HEADER FILES

```
#include<bits/stdc++.h>
#include<ext/pb_ds/assoc_container.hpp>
#include<ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
using namespace std;
```

ADHOC

COORDINATE COMP

```
inline namespace MY{
   struct custom_hash_cc{
      static uint64_t splitmix64(uint64_t x){
         x += 0x9e3779b97f4a7c15;
         x = (x ^ (x >> 30)) * 0xbf58476d1ce4e5b9;
         x = (x ^ (x >> 27)) * 0x94d049bb133111eb;
         return x ^ (x >> 31);
      }
      size_t operator()(uint64_t x) const{
         static const uint64_t FIXED_RANDOM =
chrono::steady_clock::now().time_since_epoch().count();
         return splitmix64(x + FIXED_RANDOM);
      }
   };

   template<typename Key, typename Value>
   using HashMap = gp_hash_table<Key, Value,
custom_hash_cc>;

   template<typename T>
   class CoordinateCompressor{
   private:
      vector<T> original, sorted_unique;
      HashMap<T, int> compress_map;

   public:
      // O(n log n)
      CoordinateCompressor(const vector<T>& values){
         original = values;
         sorted_unique = values;
         sort(sorted_unique.begin(),
sorted_unique.end());

sorted_unique.erase(unique(sorted_unique.begin(),
sorted_unique.end()), sorted_unique.end());
         for(int i = 0; i < (int)sorted_unique.size(); i++)
compress_map[sorted_unique[i]] = i;
         original.shrink_to_fit();
         sorted_unique.shrink_to_fit();
      }
      // O(1) avg
      int compress(const T& value) const{
         auto it = compress_map.find(value);
         if(it == compress_map.end()){
            cerr << "Runtime Error: Value not found in
CoordinateCompressor -> " << value << "\n";
            exit(1);
         }
         return it->second;
      }
      // O(1)
      T decompress(int index) const{
         if(index < 0 || index >= size()){
            cerr << "Runtime Error: Invalid index in
decompress -> " << index << "\n";
            exit(1);
         }
         return sorted_unique[index];
      }
      // O(n)
      vector<int> get_compressed() const{
         vector<int> res; res.reserve(original.size());
         for(auto& v : original)
res.push_back(compress(v));
         return res;
      }
      // O(n log n)
      vector<int> get_sorted_compressed() const{
         vector<int> res = get_compressed();
         sort(res.begin(), res.end());
         return res;
      }
      // O(1)
      const vector<T>& get_sorted_unique_original()
const{ return sorted_unique; }
      // O(n)
      vector<int> get_frequency_array() const{
         vector<int> freq(size(), 0);
         for(auto& v : original) freq[compress(v)]++;
         return freq;
      }
      // O(u)
      vector<int> get_empty_freq_array() const{ return
vector<int>(size(), 0); }
      // O(1)
      const vector<T>& get_original() const{ return
original; }
      // O(1)
      int size() const{ return (int)sorted_unique.size(); }
   };
}
```

TREES

EULER TOUR

```
inline namespace MY{
   class EulerTour{
   public:
      int n, timer, type;
      vector<vector<int>> g;
      vector<int> tin, tout, depth, parent;
      vector<int> entry, exit_tl, entryExitTour, fullTour;
      vector<pair<int,int>> edgeTimeline;
      vector<int> firstOccur;

      // O(n)
      EulerTour(int _n, int _type=0){ init(_n,_type); }
      // O(n)
      void init(int _n, int _type){
         n=_n; type=_type; timer=0;
         g.assign(n,{}); tin.assign(n,-1); tout.assign(n,-1);
         depth.assign(n,0); parent.assign(n,-1);
         firstOccur.assign(n,-1);
         entry.clear(); exit_tl.clear(); entryExitTour.clear();
         fullTour.clear(); edgeTimeline.clear();
      }
      // O(1)
      void addEdge(int u,int v){ g[u].push_back(v);
g[v].push_back(u); }
      // O(n)
      void build(int root=0){ dfs(root,-1,0); }
      // O(n)
      void dfs(int v,int p,int d){
         parent[v]=p; depth[v]=d;
         if(type==-1 || type==0){
            tin[v]=timer++; entry.push_back(v);
entryExitTour.push_back(v);
         }
         fullTour.push_back(v);
         if(firstOccur[v]==-1)
firstOccur[v]=(int)fullTour.size()-1;
         for(int to:g[v]) if(to!=p){
            edgeTimeline.push_back({v,to});
            dfs(to,v,d+1);
            edgeTimeline.push_back({to,v});
            fullTour.push_back(v);
            if(type==0) entryExitTour.push_back(v);
         }
         if(type==0) entryExitTour.push_back(v);
         if(type==1 || type==0){
            tout[v]=timer++; exit_tl.push_back(v);
         }
      }
      // O(1)
      bool isAncestor(int u,int v){ return tin[u]<=tin[v] &&
tout[v]<=tout[u]; }
      // O(1)
      int subtreeSize(int v){ return tout[v]-tin[v]; }
      // O(1)
      const vector<int>& getTin() const{ return tin; }
      // O(1)
      const vector<int>& getTout() const{ return tout; }
      // O(1)
      const vector<int>& getEntryTimeline() const{ return
entry; }
      // O(1)
      const vector<int>& getExitTimeline() const{ return
exit_tl; }
      // O(1)
      const vector<int>& getEntryExitTimeline() const{
return entryExitTour; }
      // O(1)
      const vector<int>& getFullTour() const{ return
fullTour; }
      // O(1)
      const vector<int>& getDepthArray() const{ return
depth; }
      // O(1)
      const vector<int>& getParentArray() const{ return
parent; }
      // O(1)
      const vector<pair<int,int>>& getEdgeTimeline()
const{ return edgeTimeline; }
      // O(1)
      const vector<int>& getFirstOccur() const{ return
firstOccur; }
   };
}
```

HLD

```
inline namespace MY{
   class HLD{
   private:
      int N, timer; bool oneBased;
      struct Edge{ int to, weight; };
      vector<vector<Edge>> g;
      vector<int> parent, depth, heavy, head, pos,
subtree, nodeValue, edgeValue, flat;

      // O(N)
      int dfs(int u,int p){
         parent[u]=p; subtree[u]=1; int max_size=0;
         for(auto e:g[u]){
            int v=e.to; if(v==p) continue;
            depth[v]=depth[u]+1; edgeValue[v]=e.weight;
            int sz=dfs(v,u); subtree[u]+=sz;
            if(sz>max_size){ max_size=sz; heavy[u]=v; }
         }
         return subtree[u];
      }
      // O(N)
      void decompose(int u,int h){
         head[u]=h; pos[u]=timer++;
flat[pos[u]]=nodeValue[u];
         if(heavy[u]!=-1) decompose(heavy[u],h);
         for(auto e:g[u]){
            int v=e.to;
```

```
            if(v!=parent[u] && v!=heavy[u])
decompose(v,v);
        }
    }
    // O(1)
    int convert(int u)const{ return oneBased?u-1:u; }

  public:
    // O(N)
    HLD(int n,bool
oneBasedInput=false):N(n),timer(0),oneBased(oneBase
dInput){
        g.assign(N,{}); parent.assign(N,-1);
depth.assign(N,0); heavy.assign(N,-1);
        head.assign(N,-1); pos.assign(N,0);
subtree.assign(N,0);
        nodeValue.assign(N,0); edgeValue.assign(N,0);
flat.assign(N,0);
    }
    // O(1)
    void addEdge(int u,int v,int w=0){
        u=convert(u); v=convert(v);
        g[u].push_back({v,w}); g[v].push_back({u,w});
    }
    // O(1)
    void setNodeValue(int u,int val){ u=convert(u);
nodeValue[u]=val; }
    // O(N)
    void init(int root=0){
        root=convert(root); dfs(root,-1);
decompose(root,root); edgeValue[root]=0;
    }
    // O(N)
    void flattenEdges(int root=0){
        root=convert(root);
        for(int i=0;i<N;++i) flat[pos[i]]=edgeValue[i];
        flat[pos[root]]=0;
    }
    // O(1)
    const vector<int>& getFlat()const{ return flat; }
    // O(1)
    int getPos(int u)const{ return pos[convert(u)]; }
    // O(1)
    int getChainHead(int u)const{ return
head[convert(u)]+(oneBased?1:0); }
    // O(1)
    int getParent(int u)const{
        int p=parent[convert(u)];
        return (p==-1?-1:p+(oneBased?1:0));
    }
    // O(1)
    int getHeavyChild(int u)const{
        int h=heavy[convert(u)];
        return (h==-1?-1:h+(oneBased?1:0));
    }
    // O(1)
    int getDepth(int u)const{ return depth[convert(u)]; }
    // O(1)
    int getSubtreeSize(int u)const{ return
subtree[convert(u)]; }
  };
}
```

CENTROID DCP

```
inline namespace MY{
  class CentroidDecomposition{
  public:
    int n; bool isOneBased;
    vector<vector<int>> tree;
    vector<bool> isCentroid;
    vector<int> subSize, origSubSize, parentCentroid,
depthInCT;
    vector<vector<pair<int,int>>> centroidPath;
    // O(n)
    CentroidDecomposition(int nodes, bool oneBased =
true){
        n = nodes; isOneBased = oneBased;
        tree.assign(n, {}); isCentroid.assign(n, false);
        subSize.assign(n, 0); origSubSize.assign(n, 0);
        parentCentroid.assign(n, -1);
depthInCT.assign(n, 0);
        centroidPath.assign(n, {});
    }
    // O(1)
    void addEdge(int u, int v){
        if(isOneBased) u--, v--;
        tree[u].push_back(v); tree[v].push_back(u);
    }
  private:
    // O(subtree)
    void dfsSubSize(int u, int p){
        subSize[u] = 1;
        for(int v : tree[u]) if(v != p && !isCentroid[v])
dfsSubSize(v, u), subSize[u] += subSize[v];
    }
    // O(subtree)
    int findCentroid(int u, int p, int n){
        for(int v : tree[u]) if(v != p && !isCentroid[v] &&
subSize[v] > n / 2) return findCentroid(v, u, n);
        return u;
    }
    // O(subtree)
    void buildCentroidPaths(int u, int p, int dist, int
root){
        centroidPath[u].push_back({root, dist});
        for(int v : tree[u]) if(v != p && !isCentroid[v])
buildCentroidPaths(v, u, dist + 1, root);
    }
  public:
    // O(n)
    void computeOriginalSubtreeSizes(int root = 0){
        dfsSubSize(root, -1);
        origSubSize = subSize;
    }
    // O(n log n)
    int decompose(int u, int p = -1, int depth = 0){
        dfsSubSize(u, -1);
        int c = findCentroid(u, -1, subSize[u]);
        isCentroid[c] = true; parentCentroid[c] = p;
depthInCT[c] = depth;
        for(int v : tree[c]) if(!isCentroid[v]) decompose(v,
c, depth + 1);
        return c;
    }
    // O(n log n)
    void initPaths(){
        fill(isCentroid.begin(), isCentroid.end(), false);
        queue<int> q; int root = -1;
        for(int i = 0; i < n; i++) if(parentCentroid[i] == -1){
root = i; break; }
        if(root == -1) return;
        q.push(root);
        while(!q.empty()){
            int c = q.front(); q.pop();
            isCentroid[c] = true;
            buildCentroidPaths(c, -1, 0, c);
            for(int v : tree[c]) if(!isCentroid[v]) q.push(v);
        }
        fill(isCentroid.begin(), isCentroid.end(), false);
    }
    // O(n)
    vector<int> getParentCentroid(){ return
parentCentroid; }
    // O(n)
    vector<int> getDepthInCT(){ return depthInCT; }
    // O(n)
    vector<int> getSubtreeSizes(){ return subSize; }
    // O(n)
    vector<int> getOriginalSubtreeSizes(){ return
origSubSize; }
    // O(n)
    vector<vector<pair<int,int>>> getCentroidPaths(){
return centroidPath; }
  };
}
```

