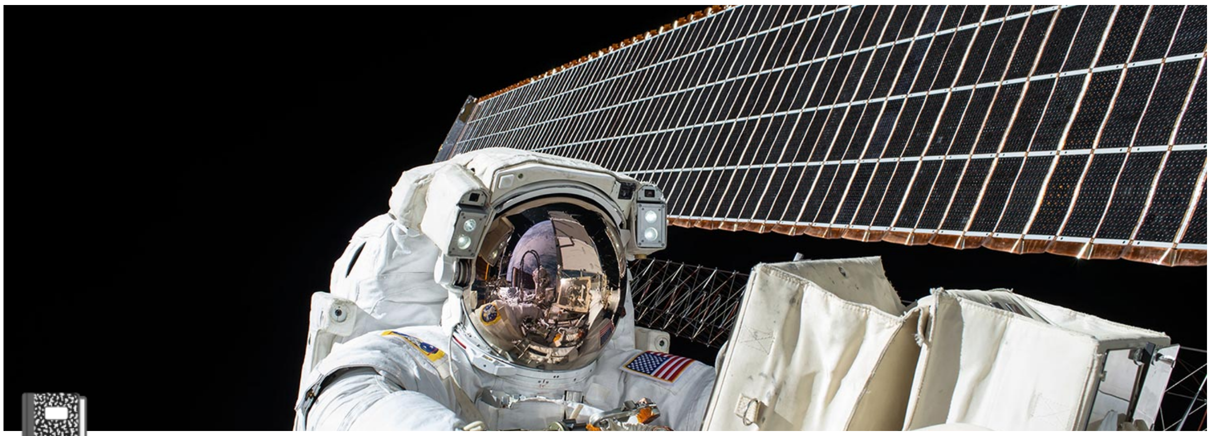


📓

NOI Printed Notes

Compiling
Headers
Interactive and Communication problems
Bitset
Binary Search
Bitwise operations/Masking
BFS
BFS and DFS with Matrix Adjacency
__builtin
Bellman-Ford for single source vertex with cycles (Condition: No negative cycles - ie cycle cannot add up to be <0)
Combining adjacent values with same sign
Counting Sort
DFS
Diameter of tree (ie longest path in a tree) using double BFS
DFS parenting
Dijkstra
Disjoint with path compression
Disjoint with rank/size (check code)
Discretization
Euler Tour
Floyd-Warshall for All nodes
Fenwick tree → Range query, point update. Range update, point query. Range update, range query.
Kruskal’s algorithm
Longest subarray with the same value
LCA Usage
LCA using DFS parenting/recursion
LCA using Binary Lifting
Monotonic Stack and Monotonic Queue
Modular Arithmatics
Pointers and Iterators
Priority Queue
Prim’s algorithm
Prefix Sum (1D and 2D) and Prefix
Segment tree, minimum query
Segment tree, range sum query, with and without lazy propagation
Strongly Connected Component - Tarjan
Sorting by Edges

Sets of Pairs
Unordered vs Ordered maps (Counting stuff)
Notes/Tricks/Utility/Etc
Inspiration from History
New
  Bitwise Tricks
  2D Fenwick Tree Point Update
  2D Fenwick with Range query and update (Replaced 2d segment tree)

Compiling

```
g++ -Wall -O2 -std=c++17 -o "%e" "%f" -I . //build
g++ -std=c++11 -Wall -Wl,--stack -Wl,500000000 -o "%e" "%f". //compile
```

Headers

```
#include<bits/stdc++.h>
using namespace std;
#define fi first
#define se second
#define pii pair<int,int>
#define pll pair<long long,long long>
#define pb push_back
#define debug(x) cerr<<#x<<"="<<x<<endl
#define pq priority_queue
#define ll long long
void inc(int &a,int b) {a=(a+b)%mod;}
void dec(int &a,int b) {a=(a-b+mod)%mod;}
int lowbit(int x) {return x&(-x);}
ll p0w(ll base,ll p) {ll ret=1;while(p>0){if (p%2ll==1ll)
ret=ret*base%mod;base=base*base%mod;p/=2ll;}return ret;}


int main() {
// freopen("input.txt","r",stdin);
std::ios::sync_with_stdio(false);cin.tie();
return 0;
}
```

Interactive and Communication problems

Interactive Questions

- Given two files.
- grader.cpp and ping.cpp (for question ‘ping’)
- Submit the ping.cpp file
- Tactic: Copy everything from the grader file into the ping.cpp file
- Copy it above.
- Then just call the stuff. This makes compiling much much simpler.
- For searching questions: binary search or noticing a certain position gives the answer is the best way. Probably the former, the latter is too easy.

Communicative Questions

- Usually have 2 functions, send and receive.
- You send data with one
- You receive with the other
- You have to decode what the data input was

```
int send(int answer){
  return a;
}
int receive(int a){
  return answer;
}
```

- the key thing to note is that in these questions, global variables, if used in ‘send’, will be scrambled. Don’t know how, don’t ask me. So, don’t use global variables.

Bitset

`bitset`

The `bitset` class also provides various other methods to manipulate the bits, such as:

- `any()`: returns `true` if at least one bit is set to 1, `false` otherwise.
- `none()`: returns `true` if no bits are set to 1, `false` otherwise.
- `count()`: returns the number of bits set to 1.
- `set()`: sets all the bits to 1.
- `reset()`: sets all the bits to 0.
- `flip()`: inverts the value of all the bits.
- `to_ulong()`: converts the `bitset` to a `unsigned long` integer.
- `to_ullong()`: converts the `bitset` to a `unsigned long long` integer.
- `to_string()`: returns a string representation of the `bitset`.

For example, the following code sets all the bits to 1, then inverts their values and prints the number of bits set to 1:

```
std::bitset<8> mybitset;
```

Binary Search

- If minimum and function is monotonic → ie if x is the answer, x+1 also possible (but not optimal) → then use BSTA

```
while (high - low > 1) {
        int mid = (high + low) >> 1;
        if (possible(mid)) high = mid;
        else low = mid;
    }
```

```
//for questions you don't need to track back the answer and there is a
//check you can do -> logN * (Check complexi
```

Bitwise operations/Masking

1. The **& (bitwise AND)** in C or C++ takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
2. The **| (bitwise OR)** in C or C++ takes two numbers as operands and does OR on every bit of two numbers. The result of OR is 1 if any of the two bits is 1.
3. The **^ (bitwise XOR)** in C or C++ takes two numbers as operands and does XOR on every bit of two numbers. The result of XOR is 1 if the two bits are different.
4. The **<< (left shift)** in C or C++ takes two numbers, left shifts the bits of the first operand, the second operand decides the number of places to shift. Essentially *2
5. The **>> (right shift)** in C or C++ takes two numbers, right shifts the bits of the first operand, the second operand decides the number of places to shift. Essentially /2
6. The **~ (bitwise NOT)** in C or C++ takes one number and inverts all bits of it.

```
int main(){
  cin >> n >> num;
  for(int i =n;i>=0;i--){//note that it is i=n, i>=0, i--, not i++
    if((num >> i) & 1) cout << "1";
    else cout << "0";
  }
}
```

(mask >> i) & 1 → Can know if the bit digit you are at is a 1 or a 0.

