```
--- Sparse Table
#include <cmath>
using namespace std;
#define MAX_N 1000                          // adjust this value as needed
#define LOG_TWO_N 10              // 2^10 > 1000, adjust this value as needed

struct RMQ {                                         // Range Minimum Query
  int _A[MAX_N], SpT[MAX_N][LOG_TWO_N];
  RMQ(int n, int A[]) {     // constructor as well as pre-processing routine
    for (int i = 0; i < n; i++) {
      _A[i] = A[i];
      SpT[i][0] = i; // RMQ of sub array starting at index i + length 2^0=1
    }
    // the two nested loops below have overall time complexity = O(n log n)
    for (int j = 1; (1<<j) <= n; j++) // for each j s.t. 2^j <= n, O(log n)
      for (int i = 0; i + (1<<j) - 1 < n; i++)     // for each valid i, O(n)
        if (_A[SpT[i][j-1]] < _A[SpT[i+(1<<(j-1))][j-1]])            // RMQ
          SpT[i][j] = SpT[i][j-1];     // start at index i of length 2^(j-1)
        else                      // start at index i+2^(j-1) of length 2^(j-1)
          SpT[i][j] = SpT[i+(1<<(j-1))][j-1];
```

```
  }

  int query(int i, int j) {
    int k = (int)floor(log((double)j-i+1) / log(2.0));    // 2^k <= (j-i+1)
    if (_A[SpT[i][k]] <= _A[SpT[j-(1<<k)+1][k]]) return SpT[i][k];
    else                                         return SpT[j-(1<<k)+1][k];
} };

int main() {
  // same example as in chapter 2: segment tree
  int n = 7, A[] = {18, 17, 13, 19, 15, 11, 20};
  RMQ rmq(n, A);
  for (int i = 0; i < n; i++)
    for (int j = i; j < n; j++)
      printf("RMQ(%d, %d) = %d\n", i, j, rmq.query(i, j));

  return 0;
}

---  Fenwick Tree/Binary Indexed Tree (1D 3 mode)
// all are 1-based indexing
/*

// mode 1: point update, range query
int bit[100100], n;

void update(int x, int v){
    for(int i = x; i <= n; i += i&-i)
        bit[i] += v;
}

int query(int x){
    int ans = 0;
    for(int i = x; i >= 1; i -= i&-i)
        ans += bit[i];

    return ans;
}

int query(int x, int y){
    return query(y) - query(x-1);
}

// mode 2: range update, point query
int bit[100100], n, a[100100];

void update(int x, int v){
    for(int i = x; i <= n; i += i&-i)
        bit[i] += v;
}

void update(int x, int y, int v){
    update(x, v);
    update(y+1, -v);
}

int query(int x){
    int ans = 0;
    for(int i = x; i >= 1; i -= i&-i)
        ans += bit[i];

    return ans + a[x];
}

*/

// mode 3: range update, range query
long long bit1[100100], bit2[100100], n;

void update(long long bit[], int x, long long v){
    for(int i = x; i <= n; i += i&-i)
        bit[i] += v;
}

void update(int x, int y, long long v){
    update(bit1, x, v);
    update(bit1, y+1, -v);
    update(bit2, x, v*(x-1));
    update(bit2, y+1, -v*y);
}
```

```
long long query(long long bit[], int x){
    long long ans = 0;
    for(int i = x; i >= 1; i -= i&-i)
        ans += bit[i];

    return ans;
}
long long query(int x){
    return query(bit1, x) * x - query(bit2, x);
}
long long query(int x, int y){
    return query(y) - query(x-1);
}
int main(){
    int t;
    cin >> t;

    while(t--){
        int c;
        cin >> n >> c;
        memset(bit1, 0, sizeof bit1);
        memset(bit2, 0, sizeof bit2);
        while(c--){
            int type, x, y, val;
            cin >> type;

            if(type == 0){
                cin >> x >> y >> val;
                update(x, y, val);
            }
            else{
                cin >> x >> y;
                cout << query(x, y) << '\n';
            }
        }
    }

    return 0;
}

--- 2D BIT
long long bit[2000][2000], n, a[2000][2000];

void update(int x, int y, int v){
    for(int i = x; i <= n; i += i&-i){
        for(int j = y; j <= n; j += j&-j){
            bit[i][j] += v;
        }
    }
}
long long query(int x, int y){
    long long ans = 0;
    for(int i = x; i >= 1; i -= i&-i){
        for(int j = y; j >= 1; j -= j&-j){
            ans += bit[i][j];
        }
    }
    return ans;
```

```
}
int query(int x1, int y1, int x2, int y2){
    x1--; y1--;
    return query(x2, y2) - query(x2, y1) - query(x1, y2) + query(x1, y1);
}

--- Segment Tree (standard + lazy)
#define l_node  ( 2*node )
#define r_node  ( 2*node + 1 )
#define mid     ( (l+r)/2 )

vector<long long> st, lazy;
int a[100100], n;

void init(int node, int l, int r){
    if(l == r){
        st[node] = a[l];
        return;
    }

    init(l_node, l, mid);
    init(r_node, mid+1, r);

    st[node] = st[l_node] + st[r_node];
}

void init(){
    st.assign(4*n, 0);
    lazy.assign(4*n, 0);

    // 1-based indexing
    init(1, 1, n);
}

void update(int node, int l, int r, int x, int y, long long val){
    if(lazy[node] != 0){
        // update this node
        st[node] += lazy[node]*(r-l+1);

        if(l != r){
            // propogate down
            lazy[l_node] += lazy[node];
            lazy[r_node] += lazy[node];
        }
        lazy[node] = 0;
    }

    if(x <= l && r <= y){   // in range, lazy update it
        // ensure this node value is correct for parent updating
        st[node] += val*(r-l+1);

        if(l != r){
            lazy[l_node] += val;
            lazy[r_node] += val;
        }
        return;
    }

    if(y < l || x > r){
```

```
        // out of range
        return;
    }

    update(l_node, l, mid, x, y, val);
    update(r_node, mid+1, r, x, y, val);

    st[node] = st[l_node] + st[r_node];
}

// range update
void update(int x, int y, int val){
    update(1, 1, n, x, y, val);
}

// point update
void update(int k, int val){
    update(1, 1, n, k, k, val);
}

long long query(int node, int l, int r, int x, int y){
    if(lazy[node] != 0){
        // update this node
        st[node] += lazy[node]*(r-l+1);

        if(l != r){
            // propogate down
            lazy[l_node] += lazy[node];
            lazy[r_node] += lazy[node];
        }
        lazy[node] = 0;
    }

    if(x <= l && r <= y){
        // in range, return value
        return st[node];
    }

    if(y < l || x > r){
        // out of range
        return 0;
    }

    long long q1 = query(l_node, l, mid, x, y);
    long long q2 = query(r_node, mid+1, r, x, y);

    return q1 + q2;
}

// range query
long long query(int x, int y){
    return query(1, 1, n, x, y);
}

--- Segment Tree (1D Implicit)
#define mid ( (l+r)/2 )

struct st_node{
    int l, r;
```

```cpp
    long long val, lazy;

    st_node *l_node, *r_node;

    st_node(int _l, int _r){
        l = _l; r = _r;
        l_node = NULL; r_node = NULL;
    }

    void expand_node(){
        if(l_node == NULL){
            l_node = new st_node(l, mid);
            r_node = new st_node(mid+1, r);
        }
    }

    void clear_lazy(){
        if(lazy != 0){
            val += lazy*(r-l+1);

            if(l != r){
                this->expand_node();

                l_node->lazy += lazy;
                r_node->lazy += lazy;
            }

            lazy = 0;
        }
    }
};

void update(st_node* node, int x, int y, long long val){
    node->clear_lazy();

    int l = node->l, r = node->r;

    if(x <= l && r <= y){    // in range, lazy update it
        // ensure this node value is correct for parent updating
        node->val += val*(r-l+1);

        if(l != r){
            node->expand_node();

            node->l_node->lazy += val;
            node->r_node->lazy += val;
        }
        return;
    }

    if(y < l || x > r){
        // out of range
        return;
    }

    node->expand_node();
    update(node->l_node, x, y, val);
    update(node->r_node, x, y, val);
```

```
    node->val = node->l_node->val + node->r_node->val;
}


long long query(st_node* node, int x, int y){
    node->clear_lazy();

    int l = node->l, r = node->r;

    if(x <= l && r <= y){
        // in range, return value
        return node->val;
    }

    if(y < l || x > r){
        // out of range
        return 0;
    }

    node->expand_node();
    long long q1 = query(node->l_node, x, y);
    long long q2 = query(node->r_node, x, y);

    return q1 + q2;
}

--- Segment Tree (2D Implicit)
// O(n log^2 n), fesible for n < 10^4 - 10^5

// change the range of st_node1 and st_node2
#define mid     ( (l+r)/2 )
#define inf     ( 1<<30 )
#define MAX     ( 1000000100 )

int n;
struct st_node2{
    int l, r, val;

    st_node2 *l_node, *r_node;

    st_node2(int _l, int _r){
        l = _l; r = _r; val = 0;
        l_node = NULL; r_node = NULL;
    }

    void expand_node(){
        if(l_node == NULL){
            l_node = new st_node2(l, mid);
            r_node = new st_node2(mid+1, r);
        }
    }

};

struct st_node1{
    int l, r;
    st_node2 *node2_root;
    st_node1 *l_node, *r_node;
```

```cpp
    st_node1(int _l, int _r){
        l = _l; r = _r;
        l_node = NULL; r_node = NULL;
        node2_root = new st_node2(1, n);
    }

    void expand_node(){
        if(l_node == NULL){
            l_node = new st_node1(l, mid);
            r_node = new st_node1(mid+1, r);
        }
    }
};

void updatey(st_node2* node, int x, int y, int val){
    int l = node->l, r = node->r;
    if(y < l || r < y){
        // out of range
        return;
    }

    if(l == y && r == y){
        node->val = max(node->val, val);
        return;
    }

    node->expand_node();
    updatey(node->l_node, x, y, val);
    updatey(node->r_node, x, y, val);

    node->val = max(node->val, val);
}

void update(st_node1* node, int x, int y, int val){
    int l = node->l, r = node->r;
    if(x < l || r < x){
        // out of range
        return;
    }

    if(l == x && r == x){
        updatey(node->node2_root, x, y, val);
        return;
    }

    node->expand_node();
    update(node->l_node, x, y, val);
    update(node->r_node, x, y, val);

    updatey(node->node2_root, x, y, val);
}

int queryy(st_node2* node, int y1, int y2){
    int l = node->l, r = node->r;
    if(y2 < l || r < y1){
        // out of range
        return 0;
    }
```

```
    if(l >= y1 && y2 >= r){
        // in range
        return node->val;
    }

    node->expand_node();
    int q1 = queryy(node->l_node, y1, y2);
    int q2 = queryy(node->r_node, y1, y2);

    // merge update
    return max(q1, q2);
}

int query(st_node1* node, int x1, int x2, int y1, int y2){
    int l = node->l, r = node->r;
    if(x2 < l || r < x1){
        // out of range
        return 0;
    }

    if(l >= x1 && x2 >= r){
        // in range
        return queryy(node->node2_root, y1, y2);
    }

    node->expand_node();
    int q1 = query(node->l_node, x1, x2, y1, y2);
    int q2 = query(node->r_node, x1, x2, y1, y2);

    // merge update
    return max(q1, q2);
}

--- Segment Tree (2D)
#define l_node  ( 2*node )
#define r_node  ( 2*node + 1 )
#define mid     ( (l+r)/2 )
#define inf     ( 1<<30 )

typedef pair<int, int> ii;

int table[600][600], n, m;
