21. Fraction ( vlong a, vlong b ) {
22. num = a;
23. denom = b;
24. simplify();
25. }
26. Fraction ( vlong x ) {
27. num = x;
28. denom = 1;
29. }
30. Fraction ( pll x ) {
31. num = x.ff;
32. denom = x.ss;
33. simplify();
34. }
35. Fraction ( **const** Fraction &b ) {
36. num = b.num;
37. denom = b.denom;
38. }
39. **void** operator = ( vlong x ) {
40. num = x;
41. denom = 1;
42. }
43. **void** operator = ( pll x ) {
44. num = x.ff;
45. denom = x.ss;
46. simplify();
47. }
48. **void** operator = ( Fraction b ) {
49. num = b.num;
50. denom = b.denom;
51. }
53. ///Basic Arithmetic operations
54. Fraction operator - () { ///Negation
55. **return** Fraction( -num, denom );
56. }
57. Fraction operator + ( Fraction b ) { ///Addition
58. Fraction res;
59. res.denom = abs( LCM(denom,b.denom) );
60. res.num = (res.denom/denom)\*num + (res.denom/b.denom)\*b.num;
61. res.simplify();
62. **return** res;
63. }
64. Fraction operator - ( Fraction b ) { ///Subtraction
65. **return** (\***this**) + (-b);
66. }
67. Fraction operator \* ( Fraction b ) {
68. Fraction res ( b.num / gcd(b.num,denom), b.denom / gcd(b.denom,num) );
69. res.num \*= num / gcd(num,b.denom);
70. res.denom \*= denom / gcd(denom,b.num);
71. res.simplify();
72. **return** res;
73. }
74. Fraction operator / ( Fraction b ) { ///Division
75. Fraction rev ( b.denom, b.num );
76. **if** ( rev.denom < 0 ) {
77. rev.denom \*= -1;
78. rev.num \*= -1;
79. }
80. **return** (\***this**)\*rev;
81. }
83. ///Basic Arithmetic Operations overloaded
84. **void** operator += ( Fraction b ) {
85. \***this** = \***this** + b;
86. }
87. **void** operator -= ( Fraction b ) {
88. \***this** = \***this** - b;
89. }
90. **void** operator \*= ( Fraction b ) {
91. \***this** = \***this** \* b;
92. }
93. **void** operator /= ( Fraction b ) {
94. \***this** = \***this** / b;
95. }
97. ///Comparison
98. **bool** operator == ( Fraction b ) {
99. **if** ( num == b.num && denom == b.denom ) **return** **true**;
100. **else** **return** **false**;
101. }
102. **bool** operator < ( Fraction b ) {
103. **if** ( num \* b.denom < b.num \* denom ) **return** **true**;
104. **else** **return** **false**;
105. }
106. **bool** operator > ( Fraction b ) {
107. **return** ( b < \***this** );
108. }
109. **bool** operator <= ( Fraction b ) {
110. **if** ( \***this** == b || \***this** < b ) **return** **true**;
111. **else** **return** **false**;
112. }
113. **bool** operator >= ( Fraction b ) {
114. **return** ( b <= \***this** );
115. }
117. ///Output
118. **void** print() {
119. printf ( "%lld/%lld %lf\n",num, denom, num/(denom-0.0) );
120. }
122. Fraction getAbs() {
123. **return** Fraction(abs(num),denom);
124. }
125. };

**Gaussian Elimination:**

1. /\*
2. 1. Set row and col of mat
3. 2. Call rank() to perform gauss-elimination and find rank
4. 3. Call isValid() to find if solution exists.
6. Careful about int a[x][x]. If mod^2 crosses int, take vlong
7. If mod is 2, it is better to use XOR since it a lot faster.
8. \*/
9. **struct** GAUSS {
10. **int** row, col;
11. vlong a[x][x];
12. **int** mod;
13. **bool** valid;
14. GAUSS() {
15. mod = xyz;
16. }
17. **void** clear() {
18. memset(a, 0, **sizeof** a);
19. }
20. **void** isValid(**int** st) {
21. **int** i;
22. valid = **true**;
23. **for** (i = st; i < row; i++) {
24. **if** (a[i][col - 1]) {
25. valid = **false**;
26. **return**;
27. }
28. }
29. }
30. ///Return Rank of Matrix
31. ///Free variable = Variable - Rank or Col - Rank - 1
32. **int** rank() {
33. **int** i = 0, j = 0, k, r, u;
34. **while** (i < row && j < col - 1) {
35. r = i;
36. **for** (k = i; k < row; k++)
37. **if** (a[k][j]) {
38. r = k; ///Find non-zero coefficient
39. **break**;
40. }
41. **if** (a[r][j]) {
42. **if** (r != i) ///Swap row if required
43. **for** (k = 0; k < col; k++)
44. swap(a[r][k], a[i][k]);
46. ///Neutralize if required. Depends on whether double or modular division
47. vlong v = a[i][j];
48. v = modInv(v, mod);
49. **for** (u = j; u < col; u++) {
50. a[i][u] = (a[i][u] \* v) % mod;
51. }
52. /\*
53. double v = a[i][j];
54. for ( u = j; u < col; u++ ) {
55. a[i][u] /= v;
56. }
57. \*/
59. **for** (u = i + 1; u < row; u++)
60. **if** (a[u][j]) { ///Eliminate
61. **int** v = a[u][j];
62. **for** (k = j; k < col; k++) {
63. a[u][k] = ((a[i][k] \* v) - a[u][k]) % mod; ///Change here if no mod
64. **if** (a[u][k] < 0) a[u][k] += mod;
65. }
66. }
67. i++;
68. }
69. j++;
70. }
71. **return** i;
72. }
74. **void** print() {
75. FOR(i, 0, row - 1) {
76. FOR(j, 0, col - 1) {
77. printf("%d ", a[i][j]);
78. }
79. nl;
80. }
81. }
82. }mat;