DATA STRUCTURE

BINARY TRIE

```
inline namespace MY{
  template<typename T,int MAX_BITS=(sizeof(T)*8)-1>
  class BinaryTrieNode{
  public:
    BinaryTrieNode* child[2];
    unsigned int used;
    // O(1)
    BinaryTrieNode(){child[0]=child[1]=nullptr;used=0;}
  };
  template<typename T,int MAX_BITS=(sizeof(T)*8)-1>
  class BinaryTrie{
  private:
    using Node=BinaryTrieNode<T,MAX_BITS>;
    Node* root;
    // O(N)
    void freeNode(Node* node){
        if(!node) return;
        freeNode(node->child[0]);
        freeNode(node->child[1]);
        delete node;
    }
    // O(B)
    bool eraseHelper(Node* node,T num,int bitPos){

if(bitPos<0){node->used--;return(node->used==0&&!nod
e->child[0]&&!node->child[1]);}
        int bit=(num>>bitPos)&1;
        Node* next=node->child[bit];
        if(!next) return false;
        bool
shouldDelete=eraseHelper(next,num,bitPos-1);
        if(shouldDelete){delete
next;node->child[bit]=nullptr;}
        node->used--;

return(node->used==0&&!node->child[0]&&!node->child[
1]);
    }
  public:
    // O(1)
    BinaryTrie(){root=new Node();}
    // O(N)
    ~BinaryTrie(){freeNode(root);}
    // O(B)
    void insert(T num){
        Node* ptr=root;ptr->used++;
        for(int i=MAX_BITS;i>=0;i--){
            int bit=(num>>i)&1;
            if(!ptr->child[bit]) ptr->child[bit]=new Node();
            ptr=ptr->child[bit];
            ptr->used++;
        }
    }
    // O(B)
    bool erase(T num){
        if(!exists(num)) return false;
        eraseHelper(root,num,MAX_BITS);
        return true;
    }
    // O(B)
```

```
    bool exists(T num)const{
        Node* ptr=root;
        for(int i=MAX_BITS;i>=0;i--){
            int bit=(num>>i)&1;
            if(!ptr->child[bit]) return false;
            ptr=ptr->child[bit];
        }
        return(ptr->used>0);
    }
    // O(B)
    T maxXor(T num)const{
        Node* ptr=root;
        if(!ptr||ptr->used==0) return 0;
        T res=0;
        for(int i=MAX_BITS;i>=0;i--){
            int bit=(num>>i)&1;
            int want=1-bit;

if(ptr->child[want]&&ptr->child[want]->used>0){res|=(T(1)<<i);ptr=ptr->child[want];}
            else if(ptr->child[bit]&&ptr->child[bit]->used>0) ptr=ptr->child[bit];
            else break;
        }
        return res;
    }
    // O(B)
    T minXor(T num)const{
        Node* ptr=root;
        if(!ptr||ptr->used==0) return 0;
        T res=0;
        for(int i=MAX_BITS;i>=0;i--){
            int bit=(num>>i)&1;
            if(ptr->child[bit]&&ptr->child[bit]->used>0) ptr=ptr->child[bit];
            else if(ptr->child[1-bit]&&ptr->child[1-bit]->used>0){res|=(T(1)<<i);ptr=ptr->child[1-bit];}
            else break;
        }
        return res;
    }
    // O(B)
    unsigned int countXorLessThan(T num,T k)const{
        Node* ptr=root;unsigned int res=0;
        for(int i=MAX_BITS;i>=0&&ptr;i--){
            int bitNum=(num>>i)&1,bitK=(k>>i)&1;
            if(bitK==1){if(ptr->child[bitNum]) res+=ptr->child[bitNum]->used;ptr=ptr->child[1-bitNum];}
            else ptr=ptr->child[bitNum];
        }
        return res;
    }
    // O(1)
    bool isEmpty()const{return root->used==0;}
};
}
```

CHAR TRIE

```
inline namespace MY{
class CharTrieNode{
public:
    char data;vector<CharTrieNode*> children;bool isTerminal;int count,used;
    // O(1)
    CharTrieNode(char ch,int alphabet_size):data(ch),children(alphabet_size,nullptr),isTerminal(false),count(0),used(0){}
};
class CharTrie{
private:
    CharTrieNode* root;vector<char> alphabet;unordered_map<char,int> charToIdx;int alphabet_size;
    // O(N)
    void freeNode(CharTrieNode* node){if(!node)return;for(auto child:node->children)freeNode(child);delete node;}
    // O(L)
    bool eraseHelper(CharTrieNode* node,const string&word,int depth){

if(depth==word.size()){if(node->count>0){node->count--;if(node->count==0)node->isTerminal=false;return true;}return false;}
        char c=word[depth];if(charToIdx.find(c)==charToIdx.end())return false;int idx=charToIdx[c];CharTrieNode* child=node->children[idx];if(!child)return false;
        bool deleted=eraseHelper(child,word,depth+1);

if(deleted){child->used--;if(child->count==0&&!child->isTerminal&&child->used==0){delete child;node->children[idx]=nullptr;}}
        return deleted;
    }
    // O(M)
    void dfs(CharTrieNode* node,string current,vector<pair<string,int>>&res){
        if(!node)return;

if(node->isTerminal)res.push_back({current,node->count});
        for(int i=0;i<alphabet_size;i++)if(node->children[i])dfs(node->children[i],current+alphabet[i],res);
    }
public:
    // O(A)
    CharTrie(bool lowercase=true,bool uppercase=true){
        if(lowercase)for(char c='a';c<='z';c++)alphabet.push_back(c);
        if(uppercase)for(char c='A';c<='Z';c++)alphabet.push_back(c);
        if(alphabet.empty())for(char c='a';c<='z';c++)alphabet.push_back(c);
        alphabet_size=alphabet.size();
        for(int i=0;i<alphabet_size;i++)charToIdx[alphabet[i]]=i;
        root=new CharTrieNode('\0',alphabet_size);
    }
    // O(N)
    ~CharTrie(){freeNode(root);}
    // O(L)
    void insert(const string&word){
        CharTrieNode* node=root;node->used++;
        for(char c:word){

if(charToIdx.find(c)==charToIdx.end())continue;
            int idx=charToIdx[c];

if(!node->children[idx])node->children[idx]=new CharTrieNode(c,alphabet_size);
            node=node->children[idx];node->used++;
        }
        node->isTerminal=true;node->count++;
    }
    // O(L)
    int countWordsEqualTo(const string&word){
        CharTrieNode* node=root;
        for(char c:word){
            if(charToIdx.find(c)==charToIdx.end())return 0;
            int idx=charToIdx[c];
            if(!node->children[idx])return 0;
            node=node->children[idx];
        }
        return node->count;
    }
    // O(L)
    int countWordsStartingWith(const string&prefix){
        CharTrieNode* node=root;
        for(char c:prefix){
            if(charToIdx.find(c)==charToIdx.end())return 0;
            int idx=charToIdx[c];
            if(!node->children[idx])return 0;
            node=node->children[idx];
        }
        return node->used;
    }
    // O(L)
    void erase(const string&word){if(eraseHelper(root,word,0))root->used--;}
    // O(L+M)
    vector<pair<string,int>> getWordsWithPrefix(const string&prefix){
        CharTrieNode* node=root;
        for(char c:prefix){
            if(charToIdx.find(c)==charToIdx.end())return{};
            int idx=charToIdx[c];
            if(!node->children[idx])return{};
            node=node->children[idx];
        }
        vector<pair<string,int>> res;dfs(node,prefix,res);return res;
    }
    // O(N)
    void clear(){freeNode(root);root=new CharTrieNode('\0',alphabet_size);}
};
}
```