ii st[2100][2100];

ii min_max(ii node_a, ii node_b){
    return ii(max(node_a.first, node_b.first), min(node_a.second, node_b.second));
}

void inity(int node, int l, int r, int nodex, int x){
    if(l == r){
        st[nodex][node] = ii(table[x][l], table[x][l]);
        return;
    }
    inity(l_node, l, mid, nodex, x);
    inity(r_node, mid+1, r, nodex, x);
    st[nodex][node] = min_max(st[nodex][l_node], st[nodex][r_node]);
}
void initx(int node, int l, int r){
    if(l == r){
```

```
        inity(1, 0, m-1, node, 1);
        return;
    }
    initx(l_node, l, mid);
    initx(r_node, mid+1, r);
    // merge l, r node
    for(int i = 0; i < 4*m; i++){
        st[node][i] = min_max(st[l_node][i], st[r_node][i]);
    }
}
void init(){
    // 0-based index
    initx(1, 0, n-1);
}
void merge_update(int node, int l, int r, int x, int y, int nodex){
     if(y < l || r < y){
        // out of range
        return;
    }
    if(l == y && r == y){
        st[nodex][node] = min_max(st[2*nodex][node], st[2*nodex+1][node]);
        return;
    }
    merge_update(l_node, l, mid, x, y, nodex);
    merge_update(r_node, mid+1, r, x, y, nodex);
    st[nodex][node] = min_max(st[2*nodex][node], st[2*nodex+1][node]);
}
void updatey(int node, int l, int r, int x, int y, int val, int nodex){
    if(y < l || r < y){
        // out of range
        return;
    }
    if(l == y && r == y){
        st[nodex][node] = ii(val, val);
        return;
    }
    updatey(l_node, l, mid, x, y, val, nodex);
    updatey(r_node, mid+1, r, x, y, val, nodex);
    st[nodex][node] = min_max(st[nodex][l_node], st[nodex][r_node]);
}
void updatex(int node, int l, int r, int x, int y, int val){
    if(x < l || r < x){
        // out of range
        return;
    }
    if(l == x && r == x){
        updatey(1, 0, m-1, x, y, val, node);
        return;
    }
    updatex(l_node, l, mid, x, y, val);
    updatex(r_node, mid+1, r, x, y, val);
    // merge update
    merge_update(1, 0, m-1, x, y, node);
}
void update(int x, int y, int val){
    updatex(1, 0, n-1, x, y, val);
}
ii queryy(int node, int l, int r, int y1, int y2, int nodex){
    if(y2 < l || r < y1){
```

```
        // out of range
        return ii(-inf, inf);
    }
    if(l >= y1 && y2 >= r){
        // in range
        return st[nodex][node];
    }
    ii q1 = queryy(l_node, l, mid, y1, y2, nodex);
    ii q2 = queryy(r_node, mid+1, r, y1, y2, nodex);
    // merge update
    return min_max(q1, q2);
}
ii queryx(int node, int l, int r, int x1, int x2, int y1, int y2){
    if(x2 < l || r < x1){
        // out of range
        return ii(-inf, inf);
    }
    if(l >= x1 && x2 >= r){
        // in range
        return queryy(1, 0, m-1, y1, y2, node);
    }
    ii q1 = queryx(l_node, l, mid, x1, x2, y1, y2);
    ii q2 = queryx(r_node, mid+1, r, x1, x2, y1, y2);
    // merge update
    return min_max(q1, q2);
}
ii query(int x1, int y1, int x2, int y2){
    return queryx(1, 0, n-1, min(x1, x2), max(x1, x2), min(y1, y2), max(y1, y2));
}

--- PBDS
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace std;
using namespace __gnu_pbds;

typedef tree<int, null_type, less<int>, rb_tree_tag,
tree_order_statistics_node_update>     ordered_set;

int main() {
    ordered_set X;
    X.insert(1);

    cout<<*X.find_by_order(1)<<endl; // 2
    cout<<(X.end()==X.find_by_order(6))<<endl; // true

    cout<<X.order_of_key(-5)<<endl;  // 0
    return 0;
}

--- Articulation Points & Bridges
vector<int> dfs_low, articulation_vertex;
int dfsNumberCounter, dfsRoot, rootChildren;

void articulationPointAndBridge(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++;      // dfs_low[u] <= dfs_num[u]
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
```

```
    if (dfs_num[v.first] == DFS_WHITE) {                          // a tree edge
      dfs_parent[v.first] = u;
      if (u == dfsRoot) rootChildren++;  // special case, count children of root

      articulationPointAndBridge(v.first);

      if (dfs_low[v.first] >= dfs_num[u])               // for articulation point
        articulation_vertex[u] = true;            // store this information first
      if (dfs_low[v.first] > dfs_num[u])                          // for bridge
        printf(" Edge (%d, %d) is a bridge\n", u, v.first);
      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);       // update dfs_low[u]
    }
    else if (v.first != dfs_parent[u])       // a back edge and not direct cycle
      dfs_low[u] = min(dfs_low[u], dfs_num[v.first]);       // update dfs_low[u]
} }

// inside int main()
printThis("Articulation Points & Bridges (the input graph must be UNDIRECTED)");
dfsNumberCounter = 0; dfs_num.assign(V, DFS_WHITE); dfs_low.assign(V, 0);
dfs_parent.assign(V, -1); articulation_vertex.assign(V, 0);
printf("Bridges:\n");
for (int i = 0; i < V; i++)
    if (dfs_num[i] == DFS_WHITE) {
        dfsRoot = i; rootChildren = 0;
        articulationPointAndBridge(i);
        articulation_vertex[dfsRoot] = (rootChildren > 1); }       // special case
printf("Articulation Points:\n");
for (int i = 0; i < V; i++)
    if (articulation_vertex[i])
        printf(" Vertex %d\n", i);


--- DFS SCC Toposort DAG
#include <vector>
#include <stack>

using namespace std;

int height[100100], cur_num;
vector<vector<int> > graph, dag;
vector<int> dag_label, dag_max, dag_min, dfs_num, dfs_low, topological;
vector<bool> is_parent, visited;
stack<int> s;

void topological_sort(int u){
    visited[u] = true;
    for(int i = 0; i < dag[u].size(); i++){
        int v = dag[u][i];
        if(!visited[v])
            topological_sort(v);
    }
    topological.push_back(u);
}
void scc_dfs(int u){
    dfs_num[u] = dfs_low[u] = cur_num++;
    is_parent[u] = true;
    s.push(u);

    for(int i = 0; i < graph[u].size(); i++){
```

```
        int v = graph[u][i];
        if(dfs_num[v] == -1){
            scc_dfs(v);
        }
        // a back edge
        if(is_parent[v]){
            dfs_low[u] = min(dfs_low[u], dfs_low[v]);
        }
    }
    if(dfs_low[u] == dfs_num[u]){
        // root of the scc
        int cur_dag = dag_max.size(), v;
        dag_max.push_back(height[u]);
        dag_min.push_back(height[u]);
        do{
            v = s.top(); s.pop();
            is_parent[v] = false;

            dag_label[v] = cur_dag;
            dag_max[cur_dag] = max(dag_max[cur_dag], height[v]);
            dag_min[cur_dag] = min(dag_min[cur_dag], height[v]);
        }while(u != v);
    }
}
int main(){
    int t;
    cin >> t;

    while(t--){
        int n, m;
        cin >> n >> m;

        for(int i = 1; i <= n; i++)
            cin >> height[i];

        graph.assign(n+1, vector<int> ());
        while(m--){
            int u, v;
            cin >> u >> v;
            graph[u].push_back(v);
        }

        // find all scc
        dag_max.clear();
        dag_min.clear();
        dfs_num.assign(n+1, -1);
        dfs_low.assign(n+1, -1);
        dag_label.assign(n+1, -1);
        is_parent.assign(n+1, false);
        cur_num = 1;
        for(int i = 1; i <= n; i++){
            if(dfs_num[i] == -1)
                scc_dfs(i);
        }

        // build dag, dont care repeated edge,
        // just repeat it, not going to harm u in anyway
        dag.assign(dag_max.size(), vector<int> ());
        for(int i = 1; i <= n; i++){
```

```
            int u = dag_label[i];
            for(int j = 0; j < graph[i].size(); j++){
                int v = dag_label[graph[i][j]];

                if(u == v) continue;
                dag[u].push_back(v);
            }
        }

        topological.clear();
        visited.assign(dag_max.size(), false);
        for(int i = 0; i < dag_max.size(); i++){
            if(!visited[i])
                topological_sort(i);
        }

        int ans = 0;
        for(vector<int>::iterator it = topological.begin(); it != topological.end(); it++){
            int u = *it;
            for(int i = 0; i < dag[u].size(); i++){
                int v = dag[u][i];
                dag_max[u] = max(dag_max[u], dag_max[v]);
            }
            ans = max(ans, dag_max[u] - dag_min[u]);
        }
        cout << ans << endl;
    }
    return 0;
}

--- SPFA
#include <vector>
#include <queue>
using namespace std;
typedef pair<int, int> ii;
vector<vector<ii> > graph;
int main() {
    int tc;
    cin >> tc;
    while(tc--){
        int n, m;
        cin >> n >> m;
        graph.assign(n, vector<ii> () );
        while(m--){
            int a, b, t;
            cin >> a >> b >> t;
            graph[a].push_back(ii(b, t));
        }
        vector<int> dist(n, 1<<30);     dist[0] = 0;
        vector<int> queuetime(n, 0);    queuetime[0] = 1;
        vector<bool> inqueue(n, 0);     inqueue[0] = true;
        queue<int> q;                   q.push(0);
        bool negativecycle = false;

        while(!q.empty() && !negativecycle){
            int u = q.front(); q.pop();
            inqueue[u] = false;

            for(int i = 0; i < graph[u].size(); i++){
```

```
                int v = graph[u][i].first, w = graph[u][i].second;

                if(dist[u] + w < dist[v]){
                    dist[v] = dist[u] + w;

                    if(!inqueue[v]){
                        q.push(v);
                        queuetime[v]++;
                        inqueue[v] = true;

                        if(queuetime[v] == n+2){
                            negativecycle = true;
                            break;
        }   }   }   }   }
        cout << (negativecycle?"possible":"not possible") << endl;
    }
    return 0;
}

--- HLD
// change segment tree merge, query merge
#include <cstring>
#include <cstdio>
#include <vector>

using namespace std;

// change this
#define MAXN 100100
#define LN   20

#define l_node  ( 2*node )
#define r_node  ( 2*node + 1 )
#define mid     ( (l+r)/2 )

typedef pair<int, int> ii;

int n;
vector<vector<ii> > graph;
int parent[MAXN][LN], subtree[MAXN], depth[MAXN], specialChild[MAXN], chainId[MAXN],
edgePosInChain[MAXN];

void init(int node, int l, int r, vector<int>& st, vector<int>& a);
void update(int node, int l, int r, int x, int y, long long val, vector<int>& st, vector<int
>& lazy);
int query(int node, int l, int r, int x, int y, vector<int>& st, vector<int>& lazy);

struct Chain{
    int connect, id;
    vector<int> chainEdge, st, lazy;

    Chain(int u, int _id){
        id = _id;
        connect = u;
    }

    void insert(int v, int w){
        edgePosInChain[v] = chainEdge.size();
        chainEdge.push_back(w);
```

```
        chainId[v] = id;
    }

    void initST(){
        st.assign(4*chainEdge.size(), 0);
        lazy.assign(4*chainEdge.size(), 0);

        init(1, 0, chainEdge.size()-1, st, chainEdge);
    }

    int queryChain(int v){
