

```

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.                                     // Now, let's build the
    failure function
39.     queue<int> q;
40.     for(int c = 0; c <= highestChar - lowestChar; ++c) // Iterate over every
    possible input
41.         if(g[0][c] != -1 and g[0][c] != 0) { // All nodes s of depth 1
    have f[s] = 0
42.             f[g[0][c]] = 0;
43.             q.push(g[0][c]);
44.         }
45.
46.     while(q.size()) {
47.         int state = q.front();
48.         q.pop();
49.         for(int c = 0; c <= highestChar - lowestChar; ++c) {
50.             if(g[state][c] != -1) {

```

```

51.         int failure = f[state];
52.
53.         while(g[failure][c] == -1)
54.             failure = f[failure];
55.
56.         failure = g[failure][c];
57.         f[g[state][c]] = failure;
58.         out[g[state][c]] |= out[failure];           // Merge out values
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 cnt[TOTKEY];
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. }
110.
111. // All Pair Shortest Path
112. // Floyd Warshal
113. // Complexity : O(V^3)
114.
115. int G[MAX][MAX], parent[MAX][MAX];
116. void graphINIT() {
117.     for(int i = 0; i < MAX; i++)
118.         for(int j = 0; j < MAX; j++)
119.             G[i][j] = INF;
120.     for(int i = 0; i < MAX; i++)
121.         G[i][i] = 0;
122. }
123. void floydWarshall(int V) {
124.     for(int i = 0; i < V; i++) // path printing matrix initialization
125.         for(int j = 0; j < V; j++)
126.             parent[i][j] = i; // we can go to j from i by only obtaining i (by
default)
127.     for(int k = 0; k < V; k++) // Selecting a middle point as k
128.         for(int i = 0; i < V; i++) // Selecting all combination of source (i) and
destination (j)
129.             for(int j = 0; j < V; j++)
130.                 if(G[i][k] != INF && G[k][j] != INF) { // if the graph contains
negative edges, then min(INF, INF+ negative edge) = +-INF!
131.                     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
132.                     parent[i][j] = parent[k][j]; // if path printing needed
133.                 }
134. void printPath(int i, int j) {
135.     if(i != j) printPath(i, parent[i][j]);
136.     printf(" %d", j);
137. }
138. void minMax(int V) {
139.     for(int k = 0; k < V; k++)
140.         for(int i = 0; i < V; i++)
141.             for(int j = 0; j < V; j++)
142.                 G[i][j] = min(G[i][j], max(G[i][k], G[k][j]));
143. }
144. void transitiveClosure(int V) {
145.     for(int k = 0; k < V; k++)
146.         for(int i = 0; i < V; i++)
147.             for(int j = 0; j < V; j++)
148.                 G[i][j] |= (G[i][k] & G[k][j]);
149. }
150.
151. // Articulation Point
152. // Complexity O(V+E)
153. // Tarjan, DFS
154.
155. vector<int>G[101];
156. int dfs_num[101], dfs_low[101], parent[101], isAtriculationPoint[101], dfsCounter,
rootChildren, dfsRoot;
157. void articulationPoint(int u) {
158.     dfs_low[u] = dfs_num[u] = ++dfsCounter;
159.     for(int i = 0; i < G[u].size(); i++) {
160.         int v = G[u][i];

```

```

161.     if(dfs_num[v] == 0) {
162.         parent[v] = u;
163.         if(u == dfsRoot)           // Special case for root node
164.             rootChildren++;       // if root node has child, increment counter
165.         articulationPoint(v);
166.         // 1 : if dfs_num[u] == dfs_low[v], then it is a back edge
167.         // 2 : if dfs_num[u] < dfs_low[v], then u is ancestor of v and there is no back
edge
168.         // so, if u is not root node, then we can chose u for Articulation Point
169.         if(dfs_num[u] <= dfs_low[v] && u != dfsRoot)    //Avoiding root node
170.             isArticulationPoint[u]++;
171.         // if there is any child node of u that is a back edge of a previous node
172.         // then the value of dfs_low[v] might be less than the present dfs_low[u]
173.         // we try to save the lowest value possible
174.         dfs_low[u] = min(dfs_low[v], dfs_low[u]);
175.     }
176.     // As nodes are bi-directional, avoiding direct child node
177.     // if it is not direct child node, and visited, then there is a back edge
178.     // so we try to decrease the value of dfs_low[u] with the dfs_num[v]
179.     // the dfs_num[v] is less than dfs_num[u] (as it is a back edge)
180.     else if(parent[u] != v)
181.         dfs_low[u] = min(dfs_low[u], dfs_num[v]);
182. }}
183.
184. int main() {
185.     // Actual code of Articulation Point starts here
186.     dfsCounter = 0;
187.     memset(dfs_num, 0, sizeof(dfs_num));
188.     isArticulationPoint.reset();
189.     for(int i = 1; i <= n; i++) {
190.         if(dfs_num[i] == 0) {
191.             dfsCounter = rootChildren = 0;
192.             dfsRoot = i;
193.             articulationPoint(i);
194.             isArticulationPoint[i] = (rootChildren > 1);
195.         }
196.         // Important
197.         isAtriculationPoint + 1 = number of nodes that is disconnected
198.     }
199.     // Printing Articulation Points
200.     /*for(int i = 0; i < 101; i++)
201.         if(isArticulationPoint[i])
202.             printf("%d ", i);
203.     printf("\n");*/
204.     printf("%d\n", (int)isArticulationPoint.count());
205. }
206.
207. // Basic BFS with path printing
208. // Complexity : O(V+E)
209.
210. vector<int>parent, G[MAX];
211. void printPath(int u, int source_node) {           // destination, source
212.     if(u == source_node) {
213.         printf("%d", u);
214.         return;
215.     }
216.     printPath(parent[u], source_node);
217.     printf(" %d", u);

```

```

218. }
219.
220. int BFS(int source_node, int finish_node, int vertices) {
221.     vector<int>dist(vertices+5, INF);           //contains the distance from source to
end point
222.     queue<int>Q;
223.     Q.push(source_node);
224.     parent.resize(vertices+5, -1);             //for path printing
225.     dist[source_node] = 0;
226.
227.     while(!Q.empty()) {
228.         int u = Q.front();
229.         Q.pop();
230.         if(u == finish_node)                   //remove this line if shortest path to
all nodes are needed
231.             return dist[u];
232.         for(int i = 0; i < G[u].size(); i++) {
233.             int v = G[u][i];
234.             if(dist[v] == INF) {
235.                 dist[v] = dist[u] + 1;
236.                 parent[v] = u;
237.                 Q.push(v);
238.             }
239.         }
240.     }
241.
242. int color[100];                               // Contains Color (1, 2)
243. void Bicolor(int u) {                         // Bicolor Check
244.     queue<int>q;
245.     q.push(u);
246.     color[u] = 1;                             // Color is -1 initialized
247.     while(!q.empty()) {
248.         u = q.front();
249.         q.pop();
250.         for(int i = 0; i < (int)G[u].size(); ++i) {
251.             int v = G[u][i];
252.             if(color[v] == -1) {
253.                 if(color[u] == 1)    color[v] = 2;
254.                 else                 color[v] = 1;
255.                 q.push(v);
256.             }
257.         }
258.     }
259.
260. // BigInteger By Jane Alam Jan
261. struct Bigint {
262.     string a;                                // to store the digits (in reverse order)
263.     int sign;                                // sign = -1 for negative numbers, sign = 1 otherwise
264.     Bigint() {}                               // default constructor
265.     Bigint(string b) {(*this) = b;}           // constructor for string
266.     Bigint(long long n) {
267.         sign = n >= 0 ? 1:-1;
268.         if(n == 0) {
269.             a.push_back('0');
270.             return;
271.         }
272.         while(n) {
273.             a.push_back(n%10 + '0');
274.             n /= 10;

```

```

274.     }
275.     //reverse(a.begin(), a.end());
276. }
277. int size() {           // returns number of digits
278.     return a.size();
279. }
280. Bigint inverseSign() { // changes the sign
281.     sign *= -1;
282.     return (*this);
283. }
284. Bigint normalize(int newSign) {           // removes leading 0, fixes sign
285.     for( int i = a.size() - 1; i > 0 && a[i] == '0'; i-- )
286.         a.erase(a.begin() + i);
287.     sign = ( a.size() == 1 && a[0] == '0' ) ? 1 : newSign;
288.     return (*this);
289. }
290. //----- assignment operator
291. void operator = (string b) {           // assigns a string to Bigint
292.     a = b[0] == '-' ? b.substr(1) : b;
293.     reverse(a.begin(), a.end());
294.     this->normalize( b[0] == '-' ? -1 : 1 );
295. }
296. //----- conditional operators
297. bool operator < ( const Bigint &b ) const { // less than operator
298.     if( sign != b.sign ) return sign < b.sign;
299.     if( a.size() != b.a.size() )
300.         return sign == 1 ? a.size() < b.a.size() : a.size() > b.a.size();
301.     for( int i = a.size() - 1; i >= 0; i-- ) if( a[i] != b.a[i] )
302.         return sign == 1 ? a[i] < b.a[i] : a[i] > b.a[i];
303.     return false;
304. }
305.
306. bool operator == ( const Bigint &b ) const { // operator for equality
307.     return a == b.a && sign == b.sign;
308. }
309. // mathematical operators
310. void Pow(int p) { // Raises a Bigint to power of p
311.     Bigint res("1");
312.     while(p > 0) {
313.         if(p&1) res = res * (*this);
314.         p = p >> 1;
315.         (*this) = (*this) * (*this);
316.     }
317.     (*this) = res;
318. }
319. Bigint operator + ( Bigint b ) { // addition operator overloading
320.     if( sign != b.sign ) return (*this) - b.inverseSign();
321.     Bigint c;
322.     for(int i = 0, carry = 0; i < a.size() || i < b.size() || carry; i++) {
323.         carry+=(i<a.size() ? a[i]-48 : 0)+(i<b.a.size() ? b.a[i]-48 : 0);
324.         c.a += (carry % 10 + 48);
325.         carry /= 10;
326.     }
327.     return c.normalize(sign);
328. }
329. Bigint operator - ( Bigint b ) { // subtraction operator overloading
330.     if( sign != b.sign ) return (*this) + b.inverseSign();
331.     int s = sign; sign = b.sign = 1;

```

```

332.     if( (*this) < b ) return ((b - (*this)).inverseSign()).normalize(-s);
333.     Bigint c;
334.     for(int i = 0, borrow = 0; i < a.size(); i++ ) {
335.         borrow = a[i] - borrow - (i < b.size() ? b.a[i] : 48);
336.         c.a += borrow >= 0 ? borrow + 48 : borrow + 58;
337.         borrow = borrow >= 0 ? 0 : 1;
338.     }
339.     return c.normalize(s);
340. }
341. Bigint operator * (Bigint b) { // multiplication operator
overloading
342.     Bigint c("0");
343.     for(int i = 0, k = a[i] - 48; i < a.size(); i++, k = a[i] - 48) {
344.         while(k--) c = c + b; // ith digit is k, so, we add k
times
345.         b.a.insert(b.a.begin(), '0'); // multiplied by 10
346.     }
347.     return c.normalize(sign * b.sign);
348. }
349. Bigint operator / ( Bigint b ) { // division operator overloading
350.     if( b.size() == 1 && b.a[0] == '0' ) b.a[0] /= ( b.a[0] - 48 );
351.     Bigint c("0"), d;
352.     for( int j = 0; j < a.size(); j++ ) d.a += "0";
353.     int dSign = sign * b.sign; b.sign = 1;
354.     for( int i = a.size() - 1; i >= 0; i-- ) {
355.         c.a.insert( c.a.begin(), '0' );
356.         c = c + a.substr( i, 1 );
357.         while( !( c < b ) ) c = c - b, d.a[i]++;
358.     }
359.     return d.normalize(dSign);
360. }
361. Bigint operator % ( Bigint b ) { // modulo operator overloading
362.     if( b.size() == 1 && b.a[0] == '0' ) b.a[0] /= ( b.a[0] - 48 );
363.     Bigint c("0");
364.     b.sign = 1;
365.     for( int i = a.size() - 1; i >= 0; i-- ) {
366.         c.a.insert( c.a.begin(), '0' );
367.         c = c + a.substr( i, 1 );
368.         while( !( c < b ) ) c = c - b;
369.     }
370.     return c.normalize(sign);
371. }
372. //-----output method
373. void print() {
374.     if( sign == -1 ) putchar('-');
375.     for( int i = a.size() - 1; i >= 0; i-- ) putchar(a[i]);
376. };
377.
378. int main() {
379.     Bigint a, b, c; // declared some Bigint variables
380.     string input; // string to take input
381.     cin >> input; // take the Big integer as string
382.     a = input; // assign the string to Bigint a
383.     cin >> input; // take the Big integer as string
384.     b = input; // assign the string to Bigint
385.     c = a + b; // adding a and b
386.     c.print(); // printing the Bigint
387.     puts(""); // newline

```

```

388.     return 0;
389. }
390.
391. // Vertex Cover
392. // Wiki: Vertex Cover:
393. // In the mathematical discipline of graph theory, a vertex cover (sometimes node cover)
394. // of a graph is a set of vertices such that each edge of the graph is incident to at least
395. // one vertex of the set
396. // Wiki: Edge Cover:
397. // In graph theory, an edge cover of a graph is a set of edges such that every vertex of the
398. // graph
399. // is incident to at least one edge of the set
400.
401. // Min Edge Cover = TotalNodes - MinVertexCover
402.
403. bitset<MAX>vis;
404. int lft[MAX], rht[MAX];
405. vector<int>G[MAX];
406.
407. int VertexCover(int u) { // Min Vertex Cover
408.     vis[u] = 1;
409.     for(int i = 0; i < (int)G[u].size(); ++i) {
410.         int v = G[u][i];
411.         if(vis[v]) continue; // If v is used earlier, skip
412.         vis[v] = 1;
413.         if(lft[v] == -1) { // If there is no node present on left of v
414.             lft[v] = u, rht[u] = v;
415.             return 1;
416.         }
417.         else if(VertexCover(lft[v])) { // If there is one node present on the left side
418.             of v (Let it be u')
419.             lft[v] = u, rht[u] = v; // and if it is possible to match u' with
420.             another node (not v ofcourse!)
421.             return 1; // then we can match this u with v, and u' is
422.             matched with another node as well
423.         }
424.     }
425.     return 0;
426. }
427.
428. int BPM(int n) { // Bipartite Matching
429.     int cnt = 0;
430.     memset(lft, -1, sizeof lft);
431.     memset(rht, -1, sizeof rht);
432.     for(int i = 1; i <= n; ++i) { // Nodes are numbered from 1 to n
433.         vis.reset();
434.         cnt += VertexCover(i); // Check if there exists a match for node i
435.     }
436.     return cnt;
437. }
438.
439. //Complexity : O(V+E)
440. //Finding Bridges (Graph)
441.
442. vector<int> G[MAX];
443. vector<pair<int, int> >ans;
444. int dfs_num[MAX], dfs_low[MAX], parent[MAX], dfsCounter;
445.
446. void bridge(int u) {
447.     // dfs_num[u] is the dfs counter of u node

```



```

441. // dfs_low[u] is the minimum dfs counter of u node (it is minimum if a backedge exists)
442. dfs_num[u] = dfs_low[u] = ++dfsCounter;
443. for(int i = 0; i < (int)G[u].size(); i++) {
444.     int v = G[u][i];
445.     if(dfs_num[v] == 0) {
446.         parent[v] = u;
447.         bridge(v);
448.         // if dfs_num[u] is lower than dfs_low[v], then there is no back edge on u node
449.         // so u - v can be a bridge
450.         if(dfs_num[u] < dfs_low[v])
451.             ans.push_back(make_pair(min(u, v), max(u, v)));
452.         // obtaining lower dfs counter (if found) from child nodes
453.         dfs_low[u] = min(dfs_low[u], dfs_low[v]);
454.     }
455.     // if v is not parent of u then it is a back edge
456.     // also dfs_num[v] must be less than dfs_low[u]
457.     // so we update it
458.     else if(parent[u] != v)
459.         dfs_low[u] = min(dfs_low[u], dfs_num[v]);
460. }
461. void FindBridge(int V){ //Bridge finding code
462.     memset(dfs_num, 0, sizeof(dfs_num));
463.     dfsCounter = 0;
464.     for(int i = 0; i < V; i++)
465.         if(dfs_num[i] == 0)
466.             bridge(i);
467. }
468. int main() {
469.     FindBridge(100);
470.     // Output
471.     sort(ans.begin(), ans.end());
472.     for(int i = 0; i < ans.size(); i++)
473.         printf("%d - %d\n", ans[i].first, ans[i].second);
474.     printf("\n");
475.     return 0;
476. }
477.
478. vi DecimalVal(int a, int b) { // Calculate Decimal values (after .) of a/b
479.     vi v;
480.     a %= b;
481.     if(a == 0) {
482.         v.pb(0);
483.         return v;
484.     }
485.     bool first = 1;
486.     while(SIZE(v) <= 200) { // Define the Maximum Length of decimal values
487.         if(a == 0) // If any Zero divisor is found (then, rest all will
488.             return v; // be Zero) return values
489.         else if(a < b && !first) { // If we need to add another zero (add zero after
first time)
490.             a*=10;
491.             v.pb(0);
492.         }
493.         else if(a < b && first) { // If we need to add a extra zero (adding zero first
time)
494.             first = 0;
495.             a *= 10;

```

```

496.         continue;
497.     }
498.     else {
499.         v.pb(a/b);
500.         a%=b;
501.         first = 1;
502.     }
503. }
504. return v;
505. }
506.
507. // Repetation (PunoPonik) is also calculated
508. vi dec1, dec2; // Before . (decimal), after . (decimal)
509. int DecimalRepeated(int a, int b) { // Calculate Decimal values (after .) of a/b
510.     unordered_map<int, int>mp;
511.     int k = 0, point = -1;
512.     bool divisible = 0;
513.     if(a >= b) { // Before Decimal Calculation
514.         dec1.push_back(a/b);
515.         a %= b;
516.     }
517.     if(dec1.size() == 0)
518.         dec1.push_back(0);
519.     while(a != 0) {
520.         if(mp.find(a) != mp.end()) { // if the remainder is found again, there exists
a loop
521.             point = mp[a];
522.             break;
523.         }
524.         if(a%b == 0) {
525.             dec2.push_back(a/b);
526.             break;
527.         }
528.         mp[a] = k++;
529.         int cnt = 0;
530.         while(a < b) {
531.             a *= 10;
532.             if(cnt != 0) {
533.                 dec2.push_back(0);
534.                 k++;
535.             }
536.             ++cnt;
537.         }
538.         if(cnt != 0 && mp.find(a) != mp.end()) {
539.             point = mp[a];
540.             break;
541.         }
542.         if(cnt == 1)
543.             mp[a] = (k-1);
544.         dec2.push_back(a/b);
545.         a %= b;
546.         if(a == 0) {
547.             divisible = 1;
548.             break;
549.         }
550.     }
551.     return divisible == 1 ? 1 : ((int)dec2.size()-point);
552. }

```

```

553. int main() {
554.     int a, b;
555.     cin >> a >> b;
556.     vi v = DecimalVal(a, b);
557.     for(auto it : v)
558.         cout << it;
559.     cout << endl;
560.     int Cycle = DecimalRepeated(a, b);
561.     for(auto it : dec1)
562.         cout << it;
563.     cout << ".";
564.     for(auto it : dec2)
565.         cout << it;
566.     cout << "\n\n";
567.     cout << "Last Repeating Cycle " << Cycle << endl;
568.     return 0;
569. }
570.
571. // Cycle in Directed graph
572. // http://codeforces.com/contest/915/problem/D
573.
574. vi G[550];
575. int color[550], Cycle = 0;           // Cycle will contain the number of cycles found in graph
576. void dfs(int u) {
577.     color[u] = 2;                     // Mark as parent
578.     for(auto v : G[u]) {
579.         if(color[v] == 2)             // If any Parent found (BackEdge)
580.             Cycle++;
581.         else if(!color[v])
582.             dfs(v);
583.     }
584.     color[u] = 1;                     // Visited
585. }
586.
587. // Shortest Path (Dijkstra)
588. // Complexity : (V*logV + E)
589.
590. vector<int>dist, G[MAX], W[MAX];
591. void printPath(int u) {               // call with ending node
592.     if (u == s) {                     // s is the starting node
593.         printf("%d", s);             // base case, at the source s
594.         return;
595.     }
596.     printPath(p[u]);                  // recursive: to make the output format: s -> ... -> t
597.     printf(" %d", u);
598. }
599.
600. void dikjstra(int u, int destination, int nodes) {
601.     dist.resize(nodes+1, INF);         // dist[v] contains the distance from u
        to v
602.     dist[u] = 0;
603.     priority_queue<pair<int, int> > pq; // pq is sorted in ascending order
        according to weight and edge
604.     pq.push({0, -u});
605.
606.     while(!pq.empty()) {
607.         int u = -pq.top().second;
608.         int wu = -pq.top().first;

```

```

609.         pq.pop();
610.         if(u == destination) return;           // if we only need distance of
destination, then we may return
611.         if(wu > dist[u]) continue;           // skipping the longer edges, if we have
found shorter edge earlier
612.
613.         for(int i = 0; i < G[u].size(); i++) {
614.             int v = G[u][i];
615.             int wv = W[u][i];
616.             if(wu + wv < dist[v]) {           // path relax
617.                 dist[v] = wu + wv;
618.                 p[v] = u;                     // path printing
619.                 pq.push({-dist[v], -v});
620.             }
621.
622. // Kth Path Using Modified Dijkstra
623. // Complexity : O(K*(V*logV + E))
624. // http://codeforces.com/blog/entry/16821
625.
626. vector<int>G[MAX], W[MAX], dist[MAX];
627. int KthDijkstra(int Start, int End, int Kth) { // Kth Shortest Path (Visits Edge Only
Once)
628.     for(int i = 0; i < MAX; ++i)
629.         dist[i].clear();
630.     priority_queue<pii>pq;                     // Weight, Node
631.     pq.push(make_pair(0, Start));
632.
633.     while(!pq.empty()) {
634.         int u = pq.top().second;
635.         int w = -pq.top().first;
636.         pq.pop();
637.
638.         if((int)dist[End].size() == Kth)       // We can also break if the Kth path is found
639.             return dist[End].back();
640.         if(dist[u].empty())
641.             dist[u].push_back(w);
642.         else if(dist[u].back() != w)           // Not taking same cost paths
643.             dist[u].push_back(w);             // As priority queue greedily chooses edge, it's
guranteed that this edge is bigger than previous
644.         if((int)dist[u].size() > Kth)         // Like basic dijkstra, we'll not take the Kth+
edges
645.             continue;
646.         for(int i = 0; i < (int)G[u].size(); ++i) {
647.             int v = G[u][i];
648.             int _w = w + W[u][i];
649.             if((int)dist[v].size() == Kth)
650.                 continue;
651.             pq.push(make_pair(-_w, v));
652.         }
653.         return -1;
654.     }
655.
656. int KthDijkstra(int Start, int End, int Kth) { // Kth Shortest Path (Visits Same Edge
More Than Once if required)
657.     for(int i = 0; i < MAX; ++i)
658.         dist[i].clear();
659.     priority_queue<pii>pq;                     // Weight, Node
660.     pq.push(make_pair(0, Start));

```

```

661.
662.     while(!pq.empty()) {
663.         int u = pq.top().second;
664.         int w = -pq.top().first;
665.         pq.pop();
666.
667.         if(dist[u].empty())
668.             dist[u].push_back(w);
669.         else if(dist[u].back() != w) {           // if the weight is not same
670.             if((int)dist[u].size() < Kth)         // if we have to take more costs, take it
671.                 dist[u].push_back(w);
672.             else if(dist[u].back() <= w)         // if the cost is greater than previous,
then, don't go further
673.                 continue;
674.             else {                               // we have to take this cost, and remove the
greater one
675.                 dist[u].push_back(w);
676.                 sort(dist[u].begin(), dist[u].end());
677.                 dist[u].pop_back();
678.             }}
679.             for(int i = 0; i < (int)G[u].size(); ++i) {
680.                 int v = G[u][i];
681.                 int _w = w + W[u][i];
682.                 pq.push(make_pair(-_w, v));
683.             }}
684.             if((int)dist[End].size() < Kth) return -1;
685.             return dist[End].back();
686.     }
687.
688. // Kth Shortest Path (Every edge and shortest path of previous calculation is not used)
689.
690. vector<int>G[MAX], W[MAX], S[MAX];           //edge, edge_weight,
reverse_shortest_paths_graph
691. int dist[MAX];
692. bool cut_node[MAX], cut_edge[MAX][MAX];
693.
694. int dikjstra(int source, int end, int nodes) {
695.     for(int i = 0; i < nodes; i++)           // dist[v] contains the distance from u to v
696.         dist[i] = INF;
697.     dist[source] = 0;
698.     priority_queue<pair<int, int> > pq;       // pq is sorted in ascending order according to
weight and edge
699.     pq.push({0, -source});
700.
701.     while(!pq.empty()) {
702.         int u = -pq.top().second;
703.         int wu = -pq.top().first;
704.         pq.pop();
705.         if(wu > dist[u]) continue;           // skipping the longer edges, if we have found
shorter edge earlier
706.
707.         for(int i = 0; i < (int)G[u].size(); i++) {
708.             int v = G[u][i];
709.             int wv = W[u][i];
710.
711.             if(cut_node[v] || cut_edge[u][v]) // if there exists node/edge that is used in
previous shortest path
712.                 continue;