SUFFIX ARRAY

```
inline namespace MY{
class SuffixArray{
  string s; int n0,n; vector<int> sa,rank_,lcp,lg2; vector<vector<int>> st; vector<long long> numSubstrings; bool lcp_built=false,rmq_built=false;
public:
  // O(n log n)
  explicit SuffixArray(const string& text){
    n0=text.size(); s=text; s.push_back('\0'); n=n0+1; build_sa(); build_rank();
  }

  // O(n log n)
  void build_sa(){
    sa.resize(n); vector<int> r(n),tmp(n);
    for(int i=0;i<n;i++){ sa[i]=i; r[i]=(unsigned char)s[i]; }
    for(int k=1;k<n;k<<=1){
      auto radix=[&](int maxv){
        vector<int> sa2(n); int C=max(maxv+2,n+1); vector<int> cnt(C);
        for(int i=0;i<n;i++){ int key=(i+k<n?r[i+k]+1:0); cnt[key]++; }
        for(int i=1;i<C;i++) cnt[i]+=cnt[i-1];
        for(int i=n-1;i>=0;i--){ int idx=sa[i]; int key=(idx+k<n?r[idx+k]+1:0); sa2[--cnt[key]]=idx; }
        fill(cnt.begin(),cnt.end(),0);
        for(int i=0;i<n;i++) cnt[r[i]+1]++;
        for(int i=1;i<C;i++) cnt[i]+=cnt[i-1];
        for(int i=n-1;i>=0;i--){ int idx=sa2[i]; sa[--cnt[r[idx]+1]]=idx; }
      };
      int maxv=*max_element(r.begin(),r.end()); radix(maxv);
      tmp[sa[0]]=0; int classes=1;
```

```
        for(int i=1;i<n;i++){ int a=sa[i-1],b=sa[i];
if(r[a]!=r[b]||(a+k<n?r[a+k]:-1)!=(b+k<n?r[b+k]:-1)) classes++;
            tmp[b]=classes-1;
        }
        r.swap(tmp); if(classes==n) break;
      }
      rank_.resize(n); for(int i=0;i<n;i++) rank_[sa[i]]=i;
   }

   // O(n)
   void build_rank(){
      if((int)rank_.size()!=n){ rank_.resize(n); for(int i=0;i<n;i++) rank_[sa[i]]=i; }
   }

   // O(n)
   void build_lcp(){
      if(lcp_built) return; lcp.assign(max(0,n-1),0); int k=0;
      for(int i=0;i<n;i++){ if(rank_[i]==n-1){ k=0; continue; }
         int j=sa[rank_[i]+1];
while(i+k<n&&j+k<n&&s[i+k]==s[j+k]) k++;
         lcp[rank_[i]]=k; if(k) k--;
      }
      lcp_built=true;
   }

   // O(n)
   void build_numSubstrings(){
      build_lcp(); numSubstrings.assign(n,0);
      for(int i=0;i<n;i++){ if(sa[i]>=n0) continue;
         int prevLcp=(i>0?lcp[i-1]:0); int suffixLen=n0-sa[i];
         numSubstrings[i]=suffixLen-prevLcp;
      }
      for(int i=1;i<n;i++) numSubstrings[i]+=numSubstrings[i-1];
   }

   // O(n log n)
   void build_rmq(){
      if(rmq_built) return; build_lcp(); int m=lcp.size();
      if(m==0){ lg2.assign(1,0); rmq_built=true; return; }
      lg2.assign(m+1,0); for(int i=2;i<=m;i++) lg2[i]=lg2[i/2]+1;
      int K=lg2[m]+1; st.assign(K,vector<int>(m));
st[0]=lcp;
      for(int k=1;k<K;k++) for(int i=0;i+(1<<k)<=m;i++) st[k][i]=min(st[k-1][i],st[k-1][i+(1<<(k-1))]);
      rmq_built=true;
   }

   // O(1)
   int LCP_between_suffixes(int i,int j){
      if(i==j) return n0-i; build_rmq(); if(lcp.empty()) return 0;
      int ri=rank_[i],rj=rank_[j]; if(ri>rj) swap(ri,rj);
      int len=rj-ri; if(len<=0) return 0; int k=lg2[len];
      return min(st[k][ri],st[k][rj-(1<<k)]);
   }

   // O(|pat| log n)
   pair<int,int> range_in_SA(const string& pat)const{
      int m=pat.size(),lo=0,hi=n,lcpL=0,lcpR=0;
      while(lo<hi){ int mid=(lo+hi)>>1,idx=sa[mid],lcpMid=min(lcpL,lcpR);
while(lcpMid<m&&idx+lcpMid<n&&s[idx+lcpMid]==pat[lcpMid]) lcpMid++;
if(idx+lcpMid==n||(lcpMid<m&&s[idx+lcpMid]<pat[lcpMid])){ lo=mid+1; lcpL=lcpMid; }
         else{ hi=mid; lcpR=lcpMid; }
      }
      int L=lo; lo=0; hi=n; lcpL=lcpR=0;
      while(lo<hi){ int mid=(lo+hi)>>1,idx=sa[mid],lcpMid=min(lcpL,lcpR);
while(lcpMid<m&&idx+lcpMid<n&&s[idx+lcpMid]==pat[lcpMid]) lcpMid++;
if(idx+lcpMid==n||(lcpMid<m&&s[idx+lcpMid]<=pat[lcpMid])){ lo=mid+1; lcpL=lcpMid; }
         else{ hi=mid; lcpR=lcpMid; }
      }
      return {L,lo};
   }

   // O(n·|pat|)
   bool contains(const string& pat)const{
      int m=pat.size();
      for(int i=0;i+m<=n0;i++){ bool ok=true;
         for(int j=0;j<m;j++){ if(s[i+j]!=pat[j]){ ok=false; break; } }
         if(ok) return true;
      }
      return false;
   }

   // O(n·|pat|)
   vector<int> findAllOccurrences(const string& pat)const{
      int m=pat.size(); vector<int> res;
      for(int i=0;i+m<=n0;i++){ bool ok=true;
         for(int j=0;j<m;j++){ if(s[i+j]!=pat[j]){ ok=false; break; } }
         if(ok) res.push_back(i);
      }
      return res;
   }

   // O(n)
   string longestRepeatedSubstring(){
      build_lcp(); if(lcp.empty()) return ""; int best=0,start=-1;
      for(int i=0;i<(int)lcp.size();i++) if(lcp[i]>best){ best=lcp[i]; start=min(sa[i],sa[i+1]); }
      return (best==0?"":s.substr(start,best));
   }

   // O(n)
   long long countDistinctSubstrings(){
      build_lcp(); long long total=1LL*n0*(n0+1)/2; for(int x:lcp) total-=x; return total;
   }

   // O(n)
   string kthSubstring(long long k){
      build_numSubstrings();
if(k<=0||numSubstrings.back()<k) return "";
      int lo=0,hi=n-1; while(lo<hi){ int mid=(lo+hi)/2;
if(numSubstrings[mid]>=k) hi=mid; else lo=mid+1; }
      int idx=sa[lo]; int prev=(lo>0?numSubstrings[lo-1]:0); int len=(lcp_built&&lo>0?lcp[lo-1]:0)+(int)(k-prev);
      return s.substr(idx,len);
   }

   // O((n+m) log(n+m))
   static string longestCommonSubstring(const string& a,const string& b){
      string joined=a; joined.push_back(char(1)); joined+=b; SuffixArray st(joined);
      st.build_lcp(); int nA=a.size(),best=0,pos=-1;
      for(int i=0;i<(int)st.lcp.size();i++){ int x=st.sa[i],y=st.sa[i+1];
         if((x<nA)!=(y<nA)&&st.lcp[i]>best){ best=st.lcp[i]; pos=st.sa[i]; }
      }
      return (best==0?"":joined.substr(pos,best));
   }

   // O(1)
   const vector<int>& getSA()const{return sa;}

   // O(n)
   const vector<int>& getLCP(){ build_lcp(); return lcp; }
};
}
```

GRAPHS

CONNECTIVITY

EBCC

```
inline namespace MY{
vector<vector<pair<int,int>>> BCC; stack<pair<int,int>> edgeStack; int timerBCC;

// O(V + E)
void dfsBCC(int node,int parent,unordered_map<int,list<int>>&adj,vector<int>&disc,vector<int>&low,vector<bool>&visited){
   visited[node]=true;
disc[node]=low[node]=timerBCC++;
   for(int nbr:adj[node]){
      if(nbr==parent) continue;
      if(!visited[nbr]){
         edgeStack.push({node,nbr});
         dfsBCC(nbr,node,adj,disc,low,visited);
         low[node]=min(low[node],low[nbr]);
         if(low[nbr]>=disc[node]){
            vector<pair<int,int>>component;
while(!edgeStack.empty()&&edgeStack.top()!=make_pair(node,nbr)){
               component.push_back(edgeStack.top()); edgeStack.pop();
            }
            component.push_back(edgeStack.top()); edgeStack.pop();
            BCC.push_back(component);
         }
      }else if(disc[nbr]<disc[node]){
         low[node]=min(low[node],disc[nbr]);
         edgeStack.push({node,nbr});
      }
   }
}

// O(V + E)
vector<vector<pair<int,int>>> findBCC(int n,unordered_map<int,list<int>>&adj){
   BCC.clear(); while(!edgeStack.empty()) edgeStack.pop(); timerBCC=0;
   vector<int>disc(n,-1),low(n,-1);
vector<bool>visited(n,false);
   for(int i=0;i<n;i++){
      if(!visited[i]){
         dfsBCC(i,-1,adj,disc,low,visited);
         if(!edgeStack.empty()){
            vector<pair<int,int>>component;
```

```
            while(!edgeStack.empty()){
component.push_back(edgeStack.top());
edgeStack.pop(); }
            BCC.push_back(component);
          }
        }
    }
    return BCC;
}
}
```

VBCC

```
inline namespace MY{
    // O(V + E)
    void vbccDFS(int node, int parent,
unordered_map<int, vector<int>>& adj, vector<int>&
disc, vector<int>& low, vector<bool>& visited,
stack<int>& st, vector<vector<int>>& components, int&
timer){
        visited[node]=true;
        disc[node]=low[node]=timer++;
        st.push(node);
        int childCount=0;
        for(int nbr:adj[node]){
            if(nbr==parent) continue;
            if(!visited[nbr]){
                childCount++;

vbccDFS(nbr,node,adj,disc,low,visited,st,components,ti
mer);
                low[node]=min(low[node],low[nbr]);

if((parent!=-1&&low[nbr]>=disc[node])||(parent==-1&&chi
ldCount>1)){
                    vector<int> component;
                    while(!st.empty()&&st.top()!=nbr){
                        component.push_back(st.top());
                        st.pop();
                    }
                    component.push_back(nbr);
                    component.push_back(node);
                    components.push_back(component);
                }
            }else low[node]=min(low[node],disc[nbr]);
        }
    }

    // O(V + E)
    vector<vector<int>>
vertexBiconnectedComponents(int n,
unordered_map<int, vector<int>>& adj){
        vector<int> disc(n,-1),low(n,-1);
        vector<bool> visited(n,false);
        stack<int> st;
        vector<vector<int>> components;
        int timer=0;
        for(int i=0;i<n;i++){
            if(!visited[i]){

vbccDFS(i,-1,adj,disc,low,visited,st,components,timer);
                if(!st.empty()){
                    vector<int> comp;
                    while(!st.empty()){
                        comp.push_back(st.top());
                        st.pop();
                    }
                    components.push_back(comp);
                }
            }
        }
        return components;
    }
}
```

ARTC. POINTS

```
inline namespace MY{
    vector<int> articulationPoints; int timerAP;

    // O(V + E)
    void dfsAP(int node,int
parent,unordered_map<int,list<int>>& adj,vector<int>&
disc,vector<int>& low,vector<bool>&
visited,vector<bool>& isAP){
        visited[node]=true;
        disc[node]=low[node]=timerAP++;
        int childCount=0;
        for(auto nbr:adj[node]){
            if(nbr==parent) continue;
            if(!visited[nbr]){
                childCount++;
                dfsAP(nbr,node,adj,disc,low,visited,isAP);
                low[node]=min(low[node],low[nbr]);
                if(parent!=-1&&low[nbr]>=disc[node])
isAP[node]=true;
            }else low[node]=min(low[node],disc[nbr]);
        }
        if(parent==-1&&childCount>1) isAP[node]=true;
    }

    // O(V + E)
    vector<int> findArticulationPoints(int
n,unordered_map<int,list<int>>& adj){
        timerAP=0;
        articulationPoints.clear();
        vector<int> disc(n,-1),low(n,-1);
        vector<bool> visited(n,false),isAP(n,false);
        for(int i=0;i<n;i++) if(!visited[i])
dfsAP(i,-1,adj,disc,low,visited,isAP);
        for(int i=0;i<n;i++) if(isAP[i])
articulationPoints.push_back(i);
        return articulationPoints;
    }
}
```

