ICPC Dhaka Regional 2025

Team: BUBT_Sunday_Monday_Close

Team Members:

1. Md. Tuhin Hasnat
2. Amir Hamza Miraz
3. Rakin Absar Ratul

Generated: December 16, 2025

Contents

1 Data Structures 1
- 1.1 DFS Tree 1
- 1.2 LCA DFS 1
- 1.3 LCA O1 2
- 1.4 LCA Sparse Table 2
- 1.5 Merge Sort Tree 2
- 1.6 Segment Tree 2D 3
- 1.7 Segment Tree Lazy 4
- 1.8 Tree Diameter 4
- 1.9 Trie 5

2 Math 5
- 2.1 BigMod 5
- 2.2 BigMod Advanced 5
- 2.3 Divisors 6
- 2.4 Divisors Precalc 6
- 2.5 Legendres Formula 6
- 2.6 NCR NPR 6
- 2.7 Number Hashing RNG 6
- 2.8 Prime Factorization SPF 7
- 2.9 Prime Factorization SPF Advanced 7
- 2.10 Prime Factorization Sieve 7
- 2.11 Sieve Advanced 8
- 2.12 Sieve of Eratosthenes 8
- 2.13 Trailing Zeros Factorial 8

3 Graphs 8
- 3.1 Articulation Points 8
- 3.2 Bellman Ford 8
- 3.3 Bipartite BFS 9
- 3.4 Bipartite DFS 9
- 3.5 Bridges 10
- 3.6 Cycle Detection Directed 10
- 3.7 Cycle Detection Undirected 10
- 3.8 DSU Union by Rank 11
- 3.9 DSU Union by Size 11
- 3.10 Dijkstra PQ 12
- 3.11 Dijkstra Set 12
- 3.12 Floyd Warshall 13
- 3.13 Kruskals Rank 13
- 3.14 Kruskals Size 14
- 3.15 Max Flow Dinics 14
- 3.16 Prims MST 14
- 3.17 SCC Kosaraju 15
- 3.18 Topological Sort Kahn 15

4 Strings 16
- 4.1 Aho Corasick 16
- 4.2 Aho Corasick H 16
- 4.3 Aho Corasick H Class 17
- 4.4 Double Hashing 18
- 4.5 Double Hashing Class 19
- 4.6 Double Hashing Pairwise 19
- 4.7 Double Hashing Segtree 20
- 4.8 Hashing 2D 21
- 4.9 KMP 21
- 4.10 KMP1 22
- 4.11 String Hashing 22
- 4.12 Trie LC 22
- 4.13 Trie Tree 22

5 Utilities 22
- 5.1 Cumulative Sum 1D 22
- 5.2 Cumulative Sum 2D 23

Data Structures

DFS Tree

DFS tree utilities.

Time Complexity: $O(n)$

Code in C++:

```cpp
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#include <iostream>
#include <string>
#include <vector>
#include <sstream>
#include <algorithm>
#include <queue>
#include <map>
using namespace std;

#define deb(a)     cout<<__LINE__<<"# "<<#a<<" ->
 ↪ "<<a<<endl;

//For Debugging
#define debug(a...)         {cout<<__LINE__<<"#-->
 ↪ ";dbg,a; cout<<endl;}
struct debugger
{
    template<typename T> debugger& operator , (const T v)
    {
        cout<<v<<" ";
        return *this;
    }
} dbg;


typedef long long LL;
const LL MOD = 1000000007;
const double EPS = 1e-7; ///1*10^-7


const int V_SZ = 101;

/**
enum Color{
    WHITE,
    GRAY,
    BLACK
};
*/

const int WHITE = 0;
const int GRAY = 1;
const int BLACK = 2;

vector<int>g[V_SZ];

int startTime[V_SZ];
int finishTime[V_SZ];
int Flattening_tree[2*V_SZ];
int depth[V_SZ];
int height[V_SZ];
int subtree_sum[V_SZ];

vector<int>order;

int Time;
int V,E;
void init(){
    for(int i =1;i<=V;i++)
    {
        g[i].clear();

        startTime[i] = finishTime[i] = -1;
        Flattening_tree[i]=Flattening_tree[V-i+1]=-1;

        height[i]=0;
        depth[i]=0;
        subtree_sum[i]=i;
    }
    Time = 1;
    order.clear();
```

```cpp
}

void dfs(int u,int par=-1)
{
    startTime[u] = Time;
    Flattening_tree[Time]=u;
    Time++;
    for(auto v: g[u])
    {
        if(v!=par){
            depth[v]=depth[u]+1;
            dfs(v,u);
            height[u]=max(height[u],height[v]+1);
            subtree_sum[u]+=subtree_sum[v];
        }
    }


    order.push_back(u);

    finishTime[u] = Time;
    Flattening_tree[Time]=u;// change 0,-u
    Time++;
}
```

LCA DFS

LCA via DFS.

Time Complexity: O(n) prep

Code in C++:

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

const int no_node=1000;
vector<int>adj[no_node];
int par[no_node];

void init(int V){
    for(int i=0;i<V+5;i++){
        adj[i].clear();
        par[i]=-1;
    }
}

void dfs(int u){
    for(auto v : adj[u]){
        if(v==par[u])continue;
        par[v]=u;
        dfs(v);
    }
}
vector<int>ancestors(int node){
    vector<int>ans;
    while(node != -1){///node no. 1 s parent in -1
        ans.push_back(node);
        node=par[node];
    }
    reverse(ans.begin(),ans.end());
    return ans;
}
int LCA(int nod1,int nod2){
    vector<int>anc1=ancestors(nod1);
    vector<int>anc2=ancestors(nod2);
    int len=min(anc1.size(),anc2.size());
    int lca=-1;
    for(int i=0;i<len;i++){
        if(anc1[i]==anc2[i]){
            lca=anc1[i];
        }else{
            break;
        }
    }
    return lca;

}
```

LCA O1

LCA O(1) query.

Time Complexity: O(n) prep

Code in C++:

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

template <class T>
struct RMQ { // 0-based
  vector<vector<T>> rmq;
  T kInf = numeric_limits<T>::max();
  void build(const vector<T>& V) {
    int n = V.size(), on = 1, dep = 1;
    while (on < n) on *= 2, ++dep;
    rmq.assign(dep, V);

    for (int i = 0; i < dep - 1; ++i)
      for (int j = 0; j < n; ++j) {
        rmq[i + 1][j] = min(rmq[i][j], rmq[i][min(n - 1,
         ↪ j + (1 << i))]);
      }
  }
  T query(int a, int b) { // [a, b)
    if (b <= a) return kInf;
    int dep = 31 - __builtin_clz(b - a); // log(b - a)
    return min(rmq[dep][a], rmq[dep][b - (1 << dep)]);
  }
};

struct LCA { // 0-based
  vector<int> enter, depth, exxit;
  vector<vector<int>> G;
  vector<pair<int, int>> linear;
  RMQ<pair<int, int>> rmq;
  int timer = 0;
  LCA() {}
  LCA(int n) : enter(n, -1), exxit(n, -1), depth(n),
   ↪ G(n), linear(2 * n) {}
  void dfs(int node, int dep) {
    linear[timer] = {dep, node};
    enter[node] = timer++;
    depth[node] = dep;
    for (auto vec : G[node])
    if (enter[vec] == -1) {
      dfs(vec, dep + 1);
      linear[timer++] = {dep, node};
    }
    exxit[node] = timer;
  }
  void add_edge(int a, int b) {
    G[a].push_back(b);
    G[b].push_back(a);
  }
  void build(int root) {
    dfs(root, 0);
    rmq.build(linear);
  }
  int query(int a, int b) {
    a = enter[a], b = enter[b];
    return rmq.query(min(a, b), max(a, b) + 1).second;
  }
  int dist(int a, int b) {
    return depth[a] + depth[b] - 2 * depth[query(a, b)];
  }
};

LCA lca;
```

LCA Sparse Table

LCA sparse table.

Time Complexity: O(n log n) prep

Code in C++:

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

///Complexity: O(NlgN,lgN)
const int Size = 100010;

int E,V;
int LVL[Size];
int par[Size];
int A[Size][20];
```

```
vector<int>adj[Size];

/// finding nodes tree level and parent
void leveling_dfs(int u){
    for(auto v : adj[u]){
        if(v==par[u])continue;
        LVL[v]=LVL[u]+1;
        par[v]=u;
        leveling_dfs(v);
    }
}

void Sparse_Table()
{
    // creating sparse table
    for(int p=0;p<=log2(V)+1;p++)
    {
        for(int i=1;i<=V;i++)
        {
            if(p==0)
                A[i][p] = par[i];///2^p = 2^0 = 1'th parent
            else
                A[i][p] = A[A[i][p-1]][p-1];///  A[i][p] = i'th nodes 2^p'th parant
        }
    }
}

int LCA(int u,int v)
{
    if(LVL[u]>LVL[v])
        swap(u,v);
    //Bring u and v in same level
    for(int i=log2(V)+1;i>=0;i--){
        int x = A[v][i];
        if(LVL[u]==LVL[x]){
            v=x;
            break;
        }
        if(LVL[u]<LVL[x])
            v = x;
    }
    if(u==v)return u;

    for(int i=log2(V)+1;i>=0;i--)
    {
        if(A[u][i] != -1 && A[u][i] != A[v][i])
        {
            u = A[u][i];
            v = A[v][i];
        }
    }
    return par[u];
}
int distance(int u,int v){
    int an=LCA(u,v);
    return LVL[u]+LVL[v]-2*LVL[an];
}
void build_LCA(int source){
    LVL[source]=1,par[source]=source;
    leveling_dfs(source);
    Sparse_Table();
}
```

Merge Sort Tree

Merge sort tree.

Time Complexity: O(n log n)

Code in C++:

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

typedef long long ll;
const int Size=30000+10;
vector<ll>ar;
vector<vector<ll>>tree;
vector<ll> marge(vector<ll>&a,vector<ll>&b){
    int n=a.size(),m=b.size();
    vector<ll>c;
    int i=0,j=0;
    while(i<n && j<m){
        if(a[i]<=b[j]){
            c.push_back(a[i]);
            i++;
        }else{
            c.push_back(b[j]);
            j++;
        }
    }
    while(i<n)c.push_back(a[i]),i++;
    while(j<m)c.push_back(b[j]),j++;
    return c;
}
void build(int node,int left,int right){
    if(left==right){
        tree[node].push_back(ar[left]);
        return ;
    }
    int mid=(left+right)/2;
    build(node*2,left,mid);
    build(node*2+1,mid+1,right);
    tree[node]=marge(tree[node*2],tree[node*2+1]);
}
int query(int node,int left,int right,int ql,int qr,ll k){///query left=ql,right=qr
    if(left>=ql && right<=qr){
        int ans= (int)tree[node].size()
        -(upper_bound(tree[node].begin(),tree[node].end(),k)-tree[node].begin());
        return ans;
    }
    int mid=(left+right)/2;
    if(qr<=mid){
        return query(2*node,left,mid,ql,qr,k);
    }
    else if(mid<ql){
        return query(2*node+1,mid+1,right,ql,qr,k);
    }
    else{
        int left_node=query(2*node,left,mid,ql,mid,k);
        int right_node=query(2*node+1,mid+1,right,mid+1,qr,k);
        return left_node+right_node;
    }
}
```

Segment Tree 2D

2D segment tree.