```

```

713.         if(wu + wv < dist[v]) {                // path relax
714.             dist[v] = wu + wv;
715.             S[v].clear();                        // if this edge is smaller than other edge,
then we refresh the reverse paths of this node
716.             S[v].push_back(u);                  // then push back the node, (building a
reverse graph of shortest path(s) )
717.             pq.push({-dist[v], -v});
718.         }
719.         else if(wu + wv == dist[v])              // if there is more than one shortest paths,
then only add it in the reverse graph, nothing else
720.             S[v].push_back(u);
721.     }}
722.     return dist[end];
723. }
724.
725. void cut_off(int start, int destination) {        // this function cuts off all the nodes
726.     if(destination == start) return;
727.     for(int i = 0; i < S[destination].size(); i++) {
728.         int v = S[destination][i];
729.         cut_node[v] = 1;
730.         cut_edge[destination][v] = cut_edge[v][destination] = 1;
731.         cut_off(start, v);
732.     }}
733.
734. //-----String DP-----
735. int Palindrome(int l, int r) {                    // Building Palindrome in minimum move
736.     if(dp[l][r] != INF) return dp[l][r];
737.     if(l >= r) return dp[l][r] = 0;
738.     if(l+1 == r) return dp[l][r] = (s[l] != s[r]);
739.     if(s[l] == s[r]) return dp[l][r] = Palindrome(l+1, r-1);
740.     return dp[l][r] = min(Palindrome(l+1, r), Palindrome(l, r-1))+1;    // Adding a
alphabet on right, left
741. }
742.
743. void dfs(int l, int r) {                          // Palindrome printing, for above DP function
744.     if(l > r) return;
745.     if(s[l] == s[r]) {
746.         Palin.push_back(s[l]);
747.         dfs(l+1, r-1);
748.         if(l != r) Palin.push_back(s[l]);
749.         return;
750.     }
751.     int P = min(make_pair(dp[l+1][r], 1), make_pair(dp[l][r-1], 2)).second;
752.     if(P == 1) {
753.         Palin.push_back(s[l]);
754.         dfs(l+1, r);
755.         Palin.push_back(s[l]);
756.     }
757.     else {
758.         Palin.push_back(s[r]);
759.         dfs(l, r-1);
760.         Palin.push_back(s[r]);
761.     }}
762.
763. bool isPalindrome(int l, int r) {                 // Checks if substring l-r is palindrome
764.     if(l == r || l > r) return 1;
765.     if(dp[l][r] != -1) return dp[l][r];
766.     if(s[l] == s[r]) return dp[l][r] = isPalindrome(l+1, r-1);

```

```

767.     return 0;
768. }
769.
770. int recur(int p1, int p2) {          // make string s1 like s2, in minimum move
771.     if(dp[p1][p2] != INF)
772.         return dp[p1][p2];
773.     if(p1 == l1 or p2 == l2) {      // reached end of string s1 or s2
774.         if(p1 < l1) return dp[p1][p2] = recur(p1+1, p2)+1;
775.         if(p2 < l2) return dp[p1][p2] = recur(p1, p2+1)+1;
776.         return dp[p1][p2] = 0;
777.     }
778.     if(s1[p1] == s2[p2])            // match found
779.         return dp[p1][p2] = recur(p1+1, p2+1);
780.     // change at position p1, delete position p1, insert at position p1
781.     return dp[p1][p2] = min(recur(p1+1, p2+1), min(recur(p1+1, p2), recur(p1, p2+1)))+1;
782. }
783.
784. void dfs(int p1, int p2) {          // printing function for above dp
785.     if(dp[p1][p2] == 0)            // end point (value depends on topdown/bottomup)
786.         return;
787.     if(s1[p1] == s2[p2]) {         // match found, no operation
788.         dfs(p1+1, p2+1);
789.         return;
790.     }
791.     int P = min(mp(dp[p1+1][p2], 1), min(mp(dp[p1][p2+1], 2), mp(dp[p1+1][p2+1],
792. 3))).second;
793.     if(P == 1) dfs(p1+1, p2);      // delete s1[p1] from position p2 of s1
794.     else if(P == 2) dfs(p1, p2+1); // insert s2[p2] on position p2 of s1 string
795.     else dfs(p1+1, p2+1);         // change s1[p2] to s2[p2] on position p2 of
796.     string s1
797. }
798. int reduce(int l, int r) {          // Reduce string AXD00D00 (len : 8) to AX(D0^2)^2
799.     (len : 4)
800.     if(l > r) return INF;
801.     if(l == r) return 1;
802.     if(dp[l][r] != -1) return dp[l][r];
803.     int ret = r-l+1;
804.     int len = ret;
805.     for(int i = l; i < r; ++i)      // A B D O O D O O remove A X substring
806.         ret = min(ret, reduce(l, i)+reduce(i+1, r));
807.     for(int d = 1; d < len; ++d) { // D O O D O O to check all divisable length
808.         substring
809.         if(len%d != 0) continue;
810.         for(int i = l+d; i <= r; i += d)
811.             for(int k = 0; k < d; ++k)
812.                 if(s[l+k] != s[i+k])
813.                     goto pass;
814.         ret = min(ret, reduce(l, l+d-1));
815.         pass;;
816.     }
817.     return dp[l][r] = ret;
818. }
819.
820. // Light OJ 1073 - DNA Sequence
821. // FIND and PRINT shortest string after merging multiple string together
822.

```

```

820. int matchDP[20][20];
821. int TryMatch(int x, int y) { // Finds First overlap of two string
822.     if(matchDP[x][y] != -1) // ABAAB + AAB : Match at 2
823.         return matchDP[x][y];
824.     for(size_t i = 0; i < v[x].size(); ++i) {
825.         for(size_t j = i, k = 0; j < v[x].size() && k < v[y].size(); ++j, ++k)
826.             if(v[x][j] != v[y][k])
827.                 goto pass;
828.         return matchDP[x][y] = i;
829.     pass::;
830. }
831. return matchDP[x][y] = v[x].size();
832. }
833.
834. int dp[16][(1<<15)+100];
835. int recur(int mask, int last) { // DP part of LIGHT OJ
836.     if(dp[last][mask] != -1) // eliminate all substrings from n string
first in main function!
837.         return dp[last][mask]; // it's not handled here
838.     if(mask == (1<<n)-1)
839.         return dp[last][mask] = v[last].size();
840.     int ret = INF, cost;
841.     for(int i = 0; i < n; ++i) {
842.         if(isOn(mask, i))
843.             continue;
844.         int mPos = TryMatch(last, i);
845.         if(mPos < (int)v[last].size())
846.             cost = (int)v[last].size() - ((int)v[last].size() - mPos);
847.         else
848.             cost = v[last].size();
849.         ret = min(ret, recur(mask | (1 << i), i) + cost);
850.     }
851.     return dp[last][mask] = ret;
852. }
853.
854. string ans;
855. void dfs(int mask, int last, string ret) { // PRINTING part of LIGHT OJ
856.     if(!ret.empty() && ans < ret)
857.         return;
858.     if(mask == (1<<n)-1) {
859.         ret += v[last];
860.         if(ret < ans)
861.             ans = ret;
862.         return;
863.     }
864.     for(int i = 0; i < n; ++i) {
865.         if(isOn(mask, i))
866.             continue;
867.         int mPos = TryMatch(last, i);
868.         int cost;
869.         if(mPos < (int)v[last].size())
870.             cost = (int)v[last].size() - ((int)v[last].size() - mPos);
871.         else
872.             cost = v[last].size();
873.         if(dp[last][mask] - cost == dp[i][mask | (1<<i)])
874.             dfs(mask | (1<<i), i, ret + v[last].substr(0, cost));
875.     }
876. }

```



```

877. //-----Digit DP-----
878.
879. // Complexity : O(10*idx*sum*tight) : LightOJ 1068
880. // Tight contains if there is any restriction to number (Tight is initially 1)
881. // Initial Params: (MaxDigitSize-1, 0, 0, 1, modVal, allowed_digit_vector)
882.
883. ll dp[15][100][100][2];
884. ll digitSum(int idx, int sum, ll value, bool tight, int mod, vector<int>&MaxDigit) {
885.     if (idx == -1)
886.         return ((value == 0) && (sum == 0));
887.     if (dp[idx][sum][value][tight] != -1)
888.         return dp[idx][sum][value][tight];
889.     ll ret = 0;
890.     int lim = (tight)? MaxDigit[idx] : 9; // Numbers are
generated in reverse order
891.     for (int i = 0; i <= lim; i++) {
892.         bool newTight = (MaxDigit[idx] == i)? tight : 0; // calculating newTight
value for next state
893.         ll newValue = value ? ((value*10) % mod)+i : i;
894.         ret += digitSum(idx-1, (sum+i)%mod, newValue%mod, newTight, mod, MaxDigit);
895.     }
896.     return dp[idx][sum][value][tight] = ret;
897. }
898.
899. // Bit DP (Almost same as Digit DP) : LighOJ 1032
900. // Complexity O(2*pos*total_bits*tights*number_of_bits)
901. // Initial Params : (MostSignificantOnBitPos, N, 0, 0, 1)
902. // Call as : bitDP(SigOnBitPos, N, 0, 0, 1) N is the Max Value, calculating [0 - N]
903. // Tight is initially on
904. // pairs are number of paired bits, prevOn shows if previous bit was on (it is for this
problem)
905.
906. #define isOn(x, i) (x & (1LL<<i))
907. #define On(x, i) (x | (1LL<<i))
908. #define Off(x, i) (x & ~(1LL<<i))
909. int N, lastBit;
910. long long dp[33][33][2][2];
911. ll bitDP(int pos, int mask, int pairs, bool prevOn, bool tight) {
912.     if(pos < 0)
913.         return pairs;
914.     if(dp[pos][pairs][prevOn][tight] != -1)
915.         return dp[pos][pairs][prevOn][tight];
916.     bool newTight = tight & !isOn(mask, pos); // Turn off tight when we are turning off a
bit which was initially on
917.     ll ans = bitDP(pos-1, Off(mask, pos), pairs, 0, newTight);
918.     if(On(mask, pos) <= N)
919.         ans += bitDP(pos-1, On(mask, pos), pairs + prevOn, 1, tight && isOn(mask, pos));
920.     return dp[pos][pairs][prevOn][tight] = ans;
921. }
922.
923. // Memory Optimized DP + Bottom Up solution (LOJ : 1126 - Building Twin Towers)
924. // given array v of n elements, make two value x1 and x2 where x1 == x2, output maximum of
it
925.
926. int dp[2][500010], n; // present dp table and past dp
table
927. int BottomUp(int TOT) { // TOT = (Cumulative Sum of v)/2
928.     memset(dp, -1, sizeof dp);

```

```

929.     dp[0][0] = 0;
930.     bool present = 0, past = 1;
931.     for(int i = 0; i < n; ++i) {
932.         present ^= 1, past ^= 1; // Swapping present and past dp
table
933.         for(int diff = 0; diff <= TOT; ++diff)
934.             if(dp[past][diff] != -1) {
935.                 int moreDiff = diff + v[i], lessDiff = abs(diff - v[i]);
936.                 dp[present][diff] = max(dp[present][diff], dp[past][diff]);
937.                 dp[present][lessDiff] = max(dp[present][lessDiff], max(dp[past][lessDiff],
dp[past][diff] + v[i]));
938.                 dp[present][moreDiff] = max(dp[present][moreDiff], max(dp[past][moreDiff],
dp[past][diff] + v[i]));
939.             }
940.         return (max(dp[0][0], dp[1][0]))/2; // Returns the maximum possible
answer
941.     }
942.
943. // 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)
944. ll CountNumberOfWays(int r, int c, int n) {
945.     v[n] = {r, c}; // also add the last point as
unaccessable point, to find how many
946.     sort(v.begin(), v.end()); // ways we can come to this point, which
is the answer
947.     for(int i = 0; i <= n; ++i) {
948.         dp[i] = CountWay(1, 1, v[i].first, v[i].second); // Number of ways we
can come from starting square
949.         for(int j = 0; j < i; ++j)
950.             if(v[j].first <= v[i].first and v[j].second <= v[i].second)
951.                 dp[i] = (dp[i] - (dp[j] * CountWay(v[j].first, v[j].second, v[i].first,
v[i].second))%MOD + MOD)%MOD;
952.     } // Number of ways we can reach from (1,
1) to (r, c)
953.     return dp[n]; // The last state is always (r, c),
which is the answer
954. }
955.
956. // Travelling Salesman
957. // dist[u][v] = distance from u to v
958. // dp[u][bitmask] = dp[node][set_of_taken_nodes] (saves the best(min/max) path)
959. // call with tsp(starting node, 1)
960.
961. int n, x[11], y[11], dist[11][11], memo[11][1 << 11], dp[11][1 << 11];
962. int TSP(int u, int bitmask) { // startin node and bitmask of taken nodes
963.     if(bitmask == ((1 << (n)) - 1)) // when it steps in this node, if all nodes are
visited
964.         return dist[u][0]; // then return to 0'th (starting) node [as the
path is hamiltonian]
965.     // or use return dist[u][start] if starting node is not 0
966.     if(dp[u][bitmask] != -1) // if we have previous value set up
967.         return dp[u][bitmask]; // use that previous value
968.     int ans = 1e9; // set an infinite value
969.     for(int v = 0; v <= n; v++) // for all the nodes
970.         if(u != v && !(bitmask & (1 << v))) // if this node is not the same node, and if
this node is not used yet(in bitmask)
971.             ans = min(ans, dist[u][v] + tsp(v, bitmask | (1 << v))); // min(past_val,
dist u->v + dist(v->all other untaken nodes))

```

```

972.     return dp[u][bitmask] = ans;                // save in dp and return
973. }
974.
975. // Basic DSU with compression
976.
977. struct DSU {
978.     vector<int>u_list, u_set;                // u_list[x] : the size of a set x, u_set[x] : the
root of x
979.     DSU() {}
980.     DSU(int SZ) { init(SZ); }
981.     int unionRoot(int n) {                    // Union making with dynamic compression
982.         if(u_set[n] == n) return n;
983.         return u_set[n] = unionRoot(u_set[n]); // Directly set the actual root of this
set as root (Compress)
984.     }
985.     int makeUnion(int a, int b) {              // Union making with compression
986.         int x = unionRoot(a), y = unionRoot(b);
987.         if(x == y) return x;                  // If both are in same set
988.         else if(u_list[x] > u_list[y]) {       // Makes x root (y -> x)
989.             u_set[y] = x;
990.             u_list[x] += u_list[y];            // Root's size is increased
991.             return x;
992.         }
993.         else {                                 // Makes y root (x -> y)
994.             u_set[x] = y;
995.             u_list[y] += u_list[x];            // Root's size is increased
996.             return y;
997.         }
998.     }
999.     void init(int len) {                       // Initializer
1000.         u_list.resize(len+5);
1001.         u_set.resize(len+5);
1002.         for(int i = 0; i <= len+3; i++)
1003.             u_set[i] = i, u_list[i] = 1;        // Each node contains itself, so size of
each node set to 1
1004.     }
1005.     bool isRoot(int x) {                       // Returns true if this is a root (May
contain one or many nodes)
1006.         return u_set[x] == x;
1007.     }
1008.     bool isRootContainsMany(int x) {           // If the root contains more than one
value (Actual Root)
1009.         return (isRoot(x) && (u_list[x] > 1));
1010.     }
1011.     bool isSameSet(int a, int b) {             // If a and b is in same set/component
1012.         return (unionRoot(a) == unionRoot(b));
1013.     };
1014. // Bipartite DSU (Tested)
1015.
1016. struct BipartiteDSU {
1017.     vector<int>u_list, u_set, u_color;
1018.     vector<bool>missmatch;                    // Bicolor mismatch
1019.
1020.     BipartiteDSU() {}
1021.     BipartiteDSU(int SZ) { init(SZ); }
1022.
1023.     pll unionRoot(int n) {                     // Union making with dynamic
compression

```

```

1024.     if(u_set[n] == n) return {n, u_color[n]};
1025.     pll root = unionRoot(u_set[n]);
1026.     if(mismatch[u_set[n]] or mismatch[n])
1027.         mismatch[n] = mismatch[u_set[n]] = 1;
1028.     u_color[n] = (u_color[n] + root.second)&1;
1029.     u_set[n] = root.first; // Directly set the actual root
of this set as root (Compress)
1030.     return {u_set[n], u_color[n]};
1031. }
1032. int makeUnion(int a, int b) { // Union making with
compression
1033.     int x = unionRoot(a).first, y = unionRoot(b).first;
1034.     if(x == y) { // If both are in same
set and bipartite mismatch exists
1035.         if(u_color[a] == u_color[b]) mismatch[x] = 1;
1036.         return x;
1037.     }
1038.     if(mismatch[x] or mismatch[y]) // Checks if Bipartite
mismatch exists
1039.         mismatch[x] = mismatch[y] = 1;
1040.         if(u_list[x] < u_list[y]) { // Makes x root (y -> x)
1041.             u_set[x] = y;
1042.             u_list[x] += u_list[y]; // Root's size is
increased
1043.             u_color[x] = (u_color[a]+u_color[b]+1)&1; // Setting color of
component y according to the color of a & b
1044.             return y;
1045.         }
1046.         else { // Makes y root (x -> y)
1047.             u_set[y] = x;
1048.             u_list[y] += u_list[x]; // Root's size is increased
1049.             u_color[y] = (u_color[a]+u_color[b]+1)&1; // Setting color of component y
according to the color of a & b
1050.             return x;
1051.         }
1052.     }
1053. void init(int len) { // Initializer
1054.     u_list.resize(len+5);
1055.     u_set.resize(len+5);
1056.     u_color.resize(len+5);
1057.     mismatch.resize(len+5);
1058.     for(int i = 0; i <= len+3; i++)
1059.         u_set[i] = i, u_list[i] = 1, u_color[i] = 0, mismatch[i] = 0;
1060. }
1061. bool isRoot(int x) { // Returns true if this is a root (May contain one
or many nodes)
1062.     return u_set[x] == x;
1063. }
1064. bool isRootContainsMany(int x) { // If the root contains more than one value (Actual
Root)
1065.     return (isRoot(x) && (u_list[x] > 1));
1066. }
1067. bool isSameSet(ll a, ll b) { // If a and b is in same set/component
1068.     return (unionRoot(a).first == unionRoot(b).first);
1069. }
1070. int getColor(ll u) { // Color of node u (DONT get the color of root)
1071.     return u_color[u];
}

```

```

1072.     bool hasMismatch(int x) { // If there is bipartite mismatch in this
set/component
1073.         return mismatch[x];
1074.     };
1075.
1076. // Dynamic Weighted DSU (Checked, Not Tested)
1077.
1078. struct WeightedDSU {
1079.     vector<int>u_list, u_set, u_weight, weight;
1080.     WeightedDSU() {}
1081.     WeightedDSU(int SZ) { init(SZ); }
1082.     int unionRoot(int n) { // Union making with compression
1083.         if(u_set[n] == n) return n;
1084.         return u_set[n] = unionRoot(u_set[n]); // Directly set the actual root
of this set as root (Compress)
1085.     }
1086.     void changeWeight(int u, int w, bool first = 1) { // Change any component's weight
(Dynamic)
1087.         if(first) w = w - weight[u];
1088.         u_weight[u] += w;
1089.         if(u_set[u] != u)
1090.             changeWeight(u_set[u], w, 0);
1091.     }
1092.     int makeUnion(int a, int b) { // Union making with compression
1093.         int x = unionRoot(a), y = unionRoot(b);
1094.         if(x == y) return x;
1095.         if(u_list[x] > u_list[y]) { // Makes x root (y -> x)
1096.             u_set[y] = x;
1097.             u_list[x] += u_list[y]; // Root's size is increased
1098.             u_weight[x] += u_weight[y]; // Root's weight is increased
1099.             return x;
1100.         }
1101.         else { // Makes y root (x -> y)
1102.             u_set[x] = y;
1103.             u_list[y] += u_list[x]; // Root's size is increased
1104.             u_weight[y] += u_weight[x]; // Root's weight is increased
1105.             return y;
1106.         }
1107.     }
1108.     void init(int len) { // Initializer
1109.         u_list.resize(len+5);
1110.         u_set.resize(len+5);
1111.         u_weight.resize(len+5);
1112.         weight.resize(len+5);
1113.         for(int i = 0; i <= len+3; i++)
1114.             u_set[i] = i, u_list[i] = 1, u_weight[i] = weight[i] = 0;
1115.     }
1116.     bool isRoot(int x) { // Returns true if this is a root (May contain one
or many nodes)
1117.         return u_set[x] == x;
1118.     }
1119.     bool isRootContainsMany(int x) { // If the root contains more than one value
(Actual Root)
1120.         return (isRoot(x) && (u_list[x] > 1));
1121.     }
1122.     bool isSameSet(int a, int b) { // If a and b is in same set/component
1123.         return (unionRoot(a) == unionRoot(b));
1124.     }

```

```

1124.     void setWeight(int u, int w) { // Set weight of node u to w, run before
union
1125.         u_weight[u] = w;
1126.         weight[u] = w;
1127.     }
1128.     int getComponentWeight(int u) { // Get weight sum of the set/comopnent
1129.         return u_weight[unionRoot(u)];
1130.     };
1131.
1132. // 1D Fenwick Tree
1133.
1134. struct BIT {
1135.     ll tree[MAX];
1136.     int MaxVal;
1137.     void init(int sz=1e7) {
1138.         memset(tree, 0, sizeof tree);
1139.         MaxVal = sz+1;
1140.     }
1141.     void update(int idx, ll val) {
1142.         for( ;idx <= MaxVal; idx += (idx & -idx))
1143.             tree[idx] += val;
1144.     }
1145.     void update(int l, int r, ll val) {
1146.         if(l > r) swap(l, r);
1147.         update(l, val);
1148.         update(r+1, -val);
1149.     }
1150.     ll read(int idx) {
1151.         ll sum = 0;
1152.         for( ;idx > 0; idx -= (idx & -idx))
1153.             sum += tree[idx];
1154.         return sum;
1155.     }
1156.     ll read(int l, int r) {
1157.         ll ret = read(r) - read(l-1);
1158.         return ret;
1159.     }
1160.     ll readSingle(int idx) { // Point read in log(n)
1161.         ll sum = tree[idx];
1162.         if(idx > 0) {
1163.             int z = idx - (idx & -idx);
1164.             --idx;
1165.             while(idx != z) {
1166.                 sum -= tree[idx];
1167.                 idx -= (idx & -idx);
1168.             }
1169.             return sum;
1170.         }
1171.     int search(int cSum) {
1172.         int pos = -1, lo = 1, hi = MaxVal, mid;
1173.         while(lo <= hi) {
1174.             mid = (lo+hi)/2;
1175.             if(read(mid) >= cSum) { // read(mid) >= cSum : to find the lowest index of
cSum value
1176.                 pos = mid; // read(mid) == cSum : to find the greatest index of
cSum value
1177.                 hi = mid-1;
1178.             }

```

```

1179.         else
1180.             lo = mid+1;
1181.         }
1182.         return pos;
1183.     }
1184.     ll size() {
1185.         return read(MaxVal);
1186.     }
1187.     // Modified BIT, this section can be used to add/remove/read 1 to all elements from 1 to
pos
1188.     // all of the inverse functions must be used, for any manipulation
1189.     ll invRead(int idx) {                // gives summation from 1 to idx
1190.         return read(MaxVal-idx);
1191.     }
1192.     void invInsert(int idx) {            // adds 1 to all index less than idx
1193.         update(MaxVal-idx, 1);
1194.     }
1195.     void invRemove(int idx) {           // removes 1 from idx
1196.         update(MaxVal-idx, -1);
1197.     }
1198.     void invUpdate(int idx, ll val) {
1199.         update(MaxVal-idx, val);
1200.     };
1201.     // ----- 2D Fenwick Tree -----
1202.     /* /\
1203.      y |
1204.        | (x1,y2) ----- (x2,y2)
1205.        | |             |
1206.        | |             |
1207.        | |             |
1208.        | |-----|
1209.        | (x1,y1)      (x2, y1)
1210.        |
1211.        |_____
1212.      (0, 0)                x--> */
1213.
1214.     ull tree[2510][2510];
1215.     int xMax = 2505, yMax = 2505;
1216.     // Updates from min point to MAX LIMIT
1217.     void update(int x, int y, ll val) {
1218.         int y1;
1219.         while(x <= xMax) {
1220.             y1 = y;
1221.             while(y1 <= yMax) {
1222.                 tree[x][y1] += val;
1223.                 y1 += (y1 & -y1);
1224.             }
1225.             x += (x & -x);
1226.         }
1227.     }
1227.     ll read(int x, int y) {              // Reads from (0, 0) to (x, y)
1228.         ll sum = 0;
1229.         int y1;
1230.         while(x > 0) {
1231.             y1 = y;
1232.             while(y1 > 0) {
1233.                 sum += tree[x][y1];
1234.                 y1 -= (y1 & -y1);
1235.             }