BFS

```
//BFS
vector<int> adj[100005];
int dis[100005];

//inisde main
queue<int> Q;
dis[0] = 0;
Q.push(0);

while(not Q.empty()){
  int cur = Q.front(); ///choose the node
  for(int neigh : adj[cur]){
    if(dis[neigh] == inf){ ///unexplored
      dis[neigh] = dis[cur] + 1;
      Q.push(neigh);
    }
  }
  Q.pop(); ///pop the nodes
}
```

BFS and DFS with Matrix Adjacency

```
void dfs(int start, vector<bool>& visited)
{

  // Print the current node
  cout << start << " ";

  // Set current node as visited
  visited[start] = true;

  // For every node of the graph
  for (int i = 0; i < adj[start].size(); i++) {

    // If some node is adjacent to the current node
    // and it has not already been visited
    if (adj[start][i] == 1 && (!visited[i])) {
      dfs(i, visited);
    }
  }
}
```

```
void bfs(int start)
{
    // Visited vector to so that
    // a vertex is not visited more than once
    // Initializing the vector to false as no
    // vertex is visited at the beginning
    vector<bool> visited(adj.size(), false);
    vector<int> q;
    q.push_back(start);

    // Set source as visited
    visited[start] = true;

    int vis;
    while (!q.empty()) {
        vis = q[0];

        // Print the current node
        cout << vis << " ";
        q.erase(q.begin());

        // For every adjacent vertex to the current vertex
        for (int i = 0; i < adj[vis].size(); i++) {
            if (adj[vis][i] == 1 && (!visited[i])) {

                // Push the adjacent node to the queue
                q.push_back(i);

                // Set
                visited[i] = true;
            }
        }
    }
}
```

__builtin

```
int small_expo(int num){
    int exponent = sizeof( int) * 8 - 1 - __builtin_clz(num);
    return exponent;
}
8-> return 3
20 -> return 4
30 -> return 4
34 -> return 5
__builtin_popcount() →
This function is used to count the number of one's(set bits)
in an integer. Example: if x = 4 binary value of 4 is 100, output is 1.
```

Bellman-Ford for single source vertex with cycles (Condition: No negative cycles - ie cycle cannot add up to be <0)

```
void BellmanFord(int graph[][3], int V, int E, int src){
    // Initialize distance of all vertices as infinite.
    int dis[V];
    for (int i = 0; i < V; i++)dis[i] = INT_MAX;


    dis[src] = 0;

    // Relax all edges |V| - 1 times. A simple
    // shortest path from src to any other
    // vertex can have at-most |V| - 1 edges
    for (int i = 0; i < V - 1; i++) {
        for (int j = 0; j < E; j++) {
            if (dis[graph[j][0]] != INT_MAX && dis[graph[j][0]] + graph[j][2] < dis[graph[j][1]])
                dis[graph[j][1]] =dis[graph[j][0]] + graph[j][2];
        }
    }

    // check for negative-weight cycles.
    // The above step guarantees shortest
    // distances if graph doesn't contain
    // negative weight cycle.  If we get a
    // shorter path, then there is a cycle.
    for (int i = 0; i < E; i++) {
        int x = graph[i][0];
        int y = graph[i][1];
        int weight = graph[i][2];
        if (dis[x] != INT_MAX && dis[x] + weight < dis[y])
            cout << "Graph contains negative weight cycle" << endl;
    }

    cout << "Vertex Distance from Source" << endl;
    for (int i = 0; i < V; i++)
        cout << i << "\t\t" << dis[i] << endl;
}
// Every edge has three values (u, v, w) where
    // the edge is from vertex u to v. And weight
    // of the edge is w.
```

Combining adjacent values with same sign

```
vector<int> res;

    res.push_back(arr[0]);

    for (int i = 1; i < arr.size(); i++) {
        if(arr[i]<=0ll and res[res.size()-1]<=0ll) res[res.size()-1]+=arr[i];
        else if(arr[i]>0ll and res[res.size()-1]>0ll) res[res.size()-1]+=arr[i];
        else res.push_back(arr[i]);
    }
```

Counting Sort

ONLY use when the number of possible values are small - ie 26 like alphabet or 1-10, etc. Once the number becomes large, it becomes significantly more inefficient.

The idea is you maintain a frequency table, then convert it to a prefix sum table. This allows for O(1) query of frequency. Then you simply iterate from 0 to n, using prefix sum to find the frequency of each occurrence.

Actually, now that I think of it, using discretization on this can result in pretty good algorithmic timing. Of course, using a sort() would be faster competition-wise, but its an idea.

```
vector<int> countingSort(vector<int> vec, int n)
{
    for (int i = 0; i<n; i++)cin>> vec[i];

    map<int, int> count;
    // Here we are initializing every element of count to 0
    // from 1 to n
    for (int i = 0; i < n;i++)count[i] = 0;
    // Here we are storing count of every element
    for (int i = 0; i < n; i++)count[vec[i]]++;

    vector<int> sortedArr;
    int i = 0;
    while (n > 0) {
        // Here we are checking if the count[element] = 0
        // then incrementing for the next Element
        if (count[i] == 0) {
            i++;
        }
        // Here we are inserting the element into the
        // sortedArr decrementing count[element] and n by 1
        else {
            sortedArr.push_back(i);
            count[i]--;
            n--;
        }
    }
    return sortedArr;
}
```

DFS

```
///DFS
bool vis[1000005];
vector<int> adj[1000005];

void dfs(int cur){
  vis[cur] = true;
  for(int neigh : adj[cur]){
    if(not vis[neigh]){
      dfs(neigh);
    }
  }
}
```

Diameter of tree (ie longest path in a tree) using double BFS

```
int deepest(int n, int x) {
    queue<int> q;
    vector<int> ans(n, -1);
    ans[x] = 0;
    int biggest = x;
    q.push(x);
    while(q.size()) {
        int next = q.front();
        q.pop();
        for(int i: adj[next]) {
            if(ans[i] == -1 && !ban[i]) {
                ans[i] = ans[next] + 1;
                if(ans[i] > ans[biggest]) {
                    biggest = i;
                }
                q.push(i);
            }
        }
    }
    x = biggest;
    assert(q.empty()); //check actually empty
    ans = vector<int>(n, -1); //start declaring again
    ans[x] = 0;
    biggest = x;
    q.push(x);
    while(q.size()) {
        int next = q.front();
        q.pop();
        for(int i: adj[next]) {
            if(ans[i] == -1 && (!ban[i] || !ban[next])) {
                ans[i] = ans[next] + 1;
                if(ans[i] > ans[biggest]) {
                    biggest = i;
                }
                q.push(i);
            }
        }
    }//repeat bfs
    return ans[biggest];
}
```

DFS parenting

```
int parents[100005];
void dfs(int current){

  //start from node 1

  for(int neigh : adj[current]){
    if(neigh != parents[current]){
//for dfs parenting, note that there is no visited node,
//because you want to know the parent of EVERY node.
//however, this makes the algorithm pertty inefficient. probably n^2 for dfs
      parents[neigh] = current;
      dfs(neigh);
    }
  }
}

Example query:
while(x){

    if(bugs[x]){
      bugs[x]--;
      species_eaten.insert(species[x]);
    }
    x = parents[x]; //this part is the most important -> x becomes parent
  }
```

Dijkstra

```
int n,e;
vector <pii> arr[ 100005];
int dis[100005];
priority_queue<pii,vector<pii>,greater<pii>> q;

int main(){
    ios_base::sync_with_stdio(false);
    cin.tie(NULL);
    memset(dis, 0x3F , sizeof(dis));
    cin >> n >> e;

    int a,b,c;
    for(int i =0;i<e;i++){
        cin >> a >> b >> c;
        arr[a].push_back(make_pair(b,c));
        arr[b].push_back(make_pair(a,c));
    }
    dis[1] = 0;
    q.push(make_pair(dis[1], 1));
    int dist, u;
    while(!q.empty()){
        pii cur = q.top();
        q.pop();
        dist = cur.first;
        u = cur.second;
        if(dis[u] < dist)continue;
        for(pii e: arr[u]){
           if(dis[e.first] > dis[u] + e.second){
               dis[e.first] = dis[u] + e.second;
               q.push(make_pair(dis[e.first],e.first));
            }
        }
    }
    if(dis[n] == INT_MAX) cout << "-1";
    else cout << dis[n];
}
//btw add all the starting nodes before starting the q.empty() loop. you can have multiple
//starting nodes too

//a trick for questions that may require you to find shortest distance from one node
//to a number of other end nodes is to reverse it and use the end nodes as the start nodes
//and the start node as the final destination
```

Disjoint with path compression

```
int find_set(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set(parent[v]);
}//log(N) searching

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) parent[b] = a;
}

for(int i=0;i<=n;i++){
   parent[i] = i;
}
```

Disjoint with rank/size (check code)

```
void make_set(int v) {
    parent[v] = v;
    rank[v] = 0;
}
int find_set(int v) {
    if (v == parent[v])
        return v;
    return find_set(parent[v]);
}
void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (rank[a] < rank[b]) swap(a, b);
        parent[b] = a;
        if (rank[a] == rank[b]) rank[a]++;
    }
}
//for rank-> ie the 'depth' of the subgraph
```

```
void make_set(int v) {
    parent[v] = v;
    size[v] = 1;
}
int find_set(int v) {
    if (v == parent[v])
        return v;
    return find_set(parent[v]);
}
void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
        if (size[a] < size[b]) swap(a, b);
        parent[b] = a;
        size[a] += size[b];
    }
}
//for size -> the size lol
```

For rank/size disjoint, I don’t think that path compression is viable since it would ‘defeat’ the purpose of it.