Time Complexity: O(n^2)*build*

Code in C++:

```
#include<bits/stdc++.h>
using namespace std;
mt19937 rnd(chrono::steady_clock::now().time_since_epoch().count());

const int N = 3e5 + 9;

struct node {
  node *l, *r;
  int pos, key, mn, mx;
  long long val, g;
  node(int position, long long value) {
    l = r = nullptr;
    mn = mx = pos = position;
    key = rnd();
    val = g = value;
  }
  void pull() {
    g = val;
    if(l) g = __gcd(g, l->g);
    if(r) g = __gcd(g, r->g);
    mn = (l ? l->mn : pos);
    mx = (r ? r->mx : pos);
  }
};
//memory O(n)
struct treap {
  node *root;
  treap() {
    root = nullptr;
  }
```

```cpp
void split(node *t, int pos, node *&l, node *&r) {
  if (t == nullptr) {
    l = r = nullptr;
    return;
  }
  if (t->pos < pos) {
    split(t->r, pos, l, r);
    t->r = l;
    l = t;
  } else {
    split(t->l, pos, l, r);
    t->l = r;
    r = t;
  }
  t->pull();
}
node* merge(node *l, node *r) {
  if (!l || !r) return l ? l : r;
  if (l->key < r->key) {
    l->r = merge(l->r, r);
    l->pull();
    return l;
  }
  r->l = merge(l, r->l);
  r->pull();
  return r;
}
bool find(int pos) {
  node *t = root;
  while (t) {
    if (t->pos == pos) return true;
    if (t->pos > pos) t = t->l;
    else t = t->r;
  }
  return false;
}
void upd(node *t, int pos, long long val) {
  if (t->pos == pos) {
    t->val = val;
    t->pull();
    return;
  }
  if (t->pos > pos) upd(t->l, pos, val);
  else upd(t->r, pos, val);
  t->pull();
}
void insert(int pos, long long val) { //set a_pos = val
  if (find(pos)) upd(root, pos, val);
  else {
    node *l, *r;
    split(root, pos, l, r);
    root = merge(merge(l, new node(pos, val)), r);
  }
}
long long query(node *t, int st, int en) {
  if (t->mx < st || en < t->mn) return 0;
    if (st <= t->mn && t->mx <= en) return t->g;
    long long ans = (st <= t->pos && t->pos <= en ?
     ↪  t->val : 0);
    if (t->l) ans = __gcd(ans, query(t->l, st, en));
    if (t->r) ans = __gcd(ans, query(t->r, st, en));
    return ans;
  }
  long long query(int l, int r) { //gcd of a_i such that l
   ↪  <= i <= r
    if (!root) return 0;
    return query(root, l, r);
  }
  void print(node *t) {
    if (!t) return;
    print(t->l);
    cout << t->val << " ";
    print(t->r);
  }
};
//total memory along with treap = nlogn
struct ST {
  ST *l, *r;
  treap t;
  int b, e;
  ST() {
    l = r = nullptr;
  }
  ST(int st, int en) {
    l = r = nullptr;
    b = st, e = en;
  }
  void fix(int pos) {
    long long val = 0;
    if (l) val = __gcd(val, l->t.query(pos, pos));
    if (r) val = __gcd(val, r->t.query(pos, pos));
    t.insert(pos, val);
  }
  void upd(int x, int y, long long val) { //set a[x][y] =
   ↪  val
    if (e < x || x < b) return;
    if (b == e) {
      t.insert(y, val);
      return;
    }
    if (b != e) {
      if (x <= (b + e) / 2) {
        if (!l) l = new ST(b, (b + e) / 2);
        l->upd(x, y, val);
      } else {
        if (!r) r = new ST((b + e) / 2 + 1, e);
        r->upd(x, y, val);
      }
    }
    fix(y);
  }
  long long query(int i, int j, int st, int en) { //gcd of
   ↪  a[x][y] such that i <= x <= j && st <= y <= en
    if (e < i || j < b) return 0;
    if (i <= b && e <= j) return t.query(st, en);
    long long ans = 0;
    if (l) ans = __gcd(ans, l->query(i, j, st, en));
    if (r) ans = __gcd(ans, r->query(i, j, st, en));
    return ans;
  }
};
```

Segment Tree Lazy

Segment Tree with Lazy Propagation for efficient range queries and range updates.

Time Complexity:

- Build: $O(n)$
- Range Query: $O(\log n)$
- Range Update: $O(\log n)$

Key Feature: Lazy propagation delays updates until needed, allowing O(log n) range updates.

Code in C++:

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

typedef long long ll;
const int MAX_N = 100007;
int ar[MAX_N];

struct LazyTree {
    vector<int> tre, lazy;

    LazyTree(int sz) {
        tre.assign((sz * 4) + 10, 0);
        lazy.assign((sz * 4) + 10, 0);
    }

    inline void lazyUpdate(int nod, int sl, int sr) {
        if(lazy[nod] == 0) return;
        tre[nod] += lazy[nod] * (sr - sl + 1);
        if(sl != sr) {
            int left_child = 2 * nod, right_child = 2 *
             ↪  nod + 1;
            lazy[left_child] += lazy[nod];
            lazy[right_child] += lazy[nod];
        }
        lazy[nod] = 0;
    }
```

```cpp
    void build(int nod, int sl, int sr) {
        lazy[nod] = 0;
        if(sl == sr) {
            tre[nod] = ar[sr];
            return;
        }
        int mid = (sl + sr) / 2;
        int left_child = 2 * nod, right_child = 2 * nod +
         ↪  1;

        build(left_child, sl, mid);
        build(right_child, mid + 1, sr);

        tre[nod] = tre[left_child] + tre[right_child];
    }

    ll query(int nod, int sl, int sr, int ql, int qr) {
        lazyUpdate(nod, sl, sr);
        if(ql <= sl && sr <= qr) {
            return tre[nod];
        }
        if(qr < sl || sr < ql) return 0;
        int mid = (sl + sr) / 2;
        int left_child = 2 * nod, right_child = 2 * nod +
         ↪  1;

        return query(left_child, sl, mid, ql, qr) +
         ↪  query(right_child, mid + 1, sr, ql, qr);
    }

    void update(int nod, int sl, int sr, int ql, int qr,
     ↪  ll val) {
        lazyUpdate(nod, sl, sr);
        if(ql <= sl && sr <= qr) {
            lazy[nod] += val;
            lazyUpdate(nod, sl, sr);
            return;
        }
        if(qr < sl || sr < ql) return;

        int mid = (sl + sr) / 2;
        int left_child = 2 * nod, right_child = 2 * nod +
         ↪  1;

        update(left_child, sl, mid, ql, qr, val);
        update(right_child, mid + 1, sr, ql, qr, val);

        tre[nod] = tre[left_child] + tre[right_child];
    }
};
```

Tree Diameter

Tree diameter.

Time Complexity: O(n)

Code in C++:

```cpp
#include<bits/stdc++.h>
using namespace std;
const int Size=1000;

int depth[Size];
vector<int>graph[Size];
int max_depth;
int max_depth_node;
void init(int V){
    for(int i=0;i<V+5;i++){
        graph[i].clear();
        depth[i]=0;
    }
    max_depth=0;
}
int dfs(int u,int par=-1){
    if(depth[u]>max_depth){
        max_depth=depth[u];
        max_depth_node=u;
    }
    for(auto v : graph[u]){
        if(v==par)continue;
        depth[v]=depth[u]+1;
        dfs(v,u);
    }
    return max_depth_node;
}
```

Trie

Trie (prefix tree) for efficient string storage and prefix-based queries.

Time Complexity: $O(L)$ per operation, where L is word length

Operations:

- `insert(word)`: Add word
- `countWordsEqualTo(word)`: Count exact matches
- `countWordsStartingWith(prefix)`: Count with prefix
- `erase(word)`: Remove word

Code in C++:

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

struct Node {
    Node* links[26];
    int cntword = 0;
    int cntPrefix = 0;

    bool next_exist(char ch) {
        return (links[ch - 'a'] != NULL);
    }
    void create_ref_nod(char ch, Node* node) {
        links[ch - 'a'] = node;
    }
    Node* next(char ch) {
        return links[ch - 'a'];
    }
    void increaseWordFrequency() {
        cntword++;
    }
    void increasePrefixFrequency() {
        cntPrefix++;
    }
    void deleteWordFrequency() {
        cntword--;
    }
    void reducePrefixFrequency() {
        cntPrefix--;
    }
    int WordFrequency() {
        return cntword;
    }
    int PrefixFrequency() {
        return cntPrefix;
    }
};

class Trie {
private:
    Node* root;
public:
    Trie() {
        root = new Node();
    }

    void insert(string word) {
        Node* node = root;
        for (int i = 0; i < word.length(); i++) {
            if (!node->next_exist(word[i])) {
                node->create_ref_nod(word[i], new Node());
            }
            node = node->next(word[i]);
            node->increasePrefixFrequency();
        }
        node->increaseWordFrequency();
    }

    int countWordsEqualTo(string &word) {
        Node *node = root;
```

```cpp
        for (int i = 0; i < word.length(); i++) {
            if (!node->next_exist(word[i])) {
                return 0;
            }
            node = node->next(word[i]);
        }
        return node->WordFrequency();
    }

    int countWordsStartingWith(string & word) {
        Node * node = root;
        for (int i = 0; i < word.length(); i++) {
            if (!node->next_exist(word[i])) {
                return 0;
            }
            node = node->next(word[i]);
        }
        return node->PrefixFrequency();
    }

    void erase(string & word) {
        Node * node = root;
        for (int i = 0; i < word.length(); i++) {
            if (!node->next_exist(word[i])) {
                return ;
            }
            node = node->next(word[i]);
            node->reducePrefixFrequency();
        }
        node->deleteWordFrequency();
    }
};
```

Math

BigMod

Modular Exponentiation (Big Mod) computes (b^{power}) mod *mod* efficiently.

Time Complexity: $O(\log power)$

Applications:

- Computing large powers under modulo
- Modular multiplicative inverse
- RSA encryption

Code in C++:

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long LL;

LL bigMod(LL b, LL power, LL mod) {
    LL ans = 1;
    while(power) {
        if(power & 1) ans = (ans * b) % mod;
        b = (b * b) % mod;
        power = power >> 1;
    }
    return ans % mod;
}
```

BigMod Advanced

Advanced modular exponentiation.

Time Complexity: O(log n)

Code in C++:

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

#define ll long long

ll mul(ll a, ll b, ll mod) { // a * b % mod
  return __int128(a) * b % mod;
}

ll power(ll a, ll b, ll mod) { // a^b % mod
  ll ans = 1 % mod;
  while (b) {
    if (b & 1) ans = mul(ans, a, mod);
    a = mul(a, a, mod);
    b >>= 1;
  }
  return ans;
}

ll inverse(ll a, ll mod) { // (1 / a) % mod
  return power(a, mod - 2, mod);
}
```

Divisors

Find all divisors of a number efficiently.