```

```
1236.         x -= (x & -x);
1237.     }
1238.     return sum;
1239. }
1240. ll readSingle(int x, int y) {
1241.     return read(x, y) + read(x-1, y-1) - read(x-1, y) - read(x, y-1);
1242. }
1243. void updateSquare(pii p1, pii p2, ll val) {    // p1 : lower left point, p2 : upper right
point
1244.     update(p1.first, p1.second, val);
1245.     update(p1.first, p2.second+1, -val);
1246.     update(p2.first+1, p1.second, -val);
1247.     update(p2.first+1, p2.second+1, val);
1248. }
1249. ll readSquare(pii p1, pii p2) {                // p1 : lower left point, p2 : upper right
point
1250.     ll ans = read(p2.first, p2.second);
1251.     ans -= read(p1.first-1, p2.second);
1252.     ans -= read(p2.first, p1.second-1);
1253.     ans += read(p1.first-1, p1.second-1);
1254.     return ans;
1255. }
1256.
1257. // // ----- 3D Fenwick Tree -----
1258.
1259. ll tree[105][105][105];
1260. ll xMax = 100, yMax = 100, zMax = 100;
1261. void update(int x, int y, int z, ll val) {
1262.     int y1, z1;
1263.     while(x <= xMax) {
1264.         y1 = y;
1265.         while(y1 <= yMax) {
1266.             z1 = z;
1267.             while(z1 <= zMax) {
1268.                 tree[x][y1][z1] += val;
1269.                 z1 += (z1 & -z1);
1270.             }
1271.             y1 += (y1 & -y1);
1272.         }
1273.         x += (x & -x);
1274.     }
1275. ll read(int x, int y, int z) {
1276.     ll sum = 0;
1277.     int y1, z1;
1278.     while(x > 0) {
1279.         y1 = y;
1280.         while(y1 > 0) {
1281.             z1 = z;
1282.             while(z1 > 0) {
1283.                 sum += tree[x][y1][z1];
1284.                 z1 -= (z1 & -z1);
1285.             }
1286.             y1 -= (y1 & -y1);
1287.         }
1288.         x -= (x & -x);
1289.     }
1290.     return sum;
1291. }
```



```
1292. 11 readRange(11 x1, 11 y1, 11 z1, 11 x2, 11 y2, 11 z2) {
1293.     --x1, --y1, --z1;
1294.     return read(x2, y2, z2)
1295.         - read(x1, y2, z2)
1296.         - read(x2, y1, z2)
1297.         - read(x2, y2, z1)
1298.         + read(x1, y1, z2)
1299.         + read(x1, y2, z1)
1300.         + read(x2, y1, z1)
1301.         - read(x1, y1, z1);
1302. }
1303. void updateRange(int x1, int y1, int z1, int x2, int y2, int z2) {      // Not tested yet!!
1304.     update(x1, y1, z1, val);
1305.     update(x2+1, y1, z1, -val);
1306.     update(x1, y2+1, z1, -val);
1307.     update(x1, y1, z2+1, -val);
1308.     update(x2+1, y2+1, z1, val);
1309.     update(x1, y2+1, z2+1, val);
1310.     update(x2+1, y1, z2+1, val);
1311.     update(x2+1, y2+1, z2+1, -val);
1312. }
1313. // Pattens to built BIT update read:
1314. // always starts with first(starting point), add val
1315. // take (1 to n) elements from ending point with all combination add it to staring point,
1316. // add (-1)^n * val
1317. struct fraction {                // Fraction Calculation Template
1318.     int a, b;
1319.     fraction() {
1320.         a = 1;
1321.         b = 1;
1322.     }
1323.     fraction(int x, int y) : a(x), b(y) {}
1324.     flip() {swap(a, b);}
1325.     fraction operator + (fraction other) {
1326.         fraction temp;
1327.         temp.b = ((b)*(other.b))/(__gcd((b), other.b));
1328.         temp.a = (temp.b/b)*a + (temp.b/other.b)*other.a;
1329.         int x = __gcd(temp.a, temp.b);
1330.         if(x != 1) {temp.a/=x; temp.b/=x;}
1331.         return temp;
1332.     }
1333.     fraction operator - (fraction other) {
1334.         fraction temp;
1335.         temp.b = (b*other.b)/__gcd(b, other.b);
1336.         temp.a = (temp.b/b)*a - (temp.b/other.b)*other.a;
1337.         int x = __gcd(temp.a, temp.b);
1338.         if(x != 1) {temp.a/=x; temp.b/=x;}
1339.         return temp;
1340.     }
1341.     fraction operator / (fraction other) {
1342.         fraction temp;
1343.         temp.a = a*other.b;
1344.         temp.b = b*other.a;
1345.         int x = __gcd(temp.a, temp.b);
1346.         if(x != 1) {temp.a/=x; temp.b/=x;}
1347.         return temp;
1348.     }
}
```

```

1349.     fraction operator * (fraction other) {
1350.         fraction temp;
1351.         temp.a = a*other.a;
1352.         temp.b = b*other.b;
1353.         int x = __gcd(temp.a, temp.b);
1354.         if(x != 1) {temp.a/=x; temp.b/=x;}
1355.         return temp;
1356.     };
1357.
1358. struct BaseInt {                                // Number Base Conversions
1359.     string val;
1360.     ll base;
1361.     BaseInt() {}
1362.     BaseInt(string _val, int _base = 10) {        // Do check if any value if val is
greater than base
1363.         val = _val;                                // Which is impossible
1364.         base = _base;
1365.     }
1366.     char reVal(int num) {
1367.         if(num >= 0 && num <= 9) return (char)(num + '0');
1368.         return (char)(num - 10 + 'A');
1369.     }
1370.     int getVal(char c) {
1371.         if(c <= '9' && c >= '0') return c-'0';
1372.         return c-'A'+10;
1373.     }
1374.     void DecimalTo(int _base) {
1375.         ll v = stoll(val);
1376.         base = _base;
1377.         val.clear();
1378.         while(v) {
1379.             val.push_back(reVal(v%base));
1380.             v /= base;
1381.         }
1382.         reverse(val.begin(), val.end());
1383.         if(val.empty()) val.push_back('0');
1384.     }
1385.     bool ToDecimal() {
1386.         ll ret = 0;
1387.         for(int i = 0; i < (int)val.size(); ++i) {
1388.             int v = getVal(val[i]);
1389.             if(v >= base) return 0;
1390.             if(i) ret *= base;
1391.             ret += v;
1392.         }
1393.         val = to_string(ret), base = 10;
1394.         return 1;
1395.     }
1396.     void ChangeBase(int to) {
1397.         if(base == to) return;                    // If input is "000", then output will also be
"000" (if base remains same)
1398.         if(base != 10) ToDecimal();                // remove the if statements to recover
1399.         if(to != 10) DecimalTo(to);
1400.     }
1401.     void Reverse() {
1402.         reverse(val.begin(), val.end());
1403.     }
1404.     BaseInt operator + (BaseInt other) const {

```

```
1405.     BaseInt a(val, base), b = other;
1406.     a.ToDecimal(), b.ToDecimal();
1407.     string sum = to_string(stoi(a.val, 0) + stoi(b.val, 0));
1408.     BaseInt ret(sum);
1409.     ret.ChangeBase(base);
1410.     return ret;
1411. };
1412.
1413. // The trig functions of C++ take radian exclusively
1414. // 0D Objects-----
1415. struct point {          // In Integer
1416.     int x, y;
1417.     point() { x = y = 0; }
1418.     point(int _x, int _y) : x(_x), y(_y) {}
1419.
1420.     bool operator < (point other) const {
1421.         if(x != other.x)
1422.             return x < other.x;
1423.         return y < other.y;
1424.     }
1425.
1426.     bool operator == (point other) const {
1427.         return (x == other.x) && (y == other.y);
1428.     };
1429.
1430. struct point {          // In Double
1431.     double x, y;
1432.     point() { x = y = 0.0; }
1433.     point(double _x, double _y) : x(_x), y(_y) {}
1434.
1435.     bool operator < (point other) const {
1436.         if (fabs(x - other.x) > EPS)
1437.             return x < other.x;
1438.         return y < other.y;
1439.     }
1440.     bool operator == (point other) const {
1441.         return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS));
1442.     };
1443.
1444. bool Equal(double a, double b) {
1445.     return (fabs(a-b) <= EPS);
1446. }
1447.
1448. int hypot(point p1, point p2) {
1449.     int x = p1.x-p2.x;
1450.     int y = p1.y-p2.y;
1451.     return x*x + y*y;
1452. }
1453.
1454. double dist(point p1, point p2) {
1455.     int x = p1.x-p2.x;
1456.     int y = p1.y-p2.y;
1457.     return sqrt(x*x + y*y);
1458. }
1459.
1460. double DEG_to_RAD(double deg) {          // Converts Degree to Radian
1461.     return (deg*PI)/180;
1462. }
```

```

1463.
1464. double RAD_to_DEG(double rad) {
1465.     return (180/PI)*rad;
1466. }
1467.
1468. point rotate(point p, double theta) {          // Rotates point p w.r.t. origin. (theta is in
degree)
1469.     double rad = DEG_to_RAD(theta);
1470.     return point(p.x * cos(rad) - p.y * sin(rad), p.x * sin(rad) + p.y * cos(rad));
1471. }
1472.
1473. double PointToArea(point p1, point p2, point p3) {          // Returns Positive
Area in if the points are clockwise, Negative for Anti-Clockwise
1474.     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
1475. }
1476.
1477. double whichSide(point p, point q, point r) {          // returns on which side point r is w.r.t
pq line
1478.     double slope = (p.y-q.y)*(q.x-r.x) - (q.y-r.y)*(p.x-q.x);
1479.     return slope;          // slope = 0 : linear, slope > 0 : right, slope < 0
: left
1480. }
1481.
1482. // 1D Objects-----
1483. struct line {
1484.     double a, b, c;
1485. };
1486. void pointsToLine(point p1, point p2, line &l) {          // ax + by + c = 0 [comes from y = mx +
c]
1487.     if (fabs(p1.x - p2.x) < EPS)          // vertical line is fine
1488.         l.a = 1.0, l.b = 0.0, l.c = -p1.x;          // default values
1489.     else {
1490.         l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
1491.         l.b = 1.0;          // IMPORTANT: we fix the value of b to
1.0
1492.         l.c = -(double)(l.a * p1.x) - p1.y;
1493.     }
1494. bool areParallel(line l1, line l2) {          // check coefficients a & b
1495.     return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS);
1496. }
1497. bool areSame(line l1, line l2) {          // also check coefficient c
1498.     return areParallel(l1, l2) && (fabs(l1.c - l2.c) < EPS);
1499. }
1500. bool areIntersect(line l1, line l2, point &p) {
1501.     if(areParallel(l1, l2)) return 0;          // no intersection
1502.     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
1503.     if(fabs(l1.b) > EPS) p.y = -(l1.a * p.x + l1.c);          // special case:
test for vertical line to avoid division by zero
1504.     else p.y = -(l2.a * p.x + l2.c);
1505.     return 1;
1506. }
1507. line perpendicularLine(line l, point p) {          // returns a perpendicular line on l
which goes through
1508.     line ret;          // point p
1509.     ret.a = l.b, ret.b = -l.a;
1510.     ret.c = -(ret.a*p.x + ret.b*p.y);

```

```

1511.     if(ret.b < 0) ret.a *= -1, ret.b *= -1, ret.c *= -1;    // as line must contain b = 1.0
by default
1512.     if(ret.b != 0) {
1513.         ret.a /= ret.b;
1514.         ret.c /= ret.b;
1515.         ret.b = 1;
1516.     }
1517.     return ret;
1518. }
1519.
1520. // Vectors -----
1521. struct vec {
1522.     double x, y;                // name: 'vec' is different from STL vector
1523.     vec(double _x, double _y) : x(_x), y(_y) {}
1524. };
1525. vec toVec(point a, point b) {    // convert 2 points to vector a->b
1526.     return vec(b.x - a.x, b.y - a.y);
1527. }
1528. vec scale(vec v, double s) {    // nonnegative s = [<1 .. 1 .. >1]
1529.     return vec(v.x * s, v.y * s);    // shorter.same.longer
1530. }
1531. point translate(point p, vec v) {    // translate p according to v
1532.     return point(p.x + v.x, p.y + v.y);
1533. }
1534. double dot(vec a, vec b) {
1535.     return (a.x * b.x + a.y * b.y);
1536. }
1537. double norm_sq(vec v) {
1538.     return v.x * v.x + v.y * v.y;
1539. }
1540.
1541. // Parametric Line -----
1542. struct ParaLine {                // Line in Parametric Form
1543.     point a, b;                // points must be in DOUBLE
1544.     ParaLine() { a.x = a.y = b.x = b.y = 0; }
1545.     ParaLine(point _a, point _b) : a(_a), b(_b) {}    // {Start, Finish} or {from, to}
1546.
1547.     point getPoint(double t) {    // Parametric Line : a + t * (b - a)
1548.         t = [-inf, +inf]
1549.         return point(a.x + t*(b.x-a.x), a.y + t*(b.y-a.y));
1550.     };
1551. // Returns the distance from p to the line defined by two points a and b (a and b must be
1552. // different)
1553. // the closest point is stored in the 4th parameter (byref)
1554. double distToLine(point p, point a, point b, point &c) {    // formula: c = a + u * ab
1555.     vec ap = toVec(a, p), ab = toVec(a, b);
1556.     double u = dot(ap, ab) / norm_sq(ab);
1557.     c = translate(a, scale(ab, u));    // translate a to c
1558.     return dist(p, c);    // Euclidean distance
1559. }
1560. // Returns the distance from p to the line segment ab defined by two points a and b (still
1561. // OK if a == b)
1562. // the closest point is stored in the 4th parameter (byref)
1563. double distToLineSegment(point p, point a, point b, point &c) {
1564.     vec ap = toVec(a, p), ab = toVec(a, b);

```

```

1564.     double u = dot(ap, ab) / norm_sq(ab);
1565.     if(u < 0.0) {
1566.         c = point(a.x, a.y);           // closer to a
1567.         return dist(p, a);             // Euclidean distance between p and a
1568.     }
1569.     if (u > 1.0) {
1570.         c = point(b.x, b.y);           // closer to b
1571.         return dist(p, b);             // Euclidean distance between p and b
1572.     }
1573.     return distToLine(p, a, b, c); // run distToLine as above
1574. }
1575.
1576. // Returns the angle aob given three points: a, o, and b, (using dot product)
1577. double angle(point a, point o, point b) { // returns angle aob in rad
1578.     vec oa = toVec(o, a), ob = toVec(o, b);
1579.     return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob)));
1580. }
1581.
1582. double cross(vec a, vec b) { // Cross product of two vectors
1583.     return a.x * b.y - a.y * b.x; // note: to accept collinear points, we have
to change the '> 0'
1584. }
1585.
1586. bool ccw(point p, point q, point r) { // returns true if point r is on the left
side of line pq
1587.     return cross(toVec(p, q), toVec(p, r)) > 0;
1588. }
1589.
1590. bool collinear(point p, point q, point r) { // returns true if point r is on the same
line as the line pq
1591.     return fabs(cross(toVec(p, q), toVec(p, r))) < EPS;
1592. }
1593.
1594. // 2D Objects -----
1595. // ----- CIRCLE -----
1596. struct circle {
1597.     int x, y, r;
1598.     circle(int _x, int _y, int _r) {
1599.         x = _x;
1600.         y = _y;
1601.         r = _r;
1602.     }
1603.     double Area() {
1604.         return PI*r*r;
1605.     }
1606. };
1607.
1608. // Reference: https://www.mathsisfun.com/geometry/circle-sector-segment.html
1609. double CircleSegmentArea(double r, double theta) { // Circle Radius, Center
Angle(degree)
1610.     return r * r * 0.5 * (DEG_to_RAD(theta) - sin(DEG_to_RAD(theta)));
1611. }
1612. double CircleSectorArea(double r, double theta) { // Circle Radius, Center
Angle(degree)
1613.     return r * r * 0.5 * DEG_to_RAD(theta);
1614. }
1615. double CircleArcLength(double r, double theta) { // Circle Radius, Center
Angle(degree)

```

```

1616.     return r * DEG_to_RAD(theta);
1617. }
1618. bool doIntersectCircle(circle c1, circle c2) {
1619.     int dis = dist(point(c1.x, c1.y), point(c2.x, c2.y));
1620.     if(sqrt(dis) < c1.r+c2.r) return 1;
1621.     return 0;
1622. }
1623. bool isInside(circle c1, circle c2) { // Returns true if any one of the
circle is fully into another circle
1624.     int dis = dist(point(c1.x, c1.y), point(c2.x, c2.y));
1625.     return ((sqrt(dis) <= max(c1.r, c2.r)) and (sqrt(dis) + min(c1.r, c2.r) < max(c1.r,
c2.r)));
1626. }
1627. // Returns where a point p lies according to a circle of center c and radius r
1628. int insideCircle(point p, point c, int r) { // all integer version
1629.     int dx = p.x - c.x, dy = p.y - c.y;
1630.     int Euc = dx * dx + dy * dy, rSq = r * r; // all integer
1631.     return Euc < rSq ? 0 : Euc == rSq ? 1 : 2; // inside(0)/border(1)/outside(2)
1632. }
1633.
1634. // Given 2 points on the circle (p1 and p2) and radius r of the corresponding circle,
1635. // determine the location of the centers (c1 and c2) of the two possible circles
1636. bool circle2PtsRad(point p1, point p2, double r, point &c) {
1637.     double d2 = (p1.x - p2.x) * (p1.x - p2.x) + (p1.y - p2.y) * (p1.y - p2.y);
1638.     double det = r * r / d2 - 0.25;
1639.     if(det < 0.0) return false;
1640.     double h = sqrt(det);
1641.     c.x = (p1.x + p2.x) * 0.5 + (p1.y - p2.y) * h;
1642.     c.y = (p1.y + p2.y) * 0.5 + (p2.x - p1.x) * h;
1643.     return true; // to get the other center,
reverse p1 and p2
1644. }
1645.
1646. // ----- Triangle -----
1647. double TriangleArea(double AB, double BC, double CA) {
1648.     double s = (AB + BC + CA)/2.0;
1649.     return sqrt(s*(s-AB)*(s-BC)*(s-CA));
1650. }
1651. double getAngle(double AB, double BC, double CA) { // Returns the angle(IN RADIAN)
opposite of side CA
1652.     return acos((AB*AB + BC*BC - CA*CA)/(2*AB*BC));
1653. }
1654. double rInCircle(double ab, double bc, double ca) { // Returns radius of inCircle of a
triangle
1655.     return TriangleArea(ab, bc, ca) / (0.5 * (ab + bc+ ca));
1656. }
1657. int inCircle(point p1, point p2, point p3, point &ctr, double &r) {
1658.     r = rInCircle(p1, p2, p3);
1659.     if (fabs(r) < EPS) return 0; // no inCircle center
1660.     line l1, l2;
1661.     double ratio = dist(p1, p2) / dist(p1, p3); // compute these two angle bisectors
1662.     point p = translate(p2, scale(toVec(p2, p3), ratio / (1 + ratio)));
1663.     pointsToLine(p1, p, l1);
1664.     ratio = dist(p2, p1) / dist(p2, p3);
1665.     p = translate(p1, scale(toVec(p1, p3), ratio / (1 + ratio)));
1666.     pointsToLine(p2, p, l2);
1667.     areIntersect(l1, l2, ctr);
1668.     return 1;

```

```
1669. }
1670. // radius of Circle Outside of a Triangle
1671. double rCircumCircle(double ab, double bc, double ca) { // ab, ac, ad are sides of
triangle
1672.     return ab * bc * ca / (4.0 * TriangleArea(ab, bc, ca));
1673. }
1674. point CircumCircleCenter(point a, point b, point c, double &r) { // returns center and
radius of circumcircle
1675.     double ab = dist(a, b);
1676.     double bc = dist(b, c);
1677.     double ca = dist(c, a);
1678.     r = rCircumCircle(ab, bc, ca);
1679.     if(Equal(r, ab)) return point((a.x+b.x)/2, (a.y+b.y)/2);
1680.     if(Equal(r, bc)) return point((b.x+c.x)/2, (b.y+c.y)/2);
1681.     if(Equal(r, ca)) return point((c.x+a.x)/2, (c.y+a.y)/2);
1682.     line AB, BC;
1683.     pointsToLine(a, b, AB);
1684.     pointsToLine(b, c, BC);
1685.     line perpenAB = perpendicularLine(AB, point((a.x+b.x)/2, (a.y+b.y)/2));
1686.     line perpenBC = perpendicularLine(BC, point((b.x+c.x)/2, (b.y+c.y)/2));
1687.     point center;
1688.     areIntersect(perpenAB, perpenBC, center);
1689.     return center;
1690. }
1691.
1692. // ----- Trapezoid -----
1693. double TrapeziodArea(double a, double b, double c, double d) { // a and c are parallel
1694.     double BASE = fabs(a-c);
1695.     double AREA = TriangleArea(d, b, BASE);
1696.     double h = (AREA*2)/BASE;
1697.     return ((a+c)/2)*h;
1698. }
1699.
1700. // Hashing
1701. // p = 31, 51
1702. // MOD = 1e9+9, 1e7+7
1703. const ll p = 31;
1704. const ll mod1 = 1e9+9, mod2 = 1e9+7;
1705.
1706. // Returns Single Hash Val
1707. ll hash(char *s, int len, ll mod = 1e9+9) {
1708.     int p = 31;
1709.     ll hashVal = 0;
1710.     ll pPow = 1;
1711.     for(int i = 0; i < len; ++i) {
1712.         hashVal = (hashVal + (s[i] - 'a' + 1) * pPow)%mod;
1713.         pPow = (pPow * p)%mod;
1714.     }
1715.     return hashVal;
1716. }
1717. vll Hash(char *s, int len) {
1718.     ll hashVal = 0;
1719.     vector<ll>v;
1720.     for(int i = 0; i < len; ++i) {
1721.         hashVal = (hashVal + (s[i] - 'a' + 1) * Power[i])%mod;
1722.         v.push_back(hashVal);
1723.     }
1724.     return v;
```



```

1725. }
1726. bool MATCH(p1l a, p1l b) {
1727.     while(a.fi <= a.se) {
1728.         if(s1[a.fi] != s2[b.fi])
1729.             return 0;
1730.         a.fi++, b.fi++;
1731.     }
1732.     return 1;
1733. }
1734. void PowerGen(int n) {
1735.     Power.resize(n+1);
1736.     Power[0] = 1;
1737.     for(int i = 1; i < n; ++i)
1738.         Power[i] = (Power[i-1] * p)%mod;
1739. }
1740. ll SubHash(v1 &Hash, ll l, ll r, ll LIM) {
1741.     ll H;
1742.     H = (Hash[r] - (l-1 >= 0 ? Hash[l-1]:0) + mod)%mod;
1743.     H = (H * Power[LIM-1])%mod;
1744.     return H;
1745. }
1746.
1747. // ----- DOUBLE HASH GENERATORS -----
1748. // Generates Hash of entire string without PowerGen func
1749. vector<pair<ll, ll>> doubleHash(char *s, int len, ll mod1 = 1e9+7, ll mod2 = 1e9+9) {
1750.     ll hashVal1 = 0, hashVal2 = 0, pPow1 = 1, pPow2 = 1;
1751.     vector<pair<ll, ll>> v;
1752.     for(int i = 0; i < len; ++i) {
1753.         hashVal1 = (hashVal1 + (s[i] - 'a' + 1)* pPow1)%mod1;
1754.         hashVal2 = (hashVal2 + (s[i] - 'a' + 1)* pPow2)%mod2;
1755.         pPow1 = (pPow1 * p)%mod1;
1756.         pPow2 = (pPow2 * p)%mod2;
1757.         v.push_back({hashVal1, hashVal2});
1758.     }
1759.     return v;
1760. }
1761. void PowerGen(int n) {
1762.     Power.resize(n+1);
1763.     Power[0] = {1, 1};
1764.     for(int i = 1; i < n; ++i) {
1765.         Power[i].first = (Power[i-1].first * p)%mod1;
1766.         Power[i].second = (Power[i-1].second * p)%mod2;
1767.     }
1768. v1l doubleHash(char *s, int len) { // Returns Double Hash vector for a full string
1769.     ll hashVal1 = 0, hashVal2 = 0;
1770.     vector<p1l>v;
1771.     for(int i = 0; i < len; ++i) {
1772.         hashVal1 = (hashVal1 + (s[i] - 'a' + 1)* Power[i].fi)%mod1;
1773.         hashVal2 = (hashVal2 + (s[i] - 'a' + 1)* Power[i].se)%mod2;
1774.         v.push_back({hashVal1, hashVal2});
1775.     }
1776.     return v;
1777. }
1778. p1l SubHash(v1l &Hash, ll l, ll r, ll LIM) { // Produce SubString Hash
1779.     p1l H;
1780.     H.fi = (Hash[r].fi - (l-1 >= 0 ? Hash[l-1].fi:0) + mod1)%mod1;
1781.     H.se = (Hash[r].se - (l-1 >= 0 ? Hash[l-1].se:0) + mod2)%mod2;
1782.     H.fi = (H.fi * Power[LIM-1].fi)%mod1;

```

```

1783.     H.se = (H.se * Power[LIM-1].se)%mod2;
1784.     return H;
1785. }
1786. // Returns True if the Hashval of length len exists in subrange [l, r] of Hash vector
1787. bool MatchSubStr(int l, int r, vector<pll>&Hash, pll HashVal, int len) {
1788.     for(int Start = l, End = l+len-1; End <= r; ++End, ++Start) {
1789.         pll pattHash, strHash;
1790.         pattHash.first = (HashVal.first*Power[Start].first)%mod1;
1791.         pattHash.second = (HashVal.second*Power[Start].second)%mod2;
1792.         strHash.first = (Hash[End].first - (Start == 0 ? 0:Hash[Start-1].first) +
mod1)%mod1;
1793.         strHash.second = (Hash[End].second - (Start == 0 ? 0:Hash[Start-1].second) +
mod2)%mod2;
1794.         if(strHash == pattHash) return 1;
1795.     }
1796.     return 0;
1797. }
1798.
1799. // Heavy Light Decompose + Segment Tree
1800. // Tree node value update, Tree node distance
1801.
1802. int parent[MAX], level[MAX], nextNode[MAX], chain[MAX], num[MAX], val[MAX], numToNode[MAX],
top[MAX], ChainSize[MAX], mx[MAX];
1803. int ChainNo = 1, all = 1, n;
1804. vi G[MAX];
1805. void dfs(int u, int Parent) {
1806.     parent[u] = Parent;           // Parent of u
1807.     ChainSize[u] = 1;           // Number of child (initially the size is 1, contains
only 1 node. itself) (resued array in hld)
1808.     for(int i = 0; i < SIZE(G[u]); ++i) {
1809.         int v = G[u][i];
1810.         if(v == Parent)           // if the connected node is parent, skip
1811.             continue;
1812.         level[v] = level[u]+1;     // level of the child node is : level of
parent node + 1
1813.         dfs(v, u);
1814.         ChainSize[u] += ChainSize[v]; // Modify this line if max Chain is needed
1815.         if(nextNode[u] == -1 || ChainSize[v] > ChainSize[nextNode[u]])
1816.             nextNode[u] = v;       // next selected node of u (select the node which
has more child, (HEAVY))
1817.     }
1818. void hld(int u, int Parent) {
1819.     chain[u] = ChainNo;           // Chain Number
1820.     num[u] = all++;              // Numbering all nodes
1821.     if(ChainSize[ChainNo] == 0)   // if this is the first node
1822.         top[ChainNo] = u;         // mark this as the root node of the n'th chain
1823.     ChainSize[ChainNo]++;
1824.     if(nextNode[u] != -1)         // if this node has a child, go to it
1825.         hld(nextNode[u], u);      // the next node is included in the chain
1826.     for(int i = 0; i < SIZE(G[u]); ++i) {
1827.         int v = G[u][i];         // if this node is parent node or, this node is
already included in the chain, skip
1828.         if(v == Parent || v == nextNode[u]) continue;
1829.         ChainNo++;               // this is a new (light) chain, so increment the
chain no. counter
1830.         hld(v, u);
1831.     }
1832. int GetSum(int u, int v) {

```

```

1833.     int res = 0;
1834.     while(chain[u] != chain[v]) {                                     // While two nodes are not in
same chain
1835.         if(level[top[chain[u]]] < level[top[chain[v]]])           // u is the chain which's
topmost node is deeper
1836.             swap(u, v);
1837.             int start = top[chain[u]];
1838.             res += query(1, 1, n, num[start], num[u]);             // Run query in u node's chain
1839.             u = parent[start];                                       // go to the upper chain of u
1840.         }
1841.         if(num[u] > num[v]) swap(u, v);
1842.         res += query(1, 1, n, num[u], num[v]);
1843.         return res;
1844.     }
1845. void updateNodeVal(int u, int val) {
1846.     update(1, 1, n, num[u], val);                                     // Updating the value of chain
1847. }
1848. void numToNodeConv(int n) {
1849.     for(int i = 1; i <= n; ++i) numToNode[num[i]] = i;
1850. }
1851. int main() {
1852.     memset(nextNode, -1, sizeof nextNode);
1853.     ChainNo = 1, all = 1;
1854.     dfs(1, 1);
1855.     memset(ChainSize, 0, sizeof ChainSize);                         // array reused in hld
1856.     hld(1, 1);
1857.     numToNodeConv(n);
1858.     init(1, 1, n);
1859. }
1860.
1861. // Interval Sum
1862. // Complexity: query*log(query)
1863. // http://codeforces.com/contest/915/problem/E
1864.
1865. struct Interval {
1866.     set<pair<ll, ll> >Set;                                           // Contains Segment Endpoints {r, l}
1867.     map<pair<ll, ll>, ll>Val;                                         // Contains Segment Values {l, r} = k
1868.     int TOTlen;                                                       // Contains Total Segment Covered Length
1869.     void init(int sz = -1) {
1870.         Set.clear(), Val.clear(), TOTlen = 0;
1871.         if(sz > 0)                                                     // Will be initialized if size declared (NOT
needed)
1872.             Set.insert(make_pair(sz, 1)), Val[make_pair(1, sz)] = 0;
1873.     }
1874.     void Insert(ll l, ll r, ll val) {
1875.         set<pair<ll, ll> > :: iterator it = Set.lower_bound({l, 0LL});
1876.         while(it != Set.end() && it->second <= r) {
1877.             ll segL = it->second, segR = it->first;                   // Overlapped segment
1878.             Set.erase(it++);                                           // Erase and point to the
next segment
1879.             ll L = max(segL, l), R = min(segR, r);                   // Erased segment's partial
L and R
1880.             TOTlen -= R-L+1;
1881.             if(segL < l) {
1882.                 Set.insert({l-1, segL});
1883.                 Val[{segL, l-1}] = Val[{segL, segR}];
1884.             }
1885.             if(segR > r) {