**Geometry Routine:**

1. #define MAXD 2
3. **double** cosineRule3Side ( **double** a, **double** b, **double** c ) {
4. **double** res = (SQ(a)+SQ(b)-SQ(c)) / (2\*a\*b);
5. **if** ( res < -1 ) res = -1; **if** ( res > 1 ) res = 1;
6. **return** acos ( res );
7. }
9. **struct** myVec {
10. **int** d; //Dimension
11. **double** val[MAXD];//Contains value of each component
13. myVec() {
14. d = x; ///Change Here
15. }
17. myVec add ( myVec b ) {
18. myVec res; FOR(i,0,d-1) res.val[i] = val[i] + b.val[i];**return** res;
19. }
20. myVec sub ( myVec b ) {
21. myVec res; FOR(i,0,d-1) res.val[i] = val[i] - b.val[i];**return** res;
22. }
23. myVec mul ( **double** t ) {
24. myVec res; FOR(i,0,d-1)res.val[i] = val[i] \* t;**return** res;
25. }
26. myVec div ( **double** t ) {
27. myVec res; FOR(i,0,d-1) res.val[i] = val[i] / t;**return** res;
28. }
29. **bool** operator == ( myVec b ) {
30. FOR(i,0,d-1) **if** ( fabs ( val[i] - b.val[i] ) > eps ) **return** **false**; **return** **true**;
31. }
32. myVec perp2D() {
33. myVec res = (\***this**);
34. swap ( res.val[0], res.val[1] );
35. res.val[0] \*= -1;
36. **return** res;
37. }
38. **double** dot ( myVec v ) { //Finds \*this (dot) v
39. **double** res = 0; **for** ( **int** i = 0; i < d; i++ ) res += val[i] \* v.val[i];
40. **return** res;
41. }
42. **double** length () { //Finds length of current vector
43. **return** sqrt ( **this**->dot( \***this** ) );
44. }
45. myVec unitVec () {
46. **return** (\***this**).div ( length() ); // v / ||v||
47. }
49. **double** angleBetween2D ( myVec b ) {
50. **double** pol1 = atan2 ( val[1], val[0] );
51. **double** pol2 = atan2 ( b.val[1], b.val[0] );
52. **if** ( pol2 + eps < pol1 ) pol2 += 2 \* pi;
54. **double** x = pol2 - pol1;
55. **if** ( x > pi + eps ) x = (2\*pi) - x;
57. //For direction, use sign of cross2D
58. **return** x;
59. }
61. //Causes precision error. Use angleBetween2D when 2D.
62. **double** angleBetween ( myVec b ) { //Angle between two vectors
63. **double** res = dot( b ) / ( length() \* b.length() );
64. **if** ( res > 1 ) res = 1; **if** ( res < -1 ) res = -1;
65. **return** acos (res);
66. }
67. **double** polarAngle2D() { //Angle from x-axis
68. **double** res = atan2 ( val[1], val[0] );
69. **if** ( res + eps < 0 ) res += 2 \* pi;
70. **return** res;
71. }
72. **double** cross2D ( myVec v ) { //Cross the two values. Only for 2D. Z compo 0.
73. **return** val[0]\*v.val[1] - val[1]\*v.val[0];
74. }
76. //Provided, a comes before b. Otherwise, need to swap
77. **bool** between ( myVec a, myVec b ) {
78. **if** ( val[0] + eps < a.val[0] || val[0] > b.val[0] + eps ) **return** **false**;
79. **if** ( val[1] + eps < a.val[1] || val[1] > b.val[1] + eps ) **return** **false**;
80. **return** **true**;
81. }
82. };
84. **double** triangleArea ( myVec a, myVec b, myVec c ) {
85. **double** area = a.val[0] \* b.val[1] + b.val[0] \* c.val[1] + c.val[0] \* a.val[1];
86. area -= b.val[0] \* a.val[1] + c.val[0] \* b.val[1] + a.val[0] \* c.val[1];
87. area /= 2;
88. **return** area;
89. }
91. **struct** myLine {
92. myVec a, b; //a is displacement, b is direction.
93. //Builds a line from two points
94. myLine lineFromPoints ( myVec x, myVec y ) {
95. myLine m;m.a = x; m.b = y.sub ( x );
96. **return** m;
97. }
98. //Finds point on line, given t.
99. myVec atPos ( **double** t ) {
100. **return** a.add ( b.mul ( t ) ); // a + tb;
101. }
102. **double** lineToPointDistance ( myVec p, **double** &t ) {
103. p = p.sub ( a ); //Take it to origin
104. t = b.dot ( p ) / ( b.length() \* b.length() ); //point of intersection
105. myVec x = b.mul ( t ); //tb
106. **return** ( p.sub(x).length() ); //xp length()
107. }
108. **double** segmentToPointDistance ( myVec p, **double** &t ) {