Discretization

```
for(int i =0;i<n;i++){
        scanf("%d",&a[i]);
        vec.push_back(a[i]);
    }

    sort(vec.begin(),vec.end());
    vec.erase(unique(vec.begin(),vec.end()),vec.end());

    for(int i =0;i<n;i++){
        a[i] = upper_bound(vec.begin(),vec.end(),a[i])-vec.begin();
    }
```

```
const int maxn = 100;
int lowbit(int x) {return x&(-x);}
map<int,int> mp;
int n;
int a[maxn];

int main() {
//    freopen("input.txt","r",stdin);
    std::ios::sync_with_stdio(false);cin.tie(0);
    cin>>n;
    for(int i =0;i<n;i++){
    cin >> a[i];
    mp[a[i]]=1;
  }

    int idx=0;
    for (auto &i:mp) i.second = idx++;
    for(int i =0;i<n;i++){
    a[i] = mp[a[i]];
    debug(a[i]);
    }
    return 0;
}
```

Euler Tour

```
//N*2-1
#include <bits/stdc++.h>
using namespace std;

#define MAX 1001

vector<int> adj[MAX];
int vis[MAX];
int Euler[2 * MAX];
```

```
void add_edge(int u, int v)
{
  adj[u].push_back(v);
  adj[v].push_back(u);
}

void eulerTree(int u, int &index)
{
  vis[u] = 1;
  Euler[index++] = u;
  for (auto it : adj[u]) {
    if (!vis[it]) {
      eulerTree(it, index);
      Euler[index++] = u;
    }
  }
}

void printEulerTour(int root, int N)
{
  int index = 0;
  eulerTree(root, index);
  for (int i = 0; i < (2*N-1); i++)
    cout << Euler[i] << " ";
}

int main()
{
  int N = 4;

  add_edge(1, 2);
  add_edge(2, 3);
  add_edge(2, 4);
  printEulerTour(1, N);

  return 0;
}
```

Floyd-Warshall for All nodes

```
void floydWarshall(int dist[][V])
{
  int i, j, k;

    /* Add all vertices one by one to
      the set of intermediate vertices.
      ---> Before start of an iteration, we
      have shortest distances between all
      pairs of vertices such that the shortest
      distances consider only the
      vertices in set {0, 1, 2, .. k-1} as
      intermediate vertices.
      ----> After the end of an iteration,
      vertex no. k is added to the set of
      intermediate vertices and the set
      becomes {0, 1, 2, .. k} */
    for (k = 0; k < V; k++) {
        // Pick all vertices as source one by one
        for (i = 0; i < V; i++) {
            // Pick all vertices as destination for the
            // above picked source
            for (j = 0; j < V; j++) {
                // If vertex k is on the shortest path from
                // i to j, then update the value of
                // dist[i][j]
                if (dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];
            }
        }
    }

    // Print the shortest distance matrix
    printSolution(dist);
}
//but note that the matrix dist representation is not that of adjacency matrix
//it is like this:
/* Let us create the following weighted graph
            10
       (0)------->(3)
        |         /|\
      5 |          |
        |          | 1
       \|/         |
       (1)------->(2)
            3           */
    //int graph[V][V] = { { 0, 5, INF, 10 },
      //                  { INF, 0, 3, INF },
        //                { INF, INF, 0, 1 },
          //              { INF, INF, INF, 0 } };
//where it points
//ie this uses matrix adjacency
```

The Floyd-Warshall algorithm is an algorithm for finding the shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles).

Sometimes, it is better just to use Dijkstra on all nodes → $N^2\log(N)$. But Dijkstra cannot handle negative weights so yeah.

Fenwick tree → Range query, point update. Range update, point query. Range update, range query.

Fenwick can only solve sum query, not minimum, maximum and more complicated stuff

Also $log(N)$ update and $log(N)$ query. Theoretically, it is easier to implement. And also faster. Fenwick tree can also be modified to point query and range update, as well as range update and range query. ONLY 1 index, since it is only then does the bit indexing works Note that val to update is an increment value → For example, want to update 5 to 6? val = 1.

So for range sum→ `return sum from 1 to 6 = tree[6] + tree[6-2 = 4].` You know that the size of 6 is 2 and size of 4 is 4. How do you know? The lowest bit that it is on. For example, for 4 = 100 → Lowest bit is 100→ 4, thus size is 4. To build, just use the update function. Simple.

Range query, point update.

```
#include <bits/stdc++.h>
using namespace std;
int lowbit(int x){return x&(-x);}

int n,m;
int arr[100010];
int tree[200010];
void update(int pos, int val){
  pos++;
  while(pos <= n){
    tree[pos] += val;
    pos += lowbit(pos);//jumping to the next range to update it
  }
}
// Returns sum of arr[0..index].
int query(int pos){
  int ret = 0;
  pos++;
  while(pos>0){
    ret+= tree[pos];
    pos -= lowbit(pos);
  }
  return ret;
}
```

```
/**

10
1 2 9 5 7 6 3 4 2 2
4
0 1
0 9
0 9
5 7
 * **/
int main(){
  cin >> n;
  for(int i =0;i<n;i++){
    cin >> arr[i];
    update(i, arr[i]);//incremental update btw
  }
  cin >> m;
  for(int i =0;i<m;i++){
      int a,b;
      cin >> a >> b;
      cout << query(b) - query(a-1) << "\n";
      //needs to be query(a-1) or else it includes the element at a into it as well
  }

}
```

Range update, point query

```
//modify into difference between elements first.
#include <bits/stdc++.h>
using namespace std;
int lowbit(int x){return x&(-x);}

int n,m;
int arr[100010];
int dif[100010];
int tree[200010];
void update(int pos, int val){
  pos++;
  while(pos <= n){
    tree[pos] += val;
    pos += lowbit(pos);//jumping to the next range to update it
  }
}
// Returns sum of arr[0..index].

void range_update(int l, int r, int val){
  update(l,val);
  update(r+1, -val);
}

int query(int pos){
  int ret = 0;
  pos++;
  while(pos>0){
    ret+= tree[pos];
    pos -= lowbit(pos);
  }
  return ret;
}

/**

10
1 2 9 5 7 6 3 4 2 2
10
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
0
1
0
5
```

```
5 3 -> becomes:
1 2 9 5 7 9 6 7 5 5
 * **/
int main(){
  cin >> n;
  for(int i =0;i<n;i++){
    cin >> arr[i];
    //update(i, arr[i]);
  }
  for(int i =0;i<n;i++){
    dif[i] = arr[i] - arr[i-1];
    update(i, dif[i]);
  }
  cin >> m;

  for(int i =0;i<m;i++){
    int a;
    cin >> a;
    printf("A:%d: %d\n", i, query(a));
  }
  range_update(1,3, 2);
  for(int i =0;i<m;i++){
    int a;
    cin >> a;
    printf("B:%d: %d\n", i, query(a));
  }
}
```

