

File: /mnt/Work/notes/Geometry.cpp

```
1 // The trig functions of C++ take radian exclusively
2 // 0D Objects-----
3 struct point { // In Integer
4     int x, y;
5     point() { x = y = 0; }
6     point(int _x, int _y) : x(_x), y(_y) {}
7
8     bool operator < (point other) const {
9         if(x != other.x)
10             return x < other.x;
11         return y < other.y;
12     }
13
14     bool operator == (point other) const {
15         return (x == other.x) && (y == other.y);
16     };
17
18 struct point { // In Double
19     double x, y;
20     point() { x = y = 0.0; }
21     point(double _x, double _y) : x(_x), y(_y) {}
22
23     bool operator < (point other) const {
24         if (fabs(x - other.x) > EPS)
25             return x < other.x;
26         return y < other.y;
27     }
28     bool operator == (point other) const {
29         return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS));
30     };
31
32 bool Equal(double a, double b) {
33     return (fabs(a - b) <= EPS);
34 }
35
36 int hypot(point p1, point p2) {
37     int x = p1.x - p2.x;
38     int y = p1.y - p2.y;
39     return x*x + y*y;
40 }
41
42 double dist(point p1, point p2) {
43     int x = p1.x - p2.x;
44     int y = p1.y - p2.y;
45     return sqrt(x*x + y*y);
46 }
47
48 double DEG_to_RAD(double deg) { // Converts Degree to Radian
49     return (deg * PI) / 180;
50 }
51
52 double RAD_to_DEG(double rad) {
53     return (180 / PI) * rad;
54 }
55
56 point rotate(point p, double theta) { // Rotates point p w.r.t. origin. (theta is in degree)
57     double rad = DEG_to_RAD(theta);
58     return point(p.x * cos(rad) - p.y * sin(rad), p.x * sin(rad) + p.y * cos(rad));
59 }
60
61 double PointToArea(point p1, point p2, point p3) { // Returns Positive Area in if the points are clockwise, Negative for Anti-Clockwise
62     return (p1.x*(p2.y - p3.y) + p2.x*(p3.y - p1.y) + p3.x*(p1.y - p2.y)); // Divide by 2 if Triangle area is needed
63 }
64
65 double whichSide(point p, point q, point r) { // returns on which side point r is w.r.t pq line
66     double slope = (p.y - q.y)*(q.x - r.x) - (q.y - r.y)*(p.x - q.x);
67     return slope; // slope = 0 : linear, slope > 0 : right, slope < 0 : left
68 }
69
70 // 1D Objects-----
71 struct line {
72     double a, b, c;
73 };
74 void pointsToLine(point p1, point p2, line &l) { // ax + by + c = 0 [comes from y = mx + c]
75     if (fabs(p1.x - p2.x) < EPS) // vertical line is fine
76         l.a = 1.0, l.b = 0.0, l.c = -p1.x; // default values
77     else {
78         l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
79         l.b = 1.0; // IMPORTANT: we fix the value of b to 1.0
80         l.c = -(double)(l.a * p1.x) - p1.y;
81     }
82 bool areParallel(line l1, line l2) { // check coefficients a & b
83     return (fabs(l1.a - l2.a) < EPS) && (fabs(l1.b - l2.b) < EPS);
84 }
85 bool areSame(line l1, line l2) { // also check coefficient c
86     return (areParallel(l1, l2) && fabs(l1.c - l2.c) < EPS);
87 }
```

```

86     return areParallel(l1, l2) && (fabs(l1.c - l2.c) < EPS);
87 }
88 bool areIntersect(line l1, line l2, point &p) {
89     if(areParallel(l1, l2)) return 0; // no intersection
90     p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b); // solve system of 2 linear algebraic equations with 2 unknowns
91     if(fabs(l1.b) > EPS) p.y = -(l1.a * p.x + l1.c); // special case: test for vertical line to avoid division by zero
92     else p.y = -(l2.a * p.x + l2.c);
93     return 1;
94 }
95 line perpendicularLine(line l, point p) { // returns a perpendicular line on l which goes through
96     line ret; // point p
97     ret.a = l.b, ret.b = -l.a;
98     ret.c = -(ret.a * p.x + ret.b * p.y);
99     if(ret.b < 0) ret.a *= -1, ret.b *= -1, ret.c *= -1; // as line must contain b = 1.0 by default
100    if(ret.b != 0) {
101        ret.a /= ret.b;
102        ret.c /= ret.b;
103        ret.b = 1;
104    }
105    return ret;
106 }
107
108 // Vectors -----
109 struct vec {
110     double x, y; // name: 'vec' is different from STL vector
111     vec(double _x, double _y) : x(_x), y(_y) {}
112 };
113 vec toVec(point a, point b) { // convert 2 points to vector a->b
114     return vec(b.x - a.x, b.y - a.y);
115 }
116 vec scale(vec v, double s) { // nonnegative s = [<1 .. 1 .. >1]
117     return vec(v.x * s, v.y * s); // shorter.same.longer
118 }
119 point translate(point p, vec v) { // translate p according to v
120     return point(p.x + v.x, p.y + v.y);
121 }
122 double dot(vec a, vec b) {
123     return (a.x * b.x + a.y * b.y);
124 }
125 double norm_sq(vec v) {
126     return v.x * v.x + v.y * v.y;
127 }
128
129 // Parametric Line -----
130 struct ParaLine { // Line in Parametric Form
131     point a, b; // points must be in DOUBLE
132     ParaLine() { a.x = a.y = b.x = b.y = 0; }
133     ParaLine(point _a, point _b) : a(_a), b(_b) {} // {Start, Finish} or {from, to}
134 };
135 point getPoint(double t) { // Parametric Line : a + t * (b - a) t = [-inf, +inf]
136     return point(a.x + t*(b.x - a.x), a.y + t*(b.y - a.y));
137 };
138
139 // Returns the distance from p to the line defined by two points a and b (a and b must be different)
140 // the closest point is stored in the 4th parameter (byref)
141 double distToLine(point p, point a, point b, point &c) { // formula: c = a + u * ab
142     vec ap = toVec(a, p), ab = toVec(a, b);
143     double u = dot(ap, ab) / norm_sq(ab);
144     c = translate(a, scale(ab, u)); // translate a to c
145     return dist(p, c); // Euclidean distance between p and c
146 }
147
148 // Returns the distance from p to the line segment ab defined by two points a and b (still OK if a == b)
149 // the closest point is stored in the 4th parameter (byref)
150 double distToLineSegment(point p, point a, point b, point &c) {
151     vec ap = toVec(a, p), ab = toVec(a, b);
152     double u = dot(ap, ab) / norm_sq(ab);
153     if(u < 0.0) {
154         c = point(a.x, a.y); // closer to a
155         return dist(p, a); // Euclidean distance between p and a
156     }
157     if(u > 1.0) {
158         c = point(b.x, b.y); // closer to b
159         return dist(p, b); // Euclidean distance between p and b
160     }
161     return distToLine(p, a, b, c); // run distToLine as above
162 }
163
164 // Returns the angle aob given three points: a, o, and b, (using dot product)
165 double angle(point a, point o, point b) { // returns angle aob in rad
166     vec oa = toVec(o, a), ob = toVec(o, b);
167     return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob)));
168 }
169
170 double cross(vec a, vec b) { // Cross product of two vectors
171     return a.x * b.y - a.y * b.x; // note: to accept collinear points, we have to change the '> 0'
172 }
173
174 bool ccw(point p, point a, point b) { // returns true if point p is on the left side of line ab

```

```

174 bool ccw(point p, point q, point r) { // returns true if point r is on the left side of line pq
175     return cross(toVec(p, q), toVec(p, r)) > 0;
176 }
177
178 bool collinear(point p, point q, point r) { // returns true if point r is on the same line as the line pq
179     return fabs(cross(toVec(p, q), toVec(p, r))) < EPS;
180 }
181
182 // 2D Objects -----
183 // ----- CIRCLE -----
184 struct circle {
185     int x, y, r;
186     circle(int _x, int _y, int _r) {
187         x = _x;
188         y = _y;
189         r = _r;
190     }
191     double Area() {
192         return PI*r*r;
193     }
194 };
195
196 // Reference: https://www.mathsisfun.com/geometry/circle-sector-segment.html
197 double CircleSegmentArea(double r, double theta) { // Circle Radius, Center Angle(degree)
198     return r*r*0.5*(DEG_to_RAD(theta) - sin(DEG_to_RAD(theta)));
199 }
200 double CircleSectorArea(double r, double theta) { // Circle Radius, Center Angle(degree)
201     return r*r*0.5*DEG_to_RAD(theta);
202 }
203 double CircleArcLength(double r, double theta) { // Circle Radius, Center Angle(degree)
204     return r*DEG_to_RAD(theta);
205 }
206 bool doIntersectCircle(circle c1, circle c2) {
207     int dis = dist(point(c1.x, c1.y), point(c2.x, c2.y));
208     if(sqrt(dis) < c1.r+c2.r) return 1;
209     return 0;
210 }
211 bool isInside(circle c1, circle c2) { // Returns true if any one of the circle is fully into another circle
212     int dis = dist(point(c1.x, c1.y), point(c2.x, c2.y));
213     return ((sqrt(dis) <= max(c1.r, c2.r)) and (sqrt(dis) + min(c1.r, c2.r) < max(c1.r, c2.r)));
214 }
215 // Returns where a point p lies according to a circle of center c and radius r
216 int insideCircle(point p, point c, int r) { // all integer version
217     int dx = p.x - c.x, dy = p.y - c.y;
218     int Euc = dx*dx + dy*dy, rSq = r*r; // all integer
219     return Euc < rSq ? 0 : Euc == rSq ? 1 : 2; // inside(0)/border(1)/outside(2)
220 }
221
222 // Given 2 points on the circle (p1 and p2) and radius r of the corresponding circle,
223 // determine the location of the centers (c1 and c2) of the two possible circles
224 bool circle2PtsRad(point p1, point p2, double r, point &c) {
225     double d2 = (p1.x - p2.x)*(p1.x - p2.x) + (p1.y - p2.y)*(p1.y - p2.y);
226     double det = r*r / d2 - 0.25;
227     if(det < 0.0) return false;
228     double h = sqrt(det);
229     c.x = (p1.x + p2.x)*0.5 + (p1.y - p2.y)*h;
230     c.y = (p1.y + p2.y)*0.5 + (p2.x - p1.x)*h;
231     return true; // to get the other center, reverse p1 and p2
232 }
233
234 // ----- Triangle -----
235 double TriangleArea(double AB, double BC, double CA) {
236     double s = (AB + BC + CA)/2.0;
237     return sqrt(s*(s-AB)*(s-BC)*(s-CA));
238 }
239 double getAngle(double AB, double BC, double CA) { // Returns the angle(IN RADIAN) oppposite of side CA
240     return acos((AB*AB + BC*BC - CA*CA)/(2*AB*BC));
241 }
242 double rInCircle(double ab, double bc, double ca) { // Returns radius of inCircle of a triangle
243     return TriangleArea(ab, bc, ca) / (0.5*(ab + bc + ca));
244 }
245 int inCircle(point p1, point p2, point p3, point &ctr, double &r) {
246     r = rInCircle(p1, p2, p3);
247     if(fabs(r) < EPS) return 0; // no inCircle center
248     line l1, l2;
249     double ratio = dist(p1, p2) / dist(p1, p3); // compute these two angle bisectors
250     point p = translate(p2, scale(toVec(p2, p3), ratio / (1 + ratio)));
251     pointsToLine(p1, p, l1);
252     ratio = dist(p2, p1) / dist(p2, p3);
253     p = translate(p1, scale(toVec(p1, p3), ratio / (1 + ratio)));
254     pointsToLine(p2, p, l2);
255     areIntersect(l1, l2, ctr);
256     return 1;
257 }
258 // radius of Circle Outside of a Triangle
259 double rCircumCircle(double ab, double bc, double ca) { // ab, ac, ad are sides of triangle
260     return ab*bc*ca / (4.0*TriangleArea(ab, bc, ca));
261 }
262 point CircumCircleCenter(point a, point b, point c, double &r) { // returns center and radius of circumcircle

```

```

262 point CircumferenceCenter(point a, point b, point c, double a1) { // returns center and radius of circumference
263     double ab = dist(a, b);
264     double bc = dist(b, c);
265     double ca = dist(c, a);
266     r = rCircumCircle(ab, bc, ca);
267     if(Equal(r, ab)) return point((a.x+b.x)/2, (a.y+b.y)/2);
268     if(Equal(r, bc)) return point((b.x+c.x)/2, (b.y+c.y)/2);
269     if(Equal(r, ca)) return point((c.x+a.x)/2, (c.y+a.y)/2);
270     line AB, BC;
271     pointsToLine(a, b, AB);
272     pointsToLine(b, c, BC);
273     line perpenAB = perpendicularLine(AB, point((a.x+b.x)/2, (a.y+b.y)/2));
274     line perpenBC = perpendicularLine(BC, point((b.x+c.x)/2, (b.y+c.y)/2));
275     point center;
276     areIntersect(perpenAB, perpenBC, center);
277     return center;
278 }
279
280 // ----- Trapezoid -----
281 double TrapezoidArea(double a, double b, double c, double d) { // a and c are parallel
282     double BASE = fabs(a-c);
283     double AREA = TriangleArea(d, b, BASE);
284     double h = (AREA*2)/BASE;
285     return ((a+c)/2)*h;
286 }

```

File: /mnt/Work/notes/AhoCorasick.cpp

```

1 // Aho-Corasick
2 // Complexity : O(n+m+z)
3 // n : Length of text
4 // m : total length of all keywords
5 // z : total number of occurrence of word in text
6
7 const int TOTKEY = 505; // Total number of keywords
8 const int KEYLEN = 505; // Size of maximum keyword
9 const int MAXS = TOTKEY*KEYLEN + 10; // Max number of states in the matching machine.
10 // Should be equal to the sum of the length of all keywords.
11 const int MAXC = 26; // Number of characters in the alphabet.
12 bitset<TOTKEY> out[MAXS]; // Output for each state, as a bitwise mask.
13 int f[MAXS]; // Failure function
14 int g[MAXS][MAXC]; // Goto function, or -1 if fail.
15
16 int buildMatchingMachine(const vector<string> &words, char lowestChar = 'a', char highestChar = 'z') {
17     for(int i = 0; i < MAXS; ++i)
18         out[i].reset();
19     memset(f, -1, sizeof f);
20     memset(g, -1, sizeof g);
21
22     int states = 1; // Initially, we just have the 0 state
23     for(int i = 0; i < (int)words.size(); ++i) {
24         const string &keyword = words[i];
25         int currentState = 0;
26         for(int j = 0; j < (int)keyword.size(); ++j) {
27             int c = keyword[j] - lowestChar;
28             if(g[currentState][c] == -1) // Allocate a new node
29                 g[currentState][c] = states++;
30             currentState = g[currentState][c];
31         }
32         out[currentState].set(i); // There's a match of keywords[i] at node currentState.
33     }
34
35     for(int c = 0; c < MAXC; ++c) // State 0 should have an outgoing edge for all characters.
36         if(g[0][c] == -1)
37             g[0][c] = 0;
38
39     // Now, let's build the failure function
40     queue<int> q;
41     for(int c = 0; c <= highestChar - lowestChar; ++c) // Iterate over every possible input
42         if(g[0][c] != -1 and g[0][c] != 0) { // All nodes s of depth 1 have f[s] = 0
43             f[g[0][c]] = 0;
44             q.push(g[0][c]);
45         }
46
47     while(q.size()) {
48         int state = q.front();
49         q.pop();
50         for(int c = 0; c <= highestChar - lowestChar; ++c) {
51             if(g[state][c] != -1) {
52                 int failure = f[state];
53                 while(g[failure][c] == -1)
54                     failure = f[failure];
55                 failure = g[failure][c];
56                 f[g[state][c]] = failure;
57                 out[q.state][c] |= out[failure]; // Merge out values
58             }
59         }
60     }
61 }

```

```

59     q.push(g.state[c]);
60 }}}}
61 return states;
62 }
63
64 int findNextState(int currentState, char nextInput, char lowestChar = 'a') {
65     int answer = currentState;
66     int c = nextInput - lowestChar;
67     while(g[answer][c] == -1)
68         answer = f[answer];
69     return g[answer][c];
70 }
71
72 int cntTOTKEY;
73 void Matcher(vector<string> &keywords, string &text) {
74     int currentState = 0;
75     memset(cnt, 0, sizeof cnt);
76
77     for(int i = 0; i < (int)text.size(); ++i) {
78         currentState = findNextState(currentState, text[i]);
79         if(out[currentState] == 0) // Nothing new, let's move on to the next character.
80             continue;
81
82         for(int j = 0; j < (int)keywords.size(); ++j)
83             if(out[currentState][j]) // Matched keywords[j]
84                 ++cnt[j];
85     }
86 }
87
88 string text; str;
89 vector<string> keywords;
90 // RETURN NUMBER OF MATHCES FOR EACH WORD APPEARING IN "KEYWORD" VECTOR
91 // INPUT STRING IS "TEXT"
92 int main() {
93     int t, n;
94     cin >> t;
95     for(int Case = 1; Case <= t; ++Case) {
96         cin >> n >> text;
97         while(n--) {
98             cin >> str;
99             keywords.push_back(str);
100         }
101         buildMatchingMachine(keywords);
102         cout << "Case " << Case << "\n";
103         Matcher(keywords, text);
104         for(int i = 0; i < (int)keywords.size(); ++i)
105             cout << cnt[i] << "\n";
106         keywords.clear();
107     }
108     return 0;
109 }

```

File: /mnt/Work/notes/APSPfloydwarshall.cpp

```

1 // All Pair Shortest Path
2 // Floyd Warshal
3 // Complexity : O(V^3)
4
5 int G[MAX][MAX], parent[MAX][MAX];
6 void graphINIT() {
7     for(int i = 0; i < MAX; i++)
8         for(int j = 0; j < MAX; j++)
9             G[i][j] = INF;
10    for(int i = 0; i < MAX; i++)
11        G[i][i] = 0;
12 }
13 void floydWarshall(int V) {
14     for(int i = 0; i < V; i++) // path printing matrix initialization
15         for(int j = 0; j < V; j++)
16             parent[i][j] = i; // we can go to j from i by only obtaining i (by default)
17     for(int k = 0; k < V; k++) // Selecting a middle point as k
18         for(int i = 0; i < V; i++) // Selecting all combination of source (i) and destination (j)
19             for(int j = 0; j < V; j++)
20                 if(G[i][k] != INF && G[k][j] != INF) { // if the graph contains negative edges, then min(INF, INF+ negative edge) = +-INF!
21                     G[i][j] = min(G[i][j], G[i][k] + G[k][j]); // if G[i][i] = negative, then node i is in negative circle
22                     parent[i][j] = parent[k][j]; // if path printing needed
23                 }
24 void printPath(int i, int j) {
25     if(i != j) printPath(i, parent[i][j]);
26     printf("%d", j);
27 }
28 void minMax(int V) {
29     for(int k = 0; k < V; k++)
30         for(int i = 0; i < V; i++)
31             for(int j = 0; j < V; j++)

```

```

32         G[i][j] = min(G[i][j], max(G[i][k], G[k][j]));
33     }
34     void transitiveClosure(int V) {
35         for(int k = 0; k < V; k++)
36             for(int i = 0; i < V; i++)
37                 for(int j = 0; j < V; j++)
38                     G[i][j] |= (G[i][k] & G[k][j]);
39     }

```

File: /mnt/Work/notes/ArticulationPoint.cpp

```

1 //Articulation Point
2 //Complexity O(V+E)
3 //Tarjan, DFS
4
5 vector<int> G[101];
6 int dfs_num[101], dfs_low[101], parent[101], isArticulationPoint[101], dfsCounter, rootChildren, dfsRoot;
7 void articulationPoint(int u) {
8     dfs_low[u] = dfs_num[u] = ++dfsCounter;
9     for(int i = 0; i < G[u].size(); i++) {
10         int v = G[u][i];
11         if(dfs_num[v] == 0) {
12             parent[v] = u;
13             if(u == dfsRoot) // Special case for root node
14                 rootChildren++; // if root node has child, increment counter
15             articulationPoint(v);
16             // 1 : if dfs_num[u] == dfs_low[v], then it is a back edge
17             // 2 : if dfs_num[u] < dfs_low[v], then u is ancestor of v and there is no back edge
18             // so, if u is not root node, then we can chose u for Articulation Point
19             if(dfs_num[u] <= dfs_low[v] && u != dfsRoot) //Avoiding root node
20                 isArticulationPoint[u]++;
21             // if there is any child node of u that is a back edge of a previous node
22             // then the value of dfs_low[v] might be less than the present dfs_low[u]
23             // we try to save the lowest value possible
24             dfs_low[u] = min(dfs_low[v], dfs_low[u]);
25         }
26         // As nodes are bi-directional, avoiding direct child node
27         // if it is not direct child node, and visited, then there is a back edge
28         // so we try to decrease the value of dfs_low[u] with the dfs_num[v]
29         // the dfs_num[v] is less than dfs_num[u] (as it is a back edge)
30         else if(parent[u] != v)
31             dfs_low[u] = min(dfs_low[u], dfs_num[v]);
32     }
33 }
34 int main() {
35     // Actual code of Articulation Point starts here
36     dfsCounter = 0;
37     memset(dfs_num, 0, sizeof(dfs_num));
38     isArticulationPoint.reset();
39     for(int i = 1; i <= n; i++) {
40         if(dfs_num[i] == 0) {
41             dfsCounter = rootChildren = 0;
42             dfsRoot = i;
43             articulationPoint(i);
44             isArticulationPoint[i] = (rootChildren > 1);
45         }
46         // Important
47         isArticulationPoint + 1 = number of nodes that is disconnected
48     }
49     // Printing Articulation Points
50     /*for(int i = 0; i < 101; i++)
51         if(isArticulationPoint[i])
52             printf("%d ", i);
53     printf("\n");*/
54     printf("%d\n", (int)isArticulationPoint.count());
55 }

```

File: /mnt/Work/notes/BFS_Bicolor.cpp

```

1 // Basic BFS with path printing
2 // Complexity : O(V+E)
3
4 vector<int> parent, G[MAX];
5 void printPath(int u, int source_node) { // destination, source
6     if(u == source_node) {
7         printf("%d", u);
8         return;
9     }
10    printPath(parent[u], source_node);
11    printf(" %d", u);
12 }
13

```

```

14 int BFS(int source_node, int finish_node, int vertices) {
15     vector<int> dist(vertices+5, INF); //contains the distance from source to end point
16     queue<int> Q;
17     Q.push(source_node);
18     parent.resize(vertices+5, -1); //for path printing
19     dist[source_node] = 0;
20
21     while(!Q.empty()) {
22         int u = Q.front();
23         Q.pop();
24         if(u == finish_node) //remove this line if shortest path to all nodes are needed
25             return dist[u];
26         for(int i = 0; i < G[u].size(); i++) {
27             int v = G[u][i];
28             if(dist[v] == INF) {
29                 dist[v] = dist[u] + 1;
30                 parent[v] = u;
31                 Q.push(v);
32             }
33         }
34     }
35     return -1;
36 }
37
38 int color[100]; // Contains Color (1, 2)
39 void Bicolor(int u) { // Bicolor Check
40     queue<int> q;
41     q.push(u);
42     color[u] = 1; // Color is -1 initialized
43     while(!q.empty()) {
44         u = q.front();
45         q.pop();
46         for(int i = 0; i < (int)G[u].size(); ++i) {
47             int v = G[u][i];
48             if(color[v] == -1) {
49                 if(color[u] == 1) color[v] = 2;
50                 else color[v] = 1;
51                 q.push(v);
52             }
53         }
54     }
55 }

```

File: /mnt/Work/notes/BigInt.cpp

```

1 // BigInteger By Jane Alam Jan
2
3 struct BigInt {
4     string a // to store the digits (in reverse order)
5     int sign // sign = -1 for negative numbers, sign = 1 otherwise
6     BigInt() {} // default constructor
7     BigInt(string b) { (*this) = b; } // constructor for string
8     BigInt(long long n) {
9         sign = n >= 0 ? 1 : -1;
10        if(n == 0) {
11            a.push_back('0');
12            return;
13        }
14        while(n) {
15            a.push_back(n%10 + '0');
16            n /= 10;
17        }
18        //reverse(a.begin(), a.end());
19    }
20    int size() { // returns number of digits
21        return a.size();
22    }
23    BigInt inverseSign() { // changes the sign
24        sign *= -1;
25        return (*this);
26    }
27    BigInt normalize(int newSign) { // removes leading 0, fixes sign
28        for(int i = a.size() - 1; i > 0 && a[i] == '0'; i--)
29            a.erase(a.begin() + i);
30        sign = (a.size() == 1 && a[0] == '0') ? 1 : newSign;
31        return (*this);
32    }
33    //----- assignment operator
34    void operator = (string b) { // assigns a string to BigInt
35        a = b[0] == '-' ? b.substr(1) : b;
36        reverse(a.begin(), a.end());
37        this->normalize(b[0] == '-' ? -1 : 1);
38    }
39    //----- conditional operators
40    bool operator < (const BigInt &b) const { // less than operator
41        if(sign != b.sign) return sign < b.sign;
42        if(a.size() != b.a.size())
43            return sign == 1 ? a.size() < b.a.size() : a.size() > b.a.size();
44        for(int i = a.size() - 1; i >= 0; i--) if(a[i] != b.a[i])
45            return sign == 1 ? a[i] < b.a[i] : a[i] > b.a[i];