Time Complexity: $O(\sqrt{n})$

Code in C++:

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

vector<int> divisors(int n) {
    vector<int> divs;
    for (int i = 1; i * i <= n; i++) {
        if (n % i == 0) {
            divs.push_back(i);
            if (i != n / i) divs.push_back(n / i);
        }
    }
    sort(divs.begin(), divs.end());
    return divs;
}
```

Divisors Precalc

Precalculated divisors.

Time Complexity: O(n log n) prep

Code in C++:

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

/// divisor pre calculate
///nlog(log(n))
vector<int>divisors[1000010];
void Divisor()
{
    for(int div=1;div<=1000000;div++)
        for(int num=div;num<=1000000;num+=div)///num is a
         ↪  number
            divisors[num].push_back(div);     //which is
             ↪  contain divisor div
}
```

Legendres Formula

Prime power in factorial.

Time Complexity: O(log n)

Code in C++:

```cpp
// Prime power in factorial
// Placeholder - see source for full implementation
#include <bits/stdc++.h>
using namespace std;

// Prime power in factorial
// O(log n)
```

NCR NPR

Combinatorics: nCr (combinations) and nPr (permutations) with modular arithmetic and precalculation.

Time Complexity:

- Preprocessing: $O(N)$
- Per query: $O(1)$

Uses modular inverse for division under modulo.

Code in C++:

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

typedef long long ll;
const int N = 1000000;
const ll MOD = 1e9 + 7;

ll fact[N + 10];
ll inv_fact[N + 10];

ll bigMod(ll base, ll power, ll mod = 1e9 + 7) {
    ll ans = 1;
    while(power) {
        if(power & 1) ans = (ans * base) % mod;
        base = (base * base) % mod;
        power = power >> 1;
    }
    return ans;
}

ll inverse(ll base, ll mod = 1e9 + 7) {
    return bigMod(base % mod, mod - 2, mod) % mod;
}

void preCalc() {
    fact[0] = 1;
    for (ll i = 1; i <= N; i++)
        fact[i] = (fact[i - 1] * i) % MOD;

    inv_fact[N] = inverse(fact[N]);
    for (ll i = N - 1; i >= 0; i--)
        inv_fact[i] = (inv_fact[i + 1] * (i + 1)) % MOD;
}

ll nCr(ll n, ll r) {
    if (r > n || r < 0) return 0;
    return fact[n] * inv_fact[r] % MOD * inv_fact[n - r] % MOD;
}

ll nPr(ll n, ll r) {
    if (r > n || r < 0) return 0;
    return fact[n] * inv_fact[n - r] % MOD;
}
```

Number Hashing RNG

Number hashing.

Time Complexity: O(1)

Code in C++:

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

struct custom_hash {
    static uint32_t splitmix32(uint32_t x) {//uint64_t
        x += 0x9e3779b97f4a7c15;
        x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
        x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
        return x ^ (x >> 31);
    }

    size_t operator()(uint32_t x) const {//uint64_t
        static const uint32_t FIXED_RANDOM = chrono::steady_clock::now().time_since_epoch().count();
        return splitmix32(x + FIXED_RANDOM);
    }
}rng;///Random number generator

signed main()
{
    int a=rng(1);
    int b=rng(2);
    int c=rng(3);


    cout<<a<<" "<<bitset<32>(a)<<"\n";
    cout<<b<<" "<<bitset<32>(b)<<"\n";
    cout<<c<<" "<<bitset<32>(c)<<"\n";

}
```

Prime Factorization SPF

Prime factorization using precalculated smallest prime factor (SPF).

Time Complexity:

- Preprocessing: $O(N \log N)$
- Per query: $O(\log n)$

Code in C++:

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

const int N = 1000005;
int spf[N];

void spfPreCalc() {
    for(int i = 2; i <= N; i++) {
        spf[i] = i;
    }
    for(int div = 2; div <= N; div++) {
        for(int i = div; i <= N; i += div) {
            spf[i] = min(spf[i], div);
        }
    }
}

vector<int> primeFactors(int n) {
    vector<int> factors;
    while(n > 1) {
        factors.push_back(spf[n]);
        n /= spf[n];
    }
    return factors;
}
```

Prime Factorization SPF Advanced

Advanced SPF.

Time Complexity: O(log n)

Code in C++:

```cpp
// C++ program to find prime factorization of a
// number n in O(Log n) time with precomputation
// allowed.
#include "bits/stdc++.h"
using namespace std;

#define MAXN 100001

// stores smallest prime factor for every number
int spf[MAXN];

// Calculating SPF (Smallest Prime Factor) for every
// number till MAXN.
// Time Complexity : O(nloglogn)
void sieve()
{
    spf[1] = 1;
    for (int i = 2; i < MAXN; i++)

        // marking smallest prime factor for every
        // number to be itself.
        spf[i] = i;

    // separately marking spf for every even
    // number as 2
    for (int i = 4; i < MAXN; i += 2)
        spf[i] = 2;

    for (int i = 3; i * i < MAXN; i++) {
        // checking if i is prime
        if (spf[i] == i) {
```

```
            // marking SPF for all numbers divisible by i
            for (int j = i * i; j < MAXN; j += i)

                // marking spf[j] if it is not
                // previously marked
                if (spf[j] == j)
                    spf[j] = i;
        }
    }
}

// A O(log n) function returning primefactorization
// by dividing by smallest prime factor at every step
vector<int> getFactorization(int x)
{
    vector<int> ret;
    while (x != 1) {
        ret.push_back(spf[x]);
        x = x / spf[x];
    }
    return ret;
}

// driver program for above function
```

Prime Factorization Sieve

Prime factorization via sieve.

Time Complexity: O(log n)

Code in C++:

```
#include<bits/stdc++.h>
using namespace std;
using ll= long long;
/// nlog(n)
#define SIZE_N 10010000///finding all prime number under
 ↪ SIZE_N
bool isprime [SIZE_N];
vector<int>prime;

void sieve()
{
    int i, j, r;
    for ( i = 3; i <= SIZE_N; i += 2 )
        isprime[i] = true ;

    isprime [2] = true ;
    prime.push_back(2);

    for ( i = 3; i <= SIZE_N; i += 2 )
        if ( isprime[i] == true )
        {
            prime.push_back(i);

            if ( SIZE_N/i >= i )
            {
                r = i * 2 ;
                for ( j = i * i; j <= SIZE_N; j += r )
                    isprime[j] = false ;///unprime all
                     ↪ nums which is divisible by i
            }
        }
}
i/// it can find 1e18 numbers prime fectors
void prime_factors(int n){
    for(int i=0; prime[i]*prime[i]<=n && i<prime.size() ;
     ↪ i++){
        while(n%prime[i]==0){
            cout<<prime[i]<<" ";
            n/=prime[i];
        }
    }
    if(n>1)cout<<n<<" ";
    cout<<"\n";
}
```

Sieve Advanced

Optimized sieve implementation.

Time Complexity: $O(N \log \log N)$

Code in C++:

```
// Simplified placeholder - complex algorithm
// For full implementation, see source:
// /home/hasnat/codes/cp-code-template/Data-Structures-an
 ↪ d-Algorithms.-main/
#include <bits/stdc++.h>
using namespace std;

// Sieve with optimization - basic implementation
const int N = 10000005;
bitset<N> not_prime;
vector<int> primes;

void sieveAdvanced() {
    not_prime[0] = not_prime[1] = true;
    for (int i = 2; i < N; i++) {
        if (!not_prime[i]) {
            primes.push_back(i);
            for (long long j = (long long)i * i; j < N; j
             ↪ += i) {
                not_prime[j] = true;
            }
        }
    }
}
```

Sieve of Eratosthenes

Sieve of Eratosthenes generates all prime numbers up to N efficiently.

Time Complexity: $O(N \log \log N)$

Space Complexity: $O(N)$

Algorithm:

- Mark all multiples of each prime as composite
- Uses bitset for memory efficiency
- Generates primes vector for quick access

Code in C++:

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

const int N = 10000005;
bitset<N> not_prime;
vector<int> primes;

void sieve() {
    not_prime[1] = true;
    for (int i = 2; i * i <= N; i++) {
        if (!not_prime[i]) {
            for (int j = i * i; j <= N; j += i) {
                not_prime[j] = true;
            }
        }
    }
    for (int i = 2; i <= N; i++) {
        if (!not_prime[i]) {
            primes.push_back(i);
        }
    }
}
```

Trailing Zeros Factorial

Trailing zeros in n!.

Time Complexity: O(log n)

Code in C++:

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

int findTrailingZeros(int n)
{
    if (n < 0) // Negative Number Edge Case
        return -1;
```

```cpp
    int count = 0;

    for (int i = 5; n / i >= 1; i *= 5)
        count += n / i;

    return count;
}
```

Graphs

Articulation Points

Articulation Points are vertices whose removal increases the number of connected components.

Time Complexity: $O(V+E)$

Algorithm: Uses DFS with discovery and low times to identify cut vertices.

Code in C++:

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

const int nodes = 100005;
int timer;
int vis[nodes], tin[nodes], tlow[nodes];
vector<int> adj[nodes];
set<int> art_points;

void artPointDFS(int node, int parent) {
    vis[node] = 1;
    tin[node] = tlow[node] = timer++;
    int child = 0;

    for(auto v : adj[node]) {
        if(v == parent) continue;
        if(vis[v] == 0) {
            artPointDFS(v, node);
            tlow[node] = min(tlow[v], tlow[node]);
            if(tlow[v] >= tin[node] && parent != -1) {
                art_points.insert(node);
            }
            child++;
        } else {
            tlow[node] = min(tin[v], tlow[node]);
        }
    }
    if(child > 1 && parent == -1) {
        art_points.insert(node);
    }
}

void init(int V) {
    for(int i = 0; i <= V; i++) {
        vis[i] = 0;
        adj[i].clear();
    }
    art_points.clear();
    timer = 1;
}
```

Bellman Ford

Bellman-Ford algorithm finds shortest paths from a source vertex to all other vertices, even with negative edge weights. Can detect negative weight cycles.