109. p = p.sub ( a ); //Take it to origin
110. t = b.dot ( p ) / ( b.length() \* b.length() );
111. **if** ( t + eps < 0 || t > 1 + eps ) { //Not on segment
112. **return** min ( p.length(), p.sub(b).length() );
113. }
114. myVec x = b.mul ( t ); //tb
115. **return** ( p.sub(x).length() ); //xp length()
116. }
117. **bool** overlapParallel ( myLine l ) {
118. **double** p, q, r, s;
119. **if** ( b.val[0] == 0 ) {
120. p = a.val[1]; q = atPos(1).val[1]; r = l.a.val[1]; s = l.atPos ( 1 ).val[1];
121. **if** ( min ( r, s ) > max ( p, q ) ) **return** **false**;
122. **if** ( max ( r, s ) < min ( p, q ) ) **return** **false**;
123. **return** **true**;
124. }
125. **else** {
126. p = a.val[0]; q = atPos(1).val[0]; r = l.a.val[0]; s = l.atPos ( 1 ).val[0];
127. **if** ( min ( r, s ) > max ( p, q ) ) **return** **false**;
128. **if** ( max ( r, s ) < min ( p, q ) ) **return** **false**;
129. **return** **true**;
130. }
131. }
132. **char** lineAndLineIntersection2D ( myLine l, **double** &t, **double** &s ) {
133. **if** ( b.cross2D ( l.b) == 0 ) {
134. **if** ( l.a.sub(a).cross2D(l.b) == 0 ) {
135. **if** ( overlapParallel ( l ) ) **return** 'o'; //overlaps
136. **else** **return** 'p'; //parallel
137. }
138. **else** **return** 'd'; //disjoint and parallel
139. }
140. myVec w = a.sub ( l.a );
141. myVec p = l.b.perp2D(), z = b.perp2D();
142. t = -(w.dot(p))/p.dot(b); //for current line
143. s = w.dot(z)/z.dot(l.b); //for line l
144. **return** 'i';
145. }
146. **double** lineAndLineDistance2D ( myLine l ) {
147. **double** t, s; //First check if the intersect
148. **char** r = lineAndLineIntersection2D ( l, t, s );
149. **if** ( r == 'i' ) **return** 0; //Intersects. 0 distance.
150. //Parallel Lines
151. **return** lineToPointDistance ( l.a, t );
152. }
153. **double** lineAndSegmentDistance2D ( myLine l ) {
154. **double** t, s;
155. **char** r = lineAndLineIntersection2D ( l, t, s );
156. **if** ( r == 'i' && s + eps > 0 && s < 1 + eps ) {
157. **return** 0; //Valid intersection
158. }
159. **double** res = lineToPointDistance ( l.a, t );
160. res = min ( res, lineToPointDistance ( l.a.add(l.b), t ) );
161. **return** res;
162. }
163. **double** segmentAndSegmentDistance2D ( myLine l ) {
164. **double** t, s;
165. **char** r = lineAndLineIntersection2D ( l, t, s );
166. **if** ( r =='i' && t+eps > 0 && t < 1 + eps && s + eps > 0 && s < 1 + eps ) {
167. **return** 0; //Valid intersection
168. }
169. **double** res = segmentToPointDistance ( l.a, t );
170. res = min ( res, segmentToPointDistance ( l.a.add(l.b), t ) );
171. res = min ( res, l.segmentToPointDistance ( a, t ) );
172. res = min ( res, l.segmentToPointDistance ( a.add ( b ), t ) );
173. **return** res;
174. }
175. myLine reflect ( myVec p, myVec norm ) {
176. myVec ap = p.sub ( a ); //Starting to Point of Reflection
177. norm = norm.unitVec();
179. **double** d = fabs ( ap.dot ( norm ) );
181. myVec m = p.add ( norm.mul ( d ) );
182. myVec h = m.sub ( a ).mul ( 2 );
183. m = a.add ( h );
185. myLine ray = ray.lineFromPoints ( p, m );
186. **return** ray;
187. }
188. };
190. **struct** myCir {
191. myVec a;
192. **double** r;
193. myVec atPos ( **double** t ) {
194. myVec res;
195. res.val[0] = a.val[0] + r \* cos ( t );
196. res.val[1] = a.val[1] + r \* sin ( t );
197. **return** res;
198. }
199. **char** circleAndLineIntersection2D ( myLine l, **double** &t1, **double** &t2 ) {
200. **double** t3;
201. **double** d = l.lineToPointDistance ( a, t3 );
202. **if** ( d > r + eps ) **return** 'd';
203. **if** ( fabs ( d - r ) <= eps ) **return** 't';
204. ///Pass through Center
205. **if** ( fabs ( d ) <= eps ) {
206. t1 = l.b.polarAngle2D();
207. t2 = t1 + pi;
208. **return** 'i';
209. }
211. myVec m = l.atPos ( t3 );
212. myVec am = m.sub ( a );