```

```

1886.         Set.insert({segR, r+1});
1887.         Val[{r+1, segR}] = Val[{segL, segR}];
1888.     }
1889.     Val.erase({segL, segR});
1890. }
1891. TOTlen += r-l+1;
1892. Set.insert(make_pair(r, l));
1893. Val[make_pair(l, r)] = val;
1894. }
1895. ll getSum(ll l, ll r) {
1896.     ll sum = 0;
1897.     set<pair<ll, ll> > :: iterator it = Set.lower_bound({l, 0LL});
1898.     while(it != Set.end() && it->second <= r) {
1899.         ll segL = it->second, segR = it->first;           // Overlapped segment
1900.         ll V = Val[{segL, segR}];
1901.         sum += (segR-segL+1) * V;
1902.         if(segL < l) sum -= (l-segL)*V;
1903.         if(segR > r) sum -= (segR-r)*V;
1904.         ++it;
1905.     }
1906.     return sum;
1907. };
1908. vector<ll> CountInterval(int n) // returns number of overlaps of all inclusive points
1909.     vector<pair<ll, int> >v;    // segments start/end and marker (segments are l - r
inclusive)
1910.     vector<ll>ret(n+1);        // returns : ret[number_of_overlaps] =
total_number_of_points
1911.     ll l, r;
1912.     while(n-- > 0) {
1913.         cin >> l >> r;           // input
1914.         v.push_back({l, 1}), v.push_back({r+1, -1});    // r+1 as l - r is segment boundary
1915.     }
1916.     sort(v.begin(), v.end());
1917.     for(int i = 0, cnt = v[0].second; i < (int)v.size(); ++i, cnt += v[i].second) // cnt
contains the overlaps
1918.         if(i+1 < (int)v.size() and v[i].first != v[i+1].first) // v[i].first ==
v[i+1].first then
1919.             ret[cnt] += v[i+1].first - v[i].first // there may exist more points at front,
so take them first
1920.     return ret;
1921. }
1922.
1923. // Knuth Morris Pratt
1924. // Complexity : O(String + Token)
1925.
1926. //-----Genuine PrefixTable (Prefix-Suffix Length)-----
1927. // Some Tricky Cases:   aaaaaa : 0 1 2 3 4 5           aaaabaa : 0 1 2 3 0 1 2           abcdabcd : 0
0 0 0 1 2 3 4
1928. void prefixTable(int n, char pat[], int table[]) {
1929.     int len = 0, i = 1;           // length of the previous longest prefix suffix
1930.     table[0] = 0;                 // table[0] is always 0
1931.     while (i < n) {
1932.         if pat[i] == pat[len]) {
1933.             len++;
1934.             table[i] = len;
1935.             i++;
1936.         }
1937.         else {                     // pat[i] != pat[len]

```

```

1938.         if(len != 0)    len = table[len-1];           // find previous match
1939.         else             table[i] = 0, i++;           // if (len == 0) and
mismatch
1940.     }}}                                           // set table[i] = 0, and
go to next index
1941.
1942. void KMP(int strLen, int patLen, char str[], char pat[], int table[]) {
1943.     int i = 0, j = 0;                               // i : string index, j : pattern index
1944.     while (i < N) {
1945.         if(str[i] == pat[j]) i++, j++;
1946.         if(j == M) {
1947.             printf("Found pattern at index %d n", i-j);
1948.             j = table[j-1];                          // Match found, try for next match
1949.         }
1950.         else if(i < strLen && str[i] != pat[j]) {     // Match not found
1951.             if(j != 0) j = table[j-1];               // if j != 0, then go to the prev
match index
1952.             else i = i+1;                            // if j == 0, then we need to go to
next index of str
1953.         }}}
1954.
1955. // ----- 2D KMP -----
1956.
1957. unordered_map<string, int>patt;                      // Clear after each Kmp2D call
1958. int flag = 0;                                       // Set to zero before calling PrefixTable
1959. // r : Pattern row, c : Pattern column             // table : prefix table (1D array)
1960. // s : Pattern String (C++ string)                 // Followed Felix-Halim KMP
1961. vector<int> PrefixTable2D(int r, int c, int table[], string s[]) {
1962.     vector<int>Row;                                // Contains Row mapped string index
1963.     for(int i = 0; i < r; ++i) {
1964.         if(patt.find(s[i]) == patt.end()) {
1965.             patt[s[i]] = ++flag;
1966.             Row.push_back(flag);
1967.         }
1968.         else Row.push_back(patt[s[i]]);
1969.     }
1970.     table[0] = -1;
1971.     int i = 0, j = -1;
1972.     while(i < r) {
1973.         while(j >= 0 && Row[i] != Row[j])
1974.             j = table[j];
1975.         ++i, ++j;
1976.         table[i] = j;
1977.     }
1978.     return Row;                                     // Returns Hashed index of each row in pattern string
1979. }
1980.
1981. // StrR StrC : String Row and Column                // PattR PattC : Pattern row and column
1982. // Str : String (C++ String)                        // Patt : Pattern (C++ String)      // table :
Prefix table of pattern (1D array)
1983. vector<pair<int, int> > Kmp2D(int StrR, int StrC, int PattR, int PattC, string Str[], string
Patt[], int table[]) {
1984.     int mat[StrR][StrC];
1985.     int limC = StrC - PattC;
1986.     vector<int>PattRow = PrefixTable2D(PattR, PattC, table, Patt);
1987.     for(int i = 0; i < StrR; ++i)
1988.         for(int j = 0; j <= limC; ++j) {
1989.             string tmp = Str[i].substr(j, PattC);

```

```

1990.         if(patt.find(tmp) == patt.end()) {           // Generating String Mapped using
same mapping values
1991.             patt[tmp] = ++flag;                       // Stored in matrix
1992.             mat[i][j] = flag;
1993.         }
1994.         else mat[i][j] = patt[tmp];
1995.     }
1996.     vector<pair<int, int> >match;                         // This will contain the starting
Row & Column of matched string
1997.     for(int c = 0; c <= limC; ++c) {                     // Scan columnwise
1998.         int i = 0, j = 0;
1999.         while(i < StrR) {
2000.             while(j >= 0 && mat[i][c] != PattRow[j])
2001.                 j = table[j];
2002.             ++i, ++j;
2003.             if(j == PattR) match.push_back(make_pair(i-j,c));
2004.         }}
2005.     return match;
2006. }
2007.
2008. // LCA
2009. // Least Common Ancestor with sparse table
2010.
2011. v1 G[MAX], W[MAX];
2012. int level[MAX], parent[MAX], sparse[MAX][20];
2013. ll dist[MAX], DIST[MAX][20];
2014.
2015. void dfs(int u, int par, int lvl, ll d) {                 // Tracks distance as well (From root 1
to all nodes)
2016.     level[u] = lvl;                                       // parent[] and level[] is necessary
2017.     parent[u] = par;
2018.     dist[u] = d;                                          // remove distance if not needed
2019.     for(int i = 0; i < (int)G[u].size(); ++i)
2020.         if(parent[u] != G[u][i])
2021.             dfs(G[u][i], u, lvl+1, d+W[u][i]);
2022. }
2023.
2024. void LCAinit(int V) {
2025.     memset(parent, -1, sizeof parent);
2026.     dfs(0, -1, 0);                                       // DFS first
2027.     memset(sparse, -1, sizeof sparse);                   // Main initialization of sparse table
LCA starts here
2028.     for(int u = 1; u <= V; ++u)                           // node u's 2^0 parent
2029.         sparse[u][0] = parent[u];
2030.     for(int p = 1, v; (1LL<<p) <= V; ++p)
2031.         for(int u = 1; u <= V; ++u)
2032.             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) ]
2033.                 sparse[u][p] = sparse[v][p-1];
2034. }
2035.
2036. int LCA(int u, int v) {
2037.     if(level[u] > level[v]) swap(u, v);                 // v is deeper
2038.     int p = ceil(log2(level[v]));
2039.
2040.     for(int i = p ; i >= 0; --i)                         // Pull up v to same level as u
2041.         if(level[v] - (1LL<<i) >= level[u])
2042.             v = sparse[v][i];

```

```

2043.     if(u == v) return u;                                // if u WAS the parent
2044.
2045.     for(int i = p; i >= 0; --i)                            // Pull up u and v
together while LCA not found
2046.         if(sparse[v][i] != -1 && sparse[u][i] != sparse[v][i])    // -1 check is if 2^i is
out of calculated range
2047.             u = sparse[u][i], v = sparse[v][i];
2048.     return parent[u];
2049. }
2050.
2051. // ----- LCA WITH DISTANCE -----
2052. void distDP(int V) {                                        // initialiser for LCA_with_DIST, call after
LCAinit()
2053.     for(int u = 1; u <= V; ++u)                            // NOTE : DIST[u][0] = weight of node u
2054.         DIST[u][0] = W[u];                                // Where W[u] = weight of node u
2055.     for(int p = 1; (1<<p)<=V; ++p)
2056.         for(int u = 1; u <=V; ++u) {
2057.             int v = sparse[u][p-1];
2058.             if(v == -1) continue;
2059.             DIST[u][p] += DIST[u][p-1] + DIST[v][p-1];
2060.         }
2061.
2062. int LCA_with_DIST(int u, int v, long long &w) {            // w returns distance from u -> v
2063.     w = 0;
2064.     if(level[u] > level[v]) swap(u, v);                    // v is deeper
2065.     int p = ceil(log2(level[v]));
2066.     for(int i = p; i >= 0; --i)                            // Pull up v to same level as u
2067.         if(level[v] - (1LL<<i) >= level[u]) {
2068.             w += DIST[v][i];
2069.             v = sparse[v][i];
2070.         }
2071.     if(u == v) {                                            // if u WAS the parent
2072.         w += DIST[v][0];
2073.         return u;
2074.     }
2075.     for(int i = p; i >= 0; --i)                            // Pull up u and v together while LCA
not found
2076.         if(sparse[v][i] != -1 && sparse[u][i] != sparse[v][i])    // -1 check is if 2^i is
out of calculated range
2077.             u = sparse[u][i], v = sparse[v][i];
2078.     w += DIST[v][0];
2079.     w += DIST[u][0];
2080.     w += DIST[sparse[v][0]][0];
2081.     return parent[u];
2082. }
2083.
2084. ll Distance(int u, int v) {
2085.     int lca = LCA(u, v);
2086.     return dist[v] + dist[u] - 2*dist[lca];
2087. }
2088.
2089. // ----- LCA WITH Sparse Table Vector -----
2090. // DFS and LCA INIT is same
2091. void MERGE(vector<int>&u, vector<int>&v) {                  // Do what is to be done to merge
2092.     for(auto it : v) u.push_back(it);                    // here taking lowest 10 values
2093.     sort(u.begin(), u.end());
2094.     while((int)u.size() > 10)
2095.         u.pop_back();

```

```

2096. }
2097.
2098. vector<int> W[MAX][20]; // W[u][0] will contain initial weight/weights at node u
2099. vector<int> LCA(int u, int v) {
2100.     vector<int> T;
2101.     if(level[u] > level[v]) swap(u, v); // v is deeper
2102.     int p = ceil(log2(level[v]));
2103.     for(int i = p ; i >= 0; --i) // Pull up v to same level as u
2104.         if(level[v] - (1LL<<i) >= level[u]) {
2105.             MERGE(T, W[v][i]);
2106.             v = sparse[v][i];
2107.         }
2108.     if(u == v) { // if u WAS the parent
2109.         MERGE(T, W[u][0]);
2110.         return T;
2111.     }
2112.     for(int i = p; i >= 0; --i) // Pull up u and v
2113.         together while LCA not found
2114.         if(sparse[v][i] != -1 && sparse[u][i] != sparse[v][i]) { // -1 check is if
2115.             2^i is out of calculated range
2116.             MERGE(T, W[u][i]);
2117.             MERGE(T, W[v][i]);
2118.             u = sparse[u][i], v = sparse[v][i];
2119.         }
2120.     MERGE(T, W[u][0]); // As W[x][0] denoted the x nodes weight
2121.     MERGE(T, W[v][0]); // every sparse node must be calculated
2122.     MERGE(T, W[sparse[v][0]][0]); // we can also calculate summation of distance like this
2123.     return T;
2124. }
2125.
2126. // Longest Increasing/Decrasing Sequence
2127. // Complexity : nLog_n
2128.
2129. // -----Non Printable Version-----
2130.
2131. int main() {
2132.     // vector v contains the sequence
2133.     for(auto it : v) { // Use -it for decrasing
2134.         sequences
2135.         auto pIT = upper_bound(LIS.begin(), LIS.end(), it); // Longest Non-Decreasing
2136.         Sequence
2137.         if(pIT == LIS.end()) // For Longest Increasing
2138.             Sequence use lower_bound
2139.             LIS.push_back(it);
2140.         else
2141.             *pIT = it;
2142.     }
2143.     return 0;
2144. }
2145.
2146. // -----Printable Version-----
2147. // DP + BinarySearch (nLog_n)
2148. // {1, 1, 9, 3, 8, 11, 4, 5, 6, 6, 4, 19, 7, 1, 7}
2149. // Incrasing : 1, 3, 4, 5, 6, 7
2150. // NonDecreasing : 1, 1, 3, 4, 5, 6, 6, 7, 7
2151.
2152. void findLIS(vector<int> &v, vector<int> &idx) { // v is the input values and idx vector
2153.     contains index of the LIS values

```



```

2148.     if(v.empty()) return;
2149.     vector<int> dp(v.size());           // The memoization part, remembers what index is
the previous index if any value is inserted or modified
2150.     idx.push_back(0);                   // Carrys index of values
2151.     int l, r;
2152.
2153.     for(int i = 1; i < (int)v.size(); i++) {
2154.         if(v[idx.back()] <= v[i]) {     // **Replace < with <= if non-decreasing
subsequence required
2155.             dp[i] = idx.back();         // If next element v[i] is greater than last
element of
2156.             idx.push_back(i);           // current longest subsequence v[idx.back()],
just push it at back of "idx" and continue
2157.             continue;
2158.         }
2159.         // Binary search to find the smallest element referenced by idx which is just bigger
than v[i] (UpperBound(v[i]))
2160.         // Note : Binary search is performed on idx (and not v)
2161.         for(l = 0, r = idx.size()-1; l < r; ) {
2162.             int mid = (l+r)/2;
2163.             if(v[idx[mid]] <= v[i]) l = mid+1; // **Replace < with <= if non-
decreasing subsequence required
2164.             else r = mid;
2165.         }
2166.         if(v[i] < v[idx[l]]) {           // Update idx if new value is smaller
then previously referenced value
2167.             if(l > 0) dp[i] = idx[l-1];
2168.             idx[l] = i;
2169.         }
2170.     }
2171.     for(l = idx.size(), r = idx.back(); l--; r = dp[r])
2172.         idx[l] = r;
2173. }
2174.
2175. // Math Formulas
2176.
2177. // Find the number of b for which [b1, b2] | [a1, a2]
2178. int FindDivisorInRange(int a1, int a2, int b1, int b2) {
2179.     int a = abs(a1 - a2);
2180.     int b = abs(b1 - b2);
2181.     int gcd = __gcd(a, b);
2182.     return 1 + gcd;
2183. }
2184.
2185. // Find how many integers from range m to n are divisible by a or b
2186. int rangeDivisor(int m, int n, int a, int b) {
2187.     int lcm = LCM(a, b);
2188.     int a_divisor = n / a - (m - 1) / a;
2189.     int b_divisor = n / b - (m - 1) / b;
2190.     int common_divisor = n / lcm - (m - 1) / lcm;
2191.     int ans = a_divisor + b_divisor - common_divisor;
2192.     return ans;
2193. }
2194.
2195. ll CSOD(ll n) {                         // Cumulative Sum of Divisors in sqrt(n)
2196.     ll ans = 0;
2197.     for(ll i = 2; i * i <= n; ++i) {
2198.         ll j = n / i;

```

```

2199.         ans += (i + j) * (j - i + 1) / 2;
2200.         ans += i * (j - i);
2201.     }
2202.     return ans;
2203. }
2204.
2205. int CountDivisible(int a, int b, int n) {    // Returns the number of divisible value in
range [a, b] by n (NOT TESTED)
2206.     if(a > b) swap(a, b);
2207.     a += n - a%n;
2208.     b -= b%n;
2209.     if(a > b) return 0;
2210.     return ceil((b-a+1)/(double)n);
2211. }
2212.
2213. int FactorialCount(int n, int p = 5) {        // Returns how many value of p is present in n!
2214.     int ret = 0, r = p;                      // returns number of trailing zero of n! if p =
5
2215.     while(n/r != 0) {
2216.         ret += n/r;
2217.         r *= p;
2218.     }
2219.     return ret;
2220. }
2221.
2222. int TrailingZero(int n, int p = 1) {          // Returns Trailing Zero of n^p
2223.     int cnt = 0;                             // Trailing Zero for any number :
min(count_2_as_prime_factor, count_5_as_prime_factor)
2224.     while(n%5 == 0 && n%2 == 0)
2225.         n /= 5, n /= 2, ++cnt;
2226.     return cnt*p;
2227. }
2228.
2229. int BirthdayParadox(int days, int targetPercent = 50) {        // Returns Number of people
required so that probability is >= target
2230.     int people = 0;                                           // Formula : 1 - (365/365) *
(364/365) * (363/365) * .....
2231.     double percent = targetPercent/100.0, gotPercent = 1;
2232.     for( ; gotPercent > percent; ++people)
2233.         gotPercent *= (days-people-1)/(double)days;
2234.     return people;
2235. }
2236.
2237. /* Euler's Totient function  $\phi(n)$  for an input n is count of numbers in {1, 2, 3, ..., n}
2238.  * that are relatively prime to n, i.e., the numbers whose GCD (Greatest Common Divisor)
with n is 1.
2239.  * Phi(4) : GCD(1, 4) = 1, GCD(3, 4)
2240.  * so, Phi(4) = 2
2241.  */
2242.
2243. int Phi(int n) {                                              // Computes phi of n
2244.     int result = n;
2245.     for(int p=2; p*p<=n; ++p) {                             // Consider all prime factors of n and subtract
their multiples from result
2246.         if(n % p == 0) {                                     // p is a prime factor of n
2247.             while (n % p == 0)                             // eliminate all p factors from n
2248.                 n /= p;
2249.             result -= result / p;

```

```

2250.     }}
2251.     if(n > 1)                                // if n is still greater than 1, then it is also a
prime
2252.         result -= result / n;
2253.     return result;
2254. }
2255.
2256. long long phi[MAX];
2257. void computeTotient(int n) {                  // Computes phi or Euler Phi 1 to n
2258.     for (int i=1; i<=n; i++)                 // Initialize
2259.         phi[i] = i;
2260.     for (int p=2; p<=n; p++) {               // Computation
2261.         if (phi[p] == p) {                   // if phi is not computed
2262.             phi[p] = p-1;                   // p is prime and phi(prime) = prime-1;
2263.             for (int i = 2*p; i<=n; i += p) { // Sieve like implementation
2264.                 phi[i] = (phi[i]/p) * (p-1); // Add contribution of p to its multiple
i by multiplying with (1 - 1/p)
2265.             }}}
2266.
2267. // Combination
2268. // Complexity O(k)
2269. long long C(int n, int k) {
2270.     long long c = 1;
2271.     if(k > n - k)
2272.         k = n-k;
2273.     for(int i = 0; i < k; i++) {
2274.         c *= (n-i);
2275.         c /= (i+1);
2276.     }
2277.     return c;
2278. }
2279.
2280. ll fa[MAX], fainv[MAX];                      // fa and fainv must be in global
2281. ll C(ll n, ll r) {                          // Usable if MOD value is present
2282.     if(fa[0] == 0) {                         // Auto initialize
2283.         fa[0] = 1, fainv[0] = powerMOD(1, MOD-2);
2284.         for(int i = 1; i < MAX; ++i) {
2285.             fa[i] = (fa[i-1]*i) % MOD;        // Constant MOD
2286.             fainv[i] = powerMOD(fa[i], MOD-2);
2287.         }
2288.         if(n < 0 || r < 0 || n-r < 0) return 0; // Exceptional Cases
2289.         return ((fa[n] * fainv[r])%MOD * fainv[n-r])%MOD;
2290.     }
2291.
2292. ll Catalan(int n) { // Cat(n) = C(2*n, n)/(n+1);
2293.     ll c = C(2*n, n);
2294.     return c/(n+1);
2295. }
2296.
2297. // Building Pascle C(n, r)
2298. ll p[MAX][MAX];
2299. void buildPascle() {                         // This Contains values of nCr : p[n][r]
2300.     p[0][0] = 1;
2301.     p[1][0] = p[1][1] = 1;
2302.     for(int i = 2; i <= 50; i++)
2303.         for(int j = 0; j <= i; j++) {
2304.             if(j == 0 || j == i)
2305.                 p[i][j] = 1;

```

```

2306.         else
2307.             p[i][j] = p[i-1][j-1] + p[i-1][j];
2308.     }}
2309.
2310. ll C(int n, int r) {
2311.     if (r<0 || r>n) return 0;
2312.     return p[n][r];
2313. }
2314.
2315. // STARS AND BARS THEOREM / Ball and Urn theorem
2316. // If We have to Make  $x_1+x_2+x_3+x_4 = 12$ 
2317. // Then, the solution can be expressed as :  $\{ * | ***** | ***** | ** \} = \{ 1+5+4+2 \}, \{ | ***** | *** | ***** \}$ 
2318. // =  $\{ 0+5+3+4 \}$ 
2319. // The summation is presented as total value, and the stars represented as 1, we use bars to
2320. // separate values
2321. // Number of ways we can produce the summation n, with k unknowns :  $C(n+k-1, n) = C(n+k-1, k-1)$ 
2322. // If numbers have lower limits, like  $x_1 \geq 3, x_2 \geq 2, x_3 \geq 1, x_4 \geq 1$  (Let, the lower
2323. // limits be l[i])
2324. // Then the solution is :  $C(n-l_1-l_2-l_3-l_4+k-1, k-1)$ 
2325. // Ball & Urn : how many ways you can put 1 to n number in k sized array so that there are
2326. // non decreasing?
2327. ll StarsAndBars(vector<int> &l, int n, int k) {
2328.     if(!l.empty()) for(int i = 0; i < k; ++i) n -= l[i]; // If l is empty, then there
2329.     // is no lower limit
2330.     return C(n+k-1, k-1);
2331. }
2332. // If numbers have both boundaries  $l_1 \leq x_1 \leq r_1, l_2 \leq x_2 \leq r_2$ , and  $x_1+x_2 = N$ 
2333. // then we can reduce the form to  $x_1+x_2 = N-l_1-l_2$  and then x only gets upper limit  $x_1 \leq r_1-l_1+1, x_2 \leq r_2-l_2+1$ 
2334. // let  $r_1-l_1+1$  be new  $l_1$ , and  $r_2-l_2+1$  be new  $l_2$ , so  $x_1 \leq l_1$  and  $x_2 \leq l_2$ , this limit is the
2335. // opposite of basic Stars
2336. // and Bars theorem, according to Principle of Inclusion Exclusion, this answer can be found
2337. // as
2338. // Answer =  $C(n+k-1, k-1) - C(n-l_1+k-1, k-1) - C(n-l_2+k-1, k-1) + C(n-l_1-l_2+k-1, k-1) \dots$ 
2339. ll StarsAndBarsInRange(ll l[], ll r[], ll n, ll k) {
2340.     ll d[k+10], p[(1<<k) + 10];
2341.     for(int i = 0; i < k; ++i) {
2342.         d[i] = r[i] - l[i] + 1;
2343.         n -= l[i];
2344.     }
2345.     ll ret = C(n+k-1, k-1); p[0] = 0;
2346.     for(int i = 0; i < k; ++i) // Optimized Complexity :
2347.         2^n
2348.         for(int mask = (1<<i); mask < (1<<(i+1)); ++mask) {
2349.             p[mask] = p[mask ^ (1<<i)] + d[i];
2350.             ret += C(n-p[mask]+k-1, k-1) * (__builtin_popcount(mask)&1 ? -1:1);
2351.             ret %= MOD;
2352.         }
2353.     return (ret+MOD)%MOD;
2354. }

