Team Name: SEC_ZYX

Institution: Sylhet Engineering College

Table of content

Binary Search

```
// return index number or -1
int binarySearch (vector<int> &nums,
int target) {
   int low = 0, high = nums.size() - 1,
mid;
   while (low <= high)
   {
      mid = (high + low) >> 1;
      if (nums[mid] == target)
      {
         return mid;
      }
      else if (nums[mid] < target)
      {
         low = mid + 1;
      }
      else
      {
         high = mid - 1;
      }
   } return -1; // return high;
}

// binary search in a rotated sorted
array(vector<int>)
bool searchInARotatedSortedArray
(vector<int>&A, int key) {
   int l=0, h=A.size()-1, m;
   while (l<=h)
   {
      m = (l+h)>>1;
      if(A[m]==key){
         return true;
      }if(A[m]==A[l] &&
A[m]==A[h]){
         ++l,--h;
         continue;
      }else if(A[l] <= A[m]){
         if(A[l]<=key && key<=A[m]){
            h = m-1;
         }else{
            l = m+1;
         }
      }else{
         if(A[m]<=key &&
key<=A[h]){
            l = m+1;
         }else{
            h = m-1;
         }
      }
   } return false;
}


// NthRoot(2.0, 25.0)
double midPowerOfN (double mid, int
n, double m)
{
   double ans = 1.0;
   for (int i = 1; i <= n; ++i)
   {
      ans *= mid;
      if (ans > m)
         return 2.0;
   }
   if (fabs(ans - m) < 1e-9)
      return 1.0;
   return 0.0;
}
double NthRoot (int n, double m)
{
   double l = 0.0, h = m, mid;
   double tolerance = 1e-9;
   double closest = -1.0;
   // while (h - l > tolerance)
   // for safety 100 is good
   for (int i = 0; i <= 100; ++i)
   {
      mid = (l + h) / 2.0;
      double ans = midPowerOfN(mid,
n, m);
      if (ans == 1.0)
      {
         return mid;
      }
      else if (ans == 0.0)
      {
         l = mid;
      }
      else
      {
         h = mid;
         closest = mid;
      }
   }
   return closest;
}


// Binary Exponentiation
int mod = 1e9+7;
int binaryExponentiation (int base,
int power){
   int carry = 1%mod;
   int x = base%mod;
   while(power){
      if(power&1) carry = (__int128_t)
(carry*x)%mod;
      x = (__int128_t) (x*x)%mod;
      power = power>>1;
   } return carry%mod;
}

// sum of digit
int sumOfDigit (int num){
   ll sum = 0;
   while(num>0){
      sum += num%10;
      num/=10;
   }return sum;
}

//divisors for a number
vector<int> divisorsForSingleValue
(int n){
   int i;
   vector<int> v1;
   for(i=1; i*i<=n; ++i){
      if(n%i==0){
         v1.push_back(i);
         v1.push_back(n/i);
      }
   }--i;
   if(i*i==n){
      v1.pop_back();
   }
   return v1;
}

// number of combination
int nCr(int n, int r)
{
   int i, ans = 1;
   for (i = 2; i <= n; ++i)
   {
      ans *= i;
      if (i <= r || i <= (n - r))
         ans /= i;
   }
   if (r < (n - r))
   {
      for (i = 2; i <= r; ++i)
      {
         ans /= i;
      }
   }
   else
   {
      for (i = 2; i <= n - r; ++i)
      {
         ans /= i;
      }
   } return ans;
}


// number of permutation
int nPr (int n, int r)
{
   int i, ans = 1;
   for (i = (n - r) + 1; i <= n; ++i)
   {
      ans *= i;
   } return ans;
}


// Prime factorization impl
void primeFactorization (int n,
map<int, int> &m1)
{
   for (int i = 2; i * i <= n; ++i)
   {
      while (n % i == 0)
      {
         m1[i]++;
         n /= i;
      }
   }
   if (n > 1)
      m1[n]++;
}
```

```
// Sieve Prime (n log log n)
const int N = 1e6 + 9;
bool isPrime[N];
vector<int> primeNumbers;
void sieve()
{
   int i, j;
   isPrime[0] = isPrime[1] = true;
   for (i = 2; i < N; ++i)
   {
      if (!isPrime[i])
      {
         primeNumbers.push_back(i);
         for (j = i * i; j < N; j += i)
         {
            isPrime[j] = true;
         }
      }
   }
}


// sum of divisors of a number
const int N = 1e5;
vector<ll> primeNumbers;
vector<bool> isPrime(N+1, 0);
const int mod = 1e9+7;
// bigmod function
ll bigmod(ll n, ll k){
   if(k==0) return 1LL;
   ll x = bigmod(n, k/2)%mod;
   x = (x*x)%mod;
   if(k%2==1) x = (x*n)%mod;
   return x;
}
// Sieve Prime
void sieve(){
   isPrime[0] = isPrime[1] = true;
   for(ll i=2; i<N; ++i){
      if(!isPrime[i]){
         primeNumbers.push_back(i);
         for(ll j=i*i; j<N; j+=i){
            isPrime[j] = true;
         }
      }
   }
}
// sum of divisor of a number
ll sumOfDivisors(ll n){
   ll cnt, sum=1LL;
   for(ll &num: primeNumbers){
      cnt=0;
      if(num*num>n) break;
      while(n%num==0){
         n/=num;
         ++cnt;
      }
      // sum = (sum*((bigmod(num,
cnt)-1LL)/(num-1LL)))%mod;
      if(cnt){
         ll a =
(bigmod(num,cnt+1)-1LL+mod)%mod
;
         ll b =
(bigmod((num-1LL),mod-2LL));
         sum =
(sum*((a*b)%mod))%mod;
      }
   }
   if(n>1){
      ll a =
(bigmod(n,2)-1LL+mod)%mod;
      ll b =
(bigmod((n-1LL),mod-2LL));
      sum = (sum*((a*b)%mod))%mod;
   }
   return sum;
}

// Sieve Prime O(n)
vector<bool> primeSieve (int N) {
   vector<bool> isPrimeNumber(N,
true);
   isPrimeNumber[0] =
isPrimeNumber[1] = false;
   for (int i = 4; i < N; i += 2)
      isPrimeNumber[i] = false;
   for (int i = 9; i < N; i += 6)
      isPrimeNumber[i] = false;
   for (int i = 1; 6 * i < N; ++i) {
      int x = 6 * i - 1, y = 6 * i + 1;
      if (isPrimeNumber[x])
         for (int j = 3 * x; j < N; j = x +
x + j)
            isPrimeNumber[j] = false;
      if (isPrimeNumber[y])
         for (int j = 3 * y; j < N; j = y +
y + j)
            isPrimeNumber[j] = false;
   }
   return isPrimeNumber;
}


// Generating subset of a set [1, n]
using bit mask
// TC(2^n * n) n-> 20-25
vector<vector<int>> subsets
(vector<int> &numbers_)
{
   int n = numbers_.size(), mask, i;
   int subset_cnt = (1 << n);
   vector<vector<int>> all_subsets;
   for (mask = 0; mask < subset_cnt;
++mask)
   {
      vector<int> subset;
      for (i = 0; i < n; ++i)
      {
         if ((mask & (1 << i)) != 0)
         {

subset.push_back(numbers_[i]);
         }
      }
      all_subsets.push_back(subset);
   }
   return all_subsets;
}

// find out n is prime or not
bool isPrime (int n)
{
   if (n <= 1)
      return false;
   else if (n <= 3)
      return true;
   else if (n % 2 == 0 || n % 3 == 0)
      return false;
   for (int i = 5; i * i <= n; i += 6)
   {
      if (n % i == 0 || n % (i + 2) == 0)
         return false;
   }
   return true;
}

// findingPrimeFactors();
// Finding All Prime Factors From 1 to
N
const int N = 2020;
vector<ll> primeFactors[N];
int primeNumbers[N];
bool mp[N];
bool isPrime(ll n)
{
   if (n <= 1)
      return false;
   else if (n <= 3)
      return true;
   else if (n % 2 == 0 || n % 3 == 0)
      return false;
   for (int i = 5; i * i <= n; i += 6)
   {
      if (n % i == 0 || n % (i + 2) == 0)
         return false;
   }
   return true;
}
void findingPrimeFactors()
{
   int i,j;
   mp[2] = true;
   mp[3] = true;
   for (i = 1; 6 * i - 1 <= 1000; ++i)
   {
      ll a = (6 * i - 1), b = (6 * i + 1);
      if (isPrime(a))
         mp[a] = true;
      if (isPrime(b))
         mp[b] = true;
   }
   for (i = 1; i <= 1000; ++i)
   {
      if (mp[i])
      {
         for (j = i + i; j <= 1000; j += i)
         {

primeFactors[j].push_back(i);
         }
      }
   }
}

int timer = 1;
```

```
// vector<vector<int>> bridges =
criticalConnections(numberOfNodes,
connections);
// dfs(0, -1, vis, adj, tin, low, bridges);
// Bridges In Graph – Using Tarjan's
Algorithm of Time In and Low Time
void dfs(int node, int parent,
vector<int> &vis, vector<int> adj[], int
tin[], int low[], vector<vector<int>>
&bridges)
{
   vis[node] = 1;
   tin[node] = low[node] = timer;
   timer++;
   for (auto &it : adj[node])
   {
      if (it == parent)
         continue;
      if (vis[it] == 0)
      {
         dfs(it, node, vis, adj, tin, low,
bridges);
         low[node] = min(low[it],
low[node]);
         if (low[it] > tin[node])
         {
            bridges.push_back({it,
node});
         }
      }
      else
      {
         low[node] = min(low[node],
low[it]);
      }
   }
}
vector<vector<int>>
criticalConnections (int n,
vector<vector<int>> &connections)
{
   vector<int> adj[n];
   for (auto it : connections)
   {
      int u = it[0], v = it[1];
      adj[u].push_back(v);
      adj[v].push_back(u);
   }
   vector<int> vis(n, 0);
   int tin[n];
   int low[n];
   vector<vector<int>> bridges;
   dfs(0, -1, vis, adj, tin, low, bridges);
   return bridges;
}


// Function to find the shortest distance
of all the vertices
// from the source vertex S.