215. **double** x = am.polarAngle2D();
216. **double** temp = d / r; **if** ( temp > 1 ) temp = 1; **if** ( temp < -1 ) temp = -1;
217. **double** theta = pi / 2 - asin ( temp ); //Using sin law find internal angle.
219. t1 = x + theta;
220. t2 = x - theta;
221. **return** 'i';
222. }
223. **char** sphereAndLineIntersect ( myLine l, **double** &t1, **double** &t2 ) {
224. **double** tp = 0;
225. **double** d = l.lineToPointDistance ( a, tp );
226. **if** ( d > r + eps ) **return** 'd';
227. **if** ( fabs ( d - r ) < eps ) {
228. t1 = tp;
229. **return** 't';
230. }
231. **double** chord = sqrt ( r \* r - d \* d );
232. t1 = tp - chord / l.b.length();
233. t2 = tp + chord / l.b.length();
234. **return** 'i';
235. }
236. **char** circleAndCircleIntersection2D ( myCir c2, **double** &t1, **double** &t2 ) {
237. myVec d = c2.a.sub ( a );
238. **if** ( d.length() > r + c2.r + eps ) **return** 'd'; //Case 1
239. **if** ( d.length() + c2.r + eps < r ) **return** 'd'; //Case 2
240. **if** ( a == c2.a && fabs ( r - c2.r ) <= eps ) {
241. **if** ( r == 0 ) {
242. t1 = 0;
243. **return** 't'; //Case 7
244. }
245. **return** 's'; //Case 6
246. }
247. **if** ( fabs ( d.length() - r - c2.r ) <= eps ||
248. fabs ( d.length() + c2.r - r ) <= eps ) {
249. t1 = d.polarAngle2D();
250. **return** 't'; //Case 3 and 4
251. }
252. **double** theta = cosineRule3Side ( r, d.length(), c2.r );
253. **double** m = d.polarAngle2D ();
254. t1 = m - theta;
255. t2 = m + theta;
256. **return** 'i'; //Case 5
257. }
258. **int** circleToCircleTangentLine (myCir c2,myLine &l1,myLine &l2,myLine &l3,myLine &l4){
259. //First circle must be smaller or equal to second circle
260. **if** (r>c2.r + eps ) **return** c2.circleToCircleTangentLine ( \***this**, l1, l2, l3, l4 );
262. myVec oo = c2.a.sub ( a );
263. **double** d = oo.length();
265. **if** ( fabs ( d ) < eps && fabs ( r - c2.r ) < eps ) //Infinite tangents
266. **return** -1;
267. **if** ( d + r + eps < c2.r ) //No tangents
268. **return** 0;
270. **double** base = oo.polarAngle2D();
272. **if** ( fabs ( d + r - c2.r ) < eps ) { //Contains Circle
273. l1 = l1.lineFromPoints ( atPos ( base + pi ), atPos ( base + pi ) );
274. **return** 1;
275. }
277. **double** ang = pi - acos ( (c2.r - r ) / d );
278. l1 = l1.lineFromPoints ( atPos ( base + ang ), c2.atPos ( base + ang ) );
279. l2 = l2.lineFromPoints ( atPos ( base - ang ), c2.atPos ( base - ang ) );
281. **if** ( d + eps < r + c2.r ) **return** 2; //Circle intersects
283. **if** ( fabs ( d - r - c2.r ) < eps ) { //Circle tangent
284. l3 = l3.lineFromPoints ( atPos ( base ), atPos ( base ) );
285. **return** 3;
286. }
288. //Disjoint Circle
289. ang = acos ( ( c2.r + r ) / d );
290. l3 = l3.lineFromPoints ( atPos ( base + ang ), c2.atPos ( base + ang + pi ) );
291. l4 = l4.lineFromPoints ( atPos ( base - ang ), c2.atPos ( base - ang + pi ) );
293. **return** 4;
294. }
296. //Used to find intersection area between circle and polygon
297. **double** circleToLineArea ( myLine l ) {
298. **if** ( l.a == l.atPos(1) ) **return** 0;
299. **double** t;
300. **double** d = l.segmentToPointDistance ( a, t );
302. //Segment Outside, no intersection
303. **if** ( d + eps > r ) {
304. **double** dif = l.a.sub(a).angleBetween(l.atPos(1).sub(a));
305. **double** area = dif/2 \* r \* r;
306. **if** ( triangleArea (a,l.a,l.atPos(1) ) + eps < 0 ) area \*= -1;
307. **return** area;
308. }
310. //Segment inside full
311. **double** d1 = l.a.sub(a).length();
312. **double** d2 = l.atPos(1).sub(a).length();
313. **if** ( d1 < r + eps && d2 < r + eps ) {
314. **double** area = triangleArea ( a, l.a, l.atPos(1) );
315. **return** area;
316. }
318. //seg points outside but intersects print
319. **if** ( d1 + eps > r && d2 + eps > r ) {
320. **double** t1, t2;
321. circleAndLineIntersection2D ( l, t1, t2 );
322. **if** ( l.a.sub(atPos(t2)).length() + eps < l.a.sub(atPos(t1)).length() ) {
323. swap ( t1, t2 );
324. }
325. **double** area = 0;
326. area += circleToLineArea ( l.lineFromPoints(l.a,atPos(t1)) );
327. area += circleToLineArea ( l.lineFromPoints(atPos(t1),atPos(t2)) );
328. area += circleToLineArea ( l.lineFromPoints(atPos(t2),l.atPos(1) ) );
329. **return** area;
330. }
332. //Half in, half out
333. **double** t1, t2;
334. circleAndLineIntersection2D ( l, t1, t2 );
335. **double** it;
336. **if** ( atPos(t1).between( l.a, l.atPos(1) ) || atPos(t1).between(l.atPos(1), l.a) ) it = t1;
337. **else** it = t2;
338. **double** area = 0;
339. area += circleToLineArea ( l.lineFromPoints ( l.a, atPos(it) ) );
340. area += circleToLineArea ( l.lineFromPoints ( atPos(it), l.atPos(1) ) );
341. **return** area;
342. }
344. };
346. **bool** collinear ( myVec a, myVec b, myVec c ) {
347. myVec ab = b.sub(a), ac = c.sub(a);
348. **double** d = fabs ( ab.dot(ac) );
349. **if** ( fabs ( d - ab.length() \* ac.length() ) <= eps ) **return** **true**;
350. **return** **false**;
351. }
353. //Find if C is between A and B or B and A
354. **bool** pointBetween ( pii a, pii b, pii c ) {
355. **if** ( MIN(a.ff,b.ff) <= c.ff && c.ff <= MAX(a.ff,b.ff) && MIN(a.ss,b.ss) <= c.ss && c.ss <= MAX(a.ss,b.ss) ) **return** **true**;
356. **else** **return** **false**;
357. }
359. //Determine if (a,b) and (c,d) segment intersects. All points are integer
360. **bool** segmentIntersection ( pii a, pii b, pii c, pii d ) {
362. vlong s1 = triArea( a, b, c );
363. vlong s2 = triArea( a, b, d );
364. vlong s3 = triArea( c, d, a );
365. vlong s4 = triArea( c, d, b );
367. **if** ( s1 < 0 ) s1 = -1; **else** **if** ( s1 > 0 ) s1 = 1;
368. **if** ( s2 < 0 ) s2 = -1; **else** **if** ( s2 > 0 ) s2 = 1;
369. **if** ( s3 < 0 ) s3 = -1; **else** **if** ( s3 > 0 ) s3 = 1;
370. **if** ( s4 < 0 ) s4 = -1; **else** **if** ( s4 > 0 ) s4 = 1;
372. **if** ( s1 \* s2 > 0 || s3 \* s4 > 0 ) **return** **false**;
374. **if** ( s1 && s2 && s3 && s4 ) **return** **true**;
375. **if** ( s1 == 0 && pointBetween ( a, b, c ) ) **return** **true**;
376. **else** **if** ( s2 == 0 && pointBetween ( a, b, d ) ) **return** **true**;
377. **else** **if** ( s3 == 0 && pointBetween ( c, d, a ) ) **return** **true**;
378. **else** **if** ( s4 == 0 && pointBetween ( c, d, b ) ) **return** **true**;
380. **return** **false**;
381. }