Range update and Range query

> We get range sum using prefix sums. How to make sure that
> the update is done in a way so that prefix sum can be done quickly?
> Consider a situation where prefix sum [0, k] (where 0 <= k < n) is
> needed after range update on the range [l, r]. Three cases arise as k
> can possibly lie in 3 regions.
>
> - **Case 1**: 0 < k < l
>   - The update query won't affect sum query.
> - **Case 2**: l <= k <= r
>   - Consider an example: Add 2 to range [2, 4], the resultant array would be: 0 0 2 2 2If k = 3 The sum from [0, k] = 4
>
> How to get this result? Simply add the val from lth index to kth index. Sum is incremented by "val*(k) – val*(l-1)" after the update query.
>
> - **Case 3**: k > r
>   - For this case, we need to add "val" from l index to r index. Sum is incremented by "val*r – val*(l-1)" due to an update query.
>
> > ### Update Query
> >
> > Update(BITree1, l, val)
> >
> > Update(BITree1, r+1, -val)

> > UpdateBIT2(BITree2, l, val*(l-1))
> >
> > UpdateBIT2(BITree2, r+1, -val*r)
> >
> > **Range Sum**
> >
> > getSum(BITTree1, k) *k) – getSum(BITTree2, k)

```
//for this we will need two fenwicks, basically using
//some tricks to combine the two previous things together
// C++ program to demonstrate Range Update
// and Range Queries using BIT
#include <bits/stdc++.h>
using namespace std;

// Returns sum of arr[0..index]. This function assumes
// that the array is preprocessed and partial sums of
// array elements are stored in BITree[]
int getSum(int BITree[], int index)
{
  int sum = 0;
  index = index + 1;

  while (index > 0) {
    sum += BITree[index];
    index -= index & (-index);
  }
  return sum;
}

void updateBIT(int BITree[], int n, int index, int val)
{
  index = index + 1;
  while (index <= n) {
    // Add 'val' to current node of BI Tree
    BITree[index] += val;
    index += index & (-index);
  }
}

int sum(int x, int BITTree1[], int BITTree2[])
{
  return (getSum(BITTree1, x) * x) - getSum(BITTree2, x);
}

void updateRange(int BITTree1[], int BITTree2[], int n,
        int val, int l, int r)
{
  updateBIT(BITTree1, n, l, val);
  updateBIT(BITTree1, n, r + 1, -val);

  updateBIT(BITTree2, n, l, val * (l - 1));
  updateBIT(BITTree2, n, r + 1, -val * r);
}

int rangeSum(int l, int r, int BITTree1[], int BITTree2[])
{

  return sum(r, BITTree1, BITTree2)- sum(l - 1, BITTree1, BITTree2);
}

int* constructBITree(int n)
{
  // Create and initialize BITree[] as 0
  int* BITree = new int[n + 1];
  for (int i = 1; i <= n; i++)
    BITree[i] = 0;

  return BITree;
}

// Driver code
int main()
{
  int n = 5;

  // Construct two BIT
  int *BITTree1, *BITTree2;

  // BIT1 to get element at any index
  // in the array
  BITTree1 = constructBITree(n);

  // BIT 2 maintains the extra term
  // which needs to be subtracted
  BITTree2 = constructBITree(n);

  // Add 5 to all the elements from [0,4]
  int l = 0, r = 4, val = 5;
  updateRange(BITTree1, BITTree2, n, val, l, r);

  // Add 10 to all the elements from [2,4]
  l = 2, r = 4, val = 10;
  updateRange(BITTree1, BITTree2, n, val, l, r);

  // Find sum of all the elements from
  // [1,4]
  l = 1, r = 4;
  cout << "Sum of elements from [" << l << "," << r
    << "] is ";
  cout << rangeSum(l, r, BITTree1, BITTree2) << "\n";

  return 0;
}
```

Kruskal’s algorithm

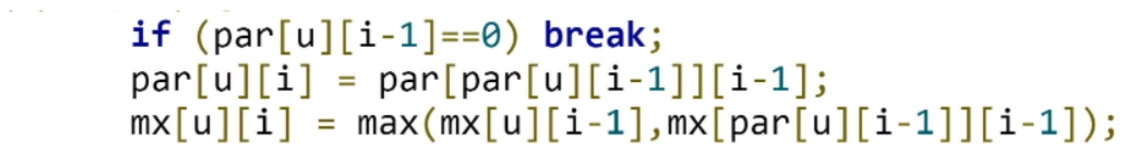
```
struct Edge{
    int u;
    int v;
    int weight;
};

bool sorter(Edge i , Edge j){
    return i.weight > j.weight;
}

int find_set(int v) {
    if (v == parent[v])
        return v;
    return parent[v] = find_set(parent[v]);
}//log(N) searching

void union_sets(int a, int b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) parent[b] = a;
}
int main(){
  sort(edges.begin(), edges.end(), sorter);
  for(Edge e : edges){
        if(find_set(e.u) != find_set(e.v)){
            union_sets(e.u, e.v);
            adj[e.u].push_back({e.v, e.weight});
            adj[e.v].push_back({e.u, e.weight});
        }
    }
}
```

Longest subarray with the same value

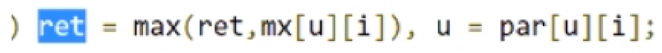
```
int max_length = 1;
    int length = 1;
    for (int i = 1; i < n; i++) {
        if (y_arr[i] == y_arr[i - 1]) {
            length++;
        } else {
            max_length = max(max_length, length);
            length = 1;
        }
    }
    max_length = max(max_length, length);

{1,1,2,3,3,3,3} -> 4 (4 3s)
```

LCA Usage

- Simply looking for LCA
- Distance between two nodes IN A TREE
  - The distance between two nodes in a tree can be calculated as the sum of the depths of the two nodes minus twice the depth of their lowest common ancestor (LCA).
- Dynamic programming problems: LCA can be used in dynamic programming problems to build a recursive solution, where the result of a subproblem is used to solve a larger problem.
- RMQ can be done using LCA
- Maximum weight between two nodes can be found using LCA
  - max[maxn][lg]. operates similarly to par[max][lg] array

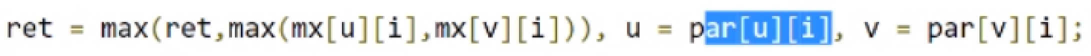


```
if (par[u][i-1]==0) break;
par[u][i] = par[par[u][i-1]][i-1];
mx[u][i] = max(mx[u][i-1],mx[par[u][i-1]][i-1]);
```



```
) ret = max(ret,mx[u][i]), u = par[u][i];
```

for moving up chunk



```
ret = max(ret,max(mx[u][i],mx[v][i])), u = par[u][i], v = par[v][i];
```

for non-equivalent chunk of LCA

LCA using DFS parenting/recursion

See DFS parenting. Simply go from each node in question and track up, until a collision.

Iterating through all the parents by storing each in a vector or array. Very inefficient, O(N*N).