        return query(1, 0, chainEdge.size()-1, 0, edgePosInChain[v], st, lazy);
    }

    int queryChain(int u, int v){
        return query(1, 0, chainEdge.size()-1, edgePosInChain[u]+1, edgePosInChain[v], st,
        lazy);
    }

    void updateChain(int v, int w){
        int pos = edgePosInChain[v];
        chainEdge[pos] = w;
        update(1, 0, chainEdge.size()-1, pos, pos, w, st, lazy);
    }

};

void dfs(int u, int p, int d){
    parent[u][0] = p;
    subtree[u] = 1;
    depth[u] = d;

    specialChild[u] = -1;
    for(int i = 0; i < graph[u].size(); i++){
        int v = graph[u][i].first;

        if(v == p) continue;
        dfs(v, u, d+1);
        subtree[u] += subtree[v];

        if(specialChild[u] == -1 || subtree[v] > subtree[specialChild[u]])
            specialChild[u] = v;
    }
}

vector<Chain*> hldChain;
void HLD(int u, int u_w, Chain* chain){
    chain->insert(u, u_w);

    for(int i = 0; i < graph[u].size(); i++){
        int v = graph[u][i].first, w = graph[u][i].second;

        if(v == parent[u][0]) continue;
        if(v == specialChild[u]){
            // extend chain
            HLD(v, w, chain);
        }
        else{
            // new chain
```

```
            hldChain.push_back(new Chain(u, hldChain.size()));
            HLD(v, w, hldChain.back());
        }
    }
}

void HLD(int root){
    memset(parent, -1, sizeof parent);

    dfs(root, -1, 0);
    for(int i = 0; i < LN-1; i++){
        for(int j = 0; j < n; j++){
            if(parent[j][i] != -1)
                parent[j][i+1] = parent[parent[j][i]][i];
        }
    }
    hldChain.push_back(new Chain(-1, hldChain.size()));
    HLD(root, -1, hldChain.back());

    for(int i = 0; i < hldChain.size(); i++)
        hldChain[i]->initST();
}

// each edge has 1-1 correspondent with a vertex except root
// v-par[v] edge is uniquely determine by v
void update(int v, int val){
    hldChain[chainId[v]]->updateChain(v, val);
}

int lca(int u, int v){
    if(depth[u] < depth[v])
        return lca(v, u);
    // u is deeper
    int diff = depth[u] - depth[v];

    // advance u with diff
    for(int bit = 0; bit < LN; bit++)
        if(diff & (1<<bit) ) // if ith bit is 1, advance
            u = parent[u][bit];
    if(u != v)
    {
        for(int power = LN-1; power >= 0; power--) // start with highest power of 2
            if(parent[u][power] != parent[v][power])     // find higest not same parent
            {
                u = parent[u][power];
                v = parent[v][power];
            }
        u = parent[u][0];
    }
    return u;
}

int queryHLD(int u, int v){
    int ans = 0;
    while(chainId[u] != chainId[v]){
        ans = max(hldChain[chainId[u]]->queryChain(u), ans);
        u = hldChain[chainId[u]]->connect;
    }
```

```
    return max(ans, hldChain[chainId[u]]->queryChain(v, u));
}

int query(int u, int v){
    int l = lca(u, v);

    int q1 = queryHLD(u, l);
    int q2 = queryHLD(v, l);

    return max(q1, q2);
}

int main(){
    int t;
    scanf("%d ", &t);

    while(t--){
        scanf("%d", &n);

        vector<ii> edge;
        graph.assign(n, vector<ii>());
        for(int i = 0; i < n-1; i++){
            int u, v, c;
            scanf("%d %d %d", &u, &v, &c);
            u--; v--;
            graph[u].push_back(ii(v, c));
            graph[v].push_back(ii(u, c));
            edge.push_back(ii(u, v));
        }

        HLD(0);

        char type[10];
        scanf("%s", type);
        while(type[0] != 'D'){
            int a, b;
            scanf("%d %d", &a, &b);

            if(type[0]=='Q'){
                printf("%d\n", query(a-1, b-1));
            }
            else{
                int u = edge[a-1].first, v = edge[a-1].second;

                update(depth[u]>depth[v]?u:v, b);
            }

            scanf("%s", type);
        }
    }

    return 0;
}

// segment tree
void init(int node, int l, int r, vector<int>& st, vector<int>& a){
    if(l == r){
        st[node] = a[l];
        return;
```

```
    }
    init(l_node, l, mid, st, a);
    init(r_node, mid+1, r, st, a);
    st[node] = max(st[l_node], st[r_node]);
}

void update(int node, int l, int r, int x, int y, long long val, vector<int>& st, vector<int
>& lazy){
    if(lazy[node] != 0){
        // update this node
        st[node] += lazy[node]*(r-l+1);
        if(l != r){
            // propogate down
            lazy[l_node] += lazy[node];
            lazy[r_node] += lazy[node];
        }
        lazy[node] = 0;
    }
    if(x <= l && r <= y){   // in range, lazy update it
        // ensure this node value is correct for parent updating
        // st[node] += val*(r-l+1);

        // if(l != r){
        //     lazy[l_node] += val;
        //     lazy[r_node] += val;
        // }
        st[node] = val;
        return;
    }
    if(y < l || x > r){
        // out of range
        return;
    }
    update(l_node, l, mid, x, y, val, st, lazy);
    update(r_node, mid+1, r, x, y, val, st, lazy);
    st[node] = max(st[l_node], st[r_node]);
}

int query(int node, int l, int r, int x, int y, vector<int>& st, vector<int>& lazy){
    if(lazy[node] != 0){
        // update this node
        st[node] += lazy[node]*(r-l+1);
        if(l != r){
            // propogate down
            lazy[l_node] += lazy[node];
            lazy[r_node] += lazy[node];
        }
        lazy[node] = 0;
    }
    if(x <= l && r <= y){
        // in range, return value
        return st[node];
    }
    if(y < l || x > r){
        // out of range
        return 0;
    }
    long long q1 = query(l_node, l, mid, x, y, st, lazy);
    long long q2 = query(r_node, mid+1, r, x, y, st, lazy);
```

```
    return max(q1, q2);
}

--- LCA
#include <vector>
#include <cstring>
#include <cstdio>

using namespace std;

typedef pair<int, int> ii;
typedef long long ll;

vector<vector<ii> > graph;
ll dist[100010];
int parent[100010][20];
int depth[100010];

void dfs(int u){
    for(int i = 0; i < graph[u].size(); i++){
        int v = graph[u][i].first, w = graph[u][i].second;
        if(dist[v] == -1){
            dist[v] = dist[u] + w;
            depth[v] = depth[u] + 1;
            dfs(v);
        }
    }
}

int lca(int u, int v){
    if(depth[u] < depth[v])
        return lca(v, u);

    // u is deeper

    int diff = depth[u] - depth[v];

    // advance u with diff
    for(int bit = 0; bit < 20; bit++)
        if(diff & (1<<bit) ) // if ith bit is 1, advance
            u = parent[u][bit];

    if(u != v)
    {
        for(int power = 19; power >= 0; power--) // start with highest power of 2
            if(parent[u][power] != parent[v][power])    // find higest not same parent
            {
                u = parent[u][power];
                v = parent[v][power];
            }
        u = parent[u][0];
    }
    return u;

}

int main() {
    int N;
    while(scanf("%d", &N) && N != 0){
```

```
        graph.assign(N, vector<ii>());
        memset(dist, -1, sizeof dist);
        memset(parent, -1, sizeof parent);

        int v, w;
        for(int i = 1; i < N; i++){
            //cin >> v >> w;
            scanf("%d %d", &v, &w);
            parent[i][0] = v;
            graph[i].push_back(ii(v, w));
            graph[v].push_back(ii(i, w));
        }
        for(int i = 0; i < 19; i++){
            for(int j = 0; j < N; j++){
                if(parent[j][i] != -1)
                    parent[j][i+1] = parent[parent[j][i]][i];
        }   }

        dist[0] = 0; depth[0] = 0; dfs(0);

        int Q;
        cin >> Q;
        while(Q--){
            int u, v;
            cin >> u >> v;
            cout << dist[u]+dist[v]-2*dist[lca(u, v)];
            if(!Q) cout << endl;
            else   cout << ' ';
    }    }
    return 0;
}

--- Kruskal + UFDS
#include <cstdio>
#include <vector>
#include <queue>

using namespace std;

typedef pair<int, int> ii;
typedef pair<int, ii> iii;

vector<int> p;

void init(int n){
    p.resize(n);
    for(int i = 0; i < n; i++){
        p[i] = i;
    }
}
int find_set(int x){
    if(p[x] == x)
        return x;
    return p[x] = find_set(p[x]);
}
bool is_same_set(int x, int y){
    return find_set(x) == find_set(y);
}
void union_set(int x, int y){
```

```
    p[find_set(x)] = find_set(y);
}
int main(){
    int n, m, a, b, w;
    cin >> n >> m;

    init(n);
    priority_queue<iii, vector<iii>, greater<iii> > pq;
    while(m--){
        cin >> a >> b >> w;
        pq.push(iii(w, ii(a-1, b-1)));
    }
    int mst = 0;
    while(!pq.empty()){
        int w = pq.top().first;
        ii v = pq.top().second;
        pq.pop();

        if(!is_same_set(v.first, v.second)){
            mst += w;
            union_set(v.first, v.second);
        }
    }
    cout << mst << endl;

    return 0;
}

--- Dinic
#include <vector>
#include <map>
#include <cstdio>

#define INF     (1<<30)
#define INFLL   (1LL<<60)

using namespace std;

typedef pair<int, int> ii;

struct MaxFlow{
    int n, s, t;
    vector<vector<ii> > graph;
    vector<long long> cap;
    vector<int> dist, q, now;

    MaxFlow(int _n){
        // 0-based indexing, init(n+1) for 1 based indexing
        n = _n;
        graph.assign(n, vector<ii> ());
        q.resize(n+10);
    }

    void addEdge(int u, int v, long long c, bool directed){
        graph[u].push_back(ii(v, cap.size()));
        cap.push_back(c);
        graph[v].push_back(ii(u, cap.size()));
        cap.push_back(directed?0:c);
    }
```

```
    long long getMaxFlow(int _s, int _t){
        s = _s; t = _t;
        long long flow = 0;
        while(bfsLevelGraph()){
            now.assign(n, 0);
            while(long long f = dfsSendFlow(s, INFLL))
                flow += f;
        }

        return flow;
    }

    bool bfsLevelGraph(){
        dist.assign(n, INF);
        int qs = 0, qe = 0;
        q[qe++] = s;
        dist[s] = 0;

        while(qs < qe){
            int u = q[qs++];
            for(int i = 0; i < graph[u].size(); i++){
                int v = graph[u][i].first, e = graph[u][i].second;

                if(dist[v] == INF && cap[e] > 0){
                    dist[v] = dist[u]+1;
                    q[qe++] = v;
        }   }   }
        return dist[t] != INF;
    }
    long long dfsSendFlow(int u, long long curFlow){
        if(u == t)          return curFlow;
        if(curFlow == 0)    return curFlow;

        for(; now[u] < graph[u].size(); now[u]++){
            int v = graph[u][now[u]].first, e = graph[u][now[u]].second;

            if(dist[v] == dist[u] +1 && cap[e] > 0){
                // an edge exist in level graph
                long long flowSent = dfsSendFlow(v, min(curFlow, cap[e]));

                if(flowSent > 0){
                    cap[e] -= flowSent;
                    cap[e^1] += flowSent;

                    return flowSent;