```

```

46     return false;
47
48 }
49 bool operator == ( const Bigint &b ) const {      // operator for equality
50     return a == b a && sign == b sign;
51 }
52 // mathematical operators
53 void Pow(int p) {                                // Raises a Bigint to power of p
54     Bigint res("1");
55     while(p > 0) {
56         if(p&1) res = res * (*this);
57         p = p >> 1;
58         (*this) = (*this) * (*this);
59     }
60     (*this) = res;
61 }
62 Bigint operator + ( Bigint b ) {                 // addition operator overloading
63     if( sign != b sign ) return (*this) - b inverseSign();
64     Bigint c;
65     for(int i = 0, carry = 0; i < a size() || i < b size() || carry; i++) {
66         carry += (i < a size() ? a[i] - 48 : 0) + (i < b size() ? b[i] - 48 : 0);
67         c.a += (carry % 10 + 48);
68         carry /= 10;
69     }
70     return c.normalize(sign);
71 }
72 Bigint operator - ( Bigint b ) {                 // subtraction operator overloading
73     if( sign != b sign ) return (*this) + b inverseSign();
74     int s = sign; sign = b sign = 1;
75     if( (*this) < b ) return ((b - (*this)).inverseSign()).normalize(s);
76     Bigint c;
77     for(int i = 0, borrow = 0; i < a size(); i++) {
78         borrow = a[i] - borrow - (i < b size() ? b[i] : 48);
79         c.a += borrow >= 0 ? borrow + 48 : borrow + 58;
80         borrow = borrow >= 0 ? 0 : 1;
81     }
82     return c.normalize(s);
83 }
84 Bigint operator * ( Bigint b ) {                 // multiplication operator overloading
85     Bigint c("0");
86     for(int i = 0, k = a[i] - 48; i < a size(); i++, k = a[i] - 48) {
87         while(k--) c = c + b                      // ith digit is k, so, we add k times
88         b.a.insert(b.a.begin(), '0');           // multiplied by 10
89     }
90     return c.normalize(sign * b sign);
91 }
92 Bigint operator / ( Bigint b ) {                 // division operator overloading
93     if( b size() == 1 && b.a[0] == '0' ) b.a[0] /= ( b.a[0] - 48 );
94     Bigint c("0"), d;
95     for( int j = 0; j < a size(); j++) d.a += "0";
96     int dSign = sign * b sign; b sign = 1;
97     for( int i = a size() - 1; i >= 0; i-- ) {
98         c.a.insert( c.a.begin(), '0' );
99         c = c + a.substr( i, 1 );
100        while( !( c < b ) ) c = c - b; d.a[i]++;
101    }
102    return d.normalize(dSign);
103 }
104 Bigint operator % ( Bigint b ) {                 // modulo operator overloading
105     if( b size() == 1 && b.a[0] == '0' ) b.a[0] /= ( b.a[0] - 48 );
106     Bigint c("0");
107     b sign = 1;
108     for( int i = a size() - 1; i >= 0; i-- ) {
109         c.a.insert( c.a.begin(), '0' );
110         c = c + a.substr( i, 1 );
111         while( !( c < b ) ) c = c - b;
112     }
113     return c.normalize(sign);
114 }
115 //-----output method
116 void print() {
117     if( sign == -1 ) putchar("-");
118     for( int i = a size() - 1; i >= 0; i-- ) putchar(a[i]);
119 };
120
121 int main() {
122     Bigint a, b, c;    // declared some Bigint variables
123     string input;      // string to take input
124     cin >> input;      // take the Big integer as string
125     a = input;         // assign the string to Bigint a
126     cin >> input;      // take the Big integer as string
127     b = input;         // assign the string to Bigint
128     c = a + b;         // adding a and b
129     c.print();         // printing the Bigint
130     puts("");         // newline
131     return 0;
132 }

```


File: /mnt/Work/notes/BipartiteMatching.cpp

```

1 // Vertex Cover
2 // Wiki: Vertex Cover:
3 // In the mathematical discipline of graph theory, a vertex cover (sometimes node cover)
4 // of a graph is a set of vertices such that each edge of the graph is incident to at least one vertex of the set
5 // Wiki: Edge Cover:
6 // In graph theory, an edge cover of a graph is a set of edges such that every vertex of the graph
7 // is incident to at least one edge of the set
8
9 // Min Edge Cover = TotalNodes - MinVertexCover
10
11 bitset<MAX> vis;
12 int lft[MAX], rht[MAX];
13 vector<int> G[MAX];
14
15 int VertexCover(int u) { // Min Vertex Cover
16     vis[u] = 1;
17     for(int i = 0; i < (int)G[u].size(); ++i) {
18         int v = G[u][i];
19         if(vis[v]) continue; // If v is used earlier, skip
20         vis[v] = 1;
21         if(lft[v] == -1) { // If there is no node present on left of v
22             lft[v] = u, rht[u] = v;
23             return 1;
24         }
25         else if(VertexCover(lft[v])) { // If there is one node present on the left side of v (Let it be u')
26             lft[v] = u, rht[u] = v; // and if it is possible to match u' with another node (not v ofcourse!)
27             return 1; // then we can match this u with v, and u' is matched with another node as well
28         }
29     }
30     return 0;
31 }
32
33 int BPM(int n) { // Bipartite Matching
34     int cnt = 0;
35     memset(lft, -1, sizeof lft);
36     memset(rht, -1, sizeof rht);
37     for(int i = 1; i <= n; ++i) { // Nodes are numbered from 1 to n
38         vis.reset();
39         cnt += VertexCover(i); // Check if there exists a match for node i
40     }
41 }

```

File: /mnt/Work/notes/Bridge.cpp

```

1 //Complexity : O(V+E)
2 //Finding Bridges (Graph)
3
4 vector<int> G[MAX];
5 vector<pair<int, int> > ans;
6 int dfs_num[MAX], dfs_low[MAX], parent[MAX], dfsCounter;
7
8 void bridge(int u) {
9     // dfs_num[u] is the dfs counter of u node
10    // dfs_low[u] is the minimum dfs counter of u node (it is minimum if a backedge exists)
11    dfs_num[u] = dfs_low[u] = ++dfsCounter;
12    for(int i = 0; i < (int)G[u].size(); i++) {
13        int v = G[u][i];
14        if(dfs_num[v] == 0) {
15            parent[v] = u;
16            bridge(v);
17            // if dfs_num[u] is lower than dfs_low[v], then there is no back edge on u node
18            // so u - v can be a bridge
19            if(dfs_num[u] < dfs_low[v])
20                ans.push_back(make_pair(min(u, v), max(u, v)));
21            // obtainig lower dfs counter (if found) from child nodes
22            dfs_low[u] = min(dfs_low[u], dfs_low[v]);
23        }
24        // if v is not parent of u then it is a back edge
25        // also dfs_num[v] must be less than dfs_low[u]
26        // so we update it
27        else if(parent[u] != v)
28            dfs_low[u] = min(dfs_low[u], dfs_num[v]);
29    }
30 }
31
32 void FindBridge(int V){ //Bridge finding code
33     memset(dfs_num, 0, sizeof(dfs_num));
34     dfsCounter = 0;
35     for(int i = 0; i < V; i++)
36         if(dfs_num[i] == 0)
37             bridge(i);
38 }
39
40 int main() {
41     FindBridge(100);
42     // Output
43 }

```

```

40 sort(ans.begin(), ans.end());
41 for(int i = 0; i < ans.size(); i++)
42     printf("%d - %d\n", ans[i].first, ans[i].second);
43 printf("\n");
44 return 0;
45 }

```

File: /mnt/Work/notes/Decimal.cpp

```

1 vi DecimalVal(int a, int b) { // Calculate Decimal values (after .) of a/b
2     vi v;
3     a %= b;
4     if(a == 0) {
5         v.pb(0);
6         return v;
7     }
8     bool first = 1;
9     while(SIZE(v) <= 200) { // Define the Maximum Length of decimal values
10         if(a == 0)
11             return v; // If any Zero divisor is found (then, rest all will be Zero) return values
12         else if(a < b && !first) { // If we need to add another zero (add zero after first time)
13             a *= 10;
14             v.pb(0);
15         }
16         else if(a < b && first) { // If we need to add a extra zero (adding zero first time)
17             first = 0;
18             a *= 10;
19             continue;
20         }
21         else {
22             v.pb(a/b);
23             a %= b;
24             first = 1;
25         }
26     }
27     return v;
28 }
29
30 // Repetation (PunoPonik) is also calculated
31 vi dec1, dec2; // Before . (decimal), after . (decimal)
32 int DecimalRepeated(int a, int b) { // Calculate Decimal values (after .) of a/b
33     unordered_map<int, int> mp;
34     int k = 0, point = -1;
35     bool divisible = 0;
36     if(a >= b) { // Before Decimal Calculation
37         dec1.push_back(a/b);
38         a %= b;
39     }
40     if(dec1.size() == 0)
41         dec1.push_back(0);
42     while(a != 0) {
43         if(mp.find(a) != mp.end()) { // if the remainder is found again, there exists a loop
44             point = mp[a];
45             break;
46         }
47         if(a % b == 0) {
48             dec2.push_back(a/b);
49             break;
50         }
51         mp[a] = k++;
52         int cnt = 0;
53         while(a < b) {
54             a *= 10;
55             if(cnt != 0) {
56                 dec2.push_back(0);
57                 k++;
58             }
59             ++cnt;
60         }
61         if(cnt != 0 && mp.find(a) != mp.end()) {
62             point = mp[a];
63             break;
64         }
65         if(cnt == 1)
66             mp[a] = (k-1);
67         dec2.push_back(a/b);
68         a %= b;
69         if(a == 0) {
70             divisible = 1;
71             break;
72         }
73     }
74     return divisible == 1 ? 1 : ((int)dec2.size() - point);
75 }
76 int main() {
77     int a, b;

```

```

78 cin >> a >> b;
79 vi v = DecimalVal(a, b);
80 for(auto it : v)
81     cout << it;
82 cout << endl;
83 int Cycle = DecimalRepeated(a, b);
84 for(auto it : dec1)
85     cout << it;
86 cout << " ";
87 for(auto it : dec2)
88     cout << it;
89 cout << "\n\n";
90 cout << "Last Repeating Cycle " << Cycle << endl;
91 return 0;
92 }

```

File: /mnt/Work/notes/DFS.cpp

```

1 // Cycle in Directed graph
2 // http://codeforces.com/contest/915/problem/D
3
4 vi G[550];
5 int color[550], Cycle = 0; // Cycle will contain the number of cycles found in graph
6 void dfs(int u) {
7     color[u] = 2; // Mark as parent
8     for(auto v : G[u]) {
9         if(color[v] == 2) // If any Parent found (BackEdge)
10             Cycle++;
11         else if(!color[v])
12             dfs(v);
13     }
14     color[u] = 1; // Visited
15 }

```

File: /mnt/Work/notes/Dijkstra.cpp

```

1 // Shortest Path (Dijkstra)
2 // Complexity : (V*logV + E)
3
4 vector<int> dist, G[MAX], W[MAX];
5 void printPath(int u) { // call with ending node
6     if(u == s) { // s is the starting node
7         printf("%d", s); // base case, at the source s
8         return;
9     }
10    printPath(p[u]); // recursive: to make the output format: s -> ... -> t
11    printf(" %d", u);
12 }
13
14 void dijkstra(int u, int destination, int nodes) {
15     dist.resize(nodes+1, INF); // dist[v] contains the distance from u to v
16     dist[u] = 0;
17     priority_queue<pair<int, int>> pq; // pq is sorted in ascending order according to weight and edge
18     pq.push({0, -u});
19
20     while(!pq.empty()) {
21         int u = -pq.top().second;
22         int wu = -pq.top().first;
23         pq.pop();
24         if(u == destination) return; // if we only need distance of destination, then we may return
25         if(wu > dist[u]) continue; // skipping the longer edges, if we have found shorter edge earlier
26
27         for(int i = 0; i < G[u].size(); i++) {
28             int v = G[u][i];
29             int wv = W[u][i];
30             if(wu + wv < dist[v]) { // path relax
31                 dist[v] = wu + wv;
32                 p[v] = u; // path printing
33                 pq.push({-dist[v], -v});
34             }
35         }
36     }
37
38     // Kth Path Using Modified Dijkstra
39     // Complexity : O(K*(V*logV + E))
40     // http://codeforces.com/blog/entry/16821
41
42     vector<int> G[MAX], W[MAX], dist[MAX];
43     int KthDijkstra(int Start, int End, int Kth) { // Kth Shortest Path (Visits Edge Only Once)
44         for(int i = 0; i < MAX; ++i)
45             dist[i].clear();
46         priority_queue<pii> pq; // Weight, Node
47         pq.push(make_pair(0, Start));
48     }
49 }

```

```

46
47 while(!pq.empty()) {
48     int u = pq.top().second;
49     int w = -pq.top().first;
50     pq.pop();
51
52     if((int)dist[End].size() == Kth) // We can also break if the Kth path is found
53         return dist[End].back();
54     if(dist[u].empty())
55         dist[u].push_back(w);
56     else if(dist[u].back() != w) // Not taking same cost paths
57         dist[u].push_back(w); // As priority queue greedily chooses edge, it's guaranteed that this edge is bigger than previous
58     if((int)dist[u].size() > Kth) // Like basic dijkstra, we'll not take the Kth+ edges
59         continue;
60     for(int i = 0; i < (int)G[u].size(); ++i) {
61         int v = G[u][i];
62         int _w = w + W[u][i];
63         if((int)dist[v].size() == Kth)
64             continue;
65         pq.push(make_pair(-_w, v));
66     }
67     return -1;
68 }
69
70 int KthDijkstra(int Start, int End, int Kth) { // Kth Shortest Path (Visits Same Edge More Than Once if required)
71     for(int i = 0; i < MAX; ++i)
72         dist[i].clear();
73     priority_queue<pii> pq; // Weight, Node
74     pq.push(make_pair(0, Start));
75
76     while(!pq.empty()) {
77         int u = pq.top().second;
78         int w = -pq.top().first;
79         pq.pop();
80
81         if(dist[u].empty())
82             dist[u].push_back(w);
83         else if(dist[u].back() != w) { // if the weight is not same
84             if((int)dist[u].size() < Kth) // if we have to take more costs, take it
85                 dist[u].push_back(w);
86             else if(dist[u].back() <= w) // if the cost is greater than previous, then, don't go further
87                 continue;
88             else { // we have to take this cost, and remove the greater one
89                 dist[u].push_back(w);
90                 sort(dist[u].begin(), dist[u].end());
91                 dist[u].pop_back();
92             }
93         }
94         for(int i = 0; i < (int)G[u].size(); ++i) {
95             int v = G[u][i];
96             int _w = w + W[u][i];
97             pq.push(make_pair(-_w, v));
98         }
99         if((int)dist[End].size() < Kth) return -1;
100     }
101     return dist[End].back();
102 }
103
104 // Kth Shortest Path (Every edge and shortest path of previous calculation is not used)
105 vector<int> G[MAX], W[MAX], S[MAX]; //edge, edge_weight, reverse_shortest_paths_graph
106 int dist[MAX];
107 bool cut_node[MAX], cut_edge[MAX][MAX];
108
109 int dijkstra(int source, int end, int nodes) {
110     for(int i = 0; i < nodes; ++i) // dist[v] contains the distance from u to v
111         dist[i] = INF;
112     dist[source] = 0;
113     priority_queue<pair<int, int>> pq; // pq is sorted in ascending order according to weight and edge
114     pq.push({0, -source});
115
116     while(!pq.empty()) {
117         int u = pq.top().second;
118         int wu = -pq.top().first;
119         pq.pop();
120         if(wu > dist[u]) continue; // skipping the longer edges, if we have found shorter edge earlier
121
122         for(int i = 0; i < (int)G[u].size(); ++i) {
123             int v = G[u][i];
124             int wv = W[u][i];
125
126             if(cut_node[v] || cut_edge[u][v]) // if there exists node/edge that is used in previous shortest path
127                 continue;
128             if(wu + wv < dist[v]) { // path relax
129                 dist[v] = wu + wv;
130                 S[v].clear(); // if this edge is smaller than other edge, then we refresh the reverse paths of this node
131                 S[v].push_back(u); // then push back the node, (building a reverse graph of shortest path(s))
132                 pq.push({dist[v], -v});
133             }
134             else if(wu + wv == dist[v]) // if there is more than one shortest paths, then only add it in the reverse graph, nothing else

```

```

134     S[v].push_back(u);
135 }
136 return dist[end];
137 }
138
139 void cut_off(int start, int destination) { // this function cuts off all the nodes
140     if(destination == start) return;
141     for(int i = 0; i < S[destination].size(); i++) {
142         int v = S[destination][i];
143         cut_node[v] = 1;
144         cut_edge[destination][v] = cut_edge[v][destination] = 1;
145         cut_off(start, v);
146     }

```

File: /mnt/Work/notes/DP.cpp

```

1 //-----String DP-----
2 int Palindrome(int l, int r) { // Building Palindrome in minimum move
3     if(dp[l][r] != INF) return dp[l][r];
4     if(l >= r) return dp[l][r] = 0;
5     if(l+1 == r) return dp[l][r] = (s[l] != s[r]);
6     if(s[l] == s[r]) return dp[l][r] = Palindrome(l+1, r-1);
7     return dp[l][r] = min(Palindrome(l+1, r), Palindrome(l, r-1))+1; // Adding a alphabet on right, left
8 }
9
10 void dfs(int l, int r) { // Palindrome printing, for above DP function
11     if(l > r) return;
12     if(s[l] == s[r]) {
13         Palin.push_back(s[l]);
14         dfs(l+1, r-1);
15         if(l != r) Palin.push_back(s[r]);
16         return;
17     }
18     int P = min(make_pair(dp[l+1][r], 1), make_pair(dp[l][r-1], 2)).second;
19     if(P == 1) {
20         Palin.push_back(s[l]);
21         dfs(l+1, r);
22         Palin.push_back(s[l]);
23     }
24     else {
25         Palin.push_back(s[r]);
26         dfs(l, r-1);
27         Palin.push_back(s[r]);
28     }
29 }
30 bool isPalindrome(int l, int r) { // Checks if substring l-r is palindrome
31     if(l == r || l > r) return 1;
32     if(dp[l][r] != -1) return dp[l][r];
33     if(s[l] == s[r]) return dp[l][r] = isPalindrome(l+1, r-1);
34     return 0;
35 }
36
37 int recur(int p1, int p2) { // make string s1 like s2, in minimum move
38     if(dp[p1][p2] != INF)
39         return dp[p1][p2];
40     if(p1 == l1 || p2 == l2) { // reached end of string s1 or s2
41         if(p1 < l1) return dp[p1][p2] = recur(p1+1, p2)+1;
42         if(p2 < l2) return dp[p1][p2] = recur(p1, p2+1)+1;
43         return dp[p1][p2] = 0;
44     }
45     if(s1[p1] == s2[p2]) // match found
46         return dp[p1][p2] = recur(p1+1, p2+1);
47     // change at position p1, delete position p1, insert at position p1
48     return dp[p1][p2] = min(recur(p1+1, p2+1), min(recur(p1+1, p2), recur(p1, p2+1)))+1;
49 }
50
51 void dfs(int p1, int p2) { // printing function for above dp
52     if(dp[p1][p2] == 0) // end point (value depends on topdown/bottomup)
53         return;
54     if(s1[p1] == s2[p2]) { // match found, no operation
55         dfs(p1+1, p2+1);
56         return;
57     }
58     int P = min(mp[dp[p1+1][p2], 1], min(mp[dp[p1][p2+1], 2], mp[dp[p1+1][p2+1], 3])); second
59     if(P == 1) dfs(p1+1, p2); // delete s1[p1] from position p2 of s1 string
60     else if(P == 2) dfs(p1, p2+1); // insert s2[p2] on position p2 of s1 string
61     else dfs(p1+1, p2+1); // change s1[p2] to s2[p2] on position p2 of string s1
62 }
63
64 int reduce(int l, int r) { // Reduce string AXDOODOO (len : 8) to AX(DO^2)^2 (len : 4)
65     if(l > r) return INF;
66     if(l == r) return 1;
67     if(dp[l][r] != -1) return dp[l][r];
68     int ret = r-l+1;
69     int len = ret;
70     for(int i = l; i < r; ++i) // A B D O O D O O remove A X substring

```

```

71     ret = min(ret, reduce(l, i)+reduce(i+1, r));
72     for(int d = 1; d < len; ++d) { // D O O D O O to check all divisable length substrng
73         if(len%d != 0) continue;
74         for(int i = l+d; i <= r; i += d)
75             for(int k = 0; k < d; ++k)
76                 if(s[l+k] != s[i+k])
77                     goto pass;
78         ret = min(ret, reduce(l, l+d-1));
79         pass;
80     }
81     return dp[l][r] = ret;
82 }
83
84 // Light OJ 1073 - DNA Sequence
85 // FIND and PRINT shortest string after merging multiple string together
86
87 int matchDP[20][20];
88 int TryMatch(int x, int y) { // Finds First overlap of two string
89     if(matchDP[x][y] != -1) // ABAAB + AAB : Match at 2
90         return matchDP[x][y];
91     for(size_t i = 0; i < v[x].size(); ++i) {
92         for(size_t j = i; k = 0; j < v[x].size() && k < v[y].size(); ++j, ++k)
93             if(v[x][j] != v[y][k])
94                 goto pass;
95         return matchDP[x][y] = i;
96         pass;
97     }
98     return matchDP[x][y] = v[x].size();
99 }
100
101 int dp[16][[(1<<15)+100];
102 int recur(int mask, int last) { // DP part of LIGHT OJ
103     if(dp[last][mask] != -1) // eliminate all substrings from n string first in main function!
104         return dp[last][mask]; // it's not handled here
105     if(mask == (1<<n)-1)
106         return dp[last][mask] = v[last].size();
107     int ret = INF, cost;
108     for(int i = 0; i < n; ++i) {
109         if(isOn(mask, i))
110             continue;
111         int mPos = TryMatch(last, i);
112         if(mPos < (int)v[last].size())
113             cost = (int)v[last].size() - ((int)v[last].size() - mPos);
114         else
115             cost = v[last].size();
116         ret = min(ret, recur(mask | (1<<i), i) + cost);
117     }
118     return dp[last][mask] = ret;
119 }
120
121 string ans;
122 void dfs(int mask, int last, string ret) { // PRINTING part of LIGHT OJ
123     if(!ret.empty() && ans < ret)
124         return;
125     if(mask == (1<<n)-1) {
126         ret += v[last];
127         if(ret < ans)
128             ans = ret;
129         return;
130     }
131     for(int i = 0; i < n; ++i) {
132         if(isOn(mask, i))
133             continue;
134         int mPos = TryMatch(last, i);
135         int cost;
136         if(mPos < (int)v[last].size())
137             cost = (int)v[last].size() - ((int)v[last].size() - mPos);
138         else
139             cost = v[last].size();
140         if(dp[last][mask] - cost == dp[i][mask | (1<<i)])
141             dfs(mask | (1<<i), i, ret + v[last].substr(0, cost));
142     }
143 }
144 //-----Digit DP-----
145
146 // Complexity : O(10*idx*sum*tight) : LightOJ 1068
147 // Tight contains if there is any restriction to number (Tight is initially 1)
148 // Initial Params: (MaxDigitSize-1, 0, 0, 1, modVal, allowed_digit_vector)
149
150 ll dp[15][100][100][2];
151 ll digitSum(int idx, int sum, ll value, bool tight, int mod, vector<int> &MaxDigit) {
152     if(idx == -1)
153         return ((value == 0) && (sum == 0));
154     if(dp[idx][sum][value][tight] != -1)
155         return dp[idx][sum][value][tight];
156     ll ret = 0;
157     int lim = (tight)? MaxDigit[idx] : 9; // Numbers are generated in reverse order
158     for(int i = 0; i <= lim; i++) {