BRIDGES

```
inline namespace MY{
    vector<pair<int,int>> bridges; int timer;

    // O(V + E)
    void dfsBridge(int node,int
parent,unordered_map<int,list<int>>& adj,vector<int>&
disc,vector<int>& low,vector<bool>& visited){
        visited[node]=true;
        disc[node]=low[node]=timer++;
        for(int nbr:adj[node]){
            if(nbr==parent) continue;
            if(!visited[nbr]){
                dfsBridge(nbr,node,adj,disc,low,visited);
                low[node]=min(low[node],low[nbr]);
                if(low[nbr]>disc[node])
bridges.push_back({node,nbr});
            }else low[node]=min(low[node],disc[nbr]);
        }
    }

    // O(V + E)
    vector<pair<int,int>> findBridges(int
n,unordered_map<int,list<int>>& adj){
        bridges.clear(); timer=0;
        vector<int> disc(n,-1),low(n,-1);
        vector<bool> visited(n,false);
        for(int i=0;i<n;i++) if(!visited[i])
dfsBridge(i,-1,adj,disc,low,visited);
        return bridges;
    }
}
```

KOSARAJU SCC

```
inline namespace MY{
    // O(V + E)
    void topoDFS(int
node,unordered_map<int,vector<int>>&
adj,vector<bool>& visited,stack<int>& st){
        visited[node]=true;
        for(auto nbr:adj[node]) if(!visited[nbr])
topoDFS(nbr,adj,visited,st);
        st.push(node);
    }

    // O(V + E)
    void revDFS(int
node,unordered_map<int,vector<int>>&
adjT,vector<bool>& visited,vector<int>& comp){
        visited[node]=true;
        comp.push_back(node);
        for(auto nbr:adjT[node]) if(!visited[nbr])
revDFS(nbr,adjT,visited,comp);
    }

    // O(V + E)
    vector<vector<int>> kosarajuSCC(int
n,unordered_map<int,vector<int>>& adj){
        vector<bool> visited(n,false); stack<int> st;
        for(int i=0;i<n;i++) if(!visited[i])
topoDFS(i,adj,visited,st);
        unordered_map<int,vector<int>> adjT;
        for(int i=0;i<n;i++) for(auto nbr:adj[i])
adjT[nbr].push_back(i);
        fill(visited.begin(),visited.end(),false);
        vector<vector<int>> scc;
        while(!st.empty()){
            int node=st.top(); st.pop();
            if(!visited[node]){
                vector<int> comp;
                revDFS(node,adjT,visited,comp);
                scc.push_back(comp);
            }
        }
        return scc;
    }
}
```

DAG

2SAT

```
inline namespace MY{
    class TwoSAT{
    private:
        int n;
        vector<vector<int>> adj,adjRev;
        vector<int> order,comp;
        vector<bool> visited;

        int var(int x){return x<<1;}
        int neg(int x){return x^1;}

        // O(V + E)
        void dfs1(int node){
            visited[node]=true;
            for(int nxt:adj[node]) if(!visited[nxt]) dfs1(nxt);
            order.push_back(node);
        }

        // O(V + E)
        void dfs2(int node,int cid){
            comp[node]=cid;
            for(int nxt:adjRev[node]) if(comp[nxt]==-1)
dfs2(nxt,cid);
```

```
    }

  public:
    TwoSAT(int n):n(n){
        adj.resize(2*n);
        adjRev.resize(2*n);
        comp.assign(2*n,-1);
        visited.assign(2*n,false);
    }

    // O(1)
    void implies(int a,int b){
        adj[a].push_back(b);
        adjRev[b].push_back(a);
    }

    // O(1)
    void addOR(int x,int y){
        implies(neg(x),y);
        implies(neg(y),x);
    }

    // O(1)
    void addEqual(int x,int y){
        addOR(x,neg(y));
        addOR(neg(x),y);
    }

    // O(1)
    void addXOR(int x,int y){
        addOR(x,y);
        addOR(neg(x),neg(y));
    }

    // O(1)
    void addImplication(int x,int y){
        addOR(neg(x),y);
    }

    // O(V + E)
    bool solve(vector<bool>& ans){
        for(int i=0;i<2*n;i++) if(!visited[i]) dfs1(i);
        int cid=0;
        for(int i=2*n-1;i>=0;i--){
            int v=order[i];
            if(comp[v]==-1) dfs2(v,cid++);
        }
        ans.resize(n);
        for(int i=0;i<n;i++){
            if(comp[var(i)]==comp[neg(var(i))]) return
false;
            ans[i]=(comp[var(i)]>comp[neg(var(i))]);
        }
        return true;
    }
  };
}
```

SCC COMPRESSED DAG

```
inline namespace MY{
  class SCCCompressor{
  public:
    int n;
    unordered_map<int,vector<int>> adj, adjT;
    vector<vector<int>> scc;
    vector<int> compId;
    vector<bool> visited;
    stack<int> st;

    SCCCompressor(int
n):n(n),visited(n,false),compId(n,-1){}

    // O(1)
    void addEdge(int u,int v){
        adj[u].push_back(v);
        adjT[v].push_back(u);
    }

    // O(V+E)
    void topoDFS(int u){
        visited[u]=true;
        for(int v:adj[u]) if(!visited[v]) topoDFS(v);
        st.push(u);
    }

    // O(V+E)
    void reverseDFS(int u,vector<int>&comp){
        visited[u]=true;
        comp.push_back(u);
        for(int v:adjT[u]) if(!visited[v])
reverseDFS(v,comp);
    }

    // O(V+E)
    void computeSCC(){
        for(int i=0;i<n;i++) if(!visited[i]) topoDFS(i);
        fill(visited.begin(),visited.end(),false);
        while(!st.empty()){
            int u=st.top(); st.pop();
            if(!visited[u]){
                vector<int>comp;
                reverseDFS(u,comp);
                scc.push_back(comp);
            }
        }
        for(int id=0;id<(int)scc.size();id++)
            for(int u:scc[id]) compId[u]=id;
    }

    // O(V+E)
    vector<vector<int>> buildCondensedDAG(){
        int sccCount=scc.size();
        vector<vector<int>> dag(sccCount);
        unordered_set<long long> edgeSet;
        for(int u=0;u<n;u++){
            for(int v:adj[u]){
                if(compId[u]!=compId[v]){
                    long long
hash=1LL*compId[u]*n+compId[v];
                    if(!edgeSet.count(hash)){
                        edgeSet.insert(hash);
                        dag[compId[u]].push_back(compId[v]);
                    }
                }
            }
        }
        return dag;
    }
  };
}
```

TOPO SORT

```
inline namespace MY{
  // O(V+E)
  void topoDFS(int node,
unordered_map<int,vector<int>>& adj, vector<bool>&
visited, stack<int>& st){
    visited[node]=true;
    for(auto nbr:adj[node]) if(!visited[nbr])
topoDFS(nbr,adj,visited,st);
    st.push(node);
  }

  // O(V+E)
  vector<int> topoSortDFS(int n,
unordered_map<int,vector<int>>& adj){
    vector<bool> visited(n,false);
    stack<int> st;
    for(int i=0;i<n;i++) if(!visited[i])
topoDFS(i,adj,visited,st);
    vector<int> topo;
    while(!st.empty()){ topo.push_back(st.top());
st.pop(); }
    return topo;
  }
}
```

DSU

DSU

```
inline namespace MY{
  class DSU{
    vector<int> parent,rank,size;
    int components;
  public:
    DSU(int n){ parent.resize(n); rank.assign(n,0);
size.assign(n,1); components=n; for(int i=0;i<n;i++)
parent[i]=i; }

    // O(α(N))
    int findParent(int node){ return
parent[node]==node?node:parent[node]=findParent(par
ent[node]); }

    // O(α(N))
    void unionByRank(int u,int v){
        int pu=findParent(u),pv=findParent(v);
        if(pu==pv) return;
        components--;
        if(rank[pu]<rank[pv]) parent[pu]=pv;
        else if(rank[pv]<rank[pu]) parent[pv]=pu;
        else{ parent[pv]=pu; rank[pu]++; }
    }

    // O(α(N))
    void unionBySize(int u,int v){
        int pu=findParent(u),pv=findParent(v);
        if(pu==pv) return;
        components--;
        if(size[pu]<size[pv]){ parent[pu]=pv;
size[pv]+=size[pu]; }
        else{ parent[pv]=pu; size[pu]+=size[pv]; }
    }

    // O(α(N))
    bool isSameSet(int u,int v){ return
findParent(u)==findParent(v); }

    // O(α(N))
    int getSize(int u){ return size[findParent(u)]; }

    // O(1)
    int getComponentCount(){ return components; }
  };
}
```

DSU ROLLBACK

```
inline namespace MY{
  struct DSURollback{
    vector<int> parent,sz;
    stack<pair<int,int>> history;
    int components;
```

```
    DSURollback(int n){ parent.resize(n);
sz.assign(n,1); components=n; for(int i=0;i<n;i++)
parent[i]=i; }

    // O(log N) amortized
    int find(int x){ while(x!=parent[x]) x=parent[x]; return
x; }

    // O(α(N))
    bool unite(int a,int b){
      a=find(a); b=find(b);
      if(a==b){ history.push({-1,-1}); return false; }
      if(sz[a]<sz[b]) swap(a,b);
      history.push({b,sz[a]});
      parent[b]=a; sz[a]+=sz[b]; components--;
      return true;
    }

    // O(1)
    void rollback(){
      auto last=history.top(); history.pop();
      if(last.first==-1) return;
      int b=last.first,oldSize=last.second,a=parent[b];
      parent[b]=b; sz[a]=oldSize; components++;
    }

    // O(1)
    int snapshot(){ return history.size(); }

    // O(#rollbacks)
    void rollbackTo(int checkpoint){
while(history.size()>checkpoint) rollback(); }
  };
}
```