LCA using Binary Lifting

```
#include <bits/stdc++.h>
using namespace std;
```

```
const int nmax = 1e5;
int LN=17; //log2(nmax)
vector<int> adj[nmax];
int dp[nmax][17];//dp[u][i] means the ancestor of u which is 2^i above u
//so for example dp[5][2] is the ancestor of node 5 which is 4=2^2 above node 5
//to find that, you first jump to the ancestor of node which is 2=2^1 above node 5
int level[nmax];
//the forloops, for some reason, MUST be ++i and --i. if it isn't, it does not work. bruh
//i think this is since for ++i and --i, the i variable is incremented BEFORE each iteration
//normally, there isnt an issue. but for this particular case, where the value of i and
//all that jazz is important, it is.
void dfs(int u, int p) {
    level[u] = level[p] + 1;
    dp[u][0] = p;
    for(int i=1; i<LN; ++i)
        dp[u][i] = dp[dp[u][i-1]][i-1];

    for(int i=0; i<adj[u].size(); ++i) {
        int v = adj[u][i];
        if(v == p) continue;
        dfs(v, u);
    }
}

int lca(int u, int v) {
    if(level[u] < level[v])
        swap(u, v);
    int diff = level[u] - level[v];
    for(int i=0; i<LN; ++i) {
        if((1<<i) & diff)//(1<<i) & diff is to find the bits which are on for the binary representation of diff
            u = dp[u][i];
//so for example diff = 5, meaning you want to go up 5 steps
//so for example diff = 5, meaning you want to go up 5 steps
    }
    if(u == v)
        return u;
    for(int i=LN-1; i>=0; --i) {
        if(dp[u][i] != dp[v][i]) {
            u = dp[u][i];
            v = dp[v][i];
        }
    }
    return dp[u][0];
}

int main(){
  int n,m;
  cin >> n >> m;//n-1 vertices, m queries
//note that the LN should literally just be log2(maxN)
//not log2(n)
for(int i =0;i<n-1;i++){
    int u,v;
    cin >> u >> v;
    adj[u].push_back(v);
    adj[v].push_back(u);
  }
  dfs(1,0);
  for(int i =0;i<m;i++){
    int u,v;
    cin>> u >> v;
    cout << lca(u,v) << "\n";
  }
}
```

```
XY's one
const ll maxn=200005;
const ll lg=20;

ll up[lg][maxn];
ll depth[maxn];
vector<ll>alist[maxn];

void dfs(ll u,ll p){
    up[0][u]=p;
    rep(i,1,lg){
        if(up[i-1][u]==-1)break;
        up[i][u]=up[i-1][up[i-1][u]];
    }
    for(auto x:alist[u]){
        if(x!=p){depth[x]=depth[u]+1;dfs(x,u);}
    }
}
ll lca(ll a,ll b){
    if(depth[a]<depth[b])swap(a,b); // let a be the deeper one
    ll k=depth[a]-depth[b];
    rep(i,0,lg){
        if((1<<i)&k)a=up[i][a];
    }
    if(a==b)return a;
    for(ll i=19;i>=0;i--){
        if(up[i][a]!=up[i][b]){
            a=up[i][a];
            b=up[i][b];
        }
    }
    return up[0][a];
}
//to initialise:
memset(up,-1,sizeof up);
dfs(1,-1);
```

Monotonic Stack and Monotonic Queue

```
For stack:
#include <bits/stdc++.h>
#include <stack>
using namespace std;
int main(){
    stack<int> blocks;
    int n;
    cin >> n;
    for(int i =0;i<n;i++){
        int x;
        cin >> x;
        if(blocks.empty())blocks.push(x);
        else{
            while(!blocks.empty() && blocks.top() <= x) blocks.pop();
            blocks.push(x);
        }
    }
    while(!blocks.empty()){
        printf("%d\n", blocks.top());
        blocks.pop();
    }
}
//this works left to right, i think?
```

```
//For deque
//A monotonic deque is a data structure that allows for efficient insertion and
//deletion of elements at both ends of the deque (double-ended queue) while maintaining
//the property of monotonicity. This means that the elements in the deque are either
 //strictly increasing or strictly decreasing.
//often useful for LIS and sliding window maximum/minimum

//the below code is for sliding window maximum/minimum in O(N)
//the question is basically: given an array size n, a window size k, print all max
//as window moves one index at a time
//deque in this case stores position, and the first element is the largest

#include <bits/stdc++.h>
using namespace std;

deque <int> Q;
```

```
int arr[1000100];
/**
8 3
1 3 -1 -3 5 3 6 7
 * **/
int main(){
  int n,k;
  cin >> n >> k;
  for(int i =0;i<n;i++)cin >> arr[i];

  for(int i =0;i<k;i++){
    while(!Q.empty() && arr[Q.back()] < arr[i]) Q.pop_back();
    Q.push_back(i);
  }

  for(int i = k;i<n;i++){
    cout << arr[Q.front()] << " ";

    while(!Q.empty() && Q.front() <= i-k) Q.pop_front();//this part is crucial
//Q.size() does not work because it is possible that the size is smaller
  //but the element in front should already have been erased.
    while(!Q.empty() && arr[Q.back()] < arr[i]) Q.pop_back();

    Q.push_back(i);
  }
  cout << arr[Q.front()] << " ";

}
```

Modular Arithmatics

```
(a x b) mod m = ((a mod m) x (b mod m)) mod m
(a / b) mod m is not equal to ((a mod m) / (b mod m)) mod m.
(a / b) mod m = (a x (inverse of b if exists)) mod m
(a + b) mod m = ((a mod m) + (b mod m)) mod m
```

Pointers and Iterators

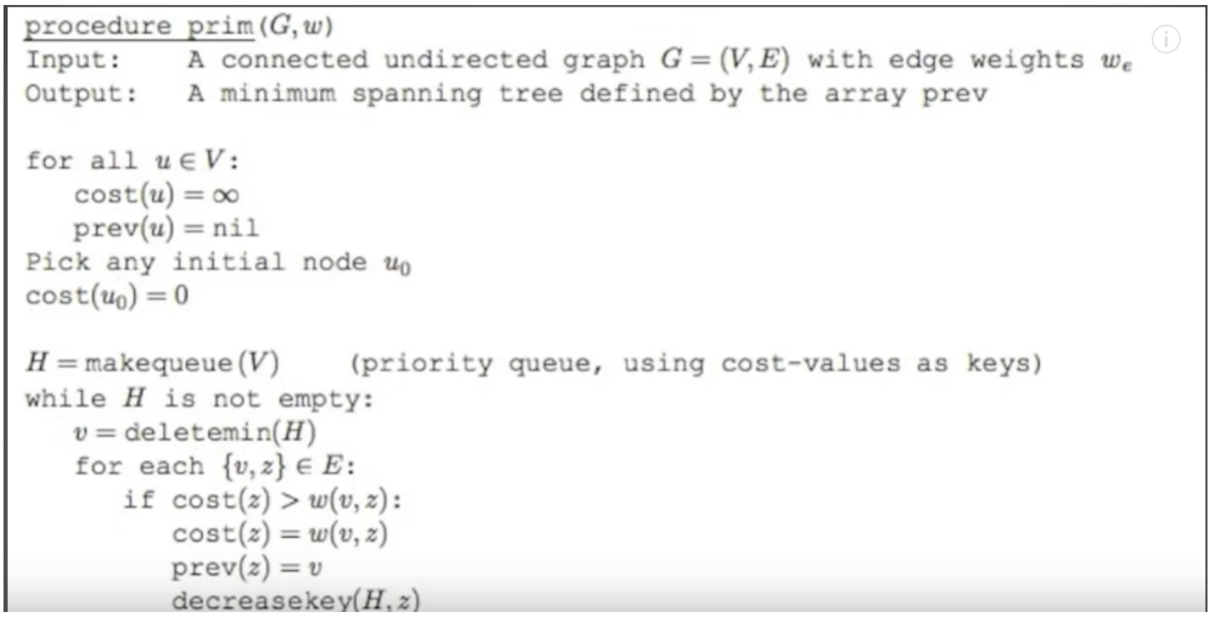
```
advance(iter , n)
prev(iter)
next(iter)
iter++;
iter--;
//advance is honestly the most reliable.

//*iter to access the value
//set.erase(iter) to erase individual values

setter.erase((--setter.end()));
setter.erase((--setter.end()));
setter.erase(setter.begin());
setter.erase(setter.begin());
sum += x->second; x++;
```

Priority Queue

```
priority_queue <pii, vector<pii>, greater<pii>>pq;

//this is MINIMUM priority_queue, ie the pq.top() gives the LOWEST value.
//the default - ie <less> gives maximum value out first

priority_queue <pii, vector<pii>>pq;
```

```
//this is maximum

.push()
.top()
.pop()
```