```

```

159     bool newTight = (MaxDigit[idx] == i)? tight : 0; // calculating newTight value for next state
160     ll newValue = value ? ((value*10) % mod)+i : i;
161     ret += digitSum[idx-1, (sum+i)%mod, newValue%mod, newTight, mod, MaxDigit];
162 }
163 return dp[idx][sum][value][tight] = ret;
164 }
165
166 // Bit DP (Almost same as Digit DP) : LighOJ 1032
167 // Complexity O(2^pos*total_bits*tights*number_of_bits)
168 // Initial Params : (MostSignificantOnBitPos, N, 0, 0, 1)
169 // Call as : bitDP(SigOnBitPos, N, 0, 0, 1) N is the Max Value, calculating [0 - N]
170 // Tight is initially on
171 // pairs are number of paired bits, prevOn shows if previous bit was on (it is for this problem)
172
173 #define isOn(x, i) (x & (1LL<<i))
174 #define On(x, i) (x | (1LL<<i))
175 #define Off(x, i) (x & ~(1LL<<i))
176 int N, lastBit;
177 long long dp[33][33][2][2];
178 ll bitDP(int pos, int mask, int pairs, bool prevOn, bool tight) {
179     if(pos < 0)
180         return pairs;
181     if(dp[pos][pairs][prevOn][tight] != -1)
182         return dp[pos][pairs][prevOn][tight];
183     bool newTight = tight & !isOn(mask, pos); // Turn off tight when we are turning off a bit which was initially on
184     ll ans = bitDP(pos-1, Off(mask, pos), pairs, 0, newTight);
185     if(On(mask, pos) <= N)
186         ans += bitDP(pos-1, On(mask, pos), pairs + prevOn, 1, tight && isOn(mask, pos));
187     return dp[pos][pairs][prevOn][tight] = ans;
188 }
189
190 // Memory Optimized DP + Bottom Up solution (LOJ : 1126 - Building Twin Towers)
191 // given array v of n elements, make two value x1 and x2 where x1 == x2, output maximum of it
192
193 int dp[2][500010], n; // present dp table and past dp table
194 int BottomUp(int TOT) { // TOT = (Cumulative Sum of v)/2
195     memset(dp, -1, sizeof dp);
196     dp[0][0] = 0;
197     bool present = 0, past = 1;
198     for(int i = 0; i < n; ++i) {
199         present ^= 1, past ^= 1; // Swapping present and past dp table
200         for(int diff = 0; diff <= TOT; ++diff)
201             if(dp[past][diff] != -1) {
202                 int moreDiff = diff + v[i], lessDiff = abs(diff - v[i]);
203                 dp[present][diff] = max(dp[present][diff], dp[past][diff]);
204                 dp[present][lessDiff] = max(dp[present][lessDiff], max(dp[past][lessDiff], dp[past][diff] + v[i]));
205                 dp[present][moreDiff] = max(dp[present][moreDiff], max(dp[past][moreDiff], dp[past][diff] + v[i]));
206             }
207     }
208     return (max(dp[0][0], dp[1][0]))/2; // Returns the maximum possible answer
209 }
210
211 // Count Number of ways to go from (1, 1) to (r, c) if there exists n unassassable points (only eight and down is valid move)
212 ll CountNumberOfWays(int r, int c, int n) {
213     v[n] = {r, c}; // also add the last point as unaccessable point, to find how many
214     sort(v.begin(), v.end()); // ways we can come to this point, which is the answer
215     for(int i = 0; i <= n; ++i) {
216         dp[i] = CountWay(1, 1, v[i].first, v[i].second); // Number of ways we can come from starting square
217         for(int j = 0; j < i; ++j)
218             if(v[j].first <= v[i].first and v[j].second <= v[i].second)
219                 dp[i] = (dp[i] - (dp[j] * CountWay(v[j].first, v[j].second, v[i].first, v[i].second))%MOD + MOD)%MOD;
220     }
221     return dp[n]; // Number of ways we can reach from (1, 1) to (r, c)
222     // The last state is always (r, c), which is the answer
223 }
224
225 // Travelling Salesman
226 // dist[u][v] = distance from u to v
227 // dp[u][bitmask] = dp[node][set_of_taken_nodes] (saves the best(min/max) path)
228 // call with tsp(starting node, 1)
229
230 int n, x[11], y[11], dist[11][11], memo[11][1<<11], dp[11][1<<11];
231 int TSP(int u, int bitmask) { // startin node and bitmask of taken nodes
232     if(bitmask == ((1<<n)-1)) // when it steps in this node, if all nodes are visited
233         return dist[u][0]; // then return to 0'th (starting) node [as the path is hamiltonian]
234     // or use return dist[u][start] if starting node is not 0
235     if(dp[u][bitmask] != -1) // if we have previous value set up
236         return dp[u][bitmask]; // use that previous value
237     int ans = 1e9; // set an infinite value
238     for(int v = 0; v < n; v++) // for all the nodes
239         if(u != v && !(bitmask & (1<<v))) // if this node is not the same node, and if this node is not used yet(in bitmask)
240             ans = min(ans, dist[u][v] + tsp(v, bitmask | (1<<v))); // min(past_val, dist u->v + dist(v->all other untaken nodes))
241     return dp[u][bitmask] = ans; // save in dp and return
242 }

```

File: /mnt/Work/notes/DSU.cpp

```

1 // Basic DSU with compression
2
3 struct DSU {
4     vector<int> u_list, u_set; // u_list[x] : the size of a set x, u_set[x] : the root of x
5     DSU() {}
6     DSU(int SZ) { init(SZ); }
7     int unionRoot(int n) { // Union making with dynamic compression
8         if(u_set[n] == n) return n;
9         return u_set[n] = unionRoot(u_set[n]); // Directly set the actual root of this set as root (Compress)
10    }
11    int makeUnion(int a, int b) { // Union making with compression
12        int x = unionRoot(a), y = unionRoot(b);
13        if(x == y) return x; // If both are in same set
14        else if(u_list[x] > u_list[y]) { // Makes x root (y -> x)
15            u_set[y] = x;
16            u_list[x] += u_list[y]; // Root's size is increased
17            return x;
18        }
19        else { // Makes y root (x -> y)
20            u_set[x] = y;
21            u_list[y] += u_list[x]; // Root's size is increased
22            return y;
23        }
24    }
25    void init(int len) { // Initializer
26        u_list.resize(len+5);
27        u_set.resize(len+5);
28        for(int i = 0; i <= len+3; i++)
29            u_set[i] = i, u_list[i] = 1; // Each node contains itself, so size of each node set to 1
30    }
31    bool isRoot(int x) { // Returns true if this is a root (May contain one or many nodes)
32        return u_set[x] == x;
33    }
34    bool isRootContainsMany(int x) { // If the root contains more than one value (Actual Root)
35        return (isRoot(x) && (u_list[x] > 1));
36    }
37    bool isSameSet(int a, int b) { // If a and b is in same set/component
38        return (unionRoot(a) == unionRoot(b));
39    }
40 }
41 // Bipartite DSU (Tested)
42 struct BipartiteDSU {
43     vector<int> u_list, u_set, u_color;
44     vector<bool> mismatch; // Bicolor mismatch
45
46     BipartiteDSU() {}
47     BipartiteDSU(int SZ) { init(SZ); }
48
49     pll unionRoot(int n) { // Union making with dynamic compression
50         if(u_set[n] == n) return {n, u_color[n]};
51         pll root = unionRoot(u_set[n]);
52         if(mismatch[u_set[n]] or mismatch[n])
53             mismatch[n] = mismatch[u_set[n]] = 1;
54         u_color[n] = (u_color[n] + root.second & 1);
55         u_set[n] = root.first; // Directly set the actual root of this set as root (Compress)
56         return {u_set[n], u_color[n]};
57     }
58     int makeUnion(int a, int b) { // Union making with compression
59         int x = unionRoot(a).first, y = unionRoot(b).first;
60         if(x == y) { // If both are in same set and bipartite mismatch exists
61             if(u_color[a] == u_color[b]) mismatch[x] = 1;
62             return x;
63         }
64         if(mismatch[x] or mismatch[y]) // Checks if Bipartite mismatch exists
65             mismatch[x] = mismatch[y] = 1;
66         if(u_list[x] < u_list[y]) { // Makes x root (y -> x)
67             u_set[x] = y;
68             u_list[x] += u_list[y]; // Root's size is increased
69             u_color[x] = (u_color[a] + u_color[b] + 1) & 1; // Setting color of component y according to the color of a & b
70             return y;
71         }
72         else { // Makes y root (x -> y)
73             u_set[y] = x;
74             u_list[y] += u_list[x]; // Root's size is increased
75             u_color[y] = (u_color[a] + u_color[b] + 1) & 1; // Setting color of component y according to the color of a & b
76             return x;
77         }
78     }
79     void init(int len) { // Initializer
80         u_list.resize(len+5);
81         u_set.resize(len+5);
82         u_color.resize(len+5);
83         mismatch.resize(len+5);
84         for(int i = 0; i <= len+3; i++)
85             u_set[i] = i, u_list[i] = 1, u_color[i] = 0, mismatch[i] = 0;
86     }
87     bool isRoot(int x) { // Returns true if this is a root (May contain one or many nodes)
88         return u_set[x] == x;
89     }
90 }

```



```

89 bool isRootContainsMany(int x) { // If the root contains more than one value (Actual Root)
90     return (isRoot(x) && (u_list[x] > 1));
91 }
92 bool isSameSet(int a, int b) { // If a and b is in same set/component
93     return (unionRoot(a).first == unionRoot(b).first);
94 }
95 int getColor(int u) { // Color of node u (DONT get the color of root)
96     return u_color[u];
97 }
98 bool hasMismatch(int x) { // If there is bipartite mismatch in this set/component
99     return mismatch[x];
100 };
101
102 // Dynamic Weighted DSU (Checked, Not Tested)
103
104 struct WeightedDSU {
105     vector<int> u_list, u_set, u_weight, weight;
106     WeightedDSU() {}
107     WeightedDSU(int SZ) { init(SZ); }
108     int unionRoot(int n) { // Union making with compression
109         if(u_set[n] == n) return n;
110         return u_set[n] = unionRoot(u_set[n]); // Directly set the actual root of this set as root (Compress)
111     }
112     void changeWeight(int u, int w, bool first = 1) { // Change any component's weight (Dynamic)
113         if(first) w = w - weight[u];
114         u_weight[u] += w;
115         if(u_set[u] != u)
116             changeWeight(u_set[u], w, 0);
117     }
118     int makeUnion(int a, int b) { // Union making with compression
119         int x = unionRoot(a), y = unionRoot(b);
120         if(x == y) return x;
121         if(u_list[x] > u_list[y]) { // Makes x root (y -> x)
122             u_set[y] = x;
123             u_list[x] += u_list[y]; // Root's size is increased
124             u_weight[x] += u_weight[y]; // Root's weight is increased
125             return x;
126         }
127         else { // Makes y root (x -> y)
128             u_set[x] = y;
129             u_list[y] += u_list[x]; // Root's size is increased
130             u_weight[y] += u_weight[x]; // Root's weight is increased
131             return y;
132         }
133     }
134     void init(int len) { // Initializer
135         u_list.resize(len+5);
136         u_set.resize(len+5);
137         u_weight.resize(len+5);
138         weight.resize(len+5);
139         for(int i = 0; i <= len+3; i++)
140             u_set[i] = i, u_list[i] = 1, u_weight[i] = weight[i] = 0;
141     }
142     bool isRoot(int x) { // Returns true if this is a root (May contain one or many nodes)
143         return u_set[x] == x;
144     }
145     bool isRootContainsMany(int x) { // If the root contains more than one value (Actual Root)
146         return (isRoot(x) && (u_list[x] > 1));
147     }
148     bool isSameSet(int a, int b) { // If a and b is in same set/component
149         return (unionRoot(a) == unionRoot(b));
150     }
151     void setWeight(int u, int w) { // Set weight of node u to w, run before union
152         u_weight[u] = w;
153         weight[u] = w;
154     }
155     int getComponentWeight(int u) { // Get weight sum of the set/comopnent
156         return u_weight[unionRoot(u)];
157     };
158 };

```

File: /mnt/Work/notes/FenwickTree.cpp

```

1 // 1D Fenwick Tree
2
3 struct BIT {
4     ll tree[MAX];
5     int MaxVal;
6     void init(int sz=1e7) {
7         memset(tree, 0, sizeof tree);
8         MaxVal = sz+1;
9     }
10    void update(int idx, ll val) {
11        for(; idx <= MaxVal; idx += (idx & -idx))
12            tree[idx] += val;
13    }
14    void update(int l, int r, ll val) {

```

```

15     if(l > r) swap(l, r);
16     update(l, val);
17     update(r+1, -val);
18 }
19 // read(int idx) {
20     // sum = 0;
21     for(; idx > 0; idx -= (idx & -idx))
22         sum += tree[idx];
23     return sum;
24 }
25 // read(int l, int r) {
26     // ret = read(r) - read(l-1);
27     return ret;
28 }
29 // readSingle(int idx) {          // Point read in log(n)
30     // sum = tree[idx];
31     if(idx > 0) {
32         int z = idx - (idx & -idx);
33         --idx;
34         while(idx != z) {
35             sum += tree[idx];
36             idx -= (idx & -idx);
37         }
38     }
39     return sum;
40 }
41 int search(int cSum) {
42     int pos = -1, lo = 1, hi = MaxVal, mid;
43     while(lo <= hi) {
44         mid = (lo+hi)/2;
45         if(read(mid) >= cSum) { // read(mid) >= cSum : to find the lowest index of cSum value
46             pos = mid; // read(mid) == cSum : to find the greatest index of cSum value
47             hi = mid-1;
48         }
49         else
50             lo = mid+1;
51     }
52     return pos;
53 }
54 // size() {
55     // return read(MaxVal);
56 }
57 // Modified BIT, this section can be used to add/remove/read 1 to all elements from 1 to pos
58 // all of the inverse functions must be used, for any manipulation
59 // invRead(int idx) { // gives summation from 1 to idx
60     // return read(MaxVal-idx);
61 }
62 void invInsert(int idx) { // adds 1 to all index less than idx
63     update(MaxVal-idx, 1);
64 }
65 void invRemove(int idx) { // removes 1 from idx
66     update(MaxVal-idx, -1);
67 }
68 void invUpdate(int idx, // val) {
69     update(MaxVal-idx, val);
70 }
71 // ----- 2D Fenwick Tree -----
72 // *
73 // y |
74 // | (x1,y2) ----- (x2,y2)
75 // | | |
76 // | | |
77 // | -----
78 // | (x1,y1) (x2, y1)
79 // |
80 // |_____
81 // (0, 0) x--> */
82
83 ull tree[2510][2510];
84 int xMax = 2505, yMax = 2505;
85 // Updates from min point to MAX LIMIT
86 void update(int x, int y, // val) {
87     int y1;
88     while(x <= xMax) {
89         y1 = y;
90         while(y1 <= yMax) {
91             tree[x][y1] += val;
92             y1 += (y1 & -y1);
93         }
94         x += (x & -x);
95     }
96 }
97 // read(int x, int y) { // Reads from (0, 0) to (x, y)
98     // sum = 0;
99     int y1;
100     while(x > 0) {
101         y1 = y;
102         while(y1 > 0) {
103             sum += tree[x][y1];

```

```

103     y1 -= (y1 & -y1);
104 }
105     x -= (x & -x);
106 }
107 return sum;
108 }
109 // readSingle(int x, int y) {
110 return read(x, y) + read(x-1, y-1) - read(x-1, y) - read(x, y-1);
111 }
112 void updateSquare(pii p1, pii p2, ll val) { // p1 : lower left point, p2 : upper right point
113     update(p1.first, p1.second, val);
114     update(p1.first, p2.second+1, -val);
115     update(p2.first+1, p1.second, -val);
116     update(p2.first+1, p2.second+1, val);
117 }
118 // readSquare(pii p1, pii p2) { // p1 : lower left point, p2 : upper right point
119 ll ans = read(p2.first, p2.second);
120 ans -= read(p1.first-1, p2.second);
121 ans -= read(p2.first, p1.second-1);
122 ans += read(p1.first-1, p1.second-1);
123 return ans;
124 }
125
126 // ----- 3D Fenwick Tree -----
127
128 // tree[105][105][105];
129 // xMax = 100, yMax = 100, zMax = 100;
130 void update(int x, int y, int z, ll val) {
131     int y1, z1;
132     while(x <= xMax) {
133         y1 = y;
134         while(y1 <= yMax) {
135             z1 = z;
136             while(z1 <= zMax) {
137                 tree[x][y1][z1] += val;
138                 z1 += (z1 & -z1);
139             }
140             y1 += (y1 & -y1);
141         }
142         x += (x & -x);
143     }
144 }
145 // read(int x, int y, int z) {
146 ll sum = 0;
147 while(x > 0) {
148     y1 = y;
149     while(y1 > 0) {
150         z1 = z;
151         while(z1 > 0) {
152             sum += tree[x][y1][z1];
153             z1 -= (z1 & -z1);
154         }
155         y1 -= (y1 & -y1);
156     }
157     x -= (x & -x);
158 }
159 return sum;
160 }
161 // readRange(ll x1, ll y1, ll z1, ll x2, ll y2, ll z2) {
162 -x1, -y1, -z1
163 return read(x2, y2, z2)
164 - read(x1, y2, z2)
165 - read(x2, y1, z2)
166 - read(x2, y2, z1)
167 + read(x1, y1, z2)
168 + read(x1, y2, z1)
169 + read(x2, y1, z1)
170 - read(x1, y1, z1);
171 }
172 void updateRange(int x1, int y1, int z1, int x2, int y2, int z2) { // Not tested yet!!
173     update(x1, y1, z1, val);
174     update(x2+1, y1, z1, -val);
175     update(x1, y2+1, z1, -val);
176     update(x1, y1, z2+1, -val);
177     update(x2+1, y2+1, z1, val);
178     update(x1, y2+1, z2+1, val);
179     update(x2+1, y1, z2+1, val);
180     update(x2+1, y2+1, z2+1, -val);
181 }
182 // Patterns to built BIT update read:
183 // always starts with first(starting point), add val
184 // take (1 to n) elements from ending point with all combination add it to starting point, add (-1)^n * val

```

File: /mnt/Work/notes/FractionAndBase.cpp

```

1 struct fraction { // Fraction Calculation Template

```

```

2  int a, b;
3  fraction() {
4      a = 1;
5      b = 1;
6  }
7  fraction(int x, int y) : a(x), b(y) {}
8  flip() {swap(a, b);}
9  fraction operator + (fraction other) {
10     fraction temp;
11     temp.b = (b)*(other.b)/(__gcd(b, other.b));
12     temp.a = (temp.b/b)*a + (temp.b/other.b)*other.a;
13     int x = __gcd(temp.a, temp.b);
14     if(x != 1) {temp.a/=x; temp.b/=x;}
15     return temp;
16 }
17 fraction operator - (fraction other) {
18     fraction temp;
19     temp.b = (b*other.b)/__gcd(b, other.b);
20     temp.a = (temp.b/b)*a - (temp.b/other.b)*other.a;
21     int x = __gcd(temp.a, temp.b);
22     if(x != 1) {temp.a/=x; temp.b/=x;}
23     return temp;
24 }
25 fraction operator / (fraction other) {
26     fraction temp;
27     temp.a = a*other.b;
28     temp.b = b*other.a;
29     int x = __gcd(temp.a, temp.b);
30     if(x != 1) {temp.a/=x; temp.b/=x;}
31     return temp;
32 }
33 fraction operator * (fraction other) {
34     fraction temp;
35     temp.a = a*other.a;
36     temp.b = b*other.b;
37     int x = __gcd(temp.a, temp.b);
38     if(x != 1) {temp.a/=x; temp.b/=x;}
39     return temp;
40 };
41
42 struct BaseInt { // Number Base Conversions
43     string val;
44     int base;
45     BaseInt() {}
46     BaseInt(string _val, int _base = 10) { // Do check if any value if val is greater than base
47         val = _val; // Which is impossible
48         base = _base;
49     }
50     char reVal(int num) {
51         if(num >= 0 && num <= 9) return (char)(num + '0');
52         return (char)(num - 10 + 'A');
53     }
54     int getVal(char c) {
55         if(c <= '9' && c >= '0') return c-'0';
56         return c-'A'+10;
57     }
58     void DecimalTo(int _base) {
59         int v = stoll(val);
60         base = _base;
61         val.clear();
62         while(v) {
63             val.push_back(reVal(v%base));
64             v /= base;
65         }
66         reverse(val.begin(), val.end());
67         if(val.empty()) val.push_back('0');
68     }
69     bool ToDecimal() {
70         int ret = 0;
71         for(int i = 0; i < (int)val.size(); ++i) {
72             int v = getVal(val[i]);
73             if(v >= base) return 0;
74             if(i) ret *= base;
75             ret += v;
76         }
77         val = to_string(ret); base = 10;
78         return 1;
79     }
80     void ChangeBase(int to) {
81         if(base == to) return; // If input is "000", then output will also be "000" (if base remains same)
82         if(base != 10) ToDecimal(); // remove the if statements to recover
83         if(to != 10) DecimalTo(to);
84     }
85     void Reverse() {
86         reverse(val.begin(), val.end());
87     }
88     BaseInt operator + (BaseInt other) const {
89         BaseInt a(val, base), b = other;

```

```

90     a.ToDecimal(), b.ToDecimal());
91     string sum = to_string(stoi(a_val, 0) + stoi(b_val, 0));
92     BaseInt ret(sum);
93     ret.ChangeBase(base);
94     return ret;
95 });

```

File: /mnt/Work/notes/Hash.cpp

```

1 // Hashing
2 // p = 31, 51
3 // MOD = 1e9+9, 1e7+7
4 const ll p = 31;
5 const ll mod1 = 1e9+9, mod2 = 1e9+7;
6
7 // Returns Single Hash Val
8 ll hash(char *s, int len, ll mod = 1e9+9) {
9     int p = 31;
10    ll hashVal = 0;
11    ll pPow = 1;
12    for(int i = 0; i < len; ++i) {
13        hashVal = (hashVal + (s[i] - 'a' + 1) * pPow)%mod;
14        pPow = (pPow * p)%mod;
15    }
16    return hashVal;
17 }
18 vl Hash(char *s, int len) {
19     ll hashVal = 0;
20     vector<ll> v;
21     for(int i = 0; i < len; ++i) {
22         hashVal = (hashVal + (s[i] - 'a' + 1) * Power[i])%mod;
23         v.push_back(hashVal);
24     }
25     return v;
26 }
27 bool MATCH(pll a, pll b) {
28     while(a.fi <= a.se) {
29         if(s1[a.fi] != s2[b.fi])
30             return 0;
31         a.fi++, b.fi++;
32     }
33     return 1;
34 }
35 void PowerGen(int n) {
36     Power.resize(n+1);
37     Power[0] = 1;
38     for(int i = 1; i < n; ++i)
39         Power[i] = (Power[i-1] * p)%mod;
40 }
41 ll SubHash(vl &Hash, ll l, ll r, ll LIM) {
42     ll H;
43     H = (Hash[r] - (l-1 >= 0 ? Hash[l-1]:0) + mod)%mod;
44     H = (H * Power[LIM-l])%mod;
45     return H;
46 }
47
48 // ----- DOUBLE HASH GENERATORS -----
49 // Generates Hash of entire string without PowerGen func
50 vector<pair<ll, ll> > doubleHash(char *s, int len, ll mod1 = 1e9+7, ll mod2 = 1e9+9) {
51     ll hashVal1 = 0, hashVal2 = 0, pPow1 = 1, pPow2 = 1;
52     vector<pair<ll, ll> > v;
53     for(int i = 0; i < len; ++i) {
54         hashVal1 = (hashVal1 + (s[i] - 'a' + 1) * pPow1)%mod1;
55         hashVal2 = (hashVal2 + (s[i] - 'a' + 1) * pPow2)%mod2;
56         pPow1 = (pPow1 * p)%mod1;
57         pPow2 = (pPow2 * p)%mod2;
58         v.push_back({hashVal1, hashVal2});
59     }
60     return v;
61 }
62 void PowerGen(int n) {
63     Power.resize(n+1);
64     Power[0] = {1, 1};
65     for(int i = 1; i < n; ++i) {
66         Power[i].first = (Power[i-1].first * p)%mod1;
67         Power[i].second = (Power[i-1].second * p)%mod2;
68     }
69 }
70 vl doubleHash(char *s, int len) { // Returns Double Hash vector for a full string
71     ll hashVal1 = 0, hashVal2 = 0;
72     vector<pll> v;
73     for(int i = 0; i < len; ++i) {
74         hashVal1 = (hashVal1 + (s[i] - 'a' + 1) * Power[i].fi)%mod1;
75         hashVal2 = (hashVal2 + (s[i] - 'a' + 1) * Power[i].se)%mod2;
76         v.push_back({hashVal1, hashVal2});
77     }
78 }

```

```

77 return v;
78 }
79 pll SubHash(vll &Hash, ll l, ll r, ll LIM) { // Produce SubString Hash
80     pll H;
81     H.fi = (Hash[r].fi - (l-1 >= 0 ? Hash[l-1].fi : 0) + mod1)%mod1;
82     H.se = (Hash[r].se - (l-1 >= 0 ? Hash[l-1].se : 0) + mod2)%mod2;
83     H.fi = (H.fi * Power[LIM-l].fi)%mod1;
84     H.se = (H.se * Power[LIM-l].se)%mod2;
85     return H;
86 }
87 // Returns True if the Hashval of length len exists in subrange [l, r] of Hash vector
88 bool MatchSubStr(int l, int r, vector<pll> &Hash, pll HashVal, int len) {
89     for(int Start = l, End = l+len-1; End <= r; ++End, ++Start) {
90         pll pattHash, strHash;
91         pattHash.first = (HashVal.first * Power[Start].first)%mod1;
92         pattHash.second = (HashVal.second * Power[Start].second)%mod2;
93         strHash.first = (Hash[End].first - (Start == 0 ? 0 : Hash[Start-1].first) + mod1)%mod1;
94         strHash.second = (Hash[End].second - (Start == 0 ? 0 : Hash[Start-1].second) + mod2)%mod2;
95         if(strHash == pattHash) return 1;
96     }
97     return 0;
98 }