DYNAMIC CONNECTIVITY

```
inline namespace MY{
  struct RollbackDSU{
    vector<int> parent,sz;
    stack<pair<int,int>> history;
    RollbackDSU()=default; RollbackDSU(int n){init(n);}
    void init(int n){ parent.resize(n); sz.assign(n,1);
while(!history.empty()) history.pop(); for(int i=0;i<n;i++)
parent[i]=i; }
    // O(log N)
    int find(int x){ while(x!=parent[x]) x=parent[x]; return
x; }
    // O(α(N))
    void unite(int a,int b){ a=find(a); b=find(b); if(a==b){
history.push({-1,-1}); return; } if(sz[a]<sz[b]) swap(a,b);
history.push({b,sz[a]}); parent[b]=a; sz[a]+=sz[b]; }
    // O(1)
    void rollback(){ if(history.empty()) return; auto
last=history.top(); history.pop(); if(last.first==-1) return; int
b=last.first,oldSizeA=last.second,a=parent[b];
parent[b]=b; sz[a]=oldSizeA; }
    // O(1)
    int snapshot()const{ return (int)history.size(); }
    // O(#rollbacks)
    void rollbackTo(int snap){
while((int)history.size()>snap) rollback(); }
  };

  struct Edge{ int u,v; long long w; int id; Edge(){}
Edge(int _u,int _v,long long _w=0,int
_id=-1):u(_u),v(_v),w(_w),id(_id){} };

  class OfflineDynamicConnectivity{
    int n,q; RollbackDSU du;
    vector<vector<Edge>> seg;
    vector<pair<int,int>> queries;
    vector<string> answers;

    // O(log q)
    void addEdgeInterval(int idx,int l,int r,int ql,int
qr,const Edge&e){
      if(ql>r||qr<l) return;
      if(ql<=l&&r<=qr){ seg[idx].push_back(e); return; }
      int mid=(l+r)>>1;
      addEdgeInterval(idx<<1,l,mid,ql,qr,e);
      addEdgeInterval(idx<<1|1,mid+1,r,ql,qr,e);
    }

    // O((V+E)logQ*α(N))
    void solveRec(int idx,int l,int r){
      int snap=dsu.snapshot();
      for(const Edge&e:seg[idx]) dsu.unite(e.u,e.v);
      if(l==r){
        if(queries[l].first!=-1){
          int u=queries[l].first,v=queries[l].second;

answers[l]=(dsu.find(u)==dsu.find(v)?string("YES"):string
("NO"));
        }
      }else{
        int mid=(l+r)>>1;
        solveRec(idx<<1,l,mid);
        solveRec(idx<<1|1,mid+1,r);
      }
      dsu.rollbackTo(snap);
    }

  public:
    OfflineDynamicConnectivity(int nNodes,int
qSlots):n(nNodes),q(qSlots){
      dsu.init(n); seg.assign(4*max(1,q),{});
queries.assign(max(1,q),{-1,-1});
answers.assign(max(1,q),"");
    }

    void reset(int nNodes,int qSlots){
      n=nNodes; q=qSlots; dsu.init(n);
      seg.clear(); seg.assign(4*max(1,q),{});
      queries.assign(max(1,q),{-1,-1});
answers.assign(max(1,q),"");
    }

    // O(log q)
    void addEdge(int u,int v,long long w,int start,int
end,int id=-1){
      if(q==0||start>end) return;
      Edge e(u,v,w,id);
addEdgeInterval(1,0,q-1,start,end,e);
    }

    // O(1)
    void addQuery(int u,int v,int t){ if(t<0||t>=q) return;
queries[t]={u,v}; }

    // O((N+Q)logQ*α(N))
    void buildAndSolve(){ if(q==0) return; dsu.init(n);
solveRec(1,0,q-1); }

    vector<string> getAnswers()const{ return answers;
}
    const vector<vector<Edge>>&
getSegmentTree()const{ return seg; }
    const vector<pair<int,int>>& getQueries()const{
return queries; }
  };
}
```

MST

PRIMS

```
inline namespace MY{
  class PrimMST{
  public:
    // O(E log V)
    long long run(int n, unordered_map<int,
list<pair<int,int>>> &adj){
      priority_queue<pair<int,int>,
vector<pair<int,int>>, greater<pair<int,int>>> pq;
      vector<bool> inMST(n, false);
      vector<int> parent(n, -1);
      vector<long long> key(n, LLONG_MAX);
      long long totalWeight = 0;
      key[0] = 0; pq.push({0,0});
      while(!pq.empty()){
        int u = pq.top().second; pq.pop();
        if(inMST[u]) continue;
        inMST[u] = true;
        totalWeight += key[u];
        for(auto &edge : adj[u]){
          int v = edge.first, wt = edge.second;
          if(!inMST[v] && wt < key[v]){
            key[v] = wt;
            parent[v] = u;
            pq.push({key[v], v});
          }
        }
      }
      return totalWeight;
    }
  };
}
```

KRUSKALS

```
inline namespace MY{
  class DSUK{
  public:
    vector<int> parent, rank, size; int components;
    DSUK(int n){ parent.resize(n); rank.assign(n,0);
size.assign(n,1); components=n; for(int i=0;i<n;i++)
parent[i]=i; }
    // O(α(n))
    int findParent(int x){ return
parent[x]==x?x:parent[x]=findParent(parent[x]); }
    // O(α(n))
    void unionByRank(int u,int v){
      u=findParent(u); v=findParent(v); if(u==v) return;
components--;
      if(rank[u]<rank[v]) parent[u]=v;
      else if(rank[v]<rank[u]) parent[v]=u;
      else parent[v]=u, rank[u]++;
    }
    // O(α(n))
    void unionBySize(int u,int v){
      u=findParent(u); v=findParent(v); if(u==v) return;
components--;
      if(size[u]<size[v]) parent[u]=v, size[v]+=size[u];
      else parent[v]=u, size[u]+=size[v];
    }
    bool isSameSet(int u,int v){ return
findParent(u)==findParent(v); }
    int getSize(int u){ return size[findParent(u)]; }
    int getComponentCount(){ return components; }
  };

  // O(E log E)
  long long kruskalMST(int n, vector<tuple<int,int,int>>
&edges){
    sort(edges.begin(), edges.end());
    DSUK dsu(n); long long mstWeight=0; int used=0;
```

```
    for(auto &[wt,u,v]:edges){
        if(!dsu.isSameSet(u,v)){
            dsu.unionBySize(u,v);
            mstWeight+=wt;
            if(++used==n-1) break;
        }
    }
    return mstWeight;
  }
}
```

SHORTEST PATHS

0/1 BFS

```
inline namespace MY{
  // O(V+E)
  vector<int> zeroOneBFS(unordered_map<int,
vector<pair<int,int>>> &adj, int n, int src){
    vector<int> dist(n,INT_MAX); deque<int> dq;
    dist[src]=0; dq.push_front(src);
    while(!dq.empty()){
      int u=dq.front(); dq.pop_front();
      for(auto &e:adj[u]){
        int v=e.first, w=e.second;
        if(dist[u]+w<dist[v]){
          dist[v]=dist[u]+w;
          if(w==0) dq.push_front(v);
          else dq.push_back(v);
        }
      }
    }
    return dist;
  }
}
```

BELLMAN FORD

```
inline namespace MY{
  // O(V*E)
  vector<int> bellmanFord(vector<tuple<int,int,int>>
&edges,int n,int src){
    vector<int> dist(n,INT_MAX); dist[src]=0;
    for(int i=1;i<n;i++){
      bool updated=false;
      for(auto &e:edges){
        int u,v,wt; tie(u,v,wt)=e;
        if(dist[u]!=INT_MAX && dist[u]+wt<dist[v]){
          dist[v]=dist[u]+wt;
          updated=true;
        }
      }
      if(!updated) break;
    }
    for(auto &e:edges){
      int u,v,wt; tie(u,v,wt)=e;
      if(dist[u]!=INT_MAX && dist[u]+wt<dist[v]){
        cout<<"Negative Weight Cycle Detected\n";
        return{};
      }
    }
    return dist;
  }
}
```

BFS UNWEIGHTED

```
inline namespace MY{
  // O(V+E)
  vector<int>
bfsShortestPath(unordered_map<int,list<int>> &adj,int
n,int src){
    vector<int> dist(n,INT_MAX); queue<int> q;
    dist[src]=0; q.push(src);
    while(!q.empty()){
      int node=q.front(); q.pop();
      for(auto &nbr:adj[node]){
        if(dist[nbr]==INT_MAX){
          dist[nbr]=dist[node]+1;
          q.push(nbr);
        }
      }
    }
    return dist;
  }
}
```

DAG SHORTEST LONGEST

```
inline namespace MY{
  // O(V+E)
  void topoDFS(int
node,unordered_map<int,vector<pair<int,int>>>
&adj,vector<bool> &visited,stack<int> &st){
    visited[node]=true;
    for(auto &edge:adj[node]){
      int nbr=edge.first;
      if(!visited[nbr]) topoDFS(nbr,adj,visited,st);
    }
    st.push(node);
  }

  // O(V+E)
  vector<long long> dagPath(int n,int
src,unordered_map<int,vector<pair<int,int>>> &adj,bool
mode){
    const long long INF=1e18;
    vector<long long> dist(n,mode?-INF:INF);
    dist[src]=0;
    vector<bool> visited(n,false);
    stack<int> st;
    for(int i=0;i<n;i++) if(!visited[i])
topoDFS(i,adj,visited,st);
    while(!st.empty()){
      int node=st.top(); st.pop();
      if(dist[node]==(mode?-INF:INF)) continue;
      for(auto &edge:adj[node]){
        int nbr=edge.first,wt=edge.second;
        if(!mode && dist[node]+wt<dist[nbr])
dist[nbr]=dist[node]+wt;
        else if(mode && dist[node]+wt>dist[nbr])
dist[nbr]=dist[node]+wt;
      }
    }
    return dist;
  }
}
```

DIJKSTRA

```
inline namespace MY{
  // O((V + E) log V)
  vector<int> dijkstra(unordered_map<int,
vector<pair<int,int>>> &adj, int n, int src){
    vector<int> dist(n, INT_MAX);
    vector<bool> vis(n, 0);
    priority_queue<pair<int,int>, vector<pair<int,int>>,
greater<pair<int,int>>> pq;
    dist[src] = 0; pq.push({0, src});
    while(!pq.empty()){
      auto [d, u] = pq.top(); pq.pop();
      if(vis[u]) continue; vis[u] = 1;
      for(auto &[v, w] : adj[u]){
        if(dist[u] + w < dist[v]){
          dist[v] = dist[u] + w;
          pq.push({dist[v], v});
        }
      }
    }
    return dist;
  }
}
```

FLOYD WARSHALL

```
inline namespace MY{
  // O(V^3)
  vector<vector<long long>> floydWarshall(int n,
vector<tuple<int,int,long long>> &edges){
    const long long INF = 1e18;
    vector<vector<long long>> dist(n, vector<long
long>(n, INF));
    for(int i=0;i<n;i++) dist[i][i]=0;
    for(auto &[u,v,w]:edges) dist[u][v]=min(dist[u][v],w);
    for(int k=0;k<n;k++) for(int i=0;i<n;i++)
if(dist[i][k]!=INF)
      for(int j=0;j<n;j++) if(dist[k][j]!=INF)
        dist[i][j]=min(dist[i][j],dist[i][k]+dist[k][j]);
    for(int i=0;i<n;i++) if(dist[i][i]<0){ cout<<"Negative
Weight Cycle\n"; return {}; }
    return dist;
  }
}
```

MULTISOURCE BFS

```
inline namespace MY{
  // O(V+E)
  vector<int> multiSourceBFS(int n,
unordered_map<int, list<int>> &adj, vector<int>
&sources){
    vector<int> dist(n,INT_MAX); queue<int> q;
    for(int src:sources){ dist[src]=0; q.push(src); }
    while(!q.empty()){
      int node=q.front(); q.pop();
      for(auto &nbr:adj[node]) if(dist[nbr]==INT_MAX){
        dist[nbr]=dist[node]+1; q.push(nbr);
      }
    }
    return dist;
  }
}
```