        }   }   }
        return 0;
    }
};

int main(){
    int n, m;
    scanf("%d %d", &n, &m);

    MaxFlow flow_problem(n+1);

    while(m--){
        int u, v, c;
```

```
        scanf("%d %d %d", &u, &v, &c);

        if(u == v)
            continue;

        flow_problem.addEdge(u, v, c, false);
    }

    printf("%lld\n", flow_problem.getMaxFlow(1, n));

    return 0;
}
```

--- MinCost Flow Dijkstra

```
// not yet tested on starting negative edge

#include <vector>
#include <queue>

using namespace std;

typedef pair<int, int> ii;

struct Edge{
    int u, v;
    long long cap, cost;

    Edge(int _u, int _v, long long _cap, long long _cost){
        u = _u; v = _v; cap = _cap; cost = _cost;
    }
};

struct MinCostFlow{
    int n, s, t;
    long long flow, cost;
    vector<vector<int> > graph;
    vector<Edge> e;
    vector<long long> dist, potential;
    vector<int> parent;
    bool negativeCost;

    MinCostFlow(int _n){
        // 0-based indexing
        n = _n;
        graph.assign(n, vector<int> ());
        negativeCost = false;
    }

    void addEdge(int u, int v, long long cap, long long cost, bool directed){
        if(cost < 0)
            negativeCost = true;

        graph[u].push_back(e.size());
        e.push_back(Edge(u, v, cap, cost));

        graph[v].push_back(e.size());
        e.push_back(Edge(v, u, 0, -cost));

        if(!directed)
```

```
        addEdge(v, u, cap, cost, true);
}

pair<long long, long long> getMinCostFlow(int _s, int _t){
    s = _s; t = _t;
    flow = 0, cost = 0;

    potential.assign(n, 0);
    if(negativeCost){
        // run Bellman-Ford to find starting potential
        dist.assign(n, 1LL<<62);
        for(int i = 0, relax = false; i < n && relax; i++, relax = false){
            for(int u = 0; u < n; u++){
                for(int k = 0; k < graph[u].size(); k++){
                    int eIdx = graph[u][i];
                    int v = e[eIdx].v, cap = e[eIdx].cap, w = e[eIdx].cost;

                    if(dist[v] > dist[u] + w && cap > 0){
                        dist[v] = dist[u] + w;
                        relax = true;
        }   }   }   }

        for(int i = 0; i < n; i++){
            if(dist[i] < (1LL<<62)){
                potential[i] = dist[i];
    }   }   }

    while(dijkstra()){
        flow += sendFlow(t, 1LL<<62);
    }

    return make_pair(flow, cost);
}

bool dijkstra(){
    parent.assign(n, -1);
    dist.assign(n, 1LL<<62);
    priority_queue<ii, vector<ii>, greater<ii> > pq;

    dist[s] = 0;
    pq.push(ii(0, s));


    while(!pq.empty()){
        int u = pq.top().second;
        long long d = pq.top().first;
        pq.pop();

        if(d != dist[u]) continue;

        for(int i = 0; i < graph[u].size(); i++){
            int eIdx = graph[u][i];
            int v = e[eIdx].v, cap = e[eIdx].cap;
            int w = e[eIdx].cost + potential[u] - potential[v];

            if(dist[u] + w < dist[v] && cap > 0){
                dist[v] = dist[u] + w;
                parent[v] = eIdx;
```

```
                    pq.push(ii(dist[v], v));
        }   }   }

        // update potential
        for(int i = 0; i < n; i++){
            if(dist[i] < (1LL<<62))
                potential[i] += dist[i];
        }

        return dist[t] != (1LL<<62);
    }

    long long sendFlow(int v, long long curFlow){
        if(parent[v] == -1)
            return curFlow;
        int eIdx = parent[v];
        int u = e[eIdx].u, w = e[eIdx].cost;

        long long f = sendFlow(u, min(curFlow, e[eIdx].cap));

        cost += f*w;
        e[eIdx].cap -= f;
        e[eIdx^1].cap += f;

        return f;
    }
};

int main(){
    int n, m, s, t;
    cin >> n >> m >> s >> t;

    MinCostFlow minCostFlowProblem(n);

    while(m--){
        int u, v, c, w;
        cin >> u >> v >> c >> w;

        minCostFlowProblem.addEdge(u, v, c, w, true);
    }

    pair<int, int> ans = minCostFlowProblem.getMinCostFlow(s, t);

    cout << ans.first << ' ' << ans.second << endl;

    return 0;
}

--- MinCost Flow SPFA
#include <vector>
#include <queue>

using namespace std;

struct Edge{
    int u, v;
    long long cap, cost;

    Edge(int _u, int _v, long long _cap, long long _cost){
```

```
        u = _u; v = _v; cap = _cap; cost = _cost;
    }
};

struct MinCostFlow{
    int n, s, t;
    long long flow, cost;
    vector<vector<int> > graph;
    vector<Edge> e;
    vector<long long> dist;
    vector<int> parent;

    MinCostFlow(int _n){
        // 0-based indexing
        n = _n;
        graph.assign(n, vector<int> ());
    }

    void addEdge(int u, int v, long long cap, long long cost, bool directed){
        graph[u].push_back(e.size());
        e.push_back(Edge(u, v, cap, cost));

        graph[v].push_back(e.size());
        e.push_back(Edge(v, u, 0, -cost));

        if(!directed)
            addEdge(v, u, cap, cost, true);
    }

    pair<long long, long long> getMinCostFlow(int _s, int _t){
        s = _s; t = _t;
        flow = 0, cost = 0;

        while(SPFA()){
            flow += sendFlow(t, 1LL<<62);
        }

        return make_pair(flow, cost);
    }

    // not sure about negative cycle
    bool SPFA(){
        parent.assign(n, -1);
        dist.assign(n, 1LL<<62);        dist[s] = 0;
        vector<int> queuetime(n, 0);    queuetime[s] = 1;
        vector<bool> inqueue(n, 0);     inqueue[s] = true;
        queue<int> q;                   q.push(s);
        bool negativecycle = false;


        while(!q.empty() && !negativecycle){
            int u = q.front(); q.pop(); inqueue[u] = false;

            for(int i = 0; i < graph[u].size(); i++){
                int eIdx = graph[u][i];
                int v = e[eIdx].v, w = e[eIdx].cost, cap = e[eIdx].cap;

                if(dist[u] + w < dist[v] && cap > 0){
                    dist[v] = dist[u] + w;
```

```
                    parent[v] = eIdx;

                    if(!inqueue[v]){
                        q.push(v);
                        queuetime[v]++;
                        inqueue[v] = true;

                        if(queuetime[v] == n+2){
                            negativecycle = true;
                            break;
            }   }   }   }   }

        return dist[t] != (1LL<<62);
    }

    long long sendFlow(int v, long long curFlow){
        if(parent[v] == -1)
            return curFlow;
        int eIdx = parent[v];
        int u = e[eIdx].u, w = e[eIdx].cost;

        long long f = sendFlow(u, min(curFlow, e[eIdx].cap));

        cost += f*w;
        e[eIdx].cap -= f;
        e[eIdx^1].cap += f;

        return f;
    }
};

int main(){
    int n, m, s, t;
    cin >> n >> m >> s >> t;

    MinCostFlow minCostFlowProblem(n);

    while(m--){
        int u, v, c, w;
        cin >> u >> v >> c >> w;

        minCostFlowProblem.addEdge(u, v, c, w, true);
    }

    pair<int, int> ans = minCostFlowProblem.getMinCostFlow(s, t);

    cout << ans.first << ' ' << ans.second << endl;

    return 0;
}

--- Bipartite Matching
#include <vector>
#include <queue>

#define INF (1<<30)

using namespace std;
```

```cpp
struct Matching{
    int n, m;
    vector<vector<int> > graph;
    vector<int> match, dist;


    Matching(int _n, int _m){
        // 1-based indexing
        n = _n; m = _m;
        graph.assign(n+m+1, vector<int> ());
        match.assign(n+m+1, 0);
        dist.resize(n+1);
    }

    void addPair(int u, int v){
        graph[u].push_back(v+n);
        graph[v+n].push_back(u);
    }

    int HopcroftKarpMatching(){
        int matching = 0;
        while(bfs()){
            for(int i = 1; i <= n; i++){
                if(match[i] == 0 && dfs(i))
                    matching++;
            }
        }

        return matching;
    }

    bool bfs(){
        // 0 is the sink

        queue<int> q;
        dist[0] = INF;
        for(int i = 1; i <= n; i++){
            if(match[i] == 0){
                dist[i] = 0;
                q.push(i);
            }
            else{
                dist[i] = INF;
            }
        }

        while(!q.empty()){
            int u = q.front(); q.pop();

            if(u != 0){
                for(int i = 0; i < graph[u].size(); i++){
                    int v = graph[u][i];

                    // v is in V, match[v] is in U
                    // match[v] is 0 (not matched by default)
                    if(dist[match[v]] == INF){
                        dist[match[v]] = dist[u] + 1;
                        q.push(match[v]);
        }   }   }   }
```

```
        // check is sink is still recheable
        return dist[0] != INF;
    }

    int dfs(int u){
        // reach sink
        if(u == 0)
            return true;

        for(int i = 0; i < graph[u].size(); i++){
            int v = graph[u][i];

            if(dist[match[v]] == dist[u] + 1 && dfs(match[v])){
                match[u] = v;
                match[v] = u;
                return true;
            }
        }
        // no more match available
        return false;
    }
};

int main(){
    int n, m, e;
    scanf("%d %d %d", &n, &m, &e);

    Matching matching_problem(n, m);

    while(e--){
        int u, v;
        scanf("%d %d", &u, &v);

        matching_problem.addPair(u, v);
    }

    printf("%d\n", matching_problem.HopcroftKarpMatching());

    return 0;
}

--- Geometry
// geometry library, mainly (99%) adapted from competitive programming by steven halim

#include <algorithm>
#include <cstdio>
#include <cmath>
#include <vector>

using namespace std;

#define INF 1e9
#define EPS 1e-9
#define PI acos(-1.0)

double DEG_to_RAD(double d) { return d * PI / 180.0; }

double RAD_to_DEG(double r) { return r * 180.0 / PI; }
```

```
struct point { double x, y;
  point() { x = y = 0.0; }
  point(double _x, double _y) : x(_x), y(_y) {}
  bool operator < (point other) const { // override less than operator
    if (fabs(x - other.x) > EPS)                 // useful for sorting
      return x < other.x;
    return y < other.y; }
  // use EPS (1e-9) when testing equality of two floating points
  bool operator == (point other) const {
   return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };

double dist(point p1, point p2) {                // Euclidean distance
  return hypot(p1.x - p2.x, p1.y - p2.y); }

struct line { double a, b, c; };

struct vec { double x, y;
  vec(double _x, double _y) : x(_x), y(_y) {} };

// the answer is stored in the third parameter (pass by reference)
void pointsToLine(point p1, point p2, line &l) {
  if (fabs(p1.x - p2.x) < EPS) {              // vertical line is fine
    l.a = 1.0;   l.b = 0.0;   l.c = -p1.x;           // default values
  } else {
    l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
    l.b = 1.0;              // IMPORTANT: we fix the value of b to 1.0