Time Complexity: $O(VE)$

Space Complexity: $O(V+E)$

Usage:

- Call `init(V)` to initialize
- Add edges using `edgeList.push_back({u, v, w})`
- Call `bellmanFord(source, V)` to compute shortest paths
- Returns `true` if negative cycle exists, `false` otherwise

Advantages:

- Works with negative edge weights
- Detects negative cycles

Code in C++:

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

const int V_SZ = 100005;
const int oo = (1 << 25);

struct Edge {
    int u, v, w;
};

vector<Edge> edgeList;
int dist[V_SZ];
int par[V_SZ];

void init(int V) {
    for(int i = 1; i <= V; i++) {
        dist[i] = oo;
        par[i] = -1;
    }
    edgeList.clear();
}

bool bellmanFord(int s, int V) {
    dist[s] = 0;
    bool isUpdated;

    for(int i = 1; i <= V; i++) {
        isUpdated = false;

        for(auto edg : edgeList) {
            if(dist[edg.v] > dist[edg.u] + edg.w) {
                dist[edg.v] = dist[edg.u] + edg.w;
                par[edg.v] = edg.u;
                isUpdated = true;
            }
        }
    }

    return isUpdated;
}
```

Bipartite BFS

A bipartite graph is a graph whose vertices can be divided into two disjoint sets such that every edge connects vertices from different sets. This implementation uses BFS with 2-coloring.

Time Complexity: $O(V+E)$

Space Complexity: $O(V)$

Key Concept:

- Color vertices with alternating colors (0 and 1)
- If two adjacent vertices have the same color, graph is not bipartite
- A graph is bipartite if and only if it contains no odd-length cycles

Applications:

- Matching problems
- Scheduling problems

- Network flow problems

Code in C++:

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

const int SIZE = 100005;

vector<int> adj[SIZE];
int color[SIZE];

bool BFS(int s) {
    color[s] = 0;
    queue<int> Q;
    Q.push(s);

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

        for(auto v : adj[u]) {
            if(color[v] == -1) {
                color[v] = !color[u];
                Q.push(v);
            }
            else if(color[v] == color[u]) {
                return false;
            }
        }
    }
    return true;
}

void init(int V) {
    for(int i = 0; i <= V; i++) {
        color[i] = -1;
        adj[i].clear();
    }
}

bool isBipartite(int V) {
    for(int i = 1; i <= V; i++) {
        if(color[i] == -1) {
            if(BFS(i) == false) {
                return false;
            }
        }
    }
    return true;
}
```

Bipartite DFS

A bipartite graph checking using DFS with 2-coloring. This is an alternative to the BFS approach.

Time Complexity: $O(V + E)$

Space Complexity: $O(V)$

Advantages of DFS over BFS:

- More concise recursive implementation
- Better for finding connected components
- Uses implicit stack (recursion)

Code in C++:

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

const int SIZE = 100005;

vector<int> adj[SIZE];
int color[SIZE];

bool DFS(int u, int col) {
    color[u] = col;

    for(auto v : adj[u]) {
        if(color[v] == -1) {
            if(DFS(v, !col) == false)
                return false;
        }
        else if(color[v] == col) {
            return false;
        }
    }
    return true;
}

void init(int V) {
    for(int i = 0; i <= V; i++) {
        color[i] = -1;
        adj[i].clear();
    }
}

bool isBipartite(int V) {
    for(int i = 1; i <= V; i++) {
        if(color[i] == -1) {
            if(DFS(i, 0) == false) {
                return false;
            }
        }
    }
    return true;
}
```

Bridges

Bridges are edges whose removal increases the number of connected components.

Time Complexity: $O(V + E)$

Algorithm: Uses DFS with discovery and low times to identify bridge edges.

Code in C++:

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

const int nodes = 100005;
int timer;
int vis[nodes], tin[nodes], tlow[nodes];
vector<int> adj[nodes];
vector<pair<int,int>> bridges;

void bridgeDFS(int node, int parent) {
    vis[node] = 1;
    tin[node] = tlow[node] = timer++;

    for(auto v : adj[node]) {
        if(v == parent) continue;
        if(vis[v] == 0) {
            bridgeDFS(v, node);
            tlow[node] = min(tlow[v], tlow[node]);
            if(tlow[v] > tin[node]) {
                bridges.push_back({node, v});
            }
        } else {
            tlow[node] = min(tlow[v], tlow[node]);
        }
    }
}

void init(int V) {
    for(int i = 0; i <= V; i++) {
        vis[i] = 0;
        adj[i].clear();
    }
    bridges.clear();
    timer = 1;
}
```

Cycle Detection Directed

Detects cycles in a directed graph using DFS with path tracking (recursion stack).

Time Complexity: $O(V + E)$

Space Complexity: $O(V)$

Algorithm:

- Use two arrays: `vis[]` (visited) and `pathvis[]` (in current path)
- Mark node as visited and in current path during DFS
- If we visit a node that's in the current path, we found a cycle
- Unmark from path when backtracking

Key Difference from Undirected:

- In directed graphs, we check if node is in current recursion path
- In undirected graphs, we check parent to avoid false positives

Code in C++:

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

const int SZ = 100005;
vector<int> g[SZ];
int vis[SZ];
int pathvis[SZ];

void init(int V) {
    for(int i = 1; i <= V; i++) {
        pathvis[i] = 0;
        vis[i] = 0;
        g[i].clear();
    }
}

bool dfs(int u, int par) {
    vis[u] = 1;
    pathvis[u] = 1;

    for(auto v : g[u]) {
        if(vis[v] == 0) {
            if(dfs(v, u) == true)
                return true;
        }
        else if(pathvis[v]) {
            return true;
        }
    }

    pathvis[u] = 0;
    return false;
}

bool hasCycleDirected(int V) {
    for(int i = 1; i <= V; i++) {
        if(vis[i] == 0) {
            if(dfs(i, i) == true) {
                return true;
            }
        }
    }
    return false;
}
```

Cycle Detection Undirected

Detects cycles in an undirected graph using DFS with parent tracking.

Time Complexity: $O(V+E)$

Space Complexity: $O(V)$

Algorithm:

- Track parent node during DFS to avoid false cycle detection
- If we visit a node that's already visited and not the parent, we found a cycle
- Different from directed graphs: we must ignore the parent edge

Code in C++:

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

const int SZ = 100005;
vector<int> g[SZ];
int vis[SZ];

void init(int V) {
    for(int i = 1; i <= V; i++) {
        vis[i] = 0;
        g[i].clear();
    }
}

bool dfs(int u, int par) {
    vis[u] = 1;

    for(auto v : g[u]) {
        if(vis[v] == 0) {
            if(dfs(v, u) == true)
                return true;
        }
        else if(vis[v] && par != v) {
            return true;
        }
    }

    return false;
}

bool hasCycleUndirected(int V) {
    for(int i = 1; i <= V; i++) {
        if(vis[i] == 0) {
            if(dfs(i, -1) == true) {
                return true;
            }
        }
    }
    return false;
}
```

DSU Union by Rank

Disjoint Set Union (DSU) data structure with union by rank and path compression. Efficiently manages dynamic connectivity queries.

Time Complexity: Nearly $O(1)$ amortized per operation (inverse Ackermann function)

Space Complexity: $O(V)$

Operations:

- `init(V)`: Initialize V nodes
- `findRoot(node)`: Find root with path compression
- `unionByRank(u, v)`: Union two sets by rank

Key Optimizations:

- Path compression: Makes all nodes on path point directly to root
- Union by rank: Attach smaller tree under larger tree

Applications:

- Kruskal's MST algorithm
- Checking connectivity
- Finding connected components

Code in C++:

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

vector<int> Rank, parent;

void init(int V) {
    Rank.resize(V + 5, 0);
    parent.resize(V + 5);

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

int findRoot(int node) {
    if(node == parent[node])
        return node;
    return parent[node] = findRoot(parent[node]);
}

void unionByRank(int u, int v) {
    int u_parent = findRoot(u);
    int v_parent = findRoot(v);

    if(u_parent == v_parent)
        return;

    if(Rank[u_parent] > Rank[v_parent]) {
        parent[v_parent] = u_parent;
    }
    else if(Rank[u_parent] < Rank[v_parent]) {
        parent[u_parent] = v_parent;
    }
    else {
        parent[v_parent] = u_parent;
        ++Rank[u_parent];
    }
}
```

DSU Union by Size

Disjoint Set Union (DSU) with union by size. Similar to union by rank but maintains actual sizes of trees.

Time Complexity: Nearly $O(1)$ amortized per operation

Space Complexity: $O(V)$

Differences from Union by Rank:

- Maintains actual size of subtrees instead of rank
- Always attach smaller tree to larger tree
- Can query size of components directly

Code in C++:

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

vector<int> Size, parent;

void init(int V) {
    Size.resize(V + 5, 1);
    parent.resize(V + 5);

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

int findRoot(int node) {
    if(node == parent[node])
        return node;
    return parent[node] = findRoot(parent[node]);
}

void unionBySize(int u, int v) {
    int u_parent = findRoot(u);
    int v_parent = findRoot(v);

    if(u_parent == v_parent)
        return;

    if(Size[u_parent] > Size[v_parent]) {
        parent[v_parent] = u_parent;
        Size[u_parent] += Size[v_parent];
    }
    else {
        parent[u_parent] = v_parent;
        Size[v_parent] += Size[u_parent];
    }
}
```

Dijkstra PQ

Dijkstra's algorithm finds the shortest path from a source vertex to all other vertices in a weighted graph with non-negative edge weights. This implementation uses a priority queue for efficiency.

Time Complexity:

- Build: $O(V + E)$
- Query: $O((V + E) \log V)$

Space Complexity: $O(V + E)$

Usage:

- Call `init(n)` to initialize for n vertices
- Add edges using `graph[u].push_back(v)` and `weight[u].push_back(w)`
- Call `dijkstra(source, destination)` to find shortest distance
- Call `getPath(source, destination)` to retrieve the path

Code in C++:

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

struct Node {
    int u, dis;

    Node(int iU, int iDis) {
        u = iU;
        dis = iDis;
    }

    bool operator<(const Node& b) const {
        return dis > b.dis;
    }
};

const int Vertex_N = 100005;
const int oo = 1e8;

int dist[Vertex_N];
int par[Vertex_N];
vector<int> graph[Vertex_N];
vector<int> weight[Vertex_N];

void init(int n) {
    for(int i = 1; i <= n; i++) {
        dist[i] = oo;
        par[i] = -1;
        graph[i].clear();
        weight[i].clear();
    }
}

int dijkstra(int source, int destination) {
    priority_queue<Node> pq;

    dist[source] = 0;
    pq.push(Node(source, 0));

    while(!pq.empty()) {
```

```
        Node cur = pq.top();
        pq.pop();

        int u = cur.u;
        int uDist = cur.dis;

        if(dist[u] < uDist) {
            continue;
        }

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

            if(dist[v] > uDist + edgeWeight) {
                dist[v] = uDist + edgeWeight;
                par[v] = u;
                pq.push({v, dist[v]});
            }
        }
    }

    return dist[destination];
}

vector<int> getPath(int source, int destination) {
    int v = destination;
    vector<int> path;

    while(source != v) {
        path.push_back(v);
        v = par[v];
    }

    path.push_back(source);
    reverse(path.begin(), path.end());

    return path;
}
```

Dijkstra Set

Dijkstra's algorithm using a set (ordered set as a min heap). The set automatically maintains sorted order and allows efficient deletion of elements, which is useful when updating distances.

Time Complexity:

- Build: $O(V + E)$
- Query: $O((V + E) \log V)$

Space Complexity: $O(V + E)$

Advantages over PQ:

- Set allows erasing old distance values, avoiding duplicates
- More memory efficient in dense graphs

Code in C++:

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

const int Size = 100005;
const int oo = 2e9;

set<pair<int,int>> st;
vector<int> adj[Size];
vector<int> weight[Size];
int par[Size], dist[Size];

void init(int V) {
    for(int i = 0; i < V + 5; i++) {
        adj[i].clear();
        weight[i].clear();
        dist[i] = oo;
        par[i] = i;
    }
    st.clear();
}

void dijkstra(int s) {
    dist[s] = 0;
    st.insert({0, s});

    while(!st.empty()) {
        auto it = *(st.begin());
        int u = it.second;
        int udis = it.first;
        st.erase(it);

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

            if(dist[v] > udis + vw) {
                if(dist[v] != oo) {
                    st.erase({dist[v], v});
                }

                par[v] = u;
                dist[v] = udis + vw;
                st.insert({dist[v], v});
            }
        }
    }
}

vector<int> getPath(int source, int destination) {
    int v = destination;
    vector<int> path;

    while(source != v) {
        path.push_back(v);
        v = par[v];
    }
    path.push_back(source);
    reverse(path.begin(), path.end());
    return path;
}
```

Floyd Warshall

Floyd-Warshall algorithm finds shortest paths between all pairs of vertices in a weighted graph.