Prim’s algorithm



```
procedure prim($(G, w)$)
Input:    A connected undirected graph $G = (V, E)$ with edge weights $w_e$
Output:   A minimum spanning tree defined by the array prev

for all $u \in V$:
    cost($u$) $= \infty$
    prev($u$) $=$ nil
Pick any initial node $u_0$
cost($u_0$) $= 0$

$H =$ makequeue($V$)     (priority queue, using cost-values as keys)
while $H$ is not empty:
    $v =$ deletemin($H$)
    for each $\{v, z\} \in E$:
        if cost($z$) $> w(v, z)$:
            cost($z$) $= w(v, z)$
            prev($z$) $= v$
            decreasekey($H, z$)
```

Prim's algorithm is a greedy algorithm used to find the minimum spanning tree of a graph. The algorithm starts at an arbitrary node and adds the least expensive edge at each step until all the nodes in the graph are connected. At each step, the algorithm finds the cheapest edge that connects an already included node to an unvisited node and adds it to the tree. The cost of the minimum spanning tree is the sum of the weights of all the edges included in the tree.

```
void primMST(vector<pair<int,int> > adj[], int V)
{
    // Create a priority queue to store vertices that

    priority_queue< iPair, vector <iPair> , greater<iPair> > pq;

    int src = 0; // Taking vertex 0 as source


    vector<int> key(V, INF);
    vector<int> parent(V, -1);
    vector<bool> inMST(V, false);

    // Insert source itself in priority queue and initialize
    // its key as 0.
    pq.push(make_pair(0, src));
    key[src] = 0;

    /* Looping till priority queue becomes empty */
    while (!pq.empty())
    {

        int u = pq.top().second;
        pq.pop();

          //Different key values for same vertex may exist in the priority queue.
          //The one with the least key value is always processed first.
          //Therefore, ignore the rest.
          if(inMST[u] == true){
            continue;
        }

        inMST[u] = true; // Include vertex in MST

        // Traverse all adjacent of u
        for (auto x : adj[u])
        {
            int v = x.first;
            int weight = x.second;

            // If v is not in MST and weight of (u,v) is smaller
            // than current key of v
            if (inMST[v] == false && key[v] > weight)
            {
                // Updating key of v
                key[v] = weight;
                pq.push(make_pair(key[v], v));
                parent[v] = u;
            }
        }
    }

    // Print edges of MST using parent array
    for (int i = 1; i < V; ++i)
        printf("%d - %d\n", parent[i], i);
}
```

Prefix Sum (1D and 2D) and Prefix

```
prefix[end] - prefix[start-1] = sum of element from start to end, inclusive
both sides
So make sure there is a 0 in front too.
1 2 3 -> 1 3 6

prefix[0] = A[0];
  for(int i =1;i<=n;i++){
    prefix[i] = max(prefix[i-1], A[i]);
  }
  suffix[n-1] = B[n-1];
  for(int i=n-2;i>=0;i--){
    suffix[i] = min(suffix[i+1], B[i]);
  }
```

```
#include <bits/stdc++.h>
using namespace std;

int n,m;
int values[1010][1010];
int prefix[1010][1010];
int temp[1010];

int prefix_sum(int lowx, int lowy, int highx, int highy){
  int ans = prefix[highx][highy];
  ans += prefix[lowx-1][lowy-1];
  ans -= prefix[lowx-1][highy];
  ans -= prefix[highx][lowy-1];
  printf("%d %d %d %d\n", prefix[highx][highy],prefix[lowx-1][lowy-1], prefix[lowx-1][highy], prefix[highx][lowy-1]);
  return ans;
}

int main(){
  cin >> n >> m;
  for(int i=1;i<=n;i++){
    for(int j=1;j<=m;j++){
      cin >> values[i][j];
    }
  }
  for(int i =1;i<=n;i++){
    for(int j=1;j<=m;j++){
      prefix[i][j] = prefix[i][j-1] + prefix[i-1][j] -  prefix[i-1][j-1];
      prefix[i][j] += values[i][j];
    }
  }
  for(int i =0;i<=n;i++){
    for(int j=0;j<=m;j++){
      printf("%d ", prefix[i][j]);
    }
    cout << "\n";
  }
  int lowx, lowy,highx,highy;
  cin >> lowx >> lowy >> highx >> highy;
  cout << prefix_sum(lowx, lowy, highx,highy);
}
/**
 4 4
 1 1 1 1
 1 1 1 1
 1 1 1 1
 1 1 1 1
**/
//this prefix sum is inclusive -> so like 1 1 4 4 would give 16. less
//complicated!!!
```

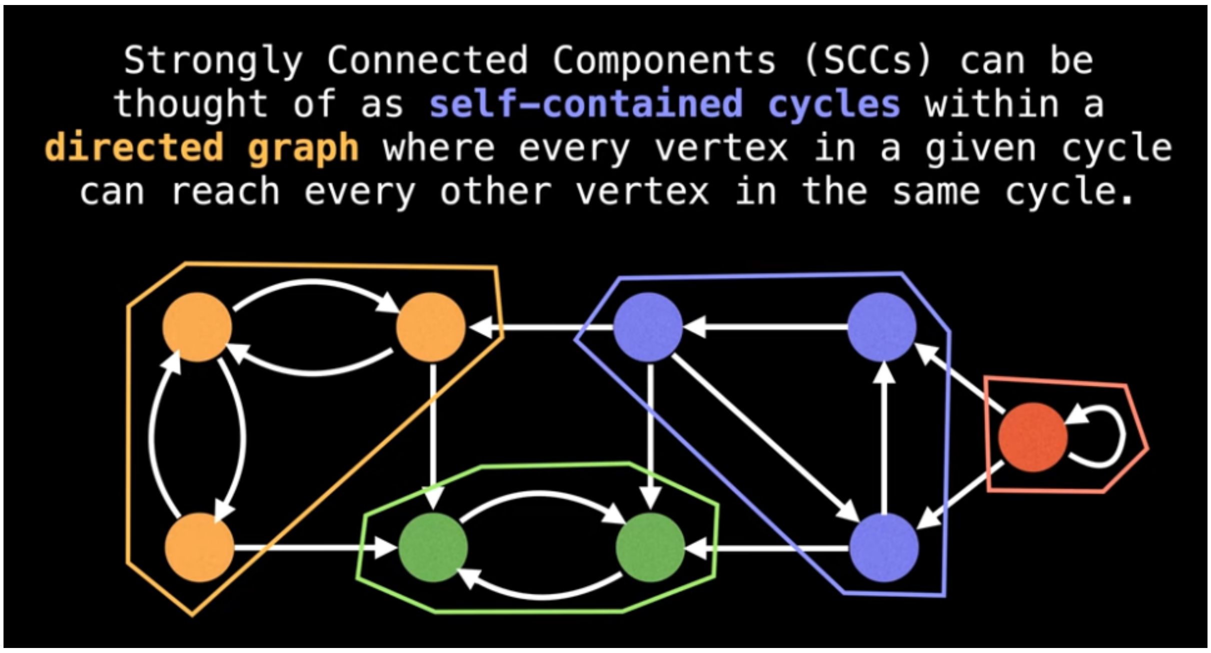
Segment tree, minimum query

```
#include <bits/stdc++.h>
using namespace std;

int n, q, tree[400005], arr[100005];

void update(int tree_pos,int cl,int cr,int pos,int val) {
    if (cl==cr) {
        tree[tree_pos]=val;
        return;
    }
    int mid=cl+cr>>1;
    if (pos<=mid) update(tree_pos<<1,cl,mid,pos,val);
    else update(tree_pos<<1|1,mid+1,cr,pos,val);
    tree[tree_pos]=min(tree[tree_pos<<1],tree[tree_pos<<1|1]);
}


int query(int tree_pos,int cl,int cr,int l,int r) {
    if (l<=cl&&cr<=r) return tree[tree_pos];
    else if(l>cr || cl>r) return INT_MAX;
    int mid=cl+cr>>1;
    int ret=2e9;
    if (l<=mid) ret=min(ret,query(tree_pos<<1,cl,mid,l,r));
    if (r>mid) ret=min(ret,query(tree_pos<<1|1,mid+1,cr,l,r));
    return ret;
}

void build(int node, int start, int end) {
    if (start == end) {
        // Leaf node
        tree[node] = arr[start];
    } else {
        int mid = (start + end) / 2;
        // Recurse on the left and right children
        build(2*node, start, mid);
        build(2*node+1, mid+1, end);
        // Update the current node with the minimum of the left and right children
        tree[node] = min(tree[2*node], tree[2*node+1]);
    }
}
int main(){
    cin >> n >> q;
    for (int i = 0, type, a, b; i < q; i++){
        cin >> type >> a >> b;
        if (type == 1){
            update(1,0, n-1, a,b);
        }
        else{
            cout << query(1, 0,n-1,a,b) << "\n";
        }
    }
}
```