```

File: /mnt/Work/notes/HeavyLightDecompose_HLD.cpp

```

1 // Heavy Light Decompose + Segment Tree
2 // Tree node value update, Tree node distance
3
4 int parent[MAX], level[MAX], nextNode[MAX], chain[MAX], num[MAX], val[MAX], numToNode[MAX], top[MAX], ChainSize[MAX], mx[MAX];
5 int ChainNo = 1, all = 1, n;
6 vi G[MAX];
7 void dfs(int u, int Parent) {
8     parent[u] = Parent; // Parent of u
9     ChainSize[u] = 1; // Number of child (initially the size is 1, contains only 1 node. itself) (resued array in hld)
10    for(int i = 0; i < SIZE(G[u]); ++i) {
11        int v = G[u][i];
12        if(v == Parent) // if the connected node is parent, skip
13            continue;
14        level[v] = level[u]+1; // level of the child node is : level of parent node + 1
15        dfs(v, u);
16        ChainSize[u] += ChainSize[v]; // Modify this line if max Chain is needed
17        if(nextNode[u] == -1 || ChainSize[v] > ChainSize[nextNode[u]])
18            nextNode[u] = v; // next selected node of u (select the node which has more child, (HEAVY))
19    }
20    void hld(int u, int Parent) {
21        chain[u] = ChainNo; // Chain Number
22        num[u] = all++; // Numbering all nodes
23        if(ChainSize[ChainNo] == 0) // if this is the first node
24            top[ChainNo] = u; // mark this as the root node of the n'th chain
25        ChainSize[ChainNo]++;
26        if(nextNode[u] != -1) // if this node has a child, go to it
27            hld(nextNode[u], u); // the next node is included in the chain
28        for(int i = 0; i < SIZE(G[u]); ++i) {
29            int v = G[u][i]; // if this node is parent node or, this node is already included in the chain, skip
30            if(v == Parent || v == nextNode[u]) continue;
31            ChainNo++; // this is a new (light) chain, so increment the chain no. counter
32            hld(v, u);
33        }
34    }
35    int GetSum(int u, int v) {
36        int res = 0;
37        while(chain[u] != chain[v]) { // While two nodes are not in same chain
38            if(level[top[chain[u]]] < level[top[chain[v]]]) // u is the chain which's topmost node is deeper
39                swap(u, v);
40            int start = top[chain[u]];
41            res += query(1, 1, n, num[start], num[u]); // Run query in u node's chain
42            u = parent[start]; // go to the upper chain of u
43        }
44        if(num[u] > num[v]) swap(u, v);
45        res += query(1, 1, n, num[u], num[v]);
46        return res;
47    }
48    void updateNodeVal(int u, int val) {
49        update(1, 1, n, num[u], val); // Updating the value of chain
50    }
51    void numToNodeConv(int n) {
52        for(int i = 1; i <= n; ++i) numToNode[num[i]] = i;
53    }
54    int main() {
55        memset(nextNode, -1, sizeof nextNode);
56        ChainNo = 1, all = 1;
57        dfs(1, 1);
58        memset(ChainSize, 0, sizeof ChainSize); // array reused in hld
59        hld(1, 1);
60        numToNodeConv(n);
61        init(1, 1, n);

```

61 }

File: /mnt/Work/notes/IntervalSum.cpp

```
1 // Interval Sum
2 // Complexity: query*log(query)
3 // http://codeforces.com/contest/915/problem/E
4
5 struct Interval {
6     set<pair<ll, ll> > Set          // Contains Segment Endpoints {r, l}
7     map<pair<ll, ll>, ll> Val;       // Contains Segment Values {l, r} = k
8     int TOTlen;                    // Contains Total Segment Covered Length
9     void init(int sz = -1) {
10         Set clear(), Val clear(), TOTlen = 0;
11         if (sz > 0)                // Will be initialized if size declared (NOT needed)
12             Set insert(make_pair(sz, 1)), Val[make_pair(1, sz)] = 0;
13     }
14     void Insert(ll l, ll r, ll val) {
15         set<pair<ll, ll> >::iterator it = Set lower_bound({l, 0LL});
16         while (it != Set end() && it->second <= r) {
17             ll segL = it->second, segR = it->first;          // Overlapped segment
18             Set erase(it++);                                // Erase and point to the next segment
19             ll L = max(segL, l), R = min(segR, r);          // Erased segment's partial L and R
20             TOTlen -= R - L + 1;
21             if (segL < l) {
22                 Set insert({l-1, segL});
23                 Val[{segL, l-1}] = Val[{segL, segR}];
24             }
25             if (segR > r) {
26                 Set insert({segR, r+1});
27                 Val[{r+1, segR}] = Val[{segL, segR}];
28             }
29             Val erase({segL, segR});
30         }
31         TOTlen += r - l + 1;
32         Set insert(make_pair(r, l));
33         Val[make_pair(l, r)] = val;
34     }
35     ll getSum(ll l, ll r) {
36         ll sum = 0;
37         set<pair<ll, ll> >::iterator it = Set lower_bound({l, 0LL});
38         while (it != Set end() && it->second <= r) {
39             ll segL = it->second, segR = it->first;          // Overlapped segment
40             ll V = Val[{segL, segR}];
41             sum += (segR - segL + 1) * V;
42             if (segL < l) sum -= (l - segL) * V;
43             if (segR > r) sum -= (segR - r) * V;
44             ++it;
45         }
46         return sum;
47     };
48     vector<ll> CountInterval(int n) // returns number of overlaps of all inclusive points
49     vector<pair<ll, int> > v;      // segments start/end and marker (segments are l - r inclusive)
50     vector<ll> ret(n+1);          // returns : ret[number_of_overlaps] = total_number_of_points
51     ll l, r;
52     while (n--) {
53         cin >> l >> r;           // input
54         v.push_back({l, 1}), v.push_back({r+1, -1});      // r+1 as l - r is segment boundary
55     }
56     sort(v.begin(), v.end());
57     for (int i = 0; cnt = v[0].second; i < (int)v.size(); ++i, cnt += v[i].second) // cnt contains the overlaps
58         if (i+1 < (int)v.size() and v[i].first != v[i+1].first) // v[i].first == v[i+1].first then
59             ret[cnt] += v[i+1].first - v[i].first // there may exist more points at front, so take them first
60     return ret;
61 }
```

File: /mnt/Work/notes/KMP.cpp

```
1 // Knuth Morris Pratt
2 // Complexity : O(String + Token)
3
4 //-----Genuine PrefixTable (Prefix-Suffix Length)-----
5 // Some Tricky Cases: aaaaaa : 0 1 2 3 4 5   aaaabaa : 0 1 2 3 0 1 2   abcdabacd : 0 0 0 0 1 2 3 4
6 void prefixTable(int n, char pat[], int table[]) {
7     int len = 0, i = 1;           // length of the previous longest prefix suffix
8     table[0] = 0;                 // table[0] is always 0
9     while (i < n) {
10         if (pat[i] == pat[len]) {
11             len++;
12             table[i] = len;
13             i++;
14         }
15     }
```

```

15     else {                                     // pat[i] != pat[len]
16         if (len != 0) len = table[len-1];      // find previous match
17         else table[i] = 0, i++;                // if (len == 0) and mismatch
18     }}                                         // set table[i] = 0, and go to next index
19
20 void KMP(int strLen, int patLen, char str[], char pat[], int table[]) {
21     int i = 0, j = 0; // i : string index, j : pattern index
22     while (i < N) {
23         if (str[i] == pat[j]) i++, j++;
24         if (j == M) {
25             printf("Found pattern at index %d n", i-j);
26             j = table[j-1]; // Match found, try for next match
27         }
28         else if (i < strLen && str[i] != pat[j]) { // Match not found
29             if (j != 0) j = table[j-1]; // if j != 0, then go to the prev match index
30             else i = i+1; // if j == 0, then we need to go to next index of str
31         }}
32
33 // ----- 2D KMP -----
34
35 unordered_map<string, int> patt; // Clear after each Kmp2D call
36 int flag = 0; // Set to zero before calling PrefixTable
37 // r : Pattern row, c : Pattern column // table : prefix table (1D array)
38 // s : Pattern String (C++ string) // Followed Felix-Halim KMP
39 vector<int> PrefixTable2D(int r, int c, int table[], string s[]) {
40     vector<int> Row; // Contains Row mapped string index
41     for (int i = 0; i < r; ++i) {
42         if (patt.find(s[i]) == patt.end()) {
43             patt[s[i]] = ++flag;
44             Row.push_back(flag);
45         }
46         else Row.push_back(patt[s[i]]);
47     }
48     table[0] = -1;
49     int i = 0, j = -1;
50     while (i < r) {
51         while (j >= 0 && Row[i] != Row[j])
52             j = table[j];
53         ++i, ++j;
54         table[i] = j;
55     }
56     return Row; // Returns Hashed index of each row in pattern string
57 }
58
59 // StrR StrC : String Row and Column // PattR PattC : Pattern row and column
60 // Str : String (C++ String) // Patt : Pattern (C++ String) // table : Prefix table of pattern (1D array)
61 vector<pair<int, int>> Kmp2D(int StrR, int StrC, int PattR, int PattC, string Str[], string Patt[], int table[]) {
62     int mat[StrR][StrC];
63     int limC = StrC - PattC;
64     vector<int> PattRow = PrefixTable2D(PattR, PattC, table, Patt);
65     for (int i = 0; i < StrR; ++i)
66         for (int j = 0; j <= limC; ++j) {
67             string tmp = Str[i].substr(j, PattC);
68             if (patt.find(tmp) == patt.end()) { // Generating String Mapped using same mapping values
69                 patt[tmp] = ++flag; // Stored in matrix
70                 mat[i][j] = flag;
71             }
72             else mat[i][j] = patt[tmp];
73         }
74     vector<pair<int, int>> match; // This will contain the starting Row & Column of matched string
75     for (int c = 0; c <= limC; ++c) { // Scan columnwise
76         int i = 0, j = 0;
77         while (i < StrR) {
78             while (j >= 0 && mat[i][c] != PattRow[j])
79                 j = table[j];
80             ++i, ++j;
81             if (j == PattR) match.push_back(make_pair(i-j, c));
82         }
83     }
84     return match;
85 }

```

File: /mnt/Work/notes/LCA.cpp

```

1 // LCA
2 // Least Common Ancestor with sparse table
3
4 v| G|MAX|, W|MAX|;
5 int level|MAX|, parent|MAX|, sparse|MAX|[20];
6 ll dist|MAX|, DIST|MAX|[20];
7
8 void dfs(int u, int par, int lvl, ll d) { // Tracks distance as well (From root 1 to all nodes)
9     level[u] = lvl; // parent[] and level[] is necessary
10    parent[u] = par;
11    dist[u] = d; // remove distance if not needed
12    for (int i = 0; i < (int)G[u].size(); ++i)

```



```

13     if(parent[u] != G[u][i])
14         dfs(G[u][i], u, lvl+1, d+W[u][i]);
15 }
16
17 void LCAinit(int V) {
18     memset(parent, -1, sizeof parent);
19     dfs(0, -1, 0); // DFS first
20     memset(sparse, -1, sizeof sparse); // Main initialization of sparse table LCA starts here
21     for(int u = 1; u <= V; ++u) // node u's 2^0 parent
22         sparse[u][0] = parent[u];
23     for(int p = 1; v; (1LL<<p) <= V; ++p)
24         for(int u = 1; u <= V; ++u)
25             if((v = sparse[u][p-1]) != -1) // node u's 2^x parent = parent of node v's 2^(x-1) [ where node v : (node u's 2^(x-1) parent) ]
26                 sparse[u][p] = sparse[v][p-1];
27 }
28
29 int LCA(int u, int v) {
30     if(level[u] > level[v]) swap(u, v); // v is deeper
31     int p = ceil(log2(level[v]));
32
33     for(int i = p; i >= 0; --i) // Pull up v to same level as u
34         if(level[v] - (1LL<<i) >= level[u])
35             v = sparse[v][i];
36     if(u == v) return u; // if u WAS the parent
37
38     for(int i = p; i >= 0; --i) // Pull up u and v together while LCA not found
39         if(sparse[v][i] != -1 && sparse[u][i] != sparse[v][i]) // -1 check is if 2^i is out of calculated range
40             u = sparse[u][i], v = sparse[v][i];
41     return parent[u];
42 }
43
44 // ----- LCA WITH DISTANCE -----
45 void distDP(int V) { // initialiser for LCA_with_DIST, call after LCAinit()
46     for(int u = 1; u <= V; ++u) // NOTE : DIST[u][0] = weight of node u
47         DIST[u][0] = W[u]; // Where W[u] = weight of node u
48     for(int p = 1; (1<<p)<=V; ++p)
49         for(int u = 1; u <=V; ++u) {
50             int v = sparse[u][p-1];
51             if(v == -1) continue;
52             DIST[u][p] += DIST[u][p-1] + DIST[v][p-1];
53         }
54 }
55 int LCA_with_DIST(int u, int v, long long &w) { // w returns distance from u -> v
56     w = 0;
57     if(level[u] > level[v]) swap(u, v); // v is deeper
58     int p = ceil(log2(level[v]));
59     for(int i = p; i >= 0; --i) // Pull up v to same level as u
60         if(level[v] - (1LL<<i) >= level[u]) {
61             w += DIST[v][i];
62             v = sparse[v][i];
63         }
64     if(u == v) { // if u WAS the parent
65         w += DIST[v][0];
66         return u;
67     }
68     for(int i = p; i >= 0; --i) // Pull up u and v together while LCA not found
69         if(sparse[v][i] != -1 && sparse[u][i] != sparse[v][i]) // -1 check is if 2^i is out of calculated range
70             u = sparse[u][i], v = sparse[v][i];
71     w += DIST[v][0];
72     w += DIST[u][0];
73     w += DIST[sparse[v][0][0][0];
74     return parent[u];
75 }
76
77 // Distance(int u, int v) {
78     int lca = LCA(u, v);
79     return dist[v] + dist[u] - 2*dist[lca];
80 }
81
82 // ----- LCA WITH Sparse Table Vector -----
83 // DFS and LCA INIT is same
84 void MERGE(vector<int>&u, vector<int>&v) { // Do what is to be done to merge
85     for(auto it : v) u.push_back(it); // here taking lowest 10 values
86     sort(u.begin(), u.end());
87     while((int)u.size() > 10)
88         u.pop_back();
89 }
90
91 vector<int> W[MAX][20]; // W[u][0] will contain initial weight/weights at node u
92 vector<int> LCA(int u, int v) {
93     vector<int> T;
94     if(level[u] > level[v]) swap(u, v); // v is deeper
95     int p = ceil(log2(level[v]));
96     for(int i = p; i >= 0; --i) // Pull up v to same level as u
97         if(level[v] - (1LL<<i) >= level[u]) {
98             MERGE(T, W[v][i]);
99             v = sparse[v][i];
100         }

```

```

101 if(u == v) { // if u WAS the parent
102     MERGE(T, W[u][0]);
103     return T;
104 }
105 for(int i = p; i >= 0; --i) // Pull up u and v together while LCA not found
106     if(sparse[v][i] != -1 && sparse[u][i] != sparse[v][i]) { // -1 check is if 2^i is out of calculated range
107         MERGE(T, W[u][i]);
108         MERGE(T, W[v][i]);
109         u = sparse[u][i], v = sparse[v][i];
110     }
111 MERGE(T, W[u][0]); // As W[x][0] denoted the x nodes weight
112 MERGE(T, W[v][0]); // every sparse node must be calculated
113 MERGE(T, W[sparse[v][0][0]]); // we can also calculate summation of distance like this
114 return T;
115 }

```

File: /mnt/Work/notes/LongestIncreasingSequence_LIS.cpp

```

1 // Longest Increasing/Decrasing Sequence
2 // Complexity : nLog_n
3
4 // -----Non Printable Version-----
5
6 int main() {
7     // vector v contains the sequence
8     for(auto it : v) { // Use -it for decreasing sequences
9         auto pIT = upper_bound(LIS.begin(), LIS.end(), it); // Longest Non-Decreasing Sequence
10        if(pIT == LIS.end()) // For Longest Increasing Sequence use lower_bound
11            LIS.push_back(it);
12        else
13            *pIT = it;
14    }
15    return 0;
16 }
17
18 // -----Printable Version-----
19 // DP + BinarySearch (nLog_n)
20 // {1, 1, 9, 3, 8, 11, 4, 5, 6, 6, 4, 19, 7, 1, 7}
21 // Increasing : 1, 3, 4, 5, 6, 7
22 // NonDecreasing : 1, 1, 3, 4, 5, 6, 6, 7, 7
23
24 void findLIS(vector<int> &v, vector<int> &idx) { // v is the input values and idx vector contains index of the LIS values
25     if(v.empty()) return;
26     vector<int> dp(v.size()); // The memoization part, remembers what index is the previous index if any value is inserted or modified
27     idx.push_back(0); // Carrys index of values
28     int l, r;
29
30     for(int i = 1; i < (int)v.size(); i++) {
31         if(v[idx.back()] <= v[i]) { // **Replace < with <= if non-decreasing subsequence required
32             dp[i] = idx.back(); // If next element v[i] is greater than last element of
33             idx.push_back(i); // current longest subsequence v[idx.back()], just push it at back of "idx" and continue
34             continue;
35         }
36         // Binary search to find the smallest element referenced by idx which is just bigger than v[i] (UpperBound(v[i]))
37         // Note : Binary search is performed on idx (and not v)
38         for(l = 0, r = idx.size()-1; l < r; ) {
39             int mid = (l+r)/2;
40             if(v[idx[mid]] <= v[i]) l = mid+1; // **Replace < with <= if non-decreasing subsequence required
41             else r = mid;
42         }
43         if(v[i] < v[idx[l]]) { // Update idx if new value is smaller then previously referenced value
44             if(l > 0) dp[i] = idx[l-1];
45             idx[l] = i;
46         }
47     }
48     for(l = idx.size(), r = idx.back(); l--; r = dp[r])
49         idx[l] = r;
50 }

```

File: /mnt/Work/notes/MathFormula.cpp

```

1 // Math Formulas
2
3 // Find the number of b for which [b1, b2] | [a1, a2]
4 int FindDivisorInRange(int a1, int a2, int b1, int b2) {
5     int a = abs(a1 - a2);
6     int b = abs(b1 - b2);
7     int gcd = __gcd(a, b);
8     return 1 + gcd;
9 }
10
11 // Find how many integers from range m to n are divisible by a or b

```

```

12 int rangeDivisor(int m, int n, int a, int b) {
13     int lcm = LCM(a, b);
14     int a_divisor = n / a - (m - 1) / a;
15     int b_divisor = n / b - (m - 1) / b;
16     int common_divisor = n / lcm - (m - 1) / lcm;
17     int ans = a_divisor + b_divisor - common_divisor;
18     return ans;
19 }
20
21 // CSOD || n { // Cumulative Sum of Divisors in sqrt(n)
22 // ans = 0;
23 for (int i = 2; i * i <= n; ++i) {
24     int j = n / i;
25     ans += (i + j) * (j - i + 1) / 2;
26     ans += i * (j - i);
27 }
28 return ans;
29 }
30
31 int CountDivisible(int a, int b, int n) { // Returns the number of divisible value in range [a, b] by n (NOT TESTED)
32     if (a > b) swap(a, b);
33     a += n - a % n;
34     b -= b % n;
35     if (a > b) return 0;
36     return ceil((b - a + 1) / (double)n);
37 }
38
39 int FactorialCount(int n, int p = 5) { // Returns how many value of p is present in n!
40     int ret = 0, r = p; // returns number of trailing zero of n! if p = 5
41     while (n / r != 0) {
42         ret += n / r;
43         r *= p;
44     }
45     return ret;
46 }
47
48 int TrailingZero(int n, int p = 1) { // Returns Trailing Zero of n^p
49     int cnt = 0; // Trailing Zero for any number : min(count_2_as_prime_factor, count_5_as_prime_factor)
50     while (n % 5 == 0 && n % 2 == 0)
51         n /= 5, n /= 2, ++cnt;
52     return cnt * p;
53 }
54
55 int BirthdayParadox(int days, int targetPercent = 50) { // Returns Number of people required so that probability is >= target
56     int people = 0; // Formula : 1 - (365/365) * (364/365) * (363/365) * .....
57     double percent = targetPercent / 100.0, gotPercent = 1;
58     for (; gotPercent > percent; ++people)
59         gotPercent *= (days - people + 1) / (double)days;
60     return people;
61 }
62
63 /* Euler's Totient function  $\Phi(n)$  for an input n is count of numbers in {1, 2, 3, ..., n}
64 * that are relatively prime to n, i.e., the numbers whose GCD (Greatest Common Divisor) with n is 1.
65 *  $\Phi(4)$  : GCD(1, 4) = 1, GCD(3, 4)
66 * so,  $\Phi(4) = 2$ 
67 */
68
69 int Phi(int n) { // Computes phi of n
70     int result = n;
71     for (int p = 2; p * p <= n; ++p) { // Consider all prime factors of n and subtract their multiples from result
72         if (n % p == 0) { // p is a prime factor of n
73             while (n % p == 0) // eliminate all p factors from n
74                 n /= p;
75             result -= result / p;
76         }
77     }
78     if (n > 1) // if n is still greater than 1, then it is also a prime
79         result -= result / n;
80     return result;
81 }
82
83 long long phi_MAX;
84 void computeTotient(int n) { // Computes phi or Euler Phi 1 to n
85     for (int i = 1; i <= n; ++i) // Initialize
86         phi[i] = i;
87     for (int p = 2; p <= n; ++p) { // Computation
88         if (phi[p] == p) { // if phi is not computed
89             phi[p] = p - 1; // p is prime and phi(prime) = prime - 1;
90             for (int i = 2 * p; i <= n; i += p) { // Sieve like implementation
91                 phi[i] = (phi[i] / p) * (p - 1); // Add contribution of p to its multiple i by multiplying with (1 - 1/p)
92             }
93         }
94     }
95     // Combination
96     // Complexity O(k)
97     long long C(int n, int k) {
98         long long c = 1;
99         if (k > n - k)
100             k = n - k;
101         for (int i = 0; i < k; ++i) {

```

```

100     c *= (n-i);
101     c /= (i+1);
102 }
103 return c;
104 }
105
106 // fa, MAX, fainv[MAX]; // fa and fainv must be in global
107 // C, n, r { // Usable if MOD value is present
108 if(fa[0] == 0) { // Auto initialize
109     fa[0] = 1; fainv[0] = powerMOD(1, MOD-2);
110     for(int i = 1; i < MAX; ++i) {
111         fa[i] = (fa[i-1]*i) % MOD; // Constant MOD
112         fainv[i] = powerMOD(fa[i], MOD-2);
113     }
114 if(n < 0 || r < 0 || n-r < 0) return 0; // Exceptional Cases
115 return ((fa[n] * fainv[r])%MOD * fainv[n-r])%MOD
116 }
117
118 // Catalan(int n) { // Cat(n) = C(2*n, n)/(n+1);
119 // c = C(2*n, n);
120 return c/(n+1);
121 }
122
123 // Building Pascale C(n, r)
124 // p[MAX][MAX];
125 void buildPascale() { // This Contains values of nCr : p[n][r]
126     p[0][0] = 1;
127     p[1][0] = p[1][1] = 1;
128     for(int i = 2; i <= 50; i++)
129         for(int j = 0; j <= i; j++) {
130             if(j == 0 || j == i)
131                 p[i][j] = 1;
132             else
133                 p[i][j] = p[i-1][j-1] + p[i-1][j];
134         }
135 }
136 // C(int n, int r) {
137 // if(r < 0 || r > n) return 0;
138 return p[n][r];
139 }
140
141 // STARS AND BARS THEOREM / Ball and Urn theorem
142 // If We have to Make x1+x2+x3+x4 = 12
143 // Then, the solution can be expressed as : {*****|*****} = {1+5+4+2}, {*****|***|*****} = {0+5+3+4}
144 // The summation is presented as total value, and the stars represented as 1, we use bars to separate values
145 // Number of ways we can produce the summation n, with k unknowns : C(n+k-1, n) = C(n+k-1, k-1)
146
147 // If numbers have lower limits, like x1 >= 3, x2 >= 2, x3 >= 1, x4 >= 1 (Let, the lower limits be l[i])
148 // Then the solution is : C(n-l1-l2-l3-l4+k-1, k-1)
149
150 // Ball & Urn : how many ways you can put 1 to n number in k sized array so that there are non decreasing?
151
152 // StarsAndBars(vector<int> &l, int n, int k) {
153 // if(l.empty()) for(int i = 0; i < k; ++i) n -= l[i]; // If l is empty, then there is no lower limit
154 return C(n+k-1, k-1);
155 }
156
157 // If numbers have both boundaries l1 <= x1 <= r1, l2 <= x2 <= r2, and x1+x2 = N
158 // then we can reduce the form to x1+x2 = N-l1-l2 and then x only gets upper limit x1 <= r1-l1+1, x2 <= r2-l2+1
159 // let r1-l1+1 be new l1, and r2-l2+1 be new l2, so x1 <= l1 and x2 <= l2, this limit is the opposite of basic Stars
160 // and Bars theorem, according to Principle of Inclusion Exclusion, this answer can be found as
161 // Answer = C(n+k-1, k-1) - C(n-l1+k-1, k-1) - C(n-l2+k-1, k-1) + C(n-l1-l2+k-1, k-1) .....
162
163 // StarsAndBarsInRange(l, l[], r[], n, k) {
164 // d[k+10], p[(1<<k)+10];
165 for(int i = 0; i < k; ++i) {
166     d[i] = r[i] - l[i] + 1;
167     n -= l[i];
168 }
169 // ret = C(n+k-1, k-1); p[0] = 0;
170 for(int i = 0; i < k; ++i) // Optimized Complexity : 2^n
171     for(int mask = (1<<i); mask < (1<<(i+1)); ++mask) {
172         p[mask] = p[mask ^ (1<<i)] + d[i];
173         ret += C(n-p[mask]+k-1, k-1) * (__builtin_popcount(mask)&1 ? -1:1);
174         ret %= MOD;
175     }
176 return (ret+MOD)%MOD;
177 }
178
179 // GetSameMOD(vector<ll> &v) { // Given an array v, find values k (k > 1), for which v[0]%k = v[1]%k ... = v[n]%k
180 // gcd // If a number K, leaves the same remainder with 2 numbers, then it must divide their difference.
181 sort(v.begin(), v.end()); // Find all numbers K which divide all the consecutive differences of all elements in the array.
182
183 for(int i = 0; i+1 < (int)v.size(); ++i) { // And it we will take the GCD of all consecutive differences
184     if(i == 0) gcd = v[i+1] - v[i];
185     else gcd = __gcd(gcd, v[i+1] - v[i]);
186 }
187 vector<ll> ret = Divisors(gcd); // GCD is the maximum value of k