    l.c = -(double)(l.a * p1.x) - p1.y;
} }

bool areParallel(line l1, line l2) {        // check coefficients a & b
  return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }

bool areSame(line l1, line l2) {            // also check coefficient c
  return areParallel(l1 ,l2) && (fabs(l1.c - l2.c) < EPS); }

// returns true (+ intersection point) if two lines are intersect
bool areIntersect(line l1, line l2, point &p) {
  if (areParallel(l1, l2)) return false;            // no intersection
  // solve system of 2 linear algebraic equations with 2 unknowns
  p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
  // special case: test for vertical line to avoid division by zero
  if (fabs(l1.b) > EPS) p.y = -(l1.a * p.x + l1.c);
  else                  p.y = -(l2.a * p.x + l2.c);
  return true; }

vec toVec(point a, point b) {       // convert 2 points to vector a->b
  return vec(b.x - a.x, b.y - a.y); }

vec scale(vec v, double s) {        // nonnegative s = [<1 .. 1 .. >1]
  return vec(v.x * s, v.y * s); }               // shorter.same.longer

point translate(point p, vec v) {        // translate p according to v
  return point(p.x + v.x , p.y + v.y); }

// rotate p by theta degrees CCW w.r.t origin (0, 0)
point rotate(point p, double theta) {
  double rad = DEG_to_RAD(theta);    // multiply theta with PI / 180.0
  return point(p.x * cos(rad) - p.y * sin(rad),
               p.x * sin(rad) + p.y * cos(rad)); }
```

```
// convert point and gradient/slope to line
void pointSlopeToLine(point p, double m, line &l) {
  l.a = -m;                                               // always -m
  l.b = 1;                                                 // always 1
  l.c = -((l.a * p.x) + (l.b * p.y)); }                // compute this

void closestPoint(line l, point p, point &ans) {
  line perpendicular;         // perpendicular to l and pass through p
  if (fabs(l.b) < EPS) {                // special case 1: vertical line
    ans.x = -(l.c);    ans.y = p.y;       return; }

  if (fabs(l.a) < EPS) {              // special case 2: horizontal line
    ans.x = p.x;      ans.y = -(l.c);   return; }

  pointSlopeToLine(p, 1 / l.a, perpendicular);          // normal line
  // intersect line l with this perpendicular line
  // the intersection point is the closest point
  areIntersect(l, perpendicular, ans); }

// returns the reflection of point on a line
void reflectionPoint(line l, point p, point &ans) {
  point b;
  closestPoint(l, p, b);                         // similar to distToLine
  vec v = toVec(p, b);                                 // create a vector
  ans = translate(translate(p, v), v); }           // translate p twice

double dot(vec a, vec b) { return (a.x * b.x + a.y * b.y); }

double norm_sq(vec v) { return v.x * v.x + v.y * v.y; }

// returns the distance from p to the line defined by
// two points a and b (a and b must be different)
// the closest point is stored in the 4th parameter (byref)
double distToLine(point p, point a, point b, point &c) {
  // formula: c = a + u * ab
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  c = translate(a, scale(ab, u));                    // translate a to c
  return dist(p, c); }           // Euclidean distance between p and c

// returns the distance from p to the line segment ab defined by
// two points a and b (still OK if a == b)
// the closest point is stored in the 4th parameter (byref)
double distToLineSegment(point p, point a, point b, point &c) {
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  if (u < 0.0) { c = point(a.x, a.y);                     // closer to a
    return dist(p, a); }         // Euclidean distance between p and a
  if (u > 1.0) { c = point(b.x, b.y);                     // closer to b
    return dist(p, b); }         // Euclidean distance between p and b
  return distToLine(p, a, b, c); }          // run distToLine as above

double angle(point a, point o, point b) {  // returns angle aob in rad
  vec oa = toVec(o, a), ob = toVec(o, b);
  return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob))); }

double cross(vec a, vec b) { return a.x * b.y - a.y * b.x; }
```

```
// note: to accept collinear points, we have to change the `> 0'
// returns true if point r is on the left side of line pq
bool ccw(point p, point q, point r) {
  return cross(toVec(p, q), toVec(p, r)) > 0; }

// returns true if point r is on the same line as the line pq
bool collinear(point p, point q, point r) {
  return fabs(cross(toVec(p, q), toVec(p, r))) < EPS; }

int insideCircle(point p, point c, int r) {                // int/double
  int dx = p.x - c.x, dy = p.y - c.y;
  int Euc = dx * dx + dy * dy, rSq = r * r;             // integer/double
  return Euc < rSq ? 0 : Euc == rSq ? 1 : 2; } //inside/border/outside

// 2 point + radius => find center of circle
bool circle2PtsRad(point p1, point p2, double r, point &c) {
  double d2 = (p1.x - p2.x) * (p1.x - p2.x) +
              (p1.y - p2.y) * (p1.y - p2.y);
  double det = r * r / d2 - 0.25;
  if (det < 0.0) return false;
  double h = sqrt(det);
  c.x = (p1.x + p2.x) * 0.5 + (p1.y - p2.y) * h;
  c.y = (p1.y + p2.y) * 0.5 + (p2.x - p1.x) * h;
  return true; }          // to get the other center, reverse p1 and p2

// theta in radian
double chordLength(double r, double theta){
    sqrt((2 * r * r) * (1 - cos(theta)));
}

double perimeter(double ab, double bc, double ca) {
  return ab + bc + ca; }

double perimeter(point a, point b, point c) {
  return dist(a, b) + dist(b, c) + dist(c, a); }

double area(double ab, double bc, double ca) {
  // Heron's formula, split sqrt(a * b) into sqrt(a) * sqrt(b); in implementation
  double s = 0.5 * perimeter(ab, bc, ca);
  return sqrt(s) * sqrt(s - ab) * sqrt(s - bc) * sqrt(s - ca); }

double area(point a, point b, point c) {
  return area(dist(a, b), dist(b, c), dist(c, a)); }

// alternative: more accurate for integer
// returns 'twice' the area of this triangle p-q-r
int area2(point p, point q, point r) {
  return p.x * q.y - p.y * q.x +
         q.x * r.y - q.y * r.x +
         r.x * p.y - r.y * p.x;
}

double rInCircle(double ab, double bc, double ca) {
  return area(ab, bc, ca) / (0.5 * perimeter(ab, bc, ca)); }

double rInCircle(point a, point b, point c) {
  return rInCircle(dist(a, b), dist(b, c), dist(c, a)); }

// assumption: the required points/lines functions have been written
```

```
// returns 1 if there is an inCircle center, returns 0 otherwise
// if this function returns 1, ctr will be the inCircle center
// and r is the same as rInCircle, passed by reference
int inCircle(point p1, point p2, point p3, point &ctr, double &r) {
  r = rInCircle(p1, p2, p3);
  if (fabs(r) < EPS) return 0;                   // no inCircle center

  line l1, l2;                    // compute these two angle bisectors
  double ratio = dist(p1, p2) / dist(p1, p3);
  point p = translate(p2, scale(toVec(p2, p3), ratio / (1 + ratio)));
  pointsToLine(p1, p, l1);

  ratio = dist(p2, p1) / dist(p2, p3);
  p = translate(p1, scale(toVec(p1, p3), ratio / (1 + ratio)));
  pointsToLine(p2, p, l2);

  areIntersect(l1, l2, ctr);           // get their intersection point
  return 1; }

double rCircumCircle(double ab, double bc, double ca) {
  return ab * bc * ca / (4.0 * area(ab, bc, ca)); }

double rCircumCircle(point a, point b, point c) {
  return rCircumCircle(dist(a, b), dist(b, c), dist(c, a)); }

// assumption: the required points/lines functions have been written
// returns 1 if there is a circumCenter center, returns 0 otherwise
// if this function returns 1, ctr will be the circumCircle center
// and r is the same as rCircumCircle
int circumCircle(point p1, point p2, point p3, point &ctr, double &r){
  double a = p2.x - p1.x, b = p2.y - p1.y;
  double c = p3.x - p1.x, d = p3.y - p1.y;
  double e = a * (p1.x + p2.x) + b * (p1.y + p2.y);
  double f = c * (p1.x + p3.x) + d * (p1.y + p3.y);
  double g = 2.0 * (a * (p3.y - p2.y) - b * (p3.x - p2.x));
  if (fabs(g) < EPS) return 0;

  ctr.x = (d*e - b*f) / g;
  ctr.y = (a*f - c*e) / g;
  r = dist(p1, ctr);  // r = distance from center to 1 of the 3 points
  return 1; }

// returns true if point d is inside the circumCircle defined by a,b,c
int inCircumCircle(point a, point b, point c, point d) {
  return (a.x - d.x) * (b.y - d.y) * ((c.x - d.x) * (c.x - d.x) + (c.y - d.y) * (c.y - d.y)) +
         (a.y - d.y) * ((b.x - d.x) * (b.x - d.x) + (b.y - d.y) * (b.y - d.y)) * (c.x - d.x) +
         ((a.x - d.x) * (a.x - d.x) + (a.y - d.y) * (a.y - d.y)) * (b.x - d.x) * (c.y - d.y) -
         ((a.x - d.x) * (a.x - d.x) + (a.y - d.y) * (a.y - d.y)) * (b.y - d.y) * (c.x - d.x) -
         (a.y - d.y) * (b.x - d.x) * ((c.x - d.x) * (c.x - d.x) + (c.y - d.y) * (c.y - d.y)) -
         (a.x - d.x) * ((b.x - d.x) * (b.x - d.x) + (b.y - d.y) * (b.y - d.y)) * (c.y - d.y)
         > 0 ? 1 : 0;
}

bool canFormTriangle(double a, double b, double c) {
  return (a + b > c) && (a + c > b) && (b + c > a); }


// returns the perimeter, which is the sum of Euclidian distances
// of consecutive line segments (polygon edges)
```

```
double perimeter(const vector<point> &P) {
  double result = 0.0;
  for (int i = 0; i < (int)P.size()-1; i++)  // remember that P[0] = P[n-1]
    result += dist(P[i], P[i+1]);
  return result; }

// returns the area, which is half the determinant
double area(const vector<point> &P) {
  double result = 0.0, x1, y1, x2, y2;
  for (int i = 0; i < (int)P.size()-1; i++) {
    x1 = P[i].x; x2 = P[i+1].x;
    y1 = P[i].y; y2 = P[i+1].y;
    result += (x1 * y2 - x2 * y1);
  }
  return fabs(result) / 2.0; }


// returns true if we always make the same turn while examining
// all the edges of the polygon one by one
bool isConvex(const vector<point> &P) {
  int sz = (int)P.size();
  if (sz <= 3) return false;   // a point/sz=2 or a line/sz=3 is not convex
  bool isLeft = ccw(P[0], P[1], P[2]);              // remember one result
  for (int i = 1; i < sz-1; i++)              // then compare with the others
    if (ccw(P[i], P[i+1], P[(i+2) == sz ? 1 : i+2]) != isLeft)
      return false;            // different sign -> this polygon is concave
  return true; }                                   // this polygon is convex

// returns true if point p is in either convex/concave polygon P
bool inPolygon(point pt, const vector<point> &P) {
  if ((int)P.size() == 0) return false;
  double sum = 0;    // assume the first vertex is equal to the last vertex
  for (int i = 0; i < (int)P.size()-1; i++) {
    if (ccw(pt, P[i], P[i+1]))
        sum += angle(P[i], pt, P[i+1]);                   // left turn/ccw
    else sum -= angle(P[i], pt, P[i+1]); }                // right turn/cw
  return fabs(fabs(sum) - 2*PI) < EPS; }

// line segment p-q intersect with line A-B.