TRAVERSAL

BFS

```
inline namespace MY{
  // O(V+E)
  vector<int> bfs(int start, unordered_map<int, list<int>>
&adj){
    unordered_map<int,bool> vis; queue<int> q;
vector<int> order;
    q.push(start); vis[start]=true;
    while(!q.empty()){
      int node=q.front(); q.pop();
order.push_back(node);
      for(int nbr:adj[node]) if(!vis[nbr]){ vis[nbr]=true;
q.push(nbr); }
    }
    return order;
  }
}
```

DFS

```
inline namespace MY{
  // O(V+E)
  void dfsHelper(int node, unordered_map<int,list<int>>
&adj, unordered_map<int,bool> &vis, vector<int>
&order){
    vis[node]=true; order.push_back(node);
    for(int nbr:adj[node]) if(!vis[nbr])
dfsHelper(nbr,adj,vis,order);
  }

  // O(V+E)
```

```
    vector<int> dfs(int start, unordered_map<int,list<int>>
&adj){
        unordered_map<int,bool> vis; vector<int> order;
        dfsHelper(start,adj,vis,order);
        return order;
    }
}
```

CYCLE UNDIRECTED

```
inline namespace MY{
    // O(V+E)
    bool dfsCycle(int node,int
par,unordered_map<int,list<int>>
&adj,unordered_map<int,bool> &vis){
        vis[node]=true;
        for(int nbr:adj[node]){
            if(!vis[nbr]){
                if(dfsCycle(nbr,node,adj,vis)) return true;
            }else if(nbr!=par) return true;
        }
        return false;
    }

    // O(V+E)
    bool hasCycleDFS(unordered_map<int,list<int>>
&adj){
        unordered_map<int,bool> vis;
        for(auto &[node,_]:adj)
            if(!vis[node] && dfsCycle(node,-1,adj,vis)) return
true;
        return false;
    }
}
```

CYCLE DIRECTED

```
inline namespace MY{
    // O(V+E)
    bool detectCycleDFS(int
node,unordered_map<int,list<int>>
&adj,unordered_map<int,bool>
&vis,unordered_map<int,bool> &pathVis){
        vis[node]=true; pathVis[node]=true;
        for(int nbr:adj[node]){
            if(!vis[nbr] &&
detectCycleDFS(nbr,adj,vis,pathVis)) return true;
            else if(pathVis[nbr]) return true;
        }
        pathVis[node]=false;
        return false;
    }

    // O(V+E)
    bool hasCycleDirected(unordered_map<int,list<int>>
&adj){
        unordered_map<int,bool> vis,pathVis;
        for(auto &[node,_]:adj)
            if(!vis[node] &&
detectCycleDFS(node,adj,vis,pathVis)) return true;
        return false;
    }
}
```

CONNECTED COMPONENTS

```
inline namespace MY{
    // O(V+E)
    vector<vector<int>>
connectedComponents(unordered_map<int,list<int>>
&adj){
        unordered_map<int,bool> vis; vector<vector<int>>
comps;
        for(auto &[start,_]:adj){
            if(!vis[start]){
                vector<int> comp; queue<int> q; q.push(start);
vis[start]=true;
                while(!q.empty()){
                    int node=q.front(); q.pop();
comp.push_back(node);
                    for(int nbr:adj[node]) if(!vis[nbr])
vis[nbr]=true,q.push(nbr);
                }
                comps.push_back(comp);
            }
        }
        return comps;
    }
}
```

BIPARTITE CHECK

```
inline namespace MY{
    // O(V+E)
    bool dfsColor(int node,int
c,unordered_map<int,list<int>>&adj,vector<int>&color){
        color[node]=c;
        for(auto &nbr:adj[node]){
            if(color[nbr]==-1){
                if(!dfsColor(nbr,1-c,adj,color)) return false;
            }else if(color[nbr]==c) return false;
        }
        return true;
    }
    // O(V+E)
    bool isBipartite(unordered_map<int,list<int>>&adj,int
n){
        vector<int> color(n,-1);
        for(int i=0;i<n;i++) if(color[i]==-1 &&
!dfsColor(i,0,adj,color)) return false;
        return true;
    }
}
```

MATHS

COMBINATORICS

```
inline namespace MY {

    class Combinatorics {
        static const long long MOD = 1e9 + 7;
        static const int MAXN = 2e6;

        static vector<long long> fact, invFact;
        static bool initDone;
        static long long mod;
        static int maxN;

        // O(log b)
        static long long modPow(long long a, long long b,
long long m) {
            long long r = 1;
            a %= m;
            while (b) {
                if (b & 1) r = (__int128)r * a % m;
                a = (__int128)a * a % m;
                b >>= 1;
            }
            return r;
        }

    public:
        // O(n)
        static void init(int n = MAXN, long long m = MOD) {
            maxN = n; mod = m;
            fact.assign(n + 1, 1);
            invFact.assign(n + 1, 1);

            for (int i = 1; i <= n; i++) fact[i] = fact[i - 1] * i % m;
            invFact[n] = modPow(fact[n], m - 2, m);
            for (int i = n - 1; i >= 0; i--) invFact[i] = invFact[i +
1] * (i + 1) % m;

            initDone = true;
        }

        // ---------- Modular Arithmetic ----------
        // O(log m)
        static long long modInverse(long long a, long long
m = MOD) { return modPow(a, m - 2, m); }

        // O(log b)
        static long long powmod(long long a, long long b,
long long m = MOD) { return modPow(a, b, m); }

        // ---------- Factorials ----------
        // O(1)
        static long long factorial(int n, long long m = MOD) {
            if (!initDone) init();
            if (n < 0 || n > maxN) return 0;
            return fact[n] % m;
        }

        // O(1)
        static long long invFactorial(int n, long long m =
MOD) {
            if (!initDone) init();
            if (n < 0 || n > maxN) return 0;
            return invFact[n] % m;
        }

        // ---------- Combinatorics ----------
        // O(1)
        static long long nCr(int n, int r, long long m = MOD)
{
            if (!initDone) init();
            if (r < 0 || r > n || n > maxN) return 0;
            return fact[n] * invFact[r] % m * invFact[n - r] %
m;
        }

        // O(1)
        static long long nPr(int n, int r, long long m = MOD)
{
            if (!initDone) init();
            if (r < 0 || r > n) return 0;
            return fact[n] * invFact[n - r] % m;
        }

        // O(m)
        static long long multinomial(const vector<int>& ks,
long long m = MOD) {
            if (!initDone) init();
            long long sum = 0;
            for (int k : ks) sum += k;
            if (sum > maxN) return 0;
            long long res = fact[sum];
            for (int k : ks) res = res * invFact[k] % m;
            return res;
        }

        // O(1)
        static long long catalan(int n, long long m = MOD) {
            if (!initDone) init();
            long long n2 = 2LL * n;
            return nCr(n2, n, m) * modInverse(n + 1, m) % m;
        }

        // O(r)
        static long long nCrLarge(long long n, long long r,
long long m = MOD) {
            if (r < 0 || r > n) return 0;
```

```
        if (r > n / 2) r = n - r;
        long long res = 1;
        for (long long i = 1; i <= r; i++) {
            long long num = (n - i + 1) % m;
            long long inv = modInverse(i, m);
            res = (__int128)res * num % m;
            res = (__int128)res * inv % m;
        }
        return res;
    }

    // ---------- Number Theory ----------
    // O(log(min(a,b)))
    static long long gcd(long long a, long long b) {
return b ? gcd(b, a % b) : a; }

    // O(log(min(a,b)))
    static long long lcm(long long a, long long b) {
return a / gcd(a, b) * b; }

    // O(sqrt(n))
    static long long phi(long long n) {
        long long r = n;
        for (long long p = 2; p * p <= n; p++) {
            if (n % p == 0) {
                while (n % p == 0) n /= p;
                r -= r / p;
            }
        }
        if (n > 1) r -= r / n;
        return r;
    }

    // O(k * log(mod_i))
    static long long CRT(const vector<long long>& rem,
const vector<long long>& modv) {
        long long x = 0, prod = 1;
        for (auto m : modv) prod *= m;
        for (size_t i = 0; i < rem.size(); i++) {
            long long pp = prod / modv[i];
            x = (x + rem[i] * modInverse(pp % modv[i],
modv[i]) * pp) % prod;
        }
        return (x + prod) % prod;
    }

    // ---------- Primality ----------
    // O(log^3 n)
    static bool isPrime(long long n) {
        if (n < 2) return false;
        for (long long p :
{2,3,5,7,11,13,17,19,23,29,31,37})
            if (n % p == 0) return n == p;

        long long d = n - 1, s = 0;
        while (!(d & 1)) d >>= 1, s++;

        auto check = [&](long long a) {
            if (a % n == 0) return true;
            long long x = modPow(a, d, n);
            if (x == 1 || x == n - 1) return true;
            for (int i = 1; i < s; i++) {
                x = (__int128)x * x % n;
                if (x == n - 1) return true;
            }
            return false;
        };

        for (long long a :
{2,325,9375,28178,450775,9780504,1795265022})
            if (!check(a)) return false;
        return true;
    }

    // ---------- Segmented Sieve ----------
    // O((R-L+1) loglogR + sqrt(R))
    static vector<long long> segmentedSieve(long long
L, long long R) {
        long long lim = sqrt(R) + 1;
        vector<char> mark(lim + 1, 1);
        vector<long long> primes;

        for (long long i = 2; i <= lim; i++) {
            if (mark[i]) {
                primes.push_back(i);
                for (long long j = i * i; j <= lim; j += i) mark[j]
= 0;
            }
        }

        vector<char> isPrimeRange(R - L + 1, 1);
        for (long long p : primes) {
            for (long long j = max(p * p, (L + p - 1) / p * p); j
<= R; j += p)
                isPrimeRange[j - L] = 0;
        }

        vector<long long> res;
        for (long long i = L; i <= R; i++)
            if (isPrimeRange[i - L] && i > 1)
                res.push_back(i);

        return res;
    }
};

// Static variable definitions
vector<long long> Combinatorics::fact;
vector<long long> Combinatorics::invFact;
bool Combinatorics::initDone = false;
long long Combinatorics::mod = Combinatorics::MOD;
int Combinatorics::maxN = Combinatorics::MAXN;


}
```