```

```

188 ret push_back(gcd);          // All other values are the divisors of k
189 sort(ret.begin(), ret.end()); // NOTE : 1 is not added in the answer
190 return ret;
191 }
192
193 // CountZerosInRangeZeroTo (string n) {          // Returns number of zeros from 0 to n
194 // x = 0, fx = 0, gx = 0;
195 for(int i = 0; i < (int)n.size(); ++i){
196     // y = n[i] - '0';
197     fx = 10LL * fx + x - gx * (9LL - y); // Our formula
198     //fx += MOD; // If ans is to be returned in modded value
199     //fx %= MOD;
200     x = 10LL * x + y; // Now calculate the new x and g(x)
201     //x %= MOD;
202     if(y == 0LL) gx++;
203 }
204 return fx+1;
205 }
206
207 // NumOfSameValueInCombination(int n, int r) { // Returns number of same value in a set of nCr combination
208 if(n < r) return 0;
209 return C(n-1, r-1);
210 }
211
212 int cnt[MAX]; // cnt[x] : how many times x occurs in input
213 vector<int> genGCD(int mx) { // Counts how many number are there of gcd x
214     vector<int> sameGCD(mx+1); // input the MAXIMUM value
215     for(int gcd = mx; gcd >= 2; --gcd) { // Complexity : mx log_mx
216         int gcdCNT = cnt[gcd];
217         for(int mul = gcd+gcd; mul <= mx; mul += gcd)
218             gcdCNT += cnt[mul];
219         sameGCD[gcd] = gcdCNT;
220     }
221     return sameGCD;
222 }
223
224 // Multinomial : nC(k1,k2,k3,...km) is such that k1+k2+k3+...km = n and ki == kj and ki != kj both can be possible.
225 // Here Multinomial can be described as : nC(k1, k2, .. km) = nCk1 * (n-k1)Ck2 * (n-k1-k2)Ck3 * ..... (n-k1-k2-...)Ckm
226 // Let (a+b+c)^3 = a^3 + b^3 + c^3 + 3a^2b + 3b^2c + 3b^2a + 3b^2c + 3c^2a + 3c^2b + 6abc
227 // The coefficient can be retrieved as : 6abc = 3C(1, 1, 1) = 6 3b^2c = 3C(0, 2, 1) = 3
228 // In general terms it tells how many ways we can place k1, k2, k3, k4 people in 3 unique teams such that k1+k2+k3
229 // NOTE: if k1=k2=k3 = 2 and n = 6, and players numbered from 1 to 6, then 1,2|3,4|5,6 and 3,4|1,2|5,6 are considered to be different
230
231 // fa [MAX] = {0}; // fa and fainv must be in global
232 // Multinomial (N, vector<int> K) { // K contains all k1, k2, k3, if k1=k2=k3, then just push k1 once!!
233 if(fa[0] == 0) { // Auto initialize
234     fa[0] = 1; // fainv[0] = powerMOD(1, MOD-2);
235     for(int i = 1; i < MAX; ++i) {
236         fa[i] = (fa[i-1]*i) % MOD; // Constant MOD
237         // fainv[i] = powerMOD(fa[i], MOD-2); // Use factorial inverse if required
238     }
239     // k = 1;
240     if((int)K.size() == 1) k = powerMOD(fa[K[0]], N/K[0]); // k1 = k2 = .. = km, so k occurs N/K time
241     else for(auto it : K) k = (k * fa[it]) % MOD;
242     return (fa[N] * powerMOD(k, MOD-2)) % MOD; // Inverse mod
243 }
244
245 // NumOfWaysToPlace (N, K) { // Number of ways to make N/K teams from N people so that each team contains K people
246 vector<int> v; // If N = 6, then 1,2|3,4|5,6 and 3,4|1,2|5,6 is considered same
247 v.push_back(K);
248 return (Multinomial(N, v) * powerMOD(fa[N/K], MOD-2)) % MOD; // divide by k!, as 1,2|3,4|5,6 and 3,4|1,2|5,6 is considered same
249 }
250
251 // partial_derangement (int n, int r) { // Finds out how many ways we can place n numbers where r of them are not in their initial place
252 ull ans = f[n]; // Factorial of n!
253 for(int i = 1; i <= r; ++i) { // Formula: n! - C(n, 1)*(n-1)! + C(n, 2)*(n-2)! ..... + (-1)^r * C(n,r)*(n-r)!
254     if(i & 1) ans = (ans % MOD - C(r, i) * f[n-i]) % MOD; // Here C(r, i) is because we only have to choose from r elements, not n elements
255     else ans = (ans % MOD + C(r, i) * f[n-i]) % MOD;
256     ans = (ans + MOD) % MOD;
257 }
258 return ans % MOD;
259 }

```

File: /mnt/Work/notes/MatirxExponent.cpp

```

1 struct matrix {
2     matrix() { memset(mat, 0, sizeof(mat)); }
3     long long mat[MAXN][MAXN];
4 };
5 matrix mul(matrix a, matrix b, int p, int q, int r) { // O(n^3) :: r1, c1, c2 [c1 = r2]
6     matrix ans;
7     for(int i = 0; i < p; ++i)
8         for(int j = 0; j < r; ++j) {
9             ans.mat[i][j] = 0;
10            for(int k = 0; k < q; ++k)
11                ans.mat[i][j] = (ans.mat[i][j] % MOD + (a.mat[i][k] % MOD * b.mat[k][j] % MOD) % MOD) % MOD;

```

```

12     }
13     return ans;
14 }
15 matrix matPow(matrix base, ll p, int s) {          // O(logN), s : size of square matrix
16     if(p == 1) return base;
17     if(p & 1) return mul(base, matPow(base, p-1, s), s, s, s);
18     matrix tmp = matPow(base, p/2, s);
19     return mul(tmp, tmp, s, s, s);
20 }

```

File: /mnt/Work/notes/MaxFlow.cpp

```

1 // MaxFlow
2 // Ford-Fulkerson
3 // Complexity: O(VE^2)
4 // Graph Type : Directed/Undirected
5
6 const int MAX = 120;
7 vector<int> edge[MAX];
8 int V, E, rG[MAX][MAX], parent[MAX];
9
10 bool bfs(int s, int d) {          // augment path : source, destination
11     memset(parent, -1, sizeof parent);
12     queue<int> q;
13     q.push(s);
14     while(!q.empty()) {
15         int u = q.front();
16         q.pop();
17         for(auto v : edge[u])
18             if(parent[v] == -1 && rG[u][v] > 0) {
19                 parent[v] = u;
20                 if(v == d) return 1;
21                 q.push(v);
22             }
23     }
24     return 0;
25 }
26 int maxFlow(int s, int d) {        // source, destination
27     int max_flow = 0;
28     while(bfs(s, d)) {
29         int flow = INT_MAX;
30         for(int v = d; v != s; v = parent[v]) {
31             int u = parent[v];
32             flow = min(flow, rG[u][v]);
33         }
34         for(int v = d; v != s; v = parent[v]) {
35             int u = parent[v];
36             rG[u][v] -= flow;
37             rG[v][u] += flow;
38         }
39         max_flow += flow;
40     }
41     return max_flow;
42 }
43
44 int main() {
45     int u, v, w, source, destination, Case = 1;
46     map<pair<int, int>, bool> Map;
47     while(scanf("%d", &V) && V) {
48         scanf("%d%d%d", &source, &destination, &E);
49         memset(rG, 0, sizeof rG);
50         for(int i = 0; i < E; ++i) {
51             scanf("%d%d%d", &u, &v, &w);
52             rG[u][v] += w;          // edges are undirected
53             rG[v][u] += w;          // remove this line if edges are directed
54             if(Map.find({u, v}) == Map.end()) {          // same edges might occur more than once
55                 edge[u].push_back(v);          // to avoid n^2 calculation
56                 edge[v].push_back(u);
57                 Map[{u, v}] = Map[{v, u}] = 1;
58             }
59             printf("Network %d\n", Case++);
60             printf("The bandwidth is %d.\n\n", maxFlow(source, destination));
61
62             for(int i = 0; i <= V; ++i) edge[i].clear();
63             Map.clear();
64         }
65     }
66     return 0;
67 }

```

File: /mnt/Work/notes/MaxSum.cpp

```

1 //1D Max Sum

```

```

2 //Algorithm : Jay Kadane
3 //Complexity : O(n)
4
5 int main() {
6     int n;
7     scanf("%d", &n);
8     int A[n+1];
9     for(int i = 0; i < n; i++)
10         scanf("%d", &A[i]);
11
12 //Main part of the code
13 int sum = 0, ans = 0;
14 for(int i = 0; i < 9; i++) {
15     sum += A[i];
16     ans = max(sum, ans); //always take the larger sum
17     if(sum < 0)
18         sum = 0; //if sum is negative, reset it (greedy)
19 }
20 printf("1D Max Sum : %d\n", ans);
21 return 0;
22 }
23
24 //2D Max Sum
25 //Algorithm : DP, Inclusion Exclusion
26 //Complexity : O(n^4)
27
28 int main() {
29     int row_column, A[100][100]; //A square matrix
30     scanf("%d", &row_column);
31
32     for(int i = 0; i < row_column; i++) //input of the matrix/2D array
33         for(int j = 0; j < row_column; j++) {
34             scanf("%d", &A[i][j]);
35             if(i > 0) A[i][j] += A[i-1][j]; //take from right
36             if(j > 0) A[i][j] += A[i][j-1]; //take from left
37             if(i > 0 && j > 0) A[i][j] -= A[i-1][j-1]; //inclusion exclusion
38         }
39
40     int maxSubRect = -1e7;
41     for(int i = 0; i < row_column; i++) //i & j are the starting coordinate of sub-rectangle
42         for(int j = 0; j < row_column; j++)
43             for(int k = i; k < row_column; k++) //k & l are the finishing coordinate of sub-rectangle
44                 for(int l = j; l < row_column; l++) {
45                     int subRect = A[k][l];
46                     if(i > 0) subRect -= A[i-1][l];
47                     if(j > 0) subRect -= A[k][j-1];
48                     if(i > 0 && j > 0) subRect += A[i-1][j-1]; //due to inclusion exclusion
49                     maxSubRect = max(subRect, maxSubRect);
50                 }
51     printf("2D Max Sum : %d\n", maxSubRect);
52     return 0;
53 }

```

File: /mnt/Work/notes/MergeSort.cpp

```

1 // MergeSort
2
3 void MergeSort(long long arr[], int l, int mid, int r) {
4     int lftArrSize = mid-l+1, rhtArrSize = r-mid, lftArr[lftArrSize+2], rhtArr[rhtArrSize+2];
5
6     for(int i = l, j = 0; i <= mid; ++i, ++j)
7         lftArr[j] = arr[i];
8     for(int i = mid+1, j = 0; i <= r; ++i, ++j)
9         rhtArr[j] = arr[i];
10
11     lftArr[lftArrSize] = rhtArr[rhtArrSize] = 1e9; // INF value in both array (Basic merge sort algo)
12     int lPos = 0, rPos = 0;
13     for(int i = l; i <= r; ++i) {
14         if(lftArr[lPos] <= rhtArr[rPos])
15             arr[i] = lftArr[lPos++];
16         else {
17             arr[i] = rhtArr[rPos++];
18             //cnt += lftArrSize - lPos; // Delete this line if not needed (Min Number of Swaps)
19     }
20
21 void Divide(long long arr[], int l, int r) {
22     if(l == r || l > r) return;
23     int mid = (l+r)>>1;
24     Divide(arr, l, mid);
25     Divide(arr, mid+1, r);
26     MergeSort(arr, l, mid, r);
27 }
28
29 int main() {
30     Divide(v, 0, n-1);

```

```

31 return 0;
32 }

```

File: /mnt/Work/notes/ModularArithmetic.cpp

```

1 // Modular Arithmetic
2
3 // (2^10 % 5) = powMod(2, 10, 5)
4 long long powMod(long long N, long long P, long long M) {
5     if(P==0) return 1;
6     if(P%2==0) {
7         long long ret = powMod(N, P/2, M)%M;
8         return (ret * ret)%M;
9     }
10    return ((N%M) * (powMod(N, P-1, M)%M))%M;
11 }
12
13 // powerMOD(ll x, ll y) {          // Can find modular inverse by a^(MOD-2), a and MOD must be co-prime
14     ll res = 1;
15     x %= MOD;
16     while(y > 0) {
17         if(y&1) res = (res*x)%MOD;    // If y is odd, multiply x with result
18         y = y >> 1; x = (x * x)%MOD;
19     }
20     return res%MOD;
21 }
22
23 // 2^100 = Pow(2, 100)
24 unsigned long long Pow(unsigned long long N, unsigned long long P) {
25     if(P == 0) return 1;
26     if(P % 2 == 0) {
27         unsigned long long ret = Pow(N, P/2);
28         return ret * ret;
29     }
30     return N * Pow(N, P-1);
31 }
32
33 // calculate A mod B, where A : 0<A<(10^100000) (or greater)
34 // take input as string and process with aftermod
35 // calculate A^B mod M, where B : 0<A<(10^100000) (or greater)
36 // take input as string and process with aftermod : afterMod(inputAsString, Mod-1)    due to fermat theorem
37
38 long long afterMod(string str, ll mod) {    // input as string, as it is big, mod is the Mod value (Mod-1 if modding an exponentiation)
39     long long ans = 0;
40     string :: iterator it;
41
42     for(it = str.begin(); it != str.end(); it++)    // mod from first to last
43         ans = (ans*10 + (*it-'0')) % mod;
44     return ans;
45 }
46
47 // Exponent of Big numbers (N^P % M)
48 // where N and P is bigger strings (both having length 10^5)
49 long long bigExpo(char *N, char *P, long long M) {
50     long long base = 0, ans = 1;
51     for(int i = 0; N[i] != '\0'; ++i)
52         base = (base*10LL + N[i] - '0')%M;
53
54     for(int j = 0; P[j] != '\0'; ++j)
55         ans = (powMod(ans, 10, M) * powMod(base, P[j] - '0', M))%M;
56     return ans;
57 }
58
59 // Extended Euclid
60 // a*x + b*y = gcd(a, b)
61 // Given a and b calculate x and y so that a * x + b * y = d (where gcd(a, b) | c)
62 // x_ans = x + (b/d)n
63 // y_ans = y - (a/d)n
64 // where n is an integer
65
66 // Solution only exists if d | c (i.e : c is divisible by d)
67 // gcdExtended(ll a, ll b, ll *x, ll *y) {    // C function for extended Euclidean Algorithm
68     if (a == 0) {    // Base Case
69         *x = 0, *y = 1;
70         return b;
71     }
72     ll x1, y1;    // To store results of recursive call
73     ll gcd = gcdExtended(b%a, a, &x1, &y1);
74     *x = y1 - (b/a) * x1;
75     *y = x1;
76     return gcd;
77 }
78
79 // modInverse(ll a, ll mod) {
80     ll x, y;
81     ll g = gcdExtended(a, mod, &x, &y);

```



```

82 if(g != 1) return -1; // ModInverse doesnt exist
83 ll res = (x%(mod + mod) % mod; // m is added to handle negative x
84 return res;
85 }

```

File: /mnt/Work/notes/MOs.cpp

```

1 // MO's Algo
2 // Complexity : Q*sqrt(N)
3
4 struct query {
5     int l, r, id;
6 };
7
8 const int block = 320; // For 100000
9 query q[MAX];
10 int ans[MAX];
11
12 bool cmp(query &a, query &b) {
13     int block_a = a.l/block, block_b = b.l/block;
14     if(block_a == block_b)
15         return a.r < b.r;
16     return block_a < block_b;
17 }
18
19 bool cmp2(query &a, query &b) { // Faster Comparison function
20     if(a.l/block != b.l/block) return a.l < b.l;
21     if((a.l/block)&1) return a.r < b.r;
22     return a.r > b.r;
23 }
24
25 void add(int x) {} // Add x'th value in range
26 void remove(int x) {} // Remove x'th value from range
27
28 int main() {
29     int Q;
30     scanf("%d", &Q);
31     for(int i = 0; i < Q; ++i) { // Query input
32         scanf("%d%d", &q[i].l, &q[i].r);
33         -q[i].l, -q[i].r, q[i].id = i; // NOTE : value index starts from 0
34     }
35
36     sort(q, q+Q, cmp);
37     int l = 0, r = -1;
38     for(int i = 0; i < Q; ++i) {
39         while(l > q[i].l) add(--l);
40         while(r < q[i].r) add(++r);
41         while(l < q[i].l) remove(l++);
42         while(r > q[i].r) remove(r--);
43         ans[q[i].id] = // Add Constraints
44     }
45     return 0;
46 }

```

File: /mnt/Work/notes/MSTDirected.cpp

```

1 // Directed Minimum Spanning Tree (Edmonds' algorithm)
2 // Complexity : O(E*V) ~ O(E + VlogV) [ works in O(E + VlogV) for almost all cases ]
3 // https://en.wikipedia.org/wiki/Edmonds%27_algorithm
4
5 struct edge {
6     int u, v, w;
7     edge() {}
8     edge(int a, int b, int c) : u(a), v(b), w(c) {}
9 };
10
11 int DMST(vector<edge> &edges, int root, int V) {
12     int ans = 0;
13     int cur_nodes = V;
14     while(1) {
15         vector<int> lo(cur_nodes, INF), pi(cur_nodes, INF); // lo[v] : contains minimum weight to go to node v (for an edge u -> v)
16         // pi[v] : contains the minimum weight edge's starting node u
17         for(int i = 0; i < (int)edges.size(); ++i) {
18             int u = edges[i].u, v = edges[i].v, w = edges[i].w;
19             if(w < lo[v] and u != v)
20                 lo[v] = w, pi[v] = u;
21         }
22
23         lo[root] = 0; // by default the weight to go to root node is 0
24         for(int i = 0; i < (int)lo.size(); ++i) {
25             if(i == root) continue;
26             if(lo[i] == INF) return -1; // if there is no way to visit a node v, then Directed MST doesn't exist

```

```

27     }
28
29     int cur_id = 0;
30     vector<int> id(cur_nodes, -1), mark(cur_nodes, -1);
31
32     for(int i = 0; i < cur_nodes; ++i) {
33         ans += lo[i]; // adding node i's minimum weight to answer
34
35         int u;
36         for(u = i; u != root && id[u] < 0 && mark[u] != i; u = pi[u]) // marking minimum weighted path from root to node i
37             mark[u] = i;
38
39         if(u != root && id[u] < 0) { // Contains cycle, as a result u can not reach to i
40             for(int v = pi[u]; v != u; v = pi[v]) // mark all cycle nodes with id
41                 id[v] = cur_id;
42             id[u] = cur_id++; // ??
43         }}
44
45         if(cur_id == 0) break; // there is no cycle, so all node is possibly visited
46         for(int i = 0; i < cur_nodes; ++i)
47             if(id[i] < 0) id[i] = cur_id++;
48
49         for(int i = 0; i < (int)edges.size(); ++i) {
50             int u = edges[i].u, v = edges[i].v;
51             edges[i].u = id[u];
52             edges[i].v = id[v];
53             if(id[u] != id[v]) edges[i].w -= lo[v];
54         }
55
56         cur_nodes = cur_id;
57         root = id[root];
58     }
59     return ans; // returns total cost of MST
60 }

```

File: /mnt/Work/notes/MSTUndirected.cpp

```

1 // MST Kruskal + Union Find Disjoint Set (DSU)
2 // Complexity of MST : O(E log V)
3
4 // Let a graph be G1, and the MST of the graph is MST1
5 // and a graph G2, where G2 contains same edges as G1 with some new edges
6 // then the new MST of graph G2 will be :
7 // MST2 = MST(of the edges used in M1 (MST of G1) + new added edges)
8
9 set<pair<int, pair<int, int> > > Edge // USED STL SET!!
10
11 int MST(int V) {
12     int mstCost = 0, edge = 0; // If Edge list is STL vector, then sort it!
13     DSU U(V+5);
14     set<pair<int, pair<int, int> > >::iterator it = Edge.begin(); // Contains {Weight, {U, V}}
15
16     for(; it != Edge.end() && edge < V; ++it) {
17         int u = (*it).second.first;
18         int v = (*it).second.second;
19         int w = (*it).first;
20
21         if(!U.isSameSet(u, v))
22             ++edge, mstCost += w, U.makeUnion(u, v);
23     }
24
25     if(edge != V-1) return -1; // Some edge is missing, so no MST found!
26     return mstCost;
27 }
28
29 //Minimum Spanning Tree
30 //Prim's Algorithm
31 //Complexity : O(E log V)
32
33 vector<int> G[MAX], W[MAX];
34 priority_queue<pair<int, int> > pq;
35 bitset<MAX> taken;
36
37 void process(int u) {
38     taken[u] = 1;
39     for(int i = 0; i < (int)G[u].size(); i++) {
40         int v = G[u][i];
41         int w = W[u][i];
42         if(!taken[v])
43             pq.push(make_pair(-w, -v));
44     }
45 }
46
47 int main() {
48     taken.reset();

```

```

49 process(0); //taking 0 node as default
50 int mst_cost = 0;
51 while(!pq.empty()) {
52     w = -pq.top().first;
53     v = -pq.top().second;
54     pq.pop();
55     //if the node is not taken, then use this node
56     //as it contains the minimum edge
57     if(!taken[v])
58         mst_cost += w, process(v);
59 }
60 printf("Prim's MST cost : %d\n", mst_cost);
61 return 0;
62 }

```

File: /mnt/Work/notes/NthPermutation.cpp

```

1 // N'th Permutation
2
3 long long per[30] = {0};
4 long long permute(int freq[]) {
5     int cnt = 0;
6     for(int i = 0; i < 26; ++i)
7         cnt += freq[i];
8     long long permutation = per[cnt] < 1 ? 0 : per[cnt];
9     for(int i = 0; i < 26; ++i)
10         if(freq[i] > 1)
11             permutation /= per[freq[i]];
12     return permutation;
13 }
14
15 string NthPermutation(string str, int n) { // string and numbet of permutation
16     string s;
17     int freq[30] = {0};
18     for(int i = 0; i < (int)str.size(); ++i)
19         freq[str[i] - 'a']++;
20     if(per[0] == 0) { // if not initialized
21         per[0] = 1;
22         for(int i = 1; i <= 25; ++i)
23             per[i] = per[i-1]*i;
24     }
25     if(permute(freq) < n)
26         return s;
27     for(int i = 0; i < (int)str.size(); ++i) {
28         for(int j = 0; j < 26; ++j) {
29             if(freq[j] <= 0) continue;
30             freq[j]--;
31             long long P = permute(freq);
32             if(P >= n) {
33                 s += char(j+'a');
34                 break;
35             }
36             else {
37                 n -= P;
38                 freq[j]++;
39             }
40         }
41     }
42     return s; // Returns empty string if not possible
43 }