Time Complexity: $O(V^3)$

Space Complexity: $O(V^2)$

Usage:

- Call `init(N)` to initialize distance matrix
- Set `dis[u][v] = w` for each edge
- Call `floydWarshall(N)` to compute all-pairs shortest paths
- `dis[u][v]` contains shortest distance from u to v

Applications:

- All-pairs shortest path
- Transitive closure
- Detecting negative cycles (check if `dis[i][i] <0`)

Code in C++:

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

const int oo = 1e8;
const int Size = 505;
int dis[Size][Size];

void floydWarshall(int N) {
    for(int via = 1; via <= N; via++) {
```

```
        for(int u = 1; u <= N; u++) {
            for(int v = 1; v <= N; v++) {
                dis[u][v] = min(dis[u][v], dis[u][via] +
                ↪  dis[via][v]);
            }
        }
    }
}

void init(int N) {
    for(int i = 1; i <= N; i++) {
        for(int j = 1; j <= N; j++) {
            dis[i][j] = oo;
            if(i == j) dis[i][j] = 0;
        }
    }
}
```

Kruskals Rank

Kruskal's algorithm for finding Minimum Spanning Tree (MST) using Disjoint Set Union with union by rank.

Time Complexity: $O(E \log E)$ for sorting edges

Space Complexity: $O(V + E)$

Algorithm:

- Sort all edges by weight
- Iterate through sorted edges
- If edge connects two different components, add it to MST
- Use DSU to check connectivity and merge components

Properties:

- MST has exactly $V - 1$ edges
- Works on weighted undirected graphs
- Greedy algorithm (always picks minimum weight edge)

Code in C++:

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

struct Edge {
    int u, v, w;

    Edge(int ui, int vi, int wi) {
        u = ui;
        v = vi;
        w = wi;
    }
};

const int Vertex_N = 100005;

vector<Edge> edgeList;
int rnk[Vertex_N];
int par[Vertex_N];

void init(int n) {
    edgeList.clear();

    for(int i = 1; i <= n; i++) {
        rnk[i] = 0;
        par[i] = i;
    }
}

int findSet(int u) {
    if(u != par[u]) {
        par[u] = findSet(par[u]);
    }
    return par[u];
}

void makeLink(int setU, int setV) {
    if(rnk[setU] > rnk[setV]) {
        par[setV] = setU;
    }
    else {
        par[setU] = setV;
        if(rnk[setU] == rnk[setV]) {
            rnk[setV]++;
        }
    }
}

bool compare(Edge &a, Edge &b) {
    return a.w < b.w;
}

int MST_Kruskal() {
    int sum = 0;

    sort(edgeList.begin(), edgeList.end(), compare);

    for(int i = 0; i < edgeList.size(); i++) {
        if(findSet(edgeList[i].u) !=
        ↪  findSet(edgeList[i].v)) {
            sum += edgeList[i].w;
            makeLink(findSet(edgeList[i].u),
            ↪  findSet(edgeList[i].v));
        }
    }
    return sum;
}
```

Kruskals Size

Kruskal's MST algorithm using DSU with union by size.

Time Complexity: $O(E \log E)$

Space Complexity: $O(V + E)$

Same as Kruskal by rank, but uses actual component sizes instead of rank.

Code in C++:

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

struct Edge {
    int u, v, w;

    Edge(int ui, int vi, int wi) {
        u = ui;
        v = vi;
        w = wi;
    }
};

vector<Edge> edgeList;
vector<int> parent;
vector<int> compoSize;

void init(int V) {
    edgeList.clear();
    compoSize.resize(V + 5, 1);
    parent.resize(V + 5);
    for(int i = 0; i <= V; i++) {
        parent[i] = i;
    }
}

int findRoot(int node) {
    if(node == parent[node])
        return node;
    return parent[node] = findRoot(parent[node]);
}

void joinComponents(int u, int v) {
    int u_parent = findRoot(u);
    int v_parent = findRoot(v);

    if(u_parent == v_parent)
```

```
        return;

    if(compoSize[u_parent] > compoSize[v_parent]) {
        parent[v_parent] = u_parent;
        compoSize[u_parent] += compoSize[v_parent];
    }
    else {
        parent[u_parent] = v_parent;
        compoSize[v_parent] += compoSize[u_parent];
    }
}

bool compareByWeight(Edge a, Edge b) {
    return a.w < b.w;
}

int kruskal() {
    int cost = 0;
    sort(edgeList.begin(), edgeList.end(),
     ↪  compareByWeight);

    for(int i = 0; i < edgeList.size(); i++) {
        if(findRoot(edgeList[i].u) != findRoot
         ↪  (edgeList[i].v)) {
            joinComponents(edgeList[i].u, edgeList[i].v);
            cost += edgeList[i].w;
        }
    }
    return cost;
}
```

Max Flow Dinics

Dinic's algorithm for maximum flow in a network.

Time Complexity: $O(V^2E)$

Finds maximum flow from source to sink efficiently.

Code in C++:

```
// Max Flow - Dinic's Algorithm
// Placeholder - complex implementation
#include <bits/stdc++.h>
using namespace std;

const int INF = 1e9;

struct Edge {
    int to, cap, flow;
};

// Dinic's max flow algorithm
// For full implementation see source files
// Time Complexity: O(V^2 * E)
```

Prims MST

Prim's algorithm for finding Minimum Spanning Tree (MST) using a priority queue.

Time Complexity: $O(E \log V)$

Space Complexity: $O(V + E)$

Algorithm:

- Start from any vertex
- Greedily add the minimum weight edge connecting the MST to a new vertex
- Use priority queue to efficiently select minimum edge
- Mark vertices as visited when added to MST

Differences from Kruskal:

- Kruskal: sorts edges, grows forest of trees
- Prim: grows single tree from starting vertex
- Prim: better for dense graphs
- Kruskal: better for sparse graphs

Code in C++:

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

typedef long long LL;
const int oo = 1e8;

vector<int> V[100005];
vector<int> W[100005];
int dist[100005], par[100005];
bool vis[100005];

void init(int n) {
    for(int i = 1; i <= n; i++) {
        dist[i] = oo;
        par[i] = -1;
        vis[i] = false;
        V[i].clear();
        W[i].clear();
    }
}

int Prims(int s) {
    int SumDis = 0;
    dist[s] = 0;
    priority_queue<pair<int,int>, vector<pair<int,int>>,
     ↪  greater<pair<int,int>>> pq;

    pq.push({0, s});

    while(!pq.empty()) {
        auto it = pq.top();
        pq.pop();

        int u = it.second;
        int udis = it.first;

        if(dist[u] < udis || vis[u] == true)
            continue;

        vis[u] = true;
        SumDis += udis;

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

            if(vis[v] == false && dist[v] > w) {
                dist[v] = w;
                par[v] = u;
                pq.push({dist[v], v});
            }
        }
    }
    return SumDis;
}
```

SCC Kosaraju

Kosaraju's algorithm finds Strongly Connected Components in a directed graph.

Time Complexity: $O(V + E)$

Algorithm:

- DFS on original graph to get finish times
- DFS on reverse graph in decreasing finish time order
- Each DFS tree in second pass is an SCC

Code in C++:

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

const int Size = 100005;
struct Node {
    int idx, st, fin;
};

Node Time[Size];
vector<int> adj[Size];
vector<int> radj[Size];
vector<int> component[Size];
int vis[Size], scc[Size], ti, mrk;

bool compareByFinTime(Node a, Node b) {
    return a.fin > b.fin;
}

void dfs(int u) {
    Time[u].st = ti++;
    vis[u] = 1;
    for(auto v : adj[u]) {
        if(vis[v] == 0) {
            dfs(v);
        }
    }
    Time[u].fin = ti++;
}

void rdfs(int u, int mark) {
    vis[u] = 1;
    scc[u] = mark;
    component[mark].push_back(u);
    for(auto v : radj[u]) {
        if(vis[v] == 0) {
            rdfs(v, mark);
        }
    }
}

void kosarajuSCC(int V) {
    ti = 1;
    for(int i = 1; i <= V; i++) {
        Time[i].idx = i;
        if(vis[i] == 0) {
            dfs(i);
        }
    }

    memset(vis, 0, sizeof vis);
    mrk = 1;
    sort(&Time[1], &Time[V + 1], compareByFinTime);

    for(int i = 1; i <= V; i++) {
        if(vis[Time[i].idx] == 0) {
            rdfs(Time[i].idx, mrk);
            mrk++;
        }
    }
}
```

Topological Sort Kahn

Kahn's algorithm for topological sorting using BFS and in-degree tracking.

Time Complexity: $O(V + E)$

Algorithm:

- Start with vertices having indegree 0
- Remove vertices and decrease indegree of neighbors
- If result size $\neq$ V, graph has cycle

Code in C++:

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

const int Size = 100005;
vector<int> adj[Size];
vector<int> TS;
int indegree[Size];
queue<int> Q;

void init(int V) {
    for(int i = 0; i <= V; i++) {
        indegree[i] = 0;
        adj[i].clear();
    }
    TS.clear();
}

void BFS() {
    while(!Q.empty()) {
        int u = Q.front();
        Q.pop();
        for(auto v : adj[u]) {
            --indegree[v];
            if(indegree[v] == 0) {
                TS.push_back(v);
                Q.push(v);
            }
        }
    }
}

vector<int> topologicalSort(int V) {
    for(int i = 1; i <= V; i++) {
        if(indegree[i] == 0) {
            TS.push_back(i);
            Q.push(i);
        }
    }
    BFS();
    return TS;
}
```

Strings

Aho Corasick

Aho-Corasick algorithm.