// vector<int> res = dijkstra(V, adj,
Source);
vector<int> dijkstra (int V,
vector<vector<int>> adj[], int S)
{
   // Create a priority queue for storing
the nodes as a pair {dist,node}
   // where dist is the distance from
source to the node.
   priority_queue<pair<int, int>,
vector<pair<int, int>>,
greater<pair<int, int>>> pq;

   // Initialising distTo list with a large
number to
   // indicate the nodes are unvisited
initially.
   // This list contains distance from
source to the nodes.
   vector<int> distTo(V, INT_MAX);

   // Source initialised with dist=0.
   distTo[S] = 0;
   pq.push({0, S});

   // Now, pop the minimum distance
node first from the min-heap
   // and traverse for all its adjacent
nodes.
   while (!pq.empty())
   {
      int node = pq.top().second;
      int dis = pq.top().first;
      pq.pop();

      // Check for all adjacent nodes of
the popped out
      // element whether the prev dist is
larger than current or not.
      for (auto it : adj[node])
      {
         int v = it[0];
         int w = it[1];
         if (dis + w < distTo[v])
         {
            distTo[v] = dis + w;

            // If current distance is
smaller,
            // push it into the queue.
            pq.push({dis + w, v});
         }
      }
   }
   // Return the list containing shortest
distances
   // from source to all the nodes.
   return distTo;
}


// Topological Sort Using BFS
Simple02
// Kahn's Algorithm
// vector<int> ans = topoSort(V, adj);
// Function to return list containing
vertices in Topological order.
vector<int> topoSort(int V,
vector<vector<int>> &adj)
{
   int indegree[V] = {0};
   for (int i = 0; i < V; i++)
   {
      for (auto it : adj[i])
      {
         indegree[it]++;
      }
   }
   queue<int> q;
   for (int i = 0; i < V; i++)
   {
      if (indegree[i] == 0)
      {
         q.push(i);
      }
   }
   vector<int> topo;
   while (!q.empty())
   {
      int node = q.front();
      q.pop();
      topo.push_back(node);
      // node is in your topo sort
      // so please remove it from the
indegree

      for (auto it : adj[node])
      {
         indegree[it]--;
         if (indegree[it] == 0)
            q.push(it);
      }
   }
   return topo;
}


// Topological Sort Using DFS
Simple01
// Function to return list containing
vertices in Topological order.
// call vector<int> ans = topoSort(V,
adj);
void dfs(int node, int vis[], stack<int>
&st, vector<vector<int>> &adj)
{
   vis[node] = 1;
   for (auto it : adj[node])
   {
      if (!vis[it])
         dfs(it, vis, st, adj);
   }
   st.push(node);
}
vector<int> topoSort (int V,
vector<vector<int>> &adj)
{
   int vis[V] = {0};
   stack<int> st;
   for (int i = 0; i < V; i++)
   {
      if (!vis[i])
      {
         dfs(i, vis, st, adj);
      }
   }
```

```
    vector<int> ans;
    while (!st.empty())
    {
        ans.push_back(st.top());
        st.pop();
    }
    return ans;
}


// Multisource BFS For Adjacency List
int dist[N];
bool visited[N];
// Multisource BFS Function
void Multisource_BFS(vector<int>
graph[], queue<int> q)
{
    while (!q.empty())
    {
        int k = q.front();
        q.pop();
        for (auto i : graph[k])
        {
            if (!visited[i])
            {
                // Pushing the adjacent
unvisited vertices
                // with distance from current
source = this
                // vertex's distance + 1
                q.push(i);
                dist[i] = dist[k] + 1;
                visited[i] = true;
            }
        }
    }
}
// This function calculates the distance
of each
// vertex from nearest source
void nearestTown(vector<int>
graph[], int n, int sources[], int s)
{
    // Create a queue for BFS
    queue<int> q;
    // Mark all the source vertices as
visited and enqueue it
    for (int i = 0; i < s; i++)
    {
        q.push(sources[i]);
        visited[sources[i]] = true;
    }
    Multisource_BFS(graph, q);
    // Printing the distances
    for (int i = 1; i <= n; i++)
    {
        cout << i << " " << dist[i] <<
endl;
    }
}

// Multisource BFS For Adjacency
Matrix
int multisourceBFS
(vector<vector<int>> &grid)
{
    queue<pair<int, int>> q;
    int n = grid.size();    // 6
    int m = grid[0].size(); // 5
    int vis[n][m];
    memset(vis, 0, sizeof(vis));
    // traverse boundary elements
    for (int i = 0; i < n; i++)
    {
        // if it is a land then store it in
queue
        if (grid[i][0] == 1)
        {
            q.push({i, 0});
            vis[i][0] = 1;
        }
        if (grid[i][m - 1] == 1)
        {
            q.push({i, m - 1});
            vis[i][m - 1] = 1;
        }
    }
    for (int i = 1; i < m - 1; i++)
    {
        // if it is a land then store it in
queue
        if (grid[0][i] == 1)
        {
            q.push({0, i});
            vis[0][i] = 1;
        }
        if (grid[n - 1][i] == 1)
        {
            q.push({n - 1, i});
            vis[n - 1][i] = 1;
        }
    }

    int delrow[] = {-1, 0, +1, 0};
    int delcol[] = {0, +1, +0, -1};

    while (!q.empty())
    {
        int row = q.front().first;
        int col = q.front().second;
        q.pop();

        // traverses all 4 directions
        for (int i = 0; i < 4; i++)
        {
            int nrow = row + delrow[i];
            int ncol = col + delcol[i];
            // check for valid coordinates
and for land cell
            if (nrow >= 0 && nrow < n &&
ncol >= 0 && ncol < m &&
vis[nrow][ncol] == 0 &&
grid[nrow][ncol] == 1)
            {
                q.push({nrow, ncol});
                vis[nrow][ncol] = 1;
            }
        }
    }

    int cnt = 0;
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < m; j++)
        {
            // check for unvisited land cell
            if (grid[i][j] == 1 && vis[i][j]
== 0)
                cnt++;
        }
    }
    return cnt;
}


// Bipartite Graph or Bicoloring
bool check(int start, int V, vector<int>
adj[], int color[])
{
    queue<int> q;
    q.push(start);
    color[start] = 0;
    while (!q.empty())
    {
        int node = q.front();
        q.pop();

        for (auto it : adj[node])
        {
            // if the adjacent node is yet not
colored
            // you will give the opposite
color of the node
            if (color[it] == -1)
            {
                color[it] = !color[node];
                q.push(it);
            }
            // is the adjacent guy having the
same color
            // someone did color it on some
other path
            else if (color[it] ==
color[node])
            {
                return false;
            }
        }
    }
    return true;
}
bool isBipartite(int V, vector<int>
adj[])
{
    int color[V];
    for (int i = 0; i < V; i++)
        color[i] = -1;

    for (int i = 0; i < V; i++)
    {
        // if not colored
        if (color[i] == -1)
        {
            if (check(i, V, adj, color) ==
false)
            {
                return false; // it is not
bipartite graph
```

```
        }
      }
    }
    return true; // it is bipartite graph
}


// Detect Cycle In A Undirected Graph
Using BFS
bool detect(int src, vector<int> adj[],
bool vis[])
{
    vis[src] = true;
    // store <source node, parent node>
    queue<pair<int, int>> q;
    q.push({src, -1});
    // traverse until queue is not empty
    while (!q.empty())
    {
        int node = q.front().first;
        int parent = q.front().second;
        q.pop();

        // go to all adjacent nodes
        for (auto adjacentNode :
adj[node])
        {
            // if adjacent node is unvisited
            if (!vis[adjacentNode])
            {
                vis[adjacentNode] = true;
                q.push({adjacentNode,
node});
            }
            // if adjacent node is visited and
is not it's own parent node
            else if (parent != adjacentNode)
            {
                // yes it is a cycle
                return true;
            }
        }
    }
    // there's no cycle
    return false;
}
// Function to detect cycle in an
undirected graph.