```

File: /mnt/Work/notes/PalindromicTree.cpp

```

1 // Palindromic Tree
2
3 struct node {
4     int start, end, length, edge[26], suffixEdg;
5 };
6
7 struct PalinTree {
8     int currNode;
9     string s; // Contains the string
10    int ptr, mxLen;
11    node root1, root2, tree[MAX];
12    PalinTree() {
13        s.clear();
14        root1.length = -1;
15        root1.suffixEdg = 1;
16        root2.length = 0;
17        root2.suffixEdg = 1;
18        tree[1] = root1;
19        tree[2] = root2;
20        ptr = 2;
21        currNode = 1;

```

```

22     mxLen = 0;
23 }
24 void insert(int idx) {
25     int tmp = currNode;
26     while(1) {
27         int curLength = tree[tmp].length;
28         if(idx - curLength >= 1 && s[idx] == s[idx - curLength - 1]) break;
29         tmp = tree[tmp].suffixEdg;
30     }
31     if(tree[tmp].edge[s[idx] - 'a'] != 0) {
32         currNode = tree[tmp].edge[s[idx] - 'a'];
33         return;
34     }
35     ptr++;
36     tree[tmp].edge[s[idx] - 'a'] = ptr;
37     tree[ptr].length = tree[tmp].length + 2;
38     tree[ptr].end = idx;
39     tree[ptr].start = idx - tree[ptr].length + 1;
40     tmp = tree[tmp].suffixEdg;
41     currNode = ptr;
42     if(tree[currNode].length == 1) {
43         tree[currNode].suffixEdg = 2;
44         return;
45     }
46     while(1) {
47         int curLength = tree[tmp].length;
48         if(idx - curLength >= 1 && s[idx] == s[idx - curLength - 1])
49             break;
50         tmp = tree[tmp].suffixEdg;
51     }
52     tree[currNode].suffixEdg = tree[tmp].edge[s[idx] - 'a'];
53 }
54 void buildTree() { // Builds Palindrome Tree of string s
55     for(int i = 0; i < (int)s.length(); ++i)
56         insert(i);
57 }
58 void CalMaxLen() {
59     for(int i = 3; i <= ptr; ++i)
60         mxLen = max(mxLen, tree[i].end - tree[i].start);
61 };
62
63 int main() {
64     PalindromeTree pt;
65     cin >> s;
66     pt.buildTree();
67     cout << "All distinct palindromic substrings for " << s << " : \n";
68     for(int i = 3; i <= ptr; i++) {
69         cout << i - 2 << " ";
70         for(int j = pt.tree[i].start; j <= pt.tree[i].end; j++)
71             cout << s[j];
72         cout << endl;
73 }

```

File: /mnt/Work/notes/PBdatastructure.cpp

```

1 // Policy Based Data Structures
2 // Source : http://codeforces.com/blog/entry/11080
3 // http://codeforces.com/blog/entry/13279
4 // Problems:
5 // http://codeforces.com/blog/entry/53735
6 // http://codeforces.com/contest/762/problem/E
7
8 #include <bits/stdc++.h>
9 #include <ext/pb_ds/assoc_container.hpp> // rb_tree_tag
10 #include <ext/pb_ds/tree_policy.hpp> // tree_order_statistics_node_update
11 #define at(X) find_by_order(X)
12 #define lessThan(x) order_of_key(X)
13 using namespace std;
14 using namespace __gnu_pbds;
15 template<class T> using ordered_set = tree<T, null_type, less_equal<T>, rb_tree_tag, tree_order_statistics_node_update>;
16
17 // key, Mapped-Policy, Key Comparison Func, Underlying data Structure, Policy for updating node interval
18
19 // Key Comparison Func : Defines how data will be stored (increasing, decreasing order)
20 // less<int> - Same value occurs once & increasing SET
21 // less_equal<int> - Same value occurs one or more & increasing MULTiset
22 // greater<int>, greater_equal<int>
23 //
24 // if Mapped-Policy set to null_type, this works as a SET
25 // else works as MAP
26 //
27 // Underlying Data Structures : rb_tree_tag - Red Black Tree
28 // splay_tree_tag - Splay Tree
29 // ov_tree_tag - Ordered Vector Tree
30 //

```

```

31 // Policy for Updaing Node : default - null_node_update
32 //      c++ implemented - tree_order_statistics_node_update
33
34 // Features :
35 // Can be used as SET/MULTISET
36 // lower_bound and upper bound works as expected
37 // insertion, deletion, clear
38 // container.find_by_order(x) returns x'th elements iterator
39 // container.order_of_key(x) returns number of values less than (or equal to) x
40 // auto casting works
41 //
42
43 int main() {
44     ordered_set<int> X;
45     // ----- INSERTION -----
46     // Data are indexed in increasing order & can occur only once (like STL SET)
47     X.insert(1); X.insert(18); X.insert(2); X.insert(2); X.insert(4); X.insert(8); X.insert(16);
48
49     // ----- ITERATION -----
50     // Index-Wise : log_n
51     cout << *X.find_by_order(0) << endl;
52     cout << *X.find_by_order(2) << endl;
53     cout << *X.find_by_order(6) << endl; // Not Present, Will return 0
54
55     if(X.end() == X.find_by_order(6)) cout << "End Reached" << endl;
56     X.erase(X.find_by_order(2)); // Deleting element by iterator
57     cout << "Iterating \n";
58     for(auto x : X) cout << x << endl;
59     cout << endl;
60
61     // ----- Range Search -----
62     // Returns number of elements STRICTLY less than value
63     cout << X.order_of_key(-5) << endl;
64     cout << X.order_of_key(1) << endl;
65     cout << X.order_of_key(3) << endl;
66     cout << X.order_of_key(4) << endl;
67     cout << X.order_of_key(400) << endl;
68     X.clear(); // Clearing
69 }

```

File: /mnt/Work/notes/PeresistantSegmentTree.cpp

```

1 // Persistant/Dynamic Segment Tree
2 // Pointer Version
3
4 struct node {
5     ll val;
6     node *lft, *rht;
7     node(node *L = NULL, node *R = NULL, ll v = 0) {
8         lft = L;
9         rht = R;
10        val = v;
11    };
12
13 node *persis[101000], *null = new node();
14 // null->lft = null->rht = null;
15
16 node *nCopy(node *x) {
17     node *tmp = new node();
18     if(x) {
19         tmp->val = x->val;
20         tmp->lft = x->lft;
21         tmp->rht = x->rht;
22     }
23     return tmp;
24 }
25
26
27 void init(node *pos, ll l, ll r) {
28     if(l == r) {
29         pos->val = val[l];
30         pos->lft = pos->rht = null;
31         return;
32     }
33     ll mid = (l+r)>>1;
34     pos->lft = new node();
35     pos->rht = new node();
36
37     init(pos->lft, l, mid);
38     init(pos->rht, mid+1, r);
39     pos->val = pos->lft->val + pos->rht->val;
40 }
41
42 // Single Position update
43 void update(node *pos, ll l, ll r, ll idx, ll val) {

```

```

44     if(l == r) {
45         pos->val += val;
46         pos->lft = pos->rht = null;
47         return;
48     }
49
50     ll mid = (l+r)>>1;
51     if(idx <= mid) {
52         pos->lft = nCopy(pos->lft);
53         if(!pos->rht)
54             pos->rht = null;
55         update(pos->lft, l, mid, idx, val);
56     }
57     else {
58         pos->rht = nCopy(pos->rht);
59         if(!pos->lft)
60             pos->lft = null;
61         update(pos->rht, mid+1, r, idx, val);
62     }
63     pos->val = 0;
64     if(pos->lft) pos->val += pos->lft->val;
65     if(pos->rht) pos->val += pos->rht->val;
66 }
67
68 // Range [L, R] update
69 void update(node *pos, ll l, ll r, ll L, ll R, ll val) {
70     if(r < L || R < l) return;
71     if(L <= l && r <= R) {
72         pos->prop += val;
73         pos->val += (r-l+1)*val;
74         return;
75     }
76     ll mid = (l+r)>>1;
77     pos->lft = nCopy(pos->lft); // Can be reduced
78     pos->rht = nCopy(pos->rht);
79     update(pos->lft, l, mid, L, R, val);
80     update(pos->rht, mid+1, r, L, R, val);
81     pos->val = pos->lft->val + pos->rht->val + (r-l+1)*pos->prop;
82 }
83
84 // Range [L, R] Sum Query
85 ll query(node *pos, ll l, ll r, ll L, ll R) {
86     if(r < L || R < l || pos == NULL) return 0;
87     if(L <= l && r <= R) return pos->val;
88     ll mid = (l+r)/2LL;
89     ll x = query(pos->lft, l, mid, L, R);
90     ll y = query(pos->rht, mid+1, r, L, R);
91     return x+y;
92 }
93
94 // Ignore LMax persistent tree positions query for finding k'th value
95 int query(node *RMax, node *LMax, int l, int r, int k) { // (LMax : past, RMax : updated)
96     if(l == r) return l;
97
98     // NO NEED THIS SECTOR STILL AC --
99     RMax->lft = nCopy(RMax->lft);
100    LMax->lft = nCopy(LMax->lft);
101    RMax->rht = nCopy(RMax->rht);
102    LMax->rht = nCopy(LMax->rht);
103    //-----
104    // for each range [l, r] we will ignore every [1, l-1] range numbers
105    int Count = RMax->lft->val - LMax->lft->val;
106    int mid = (l+r)>>1;
107    // if there exists more than or equal to k values in left range, then we'll find kth number in left segment
108    if(Count >= k) return query(RMax->lft, LMax->lft, l, mid, k);
109    else return query(RMax->rht, LMax->rht, mid+1, r, k-Count);
110 }
111
112 // ----- Propagation -----
113 bool flipProp(bool par, bool child) {
114     if(par == child) return 0;
115     return 1;
116 }
117
118 void propagate(node *pos, ll l, ll r) { // Propagation Func
119     if(l == r) return;
120     pos->lft = nCopy(pos->lft); // No need to copy in update/query function
121     pos->rht = nCopy(pos->rht);
122     if(!pos->flip) return;
123     ll mid = (l+r)>>1;
124     pos->lft->flip = flipProp(pos->flip, pos->lft->flip);
125     pos->rht->flip = flipProp(pos->flip, pos->rht->flip);
126     pos->lft->val = (mid-l+1)-pos->lft->val;
127     pos->rht->val = (r-mid)-pos->rht->val;
128     pos->flip = 0;
129 }
130
131

```

```

132
133 // Flip in range [L, R]
134 void updateFlip(node *pos, ll l, ll r, ll L, ll R) {
135     if(r < L || R < l) return;
136     propagate(pos, l, r);
137     if(L <= l && r <= R) {
138         pos->flip = 1;
139         pos->val = (r-l+1) - pos->val;
140         return;
141     }
142     ll mid = (l+r)>>1;
143     updateFlip(pos->lft, l, mid, L, R);
144     updateFlip(pos->rht, mid+1, r, L, R);
145     pos->val = 0;
146     if(pos->rht) pos->val += pos->rht->val;
147     if(pos->lft) pos->val += pos->lft->val;
148 }
149
150
151 // Erasing A segment tree, pos = root, must run for each root
152 void ClearTree(node *pos) {
153     if(pos == NULL) {
154         delete pos;
155         return;
156     }
157     ClearTree(pos->lft);
158     ClearTree(pos->rht);
159     delete pos;
160 }
161
162 int main() {
163     // MUST BE INITIALIZED
164     null->lft = null->rht = null;
165     //
166     for(int i = 1; i <= 10; ++i) {
167         persis[i] = nCopy.persis[i-1];
168         update(persis[i], 1, n, idx, val);
169     }
170     return 0;
171 }

```

File: /mnt/Work/notes/Primes.cpp

```

1 // Limit ----- No. of Primes
2 // 100          25
3 // 1000         168
4 // 10,000       1229
5 // 100,000      9592
6 // 1,000,000    78498
7 // 10,000,000   664579
8
9 bitset<1000000> isPrime;
10 vector<long long> primes;
11
12 void sieve(unsigned long long N) {
13     isPrime.set();
14     isPrime[0] = isPrime[1] = 0;
15     unsigned long long lim = sqrt(N) + 5;
16     for(unsigned long long i = 2; i <= lim; i++) { // change lim to N, if all primes in range N is needed
17         if(isPrime[i])
18             for(unsigned long long j = i*i; j <= N; j+= i)
19                 isPrime[j] = 0;
20     }}
21
22 void sieveGen(unsigned long long N) {
23     isPrime.set();
24     isPrime[0] = isPrime[1] = 0;
25     for(unsigned long long i = 2; i <= N; i++) { //Note, N isn't square rooted!
26         if(isPrime[i]) {
27             for(unsigned long long j = i*i; j <= N; j+= i)
28                 isPrime[j] = 0;
29             primes.push_back(i);
30         }}}
31
32 vector<pair<ull, ull> > primeFactor(ull n) {
33     vector<pair<ull, ull> > factor;
34     for(long long i = 0; i < (int)primes.size() && primes[i] <= n; i++) {
35         bool first = 1;
36         while(n%primes[i] == 0) {
37             if(first) {
38                 factor.push_back({primes[i], 0});
39                 first = 0;
40             }
41             factor.back().second++;
42             n/=primes[i];
43         }}

```

```

44 return factor;
45 }
46
47 int pd[MAX]; // Contains minimum prime factor/divisor, for primes pd[x] = x
48 vector<int> primes; // Contains prime numbers
49 void SieveLinear(int N) {
50     for(int i = 2; i <= N; ++i) {
51         if(pd[i] == 0) pd[i] = i; primes.push_back(i); // if pd[i] == 0, then i is prime
52         for(int j = 0; j < (int)primes.size() && primes[j] <= pd[i] && i*primes[j] <= N; ++j)
53             pd[i*primes[j]] = primes[j];
54     }
55 }
56
57 int pd[MAX]; // Contains minimum prime factor/divisor, for primes pd[x] = x
58 vector<int> primes; // Contains prime numbers
59 vector<int> PF[MAX];
60 void SieveLinearRangePF(int N, ll low, ll hi) { // also returns unique prime factors in range [low, hi]
61     for(int i = 2; i <= N; ++i) {
62         if(pd[i] == 0) {
63             pd[i] = i; primes.push_back(i); // if pd[i] == 0, then i is prime
64             for(ll x = (low-1)*(low-1)%i+i; x <= hi; x += i) // inserting all prime factors [prime will be inserted only once]
65                 PF[x-low].push_back(i); // just to be sure, used low-1, instead of low
66         }
67         for(int j = 0; j < (int)primes.size() && primes[j] <= pd[i] && i*primes[j] <= N; ++j)
68             pd[i*primes[j]] = primes[j];
69     }
70 }
71
72 vector<ll> Divisors(ll n) { // Returns the divisors
73     ll lim = sqrt(n);
74     vector<ll> divisor;
75     for(ll i = 2; i <= lim; i++) { // deal with 1 and n manually
76         if(n % i == 0) {
77             ll tmp = n/i;
78             divisor.push_back(tmp);
79             if(i != tmp)
80                 divisor.push_back(i);
81         }
82     }
83     return divisor;
84 }
85
86 vector<pair<long long, long long>> factorialFactorization(long long n) { // prime factorization of factorials (n!)
87     vector<pair<long long, long long>> V;
88     for(long long i = 0; i < (int)primes.size() && primes[i] <= n; i++) {
89         long long tmp = n, power = 0;
90         while(tmp % primes[i] == 0) {
91             power += tmp / primes[i];
92             tmp /= primes[i];
93         }
94         if(power != 0) V.push_back(make_pair(primes[i], power));
95     }
96     return V;
97 }
98
99 long long numPF(long long n) { //returns number of prime factors
100     long long num = 0;
101     for(long long i = 0; primes[i] * primes[i] <= n; i++) {
102         while(n % primes[i] == 0) {
103             n /= primes[i];
104             num++;
105         }
106     }
107     if(n > 1) num++; //there might left a prime number which is bigger than primes[i]
108     return num;
109 }
110
111 long long numDIFPF(long long n) { // returns number of different prime factors
112     long long diff_num = 0;
113     for(long long i = 0; primes[i] * primes[i] <= n; i++) {
114         bool ok = 0;
115         while(n % primes[i] == 0) {
116             n /= primes[i];
117             ok = 1;
118         }
119         if(ok) diff_num++;
120     }
121     if(n > 1) diff_num++;
122     return diff_num;
123 }
124
125 unsigned long long sumPF(long long n) { // returns sum of prime factors
126     unsigned long long sum = 0;
127     for(long long i = 0; primes[i] * primes[i] <= n; i++) {
128         while(n % primes[i] == 0) {
129             n /= primes[i];
130             sum += primes[i];
131         }
132     }
133     if(n > 1) sum += n;
134     return sum;
135 }

```



```

132 int NumberOfDivisors(long long n) { // if n = p1^a1 * p2^a2,... then NOD = (a1+1)*(a2+1)*...
133 if(n <= MAX and isPrime(n)) return 2;
134 int NOD = 1;
135 for(int i = 0; a = 0; i < (int)primes.size() and primes[i] <= n; ++i, a = 0) {
136     while(n % primes[i] == 0)
137         ++a, n /= primes[i];
138     NOD *= (a+1);
139 }
140 if(n != 1) NOD *= 2;
141 return NOD;
142 }
143
144 //-----Fast Factorization using Sieve-Like algorithm-----
145 bitset<MAX> isPrime;
146 int divisor[MAX];
147
148 void sieve(long long lim) { // Prime numbers for the limit should be sieved, otherwise WA
149     isPrime.set();
150     isPrime[0] = isPrime[1] = 0;
151     for(int i = 0; i <= lim; ++i) {
152         if(isPrime[i]) {
153             for(long long j = i; j <= lim; j += i) {
154                 isPrime[j] = 0;
155                 divisor[j] = i;
156             }
157         }
158     }
159 }
160 vector<int> factorize(long long x) { // This function only iterates over the prime numbers
161     int pastDiv = 0; // 0 : no divisor is present
162     vector<int> factor;
163     while(x > 1) {
164         if(divisor[x] != 0) {
165             factor.push_back(divisor[x]);
166             x /= divisor[x]; // now x would be reduced by factor of divisor[x]
167         }
168     }
169     return factor;
170 }
171 // Prime Probability
172 // Algorithm : Miller-Rabin primality test Complexity : k * (log n)^3
173 // This function is called for all k trials. It returns false if n is composite and returns true if n is probably prime.
174 // d is an odd number such that d*(2^r) = n-1 for some r >= 1
175 bool millerTest(int d, int n) {
176     int a = 2 + rand() % (n - 4); // Pick a random number in [2..n-2].
177     int x = Pow(a, d, n); // Compute a^d % n
178     if(x == 1 || x == n-1)
179         return 1;
180     while(d != n-1) {
181         x = (x * x) % n; // (i) d does not reach n-1
182         d *= 2; // (ii) (x^2) % n is not 1
183         if(x == 1) return 0; // (iii) (x^2) % n is not n-1
184         if(x == n-1) return 1;
185     }
186     return 0; // Return composite
187 }
188
189 bool isPrime(int n, int k = 10) { // Higher value of k gives more accuracy (Use k >= 9)
190     if(n <= 1 || n == 4) return 0; // Corner cases
191     if(n <= 3) return 1;
192     int d = n - 1; // Find r such that n = 2^d * r + 1 for some r >= 1
193     while(d % 2 == 0) d /= 2;
194     for(int i = 0; i < k; ++i) // Iterate given nber of 'k' times
195         if(millerTest(d, n) == 0)
196             return 0;
197     return 1;
198 }

```

File: /mnt/Work/notes/Searching.cpp

```

1 // Binary Search
2 // Complexity : O(n Log n)
3
4 // UpperBound(// lo, // hi, // key) { // Returns lowest position where v[i] > key
5 //     int mid, ans = -1; // 10 10 10 20 20 20 30 30
6 //     while(lo <= hi) { //
7 //         mid = (lo + hi) >> 1;
8 //         if(key >= v[mid]) ans = mid, lo = mid + 1;
9 //         else hi = mid - 1;
10 //     }
11 //     return ans+1; // Tweaking this line will return the last position of key
12 // }
13
14 // LowerBound(// lo, // hi, // key) { // Returns lowest position where v[i] == key (if value is present more than once)
15 //     int mid, ans = -1; // 10 10 10 20 20 20 30 30

```

```

16 while(lo <= hi) { // ^
17     mid = (lo+hi)>>1;
18     if(key <= v[mid]) ans = mid; hi = mid - 1;
19     else lo = mid + 1;
20 }
21 return ans;
22 }
23
24 // lo : lower value, hi : upper value, est : estimated output of the required result, delta : number of iteration in search
25 double bisection(double lo, double hi, double est, int delta) {
26     double mid, ans = -1;
27     for(int i = 0; i < delta; ++i) {
28         mid = (lo+hi)/2.0;
29         if(Equal(TestFunction(mid), est)) ans = mid; lo = mid;
30         else if(Greater(TestFunction(mid), est)) hi = mid;
31         else lo = mid;
32     }
33     return ans;
34 }
35
36 // Full Functional Ternary Search
37 /* EMAXX ::
38 If f(x) takes integer parameter, the interval [l r] becomes discrete.
39 Since we did not impose any restrictions on the choice of points m1 and m2, the correctness of the algorithm is not affected.
40 m1 and m2 can still be chosen to divide [l r] into 3 approximately equal parts.
41
42 The difference occurs in the stopping criterion of the algorithm.
43 Ternary search will have to stop when (r-l) < 3, because in that case we can no longer select m1 and m2 to
44 be different from each other as well as from ll and rr, and this can cause infinite iterating.
45 Once (r-l) < 3, the remaining pool of candidate points (l,l+1,...,r) needs to be checked
46 to find the point which produces the maximum value f(x).
47 */
48
49 ll ternarySearch(ll low, ll high) {
50     ll ret = -INF;
51     while((high - low) > 2) {
52         ll mid1 = low + (high - low) / 3;
53         ll mid2 = high - (high - low) / 3;
54         ll cost1 = f(mid1);
55         ll cost2 = f(mid2);
56         if(cost1 < cost2) {
57             low = mid1;
58             ret = max(cost2, ret);
59         }
60         else {
61             high = mid2;
62             ret = max(cost1, ret);
63         }
64     }
65     for(int i = low; i <= high; ++i)
66         ret = max(ret, f(i));
67     return ret;
68 }

```

File: /mnt/Work/notes/SegmentTree.cpp

```

1 // Segment Tree
2
3 // Only Supports Range Value SET (NOT UPDATE) and Point Query
4 struct SegTreeSetVal {
5     vector<int> tree;
6     vector<bool> prop;
7
8     void Resize(int n) {
9         tree.resize(n*5);
10        prop.resize(n*5);
11    }
12
13    void propagate(int pos, int l, int r) {
14        if(!prop[pos] || l == r) return;
15        tree[pos<<1] = tree[pos<<1+1] = tree[pos];
16        prop[pos<<1] = prop[pos<<1+1] = 1;
17        prop[pos] = 0;
18    }
19
20    void SetVal(int pos, int l, int r, int L, int R, int val) { // Set value val in range [L, R]
21        if(r < L || R < l) return;
22        propagate(pos, l, r);
23        if(L <= l && r <= R) {
24            tree[pos] = val;
25            prop[pos] = 1;
26            return;
27        }
28        int mid = (l+r)>>1;
29        SetVal(pos<<1, l, mid, L, R, val);
30        SetVal(pos<<1+1, mid+1, r, L, R, val);
31    }
32 }