384. /\*
385. Shooting Ray
387. For any point (x,y), shoot a ray towards ( x + 10^9+7, y + 10^9 + 9 ), any large coprime numbers so that no integer points fall on the ray.
389. Next find the intersection of all edges of polygon with the ray. If intersection is odd, then inside.
391. But ray shooting cannot handle points on boundary. So manually check if the give point is on boundary of an edge.
393. if ( onboundary ) handle it
394. else if ( intersect is odd ) inside
395. else outside
396. \*/

**KMP:**

1. //WARNING: Arrays are 1-based index.
3. #define LENGTH XYZ
4. **char** text[LENGTH], pat[LENGTH];
5. **int** pre[LENGTH];
7. **void** compute () {
8. **int** plen = strlen ( pat + 1 ), k = 0;
9. pre[1] = 0;
10. **for** ( **int** i = 2; i <= plen; i++ ) {
11. **while** ( k && pat[k+1] != pat[i] ) k = pre[k];
12. **if** ( pat[k+1] == pat[i] ) k++;
13. pre[i] = k;
14. }
15. }
16. **int** match () {
17. **int** tlen = strlen ( text + 1 ), plen = strlen ( pat + 1 );
18. **int** q = 0;
19. **int** i, res = 0;
20. **for** ( i = 1; i <= tlen; i++ ) {
21. **while** ( q && pat[q+1] != text[i] ) q = pre[q];
23. **if** ( pat[q+1] == text[i] ) q++;
24. **if** ( q == plen ) {
25. res++;
26. q = pre[q];
27. }
28. }
29. **return** res;
30. }
32. /// Application
33. /// Period of a string. N - pre[N] is the length of period iff period length divides N.

**Linear Diphantine Equation:**

1. //Returns char according to kind of solution
2. //Ax + By = C Run+ Rise-
3. //AX + (-B)y = C Run+ Rise+
4. //When changing X, steps are either d/run or (d+run-1)/run depending on crossing or not.
5. //When changing Y, steps are same as X, but take ABS(Rise) and assign sign to step.
6. //Always assign sign to steps and ADD values to x and y.

9. **struct** LinearDiphontine {
10. vlong a, b, c, x, y, run, rise;
11. **char** solution;
13. **char** solve () {
14. **if** ( a == 0 && b == 0 ) {
15. **if** ( c == 0 ) {
16. //Infinite solutions. Anything works
17. **return** solution = 'i';
18. }
19. **else** **return** solution = '0'; //No solution
20. }
21. **if** ( a == 0 || b == 0 ) {
22. //Vertical or Horizontal Line
23. **if** ( !a ) {
24. **if** ( c % b != 0 ) **return** solution = '0'; /// No Solution
25. run = 1; rise = 0;
26. **return** solution = 'h'; /// ( anything, c / b )
27. }
28. **else** {
29. **if** ( c % a != 0 ) **return** solution = '0'; ///No Solution
30. run = 0; rise = 1;
31. **return** solution = 'v'; /// ( c / a , anything )
32. }
33. }
35. vlong g = ext\_gcd( a, b, &x, &y );
36. **if** ( c % g != 0 ) {
37. //No solution
38. **return** solution = '0';
39. }
41. a /= g; b /= g; c /= g;
43. ext\_gcd ( a, b, &x, &y );
44. x \*= c; y \*= c;
46. ///run and rise calculation
47. run = b; rise = -a;
48. **if** ( run < 0 ) {
49. run \*= -1; rise \*= -1;
50. }
52. **return** solution = '1';
53. }
55. ///Move solution near to vertical line x = p
56. **void** moveNearVerticalLine( **int** p ) {
57. **if** ( run == 0 ) **return**; /// Both are vertical
58. vlong jump = ( p - x ) / run;
59. x += jump \* run;
60. y += jump \* rise;
62. **if** ( x < p ) { ///Keep solution on right of the line
63. x += run;
64. y += rise;
65. }
66. }
68. **void** moveNearHorizontalLine( **int** p ) {
69. **if** ( rise == 0 ) **return**; /// Both are horizontal
70. vlong jump = ( p - y ) / rise;
71. x += jump \* run;
72. y += jump \* rise;
74. **if** ( y < p ) { ///Keep solution above the line
75. **if** ( rise > 0 ) {
76. x += run;
77. y += rise;
78. }
79. **else** {
80. x -= run;
81. y -= rise;
82. }
83. }
84. }
85. };

**Manachar Algorithm:**

1. #define MANALEN 200100
2. /// Warning: Make MANALEN double of string length
4. /\*
5. 1. First assign string to manacher, arr[]
6. 2. Call init
7. 3. Call Preprocess to convert string
8. 4. Call calc to calculate array p[]
9. \*/
11. **struct** MANACHER {
12. **char** arr[MANALEN+10], brr[MANALEN+10];
13. **int** p[MANALEN+10];
14. **int** an, bn, mxpalin, mxcenter;
16. **void** init() {
17. //Assign string to arr
18. an = strlen ( arr );
19. }
21. //aba -> ^#a#b#a#$
22. **void** preprocess() {
23. **int** cur = 0;
24. brr[cur++] = '^';
25. FOR(i,0,an-1) {
26. brr[cur++] = '#';
27. brr[cur++] = arr[i];
28. }
29. brr[cur++] = '#';
30. brr[cur++] = '$';
31. brr[cur] = 0;
32. bn = cur;
33. }
35. ///For each possible center, length of palindrome it can create.
36. ///Careful, this length is without #
37. **void** calc() {
38. **int** c = 0, r = 0;
39. FOR(i,1,bn-2) {
40. **int** mi = c - (i-c);
42. p[i] = (r > i )? MIN ( r - i, p[mi] ): 0;
44. **while** ( brr[i+p[i]+1] == brr[i-p[i]-1] ) {
45. p[i]++;
46. }
48. **if** ( i + p[i] > r ) {
49. r = i + p[i];
50. c = i;
51. }
52. }
54. mxpalin = 0; mxcenter = 0;
55. ///Notice we don't multiply 2 with p[i], cause more than half of them are #
56. FOR(i,1,bn-2) {
57. **if** ( p[i] > mxpalin ) {
58. mxpalin = p[i];
59. mxcenter = i;
60. }
61. }
62. ///In brr[], for each center, the palindrome extends from i-p[i] to i+p[i]
63. }
64. }mana;