```

```

2353. vll GetSameMOD(vector<ll>&v) {          // Given an array v, find values k (k > 1), for which
v[0]%k = v[1]%k ... = v[n]%k
2354.     ll gcd;                          // If a number K, leaves the same remainder with 2
numbers, then it must divide their difference.
2355.     sort(v.begin(), v.end());          // Find all numbers K which divide all the consecutive
differences of all elements in the array.
2356.
2357.     for(int i = 0; i+1 < (int)v.size(); ++i) {      // And it we will take the GCD of all
consecutive differences
2358.         if(i == 0) gcd = v[i+1] - v[i];
2359.         else      gcd = __gcd(gcd, v[i+1] - v[i]);
2360.     }
2361.     vector<ll> ret = Divisors(gcd);          // GCD is the maximum value of k
2362.     ret.push_back(gcd);                     // All other values are the divisors of k
2363.     sort(ret.begin(), ret.end());           // NOTE : 1 is not added in the answer
2364.     return ret;
2365. }
2366.
2367. ll CountZerosInRangeZeroTo(string n) {          // Returns number of zeros from 0 to n
2368.     ll x = 0, fx = 0, gx = 0;
2369.     for(int i = 0; i < (int)n.size(); ++i){
2370.         ll y = n[i] - '0';
2371.         fx = 10LL * fx + x - gx * (9LL - y);      // Our formula
2372.         //fx += MOD;                               // If ans is to be returned in moded
value
2373.         //fx %= MOD;
2374.         x = 10LL * x + y;                          // Now calculate the new x and g(x)
2375.         //x %= MOD;
2376.         if(y == 0LL) gx++;
2377.     }
2378.     return fx+1;
2379. }
2380.
2381. ll NumOfSameValueInCombination(int n, int r) {      // Returns number of same value in a
set of nCr combination
2382.     if(n < r) return 0;
2383.     return C(n-1, r-1);
2384. }
2385.
2386. int cnt[MAX];          // cnt[x] : how many times x occurs
in input
2387. vector<int> genGCD(int mx) {          // Counts how many number are there
of gcd x
2388.     vector<int> sameGCD(mx+1);        // input the MAXIMUM value
2389.     for(int gcd = mx; gcd >= 2; --gcd) {      // Complexity : mx log_mx
2390.         int gcdCNT = cnt[gcd];
2391.         for(int mul = gcd+gcd; mul <= mx; mul += gcd)
2392.             gcdCNT += cnt[mul];
2393.         sameGCD[gcd] = gcdCNT;
2394.     }
2395.     return sameGCD;
2396. }
2397.
2398. // Multinomial :  $nC(k_1, k_2, k_3, \dots, k_m)$  is such that  $k_1 + k_2 + k_3 + \dots + k_m = n$  and  $k_i \neq k_j$  both can be possible.
2399. // Here Multinomial can be described as :  $nC(k_1, k_2, \dots, k_m) = nC_{k_1} * (n-k_1)C_{k_2} * (n-k_1-k_2)C_{k_3} * \dots * (n-k_1-k_2-\dots)C_{k_m}$ 
2400. // Let  $(a+b+c)^3 = a^3 + b^3 + c^3 + 3a^2b + 3b^2a + 3a^2c + 3c^2a + 3b^2c + 3c^2b + 6abc$ 

```

```

2401. // The coefficient can be retrieved as :  $6abc = 3C(1, 1, 1) = 6$        $3b^2c = 3C(0, 2, 1) = 3$ 
2402. // In general terms it tells how many ways we can place  $k_1, k_2, k_3, k_4$  people in 3 unique
      teams such that  $k_1+k_2+k_3$ 
2403. // NOTE: if  $k_1=k_2=k_3 = 2$  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
2404.
2405. ll fa[MAX] = {0}; // fa and fainv must be in global
2406. ll Multinomial(ll N, vector<ll>& K) { // K contains all  $k_1, k_2, k_3$ , if  $k_1=k_2=k_3$ , then
      just push  $k_1$  once!!
2407.     if(fa[0] == 0) { // Auto initialize
2408.         fa[0] = 1; //fainv[0] = powerMOD(1, MOD-2);
2409.         for(int i = 1; i < MAX; ++i) {
2410.             fa[i] = (fa[i-1]*i) % MOD; // Constant MOD
2411.             //fainv[i] = powerMOD(fa[i], MOD-2); // Use factorial inverse if required
2412.         }
2413.         ll k = 1;
2414.         if((int)K.size() == 1) k = powerMOD(fa[K[0]], N/K[0]); //  $k_1 = k_2 = \dots = k_m$ , so
      k occurs N/K time
2415.         else for(auto it : K) k = (k*fa[it])%MOD;
2416.         return (fa[N]*powerMOD(k, MOD-2))%MOD; // Inverse mod
2417.     }
2418.
2419. ll NumOfWaysToPlace(ll N, ll K) { // Number of ways to make N/K teams from N
      people so that each team contains K people
2420.     vector<ll>v; // If N = 6, then 1,2|3,4|5,6 and 3,4|1,2|5,6
      is considered same
2421.     v.push_back(K);
2422.     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
2423. }
2424.
2425. ull 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
2426.     ull ans = f[n]; // Factorial of n!
2427.     for(int i = 1; i <= r; ++i) { // Formula:  $n! - C(n, 1)*(n-1)! + C(n, 2)*(n-2)! - \dots$ 
      +  $(-1)^r * C(n, r)*(n-r)!$ 
2428.         if(i & 1) ans = (ans%MOD - (C(r, i) * f[n-i])%MOD)%MOD; // Here  $C(r, i)$  is because
      we only have to choose from r elements, not n elements
2429.         else ans = (ans%MOD + (C(r, i) * f[n-i])%MOD)%MOD;
2430.         ans = (ans + MOD)%MOD;
2431.     }
2432.     return ans%MOD;
2433. }
2434.
2435. struct matrix {
2436.     matrix() { memset(mat, 0, sizeof(mat)); }
2437.     long long mat[MAXN][MAXN];
2438. };
2439. matrix mul(matrix a, matrix b, int p, int q, int r) { //  $O(n^3)$  ::  $r_1, c_1, c_2$  [ $c_1 =$ 
       $r_2$ ]
2440.     matrix ans;
2441.     for(int i = 0; i < p; ++i)
2442.         for(int j = 0; j < r; ++j) {
2443.             ans.mat[i][j] = 0;
2444.             for(int k = 0; k < q; ++k)
2445.                 ans.mat[i][j] = (ans.mat[i][j]%MOD + (a.mat[i][k]%MOD * b.mat[k]
      [j]%MOD)%MOD)%MOD;
2446.         }

```

```

2447.     return ans;
2448. }
2449. matrix matPow(matrix base, ll p, int s) { // O(logN), s : size of square
matrix
2450.     if(p == 1) return base;
2451.     if(p & 1) return mul(base, matPow(base, p-1, s), s, s, s);
2452.     matrix tmp = matPow(base, p/2, s);
2453.     return mul(tmp, tmp, s, s, s);
2454. }
2455.
2456. // MaxFlow
2457. // Ford-Fulkerson
2458. // Complexity: O(VE^2)
2459. // Graph Type : Directed/Undirected
2460.
2461. const int MAX = 120;
2462. vector<int>edge[MAX];
2463. int V, E, rG[MAX][MAX], parent[MAX];
2464.
2465. bool bfs(int s, int d) { // augment path : source, destination
2466.     memset(parent, -1, sizeof parent);
2467.     queue<int>q;
2468.     q.push(s);
2469.     while(!q.empty()) {
2470.         int u = q.front();
2471.         q.pop();
2472.         for(auto v : edge[u])
2473.             if(parent[v] == -1 && rG[u][v] > 0) {
2474.                 parent[v] = u;
2475.                 if(v == d) return 1;
2476.                 q.push(v);
2477.             }
2478.     }
2479.     return 0;
2480. }
2481. int maxFlow(int s, int d) { // source, destination
2482.     int max_flow = 0;
2483.     while((bfs(s, d))) {
2484.         int flow = INT_MAX;
2485.         for(int v = d; v != s; v = parent[v]) {
2486.             int u = parent[v];
2487.             flow = min(flow, rG[u][v]);
2488.         }
2489.         for(int v = d; v != s; v = parent[v]) {
2490.             int u = parent[v];
2491.             rG[u][v] -= flow;
2492.             rG[v][u] += flow;
2493.         }
2494.         max_flow += flow;
2495.     }
2496.     return max_flow;
2497. }
2498.
2499. int main() {
2500.     int u, v, w, source, destination, Case = 1;
2501.     map<pair<int, int>, bool>Map;
2502.     while(scanf("%d", &V) && V) {
2503.         scanf("%d%d%d", &source, &destination, &E);

```

```

2504.     memset(rG, 0, sizeof rG);
2505.     for(int i = 0; i < E; ++i) {
2506.         scanf("%d%d%d", &u, &v, &w);
2507.         rG[u][v] += w;                                // edges are undirected
2508.         rG[v][u] += w;                                // remove this line if edges are
directed
2509.         if(Map.find({u, v}) == Map.end()) {           // same edges might occur more
than once
2510.             edge[u].push_back(v);                     // to avoid n^2 calculation
2511.             edge[v].push_back(u);
2512.             Map[{u, v}] = Map[{v, u}] = 1;
2513.         }}
2514.         printf("Network %d\n", Case++);
2515.         printf("The bandwidth is %d.\n\n", maxFlow(source, destination));
2516.
2517.         for(int i = 0; i <= V; ++i) edge[i].clear();
2518.         Map.clear();
2519.     }
2520.     return 0;
2521. }
2522.
2523. //1D Max Sum
2524. //Algorithm : Jay Kadane
2525. //Complexity : O(n)
2526.
2527. int main() {
2528.     int n;
2529.     scanf("%d", &n);
2530.     int A[n+1];
2531.     for(int i = 0; i < n; i++)
2532.         scanf("%d", &A[i]);
2533.
2534.     //Main part of the code
2535.     int sum = 0, ans = 0;
2536.     for(int i = 0; i < 9; i++) {
2537.         sum += A[i];
2538.         ans = max(sum, ans);                //always take the larger sum
2539.         if(sum < 0)
2540.             sum = 0;                        //if sum is negative, reset it (greedy)
2541.     }
2542.     printf("1D Max Sum : %d\n", ans);
2543.     return 0;
2544. }
2545.
2546. //2D Max Sum
2547. //Algorithm : DP, Inclusion Exclusion
2548. //Complexity : O(n^4)
2549.
2550. int main() {
2551.     int row_column, A[100][100];           //A square matrix
2552.     scanf("%d", &row_column);
2553.
2554.     for(int i = 0; i < row_column; i++)      //input of the matrix/2D array
2555.         for(int j = 0; j < row_column; j++) {
2556.             scanf("%d", &A[i][j]);
2557.             if(i > 0) A[i][j] += A[i-1][j];    //take from right
2558.             if(j > 0) A[i][j] += A[i][j-1];    //take from left
2559.             if(i > 0 && j > 0) A[i][j] -= A[i-1][j-1]; //inclusion exclusion

```



```

2560.     }
2561.
2562.     int maxSubRect = -1e7;
2563.     for(int i = 0; i < row_column; i++)           //i & j are the starting coordinate
of sub-rectangle
2564.         for(int j = 0; j < row_column; j++)
2565.             for(int k = i; k < row_column; k++)       //k & l are the finishing coordinate
of sub-rectangle
2566.                 for(int l = j; l < row_column; l++) {
2567.                     int subRect = A[k][l];
2568.                     if(i > 0) subRect -= A[i-1][l];
2569.                     if(j > 0) subRect -= A[k][j-1];
2570.                     if(i > 0 && j > 0) subRect += A[i-1][j-1];           //due to inclusion
exclusion
2571.                         maxSubRect = max(subRect, maxSubRect);
2572.                 }
2573.     printf("2D Max Sum : %d\n", maxSubRect);
2574.     return 0;
2575. }
2576.
2577. // MergeSort
2578.
2579. void MergeSort(long long arr[], int l, int mid, int r) {
2580.     int lftArrSize = mid-l+1, rhtArrSize = r-mid, lftArr[lftArrSize+2],
rhtArr[rhtArrSize+2];
2581.
2582.     for(int i = l, j = 0; i <= mid; ++i, ++j)
2583.         lftArr[j] = arr[i];
2584.     for(int i = mid+1, j = 0; i <= r; ++i, ++j)
2585.         rhtArr[j] = arr[i];
2586.
2587.     lftArr[lftArrSize] = rhtArr[rhtArrSize] = 1e9;           // INF value in both array (Basic
merge sort algo)
2588.     int lPos = 0, rPos = 0;
2589.     for(int i = l; i <= r; ++i) {
2590.         if(lftArr[lPos] <= rhtArr[rPos])
2591.             arr[i] = lftArr[lPos++];
2592.         else {
2593.             arr[i] = rhtArr[rPos++];
2594.             //cnt += lftArrSize - lPos;           // Delete this line if not needed
(Min Number of Swaps)
2595.         }}}
2596.
2597. void Divide(long long arr[], int l, int r) {
2598.     if(l == r || l > r) return;
2599.     int mid = (l+r)>>1;
2600.     Divide(arr, l, mid);
2601.     Divide(arr, mid+1, r);
2602.     MergeSort(arr, l, mid, r);
2603. }
2604. main() { Divide(v, 0, n-1); }
2605.
2606. // Modular Arithmetic
2607.
2608. // (2^10 % 5) = powMod(2, 10, 5)
2609. long long powMod(long long N, long long P, long long M) {
2610.     if(P==0) return 1;
2611.     if(P%2==0) {

```

```

2612.         long long ret = powMod(N, P/2, M)%M;
2613.         return (ret * ret)%M;
2614.     }
2615.     return ((N%M) * (powMod(N, P-1, M)%M))%M;
2616. }
2617.
2618. ll powerMOD(ll x, ll y) {                // Can find modular inverse by a^(MOD-2), a and
MOD must be co-prime
2619.     ll res = 1;
2620.     x %= MOD;
2621.     while(y > 0) {
2622.         if(y&1) res = (res*x)%MOD;        // If y is odd, multiply x with result
2623.         y = y >> 1, x = (x * x)%MOD;
2624.     }
2625.     return res%MOD;
2626. }
2627.
2628. // 2^100 = Pow(2, 100)
2629. unsigned long long Pow(unsigned long long N, unsigned long long P) {
2630.     if(P == 0) return 1;
2631.     if(P % 2 == 0) {
2632.         unsigned long long ret = Pow(N, P/2);
2633.         return ret*ret;
2634.     }
2635.     return N * Pow(N, P-1);
2636. }
2637.
2638. // calculate A mod B, where A : 0<A<(10^100000) (or greater)
2639. // take input as string and process with aftermod
2640. // calculate A^B mod M, where B : 0<A<(10^100000) (or greater)
2641. // take input as string and process with aftermod : afterMod(inputAsString, Mod-1)    due
to fermat theorem
2642.
2643. 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)
2644.     long long ans = 0;
2645.     string :: iterator it;
2646.
2647.     for(it = str.begin(); it != str.end(); it++)    // mod from first to last
2648.         ans = (ans*10 + (*it - '0')) % mod;
2649.     return ans;
2650. }
2651.
2652. // Exponent of Big numbers (N^P % M)
2653. // where N and P is bigger strings (both having length 10^5)
2654. long long bigExpo(char *N, char *P, long long M) {
2655.     long long base = 0, ans = 1;
2656.     for(int i = 0; N[i] != '\0'; ++i)
2657.         base = (base*10LL + N[i] - '0')%M;
2658.
2659.     for(int j = 0; P[j] != '\0'; ++j)
2660.         ans = (powMod(ans, 10, M) * powMod(base, P[j] - '0', M))%M;
2661.     return ans;
2662. }
2663.
2664. // Extended Euclud
2665. // a*x + b*y = gcd(a, b)
2666. // Given a and b calculate x and y so that a * x + b * y = d    (where gcd(a, b) | c)

```

```

2667. // x_ans = x + (b/d)n
2668. // y_ans = y - (a/d)n
2669. // where n is an integer
2670.
2671. // Solution only exists if d | c (i.e : c is divisible by d)
2672. ll gcdExtended(ll a, ll b, ll *x, ll *y) {           // C function for extended Euclidean
Algorithm
2673.     if (a == 0) {                                     // Base Case
2674.         *x = 0, *y = 1;
2675.         return b;
2676.     }
2677.     ll x1, y1;                                         // To store results of recursive call
2678.     ll gcd = gcdExtended(b%a, a, &x1, &y1);
2679.     *x = y1 - (b/a) * x1;
2680.     *y = x1;
2681.     return gcd;
2682. }
2683.
2684. ll modInverse(ll a, ll mod) {
2685.     ll x, y;
2686.     ll g = gcdExtended(a, mod, &x, &y);
2687.     if(g != 1) return -1;                             // ModInverse doesnt exist
2688.     ll res = (x%mod + mod) % mod;                     // m is added to handle negative x
2689.     return res;
2690. }
2691.
2692. // MO's Algo
2693. // Complexity : Q*sqrt(N)
2694.
2695. struct query {
2696.     int l, r, id;
2697. };
2698.
2699. const int block = 320;                                // For 100000
2700. query q[MAX];
2701. int ans[MAX];
2702.
2703. bool cmp(query &a, query &b) {
2704.     int block_a = a.l/block, block_b = b.l/block;
2705.     if(block_a == block_b)
2706.         return a.r < b.r;
2707.     return block_a < block_b;
2708. }
2709.
2710. bool cmp2(query &a, query &b){                        // Faster Comparison function
2711.     if(a.l/block != b.l/block) return a.l < b.l;
2712.     if((a.l/block)&1) return a.r < b.r;
2713.     return a.r > b.r;
2714. }
2715.
2716. void add(int x) {} // Add x'th value in range
2717. void remove(int x) {} // Remove x'th value from range
2718.
2719. int main() {
2720.     int Q;
2721.     scanf("%d", &Q);
2722.     for(int i = 0; i < Q; ++i) {                       // Query input
2723.         scanf("%d%d", &q[i].l, &q[i].r);

```

```

2724.         --q[i].l, --q[i].r, q[i].id = i;           // NOTE : value index starts from 0
2725.     }
2726.
2727.     sort(q, q+Q, cmp);
2728.     int l = 0, r = -1;
2729.     for(int i = 0; i < Q; ++i) {
2730.         while(l > q[i].l)    add(--l);
2731.         while(r < q[i].r)    add(++r);
2732.         while(l < q[i].l)    remove(l++);
2733.         while(r > q[i].r)    remove(r--);
2734.         ans[q[i].id] =           // Add Constraints
2735.     }
2736.     return 0;
2737. }
2738.
2739. // Directed Minimum Spanning Tree (Edmonds' algorithm)
2740. // Complexity :  $O(E \cdot V) \sim O(E + V \log V)$  [ works in  $O(E + V \log V)$  for
almost all cases ]
2741. // https://en.wikipedia.org/wiki/Edmonds%27\_algorithm
2742.
2743. struct edge {
2744.     int u, v, w;
2745.     edge() {}
2746.     edge(int a, int b, int c) : u(a), v(b), w(c) {}
2747. };
2748.
2749. int DMST(vector<edge> &edges, int root, int V) {
2750.     int ans = 0;
2751.     int cur_nodes = V;
2752.     while(1) {
2753.         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)
2754.                                                         // pi[v] : contains the
minimum weight edge's starting node u
2755.         for(int i = 0; i < (int)edges.size(); ++i) {
2756.             int u = edges[i].u, v = edges[i].v, w = edges[i].w;
2757.             if(w < lo[v] and u != v)
2758.                 lo[v] = w, pi[v] = u;
2759.         }
2760.
2761.         lo[root] = 0; // by default the weight to
go to root node is 0
2762.         for(int i = 0; i < (int)lo.size(); ++i) {
2763.             if(i == root) continue;
2764.             if(lo[i] == INF) return -1; // if there is no way to
visit a node v, then Directed MST doesn't exist
2765.         }
2766.
2767.         int cur_id = 0;
2768.         vector<int> id(cur_nodes, -1), mark(cur_nodes, -1);
2769.
2770.         for(int i = 0; i < cur_nodes; ++i) {
2771.             ans += lo[i]; // adding node i's minimum
weight to answer
2772.
2773.             int u;
2774.             for(u = i; u != root && id[u] < 0 && mark[u] != i; u = pi[u]) // marking
minimum weighted path from root to node i

```

```

2775.         mark[u] = i;
2776.
2777.         if(u != root && id[u] < 0) {                                // Contains cycle, as a
result u can not reach to i
2778.             for(int v = pi[u]; v != u; v = pi[v])                // mark all cycle nodes
with id
2779.                 id[v] = cur_id;
2780.                 id[u] = cur_id++;                                // ??
2781.             }}
2782.
2783.             if(cur_id == 0) break;                                // there is no cycle, so
all node is possibly visited
2784.             for(int i = 0; i < cur_nodes; ++i)
2785.                 if(id[i] < 0) id[i] = cur_id++;
2786.
2787.             for(int i = 0; i < (int)edges.size(); ++i) {
2788.                 int u = edges[i].u, v = edges[i].v;
2789.                 edges[i].u = id[u];
2790.                 edges[i].v = id[v];
2791.                 if(id[u] != id[v]) edges[i].w -= lo[v];
2792.             }
2793.
2794.             cur_nodes = cur_id;
2795.             root = id[root];
2796.         }
2797.         return ans;                                            // returns total cost of
MST
2798.     }
2799.
2800. // MST Kruskal + Union Find Disjoint Set (DSU)
2801. // Complexity of MST : O(E logV)
2802.
2803. // Let a graph be G1, and the MST of the graph is MST1
2804. // and a graph G2, where G2 contains same edges as G1 with some new edges
2805. // then the new MST of graph G2 will be :
2806. // MST2 = MST(of the edges used in M1 (MST of G1) + new added edges)
2807.
2808. set<pair<int, pair<int, int> > > Edge;                        // USED STL SET!!
2809.
2810. int MST(int V) {
2811.     int mstCost = 0, edge = 0;                                // If Edge list is STL vector, then
sort it!
2812.     DSU U(V+5);
2813.     set<pair<int, pair<int, int> > > :: iterator it = Edge.begin();    // Contains {Weight,
{U, V}}
2814.
2815.     for( ; it != Edge.end() && edge < V; ++it) {
2816.         int u = (*it).second.first;
2817.         int v = (*it).second.second;
2818.         int w = (*it).first;
2819.
2820.         if(!U.isSameSet(u, v))
2821.             ++edge, mstCost += w, U.makeUnion(u, v);
2822.     }
2823.
2824.     if(edge != V-1) return -1;                                // Some edge is missing, so no MST found!
2825.     return mstCost;
2826. }

```

```
2827.
2828. //Minimum Spanning Tree
2829. //Prim's Algorithm
2830. //Complexity : O(E logV)
2831.
2832. vector<int> G[MAX], W[MAX];
2833. priority_queue<pair<int, int> >pq;
2834. bitset<MAX>taken;
2835.
2836. void process(int u) {
2837.     taken[u] = 1;
2838.     for(int i = 0; i < (int)G[u].size(); i++) {
2839.         int v = G[u][i];
2840.         int w = W[u][i];
2841.         if(!taken[v])
2842.             pq.push(make_pair(-w, -v));
2843.     }
2844. }
2845.
2846. int main() {
2847.     taken.reset();
2848.     process(0); //taking 0 node as default
2849.     int mst_cost = 0;
2850.     while(!pq.empty()) {
2851.         w = -pq.top().first;
2852.         v = -pq.top().second;
2853.         pq.pop();
2854.         //if the node is not taken, then use this node
2855.         //as it contains the minimum edge
2856.         if(!taken[v])
2857.             mst_cost += w, process(v);
2858.     }
2859.     printf("Prim's MST cost : %d\n", mst_cost);
2860.     return 0;
2861. }
2862.
2863. // N'th Permutation
2864.
2865. long long per[30] = {0};
2866. long long permute(int freq[]) {
2867.     int cnt = 0;
2868.     for(int i = 0; i < 26; ++i)
2869.         cnt+=freq[i];
2870.     long long permutation = per[cnt] < 1 ? 0:per[cnt];
2871.     for(int i = 0; i < 26; ++i)
2872.         if(freq[i] > 1)
2873.             permutation /= per[freq[i]];
2874.     return permutation;
2875. }
2876.
2877. string NthPermutation(string str, int n) { // string and numbet of permutation
2878.     string s;
2879.     int freq[30] = {0};
2880.     for(int i = 0; i < (int)str.size(); ++i)
2881.         freq[str[i]-'a']++;
2882.     if(per[0] == 0) { // if not initialized
2883.         per[0] = 1;
2884.         for(int i = 1; i <= 25; ++i)
```

```
2885.         per[i] = per[i-1]*i;
2886.     }
2887.     if(permute(freq) < n)
2888.         return s;
2889.     for(int i = 0; i < (int)str.size(); ++i) {
2890.         for(int j = 0; j < 26; ++j) {
2891.             if(freq[j] <= 0) continue;
2892.             freq[j]--;
2893.             long long P = permute(freq);
2894.             if(P >= n) {
2895.                 s += char(j+'a');
2896.                 break;
2897.             }
2898.             else {
2899.                 n -= P;
2900.                 freq[j]++;
2901.             }
2902.         }
2903.     }
2904.     return s; // Returns empty string if not possible
2905. }
2906.
2907. // Palindromic Tree
2908.
2909. struct node {
2910.     int start, end, length, edge[26], suffixEdg;
2911. };
2912.
2913. struct PalinTree {
2914.     int currNode;
2915.     string s; // Contains the string
2916.     int ptr, mxLen;
2917.     node root1, root2, tree[MAX];
2918.     PalinTree() {
2919.         s.clear();
2920.         root1.length = -1;
2921.         root1.suffixEdg = 1;
2922.         root2.length = 0;
2923.         root2.suffixEdg = 1;
2924.         tree[1] = root1;
2925.         tree[2] = root2;
2926.         ptr = 2;
2927.         currNode = 1;
2928.         mxLen = 0;
2929.     }
2930.     void insert(int idx) {
2931.         int tmp = currNode;
2932.         while(1) {
2933.             int curLength = tree[tmp].length;
2934.             if(idx - curLength >= 1 && s[idx] == s[idx-curLength-1]) break;
2935.             tmp = tree[tmp].suffixEdg;
2936.         }
2937.         if(tree[tmp].edge[s[idx]-'a'] != 0) {
2938.             currNode = tree[tmp].edge[s[idx]-'a'];
2939.             return;
2940.         }
2941.         ptr++;
2942.         tree[tmp].edge[s[idx]-'a'] = ptr;
2943.         tree[ptr].length = tree[tmp].length + 2;
2944.         tree[ptr].end = idx;
```

```

2943.     tree[ptr].start = idx - tree[ptr].length + 1;
2944.     tmp = tree[tmp].suffixEdg;
2945.     currNode = ptr;
2946.     if(tree[currNode].length == 1) {
2947.         tree[currNode].suffixEdg = 2;
2948.         return;
2949.     }
2950.     while(1) {
2951.         int curLength = tree[tmp].length;
2952.         if(idx-curLength >= 1 && s[idx] == s[idx-curLength-1])
2953.             break;
2954.         tmp = tree[tmp].suffixEdg;
2955.     }
2956.     tree[currNode].suffixEdg = tree[tmp].edge[s[idx]-'a'];
2957. }
2958. void buildTree() {          // Builds Palindrome Tree of string s
2959.     for(int i = 0; i < (int)s.length(); ++i)
2960.         insert(i);
2961. }
2962. void CalMaxLen() {
2963.     for(int i = 3; i <= ptr; ++i)
2964.         mxLen = max(mxLen, tree[i].end - tree[i].start);
2965. };
2966.
2967. int main() {
2968.     PalinTree pt;
2969.     cin >> pt.s;
2970.     pt.buildTree();
2971.     cout << "All distinct palindromic substring for " << pt.s << " : \n";
2972.     for(int i=3; i<=pt.ptr; i++) {
2973.         cout << i-2 << ") ";
2974.         for(int j=pt.tree[i].start; j<=pt.tree[i].end; j++)
2975.             cout << pt.s[j];
2976.         cout << endl;
2977.     }
2978.
2979. // Policy Based Data Structures
2980. // Source : http://codeforces.com/blog/entry/11080
2981. //          http://codeforces.com/blog/entry/13279
2982. // Problems:
2983. // http://codeforces.com/blog/entry/53735
2984. // http://codeforces.com/contest/762/problem/E
2985.
2986. #include <bits/stdc++.h>
2987. #include <ext/pb_ds/assoc_container.hpp>    // rb_tree_tag
2988. #include <ext/pb_ds/tree_policy.hpp>        // tree_order_statistics_node_update
2989. #define at(X)          find_by_order(X)
2990. #define lessThan(x)    order_of_key(x)
2991. using namespace std;
2992. using namespace __gnu_pbds;
2993. template<class T> using ordered_set = tree<T, null_type, less_equal<T>, rb_tree_tag,
tree_order_statistics_node_update>;
2994.
2995. // key, Mapped-Policy, Key Comparison Func, Underlying data Structure, Policy for updating
node interval
2996.
2997. // Key Comparison Func : Defines how data will be stored (increasing, decreasing order)