```

```

31 }
32
33 int query(int pos, int l, int r, int idx) { // Can be modified to range query
34     if(l == r) return tree[pos];
35     propagate(pos, l, r);
36     int mid = (l+r)>>1;
37     if(idx <= mid) return query(pos<<1, l, mid, idx);
38     else return query(pos<<1|1, mid+1, r, idx);
39 };
40
41
42 // Segment Tree Range Sum : Lazy with Propagation (MOD used)
43 struct SegTreeRSQ {
44     vector<ll> sum, prop;
45
46     void Resize(int n) {
47         sum.resize(5*n);
48         prop.resize(5*n);
49     }
50
51     void init(int pos, int l, int r, ll val[]) {
52         sum[pos] = prop[pos] = 0;
53         if(l == r) {
54             sum[pos] = val[l]%MOD;
55             return;
56         }
57         int mid = (l+r)>>1;
58         init(pos<<1, l, mid, val);
59         init(pos<<1|1, mid+1, r, val);
60         sum[pos] = (sum[pos<<1] + sum[pos<<1|1])%MOD;
61     }
62
63     void propagate(int pos, int l, int r) {
64         if(prop[pos] == 0 || l == r) return;
65         int mid = (l+r)>>1;
66
67         sum[pos<<1] = (sum[pos<<1] + prop[pos]*(mid-l+1))%MOD;
68         sum[pos<<1|1] = (sum[pos<<1|1] + prop[pos]*(r-mid))%MOD;
69         prop[pos<<1] = (prop[pos<<1] + prop[pos])%MOD;
70         prop[pos<<1|1] = (prop[pos<<1|1] + prop[pos])%MOD;
71         prop[pos] = 0;
72     }
73
74     void update(int pos, int l, int r, int L, int R, ll val) {
75         if(r < L || R < l) return;
76         propagate(pos, l, r);
77         if(L <= l && r <= R) {
78             sum[pos] = (sum[pos] + val*(r-l+1))%MOD;
79             prop[pos] = (prop[pos] + val)%MOD;
80             return;
81         }
82
83         int mid = (l+r)>>1;
84         update(pos<<1, l, mid, L, R, val);
85         update(pos<<1|1, mid+1, r, L, R, val);
86         sum[pos] = (sum[pos<<1] + sum[pos<<1|1])%MOD;
87     }
88
89     ll query(int pos, int l, int r, int L, int R) {
90         if(r < L || R < l) return 0;
91         propagate(pos, l, r);
92         if(L <= l && r <= R) return sum[pos];
93         int mid = (l+r)>>1;
94         return (query(pos<<1, l, mid, L, R) + query(pos<<1|1, mid+1, r, L, R))%MOD;
95     };
96
97
98 // Dynamic Range Segment Tree (values can be deleted from right)
99 // Resize the Segment Tree with the maximum length of value
100 // Segment Tree Range Sum, Range Update, And Single point Value Change (If the last value was deleted)
101 // http://codeforces.com/contest/283/problem/A
102
103 // Range Sum with Carry Value
104 struct SegSum {
105     int tree[MAX*4], carry[MAX*4];
106
107     void init() {
108         memset(tree, 0, sizeof tree);
109         memset(carry, 0, sizeof carry);
110     }
111
112     void Update(int pos, int l, int r, int x, int y, ll val) { // Update value at range/point
113         if(y < l || x > r) return;
114         if(x <= l && r <= y) {
115             tree[pos] += (r-l+1)*val;
116             carry[pos] += val;
117             return;
118         }

```

```

119     int mid = (l+r)>>1;
120     Update(pos<<1, l, mid, x, y, val);
121     Update(pos<<1|1, mid+1, r, x, y, val);
122     tree[pos] = tree[pos<<1] + tree[pos<<1|1] + (r-l+1)*carry[pos];
123 }
124
125 // Read(int pos, int l, int r, int x, int y, ll Carry = 0) { // Read value at range/point
126     if(y < l || x > r) return 0;
127     if(x <= l && r <= y) return tree[pos] + Carry * (r-l+1);
128     int mid = (l+r)>>1;
129     ll lft = Read(pos<<1, l, mid, x, y, Carry + carry[pos]);
130     ll rht = Read(pos<<1|1, mid+1, r, x, y, Carry + carry[pos]);
131     return lft + rht;
132 }
133
134 // Sets value at idx
135 void Set(int pos, int l, int r, int idx, ll val, ll Carry = 0) {
136     if(l == r) {
137         tree[pos] = val + (-1*Carry); // Extra carry values are eliminated in such way
138         carry[pos] = 0; // that the subtraction is always the new value
139         return;
140     }
141     int mid = (l+r)>>1;
142     if(idx <= mid) Set(pos<<1, l, mid, idx, val, Carry + carry[pos]);
143     else Set(pos<<1|1, mid+1, r, idx, val, Carry + carry[pos]);
144     tree[pos] = tree[pos<<1] + tree[pos<<1|1] + (r-l+1) * carry[pos];
145 };
146
147 // SegTree with Lazy Propagation (Flip Count in Range)
148 // Prop :
149 // 0 : No prop operation
150 // 1 : Prop operation should be done
151
152 struct SegProp {
153     struct Node { int val, prop };
154
155     vector<Node> tree;
156     void init(int L, int R, int pos, ll val[]) {
157         if(L == R) {
158             tree[pos].val = 0;
159             tree[pos].prop = 0;
160             return;
161         }
162         int mid = (L+R)>>1;
163         init(L, mid, pos<<1, val);
164         init(mid+1, R, pos<<1|1, val);
165         tree[pos].val = tree[pos].prop = 0;
166     }
167
168     int flipProp(int parentVal, int childVal) {
169         if(parentVal == childVal) return 0;
170         return parentVal;
171     }
172 }
173
174 void propagate(int L, int R, int pos) {
175     if(tree[pos].prop == 0 || L == R) // If no propagation tag
176         return; // or leaf node, then no need to change
177     int mid = (L+R)>>1;
178     tree[pos<<1].val = (mid-L+1) - tree[pos<<1].val; // Set left & right child value
179     tree[pos<<1|1].val = (R-mid) - tree[pos<<1|1].val;
180     tree[pos<<1].prop = flipProp(tree[pos].prop, tree[pos<<1].prop); // Flip child prop according to problem
181     tree[pos<<1|1].prop = flipProp(tree[pos].prop, tree[pos<<1|1].prop);
182     tree[pos].prop = 0; // Clear parent propagation tag
183 }
184
185 void update(int L, int R, int l, int r, int pos) {
186     if(r < L || R < l) return;
187     propagate(L, R, pos);
188     if(l <= L && R <= r) {
189         tree[pos].val = (R-L+1) - tree[pos].val; // Value updated
190         tree[pos].prop = 1; // Propagation tag set
191         return;
192     }
193     int mid = (L+R)>>1;
194     update(L, mid, l, r, pos<<1);
195     update(mid+1, R, l, r, pos<<1|1);
196     tree[pos].val = tree[pos<<1].val + tree[pos<<1|1].val;
197 }
198
199 int querySum(int L, int R, int l, int r, int pos) {
200     if(r < L || R < l) return 0;
201     propagate(L, R, pos);
202     if(l <= L && R <= r) return tree[pos].val;
203     int mid = (L+R)>>1;
204     int lft = querySum(L, mid, l, r, pos<<1);
205     int rht = querySum(mid+1, R, l, r, pos<<1|1);
206     return lft + rht;

```

```

207 }};
208
209 // ----- Segment Tree Range Maximum Sum -----
210 struct SegTreeRMS {
211     struct node {
212         ll sum, prefix, suffix, ans;
213
214         node(ll val = 0) {
215             sum = prefix = suffix = ans = val;
216         }
217
218         void merge(node left, node right) {
219             sum = left.sum + right.sum;
220             prefix = max(left.prefix, left.sum + right.prefix);
221             suffix = max(right.suffix, right.sum + left.suffix);
222             ans = max(left.ans, max(right.ans, left.suffix + right.prefix));
223         }
224
225         vector<node> tree;
226         void init(int pos, int l, int r, ll val[]) {
227             if(l == r) {
228                 tree[pos] = node(val[l]);
229                 return;
230             }
231             int mid = (l+r)/2;
232             init(pos*2, l, mid, val);
233             init(pos*2+1, mid+1, r, val);
234             tree[pos] = node(-INF);
235             tree[pos].merge(tree[pos*2], tree[pos*2+1]);
236         }
237
238         void update(int pos, int l, int r, int x, int val) {
239             if(x < l || r < x) return;
240             if(l == r && l == x) {
241                 tree[pos] = node(val);
242                 return;
243             }
244             int mid = (l+r)/2;
245             update(pos*2, l, mid, x, val);
246             update(pos*2+1, mid+1, r, x, val);
247             tree[pos] = node(-INF);
248             tree[pos].merge(tree[pos*2], tree[pos*2+1]);
249         }
250
251         node query(int pos, int l, int r, int x, int y) {
252             if(r < x || y < l) return node(-INF);
253             if(x <= l && r <= y) return tree[pos];
254             int mid = (l+r)/2;
255             node lft = query(pos*2, l, mid, x, y);
256             node rht = query(pos*2+1, mid+1, r, x, y);
257             node parent = node(-INF);
258             parent.merge(lft, rht);
259             return parent;
260         }
261
262         // Segment Tree Insert/Remove value, Find l'th Value
263         struct SegTreeInsertRemove { // Finds/Deletes l'th value from array/SegTree
264             int tree[4];
265             void init(int pos, int L, int R) {
266                 if(L == R) {
267                     tree[pos] = 1;
268                     return;
269                 }
270                 int mid = (L+R)>>1;
271                 init(pos<<1, L, mid);
272                 init(pos<<1|1, mid+1, R);
273                 tree[pos] = tree[pos<<1] + tree[pos<<1|1];
274             }
275             int SearchVal(int pos, int L, int R, int l, bool removeVal = 0) { // Find l'th value in Segment Tree, removes it if removeVal = 1
276                 if(L == R) {
277                     tree[pos] = (removeVal ? 0 : 1);
278                     return L;
279                 }
280                 int mid = (L+R)>>1;
281                 if(l <= tree[pos<<1]) {
282                     int idx = SearchVal(pos<<1, L, mid, l, removeVal);
283                     if(removeVal) tree[pos] = tree[pos<<1] + tree[pos<<1|1];
284                     return idx;
285                 }
286                 else {
287                     int idx = SearchVal(pos<<1|1, mid+1, R, l - tree[pos<<1], removeVal);
288                     if(removeVal) tree[pos] = tree[pos<<1] + tree[pos<<1|1];
289                     return idx;
290                 }
291             }
292         }
293         // Segment Tree Range Bit flip, set, reset and Query
294         // propataion tags:
295         // 0 - no change, 1 - all set to one, 2 - all set to zero, 3 - all need to be flipped

```

```

295 struct RangeBitQuery {
296     vector<pair<int, int> > tree;
297     RangeBitQuery() { tree.resize(MAX*4); }
298
299 void init(int pos, int L, int R, string &s) {
300     tree[pos].second = 0;
301     if(L == R) {
302         tree[pos].first = (s[L] == '1');
303         return;
304     }
305     int mid = (L+R)>>1;
306     init(pos<<1, L, mid, s);
307     init(pos<<1|1, mid+1, R, s);
308     tree[pos].first = tree[pos<<1].first + tree[pos<<1|1].first;
309 }
310
311 int Convert(int tag) { // This function generates output tag of child node if the parent node is set to 3 (fipped)
312     if(tag == 1) return 2;
313     if(tag == 2) return 1;
314     if(tag == 3) return 0;
315     return 3;
316 }
317
318 // On every layer of update or query, this Propagation func should be called to pre-process previous left off operations
319 void Propagate(int L, int R, int parent) { // Propagate parent node to child nodes (left and right)
320     if(tree[parent].second == 0) return; // and sets parent node's propagation tag to 0
321     int mid = (L+R)>>1;
322     int lft = parent<<1, rht = parent<<1|1;
323     if(tree[parent].second == 1) {
324         tree[lft].first = mid-L+1;
325         tree[rht].first = R-mid;
326     }
327     else if(tree[parent].second == 2)
328         tree[lft].first = tree[rht].first = 0;
329     else if(tree[parent].second == 3) {
330         tree[lft].first = (mid-L+1) - tree[lft].first;
331         tree[rht].first = (R-mid) - tree[rht].first;
332     }
333     if(L != R) { // If the child nodes also contain propagate tag (and the childs are not leaf node)
334         if(tree[parent].second == 1 || tree[parent].second == 2)
335             tree[lft].second = tree[rht].second = tree[parent].second;
336         else {
337             tree[lft].second = Convert(tree[lft].second);
338             tree[rht].second = Convert(tree[rht].second);
339         }
340     }
341     tree[parent].second = 0; // Parent node's prop tag set to zero
342     if(L != R) tree[parent].first = tree[lft].first + tree[rht].first; // If this is not the leaf node, calculate child node's sum
343 }
344
345 void updateOn(int pos, int L, int R, int l, int r) { // Turn on bits in range [l, r]
346     if(r < L || R < l || L > R) return;
347     Propagate(L, R, pos);
348     if(l <= L && R <= r) {
349         tree[pos].first = (R-L+1);
350         tree[pos].second = 1;
351         return;
352     }
353     int mid = (L+R)>>1;
354     updateOn(pos<<1, L, mid, l, r);
355     updateOn(pos<<1|1, mid+1, R, l, r);
356     tree[pos].first = tree[pos<<1].first + tree[pos<<1|1].first;
357 }
358
359 void updateOff(int pos, int L, int R, int l, int r) { // Turn off bits in range [l, r]
360     if(r < L || R < l || L > R) return;
361     Propagate(L, R, pos);
362     if(l <= L && R <= r) {
363         tree[pos].first = 0;
364         tree[pos].second = 2;
365         return;
366     }
367     int mid = (L+R)>>1;
368     updateOff(pos<<1, L, mid, l, r);
369     updateOff(pos<<1|1, mid+1, R, l, r);
370     tree[pos].first = tree[pos<<1].first + tree[pos<<1|1].first;
371 }
372
373 void updateFlip(int pos, int L, int R, int l, int r) { // Flip bits in range [l, r]
374     if(r < L || R < l || L > R) return;
375     Propagate(L, R, pos);
376     if(l <= L && R <= r) {
377         tree[pos].first = abs(R-L+1 - tree[pos].first);
378         tree[pos].second = 3;
379         return;
380     }
381     int mid = (L+R)>>1;
382     updateFlip(pos<<1, L, mid, l, r);
383     updateFlip(pos<<1|1, mid+1, R, l, r);

```

```

383     tree[pos].first = tree[pos<<1].first + tree[pos<<1|1].first;
384 }
385
386 int querySum(int pos, int L, int R, int l, int r) {          // Returns number of set bit in range [l, r]
387     if(r < L || R < l || L > R) return 0;
388     Propagate(L, R, pos);
389     if(l <= L && R <= r) return tree[pos].first;
390     int mid = (L+R)>>1;
391     return querySum(pos<<1, L, mid, l, r) + querySum(pos<<1|1, mid+1, R, l, r);
392 };
393
394 // Merge Sort Tree
395 struct MergeSortTree {
396     vector<int> tree[MAX*4];
397
398     void init(int pos, int l, int r, ll val[]) {
399         tree[pos].clear();          // Clears past values
400         if(l == r) {
401             tree[pos].push_back(val[l]);
402             return;
403         }
404
405         int mid = (l+r)>>1;
406         init(pos<<1, l, mid, val);
407         init(pos<<1|1, mid+1, r, val);
408         merge(tree[pos<<1].begin(), tree[pos<<1].end(), tree[pos<<1|1].begin(), tree[pos<<1|1].end(), back_inserter(tree[pos]));
409     }
410
411     int query(int pos, int l, int r, int L, int R, int k) {
412         if(r < L || R < l) return 0;
413         if(L <= l && r <= R)
414             return (int)tree[pos].size() - (upper_bound(tree[pos].begin(), tree[pos].end(), k) - tree[pos].begin());    // MODIFY
415         int mid = (l+r)>>1;
416         return query(pos<<1, l, mid, L, R, k) + query(pos<<1|1, mid+1, r, L, R, k);
417     };
418
419
420 // Segment Tree Sequence (Lazy Propagation):: Contains sequence A + 2A + 3A + ..... nA
421 struct SegTreeSeq {
422     vector<ll> sum, prop;
423
424     void Resize(int n) {
425         sum.resize(n*5);
426         prop.resize(n*5);
427     }
428
429     ll intervalSum(ll l, ll r, ll val) {
430         ll interval = (r*(r+1))/2LL - (l*(l-1))/2LL;
431         return (interval*val+MOD)%MOD;
432     }
433
434     void propagate(int pos, int l, int r) {
435         if(prop[pos] == 0 || l == r) return;
436         int mid = (l+r)>>1;
437
438         sum[pos<<1] = (sum[pos<<1] + intervalSum(l, mid, prop[pos]))%MOD;
439         sum[pos<<1|1] = (sum[pos<<1|1] + intervalSum(mid+1, r, prop[pos]))%MOD;
440         prop[pos<<1] = (prop[pos<<1] + prop[pos])%MOD;
441         prop[pos<<1|1] = (prop[pos<<1|1] + prop[pos])%MOD;
442         prop[pos] = 0;
443     }
444
445     void init(int pos, int l, int r, ll val[]) {
446         sum[pos] = prop[pos] = 0;
447         if(l == r) {
448             sum[pos] = (val[l]*l)%MOD;
449             return;
450         }
451         int mid = (l+r)>>1;
452         init(pos<<1, l, mid, val);
453         init(pos<<1|1, mid+1, r, val);
454         sum[pos] = (sum[pos<<1] + sum[pos<<1|1])%MOD;
455     }
456
457     void update(int pos, int l, int r, int L, int R, ll val) {    // Range Update
458         if(r < L || R < l) return;
459         propagate(pos, l, r);
460         if(L <= l && r <= R) {
461             sum[pos] = (intervalSum(l, r, val) + sum[pos])%MOD;
462             prop[pos] = (val + prop[pos])%MOD;
463             return;
464         }
465         int mid = (l+r)>>1;
466         update(pos<<1, l, mid, L, R, val);
467         update(pos<<1|1, mid+1, r, L, R, val);
468         sum[pos] = (sum[pos<<1] + sum[pos<<1|1])%MOD;
469     }
470

```

```

471 // query(int pos, int l, int r, int L, int R) { // Range Query
472     if(r < L || R < l || L > R) return 0;
473     propagate(pos, l, r);
474     if(L <= l && r <= R) return sum[pos];
475     int mid = (l+r)>>1;
476     return (query(pos<<1, l, mid, L, R) + query(pos<<1|1, mid+1, r, L, R))%MOD;
477 };
478
479 // Segment Tree Bracket Sequencing, Modify position bracket and check if it is valid
480 struct BracketTree {
481     struct node {
482         int BrcStart, BrcEnd; // number of start bracket, number of end bracket
483         bool isOk = 0; // is the sequence valid
484
485         node(int a = 0, int b = 0) {
486             BrcStart = a;
487             BrcEnd = b;
488             isOk = (BrcStart == 0 && BrcEnd == 0);
489         }
490         node(char c) {
491             if(c == '(') BrcStart = 1, BrcEnd = 0;
492             else BrcStart = 0, BrcEnd = 1;
493         }
494         void mergeNode(node lft, node rht) {
495             if(lft.isOk && rht.isOk)
496                 BrcStart = 0, BrcEnd = 0, isOk = 1;
497             else {
498                 int match = min(lft.BrcStart, rht.BrcEnd);
499                 BrcStart = lft.BrcStart - match + rht.BrcStart;
500                 BrcEnd = lft.BrcEnd + rht.BrcEnd - match;
501                 (BrcStart == 0 && BrcEnd == 0) ? isOk = 1 : isOk = 0;
502             }
503         }
504     };
505     node tree[MAX*4];
506     void init(int pos, int L, int R, char s[]) {
507         if(L == R) {
508             tree[pos] = node(s[L]);
509             return;
510         }
511         int mid = (L+R)>>1;
512         init(pos<<1, L, mid, s);
513         init(pos<<1|1, mid+1, R, s);
514         tree[pos].mergeNode(tree[pos<<1], tree[pos<<1|1]);
515     }
516     void update(int pos, int L, int R, int idx, char val) { // idx : index of the changed value
517         if(idx < L || R < idx) return; // val : changed bracket sequence in char ( or )
518         if(L == R && L == idx) {
519             tree[pos] = node(val);
520             return;
521         }
522         int mid = (L+R)>>1;
523         update(pos<<1, L, mid, idx, val);
524         update(pos<<1|1, mid+1, R, idx, val);
525         tree[pos].mergeNode(tree[pos<<1], tree[pos<<1|1]);
526     }
527     bool isValid() { // Returns True if sequence is valid
528         return tree[1].isOk;
529     }
530 // Outputs Largest Balanced Bracket Sequence in range [L, R]
531 struct MaxBracketSeq {
532     struct node {
533         // lftBracket, rhtBracket, Max;
534         node(int lft=0, int rht=0, int Max=0) {
535             this->lftBracket = lft;
536             this->rhtBracket = rht;
537             this->Max = Max;
538         }
539     };
540     node tree[MAX*4];
541     void Merge(const node &lft, const node &rht) {
542         int common = min(lft.lftBracket, rht.rhtBracket);
543         lft.lftBracket = lft.lftBracket + rht.lftBracket - common;
544         rht.rhtBracket = lft.rhtBracket + rht.rhtBracket - common;
545         return node(lft.lftBracket, rht.rhtBracket, lft.Max+rht.Max+common);
546     }
547     void init(int pos, int l, int r, char s[]) {
548         if(l == r) {
549             if(s[l-1] == '(') tree[pos] = node(1, 0, 0);
550             else tree[pos] = node(0, 1, 0);
551             return;
552         }
553         int mid = (l+r)>>1;
554         init(pos<<1, l, mid, s);
555         init(pos<<1|1, mid+1, r, s);
556         tree[pos] = Merge(tree[pos<<1], tree[pos<<1|1]);
557     }
558 }

```



```

559
560 node query(ll pos, ll l, ll r, ll L, ll R) {
561     if(r < L || R < l) return node();
562     if(L <= l && r <= R) return tree[pos];
563     ll mid = (l+r)>>1;
564     node lft = query(pos<<1, l, mid, L, R);
565     node rht = query(pos<<1|1, mid+1, r, L, R);
566     return Merge(lft, rht);
567 }
568
569 int MaxSequence(int SEQ_SIZE, int l, int r) {
570     return 2*query(1, 1, SEQ_SIZE, l, r).Max;
571 };
572
573 // Path Compression Basics
574 // in segment tree comparison of index must be checked like (where l, r is the query range):
575 // outside of range [l, r] : r < point[L] || point[R] < l
576 // inside of range [l, r] : l <= point[L] && point[R] <= r
577 // The Queries {l, r} will be in a queue, and processed after CompressPath and initialization is done
578
579 void CompressPath(vector<int> &point) { // point contains all left and right boundary and query boundaries
580     point.push_back(0); // push_back a minimum value which is lower than input values
581     sort(point.begin(), point.end()); // so that the input values start from index 1
582     point.erase(unique(point.begin()+1, point.end()), point.end()); // Only unique points taken, this will be the compressed points
583 }
584
585 // Finding Number of Uniques in Range + OFFLINE processing
586
587 struct FindUnique {
588     int tree[4*MAX], prop[4*MAX], v[MAX], IDX[MAX];
589     map<int, vector<int> > Map;
590     map<pair<int, int>, int> Ans;
591     vector<pair<int, int> > Query;
592
593     void init() {
594         memset(IDX, -1, sizeof IDX);
595         memset(tree, 0, sizeof tree);
596         Ans.clear(), Map.clear(), Query.clear();
597     }
598
599     void update(int pos, int L, int R, int idx, int val) {
600         if(idx < L || R < idx) return;
601         if(L == R) {
602             tree[pos] += val;
603             return;
604         }
605         int mid = (L+R)>>1;
606         update(pos<<1, L, mid, idx, val);
607         update(pos<<1|1, mid+1, R, idx, val);
608         tree[pos] = tree[pos<<1] + tree[pos<<1|1];
609     }
610
611     int query(int pos, int L, int R, int l, int r) {
612         if(r < L || R < l) return 0;
613         if(l <= L && R <= r) return tree[pos];
614         int mid = (L+R)>>1;
615         int lft = query(pos<<1, L, mid, l, r);
616         int rht = query(pos<<1|1, mid+1, R, l, r);
617         return lft+rht;
618     }
619
620     void ArrayInput(int SZ) {
621         for(int i = 1; i <= SZ; ++i) scanf("%d", &v[i]);
622     }
623
624     void QueryInput(int q) {
625         int l, r;
626         while(q--) {
627             scanf("%d %d", &l, &r);
628             Query.push_back(make_pair(l, r));
629             Map[r].push_back(l); // Used for sorting
630         }
631     }
632
633     void GenAns(int SZ) {
634         map<int, vi>::iterator it;
635         int lPos = 0;
636         for(it = Map.begin(); it != Map.end(); ++it) { // For each query's right points
637             while(lPos < it->first) { // Update from last left position to this queries right position
638                 lPos++;
639                 if(IDX[v[lPos]] == -1) {
640                     IDX[v[lPos]] = lPos;
641                     update(1, 1, SZ, lPos, 1); // if new value found, increment 1 to the
642                 }
643                 else {
644                     int pastIDX = IDX[v[lPos]];
645                     IDX[v[lPos]] = lPos;
646                     update(1, 1, SZ, pastIDX, -1); // if value found previous, then remove 1 from previous index (add -1)
647                     update(1, 1, SZ, lPos, 1); // add 1 to the new position
648                 }
649             }
650             for(int i = 0; i < (int)(it->second).size(); ++i) // Range sum query for all queries that ends on this point
651                 Ans[make_pair(it->second[i], it->first)] = query(1, 1, SZ, it->second[i], it->first);
652         }
653     }
654
655     void PrintAns() {