**Miller Robin Primality Test:**

1. vlong mulmod ( vlong a, vlong b, vlong c ) {
2. **if** ( a < b ) swap ( a, b );
3. vlong res = 0, x = a;
4. **while** ( b > 0 ) {
5. **if** ( b & 1 ) {
6. res = res + x;
7. **if** ( res >= c ) res -= c;
8. }
9. x = x \* 2;
10. **if** ( x >= c ) x -= c;
11. b >>= 1;
12. }
13. **return** res % c;
14. }
16. /\*\* Smallest odd number for which Miller-Rabin primality test on bases <= n-th prime does not reveal compositeness.
17. 2047, 1373653, 25326001, 3215031751, 2152302898747, 3474749660383, 341550071728321, 341550071728321,
18. 3825123056546413051, 3825123056546413051, 3825123056546413051 \*\*/
20. vlong prime[] = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41 };
21. **int** psize = 13;
23. vlong bigmod ( vlong a, vlong p, vlong mod ) {
24. vlong x = a, res = 1;
25. **while** ( p ) {
26. **if** ( p & 1 ) res = mulmod( res, x, mod );
27. x = mulmod( x, x, mod );
28. p >>= 1;
29. }
30. **return** res;
31. }
33. ///Witness to compositeness of n
34. ///n - 1 = 2^s \* d, where d is odd
35. **bool** witness ( vlong a, vlong d, vlong s, vlong n ) {
36. vlong r = bigmod( a, d, n );
37. **if** ( r == 1 || r == n - 1 ) **return** **false**;
38. **int** i;
39. **for** ( i = 0; i < s - 1; i++ ) {
40. r = mulmod( r, r, n );
41. **if** ( r == 1 ) **return** **true**;
42. **if** ( r == n - 1 ) **return** **false**;
43. }
44. **return** **true**;
45. }
47. **bool** Miller\_Rabin ( vlong n ) {
48. **if** ( n <= 1 ) **return** **false**;
50. **int** i;
51. vlong p = n - 1;
52. vlong s = 0;
53. **while** ( ! ( p & 1 ) ) {
54. p /= 2;
55. s++;
56. }
57. vlong d = p;
58. p = n - 1;
60. **for** ( i = 0; i < psize && prime[i] < n; i++ ) {
61. **if** ( witness ( prime[i], d, s, n ) ) **return** **false**;
62. }
63. **return** **true**;
64. }

**Mobius Function:**

1. /\*
2. Mobius function is a function which has one of the following values:
3. mu(x) = -1 : square free number with odd prime factors
4. mu(x) = 1  : square free number with even prime factors
5. mu(x) = 0  : not a square free number
7. In order to generate mu() from 1-N, we need to generate all primes between 1-N
8. \*/
10. **void** sieve( **int** n ); ///generates prime from 1-N
12. **int** mobius[SIZE+10];
13. **void** mobiusCalc( **int** n ) {
14. FOR(i,1,n) mobius[i] = 1; ///Initially all value is 1
16. ///Mark all numbers with square factors 0
17. **int** sqrtn = sqrt ( n );
18. FOR(i,0,SZ(prime)-1) {
19. **if** ( prime[i] > sqrtn ) **break**;
21. **int** x = prime[i] \* prime[i];
22. **for** ( **int** j = x; j <= n; j += x ) mobius[j] = 0;
23. }
25. ///For each prime, all it's multiple gets multiplied by -1
26. FOR(i,0,SZ(prime)-1) {
27. **for** ( **int** j = prime[i]; j <= n; j += prime[i] ) mobius[j] \*= -1;
28. }
29. }
31. /\*Application of Mobius function
32. It is mainly used with inclusion exclusion, when items are the first K primes.
33. Complexity is improved from 2^k to NloglogN
34. \*/

**Pascal Triangle:**

1. #define BIOSIZE 0
3. **struct** BIONOMIAL {
4. vlong n, k, val;
6. **void** goDown() {
7. vlong a = n + 1, b = n - k + 1; vlong g = gcd ( a, b ); a /= g; b /= g;
8. val /= b; val \*= a;
9. n++;
10. }
11. **void** goRight() {
12. **if** ( k + 1 > n ) {
13. k = n + 1;
14. val = 0;
15. **return**;
16. }
17. vlong a = n - k, b = k + 1; vlong g = gcd ( a, b ); a /= g; b /= g;
18. val /= b; val \*= a;
19. k++;
20. }
22. **void** assign ( vlong \_n, vlong \_k, vlong \_val ) {
23. n = \_n; k = \_k; val = \_val;
24. }
26. **void** clear () {
27. //CLR(memo,-1);
28. }
30. //Calculate NCK
31. **int** memo[BIOSIZE][BIOSIZE];
32. vlong nck ( **int** n, **int** k ) {
33. **if** ( memo[n][k] != -1 ) **return** memo[n][k];
34. **if** ( k == 0 ) **return** 1;
35. **if** ( n == 0 ) **return** 0;
37. memo[n][k] = nck ( n - 1, k ) + nck ( n - 1, k - 1 );
38. **return** memo[n][k];
39. }
40. };