bool isCycle(int V, vector<int> adj[])
{
    // initialize them as unvisited
    bool vis[V] = {false};
    for (int i = 0; i < V; i++)
    {
        if (!vis[i])
        {
            if (detect(i, adj, vis))
                return true;
        }
    }
    return false;
}


// detect cycle in a directed graph
// Function to perform DFS and check
for cycles in the directed graph
bool dfsCheck(int node, vector<int>
adj[], bool vis[], bool pathVis[])
{
    vis[node] = true;
    pathVis[node] = true;

    // Traverse adjacent nodes
    for (auto it : adj[node])
    {
        // When the node is not visited
        if (!vis[it])
        {
            if (dfsCheck(it, adj, vis,
pathVis))
                return true;
        }
        // If the node has been previously
visited but it has to be visited on the
same path
        else if (pathVis[it])
        {
            return true; // there is a cycle
        }
    }

    pathVis[node] = false;
    return false; // if there is no cycle
}
// Function to detect cycle in a directed
graph
bool isCyclic(int V, vector<int> adj[])
{
    bool vis[V] = {false};     // Array to
track visited nodes
    bool pathVis[V] = {false}; // Array
to track visited nodes on the current
path

    for (int i = 0; i < V; i++)
    {
        if (!vis[i])
        {
            if (dfsCheck(i, adj, vis,
pathVis))
                return true;
        }
    }
    return false;
}


// Function to return Breadth First
Traversal of given graph.
vector<int> bfsOfGraph(int V,
vector<int> adj[])
{
    bool vis[V] = {false};
    vis[0] = true; // here we define
source node true
    queue<int> q;
    // push the initial starting node
    q.push(0);
    vector<int> bfs;
    // iterate till the queue is empty
    while (!q.empty())
    {
        // get the topmost element in the
queue
        int node = q.front();
        q.pop();
        bfs.push_back(node);
        // traverse for all its neighbors
        for (auto it : adj[node])
        {
            // if the neighbor has previously
not been visited,
            // store in Queue and mark as
visited
            if (!vis[it])
            {
                vis[it] = true;
                q.push(it);
            }
        }
    }
    return bfs;
}


// Trie
struct Trie
{
    struct Trie *children[10];
    bool isend;
    Trie(){
        memset(children,
0,sizeof(children));
        isend = false;
    }
};
struct Trie *root;
void insertTrie (string s1){
    struct Trie *cur = root;
    for(char &c: s1){
        int index = c-'0';
        if(cur->isend) flag = false;
        if(!cur->children[index]){
            cur->children[index] = new
Trie;
        }
        cur = cur->children[index];
    }
    cur->isend = true;
    for(int i=0; i<10; ++i){
        if(cur->children[i]){
            flag = false;
        }
    }
}


void dfs(ll vertex)
{
  visit[vertex] = true;
  for (auto child : graph[vertex])
  {
    if (!visit[child])
      dfs(child);
  }
}
```

```
// segment Tree
const int N = 2e5 + 2;
long long segmentTreeArray[N], tree[4
* N];
// segment Tree Array = given array
// tree = we build the array
// to call this function build(1, 0, n-1)
// node can't start from 0
void build(int node, int start, int end_)
{
   if (start == end_)
   {
      tree[node] =
segmentTreeArray[start];
      return;
   }

   int mid = (start + end_) >> 1;
   build((node << 1), start, mid);
   build((node << 1) + 1, mid + 1,
end_);

   // modify
   tree[node] = tree[(node << 1)] +
tree[(node << 1) + 1];
   // tree[node] = max(tree[(node<<1)],
tree[(node<<1)+1]);
   // tree[node] = min(tree[(node<<1)],
tree[(node<<1)+1]);
}

// to find sum from l to r
// query(1, 0, n - 1, l-1, r-1);
long long query(int node, int start, int
en, const int &l, const int &r)
{
   // l...r > start...en    start...en < l...r no
overlap
   if (start > r || en < l)
   {
      return 0LL;
   }
   // l start...en r complete overlap
   if (l <= start && r >= en)
   {
      return tree[node];
   }
   // start  l   en   r partial overlap
   // l  start r  en
   int mid = (start + en) >> 1;
   long long q1 = query(node << 1,
start, mid, l, r);
   long long q2 = query((node << 1) +
1, mid + 1, en, l, r);

   return q1 + q2;
   // return max(q1, q2);
   // return min(q1, q2);
}

// to update query
// update(1, 0, n - 1, index-1, val);
void update(int node, int st, int en,
const int &idx, const long long &val)
{
   if (st == en)
   {
      segmentTreeArray[st] = val;
      tree[node] = val;
      return;
   }
   int mid = (st + en) >> 1;
   if (idx <= mid)
   {
      update((node << 1), st, mid, idx,
val);
   }
   else
   {
      update((node << 1) + 1, mid + 1,
en, idx, val);
   }

   tree[node] = tree[(node << 1)] +
tree[(node << 1) + 1];
   // tree[node] = max(tree[(node<<1)],
tree[(node<<1)+1]);
   // tree[node] = min(tree[(node<<1)],
tree[(node<<1)+1]);
}


vector<long long>
maximumSumSubmatrix (int r, int c,
vector<vector<long long>> &v1)
{
   long long **prefix = new long long
*[r];
   for (int i = 0; i < r; ++i)
   {
      prefix[i] = new long long;
      for (int j = 0; j < c; ++j)
      {
         prefix[i][j] = 0;
      }
   }
   for (int i = 0; i < r; ++i)
   {
      for (int j = 0; j < c; ++j)
      {
         if (j == 0)
         {
            prefix[i][j] = v1[i][j];
         }
         else
         {
            prefix[i][j] = v1[i][j] +
prefix[i][j - 1];
         }
      }
   }
   vector<long long> solution(5);
   long long val, st, en, ans;
   // long long maxSum =
LLONG_MIN, startx, starty, endx,
endy;
   // solution[0,...,4] =
{maxSum,start_x,start_y,end_x,end_y}
   solution[0] = LLONG_MIN;
   for (int i = 0; i < c; ++i)
   {
      for (int j = i; j < c; ++j)
      {
         vector<long long> v2;
         for (int k = 0; k < r; ++k)
         {
            if (i == 0)
            {
               val = prefix[k][j];
            }
            else
            {
               val = prefix[k][j] -
prefix[k][i - 1];
            }
            v2.push_back(val);
         }
         ans = kadaneAlorithm(v2, st,
en);
         if (ans > solution[0])
         {
            solution[0] = ans, solution[1]
= st, solution[2] = i, solution[3] = en,
solution[4] = j;
         }
      }
   }
   return solution;
}


long long kadaneAlorithm
(vector<long long> &v1, long long
&finalStart, long long &finalEnd)
{
   long long currentSum = 0, maxSum
= LLONG_MIN, n = v1.size(), start =
0;
   for (int i = 0; i < n; ++i)
   {
      currentSum += v1[i];
      if (currentSum > maxSum)
      {
         maxSum = currentSum;
         finalStart = start;
         finalEnd = i;
      }
      if (currentSum < 0)
      {
         currentSum = 0;
         start = i + 1;
      }
   }
   return maxSum;
}


long long
LargestSumContiguousSubarray (ll
ar[], ll arraySize, ll &subarrayStart, ll
&subarrayEnd){
   long long sum=0, mx =
LLONG_MIN, start=-1;
   for(int i=0; i<arraySize; ++i){