```



```

2998. //          less<int>          -   Same value occurs once & increasing
           SET
2999. //          less_equal<int>      -   Same value occurs one or more & increasing
           MULTISSET
3000. //          greater<int>, greater_equal<int>
3001. //
3002. // if Mapped-Policy set to null_type, this works as a SET
3003. // else works as MAP
3004. //
3005. // Underlying Data Structures : rb_tree_tag      -   Red Black Tree
3006. //                               splay_tree_tag   -   Splay Tree
3007. //                               ov_tree_tag      -   Ordered Vector Tree
3008. //
3009. // Policy for Updaing Node : default      -   null_node_update
3010. //                               c++ implemented  -   tree_order_statistics_node_update
3011.
3012. // Features :
3013. // Can be used as SET/MULTISSET
3014. // lower_bound and upper bound works as expected
3015. // insertion, deletation, clear
3016. // container.find_by_order(x) returns x'th elements iterator
3017. // container.order_of_key(x) returns number of values less than (or equal to) x
3018. // auto casting works
3019. //
3020.
3021. int main() {
3022.     ordered_set<int> X;
3023.     // ----- INSERTION -----
3024.     // Data are indexed in increasing order & can occur only once (like STL SET)
3025.     X.insert(1); X.insert(18); X.insert(2); X.insert(2); X.insert(4); X.insert(8);
X.insert(16);
3026.
3027.     // ----- ITERATION -----
3028.     // Index-Wise : log_n
3029.     cout << *X.find_by_order(0) << endl;
3030.     cout << *X.find_by_order(2) << endl;
3031.     cout << *X.find_by_order(6) << endl;    // Not Present, Will return 0
3032.
3033.     if(X.end() == X.find_by_order(6)) cout << "End Reached" << endl;
3034.     X.erase(X.find_by_order(2));           // Deleting element by iterator
3035.     cout << "Iterating \n";
3036.     for(auto x : X) cout << x << endl;
3037.     cout << endl;
3038.
3039.     // ----- Range Search -----
3040.     // Returns number of elements STRICTLY less than value
3041.     cout << X.order_of_key(-5) << endl;
3042.     cout << X.order_of_key(1) << endl;
3043.     cout << X.order_of_key(3) << endl;
3044.     cout << X.order_of_key(4) << endl;
3045.     cout << X.order_of_key(400) << endl;
3046.     X.clear();           // Clearing
3047. }
3048.
3049. // Persistant/Dynamic Segment Tree
3050. // Pointer Version
3051.
3052. struct node {

```

```
3053.     ll val;
3054.     node *lft, *rht;
3055.     node(node *L = NULL, node *R = NULL, ll v = 0) {
3056.         lft = L;
3057.         rht = R;
3058.         val = v;
3059.     };
3060. node *persis[101000], *null = new node();
3061. // null->lft = null->rht = null;
3062. node *nCopy(node *x) {
3063.     node *tmp = new node();
3064.     if(x) {
3065.         tmp->val = x->val;
3066.         tmp->lft = x->lft;
3067.         tmp->rht = x->rht;
3068.     }
3069.     return tmp;
3070. }
3071. void init(node *pos, ll l, ll r) {
3072.     if(l == r) {
3073.         pos->val = val[l];
3074.         pos->lft = pos->rht = null;
3075.         return;
3076.     }
3077.     ll mid = (l+r)>>1;
3078.     pos->lft = new node();
3079.     pos->rht = new node();
3080.     init(pos->lft, l, mid);
3081.     init(pos->rht, mid+1, r);
3082.     pos->val = pos->lft->val + pos->rht->val;
3083. }
3084. // Single Position update
3085. void update(node *pos, ll l, ll r, ll idx, ll val) {
3086.     if(l == r) {
3087.         pos->val += val;
3088.         pos->lft = pos->rht = null;
3089.         return;
3090.     }
3091.     ll mid = (l+r)>>1;
3092.     if(idx <= mid) {
3093.         pos->lft = nCopy(pos->lft);
3094.         if(!pos->rht)
3095.             pos->rht = null;
3096.         update(pos->lft, l, mid, idx, val);
3097.     }
3098.     else {
3099.         pos->rht = nCopy(pos->rht);
3100.         if(!pos->lft)
3101.             pos->lft = null;
3102.         update(pos->rht, mid+1, r, idx, val);
3103.     }
3104.     pos->val = 0;
3105.     if(pos->lft) pos->val += pos->lft->val;
3106.     if(pos->rht) pos->val += pos->rht->val;
3107. }
3108. // Range [L, R] update
3109. void update(node *pos, ll l, ll r, ll L, ll R, ll val) {
3110.     if(r < L || R < l) return;
```

```

3111.     if(L <= l && r <= R) {
3112.         pos->prop += val;
3113.         pos->val += (r-l+1)*val;
3114.         return;
3115.     }
3116.     ll mid = (l+r)>>1;
3117.     pos->lft = nCopy(pos->lft);           // Can be reduced
3118.     pos->rht = nCopy(pos->rht);
3119.     update(pos->lft, l, mid, L, R, val);
3120.     update(pos->rht, mid+1, r, L, R, val);
3121.     pos->val = pos->lft->val + pos->rht->val + (r-l+1)*pos->prop;
3122. }
3123. ll query(node *pos, ll l, ll r, ll L, ll R) {           // Range [L, R] Sum Query
3124.     if(r < L || R < l || pos == NULL) return 0;
3125.     if(L <= l && r <= R) return pos->val;
3126.     ll mid = (l+r)/2LL;
3127.     ll x = query(pos->lft, l, mid, L, R);
3128.     ll y = query(pos->rht, mid+1, r, L, R);
3129.     return x+y;
3130. }
3131. // Ignore LMax persistant tree positions query for finding k'th value
3132. int query(node *RMax, node *LMax, int l, int r, int k) {           // (LMax :
past, RMax : updated)
3133.     if(l == r) return l;
3134.     // NO NEED THIS SECTOR STILL AC --
3135.     RMax->lft = nCopy(RMax->lft);
3136.     LMax->lft = nCopy(LMax->lft);
3137.     RMax->rht = nCopy(RMax->rht);
3138.     LMax->rht = nCopy(LMax->rht);
3139.     //-----
3140.     // for each range [l, r] we will ignore every [1, l-1] range numbers
3141.     int Count = RMax->lft->val - LMax->lft->val;
3142.     int mid = (l+r)>>1;
3143.     // if there exists more than or equal to k values in left range, then we'll find kth
number in left segment
3144.     if(Count >= k) return query(RMax->lft, LMax->lft, l, mid, k);
3145.     else return query(RMax->rht, LMax->rht, mid+1, r, k-Count);
3146. }
3147.
3148. // ----- Propagation -----
3149. bool flipProp(bool par, bool child) {
3150.     if(par == child) return 0;
3151.     return 1;
3152. }
3153. void propagate(node *pos, ll l, ll r) {           // Propagation Func
3154.     if(l == r) return;
3155.     pos->lft = nCopy(pos->lft);           // No need to copy in update/query function
3156.     pos->rht = nCopy(pos->rht);
3157.     if(!pos->flip) return;
3158.     ll mid = (l+r)>>1;
3159.     pos->lft->flip = flipProp(pos->flip, pos->lft->flip);
3160.     pos->rht->flip = flipProp(pos->flip, pos->rht->flip);
3161.     pos->lft->val = (mid-l+1)-pos->lft->val;
3162.     pos->rht->val = (r-mid)-pos->rht->val;
3163.     pos->flip = 0;
3164. }
3165. // Flip in range [L, R]
3166. void updateFlip(node *pos, ll l, ll r, ll L, ll R) {

```

```

3167.     if(r < L || R < l) return;
3168.     propagate(pos, l, r);
3169.     if(L <= l && r <= R) {
3170.         pos->flip = 1;
3171.         pos->val = (r-l+1) - pos->val;
3172.         return;
3173.     }
3174.     ll mid = (l+r)>>1;
3175.     updateFlip(pos->lft, l, mid, L, R);
3176.     updateFlip(pos->rht, mid+1, r, L, R);
3177.     pos->val = 0;
3178.     if(pos->rht) pos->val += pos->rht->val;
3179.     if(pos->lft) pos->val += pos->lft->val;
3180. }
3181. // Erasing A segment tree, pos = root, must run for each root
3182. void ClearTree(node *pos) {
3183.     if(pos == NULL) {
3184.         delete pos;
3185.         return;
3186.     }
3187.     ClearTree(pos->lft);
3188.     ClearTree(pos->rht);
3189.     delete pos;
3190. }
3191.
3192. int main() {
3193.     // MUST BE INITIALIZED
3194.     null->lft = null->rht = null;
3195.     //
3196.     for(int i = 1; i <= 10; ++i) {
3197.         persis[i] = nCopy(persis[i-1]);
3198.         update(persis[i], 1, n, idx, val);
3199.     }
3200.     return 0;
3201. }
3202.
3203. // Limit ----- No. of Primes
3204. // 100           25
3205. // 1000          168
3206. // 10,000        1229
3207. // 100,000       9592
3208. // 1,000,000     78498
3209. // 10,000,000    664579
3210.
3211. bitset<10000000>isPrime;
3212. vector<long long>primes;
3213.
3214. void sieve(unsigned long long N) {
3215.     isPrime.set();
3216.     isPrime[0] = isPrime[1] = 0;
3217.     unsigned long long lim = sqrt(N) + 5;
3218.     for(unsigned long long i = 2; i <= lim; i++) {          // change lim to N, if all primes in
range N is needed
3219.         if(isPrime[i])
3220.             for(unsigned long long j = i*i; j <= N; j+= i)
3221.                 isPrime[j] = 0;
3222.     }}
3223.

```

```

3224. void sieveGen(unsigned long long N) {
3225.     isPrime.set();
3226.     isPrime[0] = isPrime[1] = 0;
3227.     for(unsigned long long i = 2; i <= N; i++) {           //Note, N isn't square rooted!
3228.         if(isPrime[i]) {
3229.             for(unsigned long long j = i*i; j <= N; j+= i)
3230.                 isPrime[j] = 0;
3231.             primes.push_back(i);
3232.         }
3233.     }
3234.     vector<pair<ull, ull> > primeFactor(ull n) {
3235.         vector<pair<ull, ull> > factor;
3236.         for(long long i = 0; i < (int)primes.size() && primes[i] <= n; i++) {
3237.             bool first = 1;
3238.             while(n%primes[i] == 0) {
3239.                 if(first) {
3240.                     factor.push_back({primes[i], 0});
3241.                     first = 0;
3242.                 }
3243.                 factor.back().second++;
3244.                 n/=primes[i];
3245.             }
3246.             return factor;
3247.         }
3248.
3249.         int pd[MAX];                                // Contains minimum prime factor/divisor, for primes
3250.         pd[x] = x
3251.         vector<int>primes;                            // Contains prime numbers
3252.         void SieveLinear(int N) {
3253.             for(int i = 2; i <= N; ++i) {
3254.                 if(pd[i] == 0) pd[i] = i, primes.push_back(i);           // if pd[i] == 0,
3255.                 then i is prime
3256.                 for(int j=0; j < (int)primes.size() && primes[j] <= pd[i] && i*primes[j] <= N; ++j)
3257.                     pd[i*primes[j]] = primes[j];
3258.             }
3259.             int pd[MAX];                                // Contains minimum prime factor/divisor, for primes
3260.             pd[x] = x
3261.             vector<int>primes;                            // Contains prime numbers
3262.             vector<int>PF[MAX];
3263.             void SieveLinearRangePF(int N, ll low, ll hi) {           // also returns unique prime
3264.                 factors in range [low, hi]
3265.                 for(int i = 2; i <= N; ++i) {
3266.                     if(pd[i] == 0) {
3267.                         pd[i] = i, primes.push_back(i);           // if pd[i] == 0, then i is
3268.                         prime
3269.                         for(ll x = (low-1)-(low-1)%i+i; x <= hi; x += i)           // inserting all prime
3270.                             factors [prime will be inserted only once]
3271.                             PF[x-low].push_back(i);           // just to be sure, used
3272.                         low-1, instead of low
3273.                     }
3274.                     for(int j=0; j < (int)primes.size() && primes[j] <= pd[i] && i*primes[j] <= N; ++j)
3275.                         pd[i*primes[j]] = primes[j];
3276.                 }
3277.             }
3278.             vector<ll> Divisors(ll n) {           // Returns the divisors
3279.                 ll lim = sqrt(n);
3280.                 vector<ll>divisor;

```

```
3275.     for(ll i = 2; i <= lim; i++) {           // deal with 1 and n manually
3276.         if(n % i == 0) {
3277.             ll tmp = n/i;
3278.             divisor.push_back(tmp);
3279.             if(i != tmp)
3280.                 divisor.push_back(i);
3281.         }}
3282.     return divisor;
3283. }
3284.
3285. vector<pair<long long, long long> > factorialFactorization(long long n) { // prime
factorization of factorials (n!)
3286.     vector<pair<long long, long long> >V;
3287.     for(long long i = 0; i < (int)primes.size() && primes[i] <= n; i++) {
3288.         long long tmp = n, power = 0;
3289.         while(tmp/primes[i]) {
3290.             power += tmp/primes[i];
3291.             tmp /= primes[i];
3292.         }
3293.         if(power != 0) V.push_back(make_pair(primes[i], power));
3294.     }
3295.     return V;
3296. }
3297.
3298. long long numPF(long long n) {                 //returns number of prime factors
3299.     long long num = 0;
3300.     for(long long i = 0; primes[i] * primes[i] <= n; i++) {
3301.         while(n % primes[i] == 0) {
3302.             n /= primes[i];
3303.             num++;
3304.         }}
3305.     if(n > 1) num++;           //there might left a prime number which is bigger than primes[i]
3306.     return num;
3307. }
3308.
3309. long long numDIFPF(long long n) {             // returns number of different prime factors
3310.     long long diff_num = 0;
3311.     for(long long i = 0; primes[i] * primes[i] <= n; i++) {
3312.         bool ok = 0;
3313.         while(n % primes[i] == 0) {
3314.             n /= primes[i];
3315.             ok = 1;
3316.         }
3317.         if(ok) diff_num++;
3318.     }
3319.     if(n > 1) diff_num++;
3320.     return diff_num;
3321. }
3322.
3323. unsigned long long sumPF(long long n) {        // returns sum of prime factors
3324.     unsigned long long sum = 0;
3325.     for(long long i = 0; primes[i] * primes[i] <= n; i++)
3326.         while(n % primes[i] == 0) {
3327.             n /= primes[i];
3328.             sum+=primes[i];
3329.         }
3330.     if(n > 1) sum+= n;
3331.     return sum;
```

```

3332. }
3333.
3334. int NumberOfDivisors(long long n) {          // if  $n = p_1^{a_1} * p_2^{a_2}, \dots$  then  $NOD = (a_1+1) * (a_2+1) * \dots$ 
3335.     if(n <= MAX and isPrime[n]) return 2;
3336.     int NOD = 1;
3337.     for(int i = 0, a = 0; i < (int)primes.size() and primes[i] <= n; ++i, a = 0) {
3338.         while(n % primes[i] == 0)
3339.             ++a, n /= primes[i];
3340.         NOD *= (a+1);
3341.     }
3342.     if(n != 1) NOD *= 2;
3343.     return NOD;
3344. }
3345.
3346. //-----Fast Factorization using Sieve-Like algorithm-----
3347. bitset<MAX>isPrime;
3348. int divisor[MAX];
3349.
3350. void sieve(long long lim) {                  // Prime numbers for the limit should be sieved, otherwise WA
3351.     isPrime.set();
3352.     isPrime[0] = isPrime[1] = 0;
3353.     for(ll i = 0; i <= lim; ++i) {
3354.         if(isPrime[i]) {
3355.             for(long long j = i; j <= lim; j += i) {
3356.                 isPrime[j] = 0;
3357.                 divisor[j] = i;
3358.             }
3359.         }
3360.     }
3361.     vector<int> factorize(long long x) {      // This function only iterates over the prime numbers
3362.         int pastDiv = 0;                      // 0 : no divisor is present
3363.         vector<int> factor;
3364.         while(x > 1) {
3365.             if(divisor[x] != 0) {
3366.                 factor.push_back(divisor[x]);
3367.                 x /= divisor[x];               // now x would be reduced by factor of divisor[x]
3368.             }
3369.         }
3370.         return factor;
3371.     }
3372.     //-----
3373.     // Prime Probability
3374.     // Algorithm : Miller-Rabin primality test          Complexity :  $k * (\log n)^3$ 
3375.     // This function is called for all k trials. It returns false if n is composite and returns false if n is probably prime.
3376.     // d is an odd number such that  $d * (2^r) = n-1$  for some  $r \geq 1$ 
3377.     bool miillerTest(int d, int n) {
3378.         int a = 2 + rand() % (n - 4);         // Pick a random number in  $[2..n-2]$ .
3379.         int x = Pow(a, d, n);                 // Compute  $a^d \% n$ 
3380.         if (x == 1 || x == n-1)
3381.             return 1;
3382.         while (d != n-1) {                     // Keep squaring x while one of the following doesn't happen
3383.             x = (x * x) % n;                   // (i) d does not reach n-1
3384.             d *= 2;                           // (ii)  $(x^2) \% n$  is not 1

```

```

3385.         if (x == 1)         return 0;           // (iii) (x^2) % n is not n-1
3386.         if (x == n-1)       return 1;
3387.     }
3388.     return 0;                 // Return composite
3389. }
3390.
3391. bool isPrime(int n, int k = 10) {                 // Higher value of k gives more accuracy
(Use k >= 9)
3392.     if(n <= 1 || n == 4) return 0;               // Corner cases
3393.     if(n <= 3) return 1;
3394.     int d = n - 1;                               // Find r such that n = 2^d * r + 1 for some
r >= 1
3395.     while(d % 2 == 0) d /= 2;
3396.     for(int i = 0; i < k; i++)                   // Iterate given nber of 'k' times
3397.         if(miillerTest(d, n) == 0)
3398.             return 0;
3399.     return 1;
3400. }
3401.
3402. // Binary Search
3403. // Complexity : O(n Log n)
3404.
3405. ll UpperBound(ll lo, ll hi, ll key) {             // Returns lowest position where v[i] >
key
3406.     ll mid, ans = -1;                             // 10 10 10 20 20 20 30 30
3407.     while(lo <= hi) {                             //                               ^
3408.         mid = (lo + hi)>>1;
3409.         if(key >= v[mid]) ans = mid, lo = mid + 1;
3410.         else hi = mid - 1;
3411.     }
3412.     return ans+1;                                  // Tweaking this line will return the last
position of key
3413. }
3414.
3415. ll LowerBound(ll lo, ll hi, ll key) {             // Returns lowest position where v[i] == key
(if value is present more than once)
3416.     ll mid, ans = -1;                             // 10 10 10 20 20 20 30 30
3417.     while(lo <= hi) {                             //                               ^
3418.         mid = (lo+hi)>>1;
3419.         if(key <= v[mid]) ans = mid, hi = mid - 1;
3420.         else lo = mid + 1;
3421.     }
3422.     return ans;
3423. }
3424.
3425. // lo : lower value, hi : upper value, est : estimated output of the required result, delta
: number of iteration in search
3426. double bisection(double lo, double hi, double est, int delta) {
3427.     double mid, ans = -1;
3428.     for(int i = 0; i < delta; ++i) {
3429.         mid = (lo+hi)/2.0;
3430.         if(Equal(TestFunction(mid), est))          ans = mid, lo = mid;
3431.         else if(Greater(TestFunction(mid), est))    hi = mid;
3432.         else lo = mid;
3433.     }
3434.     return ans;
3435. }
3436.

```



```
3437. // Full Functional Ternary Search
3438. /* EMAXX ::
3439. If  $f(x)$  takes integer parameter, the interval  $[l\ r]$  becomes discrete.
3440. Since we did not impose any restrictions on the choice of points  $m_1$  and  $m_2$ , the correctness
3441. of the algorithm is not affected.
3442.
3443. The difference occurs in the stopping criterion of the algorithm.
3444. Ternary search will have to stop when  $(r-l) < 3$ , because in that case we can no longer
3445. select  $m_1$  and  $m_2$  to
3446. be different from each other as well as from  $ll$  and  $rr$ , and this can cause infinite
3447. iterating.
3448. Once  $(r-l) < 3$ , the remaining pool of candidate points  $(l, l+1, \dots, r)$  needs to be checked
3449. to find the point which produces the maximum value  $f(x)$ .
3450. */
3451. ll ternarySearch(ll low, ll high) {
3452.     ll ret = -INF;
3453.     while((high - low) > 2) {
3454.         ll mid1 = low + (high - low) / 3;
3455.         ll mid2 = high - (high - low) / 3;
3456.         ll cost1 = f(mid1);
3457.         ll cost2 = f(mid2);
3458.         if(cost1 < cost2) {
3459.             low = mid1;
3460.             ret = max(cost2, ret);
3461.         }
3462.         else {
3463.             high = mid2;
3464.             ret = max(cost1, ret);
3465.         }
3466.     }
3467.     for(int i = low; i <= high; ++i)
3468.         ret = max(ret, f(i));
3469.     return ret;
3470. }
3471.
3472. // Segment Tree
3473. // Only Supports Range Value SET (NOT UPDATE) and Point Query
3474. struct SegTreeSetVal {
3475.     vector<int>tree;
3476.     vector<bool>prop;
3477.
3478.     void Resize(int n) {
3479.         tree.resize(n*5);
3480.         prop.resize(n*5);
3481.     }
3482.
3483.     void propagate(int pos, int l, int r) {
3484.         if(!prop[pos] || l == r) return;
3485.         tree[pos<<1|1] = tree[pos<<1] = tree[pos];
3486.         prop[pos<<1|1] = prop[pos<<1] = 1;
3487.         prop[pos] = 0;
3488.     }
3489.
3490.     void SetVal(int pos, int l, int r, int L, int R, int val) { // Set value val in
3491.         range [L, R]
3492.         if(r < L || R < l) return;
```

```

3491.     propagate(pos, l, r);
3492.     if(L <= l && r <= R) {
3493.         tree[pos] = val;
3494.         prop[pos] = 1;
3495.         return;
3496.     }
3497.     int mid = (l+r)>>1;
3498.     SetVal(pos<<1, l, mid, L, R, val);
3499.     SetVal(pos<<1|1, mid+1, r, L, R, val);
3500. }
3501.
3502. int query(int pos, int l, int r, int idx) { // Can be modified to range
query
3503.     if(l == r) return tree[pos];
3504.     propagate(pos, l, r);
3505.     int mid = (l+r)>>1;
3506.     if(idx <= mid) return query(pos<<1, l, mid, idx);
3507.     else return query(pos<<1|1, mid+1, r, idx);
3508. };
3509.
3510.
3511. // Segment Tree Range Sum : Lazy with Propagation (MOD used)
3512. struct SegTreeRSQ {
3513.     vector<ll>sum, prop;
3514.
3515.     void Resize(int n) {
3516.         sum.resize(5*n);
3517.         prop.resize(5*n);
3518.     }
3519.
3520.     void init(int pos, int l, int r, ll val[]) {
3521.         sum[pos] = prop[pos] = 0;
3522.         if(l == r) {
3523.             sum[pos] = val[l]%MOD;
3524.             return;
3525.         }
3526.         int mid = (l+r)>>1;
3527.         init(pos<<1, l, mid, val);
3528.         init(pos<<1|1, mid+1, r, val);
3529.         sum[pos] = (sum[pos<<1] + sum[pos<<1|1])%MOD;
3530.     }
3531.
3532.     void propagate(int pos, int l, int r) {
3533.         if(prop[pos] == 0 || l == r) return;
3534.         int mid = (l+r)>>1;
3535.
3536.         sum[pos<<1] = (sum[pos<<1] + prop[pos]*(mid-l+1))%MOD;
3537.         sum[pos<<1|1] = (sum[pos<<1|1] + prop[pos]*(r-mid))%MOD;
3538.         prop[pos<<1] = (prop[pos<<1] + prop[pos])%MOD;
3539.         prop[pos<<1|1] = (prop[pos<<1|1] + prop[pos])%MOD;
3540.         prop[pos] = 0;
3541.     }
3542.
3543.     void update(int pos, int l, int r, int L, int R, ll val) {
3544.         if(r < L || R < l) return;
3545.         propagate(pos, l, r);
3546.         if(L <= l && r <= R) {
3547.             sum[pos] = (sum[pos] + val*(r-l+1))%MOD;