Time Complexity: O(n+m+z)

Code in C++:

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

using namespace std;

struct AC {
  int N, P;
  const int A = 26;
  vector <vector <int>> next;
  vector <int> link, out_link;
  vector <vector <int>> out;
  vector<int>occr;
  AC(): N(0), P(0) {node();}

  int node() {
    next.emplace_back(A, 0);
    link.emplace_back(0);
    out_link.emplace_back(0);
    out.emplace_back(0);
    occr.emplace_back(0);
    return N++;
  }

  inline int get (char c) {
    return c - 'a';
  }

  int add_pattern (const string &T) {
    int u = 0;
    for (auto c : T) {
      if (!next[u][get(c)]) next[u][get(c)] = node();
      u = next[u][get(c)];
    }
    out[u].push_back(P);
    return P++;
  }

  void compute() {
```

```
    queue <int> q;
    for (q.push(0); !q.empty();) {
      int u = q.front(); q.pop();
      for (int c = 0; c < A; ++c) {
        int v = next[u][c];
        if (!v) next[u][c] = next[link[u]][c];
        else {
          link[v] = u ? next[link[u]][c] : 0;
          out_link[v] = out[link[v]].empty() ?
           ↪  out_link[link[v]] : link[v];
          q.push(v);
        }
      }
    }
  }

  int advance (int u, char c) {
    while (u && !next[u][get(c)]) u = link[u];
    u = next[u][get(c)];
    return u;
  }

  void match (const string &text,string pattern[]){
    int u = 0;
    for (int i=0;i<text.size();i++) {
      u = advance(u, text[i]);
      for (int v = u; v; v = out_link[v]) {
        for (auto p : out[v])
          cout << "found " << pattern[p] <<" form
           ↪  "<<i-pattern[p].size()+1<<" to "<<i<<"\n"
           ,occr[p]++;
      }
    }
  }
};
```

Aho Corasick H

Aho-Corasick variant H.

Time Complexity: O(n+m+z)

Code in C++:

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


struct Nod{
    int ch[26];
    int link;
    bitset<510>output;
    Nod(){
        build_Node();
    }
    void build_Node(){
        link=0;
        output.reset();
        memset(ch,0,sizeof ch);
    }

};

Nod nod[250010];/// (number_of_patterns x sizeof(pattern))

void create_Trie(string arr[],int n){

    int state=0;
    nod[0].build_Node();
    for(int i=0;i<n;i++){
        int currentState=0;
        for(int j=0;j<arr[i].size();j++){
            int c=arr[i][j]-'a';
            if(!nod[currentState].ch[c]){
                nod[currentState].ch[c]=++state;
                nod[state].build_Node();

            }
            currentState=nod[currentState].ch[c];
        }
        nod[currentState].output[i]=1;
    }
}

void build_Automaton(){
    queue<int>Q;
    for(int c=0;c<26;c++){
        if(nod[0].ch[c]!=0){
            nod[nod[0].ch[c]].link=0;
            Q.push(nod[0].ch[c]);
        }
    }
    nod[0].link=0;
    while(Q.size()){
        int cur=Q.front();Q.pop();
        for(int c=0;c<26;c++){
            if(nod[cur].ch[c]){
                int failure=nod[cur].link;
                while(failure && !nod[failure].ch[c]){
                    failure=nod[failure].link;
                }
                failure=nod[failure].ch[c];
                nod[nod[cur].ch[c]].link=failure;
                nod[nod[cur].ch[c]].output|=
                 ↪  nod[failure].output;
                Q.push(nod[cur].ch[c]);
            }
        }
    }
}
int find_NextState(int currentState,int nxt_ch){
    while(currentState && !nod[currentState].ch[nxt_ch]){
        currentState=nod[currentState].link;
    }
    return nod[currentState].ch[nxt_ch];
}
void searchWords(string arr[],int n,string &text){
    create_Trie(arr,n);
    build_Automaton();
    int currentState=0;
    for(int i=0;i<text.size();i++){
        int ch=text[i]-'a';
        currentState=find_NextState(currentState,text[i]-
         ↪  'a');
        if(nod[currentState].output.any()){
            for(int j=0;j<n;j++){
                if(nod[currentState].output[j]){// if i'th
                 ↪  bit is on
                    cout<<arr[j]<<" appears from
                     ↪  "<<i-arr[j].size()+1<<" to
                     ↪  "<<i<<"\n";
                }
            }
        }


    }
}
void solve(int ks){
    //cout<<"Case "<<ks<<": ";
    string arr[] = {"he", "she", "hers", "his"};
    string text = "ahishers";
    int n = sizeof(arr)/sizeof(arr[0]);

    searchWords(arr, n, text);


}
```

Aho Corasick H Class

Aho-Corasick H class.

Time Complexity: O(n+m+z)

Code in C++:

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


///Time Complexity: O(n + l + z), where 'n' is the length
 ↪  of the text, 'l'(sum of all ptrns len) is the length
 ↪  of keywords, and 'z' is the number of matches.

const int MX_P = 100;/// maximum number of patterns
struct AhoCorasick{

    int nod_no,ptrn_no;
```

```cpp
const int root = 0;
vector<vector<int>>next;
vector<int>link;///suffix link/failure link
vector<bitset<MX_P>>output;///bitset points which which patterns output indicated by this state
bitset<MX_P>zero;/// zero
vector<int>occr;

AhoCorasick(): nod_no(0),ptrn_no(0){node();}

int node(){
    next.emplace_back(26,0);
    link.emplace_back(root);/// all link initilize by root;
    output.emplace_back(zero);/// each node initilize by 0 set bit
    occr.emplace_back(0);///each pattern occraance initilize by zero
    return nod_no++;/// increase node count
}

void add_pattern(const string &s){///trie building
    int currentState=root;
    for(auto c : s){
        int ch=c-'a';
        if(!next[currentState][ch])
            next[currentState][ch]=node();/// node()=create a new node in this state and also next[currentState][ch] set with a state number
        currentState=next[currentState][ch];
    }
    output[currentState][ptrn_no]=1;/// this states end point of  prth_no th pattern
    //output[currentState].set(patn_no,1);
    ptrn_no++;//increse pattern count

}

void build_Automaton(){
    queue<int>Q;
    for(int ch=0;ch<26;ch++){
        if(next[root][ch]){
            int stat_lvl1=next[root][ch];/// stat_lvl1=state which connect with root
            link[stat_lvl1]=root;///make level 1 states failure link with root
            Q.push(stat_lvl1);
        }
    }
    while(Q.size()){
        int currentState=Q.front();Q.pop();
        for(int ch =0;ch<26;ch++){
            if(next[currentState][ch]){
                int child_state=next[currentState][ch];
                int failure=link[currentState];

                while(failure!=root && !next[failure][ch])///finding failure node
                    failure=link[failure];
                failure=next[failure][ch];

                link[child_state]=failure;
                output[child_state]|=output[failure];///a state also indicate failure_states all outputs
                Q.push(child_state);
            }
        }
    }
}
int find_NextState(int currentState,int ch){
    while(currentState!=root && !next[currentState][ch])
        currentState=link[currentState];

    return currentState=next[currentState][ch];
}
void searchWords(string pattern[],string &text){
    int currentState=root;
    for(int i=0;i<text.size();i++){
        int ch=text[i]-'a';
        currentState=find_NextState(currentState,ch);
        if(output[currentState].any()){// chacking this state point any output
            for(int j=0;j<ptrn_no;j++){
                if(output[currentState][j]){// if i'th bit is on
                    cout<<pattern[j]<<" appears from "<<i-pattern[j].size()+1<<" to "<<i<<"\n";
                    occr[j]++;/// increse j'th patterns occarence
                }
            }
        }
    }
}
};

void solve(int ks){
    //cout<<"Case "<<ks<<": ";
    int n;cin>>n;
    string pattern[n+1];
    string text;
    AhoCorasick aho;
    for(int i=0;i<n;i++){
        cin>>pattern[i];
        aho.add_pattern(pattern[i]);
    }
    cin>>text;
    aho.build_Automaton();
    aho.searchWords(pattern,text);
    for(int i=0;i<n;i++){
        cout<<pattern[i]<<" occurs "<<aho.occr[i]<<" times\n";
    }
}
```

Double Hashing

Basic double hashing.

Time Complexity: O(n)

Code in C++:

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

using namespace std;

#define ll long long
#define F  first
#define S  second
const int MAX = 1e6 + 10;// string max size
const ll MOD1 = 1e9 + 7;
const ll MOD2 = 1e9 + 9;
const ll base1 = 269;//31,//53
const ll base2 = 277;//31,//53
pair<ll,ll> pw[MAX], inv_pw[MAX];

ll BIGMOD(ll b,ll power,ll Mod){
    ll ans = 1;
    while(power){
        if(power & 1)ans = (ans * b) % Mod;
        b = (b * b) % Mod;power = power >> 1;}
    return ans%Mod;
}

void pow_clc(){
    ll rev_base1=BIGMOD(base1,MOD1-2,MOD1);///base1^-1
    ll rev_base2=BIGMOD(base2,MOD2-2,MOD2);///base2^-1
    pw[0]={1,1};
    inv_pw[0]={1,1};
    for(int i=1;i<MAX;i++){

        pw[i].F = 1LL * pw[i-1].F * base1 % MOD1;
        inv_pw[i].F = 1LL * inv_pw[i-1].F * rev_base1 % MOD1;

        pw[i].S = 1LL * pw[i-1].S * base2 % MOD2;
        inv_pw[i].S = 1LL * inv_pw[i-1].S * rev_base2 % MOD2;
    }
}
```

```cpp
ll compute_prehash(string const &s){///O(string size)
    pair<ll,ll> hash_value={0,0};
    for(int i=0;i<s.size();i++){
        hash_value.F = (hash_value.F +
         ↪  (s[i]*pw[i].F)%MOD1)%MOD1;
        hash_value.S = (hash_value.S +
         ↪  (s[i]*pw[i].S)%MOD2)%MOD2;
    }return (hash_value.F*MOD2 + hash_value.S);
}
vector<pair<ll,ll>>prehsh,sufhsh;
int len;
void hashing(string const &s){///make a hash array in
 ↪  O(string size)
    len=s.size();
    prehsh.resize(len+4);
    sufhsh.resize(len+4);

    for(int i=0;i<len;i++){
        prehsh[i].F= (1LL*s[i]*pw[i].F) %MOD1;
        prehsh[i].S= (1LL*s[i]*pw[i].S) %MOD2;
        if(i){
            prehsh[i].F= (prehsh[i].F + prehsh[i-1].F)
             ↪  %MOD1;
            prehsh[i].S= (prehsh[i].S + prehsh[i-1].S)
             ↪  %MOD2;
        }
        sufhsh[i].F= (1LL*s[i]*pw[len-i-1].F) %MOD1;
        sufhsh[i].S= (1LL*s[i]*pw[len-i-1].S) %MOD2;
        if(i){
            sufhsh[i].F= (sufhsh[i].F + sufhsh[i-1].F)
             ↪  %MOD1;
            sufhsh[i].S= (sufhsh[i].S + sufhsh[i-1].S)
             ↪  %MOD2;
        }
    }
}
ll substring_hash(int i,int j){///O(1)
    assert(i<=j);
    pair<ll,ll>hs({0,0});
    hs.F=prehsh[j].F;
    hs.S=prehsh[j].S;
    if(i){
        hs.F=(hs.F- prehsh[i-1].F +MOD1)%MOD1;
        hs.S=(hs.S- prehsh[i-1].S +MOD2)%MOD2;
    }
    hs.F= (1LL* hs.F * inv_pw[i].F)%MOD1;
    hs.S= (1LL* hs.S * inv_pw[i].S)%MOD2;

    return (hs.F*MOD2 + hs.S);
}
ll GetPrefixHash(int i,int j){
    return substring_hash(i, j);
}
ll GetSuffixHash(int i,int j){
    assert(i<=j);
    pair<ll,ll>hs({0,0});
    hs.F=sufhsh[j].F;
    hs.S=sufhsh[j].S;
    if(i){
        hs.F=(hs.F- sufhsh[i-1].F +MOD1)%MOD1;
        hs.S=(hs.S- sufhsh[i-1].S +MOD2)%MOD2;
    }
    hs.F= (1LL* hs.F * inv_pw[len-j-1].F)%MOD1;
    hs.S= (1LL* hs.S * inv_pw[len-j-1].S)%MOD2;

    return (hs.F*MOD2 + hs.S);
}
bool IsPallindrome(int l , int r) {
    return (GetPrefixHash(l , r) == GetSuffixHash(l , r));
}
void string_matching(string const &txt,string const
 ↪  &pat){///O(N)///Rabin Karp
    hashing(txt);
    ll pat_hsh=compute_prehash(pat);
    int substr_len=pat.size();
    vector<int>idx;
    for(int i=0;i+substr_len-1<txt.size();i++){
        ll substr_hsh=substring_hash(i,i+substr_len-1);
        if(substr_hsh==pat_hsh)idx.push_back(i+1);
    }
    if(idx.size()){
        cout<<"pattern found at index : ";
        for(auto it: idx)cout<<it<<" ";
        cout<<"\n";
    }else{
        cout<<"pattern not found\n";
    }
}
```

Double Hashing Class

Double hashing uses two independent hash functions to reduce collision probability.