      if(sum==0){
         start = i;
```

```
    }
    sum += ar[i];
    if(mx<sum){
      mx = sum;
      subarrayStart = start;
      subarrayEnd = i;
    }
    if(sum<0){
      sum=0;
    }
  }
  return mx;
}


vector<long long>
jobSequencingProblem
(vector<vector<long long>> &v1, int
n)
  {
    sort(v1.begin(), v1.end(),
[&](vector<long long> &a,
vector<long long> &b){
      return a[1]<b[1];
    });
    priority_queue<vector<long
long>, vector<vector<long long>>,
CustomComparator> pq;
    long long mxProfit = 0,
numberOfWorkDone = 0;
    int slot_available;
    for (int i = n-1; i >= 0; --i)
    {
      slot_available = v1[i][1] -
(i>0?v1[i - 1][1]:0);
      pq.push(v1[i]);
      while (slot_available>0 &&
pq.size()>0)
      {
        vector<long long> v2 =
pq.top();
        pq.pop();
        mxProfit += v2[2];
        numberOfWorkDone++;
        slot_available--;
      }
    }
    return {numberOfWorkDone,
mxProfit};
  }


vector<int> jobSequencingProblem
(vector<vector<int>> &v1, int n)
{
  // v1 is vector first index -> id
  // v1 is vector second index -> dead
Line
  // v1 is vector profit index -> profit
  sort(v1.begin(), v1.end(),
[&](vector<int> &a, vector<int> &b)
    { return a[2] > b[2]; });
  int mx_dead = -1;
  for (int i = 0; i < n; ++i)
  {
    if (mx_dead < v1[i][1])
      mx_dead = v1[i][1];
  }
  vector<int> v2(mx_dead + 1, -1);
  int mxProfit = 0,
numberOfWorkDone = 0;
  for (int i = 0; i < n; ++i)
  {
    for (int j = v1[i][1]; j > 0; --j)
    {
      if (v2[j] == -1)
      {
        v2[j] = v1[i][0];
        ++numberOfWorkDone;
        mxProfit += v1[i][2];
        break;
      }
    }
  }
  return {numberOfWorkDone,
mxProfit};
}


const int N = 2 * 1e5 + 10;
int dp[N], arr[N];
// remember to memset(dp, -1,
sizeof(dp))
// index should be start from 0 to n
// loop(i,0,n) ans =
max(ans,DP_LCS(i));
// index = longest Subsequence 0 to
index
int DP_LCS (int int index)
{
  // base case
  // ---
  // if data already exists.
  if (dp[index] != -1)
    return dp[index];
  int length = 1;
  for (int i = 0; i < index; ++i)
  {
    if(arr[index]>arr[i])
      length = max(length,
DP_LCS(i) + 1);
  }
  // return and store dating
  return dp[index] = length;
}


const int N = 2 * 1e5 + 10;
long long dp[N], height[N];
// remember to memset(dp, -1,
sizeof(dp))
// ending point is where frog is now
// k is maximum length frog can jump
long long DP_Jumps (long long int
endingPoint, long long k)
{
  // base case
  if (endingPoint <= 0)
    return 0;
  // if data is already exist.
  if (dp[endingPoint] != -1)
    return dp[endingPoint];
  long long cost = 1e15;
  for (int i = 1; i <= k; ++i)
  {
    if(endingPoint-i>=0)
      cost = min(cost,
DP_Jumps(endingPoint - i, k) +
abs(height[endingPoint] -
height[endingPoint - i]));
  }
  // return and store dating
  return dp[endingPoint] = cost;
}


vector<int> divisorsForSingleValue
(int n){
  int i;
  vector<int> v1;
  for(i=1; i*i<=n; ++i){
    if(n%i==0){
      v1.push_back(i);
      v1.push_back(n/i);
    }
  }--i;
  if(i*i==n){
    v1.pop_back();
  }
  return v1;
}


void bfs_distance(ll root, vector<ll>
&distance)
{
  ll x;
  queue<ll> q;
  q.push(root);
  distance[root] = 0;
  visited.assign(N, false);
  while (!q.empty())
  {
    ll current = q.front();
    q.pop();
    visited[current] = true;
    for (auto child : graph[current])
    {
      if (!visited[child])
      {
        q.push(child);
        distance[child] =
distance[current] + 1;
      }
    }
  }
}

vector<vector<int>>
printMaxActivities
(vector<vector<int>> &v1, int n)
{
  //v1.push_back({index[i], start[i],
finish[i]});
  // index[]  array is at 0 index of v1
  // start[]  array is at 1 index of v1
  // finish[] array is at 2 index of v1
      int i=0, j;
```

```
   vector<vector<int>> ansVector;
   sort(v1.begin(), v1.end(),
[&](vector<int> &a, vector<int> &b){
      return a[2]<b[2];
   });
   ansVector.push_back({v1[i][0],
v1[i][1], v1[i][2]});
            for (j = 1; j < n; j++) {
                  if(v1[j][1] >=
v1[i][2]) {

ansVector.push_back({v1[j][0],
v1[j][1], v1[j][2]});
         i = j;
                  }
            }
   return ansVector;
}



int gcd (int m, int n){
  int r = 0, a, b;
  a = (m > n) ? m : n;
  b = (m < n) ? m : n;
  r = b;
  while (a % b != 0)
  {
    r = a % b;
    a = b;
    b = r;
  }
  return r;
}
int lcm (int m, int n){
  int a;
  a = (m > n) ? m : n;
  while (true)
  {
    if (a % m == 0 && a % n == 0)
      return a;
    ++a;
  }
}
long lcm(long x, long y)
{
  return (x * (y / gcd(x, y)));
}



// sieveDivisorsCount();
// for(int i=0; i<N; ++i){
// cout << i << " " << countDivisors[i]
<< '\n';}
const int N = 1e6 + 5;
int countDivisors[N + 5];
void sieveDivisorsCount()
{
   int i, j;
   for (i = 2; i < N; i++)
   {
      for (j = i; j < N; j += i)
      {
         countDivisors[j]++;
      }
   }
}



// Finding Sum Of Two Index Value Is
Equal Of Target Value
// A is sorted vector
// if there any two value with sum X
return 1 nor 0
int isPairSum_TwoPointer
(vector<int> &A, int X)
{
   int i = 0;          // represents first
pointer
   int j = A.size() - 1; // represents
second pointer
   while (i < j)
   {
      if (A[i] + A[j] == X)
         return 1; // If we find a pair
      // If sum of elements at current
      // pointers is less, we move
towards
      // higher values by doing i++
      else if (A[i] + A[j] < X)
         i++;
      // If sum of elements at current
      // pointers is more, we move
towards
      // lower values by doing j--
      else
         j--;
   }
   return 0;
}



// order set
#include
<ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template <class T>
using oset = tree<T, null_type,
less<T>, rb_tree_tag,
tree_order_statistics_node_update>;
// oset <ll> s;  --> Declare ordered set
// s.order_of_key(val)  --> index of
value val
//*(s.find_by_order(ind)) --> value at
index ind



// Multiplicative modulus
long long mulmod(long long a, long
long b, long long m) {
   long long res = 0;  // Initialize the
result

   a %= m;
   while (b > 0) {
      // If b is odd, add a to the result
      if (b % 2 == 1) {
         res = (res + a) % m;
      }
      // Double a and reduce b by half
      a = (a * 2) % m;
      b /= 2;
   }

   return res;
}



// Power value modulus
long long mod_pow(long long base,
long long exp, long long mod) {
   long long result = 1;
   while (exp > 0) {
      if (exp % 2 == 1) {
         result = (result * base) % mod;
      }
      base = (base * base) % mod;
      exp /= 2;
   }
   return result;
}



// Sieve of Erastones
void sieveOfEratosthenes()
{
   primes[1] = 1;
   for (int i = 2; i * i <= N; i++)
   {
      if (primes[i])
         continue;
      primes[i] = i;
      for (int j = i * i; j <= N; j += i)
      {
         if (!primes[j])
            primes[j] = i;
      }
   }
}



// Number of Divisors of a number
long long numberOfDivisors(long