Segment tree, range sum query, with and without lazy propagation

```
#include <bits/stdc++.h>
using namespace std;
#define int long long int
//https://codebreaker.xyz/problem/segmenttree2
struct node{
    int s,e,m;
    int val, lazy;
    node *l = nullptr,*r = nullptr;


    node(int S, int E){//just creating
        s = S, e=E;
        m = (s+e)>>1;
        val = 0, lazy = 0;


    }

    void create(){
        if(s!=e){
            l = new node(s,m);
            r = new node(m+1,e);
        }
    }

    void propagation(){
        if(l == nullptr) create();
 //need to have l==nullptr for every function lol
//or else the l->propagation() would result in the program propagating
//from l, for which l->l == nullptr, thus RTE(11)

        if(lazy == 0) return;

        val += (lazy * (e - s + 1));

        if(s != e){
            l->lazy += lazy;
            r->lazy += lazy;
        }

        lazy =0;
    }

    void update(int S, int E, int increment){
        if(l == nullptr) create(); //lazily create children


        if(S == s && E == e) lazy += increment;
        else{
            if(E <= m) l->update(S, E, increment);
            else if(S > m) r -> update(S, E, increment);// [S,E] is in the right child
            else {
                l-> update(S, m,increment);
                r->update(m+1, E, increment);
            }
            l->propagation(), r->propagation();

            val = l->val + r->val;
        }
    }


    int query(int S, int E){
        if(l == nullptr) create();

        propagation();
        if(S == s && E == e) return val;
        else if(E <= m) return l->query(S, E );
        else if(S > m) return r->query(S, E);
        else return (l->query(S, m) + r->query(m+1,E));
    }
};

signed main(){
    int n,m;
    cin >> n >> m;
    node *root = new node(1,n);

    for(int i =0;i<m;i++){
        int type;
        cin >> type;
        if(type == 1){
            int a,b,c;
            //a--;b--;

            cin >> a >> b >> c;

            root->update(a,b,c);
        }
        else{
            int a,b;
            //a--,b--;
            cin >> a >> b;

            cout << root->query(a,b) << "\n"; //inclusive
        }
    }
}
```

Strongly Connected Component - Tarjan



Strongly Connected Components (SCCs) can be thought of as **self-contained cycles** within a **directed graph** where every vertex in a given cycle can reach every other vertex in the same cycle.

```
#include <bits/stdc++.h>
using namespace std;
/**
dfn is an abbreviation for "discovery time". In Tarjan's algorithm, it is a variable used to keep track
*  of the order in which vertices are discovered during a depth-first search (DFS) of a directed graph.

For each vertex u in the graph, the dfn[u] variable stores the time when the vertex was first discovered
* during the DFS. The discovery time is incremented for each new vertex that is discovered, and it is used
*  in the calculation of the low-link value for each vertex, which is used to identify the strongly connected
*  components (SCCs) in the graph.
**/


const int N = 10010;
vector<int> adj[N];
vector<vector<int>> scc;
int dfn[N], low[N], idx=1;
bool onstack[N];

void tarjan(int u){
    static stack<int> st;

    dfn[u]=low[u]=idx++;
    st.push(u);
    onstack[u]=true;

    for(auto i:adj[u]){
        if(dfn[i]==-1){
            tarjan(i);
            low[u]=min(low[u],low[i]);
        }
        else if(onstack[i])
            low[u]=min(low[u],dfn[i]);
    }
    if(dfn[u]==low[u]){
        vector<int> scctem;
        while(1){
            int v=st.top();
            st.pop();
            onstack[v]=false;
            scctem.push_back(v);
            if(u==v)
                break;
        }
        scc.push_back(scctem);
    }
}

int main() {
  int n, m;
  cin >> n >> m;
  memset(dfn, -1, sizeof(dfn));

  for (int i = 0; i < m; i++) {
    int x, y;
    cin >> x >> y;
    adj[x].push_back(y);
  }
  for (int i = 1; i <= n; i++) {
    if (dfn[i]==-1) {
      tarjan(i);
    }
  }
  for(vector <int> e: scc){
  for(int i:e){
    printf("%d ", i);
  }
  cout << "\n";
  }

  return 0;
}
```

Sorting by Edges

```
struct Edge{
    int u;
    int v;
    int weight;
};

bool sorter(Edge i , Edge j){
    return i.weight > j.weight;
}
```

Sets of Pairs

```


int main(){
    set <pair<int,int>> setter;
    setter.insert({1,0});
    setter.insert({1,2});
    setter.insert({2,5});
    setter.insert({10,4});
    //IMPORTANT: to search, use {search_val, -inf} for lower_bound

// {search_Val, inf} for upperbound
//this is since set is default sorted like (1,0) -> (1,1)-> (2,0), etc
//so you can kind of implicitly see that the condition of +-inf is required
//to fit the normal function of lower and upperbound, lest the function gets
//messed up
    auto iter = setter.lower_bound({9, -1e9});
    pair <int,int> value = *iter;
    if(iter != setter.end()) printf("PL %d %d\n", value.first, value.second);

}

//note that if the value is < or = -> then it outputs > or = respectively
```

Unordered vs Ordered maps (Counting stuff)

```
unordered_map<int,int> hashings;

int most_frequent(int start, int end){
    hashings.clear();
    for(int i =start; i<=end;i++)hashings[prefix_sum[i]]++;

    int max_count = 0,res=-1;
    for(auto i: hashings){ // didnt know you could do this lol
        if(max_count < i.second){
            res = i.first;
            max_count = i.second;
        }
    }
    return max_count;
}
```

Use std::map when

- You need ordered data.
- You would have to print/access the data (in sorted order).
- You need predecessor/successor of elements.
- See advantages of BST over Hash Table for more cases.

Use std::unordered_map when

- You need to keep count of some data (Example – strings) and no ordering is required.
- You need single element access i.e. no traversal.