**Suffix Array:**

1. /\*
2. Suffix array implementation using bucket sorting + lcp.
3. Complexity O(n log n), str[] is the target string,
4. nnn is its length and suffix[i] contains i'th sorted suffix position.
5. \*/
7. /\*
8. 1. Fill str[] with string or augmented string.
9. 2. Assign nnn length of str[].
10. 3. Call SortSuffix()
11. 4. Result is at two different arrays: suffix[i] contains index of i'th sorted string
12. and sufRank[stp][i] contains rank of i'th suffix
13. \*/
15. #define MAXN Maximum Length of str[]
16. #define MAXL Maximum Depth of log(MaxN)
18. **int** nnn, stp, mv, suffix[MAXN], tmp[MAXN];
19. **int** sum[MAXN], cnt[MAXN], sufRank[MAXL][MAXN];
20. **char** str[MAXN];
22. **inline** **bool** equal(**const** **int** &u, **const** **int** &v){
23. **if**(!stp) **return** str[u] == str[v];
24. **if**(sufRank[stp-1][u] != sufRank[stp-1][v]) **return** **false**;
25. **int** a = u + mv < nnn ? sufRank[stp-1][u+mv] : -1;
26. **int** b = v + mv < nnn ? sufRank[stp-1][v+mv] : -1;
27. **return** a == b;
28. }
30. **void** update(){
31. **int** i, rnk;
32. **for**(i = 0; i < nnn; i++) sum[i] = 0;
33. **for**(i = rnk = 0; i < nnn; i++) {
34. suffix[i] = tmp[i];
35. **if**(i && !equal(suffix[i], suffix[i-1])) {
36. sufRank[stp][suffix[i]] = ++rnk;
37. sum[rnk+1] = sum[rnk];
38. }
39. **else** sufRank[stp][suffix[i]] = rnk;
40. sum[rnk+1]++;
41. }
42. }
44. **void** Sort() {
45. **int** i;
46. **for**(i = 0; i < nnn; i++) cnt[i] = 0;
47. memset(tmp, -1, **sizeof** tmp);
48. **for**(i = 0; i < mv; i++){
49. **int** idx = sufRank[stp - 1][nnn - i - 1];
50. **int** x = sum[idx];
51. tmp[x + cnt[idx]] = nnn - i - 1;
52. cnt[idx]++;
53. }
54. **for**(i = 0; i < nnn; i++){
55. **int** idx = suffix[i] - mv;
56. **if**(idx < 0)**continue**;
57. idx = sufRank[stp-1][idx];
58. **int** x = sum[idx];
59. tmp[x + cnt[idx]] = suffix[i] - mv;
60. cnt[idx]++;
61. }
62. update();
63. **return**;
64. }
66. **inline** **bool** cmp(**const** **int** &a, **const** **int** &b){
67. **if**(str[a]!=str[b]) **return** str[a]<str[b];
68. **return** **false**;
69. }
71. **void** SortSuffix() {
72. **int** i;
73. **for**(i = 0; i < nnn; i++) tmp[i] = i;
74. sort(tmp, tmp + nnn, cmp);
75. stp = 0;
76. update();
77. ++stp;
78. **for**(mv = 1; mv < nnn; mv <<= 1) {
79. Sort();
80. stp++;
81. }
82. stp--;
83. **for**(i = 0; i <= stp; i++) sufRank[i][nnn] = -1;
84. }
86. ///Here u and v are unsorted suffix string positions
87. **inline** **int** lcp(**int** u, **int** v) {
88. **if**(u == v) **return** nnn - u;
89. **int** ret, i;
90. **for**(ret = 0, i = stp; i >= 0; i--) {
91. **if**(sufRank[i][u] == sufRank[i][v]) {
92. ret += 1<<i;
93. u += 1<<i;
94. v += 1<<i;
95. }
96. }
97. **return** ret;
98. }

**Suffix Automata:**

1. #define MAXNODE 5000000 ///Total number of nodes in the tree. Add all length of suffix.
2. #define ALPHABET\_SIZE 27
4. **struct** NODE {
5. **int** start, end, slink; ///Except for root - Each edge has a [start and end) corresponding to text[start,end)
6. **int** child[ALPHABET\_SIZE];
8. ///Always use this to find length of edge
9. **int** edgeLen () {
10. **return** min ( pos + 1, end ) - start;
11. }
12. }t[2\*MAXNODE]; ///\*2 cause each internal node can have terminal node attached. Imagine "aaa...aa"
13. **char** text[MAXNODE];
15. **struct** SuffixTree {
16. **int** root, last\_added, pos, needSL, rem, act\_node, act\_e, act\_len;
17. ///Only need root from here. Others can be private.
19. **int** newNode ( **int** start, **int** end = inf ) {
20. NODE nd;
21. nd.start = start;
22. nd.end = end;
23. nd.slink = 0;
24. **for** ( **int** i = 0; i < ALPHABET\_SIZE; i++ ) {
25. nd.child[i] = 0;
26. }
27. t[++last\_added] = nd;
28. **return** last\_added;
29. }
30. **char** actEdge() {
31. **return** text[act\_e];
32. }
34. **void** addSL ( **int** node ) {
35. **if** ( needSL > 0 ) t[needSL].slink = node;
36. needSL = node;
37. }
39. **bool** walkdown ( **int** node ){
40. **if** ( act\_len >= t[node].edgeLen() ) {
41. act\_e += t[node].edgeLen();
42. act\_len -= t[node].edgeLen();
43. act\_node = node;
44. **return** **true**;
45. }
46. **return** **false**;
47. }
49. **void** st\_init() {
50. needSL = 0; last\_added = 0; pos = -1;
51. rem = 0; act\_node = 0; act\_e = 0; act\_len = 0;
52. root = act\_node = newNode ( -1, -1 );
53. }
55. **void** st\_extend( **int** c ) {
56. text[++pos] = c;
57. needSL = 0;
58. rem++;
60. **while** ( rem ) {
61. **if** ( act\_len == 0 ) act\_e = pos;
62. **if** ( t[act\_node].child[actEdge()] == 0  ) {
63. **int** leaf = newNode ( pos );
64. t[act\_node].child[actEdge()] = leaf;
65. addSL ( act\_node );
66. }
67. **else** {
68. **int** nxt = t[act\_node].child[actEdge()];
69. **if** ( walkdown ( nxt ) ) **continue**;
70. **if** ( text[t[nxt].start+act\_len] == c ) {
71. act\_len++;
72. addSL ( act\_node );
73. **break**;
74. }
75. **int** split = newNode ( t[nxt].start, t[nxt].start + act\_len );
76. t[act\_node].child[actEdge()] = split;
77. **int** leaf = newNode ( pos );
78. t[split].child[c] = leaf;
79. t[nxt].start += act\_len;
80. t[split].child[text[t[nxt].start]] = nxt;
82. addSL ( split );
83. }
84. rem--;
85. **if** ( act\_node == root && act\_len > 0 ) {
86. act\_len--;
87. act\_e = pos - rem + 1;
88. }
89. **else** {
90. **if** ( t[act\_node].slink > 0 ) act\_node = t[act\_node].slink;
91. **else** act\_node = root;
92. }
93. }
94. }
95. };
97. /\*
98. 1. How to insert?
99. Take input in a string, then convert each charater to **int** and insert **using** st\_extend( c ).
100. At the end of the string, extend **using** st\_extend($), where $ is an **int** > than others.
102. 2. How to check **for** terminal node?
103. **if** ( t[s].end == inf ) it is a terminal node.
104. \*/