point lineIntersectSeg(point p, point q, point A, point B) {
  double a = B.y - A.y;
  double b = A.x - B.x;
  double c = B.x * A.y - A.x * B.y;
  double u = fabs(a * p.x + b * p.y + c);
  double v = fabs(a * q.x + b * q.y + c);
  return point((p.x * v + q.x * u) / (u+v), (p.y * v + q.y * u) / (u+v)); }

// cuts polygon Q along the line formed by point a -> point b
// (note: the last point must be the same as the first point)
vector<point> cutPolygon(point a, point b, const vector<point> &Q) {
  vector<point> P;
  for (int i = 0; i < (int)Q.size(); i++) {
    double left1 = cross(toVec(a, b), toVec(a, Q[i])), left2 = 0;
    if (i != (int)Q.size()-1) left2 = cross(toVec(a, b), toVec(a, Q[i+1]));
    if (left1 > -EPS) P.push_back(Q[i]);       // Q[i] is on the left of ab
    if (left1 * left2 < -EPS)        // edge (Q[i], Q[i+1]) crosses line ab
      P.push_back(lineIntersectSeg(Q[i], Q[i+1], a, b));
  }
  if (!P.empty() && !(P.back() == P.front()))
```

```
    P.push_back(P.front());         // make P's first point = P's last point
  return P; }

point pivot;
bool angleCmp(point a, point b) {                      // angle-sorting function
  if (collinear(pivot, a, b))                                   // special case
    return dist(pivot, a) < dist(pivot, b);     // check which one is closer
  double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
  double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
  return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0; }   // compare two angles

vector<point> CH(vector<point> P) {   // the content of P may be reshuffled
  int i, j, n = (int)P.size();
  if (n <= 3) {
    if (!(P[0] == P[n-1])) P.push_back(P[0]); // safeguard from corner case
    return P;                          // special case, the CH is P itself
  }

  // first, find P0 = point with lowest Y and if tie: rightmost X
  int P0 = 0;
  for (i = 1; i < n; i++)
    if (P[i].y < P[P0].y || (P[i].y == P[P0].y && P[i].x > P[P0].x))
      P0 = i;

  point temp = P[0]; P[0] = P[P0]; P[P0] = temp;    // swap P[P0] with P[0]

  // second, sort points by angle w.r.t. pivot P0
  pivot = P[0];                     // use this global variable as reference
  sort(++P.begin(), P.end(), angleCmp);               // we do not sort P[0]

  // third, the ccw tests
  vector<point> S;
  S.push_back(P[n-1]); S.push_back(P[0]); S.push_back(P[1]);   // initial S
  i = 2;                                             // then, we check the rest
  while (i < n) {              // note: N must be >= 3 for this method to work
    j = (int)S.size()-1;
    if (ccw(S[j-1], S[j], P[i])) S.push_back(P[i++]);  // left turn, accept
    else S.pop_back(); }    // or pop the top of S until we have a left turn
  return S; }                                            // return the result

int main() {
  // note:
  // 1) usage of pseudoangle, pseudoangle = copysign(1. - dx/(fabs(dx)+fabs(dy)),dy)
  //      OR double pseudoangle(double dy, double dx){return copysign(1. -
  dx/(fabs(dx)+fabs(dy)),dy);}
  // 2) to accept collinear points, we have to change the `> 0' in ccw function
  // 3) for polygon, the last point must be same as first point, and it is defined in
  conterclockwise
  // 4) reverse point a, b to get second half of cut polygon
  return 0;
}

// Pick theorem
// Given non-intersecting polygon.
// S = area
// I = number of integer points strictly Inside
// B = number of points on sides of polygon
// S = I + B/2 - 1
```

```
// Find intersection of 2 polygons
// Helper method
#ifdef include
bool intersect_1pt(point a, point b,
    point c, point d, point &r) {
    double D =  (b - a) % (d - c);
    if (cmp(D, 0) == 0) return false;
    double t =  ((c - a) % (d - c)) / D;
    double s = -((a - c) % (b - a)) / D;
    r = a + (b - a) * t;
    return cmp(t, 0) > 0 && cmp(t, 1) < 0 && cmp(s, 0) > 0 && cmp(s, 1) < 0;
}
Polygon convex_intersect(Polygon P, Polygon Q) {
    const int n = P.size(), m = Q.size();
    int a = 0, b = 0, aa = 0, ba = 0;
    enum { Pin, Qin, Unknown } in = Unknown;
    Polygon R;
    do {
        int a1 = (a+n-1) % n, b1 = (b+m-1) % m;
        double C = (P[a] - P[a1]) % (Q[b] - Q[b1]);
        double A = (P[a1] - Q[b]) % (P[a] - Q[b]);
        double B = (Q[b1] - P[a]) % (Q[b] - P[a]);
        point r;
        if (intersect_1pt(P[a1], P[a], Q[b1], Q[b], r)) {
            if (in == Unknown) aa = ba = 0;
            R.push_back( r );
            in = B > 0 ? Pin : A > 0 ? Qin : in;
        }
        if (C == 0 && B == 0 && A == 0) {
            if (in == Pin) { b = (b + 1) % m; ++ba; }
            else           { a = (a + 1) % m; ++aa; }
        } else if (C >= 0) {
            if (A > 0) { if (in == Pin) R.push_back(P[a]); a = (a+1)%n; ++aa; }
            else       { if (in == Qin) R.push_back(Q[b]); b = (b+1)%m; ++ba; }
        } else {
            if (B > 0) { if (in == Qin) R.push_back(Q[b]); b = (b+1)%m; ++ba; }
            else       { if (in == Pin) R.push_back(P[a]); a = (a+1)%n; ++aa; }
        }
    } while ( (aa < n || ba < m) && aa < 2*n && ba < 2*m );
    if (in == Unknown) {
        if (in_convex(Q, P[0])) return P;
        if (in_convex(P, Q[0])) return Q;
    }
    return R;
}

// Find the diameter of polygon.
// Rotating callipers
double convex_diameter(Polygon pt) {
    const int n = pt.size();
    int is = 0, js = 0;
    for (int i = 1; i < n; ++i) {
        if (pt[i].y > pt[is].y) is = i;
        if (pt[i].y < pt[js].y) js = i;
    }
    double maxd = (pt[is]-pt[js]).norm();
    int i, maxi, j, maxj;
    i = maxi = is;
```

```
    j = maxj = js;
    do {
        int jj = j+1; if (jj == n) jj = 0;
        if ((pt[i] - pt[jj]).norm() > (pt[i] - pt[j]).norm()) j = (j+1) % n;
        else i = (i+1) % n;
        if ((pt[i]-pt[j]).norm() > maxd) {
            maxd = (pt[i]-pt[j]).norm();
            maxi = i; maxj = j;
        }
    } while (i != is || j != js);
    return maxd; /* farthest pair is (maxi, maxj). */
}

#endif

--- String
// string algorithm: suffix array (steven halim implementation)
//                     manacher algorithm (source unknown)

#include <algorithm>
#include <cstdio>
#include <cstring>
#include <vector>

using namespace std;

typedef pair<int, int> ii;

#define MAX_N 100010                                 // second approach: O(n log n)
char T[MAX_N];                     // the input string, up to 100K characters
int n;                                                // the length of input string
int RA[MAX_N], tempRA[MAX_N];               // rank array and temporary rank array
int SA[MAX_N], tempSA[MAX_N];           // suffix array and temporary suffix array
int c[MAX_N];                                            // for counting/radix sort

char P[MAX_N];                     // the pattern string (for string matching)
int m;                                              // the length of pattern string

int Phi[MAX_N];                               // for computing longest common prefix
int PLCP[MAX_N];
int LCP[MAX_N];  // LCP[i] stores the LCP between previous suffix T+SA[i-1]
                                                 // and current suffix T+SA[i]

bool cmp(int a, int b) { return strcmp(T + a, T + b) < 0; }       // compare

void constructSA_slow() {                          // cannot go beyond 1000 characters
  for (int i = 0; i < n; i++) SA[i] = i; // initial SA: {0, 1, 2, ..., n-1}
  sort(SA, SA + n, cmp); // sort: O(n log n) * compare: O(n) = O(n^2 log n)
}

void countingSort(int k) {                                                  // O(n)
  int i, sum, maxi = max(300, n);    // up to 255 ASCII chars or length of n
  memset(c, 0, sizeof c);                                 // clear frequency table
  for (i = 0; i < n; i++)        // count the frequency of each integer rank
    c[i + k < n ? RA[i + k] : 0]++;
  for (i = sum = 0; i < maxi; i++) {
    int t = c[i]; c[i] = sum; sum += t;
  }
  for (i = 0; i < n; i++)           // shuffle the suffix array if necessary
```

```
    tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] = SA[i];
  for (i = 0; i < n; i++)                          // update the suffix array SA
    SA[i] = tempSA[i];
}

void constructSA() {           // this version can go up to 100000 characters
  int i, k, r;
  for (i = 0; i < n; i++) RA[i] = T[i];                   // initial rankings
  for (i = 0; i < n; i++) SA[i] = i;      // initial SA: {0, 1, 2, ..., n-1}
  for (k = 1; k < n; k <<= 1) {        // repeat sorting process log n times
    countingSort(k);  // actually radix sort: sort based on the second item
    countingSort(0);           // then (stable) sort based on the first item
    tempRA[SA[0]] = r = 0;             // re-ranking; start from rank r = 0
    for (i = 1; i < n; i++)                    // compare adjacent suffixes
      tempRA[SA[i]] = // if same pair => same rank r; otherwise, increase r
      (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k]) ? r : ++r;
    for (i = 0; i < n; i++)                     // update the rank array RA
      RA[i] = tempRA[i];
    if (RA[SA[n-1]] == n-1) break;               // nice optimization trick
} }

void computeLCP_slow() {
  LCP[0] = 0;                                              // default value
  for (int i = 1; i < n; i++) {                // compute LCP by definition
    int L = 0;                                       // always reset L to 0
    while (T[SA[i] + L] == T[SA[i-1] + L]) L++;      // same L-th char, L++
    LCP[i] = L;
} }

void computeLCP() {
  int i, L;
  Phi[SA[0]] = -1;                                         // default value
  for (i = 1; i < n; i++)                            // compute Phi in O(n)
    Phi[SA[i]] = SA[i-1];    // remember which suffix is behind this suffix
  for (i = L = 0; i < n; i++) {             // compute Permuted LCP in O(n)
    if (Phi[i] == -1) { PLCP[i] = 0; continue; }            // special case
    while (T[i + L] == T[Phi[i] + L]) L++;       // L increased max n times
    PLCP[i] = L;
    L = max(L-1, 0);                             // L decreased max n times
  }
  for (i = 0; i < n; i++)                            // compute LCP in O(n)
    LCP[i] = PLCP[SA[i]];   // put the permuted LCP to the correct position
}

ii stringMatching() {                      // string matching in O(m log n)
  int lo = 0, hi = n-1, mid = lo;              // valid matching = [0..n-1]
  while (lo < hi) {                                     // find lower bound
    mid = (lo + hi) / 2;                              // this is round down
    int res = strncmp(T + SA[mid], P, m);  // try to find P in suffix 'mid'
    if (res >= 0) hi = mid;        // prune upper half (notice the >= sign)
    else          lo = mid + 1;           // prune lower half including mid
  }                                      // observe `=' in "res >= 0" above