long num) {
   long long total = 1;
   for (int i = 2; (long long)i * i <=
num; i++) {
      if (num % i == 0) {
         int e = 0;
         do {
            e++;
            num /= i;
         } while (num % i == 0);
         total *= e + 1;
      }
   }
   if (num > 1) {
      total *= 2;
   }
   return total;
}
```

```
// DSU TC (alpha * n)
const int N = 1e5 + 10;
vector<int> parent(N), sizeDSU(N),
rankDSU(N);
void disjointSet(int n)
{
   rankDSU.resize(n + 1);
   parent.resize(n + 1);
   sizeDSU.resize(n + 1);
   for (int i = 0; i <= n; i++)
   {
      parent[i] = i;
      sizeDSU[i] = 1;
      rankDSU[i] = 0;
   }
}
void makeDSU(int i)
{
   parent[i] = i;
   sizeDSU[i] = 1;
}
int findParent(int p)
{
   if (p == parent[p])
      return p;
   // Path Compression
   return parent[p] =
findParent(parent[p]);
}
void unionBySize(int a, int b)
{
   int parentOfa = findParent(a);
   int parentOfb = findParent(b);
   if (parentOfa != parentOfb)
   {
      // Union by size
      if (sizeDSU[parentOfa] <
sizeDSU[parentOfb])
         swap(parentOfa, parentOfb);
      parent[parentOfb] = parentOfa;
      sizeDSU[parentOfa] +=
sizeDSU[parentOfb];
   }
}
void unionByRank(int a, int b)
{
   int parentOfa = findParent(a);
   int parentOfb = findParent(b);
   if (parentOfa == parentOfb)
      return;
   if (rankDSU[parentOfa] <
rankDSU[parentOfb])
   {
      parent[parentOfa] = parentOfb;
   }
   else if (rankDSU[parentOfb] <
rankDSU[parentOfa])
   {
      parent[parentOfb] = parentOfa;
   }
   else
   {
      parent[parentOfb] = parentOfa;
      rankDSU[parentOfa]++;
   }
}
```

```
// auto v2 = kruskal_MST(v1, n);
int total_cost;
vector<array<int, 3>> kruskal_MST
(vector<vector<array<int, 2>>> &adj,
int noNode)
{
   vector<array<int, 3>> gra, mst;
   int u, v, w, n = adj.size(), i, j;
   disjointSet(n);
   for (i = 1; i < n; i++)
   {
      for (auto &[v, w] : adj[i])
      {
         gra.push_back({w, i, v});
      }
   }
   sort(gra.begin(), gra.end());
   total_cost = 0;
   for (auto &[w, u, v] : gra)
   {
      if (findParent(u) == findParent(v))
         continue;
      unionBySize(u, v);
      total_cost += w;
      mst.push_back({w, u, v});
   }
   return mst;
}
```

```
// Prim's Algorithm MST
// Function to find sum of weights of
edges of the Minimum Spanning Tree.
int spanningTree(int V,
vector<vector<int>> adj[])
{
   priority_queue<pair<int, int>,
vector<pair<int, int>>,
greater<pair<int, int>>> pq;
   vector<int> vis(V, 0);
   // {wt, node}
   pq.push({0, 0});
   int sum = 0;
   while (!pq.empty())
   {
      auto it = pq.top();
      pq.pop();
      int node = it.second;
      int wt = it.first;

      if (vis[node] == 1)
         continue;
      // add it to the mst
      vis[node] = 1;
      sum += wt;
      for (auto it : adj[node])
      {
         int adjNode = it[0];
         int edW = it[1];
         if (!vis[adjNode])
         {
            pq.push({edW, adjNode});
         }
      }
   }
   return sum;
}
```

```
// Maximum Sum Of Subarray
// maximumSumOfSubarray(v1)
int maximumSumOfSubarray (int
v1[])
{
   int sum2 = INT_MIN, i, sum1 = 0, n
= sizeof(v1);
   for (i = 0; i < n; ++i)
   {
      sum1 += v1[i];
      sum2 = max(sum2, sum1);
      if (sum1 < 0)
         sum1 = 0;
   }
   return sum2;
}
```

```
// Diameter Of Tree
// diameterOfTree(v1)
int mx = 1, node = 1;
void dfs(int i, int depth, bool vis[], int
dis[], vector<vector<int>> &adj)
{
   vis[i] = true;
   dis[i] = depth;
   if (depth > mx)
   {
      mx = depth;
      node = i;
   }
   for (auto &c : adj[i])
   {
      if (!vis[c])
         dfs(c, depth + 1, vis, dis, adj);
   }
}
int diameterOfTree
(vector<vector<int>> &adj)
{
   int n = adj.size();
   bool vis[n];
   int dis[n];
   memset(vis, false, sizeof(vis));
   memset(dis, 0, sizeof(dis));
   // dfs(root, 0)
   dfs(1, 0, vis, dis, adj);
   memset(vis, false, sizeof(vis));
   memset(dis, 0, sizeof(dis));
   dfs(node, 0, vis, dis, adj);
   return mx;
}
```

```
// Subtree Size
// subtreeSizeCall(v1)
int dfs(int i, bool vis[], vector<int>
&subtreeSize, vector<vector<int>>
&adj)
```

```
{
   vis[i] = true;
   int child = 1;
   for (auto &c : adj[i])
   {
      if (!vis[c])
         child += dfs(c, vis, subtreeSize,
adj);
   }
   return subtreeSize[i] = child;
}
vector<int>
subtreeSizeCall(vector<vector<int>>
&adj)
{
   int n = adj.size();
   bool vis[n];
   vector<int> subtreeSize(n);
   memset(vis, false, sizeof(vis));
   // dfs(root)
   n = dfs(0, vis, subtreeSize, adj);
   return subtreeSize;
}


/*
** Below code is the basic structure
of sparse table range query
implementation
*/
const int MAXN = 1e9; // immutable
array max size

const int K = 25; // K =
2^(log(MAXN)) that is the max size of
the immutable array

int N = 6; // the size of the array
int st[K+1][MAXN]; // sparse table
array<int, 6> arr; // the given array

int f(int a, int b) {
   // This function is used to elaborate
the min query/ max query/ sum query
   // or Other action regarding min max
of a range specified at sparsetable

   return 0;
}

void sparseTable() {
   copy(arr.begin(), arr.end(), st[0]);

   for (int i = 1; i <= K; i++)
      for (int j = 0; j + (1 << i) <= N;
j++)
         st[i][j] = f(st[i - 1][j], st[i - 1][j
+ (1 << (i - 1))]);
}

/*
** Below code is for range sum
query
*/
const int MAXN = 1e9; // immutable
array max size
const int K = 25; // K =
2^(log(MAXN)) that is the max size of
the immutable array
int N = 6; // the size of the array
long long st[K + 1][MAXN];
array<int, 6> arr; // the given array

void sparseTable() {
   copy(arr.begin(), arr.end(), st[0]);

   for (int i = 1; i <= K; i++)
      for (int j = 0; j + (1 << i) <= N;
j++)
         st[i][j] = st[i - 1][j] + st[i - 1][j
+ (1 << (i - 1))];
}

int getSum(int  L, int R) { // O(K)
long long sum = 0;
for (int i = K; i >= 0; i--) {
   if ((1 << i) <= R - L + 1) {
      sum += st[i][L];
      L += 1 << i;
   }
}
return sum;
}


/*
** Below code is for range min
query
*/

const int MAXN = 1e9; // immutable
array max size
const int K = 25; // K =
2^(log(MAXN)) that is the max size of
the immutable array
int N = 6; // the size of the array
long long st[K + 1][MAXN];
array<int, 6> arr; // the given array
int lg[MAXN+1];

void precomputeLogarithms() {
   lg[1] = 0;
   for (int i = 2; i <= MAXN; i++)
      lg[i] = lg[i/2] + 1;
}

void sparseTable() {
   copy(arr.begin(), arr.end(), st[0]);

   for (int i = 1; i <= K; i++)
      for (int j = 0; j + (1 << i) <= N;
j++)
         st[i][j] = min(st[i - 1][j], st[i -
1][j + (1 << (i - 1))]);
}

int getMinQuery(int L, int R) {
   int i = lg[R - L + 1];
   int minimum = min(st[i][L], st[i][R -
(1 << i) + 1]);
   return minimum;
}

/*
** Below code is for BIT that
calculates sum in 1D array
** Time Complexity: O(log(n)) // here
n is the size of array
** Space Complexity: O(n)
*/

struct FenwickTree {
   vector<int> bit;  // binary indexed
tree
   int n;