**Treap:**

1. /\*
2. To insert an item, pitem x = new item ( key, rand() \* rand() )
3. Many functions need a root to be passed: insert ( treap.root, x )
4. \*/
6. **struct** item {
7. **int** key, prior, cnt;
8. item \* l, \* r;
9. item() {}
10. item( **int** key, **int** prior ): key(key), prior(prior), l(NULL), r(NULL) {}
11. };
12. **typedef** item \* pitem;
14. **struct** Treap {
15. **int** size;
16. pitem root;
18. /\*Resets the Treap. Not complete yet. Need to erase all elements from treap\*/
19. **void** clear() {
20. size = 0;
21. root = NULL;
22. }
24. **int** cnt (pitem t) {
25. **return** t ? t->cnt : 0;
26. }
28. **void** upd\_cnt (pitem t) {
29. **if** (t) {
30. t->cnt = 1 + cnt(t->l) + cnt (t->r);
31. }
32. }
34. **void** split ( pitem t, **int** key, pitem &l, pitem &r ) {
35. **if** ( !t ) {
36. l = r = NULL;
37. }
38. **else** **if** ( key < t->key ) {
39. split( t->l, key, l, t->l );
40. r = t;
41. }
42. **else** {
43. split( t->r, key, t->r, r );
44. l = t;
45. }
46. upd\_cnt (t);
47. }
48. **void** insert ( pitem &t, pitem it ) {
49. **if** ( !t ) {
50. t = it;
51. size++;
52. }
53. **else** **if** ( it->prior > t->prior ) {
54. split( t, it->key, it->l, it->r );
55. t = it;
56. size++;
57. }
58. **else** {
59. insert( it->key < t->key? t->l:t->r, it );
60. }
61. upd\_cnt (t);
62. }
64. **void** merge ( pitem &t, pitem l, pitem r ) {
65. **if** ( !l || !r ) {
66. t = l? l:r;
67. }
68. **else** **if** ( l->prior > r->prior ){
69. merge ( l->r, l->r, r );
70. t = l;
71. }
72. **else** {
73. merge ( r->l, l, r->l );
74. t = r;
75. }
76. upd\_cnt( t );
77. }
78. **void** erase ( pitem &t, **int** key ) {
79. **if** ( t->key == key ) {
80. pitem e = t;
81. merge( t, t->l, t->r );
82. **delete** ( e );
83. size--;
84. }
85. **else** {
86. erase( key < t->key? t->l:t->r, key );
87. }
88. upd\_cnt (t);
89. }
91. //Find key in treap
92. pitem find ( pitem t, **int** key ) {
93. **if** ( !t ) **return** t;
94. **if** ( t->key == key ) **return** t;
95. **if** ( key < t->key ) **return** find ( t->l, key );
96. **else** **return** find ( t->r, key );
97. }
99. /\*Finds number of element <= x in the treap\*/
100. **int** KPivot ( pitem t, **int** key ) {
101. **if** ( !t ) **return** 0;
102. **else** **if** ( t->key <= key ) {
103. **return** ( cnt( t ) - cnt( t->r ) ) + KPivot(t->r, key);
104. }
105. **else** {
106. **return** KPivot( t->l, key );
107. }
108. }
110. /\*Finds kth element in sorted order\*/
111. pitem pos ( pitem t, **int** p ) {
112. **if** ( cnt ( t->l ) == p ) **return** t;
113. **if** ( cnt ( t->l ) < p ) **return** pos ( t->r, p  - cnt(t->l) - 1 );
114. **else** **return** pos ( t->l, p );
115. }
116. }treap;

**Convex Hull:**

1. /\*
2. 1. Assign hull.n
3. 2. Take input in hull.point
4. 3. Call findHull()
5. 4. Convex Hull is ready in hull.convex with hull.m points in it.
6. \*/
8. pll g;
10. vlong triArea  ( **const** pll &a, **const** pll &b, **const** pll &c ) {
11. vlong area = a.ff \* b.ss + b.ff \* c.ss + c.ff \* a.ss;
12. area -= a.ff \* c.ss + b.ff \* a.ss + c.ff \* b.ss;
13. **return** area;
14. }
16. vlong sqDist ( **const** pll &a, **const** pll &b ) {
17. **return** ( SQ(a.ff-b.ff) + SQ(a.ss-b.ss ) );
18. }
20. **bool** convexCompare ( **const** pll &a, **const** pll &b ) {
21. vlong area = triArea ( g, a, b );
22. **if** ( area > 0 ) **return** **true**;
23. **else** **if** ( area == 0 && sqDist ( g, a ) < sqDist ( g, b ) ) **return** **true**;
24. **else** **return** **false**;
25. }
27. **class** ConvexHull {
28. **void** convexSort() { ///CCW
29. g = point[0];
30. FOR(i,0,n-1) {
31. **if** ( point[i].ff < g.ff ) g = point[i];
32. **else** **if** ( point[i].ff == g.ff && point[i].ss < g.ss ) g = point[i];
33. }
34. sort ( point, point + n, convexCompare );
35. }
37. **public**:
38. **int** n, m; ///N is number of points in polygon and M is number of points in convex hull
40. #define PPP adsf
41. pll point[PPP], convex[PPP];
42. **void** findHull() {
43. convexSort();
45. **if** ( n == 1 ) {
46. convex[0] = convex[1] = point[0];
47. m = 1;
48. **return**;
49. }
50. convex[0] = point[n-1]; convex[1] = point[0]; convex[2] = point[1];
51. **int** cur = 3;
52. **for** ( **int** i = 2; i < n; i++ ) {
53. vlong area = triArea ( convex[cur-2], convex[cur-1], point[i] );
54. **if** ( area > 0 ) {
55. convex[cur] = point[i];
56. cur++;
57. }
58. **else** **if** ( area == 0 ) { ///Take action depending on what is required
59. /\*Left Vertical Line gets omitted. Manually handle it\*/
60. /\*convex[cur] = point[i];
61. cur++;\*/
62. ///If extra point needs to be removed
63. convex[cur-1] = point[i];
64. }
65. **else** {
66. cur--;
67. i--;
68. }
69. }
70. m = cur - 1;
71. }
73. }hull;