RANGE QUERIES

FENWICK TREE

BIT_1D

```
inline namespace MY{
template <typename T> struct plusOp{T operator()(const
T&a,const T&b)const{return a+b;}};
template <typename T> struct xorOp{T operator()(const
T&a,const T&b)const{return a^b;}};

template <typename T,typename Op=plusOp<T>>
class BIT_1D{
    vector<T>bit,arr;int n,offset;Op op;bool isXor;
public:
    BIT_1D(int size,bool oneIndexed=false){
        offset=oneIndexed?0:1;n=size+offset;
        bit.assign(n,T(0));arr.assign(size,T(0));
        if(is_same<Op,xorOp<T>>::value)isXor=true;
        else
if(is_same<Op,plusOp<T>>::value)isXor=false;
        else throw runtime_error("Only sum or XOR BIT
supported");
    }

    BIT_1D(const vector<T>&a,bool oneIndexed=false){
        int
len=a.size();offset=oneIndexed?0:1;n=len+offset;
        arr.assign(a.begin(),a.end());bit.assign(n,T(0));
        if(is_same<Op,xorOp<T>>::value)isXor=true;
        else
if(is_same<Op,plusOp<T>>::value)isXor=false;
        else throw runtime_error("Only sum or XOR BIT
supported");
        for(int i=1;i<n;i++){
            bit[i]=op(bit[i],a[i-offset]);
            int j=i+(i&-i);
            if(j<n)bit[j]=op(bit[j],bit[i]);
        }
    }

    // O(log n)
    T getPrefixSum(int i)const{
        T res=T(0);
        while(i>0){res=op(res,bit[i]);i-=i&-i;}
        return res;
    }

    // O(log n)
    T getRangeQuery(int l,int r)const{
        return
isXor?getPrefixSum(r)^getPrefixSum(l-1):getPrefixSum(r
)-getPrefixSum(l-1);
    }

    // O(log n)
    void updatePoint(int i,T delta){
        arr[i-offset]=op(arr[i-offset],delta);
        while(i<n){bit[i]=op(bit[i],delta);i+=i&-i;}
    }

    // O(log n)
    void setValue(int i,T newValue){
        if(!isXor){
            T delta=newValue-arr[i-offset];
            if(delta!=T(0))updatePoint(i,delta);
        }else{
            T delta=newValue^arr[i-offset];
            if(delta!=T(0))updatePoint(i,delta);
        }
        arr[i-offset]=newValue;
    }

    // O(log n)
    int lowerBound(T k)const{
        if(isXor)throw runtime_error("lower_bound only
valid for sum BIT");
        if(k<=T(0))return offset;
        int idx=0,bitMask=1<<(31-__builtin_clz(n-1));
        while(bitMask>0){
            int nxt=idx+bitMask;
            if(nxt<n&&bit[nxt]<k){k-=bit[nxt];idx=nxt;}
            bitMask>>=1;
        }
        return idx+1;
    }

    // O(log n)
    int upperBound(T k)const{
        if(isXor)throw runtime_error("upper_bound only
valid for sum BIT");
        if(k<T(0))return offset;
        int idx=0,bitMask=1<<(31-__builtin_clz(n-1));
        while(bitMask>0){
            int nxt=idx+bitMask;
            if(nxt<n&&bit[nxt]<=k){k-=bit[nxt];idx=nxt;}
            bitMask>>=1;
        }
        return idx+1;
    }

    // O(n)
```

```
    void printArray()const{for(int
i=0;i<(int)arr.size();i++)cout<<arr[i]<<" ";cout<<"\n";}
    // O(n)
    void printBIT()const{for(int i=1;i<n;i++)cout<<bit[i]<<"
";cout<<"\n";}
    // O(n log n)
    void printPrefixSums()const{for(int
i=1;i<n;i++)cout<<getPrefixSum(i)<<" ";cout<<"\n";}
    // O(1)
    bool isXorBIT()const{return isXor;}
};
}
```

BIT_1D_RU

```
inline namespace MY{
template<typename T> struct plusOp{T operator()(T a,T
b)const{return a+b;}};
template<typename T> struct xorOp{T operator()(T a,T
b)const{return a^b;}};

template<typename T,typename Op>
class BIT_1D_RU{
    int n,offset;
    Op op;
    bool isXOR;
    vector<T> BIT1,BIT2;

    // Internal helpers
    void addBIT(vector<T>& BIT,int i,T val){
        for(;i<=n;i+=i&-i) BIT[i]=op(BIT[i],val);
    }

    T queryBIT(const vector<T>& BIT,int i)const{
        T res=0;
        for(;i>0;i-=i&-i) res=op(res,BIT[i]);
        return res;
    }

    void xorPointUpdate(int idx,T val){
        if(idx&1) addBIT(BIT1,idx,val);
        else addBIT(BIT2,idx,val);
    }

    T xorPrefixQueryHelper(int idx)const{
        T res=0;
        res^=queryBIT(BIT1,idx);
        res^=queryBIT(BIT2,idx);
        return res;
    }

public:
    BIT_1D_RU(int size,bool oneIndexed=false):n(size){
        isXOR=is_same<Op,xorOp<T>>::value;
        offset=oneIndexed?0:1;
        BIT1.assign(n+2,0);
        BIT2.assign(n+2,0);
    }

    // O(log n)
    void sumRangeUpdate(int l,int r,T val){
        if(!isXOR){
            l+=offset; r+=offset;
            addBIT(BIT1,l,val);
            addBIT(BIT1,r+1,-val);
            addBIT(BIT2,l,val*(l-1));
            addBIT(BIT2,r+1,-val*r);
        }
    }

    // O(log n)
    T sumPrefixQuery(int i)const{
        if(!isXOR){
            i+=offset;
            return queryBIT(BIT1,i)*i - queryBIT(BIT2,i);
        }
        return 0;
    }

    // O(log n)
    T sumRangeQuery(int l,int r)const{
        if(!isXOR){
            T res_r=sumPrefixQuery(r);
            T res_l=(l>0)?sumPrefixQuery(l-1):0;
            return res_r-res_l;
        }
        return 0;
    }

    // O(log n)
    void sumPointUpdate(int idx,T val){
        sumRangeUpdate(idx,idx,val);
    }

    // O(r-l+1)
    void xorRangeUpdate(int l,int r,T val){
        if(isXOR){
            l+=offset; r+=offset;
            for(int i=l;i<=r;++i) xorPointUpdate(i,val);
        }
    }

    // O(log n)
    T xorPrefixQuery(int i)const{
        if(isXOR){
            i+=offset;
            return xorPrefixQueryHelper(i);
        }
        return 0;
    }

    // O(log n)
    T xorRangeQuery(int l,int r)const{
        if(isXOR){
            T res_r=xorPrefixQuery(r);
            T res_l=(l>0)?xorPrefixQuery(l-1):0;
            return res_r^res_l;
        }
        return 0;
    }

    // O(1)
    void xorPointUpdateSingle(int idx,T val){
        xorRangeUpdate(idx,idx,val);
    }

    // O(n)
    void printBITs()const{
        for(int i=1;i<=n;i++) cout<<BIT1[i]<<" ";
        cout<<"\n";
        for(int i=1;i<=n;i++) cout<<BIT2[i]<<" ";
        cout<<"\n";
    }

    // O(n log n)
    void printPrefixSums()const{
        for(int i=0;i<n;i++){
            if(isXOR) cout<<xorPrefixQuery(i)<<" ";
            else cout<<sumPrefixQuery(i)<<" ";
        }
        cout<<"\n";
    }
};
}
```

BIT_2D

```
inline namespace MY {

    template <typename T>
    class BIT_2D {
    private:
        vector<vector<T>> bit;  // 2D Fenwick tree
        int n, m;

    public:
        // Constructor: empty tree
        BIT_2D(int n, int m) : n(n), m(m) {
            bit.assign(n + 1, vector<T>(m + 1, T(0)));
        }

        // Constructor: build from matrix
        BIT_2D(const vector<vector<T>>& matrix) {
            n = matrix.size();
            m = matrix[0].size();
            bit.assign(n + 1, vector<T>(m + 1, T(0)));

            // Build by inserting each value (O(n·m·log²n))
            for (int i = 1; i <= n; i++) {
                for (int j = 1; j <= m; j++) {
                    update(i, j, matrix[i - 1][j - 1]);
                }
            }
        }

        // O(log n · log m)
        void update(int x, int y, T delta) {
            for (int i = x; i <= n; i += i & -i)
                for (int j = y; j <= m; j += j & -j)
                    bit[i][j] += delta;
        }

        // O(log n · log m)
        T prefixSum(int x, int y) const {
            T sum = T(0);
            for (int i = x; i > 0; i -= i & -i)
                for (int j = y; j > 0; j -= j & -j)
                    sum += bit[i][j];
            return sum;
        }

        // O(log n · log m)
        T rangeSum(int x1, int y1, int x2, int y2) const {
            return prefixSum(x2, y2)
                 - prefixSum(x1 - 1, y2)
                 - prefixSum(x2, y1 - 1)
                 + prefixSum(x1 - 1, y1 - 1);
        }

        // O(log n · log m)
        void setValue(int x, int y, T newValue) {
            T currentValue = rangeSum(x, y, x, y);
            update(x, y, newValue - currentValue);
        }

        // O(n·m·log²n)
        void printPrefixSums() const {
            for (int i = 1; i <= n; i++) {
                for (int j = 1; j <= m; j++)
                    cout << prefixSum(i, j) << " ";
                cout << "\n";
            }
        }

        // O(n·m)
        void printBIT() const {
            for (int i = 1; i <= n; i++) {
                for (int j = 1; j <= m; j++)
```

```
            cout << bit[i][j] << " ";
        cout << "\n";
      }
    }
  };

}
```