   FenwickTree(int n) {
      this->n = n;
      bit.assign(n, 0);
   }

   FenwickTree(vector<int> const &a)
: FenwickTree(a.size()) {
      for (size_t i = 0; i < a.size(); i++)
         add(i, a[i]);
   }

   int sum(int r) {
      int ret = 0;
      for (; r >= 0; r = (r & (r + 1)) - 1)
         ret += bit[r];
      return ret;
   }

   int sum(int l, int r) {
      return sum(r) - sum(l - 1);
   }

   void add(int idx, int delta) {
      for (; idx < n; idx = idx | (idx + 1))
         bit[idx] += delta;
   }
};

/*
** Below code is for BIT that
calculates minimum of [0, r] in 1D
array
** Time Complexity: O(log(n)) // here
n is the size of array
** Space Complexity: O(n)
*/

struct FenwickTreeMin {
   vector<int> bit;
   int n;
   const int INF = (int)1e9;

   FenwickTreeMin(int n) {
      this->n = n;
      bit.assign(n, INF);
   }

   FenwickTreeMin(vector<int> a) :
FenwickTreeMin(a.size()) {
      for (size_t i = 0; i < a.size(); i++)
```

```
        update(i, a[i]);
    }

    int getmin(int r) {
        int ret = INF;
        for (; r >= 0; r = (r & (r + 1)) - 1)
            ret = min(ret, bit[r]);
        return ret;
    }

    void update(int idx, int val) {
        for (; idx < n; idx = idx | (idx + 1))
            bit[idx] = min(bit[idx], val);
    }
};


/*
** Below code is for BIT that
calculates sum using 1-based indexing
in 1D array
** Time Complexity: O(log(n)) // here
n is the size of array
** Space Complexity: O(n)
*/

struct FenwickTreeOneBasedIndexing
{
    vector<int> bit;  // binary indexed
tree
    int n;

    FenwickTreeOneBasedIndexing(int
n) {
        this->n = n + 1;
        bit.assign(n + 1, 0);
    }


FenwickTreeOneBasedIndexing(vector
<int> a)
        :
FenwickTreeOneBasedIndexing(a.size(
)) {
        for (size_t i = 0; i < a.size(); i++)
            add(i, a[i]);
    }

    int sum(int idx) {
        int ret = 0;
        for (++idx; idx > 0; idx -= idx &
-idx)
            ret += bit[idx];
        return ret;
    }

    int sum(int l, int r) {
        return sum(r) - sum(l - 1);
    }

    void add(int idx, int delta) {
        for (++idx; idx < n; idx += idx &
-idx)
            bit[idx] += delta;
    }
};
```

```
// Lexicographical next balanced
sequence of a given string
bool next_balanced_sequence(string &
s) {
    int n = s.size();
    int depth = 0;
    for (int i = n - 1; i >= 0; i--) {
        if (s[i] == '(')
            depth--;
        else
            depth++;

        if (s[i] == '(' && depth > 0) {
            depth--;
            int open = (n - i - 1 - depth) / 2;
            int close = n - i - 1 - open;
            string next = s.substr(0, i) + ')' +
string(open, '(') + string(close, ')');
            s.swap(next);
            return true;
        }
    }
    return false;
}

// Finding kth balanced string
sequence
string kth_balanced(int n, int k) {
    vector<vector<int>> d(2*n+1,
vector<int>(n+1, 0));
    d[0][0] = 1;
    for (int i = 1; i <= 2*n; i++) {
        d[i][0] = d[i-1][1];
        for (int j = 1; j < n; j++)
            d[i][j] = d[i-1][j-1] +
d[i-1][j+1];
        d[i][n] = d[i-1][n-1];
    }

    string ans;
    int depth = 0;
    for (int i = 0; i < 2*n; i++) {
        if (depth + 1 <= n &&
d[2*n-i-1][depth+1] >= k) {
            ans += '(';
            depth++;
        } else {
            ans += ')';
            if (depth + 1 <= n)
                k -= d[2*n-i-1][depth+1];
            depth--;
        }
    }
    return ans;
}

// inversion count of the array
calculating each element inversion
using mergeSort
int _mergeSort(int arr[], int temp[], int
left, int right);
int merge(int arr[], int temp[], int left,
int mid,
          int right);
```

```
// This function sorts the
// input array and returns the
// number of inversions in the array
int mergeSort(int arr[], int array_size)
{
    int temp[array_size];
    return _mergeSort(arr, temp, 0,
array_size - 1);
}

// An auxiliary recursive function
// that sorts the input array and
// returns the number of inversions in
the array.
int _mergeSort(int arr[], int temp[], int
left, int right)
{
    int mid, inv_count = 0;
    if (right > left) {
        // Divide the array into two parts
and
        // call _mergeSortAndCountInv()
        // for each of the parts
        mid = (right + left) / 2;

        // Inversion count will be sum of
        // inversions in left-part, right-part
        // and number of inversions in
merging
        inv_count += _mergeSort(arr,
temp, left, mid);
        inv_count += _mergeSort(arr,
temp, mid + 1, right);

        // Merge the two parts
        inv_count += merge(arr, temp,
left, mid + 1, right);
    }
    return inv_count;
}

// This function merges two sorted
arrays
// and returns inversion count in the
arrays.
int merge(int arr[], int temp[], int left,
int mid,
          int right)
{
    int i, j, k;
    int inv_count = 0;

    i = left;
    j = mid;
    k = left;
    while ((i <= mid - 1) && (j <=
right)) {
        if (arr[i] <= arr[j]) {
            temp[k++] = arr[i++];
        }
        else {
            temp[k++] = arr[j++];

            // this is tricky -- see above
            // explanation/diagram for
merge()
```

```
            inv_count = inv_count + (mid -
i);
        }
    }

    // Copy the remaining elements of
left subarray
    // (if there are any) to temp
    while (i <= mid - 1)
        temp[k++] = arr[i++];

    // Copy the remaining elements of
right subarray
    // (if there are any) to temp
    while (j <= right)
        temp[k++] = arr[j++];

    // Copy back the merged elements to
original array
    for (i = left; i <= right; i++)
        arr[i] = temp[i];

    return inv_count;
}

// Euler totient phi(n), time
complexity: o(sqrt(n))
int phi(int n) {
    int result = n;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            while (n % i == 0)
                n /= i;
            result -= result / i;
        }
    }
    if (n > 1)
        result -= result / n;
    return result;
}

// Segment Tree Lazy Propagation
addition
// here, a[] is the 1-based array from
question
const int MAXN = int(1e9);
long long t[4*MAXN];

void build(int a[], int v, int tl, int tr) {
    if (tl == tr) {
        t[v] = a[tl];
    } else {
        int tm = (tl + tr) / 2;
        build(a, v*2, tl, tm);
        build(a, v*2+1, tm+1, tr);
        t[v] = 0;
    }
}

void update(int v, int tl, int tr, int l, int
r, int add) {
    if (l > r)
        return;
    if (l == tl && r == tr) {
        t[v] += add;
    } else {
        int tm = (tl + tr) / 2;
        update(v*2, tl, tm, l, min(r, tm),
add);
        update(v*2+1, tm+1, tr, max(l,
tm+1), r, add);
    }
}

int get(int v, int tl, int tr, int pos) {
    if (tl == tr)
        return t[v];
    int tm = (tl + tr) / 2;
    if (pos <= tm)
        return t[v] + get(v*2, tl, tm, pos);
    else
        return t[v] + get(v*2+1, tm+1, tr,
pos);
}