```

```

3548.         prop[pos] = (prop[pos] + val)%MOD;
3549.         return;
3550.     }
3551.
3552.     int mid = (l+r)>>1;
3553.     update(pos<<1, l, mid, L, R, val);
3554.     update(pos<<1|1, mid+1, r, L, R, val);
3555.     sum[pos] = (sum[pos<<1] + sum[pos<<1|1])%MOD;
3556. }
3557.
3558. ll query(int pos, int l, int r, int L, int R) {
3559.     if(r < L || R < l) return 0;
3560.     propagate(pos, l, r);
3561.     if(L <= l && r <= R) return sum[pos];
3562.     int mid = (l+r)>>1;
3563.     return (query(pos<<1, l, mid, L, R) + query(pos<<1|1, mid+1, r, L, R))%MOD;
3564. };
3565.
3566.
3567. // Dynamic Range Segment Tree (values can be deleted from right)
3568. // Resize the Segment Tree with the maximum length of value
3569. // Segment Tree Range Sum, Range Update, And Single point Value Change (If the last value
3570. // was deleted)
3571. // http://codeforces.com/contest/283/problem/A
3572. // Range Sum with Carry Value
3573. struct SegSum {
3574.     int tree[MAX*4], carry[MAX*4];
3575.
3576.     void init() {
3577.         memset(tree, 0, sizeof tree);
3578.         memset(carry, 0, sizeof carry);
3579.     }
3580.
3581.     void Update(int pos, int l, int r, int x, int y, ll val) { // Update value
3582.         // at range/point
3583.         if(y < l || x > r) return;
3584.         if(x <= l && r <= y) {
3585.             tree[pos] += (r-l+1)*val;
3586.             carry[pos] += val;
3587.             return;
3588.         }
3589.         int mid = (l+r)>>1;
3590.         Update(pos<<1, l, mid, x, y, val);
3591.         Update(pos<<1|1, mid+1, r, x, y, val);
3592.         tree[pos] = tree[pos<<1] + tree[pos<<1|1] + (r-l+1)*carry[pos];
3593.     }
3594.
3595.     ll Read(int pos, int l, int r, int x, int y, ll Carry = 0) { // Read value at
3596.         // range/point
3597.         if(y < l || x > r) return 0;
3598.         if(x <= l && r <= y) return tree[pos] + Carry * (r-l+1);
3599.         ll mid = (l+r)>>1;
3600.         ll lft = Read(pos<<1, l, mid, x, y, Carry + carry[pos]);
3601.         ll rht = Read(pos<<1|1, mid+1, r, x, y, Carry + carry[pos]);
3602.         return lft + rht;
3603.     }
3604. }

```

```

3603. // Sets value at idx
3604. void Set(int pos, int l, int r, int idx, ll val, ll Carry = 0) {
3605.     if(l == r) {
3606.         tree[pos] = val + (-1*Carry); // Extra carry values are eliminated in such way
3607.         carry[pos] = 0; // that the subtraction is always the new value
3608.         return;
3609.     }
3610.     int mid = (l+r)>>1;
3611.     if(idx <= mid) Set(pos<<1, l, mid, idx, val, Carry + carry[pos]);
3612.     else Set(pos<<1|1, mid+1, r, idx, val, Carry + carry[pos]);
3613.     tree[pos] = tree[pos<<1] + tree[pos<<1|1] + (r-l+1) * carry[pos];
3614. };
3615.
3616. // SegTree with Lazy Propagation (Flip Count in Range)
3617. // Prop :
3618. // 0 : No prop operation
3619. // 1 : Prop operation should be done
3620.
3621. struct SegProp {
3622.     struct Node { int val, prop; };
3623.
3624.     vector<Node>tree;
3625.     void init(int L, int R, int pos, ll val[]) {
3626.         if(L == R) {
3627.             tree[pos].val = 0;
3628.             tree[pos].prop = 0;
3629.             return;
3630.         }
3631.
3632.         int mid = (L+R)>>1;
3633.         init(L, mid, pos<<1, val);
3634.         init(mid+1, R, pos<<1|1, val);
3635.         tree[pos].val = tree[pos].prop = 0;
3636.     }
3637.
3638.     int flipProp(int parentVal, int childVal) {
3639.         if(parentVal == childVal) return 0;
3640.         return parentVal;
3641.     }
3642.
3643.     void propagate(int L, int R, int pos) {
3644.         if(tree[pos].prop == 0 || L == R) // If no propagation tag
3645.             return; // or leaf node, then no need to change
3646.         int mid = (L+R)>>1;
3647.         tree[pos<<1].val = (mid-L+1) - tree[pos<<1].val; // Set left &
right child value
3648.         tree[pos<<1|1].val = (R-mid) - tree[pos<<1|1].val;
3649.         tree[pos<<1].prop = flipProp(tree[pos].prop, tree[pos<<1].prop); // Flip child
prop according to problem
3650.         tree[pos<<1|1].prop = flipProp(tree[pos].prop, tree[pos<<1|1].prop);
3651.         tree[pos].prop = 0; // Clear parent
propagation tag
3652.     }
3653.
3654.     void update(int L, int R, int l, int r, int pos) {
3655.         if(r < L || R < l) return;
3656.         propagate(L, R, pos);
3657.         if(l <= L && R <= r) {

```

```

3658.         tree[pos].val = (R-L+1) - tree[pos].val;    // Value updated
3659.         tree[pos].prop = 1;                          // Propagation tag set
3660.         return;
3661.     }
3662.     int mid = (L+R)>>1;
3663.     update(L, mid, l, r, pos<<1);
3664.     update(mid+1, R, l, r, pos<<1|1);
3665.     tree[pos].val = tree[pos<<1].val + tree[pos<<1|1].val;
3666. }
3667.
3668. int querySum(int L, int R, int l, int r, int pos) {
3669.     if(r < L || R < l) return 0;
3670.     propagate(L, R, pos);
3671.     if(l <= L && R <= r) return tree[pos].val;
3672.     int mid = (L+R)>>1;
3673.     int lft = querySum(L, mid, l, r, pos<<1);
3674.     int rht = querySum(mid+1, R, l, r, pos<<1|1);
3675.     return lft+rht;
3676. };
3677.
3678. // ----- Segment Tree Range Maximum Sum -----
3679. struct SegTreeRMS {
3680.     struct node {
3681.         ll sum, prefix, suffix, ans;
3682.
3683.         node(ll val = 0) {
3684.             sum = prefix = suffix = ans = val;
3685.         }
3686.
3687.         void merge(node left, node right) {
3688.             sum = left.sum + right.sum;
3689.             prefix = max(left.prefix, left.sum+right.prefix);
3690.             suffix = max(right.suffix, right.sum+left.suffix);
3691.             ans = max(left.ans, max(right.ans, left.suffix+right.prefix));
3692.         };
3693.
3694.         vector<node> tree;
3695.         void init(int pos, int l, int r, ll val[]) {
3696.             if(l == r) {
3697.                 tree[pos] = node(val[l]);
3698.                 return;
3699.             }
3700.             int mid = (l+r)/2;
3701.             init(pos*2, l, mid, val);
3702.             init(pos*2+1, mid+1, r, val);
3703.             tree[pos] = node(-INF);
3704.             tree[pos].merge(tree[pos*2], tree[pos*2+1]);
3705.         }
3706.
3707.         void update(int pos, int l, int r, int x, int val) {
3708.             if(x < l || r < x) return;
3709.             if(l == r && l == x) {
3710.                 tree[pos] = node(val);
3711.                 return;
3712.             }
3713.             int mid = (l+r)/2;
3714.             update(pos*2, l, mid, x, val);
3715.             update(pos*2+1, mid+1, r, x, val);

```

```

3716.         tree[pos] = node(-INF);
3717.         tree[pos].merge(tree[pos*2], tree[pos*2+1]);
3718.     }
3719.
3720.     node query(int pos, int l, int r, int x, int y) {
3721.         if(r < x || y < l)         return node(-INF);
3722.         if(x <= l && r <= y)       return tree[pos];
3723.         int mid = (l+r)/2;
3724.         node lft = query(pos*2, l, mid, x, y);
3725.         node rht = query(pos*2+1, mid+1, r, x, y);
3726.         node parent = node(-INF);
3727.         parent.merge(lft, rht);
3728.         return parent;
3729.     };
3730.
3731. // Segment Tree Insert/Remove value, Find I'th Value
3732. struct SegTreeInsertRemove { // Finds/Deletes I'th value from
array/SegTree
3733.     int tree[MAX*4];
3734.     void init(int pos, int L, int R) {
3735.         if(L == R) {
3736.             tree[pos] = 1;
3737.             return;
3738.         }
3739.         int mid = (L+R)>>1;
3740.         init(pos<<1, L, mid);
3741.         init(pos<<1|1, mid+1, R);
3742.         tree[pos] = tree[pos<<1]+tree[pos<<1|1];
3743.     }
3744.     int SearchVal(int pos, int L, int R, int I, bool removeVal = 0) { // Find I'th
value in Segment Tree, removes it if removeVal = 1
3745.         if(L == R) {
3746.             tree[pos] = (removeVal ? 0:1);
3747.             return L;
3748.         }
3749.         int mid = (L+R)>>1;
3750.         if(I <= tree[pos<<1]) {
3751.             int idx = SearchVal(pos<<1, L, mid, I, removeVal);
3752.             if(removeVal) tree[pos] = tree[pos<<1] + tree[pos<<1|1];
3753.             return idx;
3754.         }
3755.         else {
3756.             int idx = SearchVal(pos<<1|1, mid+1, R, I-tree[pos<<1], removeVal);
3757.             if(removeVal) tree[pos] = tree[pos<<1] + tree[pos<<1|1];
3758.             return idx;
3759.         };
3760.
3761. // Segment Tree Range Bit flip, set, reset and Query
3762. // propataion tags:
3763. // 0 - no change, 1 - all set to one, 2 - all set to zero, 3 - all need to be flipped
3764. struct RangeBitQuery {
3765.     vector<pair<int, int> >tree;
3766.     RangeBitQuery() { tree.resize(MAX*4); }
3767.
3768.     void init(int pos, int L, int R, string &s) {
3769.         tree[pos].second = 0;
3770.         if(L == R) {
3771.             tree[pos].first = (s[L] == '1');

```

```

3772.         return;
3773.     }
3774.     int mid = (L+R)>>1;
3775.     init(pos<<1, L, mid, s);
3776.     init(pos<<1|1, mid+1, R, s);
3777.     tree[pos].first = tree[pos<<1].first + tree[pos<<1|1].first;
3778. }
3779.
3780. int Convert(int tag) {          // This function generates output tag of child node if
the parent node is set to 3 (fipped)
3781.     if(tag == 1) return 2;
3782.     if(tag == 2) return 1;
3783.     if(tag == 3) return 0;
3784.     return 3;
3785. }
3786.
3787. // On every layer of update or query, this Propagation func should be called to pre-
process previous left off operations
3788. void Propagate(int L, int R, int parent) {          // Propagate parent node to child nodes
(left and right)
3789.     if(tree[parent].second == 0) return;          // and sets parent node's propagation
tag to 0
3790.     int mid = (L+R)>>1;
3791.     int lft = parent<<1, rht = parent<<1|1;
3792.     if(tree[parent].second == 1) {
3793.         tree[lft].first = mid-L+1;
3794.         tree[rht].first = R-mid;
3795.     }
3796.     else if(tree[parent].second == 2)
3797.         tree[lft].first = tree[rht].first = 0;
3798.     else if(tree[parent].second == 3) {
3799.         tree[lft].first = (mid-L+1) - tree[lft].first;
3800.         tree[rht].first = (R-mid) - tree[rht].first;
3801.     }
3802.     if(L != R) {          // If the child nodes also contain
propagate tag (and the childs are not leaf node)
3803.         if(tree[parent].second == 1 || tree[parent].second == 2)
3804.             tree[lft].second = tree[rht].second = tree[parent].second;
3805.         else {
3806.             tree[lft].second = Convert(tree[lft].second);
3807.             tree[rht].second = Convert(tree[rht].second);
3808.         }
3809.         tree[parent].second = 0;          // Parent node's
prop tag set to zero
3810.         if(L!=R) tree[parent].first = tree[lft].first + tree[rht].first;    // If this is
not the leaf node, calculate child node's sum
3811.     }
3812.
3813. void updateOn(int pos, int L, int R, int l, int r) {          // Turn on bits in
range [l, r]
3814.     if(r < L || R < l || L > R) return;
3815.     Propagate(L, R, pos);
3816.     if(l <= L && R <= r) {
3817.         tree[pos].first = (R-L+1);
3818.         tree[pos].second = 1;
3819.         return;
3820.     }
3821.     int mid = (L+R)>>1;

```

```

3822.     updateOn(pos<<1, L, mid, l, r);
3823.     updateOn(pos<<1|1, mid+1, R, l, r);
3824.     tree[pos].first = tree[pos<<1].first + tree[pos<<1|1].first;
3825. }
3826.
3827. void updateOff(int pos, int L, int R, int l, int r) { // Turn off bits in
range [l, r]
3828.     if(r < L || R < l || L > R) return;
3829.     Propagate(L, R, pos);
3830.     if(l <= L && R <= r) {
3831.         tree[pos].first = 0;
3832.         tree[pos].second = 2;
3833.         return;
3834.     }
3835.     int mid = (L+R)>>1;
3836.     updateOff(pos<<1, L, mid, l, r);
3837.     updateOff(pos<<1|1, mid+1, R, l, r);
3838.     tree[pos].first = tree[pos<<1].first + tree[pos<<1|1].first;
3839. }
3840.
3841. void updateFlip(int pos, int L, int R, int l, int r) { // Flip bits in
range [l, r]
3842.     if(r < L || R < l || L > R) return;
3843.     Propagate(L, R, pos);
3844.     if(l <= L && R <= r) {
3845.         tree[pos].first = abs(R-L+1 - tree[pos].first);
3846.         tree[pos].second = 3;
3847.         return;
3848.     }
3849.     int mid = (L+R)>>1;
3850.     updateFlip(pos<<1, L, mid, l, r);
3851.     updateFlip(pos<<1|1, mid+1, R, l, r);
3852.     tree[pos].first = tree[pos<<1].first + tree[pos<<1|1].first;
3853. }
3854.
3855. int querySum(int pos, int L, int R, int l, int r) { // Returns number of
set bit in range [l, r]
3856.     if(r < L || R < l || L > R) return 0;
3857.     Propagate(L, R, pos);
3858.     if(l <= L && R <= r) return tree[pos].first;
3859.     int mid = (L+R)>>1;
3860.     return querySum(pos<<1, L, mid, l, r) + querySum(pos<<1|1, mid+1, R, l, r);
3861. };
3862.
3863. // Merge Sort Tree
3864. struct MergeSortTree {
3865.     vector<int>tree[MAX*4];
3866.
3867.     void init(int pos, int l, int r, ll val[]) {
3868.         tree[pos].clear(); // Clears past values
3869.         if(l == r) {
3870.             tree[pos].push_back(val[l]);
3871.             return;
3872.         }
3873.
3874.         int mid = (l+r)>>1;
3875.         init(pos<<1, l, mid, val);
3876.         init(pos<<1|1, mid+1, r, val);

```



```

3877.         merge(tree[pos<<1].begin(), tree[pos<<1].end(), tree[pos<<1|1].begin(),
tree[pos<<1|1].end(), back_inserter(tree[pos]));
3878.     }
3879.
3880.     int query(int pos, int l, int r, int L, int R, int k) {
3881.         if(r < L || R < l) return 0;
3882.         if(L <= l && r <= R)
3883.             return (int)tree[pos].size() - (upper_bound(tree[pos].begin(), tree[pos].end(),
k) - tree[pos].begin()); // MODIFY
3884.         int mid = (l+r)>>1;
3885.         return query(pos<<1, l, mid, L, R, k) + query(pos<<1|1, mid+1, r, L, R, k);
3886.     };
3887.
3888.
3889. // Segment Tree Sequence (Lazy Propagation):: Contains sequence A + 2A + 3A + ..... nA
3890. struct SegTreeSeq {
3891.     vector<ll>sum, prop;
3892.
3893.     void Resize(int n) {
3894.         sum.resize(n*5);
3895.         prop.resize(n*5);
3896.     }
3897.
3898.     ll intervalSum(ll l, ll r, ll val) {
3899.         ll interval = (r*(r+1))/2LL - (l*(l-1))/2LL;
3900.         return (interval*val+MOD)%MOD;
3901.     }
3902.
3903.     void propagate(int pos, int l, int r) {
3904.         if(prop[pos] == 0 || l == r) return;
3905.         int mid = (l+r)>>1;
3906.
3907.         sum[pos<<1] = (sum[pos<<1] + intervalSum(l, mid, prop[pos]))%MOD;
3908.         sum[pos<<1|1] = (sum[pos<<1|1] + intervalSum(mid+1, r, prop[pos]))%MOD;
3909.         prop[pos<<1] = (prop[pos<<1] + prop[pos])%MOD;
3910.         prop[pos<<1|1] = (prop[pos<<1|1] + prop[pos])%MOD;
3911.         prop[pos] = 0;
3912.     }
3913.
3914.     void init(int pos, int l, int r, ll val[]) {
3915.         sum[pos] = prop[pos] = 0;
3916.         if(l == r) {
3917.             sum[pos] = (val[l]*1)%MOD;
3918.             return;
3919.         }
3920.         int mid = (l+r)>>1;
3921.         init(pos<<1, l, mid, val);
3922.         init(pos<<1|1, mid+1, r, val);
3923.         sum[pos] = (sum[pos<<1] + sum[pos<<1|1])%MOD;
3924.     }
3925.
3926.     void update(int pos, int l, int r, int L, int R, ll val) { // Range Update
3927.         if(r < L || R < l) return;
3928.         propagate(pos, l, r);
3929.         if(L <= l && r <= R) {
3930.             sum[pos] = (intervalSum(l, r, val) + sum[pos])%MOD;
3931.             prop[pos] = (val + prop[pos])%MOD;
3932.             return;

```

```

3933.     }
3934.     int mid = (l+r)>>1;
3935.     update(pos<<1, l, mid, L, R, val);
3936.     update(pos<<1|1, mid+1, r, L, R, val);
3937.     sum[pos] = (sum[pos<<1] + sum[pos<<1|1])%MOD;
3938. }
3939.
3940. ll query(int pos, int l, int r, int L, int R) {    // Range Query
3941.     if(r < L || R < l || L > R) return 0;
3942.     propagate(pos, l, r);
3943.     if(L <= l && r <= R) return sum[pos];
3944.     int mid = (l+r)>>1;
3945.     return (query(pos<<1, l, mid, L, R) + query(pos<<1|1, mid+1, r, L, R))%MOD;
3946. };
3947.
3948. // Segment Tree Bracket Sequencing, Modify position bracket and check if it is valid
3949. struct BracketTree {
3950.     struct node{
3951.         int BrcStart, BrcEnd;                // number of start bracket, number of end
3952.         bool isOk = 0;                      // is the sequence valid
3953.
3954.         node(int a = 0, int b = 0) {
3955.             BrcStart = a;
3956.             BrcEnd = b;
3957.             isOk = (BrcStart == 0 && BrcEnd == 0);
3958.         }
3959.         node(char c) {
3960.             if(c == '(') BrcStart = 1, BrcEnd = 0;
3961.             else BrcStart = 0, BrcEnd = 1;
3962.         }
3963.         void mergeNode(node lft, node rht) {
3964.             if(lft.isOk && rht.isOk)
3965.                 BrcStart = 0, BrcEnd = 0, isOk = 1;
3966.             else {
3967.                 int match = min(lft.BrcStart, rht.BrcEnd);
3968.                 BrcStart = lft.BrcStart - match + rht.BrcStart;
3969.                 BrcEnd = lft.BrcEnd + rht.BrcEnd - match;
3970.                 (BrcStart == 0 && BrcEnd == 0) ? isOk = 1: isOk = 0;
3971.             }
3972.         };
3973.         node tree[MAX*4];
3974.         void init(int pos, int L, int R, char s[]) {
3975.             if(L == R) {
3976.                 tree[pos] = node(s[L]);
3977.                 return;
3978.             }
3979.             int mid = (L+R)>>1;
3980.             init(pos<<1, L, mid, s);
3981.             init(pos<<1|1, mid+1, R, s);
3982.             tree[pos].mergeNode(tree[pos<<1], tree[pos<<1|1]);
3983.         }
3984.         void update(int pos, int L, int R, int idx, char val) {    // idx : index of the
3985.             // changed value
3986.             if(idx < L || R < idx) return;    // val : changed bracket
3987.             // sequence in char ( or )
3988.             if(L == R && L == idx) {
3989.                 tree[pos] = node(val);

```

```

3988.         return;
3989.     }
3990.     int mid = (L+R)>>1;
3991.     update(pos<<1, L, mid, idx, val);
3992.     update(pos<<1|1, mid+1, R, idx, val);
3993.     tree[pos].mergeNode(tree[pos<<1], tree[pos<<1|1]);
3994. }
3995. bool isValid() { // Returns True if sequence is valid
3996.     return tree[1].isOk;
3997. };
3998.
3999. // Outputs Largest Balanced Bracket Sequence in range [L, R]
4000. struct MaxBracketSeq {
4001.     struct node {
4002.         ll lftBracket, rhtBracket, Max;
4003.         node(ll lft=0, ll rht=0, ll Max=0) {
4004.             this->lftBracket = lft;
4005.             this->rhtBracket = rht;
4006.             this->Max = Max;
4007.         };
4008.
4009.         node tree[MAX*4];
4010.         node Merge(const node &lft, const node &rht) {
4011.             ll common = min(lft.lftBracket, rht.rhtBracket);
4012.             ll lftBracket = lft.lftBracket + rht.lftBracket - common;
4013.             ll rhtBracket = lft.rhtBracket + rht.rhtBracket - common;
4014.             return node(lftBracket, rhtBracket, lft.Max+rht.Max+common);
4015.         }
4016.
4017.         void init(ll pos, ll l, ll r, char s[]) {
4018.             if(l == r) {
4019.                 if(s[l-1] == '(') tree[pos] = node(1, 0, 0);
4020.                 else tree[pos] = node(0, 1, 0);
4021.                 return;
4022.             }
4023.             ll mid = (l+r)>>1;
4024.             init(pos<<1, l, mid, s);
4025.             init(pos<<1|1, mid+1, r, s);
4026.             tree[pos] = Merge(tree[pos<<1], tree[pos<<1|1]);
4027.         }
4028.
4029.         node query(ll pos, ll l, ll r, ll L, ll R) {
4030.             if(r < L || R < l) return node();
4031.             if(L <= l && r <= R) return tree[pos];
4032.             ll mid = (l+r)>>1;
4033.             node lft = query(pos<<1, l, mid, L, R);
4034.             node rht = query(pos<<1|1, mid+1, r, L, R);
4035.             return Merge(lft, rht);
4036.         }
4037.
4038.         int MaxSequence(int SEQ_SIZE, int l, int r) {
4039.             return 2*query(1, 1, SEQ_SIZE, l, r).Max;
4040.         };
4041.
4042.         // Path Compression Basics
4043.         // in segment tree comparison of index must be checked like (where l, r is the query range):
4044.         // outside of range [l, r] : r < point[L] || point[R] < l
4045.         // inside of range [l, r] : l <= point[L] && point[R] <= r

```

```

4046. // The Queries {l, r} will be in a queue, and processed after CompressPath and
      initialization is done
4047.
4048. void CompressPath(vector<int> &point) { // point contains
      all left and right boundary and query boundaries
4049.     point.push_back(0); // push_back a
      minimum value which is lower than input values
4050.     sort(point.begin(), point.end()); // so that the input
      values start from index 1
4051.     point.erase(unique(point.begin()+1, point.end()), point.end()); // Only unique
      points taken, this will be the compressed points
4052. }
4053.
4054. // Finding Number of Uniques in Range + OFFLINE processing
4055.
4056. struct FindUnique {
4057.     int tree[4*MAX], prop[4*MAX], v[MAX], IDX[MAX];
4058.     map<int, vector<int> >Map;
4059.     map<pair<int, int>, int>Ans;
4060.     vector<pair<int, int> > Query;
4061.
4062.     void init() {
4063.         memset(IDX, -1, sizeof IDX);
4064.         memset(tree, 0, sizeof tree);
4065.         Ans.clear(), Map.clear(), Query.clear();
4066.     }
4067.     void update(int pos, int L, int R, int idx, int val) {
4068.         if(idx < L || R < idx) return;
4069.         if(L == R) {
4070.             tree[pos] += val;
4071.             return;
4072.         }
4073.         int mid = (L+R)>>1;
4074.         update(pos<<1, L, mid, idx, val);
4075.         update(pos<<1|1, mid+1, R, idx, val);
4076.         tree[pos] = tree[pos<<1] + tree[pos<<1|1];
4077.     }
4078.     int query(int pos, int L, int R, int l, int r) {
4079.         if(r < L || R < l) return 0;
4080.         if(l <= L && R <= r) return tree[pos];
4081.         int mid = (L+R)>>1;
4082.         int lft = query(pos<<1, L, mid, l, r);
4083.         int rht = query(pos<<1|1, mid+1, R, l, r);
4084.         return lft+rht;
4085.     }
4086.     void ArrayInput(int SZ) {
4087.         for(int i = 1; i <= SZ; ++i) scanf("%d", &v[i]);
4088.     }
4089.     void QueryInput(int q) {
4090.         int l, r;
4091.         while(q--) {
4092.             scanf("%d %d", &l, &r);
4093.             Query.push_back(make_pair(l, r));
4094.             Map[r].push_back(l); // Used for sorting
4095.         }
4096.     void GenAns(int SZ) {
4097.         map<int, vi> :: iterator it;
4098.         int lPos = 0;

```

```

4099.         for(it = Map.begin(); it != Map.end(); ++it) {           // For each query's right points
4100.             while(lPos < it->first) {                               // Update from last left
position to this queries right position
4101.                 lPos++;
4102.                 if(IDX[v[lPos]] == -1) {
4103.                     IDX[v[lPos]] = lPos;
4104.                     update(1, 1, SZ, lPos, 1);                     // if new value found, increment
1 to the
4105.                 }
4106.                 else {
4107.                     int pastIDX = IDX[v[lPos]];
4108.                     IDX[v[lPos]] = lPos;
4109.                     update(1, 1, SZ, pastIDX, -1);                 // if value found previous,
then remove 1 from previous index (add -1)
4110.                     update(1, 1, SZ, lPos, 1);                     // add 1 to the new position
4111.                 }}
4112.                 for(int i = 0; i < (int)(it->second).size(); ++i)    // Range sum query for
all queries that ends on this point
4113.                     Ans[make_pair(it->second[i], it->first)] = query(1, 1, SZ, it->second[i],
it->first);
4114.             }}
4115.         void PrintAns() {
4116.             for(int i = 0; i < (int)Query.size(); ++i)              // Output according to
input query
4117.                 printf("%d\n", Ans[mp(Query[i].first, Query[i].second)]);
4118.     };
4119.
4120. struct STreeMultipleOf3 {
4121.     int tree[4*MAX][3], prop[4*MAX];
4122.     void init(int pos, int L, int R) {
4123.         if(L == R) {
4124.             tree[pos][0] = 1, tree[pos][1] = tree[pos][2] = 0;
4125.             return;
4126.         }
4127.         int mid = (L+R)>>1;
4128.         init(pos<<1, L, mid);
4129.         init(pos<<1|1, mid+1, R);
4130.         for(int i = 0; i < 3; ++i)
4131.             tree[pos][i] = tree[pos<<1][i] + tree[pos<<1|1][i];
4132.     }
4133.     void shiftVal(int pos, int step) {
4134.         step %= 3;
4135.         if(step == 0) return;
4136.         swap(tree[pos][2], tree[pos][1]);
4137.         swap(tree[pos][1], tree[pos][0]);
4138.         if(step == 2) {
4139.             swap(tree[pos][2], tree[pos][1]);
4140.             swap(tree[pos][1], tree[pos][0]);
4141.         }
4142.     void propagate(int pos, int L, int R) {
4143.         if(L == R || prop[pos] == 0) return;
4144.         shiftVal(pos<<1, prop[pos]), shiftVal(pos<<1|1, prop[pos]);
4145.         prop[pos<<1] += prop[pos], prop[pos<<1|1] += prop[pos];
4146.         prop[pos] = 0;
4147.     }
4148.     void update(int pos, int L, int R, int l, int r) {              // update l to r by 1
4149.         if(r < L || R < l) return;
4150.         if(prop[pos] != 0) propagate(pos, L, R);