  if (strncmp(T + SA[lo], P, m) != 0) return ii(-1, -1);    // if not found
  ii ans; ans.first = lo;
  lo = 0; hi = n - 1; mid = lo;
  while (lo < hi) {            // if lower bound is found, find upper bound
    mid = (lo + hi) / 2;
    int res = strncmp(T + SA[mid], P, m);
    if (res > 0) hi = mid;                              // prune upper half
```

```
    else         lo = mid + 1;           // prune lower half including mid
  }                           // (notice the selected branch when res == 0)
  if (strncmp(T + SA[hi], P, m) != 0) hi--;                  // special case
  ans.second = hi;
  return ans;
} // return lower/upperbound as first/second item of the pair, respectively

ii LRS() {                 // returns a pair (the LRS length and its index)
  int i, idx = 0, maxLCP = -1;
  for (i = 1; i < n; i++)                          // O(n), start from i = 1
    if (LCP[i] > maxLCP)
      maxLCP = LCP[i], idx = i;
  return ii(maxLCP, idx);
}

int owner(int idx) { return (idx < n-m-1) ? 1 : 2; }

ii LCS() {                 // returns a pair (the LCS length and its index)
  int i, idx = 0, maxLCP = -1;
  for (i = 1; i < n; i++)                          // O(n), start from i = 1
    if (owner(SA[i]) != owner(SA[i-1]) && LCP[i] > maxLCP)
      maxLCP = LCP[i], idx = i;
  return ii(maxLCP, idx);
}

int main() {
  //printf("Enter a string T below, we will compute its Suffix Array:\n");
  strcpy(T, "GATAGACA");
  n = (int)strlen(T);
  T[n++] = '$';
  // if '\n' is read, uncomment the next line
  //T[n-1] = '$'; T[n] = 0;

  // SA stored in global variable, SA[0] = index
  constructSA();                                             // O(n log n)
  computeLCP();                                                    // O(n)

  // LRS demo
  ii ans = LRS();                  // find the LRS of the first input string
  char lrsans[MAX_N];
  strncpy(lrsans, T + SA[ans.second], ans.first);
  printf("\nThe LRS is '%s' with length = %d\n\n", lrsans, ans.first);

  // stringMatching demo
  //printf("\nNow, enter a string P below, we will try to find P in T:\n");
  strcpy(P, "A");
  m = (int)strlen(P);
  // if '\n' is read, uncomment the next line
  //P[m-1] = 0; m--;
  ii pos = stringMatching();
  if (pos.first != -1 && pos.second != -1) {
    printf("%s is found SA[%d..%d] of %s\n", P, pos.first, pos.second, T);
    printf("They are:\n");
    for (int i = pos.first; i <= pos.second; i++)
      printf("  %s\n", T + SA[i]);
  } else printf("%s is not found in %s\n", P, T);

  // LCS demo
  //printf("\nRemember, T = '%s'\nNow, enter another string P:\n", T);
```

```
  // T already has '$' at the back
  strcpy(P, "CATA");
  m = (int)strlen(P);
  // if '\n' is read, uncomment the next line
  //P[m-1] = 0; m--;
  strcat(T, P);                                        // append P
  strcat(T, "#");                                 // add '$' at the back
  n = (int)strlen(T);                                  // update n

  // reconstruct SA of the combined strings
  constructSA();                                      // O(n log n)
  computeLCP();                                             // O(n)
  printf("\nThe LCP information of 'T+P' = '%s':\n", T);
  printf("i\tSA[i]\tLCP[i]\tOwner\tSuffix\n");
  for (int i = 0; i < n; i++)
    printf("%2d\t%2d\t%2d\t%2d\t%s\n", i, SA[i], LCP[i], owner(SA[i]), T + SA[i]);

  ans = LCS();           // find the longest common substring between T and P
  char lcsans[MAX_N];
  strncpy(lcsans, T + SA[ans.second], ans.first);
  printf("\nThe LCS is '%s' with length = %d\n", lcsans, ans.first);

  return 0;
}

const char DUMMY = '.';

int manacher(string s) {
    // Add dummy character to not consider odd/even length
    // NOTE: Ensure DUMMY does not appear in input
    // NOTE: Remember to ignore DUMMY when tracing

    int n = s.size() * 2 - 1;
    vector <int> f = vector <int>(n, 0);
    string a = string(n, DUMMY);
    for (int i = 0; i < n; i += 2) a[i] = s[i / 2];

    int l = 0, r = -1, center, res = 0;
    for (int i = 0, j = 0; i < n; i++) {
        j = (i > r ? 0 : min(f[l + r - i], r - i)) + 1;
        while (i - j >= 0 && i + j < n && a[i - j] == a[i + j]) j++;
        f[i] = --j;
        if (i + j > r) {
            r = i + j;
            l = i - j;
        }

        int len = (f[i] + i % 2) / 2 * 2 + 1 - i % 2;
        if (len > res) {
            res = len;
            center = i;
        }
    }
    // a[center - f[center]..center + f[center]] is the needed substring
    return res;
}

--- Primes
#include <bitset>   // compact STL for Sieve, more efficient than vector<bool>!
```

```
#include <cmath>
#include <cstdio>
#include <map>
#include <vector>
using namespace std;

typedef long long ll;
typedef vector<int> vi;
typedef map<int, int> mii;

ll _sieve_size;
bitset<10000010> bs;   // 10^7 should be enough for most cases
vi primes;   // compact list of primes in form of vector<int>


// first part

void sieve(ll upperbound) {           // create list of primes in [0..upperbound]
  _sieve_size = upperbound + 1;                     // add 1 to include upperbound
  bs.set();                                                  // set all bits to 1
  bs[0] = bs[1] = 0;                                      // except index 0 and 1
  for (ll i = 2; i <= _sieve_size; i++) if (bs[i]) {
    // cross out multiples of i starting from i * i!
    for (ll j = i * i; j <= _sieve_size; j += i) bs[j] = 0;
    primes.push_back((int)i);  // also add this vector containing list of primes
} }                                              // call this method in main method

bool isPrime(ll N) {                   // a good enough deterministic prime tester
  if (N <= _sieve_size) return bs[N];                   // O(1) for small primes
  for (int i = 0; i < (int)primes.size(); i++)
    if (N % primes[i] == 0) return false;
  return true;                    // it takes longer time if N is a large prime!
}                       // note: only work for N <= (last prime in vi "primes")^2


// second part
vi primeFactors(ll N) {   // remember: vi is vector of integers, ll is long long
  vi factors;                    // vi `primes' (generated by sieve) is optional
  ll PF_idx = 0, PF = primes[PF_idx];     // using PF = 2, 3, 4, ..., is also ok
  while (N != 1 && (PF * PF <= N)) {    // stop at sqrt(N), but N can get smaller
    while (N % PF == 0) { N /= PF; factors.push_back(PF); }     // remove this PF
    PF = primes[++PF_idx];                                 // only consider primes!
  }
  if (N != 1) factors.push_back(N);      // special case if N is actually a prime
  return factors;         // if pf exceeds 32-bit integer, you have to change vi
}

ll numDiv(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 1;             // start from ans = 1
  while (N != 1 && (PF * PF <= N)) {
    ll power = 0;                                               // count the power
    while (N % PF == 0) { N /= PF; power++; }
    ans *= (power + 1);                                   // according to the formula
    PF = primes[++PF_idx];
  }
  if (N != 1) ans *= 2;             // (last factor has pow = 1, we add 1 to it)
  return ans;
}
```

```
ll sumDiv(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 1;              // start from ans = 1
  while (N != 1 && (PF * PF <= N)) {
    ll power = 0;
    while (N % PF == 0) { N /= PF; power++; }
    ans *= ((ll)pow((double)PF, power + 1.0) - 1) / (PF - 1);          // formula
    PF = primes[++PF_idx];
  }
  if (N != 1) ans *= ((ll)pow((double)N, 2.0) - 1) / (N - 1);         // last one
  return ans;
}

ll EulerPhi(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = N;              // start from ans = N
  while (N != 1 && (PF * PF <= N)) {
    if (N % PF == 0) ans -= ans / PF;                 // only count unique factor
    while (N % PF == 0) N /= PF;
    PF = primes[++PF_idx];
  }
  if (N != 1) ans -= ans / N;                                    // last factor
  return ans;
}

int main() {
  // first part: the Sieve of Eratosthenes
  sieve(10000000);                              // can go up to 10^7 (need few seconds)
  printf("%d\n", isPrime(2147483647));                         // 10-digits prime
  printf("%d\n", isPrime(136117223861LL));         // not a prime, 104729*1299709

  res = primeFactors(142391208960LL);   // faster, 2^10*3^4*5*7^4*11*13
  for (vi::iterator i = res.begin(); i != res.end(); i++) printf("! %d\n", *i);

  //res = primeFactors((ll)(1010189899 * 1010189899)); // "error"
  //for (vi::iterator i = res.begin(); i != res.end(); i++) printf("^ %d\n", *i);

  // third part: prime factors variants
  printf("numDiv(%d) = %lld\n", 50, numDiv(50)); // 1, 2, 5, 10, 25, 50, 6 divisors
  printf("sumDiv(%d) = %lld\n", 50, sumDiv(50)); // 1 + 2 + 5 + 10 + 25 + 50 = 93
  printf("EulerPhi(%d) = %lld\n", 50, EulerPhi(50)); // 20 integers < 50 are relatively
  prime with 50

  return 0;
}

--- Extended Euclidean
// a x + b y = gcd(a, b)
// x = x0 + (b/g)n, y = y0 - (a/g)n, for int n
int extgcd(int a, int b, int &x, int &y) {
    int g = a; x = 1; y = 0;
    if (b != 0) g = extgcd(b, a % b, y, x), y -= (a / b) * x;
    return g;
}

--- Chinese Remainder Theorem
bool linearCongruences(const vector<int> &a, const vector<int> &b,
         const vector<int> &m, int &x, int &M) {
   int n =     a.size();
   x = 0; M = 1;
   REP(i, n) {
```

```
    int a_ = a[i] % M, b_ = b[i] - a[i] * x, m_ = m[i];
    int y, t, g = extgcd(a_, m_, y, t);
    if (b_ % g) return false;
    b_ /= g; m_ /= g;
    x += M * (y * b_ % m_);
    M *= m_;
  }
  x = (x + M) % M;
  return true;
}
```

```
--- Cycle Finding
#include <cstdio>
using namespace std;

typedef pair<int, int> ii;

int caseNo = 1, Z, I, M, L;

int f(int x) { return (Z * x + I) % M; }

ii floydCycleFinding(int x0) { // function "int f(int x)" must be defined earlier
  // 1st part: finding v, hare's speed is 2x tortoise's
  int tortoise = f(x0), hare = f(f(x0)); // f(x0) is the element/node next to x0
  while (tortoise != hare) { tortoise = f(tortoise); hare = f(f(hare)); }
  // 2nd part: finding mu, hare and tortoise move at the same speed
  int mu = 0; hare = x0;
  while (tortoise != hare) { tortoise = f(tortoise); hare = f(hare); mu++; }
  // 3rd part: finding lambda, hare moves, tortoise stays
  int lambda = 1; hare = f(tortoise);
  while (tortoise != hare) { hare = f(hare); lambda++; }
  return ii(mu, lambda);
}

int main() {
  while (scanf("%d %d %d %d", &Z, &I, &M, &L), (Z || I || M || L)) {
    ii result = floydCycleFinding(L);
    printf("Case %d: %d\n", caseNo++, result.second);
  }
  return 0;
}
```

```
--- Fibonacci Matrix Exponentiation
#include <cmath>
#include <cstdio>
#include <cstring>
using namespace std;

typedef long long ll;
ll MOD;

#define MAX_N 2                                   // increase this if needed