// Segment Tree Lazy Propagation
addition and querying for maximum
// here, a[] is the 1-based array from
question
const int MAXN = int(2e5);
const int INF = int(1e9);
long long t[4*MAXN];
long long lazy[4*MAXN];

void build(int a[], int v, int tl, int tr) {
    if (tl == tr) {
        t[v] = a[tl];
    } else {
        int tm = (tl + tr) / 2;
        build(a, v*2, tl, tm);
        build(a, v*2+1, tm+1, tr);
        t[v] = max(t[v*2], t[v*2 + 1]);
    }
}

void push(int v) {
    t[v*2] += lazy[v];
    lazy[v*2] += lazy[v];
    t[v*2+1] += lazy[v];
    lazy[v*2+1] += lazy[v];
    lazy[v] = 0;
}

void update(int v, int tl, int tr, int l, int
r, int addend) {
    if (l > r)
        return;
    if (l == tl && tr == r) {
        t[v] += addend;
        lazy[v] += addend;
    } else {
        push(v);
        int tm = (tl + tr) / 2;
        update(v*2, tl, tm, l, min(r, tm),
addend);
        update(v*2+1, tm+1, tr, max(l,
tm+1), r, addend);
        t[v] = max(t[v*2], t[v*2+1]);
    }
}

int query(int v, int tl, int tr, int l, int r) {
    if (l > r)
        return -INF;
    if (l == tl && tr == r)
        return t[v];
    push(v);
    int tm = (tl + tr) / 2;
    return max(query(v*2, tl, tm, l,
min(r, tm)),
               query(v*2+1, tm+1, tr, max(l,
tm+1), r));
}

// Persistent Segment Tree that
records history of each range
struct Vertex {
    Vertex *l, *r;
    int sum;

    Vertex(int val) : l(nullptr), r(nullptr),
sum(val) {}
    Vertex(Vertex *l, Vertex *r) : l(l),
r(r), sum(0) {
        if (l) sum += l->sum;
        if (r) sum += r->sum;
    }
};

Vertex* build(int a[], int tl, int tr) {
    if (tl == tr)
        return new Vertex(a[tl]);
    int tm = (tl + tr) / 2;
    return new Vertex(build(a, tl, tm),
build(a, tm+1, tr));
}

int get_sum(Vertex* v, int tl, int tr, int
l, int r) {
    if (l > r)
        return 0;
    if (l == tl && tr == r)
        return v->sum;
    int tm = (tl + tr) / 2;
    return get_sum(v->l, tl, tm, l, min(r,
tm))
         + get_sum(v->r, tm+1, tr, max(l,
tm+1), r);
}

Vertex* update(Vertex* v, int tl, int tr,
int pos, int new_val) {
    if (tl == tr)
        return new Vertex(new_val);
    int tm = (tl + tr) / 2;
    if (pos <= tm)
        return new Vertex(update(v->l, tl,
tm, pos, new_val), v->r);
    else
        return new Vertex(v->l,
update(v->r, tm+1, tr, pos, new_val));
}


// Fermat's theorem impl
// Here, binpower(...) function refers
the binary exponentiation
```

```
bool probablyPrimeFermat(int n, int
iter=5) {
   if (n < 4)
      return n == 2 || n == 3;

   for (int i = 0; i < iter; i++) {
      int a = 2 + rand() % (n - 3);
      if (binpower(a, n - 1, n) != 1)
         return false;
   }
   return true;
}
```

// **Miller Robin theorem impl**

```
using u64 = uint64_t;
using u128 = __uint128_t;

u64 binpower(u64 base, u64 e, u64
mod) {
   u64 result = 1;
   base %= mod;
   while (e) {
      if (e & 1)
         result = (u128)result * base %
mod;
      base = (u128)base * base % mod;
      e >>= 1;
   }
   return result;
}
```

bool check_composite(u64 n, u64 a, u64 d, int s) {

```
   u64 x = binpower(a, d, n);
   if (x == 1 || x == n - 1)
      return false;
   for (int r = 1; r < s; r++) {
      x = (u128)x * x % n;
      if (x == n - 1)
         return false;
   }
   return true;
};
```

bool MillerRabin(u64 n, int iter=5) { // returns true if n is probably prime, else returns false.

```
   if (n < 4)
      return n == 2 || n == 3;

   int s = 0;
   u64 d = n - 1;
   while ((d & 1) == 0) {
      d >>= 1;
      s++;
   }

   for (int i = 0; i < iter; i++) {
      int a = 2 + rand() % (n - 3);
      if (check_composite(n, a, d, s))
         return false;
   }
   return true;
}
```

// **Finding Articulation Point of forest like graph**

```
int n; // number of nodes
vector<vector<int>> adj; // adjacency
list of graph

vector<bool> visited;
vector<int> tin, low;
int timer;

void dfs(int v, int p = -1) {
   visited[v] = true;
   tin[v] = low[v] = timer++;
   int children=0;
   for (int to : adj[v]) {
      if (to == p) continue;
      if (visited[to]) {
         low[v] = min(low[v], tin[to]);
      } else {
         dfs(to, v);
         low[v] = min(low[v], low[to]);
         if (low[to] >= tin[v] && p!=-1)
            IS_CUTPOINT(v);
         ++children;
      }
   }
   if(p == -1 && children > 1)
      IS_CUTPOINT(v);
}

void find_cutpoints() {
   timer = 0;
   visited.assign(n, false);
   tin.assign(n, -1);
   low.assign(n, -1);
   for (int i = 0; i < n; ++i) {
      if (!visited[i])
         dfs (i);
   }
}
```

// This code refers to the count of **Longest Increasing Subsequence in an array**

```
// Time Complexity: O(nlogn)
int lis(vector<int> const& a) {
   int n = a.size();
   const int INF = 1e9;
   vector<int> d(n+1, INF);
   d[0] = -INF;

   for (int i = 0; i < n; i++) {
      int l = upper_bound(d.begin(),
d.end(), a[i]) - d.begin();
      if (d[l-1] < a[i] && a[i] < d[l])
         d[l] = a[i];
   }

   int ans = 0;
   for (int l = 0; l <= n; l++) {
      if (d[l] < INF)
         ans = l;
   }
   return ans;
}
```

// **This is the impl of KMP pattern** matching of a substring to string

```
// Time Complexity: O(n)
vector<int> prefix_function(string s) {
   int n = (int)s.length();
   vector<int> pi(n);
   for (int i = 1; i < n; i++) {
      int j = pi[i-1];
      while (j > 0 && s[i] != s[j])
         j = pi[j-1];
      if (s[i] == s[j])
         j++;
      pi[i] = j;
   }
   return pi;
}
```

// **An LCA implementation in graph using Binary Lifting**

```
// Time Complexity: O(log(n))
int n, l;
vector<vector<int>> adj;

int timer;
vector<int> tin, tout;
vector<vector<int>> up;

void dfs(int v, int p)
{
   tin[v] = ++timer;
   up[v][0] = p;
   for (int i = 1; i <= l; ++i)
      up[v][i] = up[up[v][i-1]][i-1];

   for (int u : adj[v]) {
      if (u != p)
         dfs(u, v);
   }

   tout[v] = ++timer;
}

bool is_ancestor(int u, int v)
{
   return tin[u] <= tin[v] && tout[u]
>= tout[v];
}

int lca(int u, int v)
{
   if (is_ancestor(u, v))
      return u;
   if (is_ancestor(v, u))
      return v;
   for (int i = l; i >= 0; --i) {
      if (!is_ancestor(up[u][i], v))
         u = up[u][i];
   }
   return up[u][0];
}
```

```
void preprocess(int root) {
   tin.resize(n);
   tout.resize(n);
   timer = 0;
   l = ceil(log2(n));
   up.assign(n, vector<int>(l + 1));
   dfs(root, root);
}

//can be put in a knapsack of capacity W
// here, wt is the weight array, val is the
value array of each item, and n is no of
item
// w is the capacity
int knapSack(int W, int wt[], int val[],
int n) {

   // Base Case
   if (n == 0 || W == 0)
      return 0;

   // If weight of the nth item is more
   // than Knapsack capacity W, then
   // this item cannot be included
   // in the optimal solution
   if (wt[n - 1] > W)
      return knapSack(W, wt, val, n -
1);

   // Return the maximum of two cases:
   // (1) nth item included
   // (2) not included