```

```

647     for(int i = 0; i < (int)Query.size(); ++i) // Output according to input query
648         printf("%d\n", Ans[mp.Query[i].first - Query[i].second]);
649     });
650
651 struct STreeMultipleOf3 {
652     int tree[4*MAX][3], prop[4*MAX];
653     void init(int pos, int L, int R) {
654         if(L == R) {
655             tree[pos][0] = 1, tree[pos][1] = tree[pos][2] = 0;
656             return;
657         }
658         int mid = (L+R)>>1;
659         init(pos<<1, L, mid);
660         init(pos<<1|1, mid+1, R);
661         for(int i = 0; i < 3; ++i)
662             tree[pos][i] = tree[pos<<1][i] + tree[pos<<1|1][i];
663     }
664     void shiftVal(int pos, int step) {
665         step %= 3;
666         if(step == 0) return;
667         swap(tree[pos][2], tree[pos][1]);
668         swap(tree[pos][1], tree[pos][0]);
669         if(step == 2) {
670             swap(tree[pos][2], tree[pos][1]);
671             swap(tree[pos][1], tree[pos][0]);
672         }
673     void propagate(int pos, int L, int R) {
674         if(L == R || prop[pos] == 0) return;
675         shiftVal(pos<<1, prop[pos]), shiftVal(pos<<1|1, prop[pos]);
676         prop[pos<<1] += prop[pos], prop[pos<<1|1] += prop[pos];
677         prop[pos] = 0;
678     }
679     void update(int pos, int L, int R, int l, int r) { // update l to r by 1
680         if(r < L || R < l) return;
681         if(prop[pos] != 0) propagate(pos, L, R);
682         if(l <= L && R <= r) {
683             shiftVal(pos, 1);
684             prop[pos] += 1;
685             return;
686         }
687         int mid = (L+R)>>1;
688         update(pos<<1, L, mid, l, r);
689         update(pos<<1|1, mid+1, R, l, r);
690         for(int i = 0; i < 3; ++i)
691             tree[pos][i] = tree[pos<<1][i] + tree[pos<<1|1][i];
692     }
693     int query(int pos, int L, int R, int l, int r) { // return number of multiple of 3 in range l to r
694         if(r < L || R < l) return 0;
695         propagate(pos, L, R);
696         if(l <= L && R <= r) return tree[pos][0];
697         int mid = (L+R)>>1;
698         int lft = query(pos<<1, L, mid, l, r);
699         int rht = query(pos<<1|1, mid+1, R, l, r);
700         return lft+rht;
701     });
702
703 // CS Academy Candles :https://csacademy.com/contest/archive/task/candles/statement/
704 struct SortedST { // Performs -1 from n nodes and keeps nodes sorted (descending order)
705     struct node { int val, prop; };
706     node tree[5*MAX];
707     void init(int pos, int l, int r, int val[]) { // val[] must be sorted from hi to low
708         if(l == r) {
709             tree[pos].val = val[l];
710             tree[pos].prop = 0;
711             return;
712         }
713         int mid = (l+r)>>1;
714         init(pos<<1, l, mid, val), init(pos<<1|1, mid+1, r, val);
715         tree[pos].val = max(tree[pos<<1].val, tree[pos<<1|1].val);
716         tree[pos].prop = 0;
717     }
718     void propagate(int pos, int l, int r) {
719         if(tree[pos].prop == 0 || l == r) {
720             tree[pos].prop = 0;
721             return;
722         }
723         tree[pos<<1|1].prop += tree[pos].prop;
724         tree[pos<<1].prop += tree[pos].prop;
725         tree[pos<<1].val += tree[pos].prop;
726         tree[pos<<1|1].val += tree[pos].prop;
727         tree[pos].val = max(tree[pos<<1].val, tree[pos<<1|1].val);
728         tree[pos].prop = 0;
729     }
730     int findVal(int pos, int l, int r, int idx) { // Finds value in index idx
731         if(l == r) return tree[pos].val;
732         propagate(pos, l, r);
733         int mid = (l+r)>>1;
734         if(idx <= mid) return findVal(pos<<1, l, mid, idx);

```

```

735     else         return findVal(pos<<1, mid+1, r, idx);
736 }
737 void update(int pos, int l, int r, int L, int R, int val) {
738     if(r < L || R < l) return;
739     propagate(pos, l, r);
740     if(L <= l && r <= R) {
741         tree[pos].val += val;
742         tree[pos].prop += val;
743         return;
744     }
745     int mid = (l+r)>>1;
746     update(pos<<1, l, mid, L, R, val);
747     update(pos<<1, mid+1, r, L, R, val);
748     tree[pos].val = max(tree[pos<<1].val, tree[pos<<1].val);
749 }
750 int rightMost(int pos, int l, int r, int val) {      // Finds rightmost value in tree that contains val
751     if(l == r) return l;
752     if(tree[pos].val < val) return 0;
753     propagate(pos, l, r);
754     int mid = (l+r)>>1;
755     if(tree[pos<<1].val >= val) return rightMost(pos<<1, mid+1, r, val);
756     return rightMost(pos<<1, l, mid, val);
757 }
758 bool MinusQuery(int q, int n) {                      // Decreases q nodes by 1
759     if(q > n) return 0;
760     int posVal = findVal(1, 1, n, q);
761     if(posVal <= 0) return 0;
762     int r = rightMost(1, 1, n, posVal);
763     int l = rightMost(1, 1, n, posVal+1);
764     int rem = q - l;
765     if(l >= 1) update(1, 1, n, l, l, -1);
766     if(r-rem+1 <= r) update(1, 1, n, r-rem+1, r, -1);
767     return 1;
768 }};

```

File: /mnt/Work/notes/SqrtDecompose.cpp

```

1 // Sqrt Decomposition
2 // Problem: https://www.codechef.com/problems/CHEFEXQ
3
4 // Operations:
5 // 1 : Update value x at pos i
6 // 2 : Find subarray XOR of value k from index 1 to r (All Subarray starts from 1)
7 // Approach:
8 // 1 : All segment consecutive xor is calculated in Seg array
9 // 2 : All segment consecutive xor is also counted on SegMap
10 // 2 : Updates are done on each Decomposed segment array
11 // 3 : Queries are combined from all Decomposed array in the range
12
13 //----- Sqrt Decompose Functions Start-----//
14
15 int BlockSize = Seg[1010][1010]; // BlockSize is the size of each Block
16 int SegMap[330][1110007] = {0};
17
18 void Update(int v[], int l, int val) { // Updates value in position l : val
19     int idx = l/BlockSize // Block Index
20     int lft = (l/BlockSize)*BlockSize // The leftmost index of array v, which is the 0 position of Segment idx
21     v[lft] = val; // Setting value to default array to ease
22
23     // Clear full block and re-calculate // Using memset in large array will cause TLE
24     SegMap[idx][Seg[idx][0]]--; // Decreasing previous value
25     Seg[idx][0] = v[lft];
26     SegMap[idx][v[lft]]++; // Increasing with new value
27     for(int i = 1; i < BlockSize; ++i, ++lft) {
28         SegMap[idx][Seg[idx][i]]--;
29         Seg[idx][i] = Seg[idx][i-1] ^ v[lft];
30         SegMap[idx][Seg[idx][i]]++;
31     }
32
33 int Query(int l, int r, int k) { // Query in range l -- r for k
34     int Count = 0, val = 0;
35     while(l%BlockSize != 0 && l < r) { // if l partially lies inside of a sqrt segment
36         //cout << "P1" << endl;
37         Count += (Seg[l/BlockSize][l%BlockSize] == k);
38         val ^= Seg[l/BlockSize][l%BlockSize];
39         ++l;
40     }
41     while(l+BlockSize <= r) { // for all full sqrt segment
42         Count += SegMap[l/BlockSize][k^val];
43         val ^= Seg[l/BlockSize][BlockSize-1];
44         l += BlockSize;
45     }
46     while(l <= r) { // for the rightmost partial sqrt segment values
47         Count += (Seg[l/BlockSize][l%BlockSize] == (k^val));
48         ++l;
49     }
50 }

```

```

51 return Count;
52 }
53 void SqrtDecompose(int v[], int len) { // Builds Sqrt segments
54     int idx, pos, val = 0;
55     BlockSize = sqrt(len); // Calculating Block size
56     for(int i = 0; i < len; ++i) {
57         idx = i/BlockSize; // Index of block
58         pos = i%BlockSize; // Index of block element
59         if(pos == 0) val = 0;
60         val ^= v[i];
61         Seg[idx][pos] = val;
62         SegMap[idx][val]++;
63     }
64 } //----- Sqrt Decompose Functions End-----//
65
66 int v[100100];
67 int main() {
68     int n, q, idx, x, t;
69     sf("%d %d", &n, &q);
70     for(int i = 0; i < n; ++i)
71         sf("%d", &v[i]);
72     SqrtDecompose(v, n);
73     while(q--) {
74         sf("%d", &t);
75         if(t == 1) {
76             sf("%d %d", &idx, &x);
77             Update(v, idx-1, x);
78         }
79         else {
80             sf("%d %d", &idx, &x);
81             pf("%d\n", Query(0, idx-1, x));
82     }
}

```

File: /mnt/Work/notes/SSSPnegativeWeight.cpp

```

1 //Single Source Shortest Path (Negative Cycle)
2 //Complexity : O(VE)
3
4
5 vector<int> G[MAX], W[MAX];
6 int V, E, dist[MAX];
7 void bellmanFord(int source) { // If there exists disconnected graphs, then add a dummy source node which will
8     for(int i = 0; i <= V; i++) // direct to all nodes with cost 0, and run bellmanFord from that virtual node
9         dist[i] = INF; // set to -INF if max distance is needed
10    dist[source] = 0;
11    for(int i = 0; i < V-1; i++) // relax all edges V-1 times, if virtual node added, run V times
12        for(int u = 0; u < V; u++) // all the nodes, if virtual node added, run within u <= V
13            for(int j = 0; j < (int)G[u].size(); j++) {
14                int v = G[u][j], w = W[u][j]; // relax edges, set to max if max value needed
15                if(dist[u] != INF) // if there is a negative weight, then INF + negative weight < INF and INF becomes +-INF
16                    dist[v] = min(dist[v], dist[u] + w);
17    }
18
19    bool hasNegativeCycle() {
20        for(int u = 0; u < V; u++)
21            for(int i = 0; i < G[u].size(); i++) { // if bellmanFord is run for max values, then this code will
22                int v = G[u][i], w = W[u][i]; // return true for positive cycle by adding this line
23                if(dist[v] > dist[u] + w) // if(dist[v] < dist[u] + w)
24                    return 1;
25            }
26        return 0;
27    }
28
29    bool vis[MAX][2];
30    void negativePoint(int u) { // Works in undirected graph
31        queue<pair<int, bool>> q; // if vis[v][1] == 1 then there exists an negative cycle
32        q.push(make_pair(u, 0)); // vis[v][1] is true for all nodes which are in negative cycle and
33        memset(vis, 0, sizeof vis); // the nodes that can be reached from the negative cycle nodes
34        vis[u][0] = 1; // on one/more path from u to v
35        while(!q.empty()) {
36            u = q.front().first;
37            bool neg = q.front().second;
38            q.pop();
39            for(int i = 0; i < (int)G[u].size(); ++i) {
40                int v = G[u][i];
41                int w = W[u][i];
42                if(dist[v] > dist[u] + w)
43                    neg = 1;
44                if(vis[v][neg])
45                    continue;
46                vis[v][neg] = 1;
47                q.push(make_pair(v, neg));
48            }
49        }
50    }
}

```

File: /mnt/Work/notes/StronglyConnected.cpp

```
1 //Strongly Connected Component (Tarjan)
2 //Complexity : O(V+E)
3
4 vector<int> G[MAX], SCC;
5 int dfs_num[MAX], dfs_low[MAX], dfsCounter, SCC_no = 0;
6 bitset<MAX> visited;
7 map<int, int> Component; // For Creating new SCC (ConnectNode function)
8
9 void tarjanSSC(int u) {
10     // Stack, here, it is implemented as vector instead
11     SCC.push_back(u);
12     // Marking node u as visited
13     // visited[u] marks if the node u is usable in a SCC and not used on other SCC
14     // if visited[u] is false, then it is used in other SCC
15     visited[u] = 1;
16     dfs_num[u] = dfs_low[u] = ++dfsCounter;
17     // for all Strongly Connected Component (directed graph), dfs_low[u] is same
18     for(int i = 0; i < (int)G[u].size(); i++) {
19         int v = G[u][i];
20         // if it is not visited yet, backtrack it
21         if(dfs_num[v] == 0)
22             tarjanSSC(v);
23
24         // visited[v] is used to check if this node is not in any other SCC
25         if(visited[v])
26             dfs_low[u] = min(dfs_low[u], dfs_low[v]);
27     }
28
29     // in a SCC the first node of the SCC, node u is the first node in a SCC if dfs_low[u] == dfs_low[v]
30     // as we implementing stack like data structure, the nodes from top to u are on the same SCC
31     if(dfs_low[u] == dfs_num[u]) {
32         SCC_no++; // Component Node no. starts from 0
33
34         // ----- Use if ONLY IF ConnectNode / Printing needed -----
35         bool first = 1;
36         while(1) {
37             int v = SCC.back();
38             SCC.pop_back();
39
40             // node v is used, so marking it as false, so that the ancestor nodes
41             // doesn't use this node to update it's value
42
43             visited[v] = 0;
44             // printf("%d\n", v);
45             Component[v] = SCC_no; // Marking SCC nodes to as same component
46             if(u == v)
47                 break;
48         }
49         // printf("\n");
50     }
51 }
52
53 void ConnectNode() { // This function can convert Components to a new graph (G1)
54     map<int, int> :: iterator it = Component.begin();
55
56     for(; it != Component.end(); ++it) {
57         for(int i = 0; i < (int)G[it->first].size(); ++i) {
58             int v = G[it->first][i];
59             if(it->second == Component[v]) // No Self loop in new graph
60                 continue;
61             G1[it->second].push_back(Component[v]);
62         }
63     }
64
65 void RunSSC(int V) {
66     memset(dfs_num, 0, sizeof(dfs_num));
67     dfsCounter = 0;
68     visited.reset();
69     SCC_no = 0;
70     for(int i = 1; i <= V; i++)
71         if(dfs_num[i] == 0)
72             tarjanSSC(i);
73 }
```

File: /mnt/Work/notes/Templates.cpp

```
1 // Fast IO with Templates
2
3 #include <bits/stdc++.h>
4 using namespace std;
5 #define MAX 510000
6 #define EPS 1e-9
```

```

7 #define INF          1e7
8 #define MOD          1000000007
9 #define pb            push_back
10 #define mp           make_pair
11 #define fi           first
12 #define se           second
13 #define pi            acos(-1)
14 #define pf            printf
15 #define sf(XX)        scanf("%lld", &XX)
16 #define SIZE(a)        ((ll)a.size())
17 #define ALL(S)         S.begin(), S.end()
18 #define Equal(a, b)    (abs(a-b) < EPS)
19 #define Greater(a, b)   (a >= (b+EPS))
20 #define GreaterEqual(a, b) (a > (b-EPS))
21 #define FOR(i, a, b)    for(register int i = (a); i < (int)(b); ++i)
22 #define FORR(i, a, b)   for(register int i = (a); i > (int)(b); --i)
23 #define FastIO         ios_base::sync_with_stdio(false); cin.tie(NULL);
24 #define FileRead(S)     freopen(S, "r", stdin);
25 #define FileWrite(S)     freopen(S, "w", stdout);
26 #define Unique(X)        X.erase(unique(X.begin(), X.end()), X.end())
27 #define STOLL(X)         stoll(X, 0, 0)
28
29 #define isOn(S, j)       (S & (1 << j))
30 #define setBit(S, j)     (S |= (1 << j))
31 #define clearBit(S, j)   (S &= ~(1 << j))
32 #define toggleBit(S, j)  (S ^= (1 << j))
33 #define lowBit(S)        (S & (-S))
34 #define setAll(S, n)     (S = (1 << n) - 1)
35
36 typedef unsigned long long ull;
37 typedef long long ll;
38 typedef map<int, int> mii;
39 typedef map<ll, ll> mll;
40 typedef map<string, int> msi;
41 typedef vector<int> vi;
42 typedef vector<ll> vll;
43 typedef pair<int, int> pii;
44 typedef pair<ll, ll> pll;
45 typedef vector<pair<int, int>> vii;
46 typedef vector<pair<ll, ll>> vlll;
47
48 //int dx[] = {-1, 0, 1, 0}, dy[] = {0, 1, 0, -1};
49 //int dx[] = {-1, -1, -1, 0, 0, 1, 1, 1}, dy[] = {-1, 0, 1, -1, 1, -1, 0, 1};
50 //-----
51
52 inline void fastIn(int &num) { // Fast IO, with space and new line ignoring
53     bool neg = false;
54     register int c;
55     num = 0;
56     c = getchar_unlocked();
57     for(; c != '-' && (c < '0' || c > '9'); c = getchar_unlocked());
58     if (c == '-') {
59         neg = true;
60         c = getchar_unlocked();
61     }
62     for(; c >= '0' && c <= '9'; c = getchar_unlocked())
63         num = (num << 1) + (num << 3) + c - 48;
64     if (neg)
65         num *= -1;
66 }
67
68 inline void fastOut (long long n) {
69     long long N = n, rev, count = 0;
70     rev = N;
71     if (N == 0) { putchar('0'); return ;}
72     while ((rev % 10) == 0) { count++; rev /= 10;} //obtain the count of the number of 0s
73     rev = 0;
74     while (N != 0) { rev = (rev << 3) + (rev << 1) + N % 10; N /= 10;} //store reverse of N in rev
75     while (rev != 0) { putchar(rev % 10 + '0'); rev /= 10;}
76     while (count--) putchar('0');
77 }
78
79
80 // Scanf Trick
81 // input: (alpha+omega)^2
82 // scanf("[ ] %[^+ ] %*[+ ] %[^)] %s", a, b, n);
83 // %* is used for skipping
84 // %[ ] skipping (
85 // %[^+ ] take input until +
86 // %*[+ ] skipping +
87 // %[^)] skipping ^ and )

```

File: /mnt/Work/notes/TreeQuery.cpp

1 // Tree Max Distance Node

```

2 // Set any node as root, then do dfs and find the farthest node, then again from that farthest node
3 // do dfs for farthest node, the two nodes are the farthest node
4
5 pii dfs(int u, int par, int d) {
6     pii ret = {u, d}; // {distance, node}
7     for(int i = 0; i < (int)G[u].size(); ++i)
8         if(G[u][i] != par)
9             ret = max(ret, dfs(G[u][i], u, d+W[u][i]));
10    return ret;
11 }
12
13 int GetDistance() {
14     pii left = dfs(0, -1, 0);
15     pii right = dfs(left.second, -1, 0);
16     return right.first;
17 }
18
19 // Codeforces :E. Propagating tree ( http://codeforces.com/contest/384/problem/E )
20 // Given a tree (node 1 - n)
21 // perform two operations:
22 // 1. Add x value to node u, Add -x value to node u's immediate children, Add x to their immediate children, and so on
23 // in other words, add value x to all childs where (parentLevel%2 == childLevel%2), add -val otherwise
24 // 2. Output value of node u
25
26 vector<int> G[MAX];
27 int sTime[MAX], eTime[MAX], level[MAX], cst[MAX], timer;
28 BIT EvenNode, OddNode;
29
30 /* sTime : starting time of node n
31    eTime : finishing time of node n
32    1
33    /\
34   5 6
35   /\
36   7 4
37   /\
38   2 3
39 discover nodes : {1, 5, 6, 7, 4, 2, 3}
40 sTime[] = {1, 6, 7, 5, 2, 3, 4} index starts from 1, i'th index contains start time of i'th node
41 eTime[] = {7, 6, 7, 7, 2, 7, 4}
42
43 calculate child :
44 for node 6 : childs are in range sTime[6] - eTime[6] : 3 - 7
45 so child nodes are : 6, 7, 4, 2, 3 (discover node index range)
46 we don't need discover time vector to calculate distance
47 notice, if we only update with sTime and eTime, the range update will always be right */
48
49 void dfs(int u, int lv) {
50     sTime[u] = ++timer;
51     level[u] = lv;
52     for(int i = 0; i < (int)G[u].size(); ++i)
53         if(sTime[G[u][i]] == 0)
54             dfs(G[u][i], lv+1);
55     eTime[u] = timer;
56 }
57
58 void AddVal(int node, int val) {
59     if(level[node]%2 == 0) {
60         EvenNode.update(sTime[node], eTime[node], val);
61         OddNode.update(sTime[node], eTime[node], -val);
62     }
63     else {
64         EvenNode.update(sTime[node], eTime[node], -val);
65         OddNode.update(sTime[node], eTime[node], val);
66     }
67 }
68
69 int GetVal(int node) {
70     // cst[node] contains initial cost (if exists)
71     return cst[node] + (level[node]%2==0 ? EvenNode.read(sTime[node]) : OddNode.read(sTime[node]));
72 }
73
74 // Complete Binary Tree
75 // Sum of distance from a node "n" such that every nodes distance from node "n" is less than or equal to k
76 // http://mishadoff.com/blog/dfs-on-binary-tree-array/
77
78 vector<ll> v[MAX], w, sum[MAX]; // W[i] contains weight of i'th node
79 int n, m;
80 void dfs(int node = 1) { // node starts from 1
81     if(node > n) return;
82     ll lft = node<<1, rht = node<<1|1;
83     dfs(lft), dfs(rht);
84     ll lftSize = v[lft].size(), rhtSize = v[rht].size();
85     ll nodeSize = lftSize + rhtSize + 1;
86     v[node].resize(nodeSize);
87     v[node][0] = 0; // distance from this node to this node
88
89     printf("node : %d, nodeSize : %d, lftSize : %d, rhtSize : %d\n", node, nodeSize, lftSize, rhtSize);

```

```

90  ll l = 0, r = 0;
91  for(ll i = 1; i < nodeSize; ++i) {
92      if(l == lftSize)
93          v[node][i] = v[rht][r++] + w[rht];
94      else if(r == rhtSize)
95          v[node][i] = v[lft][l++] + w[lft];
96      else {
97          int lftW = v[lft][l] + w[lft], rhtW = v[rht][r] + w[rht];
98          if(lftW < rhtW) {
99              v[node][i] = lftW;
100             l++;
101         }
102         else {
103             v[node][i] = rhtW;
104             r++;
105         }
106     }
107 }
108 ll single(int node, ll d, ll delta) {
109     if(d < 0) return 0;
110     ll n = upper_bound(v[node].begin(), v[node].end(), d) - v[node].begin();
111     return sum[node][n-1] + delta * n; // delta is the common distance of all nodes
112 }
113 ll query(int node, ll k) {
114     ll ans = single(node, k, 0);
115     ll totlen = 0;
116     while(node > 2) {
117         totlen += w[node];
118         ll tmp = single(node/2, k - totlen, totlen); // distances from parent node
119         tmp -= single(node, k - totlen - w[node], totlen + w[node]); // common overlapped distance (of child node) from parent node
120         ans += tmp;
121         node /= 2;
122     }
123     return ans;
124 }
125
126 void PreCal() { // First run dfs(), then run PreCal()
127     for(int i = 1; i <= n; ++i) {
128         sum[i].resize(v[i].size());
129         sum[i][0] = v[i][0];
130         for(int j = 1; j < SIZE(v[i]); ++j)
131             sum[i][j] = sum[i][j-1] + v[i][j];
132     }
}

```

File: /mnt/Work/notes/Trie.cpp

```

1  //Trie
2  //Complexity : making a trie : O(S), searching : O(S)
3
4  #include <bits/stdc++.h>
5  using namespace std;
6
7  bool found;
8
9  struct node {
10     bool isEnd;
11     node *next[11];
12     node() {
13         isEnd = false;
14         for(int i = 0; i < 10; i++)
15             next[i] = NULL;
16     }
17 };
18
19 //trie of a string abc, ax
20 // [start] --> [a] --> [b] --> [c] --> endMark
21 // |
22 // [x] --> endMark
23
24 //creates trie, returns true if the trie we are creating is a segment of a string
25 //to only create a trie remove lines which are comment marked
26
27 bool create(char str[], int len, node *current) {
28     for(int i = 0; i < len; i++) {
29         int pos = str[i] - '0';
30         if(current->next[pos] == NULL)
31             current->next[pos] = new node();
32         current = current->next[pos];
33         if(current->isEnd) //
34             return true; //
35     }
36     current->isEnd = true; //
37     return false; //
38 }
39
40 void del(node *current) {

```



```

41 for(int i = 0; i < 10; i++)
42     if(current->next[i] != NULL)
43         del(current->next[i]);
44 delete current;
45 }
46
47 void check(node *current) {
48     for(int i = 0; i < 10; i++) {
49         if(current->next[i] != NULL)
50             check(current->next[i]);
51     }
52     if(found)
53         return;
54     if(current->isEnd && !found) {
55         for(int i = 0; i < 10 && !found; i++)
56             if(current->next[i] != NULL) {
57                 found = 1;
58             }
59     }
60 }
61
62 int main() {
63     //freopen("in", "r", stdin);
64     //freopen("out", "w", stdout);
65     int t, n;
66     char S[15];
67     scanf("%d", &t);
68     while(t--) {
69         found = 0;
70         node* root = new node(); //important part of the code
71         scanf("%d", &n);
72         while(n--) {
73             scanf("%s", S);
74             if(!found)
75                 if(create(S, strlen(S), root))
76                     found = 1;
77         }
78         if(!found)
79             check(root);
80         if(found)
81             printf("NO\n");
82         else
83             printf("YES\n");
84         del(root);
85     }
86     return 0;
87 }

```