struct Matrix { ll mat[MAX_N][MAX_N]; };      // to let us return a 2D array

Matrix matMul(Matrix a, Matrix b) {              // O(n^3), but O(1) as n=2
  Matrix ans; int i, j, k;
  for (i = 0; i < MAX_N; i++)
    for (j = 0; j < MAX_N; j++)
```

```
      for (ans.mat[i][j] = k = 0; k < MAX_N; k++) {
        ans.mat[i][j] += (a.mat[i][k] % MOD) * (b.mat[k][j] % MOD);
        ans.mat[i][j] %= MOD;              // modulo arithmetic is used here
      }
  return ans;
}

Matrix matPow(Matrix base, int p) {    // O(n^3 log p), but O(log p) as n=2
  Matrix ans; int i, j;
  for (i = 0; i < MAX_N; i++)
    for (j = 0; j < MAX_N; j++)
      ans.mat[i][j] = (i == j);                     // prepare identity matrix
  while (p) {       // iterative version of Divide & Conquer exponentiation
    if (p & 1)                    // check if p is odd (the last bit is on)
      ans = matMul(ans, base);                                // update ans
    base = matMul(base, base);                               // square the base
    p >>= 1;                                                 // divide p by 2
  }
  return ans;
}

int fastExp(int base, int p) {                                  // O(log p)
       if (p == 0) return 1;
  else if (p == 1) return base;
  else {             int res = fastExp(base, p / 2); res *= res;
                     if (p % 2 == 1) res *= base;
                     return res; } }

int main() {
  int i, n, m;

  while (scanf("%d %d", &n, &m) == 2) {
    Matrix ans;                                 // special Fibonaccci matrix
    ans.mat[0][0] = 1;  ans.mat[0][1] = 1;
    ans.mat[1][0] = 1;  ans.mat[1][1] = 0;
    for (MOD = 1, i = 0; i < m; i++)                       // set MOD = 2^m
      MOD *= 2;
    ans = matPow(ans, n);                                      // O(log n)
    printf("%lld\n", ans.mat[0][1]);                       // this if fib(n)
  }

  return 0;
}

--- Gaussian Elimination
#include <cmath>
#include <cstdio>
using namespace std;

#define MAX_N 3                              // adjust this value as needed
struct AugmentedMatrix { double mat[MAX_N][MAX_N + 1]; };
struct ColumnVector { double vec[MAX_N]; };

ColumnVector GaussianElimination(int N, AugmentedMatrix Aug) {
  // input: N, Augmented Matrix Aug, output: Column vector X, the answer
  int i, j, k, l; double t;

  for (i = 0; i < N - 1; i++) {               // the forward elimination phase
    l = i;
```

```
    for (j = i + 1; j < N; j++)        // which row has largest column value
      if (fabs(Aug.mat[j][i]) > fabs(Aug.mat[l][i]))
        l = j;                                         // remember this row l
    // swap this pivot row, reason: minimize floating point error
    for (k = i; k <= N; k++)            // t is a temporary double variable
      t = Aug.mat[i][k], Aug.mat[i][k] = Aug.mat[l][k], Aug.mat[l][k] = t;
    for (j = i + 1; j < N; j++)      // the actual forward elimination phase
      for (k = N; k >= i; k--)
        Aug.mat[j][k] -= Aug.mat[i][k] * Aug.mat[j][i] / Aug.mat[i][i];
  }

  ColumnVector Ans;                            // the back substitution phase
  for (j = N - 1; j >= 0; j--) {                            // start from back
    for (t = 0.0, k = j + 1; k < N; k++) t += Aug.mat[j][k] * Ans.vec[k];
    Ans.vec[j] = (Aug.mat[j][N] - t) / Aug.mat[j][j]; // the answer is here
  }
  return Ans;
}

int main() {
  AugmentedMatrix Aug;
  Aug.mat[0][0] = 1; Aug.mat[0][1] = 1; Aug.mat[0][2] = 2; Aug.mat[0][3] = 9;
  Aug.mat[1][0] = 2; Aug.mat[1][1] = 4; Aug.mat[1][2] = -3; Aug.mat[1][3] = 1;
  Aug.mat[2][0] = 3; Aug.mat[2][1] = 6; Aug.mat[2][2] = -5; Aug.mat[2][3] = 0;

  ColumnVector X = GaussianElimination(3, Aug);
  printf("X = %.1lf, Y = %.1lf, Z = %.1lf\n", X.vec[0], X.vec[1], X.vec[2]);

  return 0;
}

--- Pollards rho big integer factoring
import java.util.*;

class Pollardsrho {
  public static long mulmod(long a, long b, long c) { // returns (a * b) % c, and minimize
  overflow
    long x = 0, y = a % c;
    while (b > 0) {
      if (b % 2 == 1) x = (x + y) % c;
      y = (y * 2) % c;
      b /= 2;
    }
    return x % c;
  }
  public static long abs_val(long a) { return a >= 0 ? a : -a; }
  public static long gcd(long a, long b) { return b == 0 ? a : gcd(b, a % b); } // standard
  gcd

  public static long pollard_rho(long n) {
    int i = 0, k = 2;
    long x = 3, y = 3;              // random seed = 3, other values possible
    while (true) {
      i++;
      x = (mulmod(x, x, n) + n - 1) % n;              // generating function
      long d = gcd(abs_val(y - x), n);                        // the key insight
      if (d != 1 && d != n) return d;         // found one non-trivial factor
      if (i == k) { y = x; k *= 2; }
    }
```

```
  }

  public static void main(String[] args) {
    long n = 2063512844981574047L; // we assume that n is not a large prime
    long ans = pollard_rho(n);      // break n into two non trivial factors
    if (ans > n / ans) ans = n / ans;         // make ans the smaller factor
    System.out.println(ans + " " + (n / ans)); // should be: 1112041493 1855607779
  }
} -- ;

--- FFT
typedef complex<double> Base;
int rev[MN];
Base wlen_pw[MN];
void eval(Base a[], int n, bool invert) {
    for (int i=0; i<n; ++i)
        if (i < rev[i]) swap(a[i], a[rev[i]]);
    for (int len=2; len<=n; len<<=1) {
        double ang = 2*M_PI/len * (invert?-1:+1);
        int len2 = len>>1;
        Base wlen (cos(ang), sin(ang));
        wlen_pw[0] = Base(1, 0);
        for (int i=1; i<len2; ++i)
            wlen_pw[i] = wlen_pw[i-1] * wlen;
        for (int i=0; i<n; i+=len) {
            Base t,
                *pu = a+i,
                *pv = a+i+len2,
                *pu_end = a+i+len2,
                *pw = wlen_pw;
            for (; pu!=pu_end; ++pu, ++pv, ++pw) {
                t = *pv * *pw;
                *pv = *pu - t;
                *pu += t;
            }
        }
    }
    if (invert)
        for (int i=0; i<n; ++i)
            a[i] /= n;
}

void calc_rev(int n, int log_n) {
    for (int i=0; i<n; ++i) {
        rev[i] = 0;
        for (int j=0; j<log_n; ++j)
            if (i & (1<<j))
                rev[i] |= 1<<(log_n-1-j);
    }
}

// multiply a[0] * a[1] and store into a[2]
void multiply(Base a[][MN], int n) {
    int outN = 1, lg = 1;
    while (outN < n) outN <<= 1, ++lg;
    outN <<= 1;
    calc_rev(outN, lg);
    eval(a[0], outN, false);
    eval(a[1], outN, false);
```

```
    for(int i = 0; i < outN; ++i)
        a[2][i] = a[0][i] * a[1][i];
    eval(a[2], outN, true);
}

// Convex hull optimization
// Original Recurrence:
//   dp[i] = min( dp[j] + b[j]*a[i] )  for j < i
// Condition:
//   b[j] >= b[j+1]
//   a[i] <= a[i+1]
// To solve:
// Hull hull;
// FOR(i,1,n) {
//   f[i] = hull.get(a[i]);
//   hull.add(b[i], f[i]);
// }

const int MAXN = 100100;

struct Hull {
    long long a[MAXN], b[MAXN];
    double x[MAXN];
    int head, tail;

    Hull(): head(1), tail(0) {}

    long long get(long long xQuery) {
        if (head > tail) return 0;
        while (head < tail && x[head + 1] <= xQuery) head++;
        x[head] = xQuery;
        return a[head] * xQuery + b[head];
    }

    void add(long long aNew, long long bNew) {
        double xNew = -1e18;
        while (head <= tail) {
            if (aNew == a[tail]) return;
            xNew = 1.0 * (b[tail] - bNew) / (aNew - a[tail]);
            if (head == tail || xNew >= x[tail]) break;
            tail--;
        }
        a[++tail] = aNew;
        b[tail] = bNew;
        x[tail] = xNew;
    }
};

// http://codeforces.com/blog/entry/8219
// Divide and conquer optimization:
// Original Recurrence
//   dp[i][j] = min(d[i-1][k] + C[k][j]) for k < j
// Sufficient condition:
//   A[i][j] <= A[i][j+1]
//   where A[i][j] = smallest k that gives optimal answer
// How to use:
//   // compute i-th row of dp from L to R. optL <= A[i][L] <= A[i][R] <= optR
//   compute(i, L, R, optL, optR)
//       1. special case L == R
```

```
//       2. let M = (L + R) / 2. Calculate dp[i][M] and opt[i][M] using O(optR - optL + 1)
//       3. compute(i, L, M-1, optL, opt[i][M])
//       4. compute(i, M+1, R, opt[i][M], optR)

// Example: http://codeforces.com/contest/321/problem/E
#include "../template.h"

const int MN = 4011;
const int inf = 1000111000;
int n, k, cost[MN][MN], dp[811][MN];

inline int getCost(int i, int j) {
    return cost[j][j] - cost[j][i-1] - cost[i-1][j] + cost[i-1][i-1];
}

void compute(int i, int L, int R, int optL, int optR) {
    if (L > R) return ;

    int mid = (L + R) >> 1, savek = optL;
    dp[i][mid] = inf;
    FOR(k,optL,min(mid-1, optR)) {
        int cur = dp[i-1][k] + getCost(k+1, mid);
        if (cur < dp[i][mid]) {
            dp[i][mid] = cur;
            savek = k;
        }
    }
    compute(i, L, mid-1, optL, savek);
    compute(i, mid+1, R, savek, optR);
}

int main() {
    ios :: sync_with_stdio(false); cin.tie(NULL);
    while (cin >> n >> k) {
        FOR(i,1,n) FOR(j,1,n) {
            cin >> cost[i][j];
            cost[i][j] = cost[i-1][j] + cost[i][j-1] - cost[i-1][j-1] + cost[i][j];
        }

        dp[0][0] = 0;
        FOR(i,1,n) dp[0][i] = inf;

        FOR(i,1,k) {
            compute(i, 1, n, 0, n);
        }
        cout << dp[k][n] / 2 << endl;
    }
    return 0;
}

// knuth optimization
// http://codeforces.com/blog/entry/8219
// Original Recurrence:
//   dp[i][j] = min(dp[i][k] + dp[k][j]) + C[i][j]   for k = i+1..j-1
// Necessary & Sufficient Conditions:
//   A[i][j-1] <= A[i][j] <= A[i+1][j]
//   with A[i][j] = smallest k that gives optimal answer
// Also applicable if the following conditions are met:
//   1. C[a][c] + C[b][d] <= C[a][d] + C[b][c] (quadrangle inequality)
```

```
//   2. C[b][c] <= C[a][d]                    (monotonicity)
//   for all a <= b <= c <= d
// To use:
//   Calculate dp[i][i] and A[i][i]
//
//   FOR(len = 1..n-1)
//     FOR(i = 1..n-len) {
//       j = i + len
//       FOR(k = A[i][j-1]..A[i+1][j])
//         update(dp[i][j])
//     }

// OPTCUT
#include "../template.h"

const int MN = 2011;
int a[MN], dp[MN][MN], C[MN][MN], A[MN][MN];
int n;

int main() {
    while (cin >> n) {
        FOR(i,1,n) {
            cin >> a[i];
            a[i] += a[i-1];
        }
        FOR(i,1,n) FOR(j,i,n) C[i][j] = a[j] - a[i-1];

        FOR(i,1,n) dp[i][i] = 0, A[i][i] = i;

        FOR(len,1,n-1)
            FOR(i,1,n-len) {
                int j = i + len;
                dp[i][j] = 2000111000;
                FOR(k,A[i][j-1],A[i+1][j]) {
                    int cur = dp[i][k-1] + dp[k][j] + C[i][j];
                    if (cur < dp[i][j]) {
                        dp[i][j] = cur;
                        A[i][j] = k;
                    }
                }
            }
        cout << dp[1][n] << endl;
    }
}
```