   else
      return max(
         val[n - 1]
            + knapSack(W - wt[n - 1],
wt, val, n - 1),
         knapSack(W, wt, val, n - 1));
}


// Using substring Hash
int count_unique_substrings (string
const& s) {
   int n = s.size();

   const int p = 31;
   const int m = 1e9 + 9;
   vector<long long> p_pow(n);
   p_pow[0] = 1;
   for (int i = 1; i < n; i++)
      p_pow[i] = (p_pow[i-1] * p) % m;

   vector<long long> h(n + 1, 0);
   for (int i = 0; i < n; i++)
      h[i+1] = (h[i] + (s[i] - 'a' + 1) *
p_pow[i]) % m;

   int cnt = 0;
   for (int l = 1; l <= n; l++) {
      unordered_set<long long> hs;
      for (int i = 0; i <= n - l; i++) {
         long long cur_h = (h[i + l] + m
- h[i]) % m;
         cur_h = (cur_h * p_pow[n-i-1])
% m;
         hs.insert(cur_h);
      }
      cnt += hs.size();
   }
   return cnt;
}


// Polynomial Rolling Hash impl of a
string with low collision rate
long long compute_hash(string const&
s) {
   const int p = 31;
   const int m = 1e9 + 9;
   long long hash_value = 0;
   long long p_pow = 1;
   for (char c : s) {
      hash_value = (hash_value + (c -
'a' + 1) * p_pow) % m;
      p_pow = (p_pow * p) % m;
   }
   return hash_value;
}


// The number of relative primes in a
given interval
// Using inclusion & exclusion
principle
int solve (int n, int r) {
   vector<int> p;
   for (int i=2; i*i<=n; ++i)
      if (n % i == 0) {
         p.push_back (i);
         while (n % i == 0)
            n /= i;
      }
   if (n > 1)
      p.push_back (n);

   int sum = 0;
   for (int msk=1; msk<(1<<p.size());
++msk) {
      int mult = 1,
         bits = 0;
      for (int i=0; i<(int)p.size(); ++i)
         if (msk & (1<<i)) {
            ++bits;
            mult *= p[i];
         }

      int cur = r / mult;
      if (bits % 2 == 1)
         sum += cur;
      else
         sum -= cur;
   }

   return r - sum;
}


// Permutation, Combination,
Inverse Exponentiation, Binary
Exponentiation
const int N = 5e5 + 10, mod = 1e9 + 7;
ll fact[N];
/*
fact[0] = 1;
for (int i = 1; i < N; ++i)
{
   fact[i] = (fact[i - 1] * i) % mod;
}
*/
long long inverseExponentiation(long
long a)
{
   return ((a <= 1) ? (a) : mod - (mod /
a) * inverseExponentiation(mod % a)
% mod);
}
long long combination(long long n,
long long r)
{
   return (fact[n] *
inverseExponentiation(fact[r]) % mod)
* inverseExponentiation(fact[n - r]) %
mod;
}
long long permutation(long long n,
long long r)
{
   return (fact[n] *
inverseExponentiation(fact[n - r]) %
mod);
}
long long binaryExponentiation(long
long base, long long power)
{
   long long carry = 1 % mod;
   long long x = base % mod;
   while (power)
   {
      if (power & 1)
         carry = (__int128_t)(carry * x)
% mod;
      x = (__int128_t)(x * x) % mod;
      power = power >> 1;
   }
   return carry % mod;
}

——----------------------------------------
// Chinese Remainder Theorem(CRT),
modularInverse, extendedGCD
// Function to calculate gcd and
extended gcd
ll extendedGCD(ll a, ll b, ll &x, ll &y)
{
   if (b == 0)
   {
      x = 1;
      y = 0;
      return a;
   }
   ll x1, y1;
   ll gcd = extendedGCD(b, a % b, x1,
y1);
```

```
    x = y1;
    y = x1 - (a / b) * y1;
    return gcd;
}

// Function to compute modular inverse
of `a` under modulo `m`
ll modularInverse(ll a, ll m)
{
    ll x, y;
    ll gcd = extendedGCD(a, m, x, y);
    if (gcd != 1)
    {
        return -1;
    }
    return (x % m + m) % m;
}

// Chinese Remainder Theorem
implementation
// ans =
chineseRemainderTheorem(rem, num);
ll
chineseRemainderTheorem(vector<ll>
&rem, vector<ll> &num)
{
    ll prod = 1LL;
    for (ll ni : num)
        prod *= ni;

    ll result = 0;
    for (size_t i = 0; i < rem.size(); i++)
    {
        ll partialProd = prod / num[i];
        ll inv =
modularInverse(partialProd, num[i]);

        result += rem[i] * partialProd *
inv;
        result %= prod;
    }
    return (result + prod) % prod;
}


—----------------------------------------------

// Trailling zero number of n ! :
ll findTrailingZerosOf_n_Factorial(ll n)
{
    if (n < 0)
        return -1;
    ll count = 0;
    for (ll i = 5; n / i >= 1; i *= 5)
        count += n / i;
    return count;
}
—----------------------------------------------
// Inversion Count : O(nlogn)
void solve(ll t)
{
    ll n;
    cin >> n;
    ll s[n];
    for (int i = 0; i < n; i++)
        cin >> s[i];
    ll s1 = 0;
    ordered_set o_set;
    for (ll i = n - 1; i >= 0; i --)
    {
        s1 += o_set.order_of_key(s[i]);
        o_set.insert(s[i]);
    }
    cout << s1 << endl;
}

// Order set :
#include
<ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;

#define ordered_set tree<int, null_type,
less<int>, rb_tree_tag,
tree_order_statistics_node_update>
ordered_set o_set;
/ insert function to insert in
        // ordered set same as SET STL
        o_set.insert(5);
o_set.insert(1);
o_set.insert(2);

// Finding the second smallest element
// in the set using * because
//  find_by_order returns an iterator
cout << *(o_set.find_by_order(1)) <<
endl;

// Finding the number of elements
// strictly less than k=4
cout << o_set.order_of_key(4) <<
endl;

// Finding the count of elements less
// than or equal to 4 i.e. strictly less
// than 5 if integers are present
cout << o_set.order_of_key(5) <<
endl;

// Deleting 2 from the set if it exists
if (o_set.find(2) != o_set.end())
    o_set.erase(o_set.find(2));

// Now after deleting 2 from the set
// Finding the second smallest element
in the set
cout << *(o_set.find_by_order(1)) <<
endl;

// Finding the number of
// elements strictly less than k=4
cout << o_set.order_of_key(4) <<
endl;
```

-> next_permutation(s1.begin(), s1.end()); s1 = "ABC" -> s1 = "ACB"; ইহা s1 string এর index/character swap করে।
-> __builtin_popcount(Bit_int01); Bit_int01 = 10; -> 2 ইহা Bit_int01 এর 1 বিট গণনা করে।
-> v.erase(unique(v.begin(),v.end()),v.end()) ইহা সব প্রতিলিপি মুছে ফেলে।
-> n তম পদ = a + (n-1) * d
-> প্রথম n তম পদের সমষ্টি, Sn = (n/2) * {2*a + (n-1)*d}
-> n তম পদ = a*r^(n-1)
-> প্রথম n তম পদের সমষ্টি, Sn = (a) * { (r^n-1)/(r-1) } if r>1
Sn = (a) * { (1-r^n)/(1-r) } if r<1
-> A^(p-1) = 1 mod (M) — Fermat theorem
-> Sum of divisor = {(P1^(n1+1)-1)/(p1-1)} * {(P2^(n2+1)-1)/(p2-1)} * … * {(Pn^(nn+1)-1)/(pn-1)} // here n is the number of P that can divide the value.
-> Number of divisor of P1^a * P2^b * … * Pn^v = (a+1) * (b+1) * … * (v+1)
-> 1^2 + 2^2 + … + n^2 = { n * (n+1) * (2*n+1) } / 6
-> (a * 1/b) %mod = (a * bigmod(b, m−2, m)) mod m
-> a = logb c => b^a = c
-> nC0 + nC1 + nC2 + … + nCn = 2^n
-> nC0 + nC2 + nC4 + … = nC1 + nC3 + nC5 + … = 2^(n-1)
-> A xor B = A|B - A&B
-> a*x^2 + b*x + c = { - b +- Square Root(b^2 - 4*a*c) } / (2 * a)
-> #define PI 2.0 * acos(0.0)
-> upper_bound(v.begin(), v.end(), value)-v.begin(); // return the index where you can store new value.
-> sort(v1.begin(), v1.end(), [&](vector<long long> &a, vector<long long> &b){
return a[1]<b[1]; });
->