NOTE:

- Just accessing map[] EDITS the value to be zero (if declared outside)
- To ensure that this edit does NOT happen, you have to use map.count(), to see if it exist or not. (Vitamins Codeforces)

Notes/Tricks/Utility/Etc

```
inline int readInt() {
    int x = 0;
    char ch = getchar();
    while (ch < '0' || ch > '9') ch = getchar_unlocked();
    while (ch >= '0' && ch <= '9'){
    x = (x << 3) + (x << 1) + ch - '0';
    ch = getchar_unlocked();
  }
    return x;
}
```

```
//use {} instead of make_pair.

vector<int> ans(n, -1);
//creates vector of size n, all values -1

assert()
//If the argument expression of this macro with functional
//form compares equal to zero (i.e., the expression is false),
//a message is written to the standard error device and abort is called,
//terminating the program execution.

bitset<n> biter;
memset(dis, 0x3F , sizeof(dis)); -> for 'inf'
priority_queue <pii, vector<pii>, greater<pii>>pq;
//this is MINIMUM priority_queue, ie the pq.top() gives the LOWEST value.
//Yes, you need to use greater<pii>

//Endl is slower than \n
//cin is slower than scanf

__gcd(value1, value2)

move()
When you work with STL containers like `vector`,
you can use `move` function to just move container, not to copy it all.


//for bit
int main(){
  cin >> n >> num;
  for(int i =n;i>=0;i--){//note that it is i=n, i>=0, i--, not i++
    if((num >> i) & 1) cout << "1";
    else cout << "0";
  }
}
8 -> 0000001010
```

Inspiration from History

- N nodes, N-1 edges, all nodes connected. Can only be a tree!-
- Never EVER let the array hit arr[-1]. ALWAYS ALWAYS have a condition to prevent that - or else something like ‘stamp’ might happen… ie unpredictable behavior
- When doing DP questions, really really define out the dp[i] or whatever. Like write it out. Like: dp[i][j] is the minimum number of toys of type 1,2,…i such that the cost adds up to less than j, repeats allowed.
- Grab the fucking subtasks!!!
- It is good to think about wrong solutions and WHY they are wrong, not just think of correct solutions and why they are correct. This allows you to better form relations.
- (Barcode) Some DP questions, you cannot separate two processes - it must be done at once. In the same for loop, not separated
- (Chocolate) For some DP questions, you can use divide and conquer ie breaking down the problem into smaller subsections of itself, directly. And then returning with dp array. Usually, the complexity is pretty high (I think?)
- (NIS) NEGATIVE MODULUS WILL KILL YOU IF YOU ARE NOT CAREFUL!!!
    - modulus can be negative. but sometimes the interaction between updates and modulus requires it to be positive
    - so it MIGHT MIGHT just be mod + (negative modulus) ie (10^9 + 7 - something).
- (4Russian) (Barcode) Turns out that in DP, especially for questions that want a ‘range’ somewhere (like select range, don’t select range, etc), having a dp with an additional [0] or [1] for ending/starting range can LITERALLY be the answer.

New

Bitwise Tricks

```
#include <bits/stdc++.h>
using namespace std;


int main() {
  int a = 123456;//11110001001000000
  long long b = 123456789012;
  //1110010111110100110010001101000010100

  //One plus the index of the least significant 1-bit of x.
//If x is zero, returns zero
printf("%d\n", __builtin_ffs(a));

  //The number of leading 0-bits in x,
  //starting at the most significant bit.
  printf("%d\n", __builtin_clz(a)); //x = 0 is undefined
//00000000000000011110001001000000 -> 15 0s in front
```

```
  //The number of trailing 0-bits in x, starting at the least significant bit position.
  printf("%d\n", __builtin_ctz(a)); //x = 0 is undefined
//6 0s at the back



  //31 minus the position of the most significant bit.
//0 -> 31, 1 -> 30, 2 -> 29.
//1 at position 17 -> 31-17=14
  printf("%d\n", __builtin_clrsb(a));



  //The number of 1-bits in (unsigned int) x.
  printf("%d\n", __builtin_popcount(a));
  printf("%d\n", __builtin_popcountll(b));
  //Similar to __builtin_popcount, except the argument type is unsigned long long.



  //The parity of the number of 1-bits in x (odd = 1, even = 0)
  //a has 6 1s, so 0
  printf("%d\n", __builtin_parity(a));



  //Returns x with the order of the bytes reversed;
//for example, 0xaabb becomes 0xbbaa.
  //Byte here always means exactly 8 bits.
  printf("%d\n", __builtin_bswap32(a));
/**
Original number: 0x12345678
Swapped number:  0x78563412
**/

}
//dun know -> __builtin_parity , __builtin_bswap32
```

2D Fenwick Tree Point Update

```
#include <bits/stdc++.h>

using namespace std;

int bit[1025][1025];

//Update function for cell (x, y)

int S=1025;

void update(int x, int y, int val){//point update btw
    int y1;
    while(x <= S){
        y1 = y;
        while(y1 <= S){
            bit[x][y1] += val;
            y1 += (y1 & -y1);
        }
        x += (x & -x);
    }
}

//Query cumulative frequency from (1, 1) to (x, y)
int query(int x, int y){
    int sum = 0;
    while(x > 0){
        int y1 = y;
        while(y1 > 0){
            sum += bit[x][y1];
            y1 -= (y1 & -y1);
        }
        x -= (x & -x);
    }

    return sum;
}

//Overloaded function to query cumulative frequency from (x1, y1) to (x2, y2)
int query_total(int x1, int y1, int x2, int y2){
    return query(x2, y2) - query(x2, y1 - 1) - query(x1 - 1, y2) + query(x1 - 1, y1 - 1);
}
//1 index, cannot put 0 as input for position

int main(){
  int increment,query;
  cin >>increment >> query;
  for(int i =0;i<increment;i++){
    int a,b,val;
    cin >> a >> b >> val;
    update(a,b,val);
  }
  for(int i =0;i<query;i++){
    int x1,y1,x2,y2;
    cin >> x1 >> y1 >> x2 >> y2;
    int ans = query_total(x1,y1,x2,y2);
    printf("(%d %d %d %d) ANS: %d\n", x1,y1,x2,y2,ans);
  }
}
```

2D Fenwick with Range query and update (Replaced 2d segment tree)

```
#include<bits/stdc++.h>
using namespace std;

const int N = 1010;

struct BIT2D {
  long long M[N][N][2], A[N][N][2];
  BIT2D() {
    memset(M, 0, sizeof M);
    memset(A, 0, sizeof A);
  }
  void upd2(long long t[N][N][2], int x, int y, long long mul, long long add) {
    for(int i = x; i < N; i += i & -i) {
      for(int j = y; j < N; j += j & -j) {
        t[i][j][0] += mul;
        t[i][j][1] += add;
      }
    }
  }
  void upd1(int x, int y1, int y2, long long mul, long long add) {
    upd2(M, x, y1, mul, -mul * (y1 - 1));
    upd2(M, x, y2, -mul, mul * y2);
    upd2(A, x, y1, add, -add * (y1 - 1));
    upd2(A, x, y2, -add, add * y2);
  }
  void upd(int x1, int y1, int x2, int y2, long long val) {
    upd1(x1, y1, y2, val, -val * (x1 - 1));
    upd1(x2, y1, y2, -val, val * x2);
  }
  long long query2(long long t[N][N][2], int x, int y) {
    long long mul = 0, add = 0;
    for(int i = y; i > 0; i -= i & -i) {
      mul += t[x][i][0];
      add += t[x][i][1];
    }
    return mul * y + add;
  }
  long long query1(int x, int y) {
    long long mul = 0, add = 0;
    for(int i = x; i > 0; i -= i & -i) {
      mul += query2(M, i, y);
      add += query2(A, i, y);
    }
    return mul * x + add;
  }
  long long query(int x1, int y1, int x2, int y2) {
    return query1(x2, y2) - query1(x1 - 1, y2) - query1(x2, y1 - 1) + query1(x1 - 1, y1 - 1);
  }
} t;
int main() {
  int n, m;
  cin >> n >> m;
  for(int i = 1; i <= n; i++) {
    for(int j = 1; j <= m; j++) {
      int k;
      cin >> k;
      t.upd(i, j, i, j, k);
    }
  }

  //val is incremental increase/decrease
  int q;
  cin >> q;
  while(q--) {
    int ty, x1, y1, x2, y2;
    cin >> ty;
    if(ty == 1) {
      long long val;
      cin >> x1 >> y1 >> x2 >> y2 >> val;
      t.upd(x1, y1, x2, y2, val); // add val from top-left(x1, y1) to bottom-right (x2, y2);
    } else {
      cin >> x1 >> y1 >> x2 >> y2;
      cout << t.query(x1, y1, x2, y2) << '\n'; // output sum from top-left(x1, y1) to bottom-right (x2, y2);

    }
  }
  return 0;
}
```