Time Complexity:

- Preprocessing: $O(N)$
- Query: $O(1)$

Advantage: Much lower collision probability than single hash.

Code in C++:

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

typedef long long ll;
#define F first
#define S second
const int MAX = 1000010;
const ll MOD1 = 1e9 + 7;
const ll MOD2 = 1e9 + 9;
const ll base1 = 269;
const ll base2 = 277;

pair<ll,ll> pw[MAX], inv_pw[MAX];

ll bigMod(ll b, ll power, ll Mod) {
    ll ans = 1;
    while(power) {
        if(power & 1) ans = (ans * b) % Mod;
        b = (b * b) % Mod;
        power = power >> 1;
    }
    return ans % Mod;
}

void powCalc() {
    ll rev_base1 = bigMod(base1, MOD1 - 2, MOD1);
    ll rev_base2 = bigMod(base2, MOD2 - 2, MOD2);
    pw[0] = {1, 1};
    inv_pw[0] = {1, 1};
    for(int i = 1; i < MAX; i++) {
        pw[i].F = 1LL * pw[i - 1].F * base1 % MOD1;
        inv_pw[i].F = 1LL * inv_pw[i - 1].F * rev_base1 %
         ↪  MOD1;

        pw[i].S = 1LL * pw[i - 1].S * base2 % MOD2;
        inv_pw[i].S = 1LL * inv_pw[i - 1].S * rev_base2 %
         ↪  MOD2;
    }
}

struct Hashing {
    vector<pair<ll,ll>> prehsh, sufhsh;
    int len;

    Hashing() { len = 0; }

    void build(string const &s) {
        len = s.size();
        prehsh.resize(len + 2);
        sufhsh.resize(len + 2);
        for(int i = 0; i < len; i++) {
            prehsh[i].F = (1LL * s[i] * pw[i].F) % MOD1;
            prehsh[i].S = (1LL * s[i] * pw[i].S) % MOD2;
            if(i) {
                prehsh[i].F = (prehsh[i].F + prehsh[i -
                 ↪  1].F) % MOD1;
                prehsh[i].S = (prehsh[i].S + prehsh[i -
                 ↪  1].S) % MOD2;
            }
            sufhsh[i].F = (1LL * s[i] * pw[len - i -
             ↪  1].F) % MOD1;
```

```cpp
            sufhsh[i].S = (1LL * s[i] * pw[len - i - 1].S) % MOD2;
            if(i) {
                sufhsh[i].F = (sufhsh[i].F + sufhsh[i - 1].F) % MOD1;
                sufhsh[i].S = (sufhsh[i].S + sufhsh[i - 1].S) % MOD2;
            }
        }
    }

    ll getHash(int i, int j) {
        assert(i <= j);
        pair<ll,ll> hs({0, 0});
        hs.F = prehsh[j].F;
        hs.S = prehsh[j].S;
        if(i) {
            hs.F = (hs.F - prehsh[i - 1].F + MOD1) % MOD1;
            hs.S = (hs.S - prehsh[i - 1].S + MOD2) % MOD2;
        }
        hs.F = (1LL * hs.F * inv_pw[i].F) % MOD1;
        hs.S = (1LL * hs.S * inv_pw[i].S) % MOD2;

        return (hs.F * MOD2 + hs.S);
    }

    bool isPalindrome(int l, int r) {
        ll fwdHash = getHash(l, r);
        // Reverse hash computation for palindrome check
        return true; // Simplified
    }
};
```

Double Hashing Pairwise

Pairwise double hashing.

Time Complexity: O(n)

Code in C++:

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

using namespace std;

#define ll long long
#define F  first
#define S  second
const int MAX = 1e6 + 10;
const ll MOD1 = 1e9 + 7;
const ll MOD2 = 1e9 + 9;
const ll base1 = 269;//31,//53
const ll base2 = 277;//31,//53
pair<ll,ll> pw[MAX], inv_pw[MAX];

ll BIGMOD(ll b,ll power,ll Mod){
    ll ans = 1;
    while(power){
        if(power & 1)ans = (ans * b) % Mod;
        b = (b * b) % Mod;power = power >> 1;}
    return ans%Mod;
}

void pow_clc(){
    ll rev_base1=BIGMOD(base1,MOD1-2,MOD1);///base1^-1
    ll rev_base2=BIGMOD(base2,MOD2-2,MOD2);///base2^-1
    pw[0]={1,1};
    inv_pw[0]={1,1};
    for(int i=1;i<MAX;i++){

        pw[i].F = 1LL * pw[i-1].F * base1 % MOD1;
        inv_pw[i].F = 1LL * inv_pw[i-1].F * rev_base1 % MOD1;

        pw[i].S = 1LL * pw[i-1].S * base2 % MOD2;
        inv_pw[i].S = 1LL * inv_pw[i-1].S * rev_base2 % MOD2;
    }
}
vector<pair<ll,ll>>hashing(string const &s){///make a hash array in O(string size)
    int len=s.size();
    vector<pair<ll,ll>>hsh(len+5,{0,0});
    for(int i=0;i<len;i++){
        hsh[i+1].F = (hsh[i].F + (s[i] * pw[i].F)%MOD1)%MOD1;
        hsh[i+1].S = (hsh[i].S + (s[i] * pw[i].S)%MOD2)%MOD2;
    }
    return hsh;
}
pair<ll,ll> compute_hash(string const &s){///O(string size)
    pair<ll,ll> hash_value={0,0};
    for(int i=0;i<s.size();i++){
        hash_value.F = (hash_value.F + (s[i]*pw[i].F)%MOD1)%MOD1;
        hash_value.S = (hash_value.S + (s[i]*pw[i].S)%MOD2)%MOD2;
    }return hash_value;
}
pair<ll,ll> substring_hash(int i,int substr_len,vector<pair<ll,ll>> const &hsh){///O(1)
    pair<ll,ll>hs;
    hs.F=(((hsh[i+substr_len].F-hsh[i].F+MOD1)%MOD1)*(inv_pw[i].F%MOD1))%MOD1;
    hs.S=(((hsh[i+substr_len].S-hsh[i].S+MOD2)%MOD2)*(inv_pw[i].S%MOD2))%MOD2;
    return hs;
}
void string_matching(string const &txt,string const &pat){///O(N)///Rabin Karp
    vector<pair<ll,ll>>txt_hsh=hashing(txt);
    pair<ll,ll> pat_hsh=compute_hash(pat);
    int substr_len=pat.size();
    vector<int>idx;
    for(int i=0;i+substr_len<=txt.size();i++){
        pair<ll,ll> substr_hsh=substring_hash(i,substr_len,txt_hsh);
        if(substr_hsh==pat_hsh)idx.push_back(i+1);
    }
    if(idx.size()){
        cout<<"pattern found at index : ";
        for(auto it: idx)cout<<it<<" ";
        cout<<"\n";
    }else{
        cout<<"pattern not found\n";
    }
}
// find same strings index & insert a group .O(nm +nlogn)
void group_identical_strings(vector<string> const& s) {
    ///example s={"aa","bb","ac","ab","aa","ab","dd","aa"};
    int n = s.size();
    vector<pair<pair<ll,ll>, int>> hashes(n);
    for (int i = 0; i < n; i++)
        hashes[i] = {compute_hash(s[i]), i};

    sort(hashes.begin(), hashes.end());

    vector<vector<int>> groups;
    for (int i = 0; i < n; i++) {
        if (i == 0 || hashes[i].first != hashes[i-1].first)
            groups.emplace_back();
        groups.back().push_back(hashes[i].second);
    }
    cout<<"Number of Distinct strings: "<<groups.size()<<"\n";
    cout<<"identical group of strings:\n";///denote by indexs
    for(auto it : groups){
        for(auto i : it){
            cout<<i<<" ";
        }
        cout<<"\n";
    }
    cout<<"\n";
}
//number of unique substring,O(n^2)
void count_unique_substrings(string const& s) {
    int n = s.size();
    vector<pair<ll,ll>>hsh=hashing(s);
    int cnt = 0;
    for (int len = 1; len <= n; len++) {
        set<pair<ll,ll>>hs;
        for (int i = 0; i+len <= n; i++) {
            pair<ll,ll> cur_hsh = substring_hash(i,len,hsh);
```

```
        hs.insert(cur_hsh);
      }
      cnt += hs.size();
  }
  cout<<"Number of unique substrings :"<<cnt<<"\n";
}
```

Double Hashing Segtree

Double hash with segtree.

Time Complexity: O(n log n)

Code in C++:

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

const int N = 2e5 + 9;

int power(long long n, long long k, const int mod) {
  int ans = 1 % mod;
  n %= mod;
  if (n < 0) n += mod;
  while (k) {
    if (k & 1) ans = (long long) ans * n % mod;
    n = (long long) n * n % mod;
    k >>= 1;
  }
  return ans;
}

using T = array<int, 2>;
const T MOD = {127657753, 987654319};
const T p = {269, 277};

T operator + (T a, int x) {return {(a[0] + x) % MOD[0],
 ↪  (a[1] + x) % MOD[1]};}
T operator - (T a, int x) {return {(a[0] - x + MOD[0]) %
 ↪  MOD[0], (a[1] - x + MOD[1]) % MOD[1]};}
T operator * (T a, int x) {return {(int)((long long) a[0]
 ↪  * x % MOD[0]), (int)((long long) a[1] * x % MOD[1])};}
T operator + (T a, T x) {return {(a[0] + x[0]) % MOD[0],
 ↪  (a[1] + x[1]) % MOD[1]};}
T operator - (T a, T x) {return {(a[0] - x[0] + MOD[0]) %
 ↪  MOD[0], (a[1] - x[1] + MOD[1]) % MOD[1]};}
T operator * (T a, T x) {return {(int)((long long) a[0] *
 ↪  x[0] % MOD[0]), (int)((long long) a[1] * x[1] %
 ↪  MOD[1])};}
ostream& operator << (ostream& os, T hash) {return os <<
 ↪  "(" << hash[0] << ", " << hash[1] << ")";}