```

```

4151.         if(l <= L && R <= r) {
4152.             shiftVal(pos, 1);
4153.             prop[pos] += 1;
4154.             return;
4155.         }
4156.         int mid = (L+R)>>1;
4157.         update(pos<<1, L, mid, l, r);
4158.         update(pos<<1|1, mid+1, R, l, r);
4159.         for(int i = 0; i < 3; ++i)
4160.             tree[pos][i] = tree[pos<<1][i] + tree[pos<<1|1][i];
4161.     }
4162.     int query(int pos, int L, int R, int l, int r) {           // return number of
multiple of 3 in range l to r
4163.         if(r < L || R < l) return 0;
4164.         propagate(pos, L, R);
4165.         if(l <= L && R <= r) return tree[pos][0];
4166.         int mid = (L+R)>>1;
4167.         int lft = query(pos<<1, L, mid, l, r);
4168.         int rht = query(pos<<1|1, mid+1, R, l, r);
4169.         return lft+rht;
4170.     }
4171.
4172. // CS Academy Candles :https://csacademy.com/contest/archive/task/candles/statement/
4173. struct SortedST {                                           // Performs -1 from n nodes and
keeps nodes sorted (descending order)
4174.     struct node { int val, prop; };
4175.     node tree[5*MAX];
4176.     void init(int pos, int l, int r, int val[]) {           // val[] must be sorted from hi to
low
4177.         if(l == r) {
4178.             tree[pos].val = val[l];
4179.             tree[pos].prop = 0;
4180.             return;
4181.         }
4182.         int mid = (l+r)>>1;
4183.         init(pos<<1, l, mid, val), init(pos<<1|1, mid+1, r, val);
4184.         tree[pos].val = max(tree[pos<<1].val, tree[pos<<1|1].val);
4185.         tree[pos].prop = 0;
4186.     }
4187.     void propagate(int pos, int l, int r) {
4188.         if(tree[pos].prop == 0 || l == r) {
4189.             tree[pos].prop = 0;
4190.             return;
4191.         }
4192.         tree[pos<<1|1].prop += tree[pos].prop;
4193.         tree[pos<<1].prop += tree[pos].prop;
4194.         tree[pos<<1].val += tree[pos].prop;
4195.         tree[pos<<1|1].val += tree[pos].prop;
4196.         tree[pos].val = max(tree[pos<<1].val, tree[pos<<1|1].val);
4197.         tree[pos].prop = 0;
4198.     }
4199.     int findVal(int pos, int l, int r, int idx) {           // Finds value in index idx
4200.         if(l == r) return tree[pos].val;
4201.         propagate(pos, l, r);
4202.         int mid = (l+r)>>1;
4203.         if(idx <= mid) return findVal(pos<<1, l, mid, idx);
4204.         else return findVal(pos<<1|1, mid+1, r, idx);
4205.     }

```

```

4206. void update(int pos, int l, int r, int L, int R, int val) {
4207.     if(r < L || R < l) return;
4208.     propagate(pos, l, r);
4209.     if(L <= l && r <= R) {
4210.         tree[pos].val += val;
4211.         tree[pos].prop += val;
4212.         return;
4213.     }
4214.     int mid = (l+r)>>1;
4215.     update(pos<<1, l, mid, L, R, val);
4216.     update(pos<<1|1, mid+1, r, L, R, val);
4217.     tree[pos].val = max(tree[pos<<1].val, tree[pos<<1|1].val);
4218. }
4219. int rightMost(int pos, int l, int r, int val) { // Finds rightmost value in
tree that contains val
4220.     if(l == r) return l;
4221.     if(tree[pos].val < val) return 0;
4222.     propagate(pos, l, r);
4223.     int mid = (l+r)>>1;
4224.     if(tree[pos<<1|1].val >= val) return rightMost(pos<<1|1, mid+1, r, val);
4225.     return rightMost(pos<<1, l, mid, val);
4226. }
4227. bool MinusQuery(int q, int n) { // Decreases q nodes by 1
4228.     if(q > n) return 0;
4229.     int posVal = findVal(1, 1, n, q);
4230.     if(posVal <= 0) return 0;
4231.     int r = rightMost(1, 1, n, posVal);
4232.     int l = rightMost(1, 1, n, posVal+1);
4233.     int rem = q - 1;
4234.     if(l >= 1) update(1, 1, n, 1, l, -1);
4235.     if(r-rem+1 <= r) update(1, 1, n, r-rem+1, r, -1);
4236.     return 1;
4237. }
4238.
4239. // Sqrt Decomposition
4240. // Problem: https://www.codechef.com/problems/CHEFEXQ
4241.
4242. // Operations:
4243. // 1 : Update value x at pos i
4244. // 2 : Find subarray XOR of value k from index 1 to r (All Subarray starts from 1)
4245. // Approach:
4246. // 1 : All segment consecutive xor is calculated in Seg array
4247. // : All segment consecutive xor is also counted on SegMap
4248. // 2 : Updates are done on each Decomposed segment array
4249. // 3 : Queries are combined from all Decomposed array in the range
4250.
4251. //----- Sqrt Decompose Functions Start-----
-----//
4252.
4253. int BlockSize, Seg[1010][1010]; // BlockSize is the size of each Block
4254. int SegMap[330][1110007] = {0};
4255.
4256. void Update(int v[], int l, int val) { // Updates value in position l : val
4257.     int idx = l/BlockSize; // Block Index
4258.     int lft = (l/BlockSize)*BlockSize; // The leftmost index of array v, which
is the 0 position of Segment idx
4259.     v[l] = val; // Setting value to default array to
ease

```

```

4260.
4261.     // Clear full block and re-calculate           // Using memset in large array will
cause TLE
4262.     SegMap[idx][Seg[idx][0]]--;                     // Decreasing previous value
4263.     Seg[idx][0] = v[lft];
4264.     SegMap[idx][v[lft++]]++;                       // Increasing with new value
4265.     for(int i = 1; i < BlockSize; ++i, ++lft) {
4266.         SegMap[idx][Seg[idx][i]]--;
4267.         Seg[idx][i] = Seg[idx][i-1] ^ v[lft];
4268.         SegMap[idx][Seg[idx][i]]++;
4269.     }}
4270.
4271. int Query(int l, int r, int k) {                     // Query in range l -- r for k
4272.     int Count = 0, val = 0;
4273.     while(l%BlockSize != 0 && l < r) {               // if l partially lies inside of a sqrt
segment
4274.         //cout << "P1" << endl;
4275.         Count += (Seg[l/BlockSize][l%BlockSize] == k);
4276.         val = val^Seg[l/BlockSize][l%BlockSize];
4277.         ++l;
4278.     }
4279.     while(l+BlockSize <= r) {                       // for all full sqrt segment
4280.         Count += SegMap[l/BlockSize][k^val];
4281.         val ^= Seg[l/BlockSize][BlockSize-1];
4282.         l += BlockSize;
4283.     }
4284.     while(l <= r) {                                 // for the rightmost partial sqrt segment
values
4285.         Count += (Seg[l/BlockSize][l%BlockSize] == (k^val));
4286.         ++l;
4287.     }
4288.
4289.     return Count;
4290. }
4291. void SqrtDecompose(int v[], int len) {               // Builds Sqrt segments
4292.     int idx, pos, val = 0;
4293.     BlockSize = sqrt(len);                          // Calculating Block size
4294.     for(int i = 0; i < len; ++i) {
4295.         idx = i/BlockSize;                          // Index of block
4296.         pos = i%BlockSize;                          // Index of block element
4297.         if(pos == 0) val = 0;
4298.         val ^= v[i];
4299.         Seg[idx][pos] = val;
4300.         SegMap[idx][val]++;
4301.     }}
4302. //----- Sqrt Decompose Functions End-----
-----//
4303.
4304. int v[100100];
4305. int main() {
4306.     int n, q, idx, x, t;
4307.     sf("%d %d", &n, &q);
4308.     for(int i = 0; i < n; ++i)
4309.         sf("%d", &v[i]);
4310.     SqrtDecompose(v, n);
4311.     while(q--) {
4312.         sf("%d", &t);
4313.         if(t == 1) {

```



```

4314.         sf("%d %d", &idx, &x);
4315.         Update(v, idx-1, x);
4316.     }
4317.     else {
4318.         sf("%d %d", &idx, &x);
4319.         pf("%d\n", Query(0, idx-1, x));
4320.     }}}
4321.
4322. //Single Source Shortest Path (Negative Cycle)
4323. //Complexity : O(VE)
4324.
4325.
4326. vector<int>G[MAX], W[MAX];
4327. int V, E, dist[MAX];
4328. void bellmanFord(int source) { // If there exists disconnected graphs, then add
    a dummy source node which will
4329.     for(int i = 0; i <= V; i++) // direct to all nodes with cost 0, and run
        bellmanFord from that virtual node
4330.         dist[i] = INF; // set to -INF if max distance is needed
4331.     dist[source] = 0;
4332.     for(int i = 0; i < V-1; i++) // relax all edges V-1 times, if
        virtual node added, run V times
4333.         for(int u = 0; u < V; u++) // all the nodes, if virtual
            node added, run within u <= V
4334.             for(int j = 0; j < (int)G[u].size(); j++) {
4335.                 int v = G[u][j], w = W[u][j]; // relax edges, set to max if
                    max value needed
4336.                 if(dist[u] != INF) // if there is a negative
                    weight, then INF + negative weight < INF and INF becomes +-INF
4337.                     dist[v] = min(dist[v], dist[u]+w);
4338.             }
4339.
4340. bool hasNegativeCycle() {
4341.     for(int u = 0; u < V; u++)
4342.         for(int i = 0; i < G[u].size(); i++) { // if bellmanFord is run for max values,
            then this code will
4343.             int v = G[u][i], w = W[u][i]; // return true for positive cycle by
            adding this line
4344.             if(dist[v] > dist[u] + w) // if(dist[v] < dist[u] + w)
4345.                 return 1;
4346.         }
4347.     return 0;
4348. }
4349.
4350. bool vis[MAX][2];
4351. void negativePoint(int u) { // Works in undirected graph
4352.     queue<pair<int, bool> >q; // if vis[v][1] == 1 then there exists
        an negative cycle
4353.     q.push(make_pair(u, 0)); // vis[v][1] is true for all nodes
        which are in negative cycle and
4354.     memset(vis, 0, sizeof vis); // the nodes that can be reached from
        the negative cycle nodes
4355.     vis[u][0] = 1; // on one/more path from u to v
4356.     while(!q.empty()) {
4357.         u = q.front().first;
4358.         bool neg = q.front().second;
4359.         q.pop();
4360.         for(int i = 0; i < (int)G[u].size(); ++i) {

```

```

1361.         int v = G[u][i];
1362.         int w = W[u][i];
1363.         if(dist[v] > dist[u] + w)
1364.             neg = 1;
1365.         if(vis[v][neg])
1366.             continue;
1367.         vis[v][neg] = 1;
1368.         q.push(make_pair(v, neg));
1369.     }}}
1370.
1371. //Strongly Connected Component (Tarjan)
1372. //Complexity : O(V+E)
1373.
1374. vector<int>G[MAX], SCC;
1375. int dfs_num[MAX], dfs_low[MAX], dfsCounter, SCC_no = 0;
1376. bitset<MAX>visited;
1377. map<int, int>Component;          // For Creating new SCC (ConnectNode function)
1378.
1379. void tarjanSSC(int u) {
1380.     // Stack, here, it is implemented as vector instead
1381.     SCC.push_back(u);
1382.     // Marking node u as visited
1383.     // visited[u] marks if the node u is usable in a SCC and not used on other SCC
1384.     // if visited[u] is false, then it is used in other SCC
1385.     visited[u] = 1;
1386.     dfs_num[u] = dfs_low[u] = ++dfsCounter;
1387.     // for all Strongly Connected Component (directed graph), dfs_low[u] is same
1388.     for(int i = 0; i < (int)G[u].size(); i++) {
1389.         int v = G[u][i];
1390.         // if it is not visited yet, backtrack it
1391.         if(dfs_num[v] == 0)
1392.             tarjanSSC(v);
1393.
1394.         // visited[v] is used to check if this node is not in any other SCC
1395.         if(visited[v])
1396.             dfs_low[u] = min(dfs_low[u], dfs_low[v]);
1397.     }
1398.
1399.     // in a SCC the first node of the SCC, node u is the first node in a SCC if dfs_low[u]
1400.     == dfs_low[v]
1401.     // as we implementing stack like data structure, the nodes from top to u are on the same
1402.     SCC
1403.     if(dfs_low[u] == dfs_num[u]) {
1404.         SCC_no++;          // Component Node no. starts from 0
1405.
1406.         // ----- Use if ONLY IF ConnectNode / Printing needed -----
1407.         ---
1408.         bool first = 1;
1409.         while(1) {
1410.             int v = SCC.back();
1411.             SCC.pop_back();
1412.
1413.             // node v is used, so marking it as false, so that the ancestor nodes
1414.             // doesn't use this node to update it's value
1415.
1416.             visited[v] = 0;
1417.             // printf("%d\n", v);
1418.             Component[v] = SCC_no;          // Marking SCC nodes to as same component

```

```

1416.         if(u == v)
1417.             break;
1418.     }
1419.     // printf("\n");
1420. }
1421. }
1422.
1423. void ConnectNode() { // This function can convert Components to a new
graph (G1)
1424.     map<int, int> :: iterator it = Component.begin();
1425.
1426.     for( ; it != Component.end(); ++it) {
1427.         for(int i = 0; i < (int)G[it->first].size(); ++i) {
1428.             int v = G[it->first][i];
1429.             if(it->second == Component[v]) // No Self loop in new graph
1430.                 continue;
1431.             G1[it->second].push_back(Component[v]);
1432. }}}
1433.
1434.
1435. void RunSCC(int V) {
1436.     memset(dfs_num, 0, sizeof(dfs_num));
1437.     dfsCounter = 0;
1438.     visited.reset();
1439.     SCC_no = 0;
1440.     for(int i = 1; i <= V; i++)
1441.         if(dfs_num[i] == 0)
1442.             tarjanSSC(i);
1443. }
1444.
1445. // Fast IO with Templates
1446.
1447. #include <bits/stdc++.h>
1448. using namespace std;
1449. #define MAX 510000
1450. #define EPS 1e-9
1451. #define INF 1e7
1452. #define MOD 1000000007
1453. #define pb push_back
1454. #define mp make_pair
1455. #define fi first
1456. #define se second
1457. #define pi acos(-1)
1458. #define pf printf
1459. #define sf(XX) scanf("%lld", &XX)
1460. #define SIZE(a) ((ll)a.size())
1461. #define ALL(S) S.begin(), S.end()
1462. #define Equal(a, b) (abs(a-b) < EPS)
1463. #define Greater(a, b) (a >= (b+EPS))
1464. #define GreaterEqual(a, b) (a > (b-EPS))
1465. #define FOR(i, a, b) for(register int i = (a); i < (int)(b); ++i)
1466. #define FORR(i, a, b) for(register int i = (a); i > (int)(b); --i)
1467. #define FastIO ios_base::sync_with_stdio(false); cin.tie(NULL);
1468. #define FileRead(S) freopen(S, "r", stdin);
1469. #define FileWrite(S) freopen(S, "w", stdout);
1470. #define Unique(X) X.erase(unique(X.begin(), X.end()), X.end())
1471. #define STOLL(X) stoll(X, 0, 0)
1472.

```

```

1473. #define isOn(S, j)          (S & (1 << j))
1474. #define setBit(S, j)        (S |= (1 << j))
1475. #define clearBit(S, j)      (S &= ~(1 << j))
1476. #define toggleBit(S, j)     (S ^= (1 << j))
1477. #define lowBit(S)           (S & (-S))
1478. #define setAll(S, n)        (S = (1 << n) - 1)
1479.
1480. typedef unsigned long long ull;
1481. typedef long long ll;
1482. typedef map<int, int> mii;
1483. typedef map<ll, ll> mll;
1484. typedef map<string, int> msi;
1485. typedef vector<int> vi;
1486. typedef vector<ll> vll;
1487. typedef pair<int, int> pii;
1488. typedef pair<ll, ll> pll;
1489. typedef vector<pair<int, int> > vii;
1490. typedef vector<pair<ll, ll> > vlll;
1491.
1492. //int dx[] = {-1, 0, 1, 0}, dy[] = {0, 1, 0, -1};
1493. //int dx[] = {-1, -1, -1, 0, 0, 1, 1, 1}, dy[] = {-1, 0, 1, -1, 1, -1, 0, 1};
1494. //-----
1495.
1496. inline void fastIn(int &num) {          // Fast IO, with space and new line ignoring
1497.     bool neg = false;
1498.     register int c;
1499.     num = 0;
1500.     c = getchar_unlocked();
1501.     for( ; c != '-' && (c < '0' || c > '9'); c = getchar_unlocked());
1502.     if (c == '-') {
1503.         neg = true;
1504.         c = getchar_unlocked();
1505.     }
1506.     for( ; (c>47 && c<58); c=getchar_unlocked())
1507.         num = (num<<1) + (num<<3) + c - 48;
1508.     if(neg)
1509.         num *= -1;
1510. }
1511.
1512. inline void fastOut (long long n){
1513.     long long N = n, rev, count = 0;
1514.     rev = N;
1515.     if (N == 0) { putchar('0'); return ;}
1516.     while ((rev % 10) == 0) { count++; rev /= 10;}           //obtain the count of
the number of 0s
1517.     rev = 0;
1518.     while (N != 0) { rev = (rev<<3) + (rev<<1) + N % 10; N /= 10;} //store reverse of N in
rev
1519.     while (rev != 0) { putchar(rev % 10 + '0'); rev /= 10;}
1520.     while (count-->0) putchar('0');
1521. }
1522.
1523.
1524. // Scanf Trick
1525. // input: (alpha+omega)^2
1526. // scanf("%*([ ] ^+ ) %*([+ ] ^+ ) %s", a, b, n);
1527. // %* is used for skipping

```

```

4528. // %*[(] skipping (
4529. // %[^+] take input until +
4530. // %*[(] skipping +
4531. // %*[(^)] skipping ^ and )
4532.
4533. // Tree Max Distance Node
4534. // Set any node as root, then do dfs and find the farthest node, then again from that
    farthest node
4535. // do dfs for farthest node, the two nodes are the farthest node
4536.
4537. pii dfs(int u, int par, int d) {
4538.     pii ret(d, u); // {distance, node}
4539.     for(int i = 0; i < (int)G[u].size(); ++i)
4540.         if(G[u][i] != par)
4541.             ret = max(ret, dfs(G[u][i], u, d+W[u][i]));
4542.     return ret;
4543. }
4544.
4545. int GetDistance() {
4546.     pii left = dfs(0, -1, 0);
4547.     pii right = dfs(left.second, -1, 0);
4548.     return right.first;
4549. }
4550.
4551. // Codeforces :E. Propagating tree ( http://codeforces.com/contest/384/problem/E )
4552. // Given a tree (node 1 - n)
4553. // perform two operations:
4554. // 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
4555. // in other words, add value x to all childs where (parentLevel%2 == childLevel%2), add -val
    otherwise
4556. // 2. Output value of node u
4557.
4558. vector<int> G[MAX];
4559. int sTime[MAX], eTime[MAX], level[MAX], cst[MAX], timer;
4560. BIT EvenNode, OddNode;
4561.
4562. /* sTime : starting time of node n
4563.    eTime : finishing time of node n
4564.    1
4565.    / \
4566.   5  6
4567.  / \
4568. 7  4
4569. / \
4570. 2  3
4571. discover nodes : {1, 5, 6, 7, 4, 2, 3}
4572. sTime[] = {1, 6, 7, 5, 2, 3, 4}    index starts from 1, i'th index contains start time of
    i'th node
4573. eTime[] = {7, 6, 7, 7, 2, 7, 4}
4574.
4575. calculate child :
4576. for node 6 : childs are in range sTime[6] - eTime[6] : 3 - 7
4577. so child nodes are : 6, 7, 4, 2, 3 (discover node index range)
4578. we don't need discover time vector to calculate distance
4579. notice, if we only update with sTime and eTime, the range update will always be right */
4580.
4581. void dfs(int u, int lvl) {

```

```

4582.     sTime[u] = ++timer;
4583.     level[u] = lvl;
4584.     for(int i = 0; i < (int)G[u].size(); ++i)
4585.         if(sTime[G[u][i]] == 0)
4586.             dfs(G[u][i], lvl+1);
4587.     eTime[u] = timer;
4588. }
4589.
4590. void AddVal(int node, int val) {
4591.     if(level[node]%2 == 0) {
4592.         EvenNode.update(sTime[node], eTime[node], val);
4593.         OddNode.update(sTime[node], eTime[node], -val);
4594.     }
4595.     else {
4596.         EvenNode.update(sTime[node], eTime[node], -val);
4597.         OddNode.update(sTime[node], eTime[node], val);
4598.     }
4599. }
4600. int GetVal(int node) {                                     // cst[node] contains initial
cost (if exists)
4601.     return cst[node] + (level[node]%2==0 ?
EvenNode.read(sTime[node]):OddNode.read(sTime[node]));
4602. }
4603.
4604.
4605. // Complete Binary Tree
4606. // Sum of distance from a node "n" such that every nodes distance from node "n" is less than
or equal to k
4607. // http://mishadoff.com/blog/dfs-on-binary-tree-array/
4608.
4609. vector<ll>v[MAX], w, sum[MAX];                             // w[i] contains weight of I'th node
4610. int n, m;
4611. void dfs(int node = 1) {                                   // node starts from 1
4612.     if(node > n) return;
4613.
4614.     ll lft = node<<1, rht = node<<1|1;
4615.     dfs(lft), dfs(rht);
4616.     ll lftSize = v[lft].size(), rhtSize = v[rht].size();
4617.     ll nodeSize = lftSize+rhtSize+1;
4618.     v[node].resize(nodeSize);
4619.     v[node][0] = 0;                                         // distance from this node to this node
4620.
4621.     //printf("node : %d, nodeSize : %d, lftSize : %d, rhtSize : %d\n", node, nodeSize,
lftSize, rhtSize);
4622.     ll l = 0, r = 0;
4623.     for(ll i = 1; i < nodeSize; ++i) {
4624.         if(l == lftSize)
4625.             v[node][i] = v[rht][r++] + w[rht];
4626.         else if(r == rhtSize)
4627.             v[node][i] = v[lft][l++] + w[lft];
4628.         else {
4629.             int lftW = v[lft][l] + w[lft], rhtW = v[rht][r] + w[rht];
4630.             if(lftW < rhtW) {
4631.                 v[node][i] = lftW;
4632.                 l++;
4633.             }
4634.             else {
4635.                 v[node][i] = rhtW;

```

```

1636.         r++;
1637.     }}}}
1638.
1639. ll single(int node, ll d, ll delta) {
1640.     if(d < 0) return 0;
1641.     ll n = upper_bound(v[node].begin(), v[node].end(), d) - v[node].begin();
1642.     return sum[node][n-1] + delta*n; // delta is the
common distance of all nodes
1643. }
1644.
1645. ll query(int node, ll k) {
1646.     ll ans = single(node, k, 0);
1647.     ll totlen = 0;
1648.     while(node/2) {
1649.         totlen += w[node];
1650.         ll tmp = single(node/2, k-totlen, totlen); // distances from
parent node
1651.         tmp -= single(node, k-totlen-w[node], totlen + w[node]); // common overlapped
distance (of child node) from parent node
1652.         ans += tmp;
1653.         node /= 2;
1654.     }
1655.     return ans;
1656. }
1657.
1658. void PreCal() { // First run dfs(), then run PreCal()
1659.     for(int i = 1; i <= n; ++i) {
1660.         sum[i].resize(v[i].size());
1661.         sum[i][0] = v[i][0];
1662.         for(int j = 1; j < SIZE(v[i]); ++j)
1663.             sum[i][j] = sum[i][j-1] + v[i][j];
1664.     }
1665.
1666. //Trie
1667. //Complexity : making a trie : O(S), searching : O(S)
1668.
1669. struct node {
1670.     bool isEnd;
1671.     node *next[11];
1672.     node() {
1673.         isEnd = false;
1674.         for(int i = 0; i < 10; i++)
1675.             next[i] = NULL;
1676.     };
1677.
1678. //trie of a string abc, ax
1679. // [start] --> [a] --> [b] --> [c] --> endMark
1680. //      |
1681. //      [x] --> endMark
1682.
1683. //creates trie, returns true if the trie we are creating is a segment of a string
1684. //to only create a trie remove lines which are comment marked
1685.
1686. bool create(char str[], int len, node *current) {
1687.     for(int i = 0; i < len; i++) {
1688.         int pos = str[i] - '0';
1689.         if(current->next[pos] == NULL)
1690.             current->next[pos] = new node();

```

```
4691.         current = current->next[pos];
4692.         if(current->isEnd) //
4693.             return true; //
4694.     }
4695.     current->isEnd = true; //
4696.     return false; //
4697. }
4698.
4699. void del(node *current) {
4700.     for(int i = 0; i < 10; i++)
4701.         if(current->next[i] != NULL) del(current->next[i]);
4702.     delete current;
4703. }
4704.
4705. void check(node *current) {
4706.     for(int i = 0; i < 10; i++) {
4707.         if(current->next[i] != NULL)
4708.             check(current->next[i]);
4709.     }
4710.     if(found) return;
4711.     if(current->isEnd && !found) {
4712.         for(int i = 0; i < 10 && !found; i++)
4713.             if(current->next[i] != NULL) {
4714.                 found = 1;
4715.             }
4716.     }
4717. int main() {
4718.     //freopen("in", "r", stdin);
4719.     //freopen("out", "w", stdout);
4720.     int t, n;
4721.     char S[15];
4722.     scanf("%d", &t);
4723.     while(t--) {
4724.         found = 0;
4725.         node* root = new node(); //important part of the code
4726.         scanf("%d", &n);
4727.         while(n--) {
4728.             scanf("%s", S);
4729.             if(!found)
4730.                 if(create(S, strlen(S), root))
4731.                     found = 1;
4732.         }
4733.         if(!found) check(root);
4734.         if(found) printf("NO\n");
4735.         else      printf("YES\n");
4736.         del(root);
4737.     }
```