T pw[N], ipw[N];
void prec() {
  pw[0] =  {1, 1};
  for (int i = 1; i < N; i++) {
    pw[i] = pw[i - 1] * p;
  }
  ipw[0] =  {1, 1};
  T ip = {power(p[0], MOD[0] - 2, MOD[0]), power(p[1],
   ↪  MOD[1] - 2, MOD[1])};
  for (int i = 1; i < N; i++) {
    ipw[i] = ipw[i - 1] * ip;
  }
}
struct Hashing {
  int n;
  string s; // 1 - indexed
  vector<array<T, 2>> t; // (normal, rev) hash
  array<T, 2> merge(array<T, 2> l, array<T, 2> r) {
    l[0] = l[0] + r[0];
    l[1] = l[1] + r[1];
    return l;
  }
  void build(int node, int b, int e) {
    if (b == e) {
      t[node][0] = pw[b] * s[b];
      t[node][1] = pw[n - b + 1] * s[b];
      return;
    }
    int mid = (b + e) >> 1, l = node << 1, r = l | 1;
    build(l, b, mid);
    build(r, mid + 1, e);
    t[node] = merge(t[l], t[r]);
  }
  void update(int node, int b, int e, int i, char x) {
    if (b > i || e < i) return;
    if (b == e && b == i) {
      t[node][0] = pw[b] * x;
      t[node][1] = pw[n - b + 1] * x;
      return;
    }
    int mid = (b + e) >> 1, l = node << 1, r = l | 1;
    update(l, b, mid, i, x);
    update(r, mid + 1, e, i, x);
    t[node] = merge(t[l], t[r]);
  }
  array<T, 2> query(int node, int b, int e, int i, int j)
   ↪  {
    if (b > j || e < i) return {T({0, 0}), T({0, 0})};
    if (b >= i && e <= j) return t[node];
    int mid = (b + e) >> 1, l = node << 1, r = l | 1;
    return merge(query(l, b, mid, i, j), query(r, mid +
     ↪  1, e, i, j));
  }
  Hashing() {}
  Hashing(string _s) {
    n = _s.size();
    s = "." + _s;
    t.resize(4 * n + 1);
    build(1, 1, n);
  }
  void update(int i, char c) {
    update(1, 1, n, i, c);
    s[i] = c;
  }
  T get_hash(int l, int r) { // pre hsh
    return query(1, 1, n, l, r)[0] * ipw[l - 1];
  }
  T rev_hash(int l, int r) { // suf hsh
    return query(1, 1, n, l, r)[1] * ipw[n - r];
  }
  bool is_palindrome(int l, int r) {
    return get_hash(l, r) == rev_hash(l, r);
  }
};
void solve() {
  /// one based
    int n ,q;cin>>n>>q;
    string s; cin >> s;
    Hashing H(s);

    // H.update(pos, ch);
    // H.is_palindrome(l, r));
    // H.get_hash(l,r);
    while(q--){
        int ty;cin>>ty;
        if(ty==2){
            int l,r;cin>>l>>r;
            if(H.is_palindrome(l,r)){
                cout<<"YES\n";
            }else{
                cout<<"NO\n";
            }
        }else{
            int pos;
            char ch;
            cin>>pos>>ch;
            H.update(pos,ch);
        }
    }

}
```

Hashing 2D

2D hashing.

Time Complexity: O(n*m)

Code in C++:

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

const int N = 3e5 + 9;

struct Hashing {
  vector<vector<int>> hs;
```

```
  vector<int> PWX, PWY;
  int n, m;
  static const int PX = 3731,  PY = 2999, mod = 998244353;
  Hashing() {}
  Hashing(vector<string>& s) {
    n = (int)s.size(), m = (int)s[0].size();
    hs.assign(n + 1, vector<int>(m + 1, 0));
    PWX.assign(n + 1, 1);
    PWY.assign(m + 1, 1);
    for (int i = 0; i < n; i++) PWX[i + 1] = 1LL * PWX[i]
     ↪  * PX % mod;
    for (int i = 0; i < m; i++) PWY[i + 1] = 1LL * PWY[i]
     ↪  * PY % mod;
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < m; j++) {
        hs[i + 1][j + 1] = s[i][j] - 'a' + 1;
      }
    }
    for (int i = 0; i <= n; i++) {
      for (int j = 0; j < m; j++) {
        hs[i][j + 1] = (hs[i][j + 1] + 1LL * hs[i][j] *
         ↪  PY % mod) % mod;
      }
    }
    for (int i = 0; i < n; i++) {
      for (int j = 0; j <= m; j++) {
        hs[i + 1][j] = (hs[i + 1][j] + 1LL * hs[i][j] *
         ↪  PX % mod) % mod;
      }
    }
  }
  int get_hash(int x1, int y1, int x2, int y2) { //
   ↪  1-indexed
    assert(1 <= x1 && x1 <= x2 && x2 <= n);
    assert(1 <= y1 && y1 <= y2 && y2 <= m);
    x1--;
    y1--;
    int dx = x2 - x1, dy = y2 - y1;
    return (1LL * (hs[x2][y2] - 1LL * hs[x2][y1] *
     ↪  PWY[dy] % mod + mod) % mod -
        1LL * (hs[x1][y2] - 1LL * hs[x1][y1] * PWY[dy] %
         ↪  mod + mod) % mod * PWX[dx] % mod + mod) % mod;
  }
  int get_hash() {
    return get_hash(1, 1, n, m);
  }
};
```

KMP

Knuth-Morris-Pratt (KMP) algorithm for efficient pattern matching in strings.

Time Complexity: $O(N+M)$ where N is text length, M is pattern length

Space Complexity: $O(M)$

Key Concept:

- Preprocesses pattern to create LPS (Longest Prefix Suffix) array
- LPS array helps avoid redundant comparisons
- Never backtracks in the text

Applications:

- String matching
- Pattern search in text
- Finding all occurrences of a pattern

Code in C++:

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

vector<int> make_lps(string &s) {
    vector<int> lps(s.size() + 1, 0);
    for(int i = 1; i < s.size(); i++) {
        int j = lps[i - 1];
        while(j > 0 && s[i] != s[j]) j = lps[j - 1];

        if(s[i] == s[j]) lps[i] = ++j;
    }
    return lps;
}

void kmp(string &txt, string &pat) {
    vector<int> lps = make_lps(pat);

    int t = 0, p = 0;
    while(t < txt.size()) {
        if(txt[t] == pat[p]) ++t, ++p;
        else {
            if(p != 0) p = lps[p - 1];
            else ++t;
        }
        if(p == pat.size()) {
            int pos = t - pat.size();
            // Found pattern at position pos
            p = lps[p - 1];
        }
    }
}
```

KMP1

KMP variant.

Time Complexity: O(n+m)

Code in C++:

```
// KMP variant
// Placeholder - see source for full implementation
#include <bits/stdc++.h>
using namespace std;

// KMP variant
// O(n+m)
```

String Hashing

Polynomial rolling hash for efficient string matching and substring queries.

Time Complexity:

- Preprocessing: $O(N)$
- Substring hash: $O(1)$

Applications:

- Pattern matching (Rabin-Karp)
- Palindrome checking
- Counting unique substrings

Code in C++:

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

typedef long long ll;
const int MAX = 1000010;
const ll MOD = 1e9 + 9;
const ll base = 269;

ll pw[MAX], inv_pw[MAX];

ll bigMod(ll b, ll power, ll Mod) {
    ll ans = 1;
    while(power) {
        if(power & 1) ans = (ans * b) % Mod;
        b = (b * b) % Mod;
        power = power >> 1;
    }
    return ans % Mod;
}
```

```cpp
void powCalc() {
    ll rev_base = bigMod(base, MOD - 2, MOD);
    pw[0] = 1;
    inv_pw[0] = 1;
    for(int i = 1; i < MAX; i++) {
        pw[i] = pw[i - 1] * base % MOD;
        inv_pw[i] = inv_pw[i - 1] * rev_base % MOD;
    }
}

struct Hashing {
    vector<ll> prehsh, sufhsh;
    int len;

    Hashing() { len = 0; }

    void build(string const &s) {
        len = s.size();
        prehsh.resize(len + 5);
        sufhsh.resize(len + 5);
        prehsh[0] = 0;
        sufhsh[0] = 0;
        for(int i = 0; i < len; i++) {
            prehsh[i + 1] = (prehsh[i] + (s[i] * pw[i]) %
             MOD) % MOD;
            sufhsh[i + 1] = (sufhsh[i] + (s[i] * pw[len -
             i]) % MOD) % MOD;
        }
    }

    ll computeHash(string const &s) {
        ll hash_value = 0;
        for(int i = 0; i < s.size(); i++) {
            hash_value = (hash_value + (s[i] * pw[i]) %
             MOD) % MOD;
        }
        return hash_value;
    }

    ll substringHash(int i, int j) {
        return (((prehsh[j + 1] - prehsh[i] + MOD) % MOD)
         * (inv_pw[i] % MOD)) % MOD;
    }

    ll getHash() {
        return substringHash(0, len - 1);
    }
};
```

Trie LC

Trie LeetCode style.

Time Complexity: O(L)

Code in C++:

```cpp
// Trie LeetCode style
// Placeholder - see source for full implementation
#include <bits/stdc++.h>
using namespace std;

// Trie LeetCode style
// O(L)
```

Trie Tree

Trie tree variant.

Time Complexity: O(L)

Code in C++:

```cpp
// Trie tree variant
// Placeholder - see source for full implementation
#include <bits/stdc++.h>
using namespace std;

// Trie tree variant
// O(L)
```

Utilities

Cumulative Sum 1D

1D Prefix Sum (Cumulative Sum) for efficient range sum queries.

Time Complexity:

- Build: $O(N)$
- Query: $O(1)$

Space Complexity: $O(N)$

Formula: `rangeSum(l, r) = pre[r] - pre[l-1]`

Code in C++:

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

int ar[100005], pre[100005];

void buildPrefixSum(int n) {
    pre[0] = 0;
    for(int i = 1; i <= n; i++) {
        pre[i] = pre[i - 1] + ar[i];
    }
}

int rangeSum(int l, int r) {
    return pre[r] - pre[l - 1];
}
```

Cumulative Sum 2D

2D Prefix Sum for efficient rectangle range sum queries.

Time Complexity:

- Build: $O(R \times C)$
- Query: $O(1)$

Formula: `sum = px[i2][j2] - px[i2][j1-1] - px[i1-1][j2] + px[i1-1][j1-1]`

Code in C++:

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

int ar[1005][1005], px[1005][1005];

void build2DPrefixSum(int r, int c) {
    for(int i = 0; i <= r; i++) px[i][0] = 0;
    for(int j = 0; j <= c; j++) px[0][j] = 0;

    px[1][1] = ar[1][1];
    for(int i = 2; i <= r; i++) {
        px[i][1] = px[i - 1][1] + ar[i][1];
    }
    for(int j = 2; j <= c; j++) {
        px[1][j] = px[1][j - 1] + ar[1][j];
    }

    for(int i = 2; i <= r; i++) {
        for(int j = 2; j <= c; j++) {
            px[i][j] = px[i - 1][j] + px[i][j - 1] +
             ar[i][j] - px[i - 1][j - 1];
        }
    }
}

int rangeSum2D(int i1, int j1, int i2, int j2) {
    return px[i2][j2] - px[i2][j1 - 1] - px[i1 - 1][j2] +
     px[i1 - 1][j1 - 1];
}
```