BIT_PRODMOD

```
inline namespace MY {

  template<typename T>
  class BIT_prodMod {
  public:
    static constexpr long long DEFAULT_MOD =
1000000007LL;

  private:
    int n;                // number of elements
    T mod;                // modulus (must be prime)
    vector<T> bit;        // multiplicative BIT
    vector<int> bitZ;     // zero-count BIT
    vector<T> arr;        // stored values
    bool externalOneIndexed;  // true if indices
externally are 1-based

    // Convert external index to [1..n]
    inline int toInternal(int idx) const {
      if (idx < 1 || idx > n) throw
std::out_of_range("index out of range");
      return idx;
    }

    // Modular multiplication using __int128_t
    inline T mulMod(T a, T b) const {
      __int128_t x = (__int128_t)a * b;
      return (T)(x % mod);
    }

    // Modular exponentiation — O(log e)
    T modPow(T a, long long e) const {
      a %= mod;
      if (a < 0) a += mod;
      T res = 1;
      while (e > 0) {
        if (e & 1) res = mulMod(res, a);
        a = mulMod(a, a);
        e >>= 1;
      }
      return res;
    }

    // BIT internal multiplicative ops — O(log n)
    void mulUpdateInternal(int idx, T val) {
      for (; idx <= n; idx += idx & -idx)
        bit[idx] = mulMod(bit[idx], val);
    }

    // BIT prefix product — O(log n)
    T mulQueryInternal(int idx) const {
      T res = 1;
      for (; idx > 0; idx -= idx & -idx)
        res = mulMod(res, bit[idx]);
      return res;
    }

    // BIT zero-count update — O(log n)
    void addCountInternal(int idx, int delta) {
      for (; idx <= n; idx += idx & -idx)
        bitZ[idx] += delta;
    }

    // BIT zero-count prefix sum — O(log n)
    int sumCountInternal(int idx) const {
      int s = 0;
      for (; idx > 0; idx -= idx & -idx)
        s += bitZ[idx];
      return s;
    }

    // Build BIT from input vector — O(n log n)
    void buildFromVector(const vector<T>& input) {
      n = (int)input.size();
      bit.assign(n + 1, 1);
      bitZ.assign(n + 1, 0);
      arr.assign(n + 1, 1);

      for (int i = 1; i <= n; ++i) {
        T v = input[i - 1] % mod;
        if (v < 0) v += mod;
        arr[i] = v;
        if (v == 0) {
          addCountInternal(i, 1);
          bit[i] = 1;
        } else {
          bit[i] = v;
        }
      }
      for (int i = 1; i <= n; ++i) {
        int j = i + (i & -i);
        if (j <= n) bit[j] = mulMod(bit[j], bit[i]);
      }
    }

  public:
    // Constructor — empty
    BIT_prodMod(int n_ = 0, T mod_ =
(T)DEFAULT_MOD, bool oneIndexed = true)
        : n(n_), mod(mod_), bit(n_ + 1, 1), bitZ(n_ + 1,
0),
          arr(n_ + 1, 1),
externalOneIndexed(oneIndexed) {}

    // Constructor — build from vector
    BIT_prodMod(const vector<T>& input, T mod_ =
(T)DEFAULT_MOD, bool oneIndexed = true)
        : n(0), mod(mod_), bit(), bitZ(), arr(),
externalOneIndexed(oneIndexed) {
      buildFromVector(input);
    }

    // O(log n)
    void pointUpdate(int idx, T newValueRaw) {
      int i = toInternal(idx);

      T newVal = newValueRaw % mod;
      if (newVal < 0) newVal += mod;

      T old = arr[i] % mod;
      if (old < 0) old += mod;

      if (old == 0 && newVal == 0) {
        arr[i] = newVal;
        return;
      }
      if (old == 0 && newVal != 0) {
        addCountInternal(i, -1);
        mulUpdateInternal(i, newVal);
      } else if (old != 0 && newVal == 0) {
        T invOld = modPow(old, (long long)mod - 2);
        addCountInternal(i, 1);
        mulUpdateInternal(i, invOld);
      } else {
        T invOld = modPow(old, (long long)mod - 2);
        T factor = mulMod(newVal, invOld);
        mulUpdateInternal(i, factor);
      }
      arr[i] = newVal;
    }

    // O(1)
    T getValue(int idx) const {
      int i = toInternal(idx);
      return arr[i];
    }

    // O(log n)
    T prefixProduct(int r) const {
      int ri = toInternal(r);
      return mulQueryInternal(ri);
    }

    // O(log n)
    T rangeQuery(int l, int r) const {
      int li = toInternal(l);
      int ri = toInternal(r);

      int zcount = sumCountInternal(ri) -
sumCountInternal(li - 1);
      if (zcount > 0) return (T)0;

      T prefR = mulQueryInternal(ri);
      T prefLm1 = mulQueryInternal(li - 1);
      T invPrefLm1 = modPow(prefLm1, (long
long)mod - 2);
      return mulMod(prefR, invPrefLm1);
    }

    // O(1)
    int size() const { return n; }

    // O(1)
    T getMod() const { return mod; }

    // O(n)
    void printArray() const {
      for (int i = 1; i <= n; i++) cout << arr[i] << " ";
      cout << "\n";
    }

    // O(n)
    void printBIT() const {
      for (int i = 1; i <= n; i++) cout << bit[i] << " ";
      cout << "\n";
    }

    // O(n log n)
    void printPrefixProducts() const {
      for (int i = 0; i < n; i++)
        cout << rangeQuery(0, i) << " ";
      cout << "\n";
    }

    // O(n log n)
    void printZeroCount() const {
      for (int i = 1; i <= n; i++)
        cout << sumCountInternal(i) << " ";
      cout << "\n";
    }
  };

}
```

SEGMENT TREE

SGTREE

```
inline namespace MY{
template<typename T>
class SegTree1D{
  struct Node{
    T sum,mn,mx,add,assignVal; bool hasAssign; int
sz;

Node():sum(0),mn(numeric_limits<T>::max()),mx(numeri
c_limits<T>::lowest()),add(0),assignVal(0),hasAssign(fal
se),sz(0){}
  };
  int n; vector<Node> tree;

  void ensure_not_empty()const{ if(n==0) throw
runtime_error("SegTree1D: operation on empty tree"); }

  void build(int idx,int l,int r,const vector<T>& arr={}){
    tree[idx].sz=r-l+1; tree[idx].add=0;
tree[idx].hasAssign=false;
    if(l==r){ T val=arr.empty()?0:arr[l]; tree[idx].sum=val;
tree[idx].mn=val; tree[idx].mx=val; return; }
    int mid=(l+r)>>1; build(idx<<1,l,mid,arr);
build(idx<<1|1,mid+1,r,arr); pull(idx);
  }

  void applyAssign(int idx,T val){
    __int128_t tmp=(__int128_t)val*tree[idx].sz;
    tree[idx].sum=(T)tmp; tree[idx].mn=tree[idx].mx=val;
    tree[idx].assignVal=val; tree[idx].hasAssign=true;
tree[idx].add=0;
  }

  void applyAdd(int idx,T val){
    if(tree[idx].hasAssign){
      tree[idx].assignVal+=val; T
newVal=tree[idx].assignVal;
      __int128_t tmp=(__int128_t)newVal*tree[idx].sz;
      tree[idx].sum=(T)tmp;
tree[idx].mn=tree[idx].mx=newVal; return;
    }
    tree[idx].add+=val;
    __int128_t
tmp=(__int128_t)tree[idx].sum+(__int128_t)val*tree[idx].
sz;
    tree[idx].sum=(T)tmp;
    if(tree[idx].mn!=numeric_limits<T>::max())
tree[idx].mn+=val;
    if(tree[idx].mx!=numeric_limits<T>::lowest())
tree[idx].mx+=val;
  }

  void push(int idx){
    if(tree[idx].sz==1) return;
    if(tree[idx].hasAssign){
      applyAssign(idx<<1,tree[idx].assignVal);
      applyAssign(idx<<1|1,tree[idx].assignVal);
      tree[idx].hasAssign=false;
    }
    if(tree[idx].add!=0){
      T v=tree[idx].add;
      applyAdd(idx<<1,v);
      applyAdd(idx<<1|1,v);
      tree[idx].add=0;
    }
  }

  void pull(int idx){
    const Node &L=tree[idx<<1],&R=tree[idx<<1|1];
    tree[idx].sz=L.sz+R.sz;
    __int128 tmp=(__int128)L.sum+(__int128)R.sum;
    tree[idx].sum=(T)tmp;
    tree[idx].mn=min(L.mn,R.mn);
    tree[idx].mx=max(L.mx,R.mx);
  }

  void rangeAssign(int idx,int l,int r,int L,int R,T val){
    if(R<l||r<L) return;
    if(L<=l&&r<=R){ applyAssign(idx,val); return; }
    push(idx); int mid=(l+r)>>1;
    rangeAssign(idx<<1,l,mid,L,R,val);
    rangeAssign(idx<<1|1,mid+1,r,L,R,val);
    pull(idx);
  }

  void rangeAdd(int idx,int l,int r,int L,int R,T val){
    if(R<l||r<L) return;
    if(L<=l&&r<=R){ applyAdd(idx,val); return; }
    push(idx); int mid=(l+r)>>1;
    rangeAdd(idx<<1,l,mid,L,R,val);
    rangeAdd(idx<<1|1,mid+1,r,L,R,val);
    pull(idx);
  }

  T rangeSum(int idx,int l,int r,int L,int R){
    if(R<l||r<L) return T(0);
    if(L<=l&&r<=R) return tree[idx].sum;
    push(idx); int mid=(l+r)>>1;
    __int128 left=rangeSum(idx<<1,l,mid,L,R);
    __int128 right=rangeSum(idx<<1|1,mid+1,r,L,R);
    return (T)(left+right);
  }

  T rangeMin(int idx,int l,int r,int L,int R){
    if(R<l||r<L) return numeric_limits<T>::max();
    if(L<=l&&r<=R) return tree[idx].mn;
    push(idx); int mid=(l+r)>>1;
    return
min(rangeMin(idx<<1,l,mid,L,R),rangeMin(idx<<1|1,mid+
1,r,L,R));
  }

  T rangeMax(int idx,int l,int r,int L,int R){
    if(R<l||r<L) return numeric_limits<T>::lowest();
    if(L<=l&&r<=R) return tree[idx].mx;
    push(idx); int mid=(l+r)>>1;
    return
max(rangeMax(idx<<1,l,mid,L,R),rangeMax(idx<<1|1,mi
d+1,r,L,R));
  }

public:
  SegTree1D(int n_=0):n(n_){ if(n>0){
tree.assign(4*n,Node()); build(1,0,n-1); } }
  SegTree1D(const vector<T>& arr){ init(arr); }

  void init(const vector<T>& arr){ n=(int)arr.size();
tree.assign(4*max(1,n),Node()); if(n>0) build(1,0,n-1,arr);
}

  void rangeAssign(int L,int R,T val){
ensure_not_empty(); rangeAssign(1,0,n-1,L,R,val); }
  void rangeAdd(int L,int R,T val){ ensure_not_empty();
rangeAdd(1,0,n-1,L,R,val); }
  void pointAssign(int pos,T val){ ensure_not_empty();
rangeAssign(1,0,n-1,pos,pos,val); }
  void pointAdd(int pos,T val){ ensure_not_empty();
rangeAdd(1,0,n-1,pos,pos,val); }

  T rangeSum(int L,int R){ ensure_not_empty(); return
rangeSum(1,0,n-1,L,R); }
  T rangeMin(int L,int R){ ensure_not_empty(); return
rangeMin(1,0,n-1,L,R); }
  T rangeMax(int L,int R){ ensure_not_empty(); return
rangeMax(1,0,n-1,L,R); }

  T pointQuery(int pos){
    ensure_not_empty(); int idx=1,l=0,r=n-1;
vector<int> path;
    while(l!=r){ path.push_back(idx); int mid=(l+r)>>1;
      if(pos<=mid){ idx=idx<<1; r=mid; }
      else{ idx=idx<<1|1; l=mid+1; }
    }
    for(int id:path) push(id);
    return tree[idx].sum;
  }
};
}
```

SPARSE TABLE

SPARSE TABLE

```
inline namespace MY{
enum class SparseMode{ IDEMPOTENT, DISJOINT };

template<typename T>
class SparseTableGeneral{
  int n=0;
  int maxLog=0;
  vector<int> lg;
  vector<vector<T>> st;
  function<T(const T&,const T&)> op;
  SparseMode mode;

  void build_logs(int N){
    lg.assign(N+1,0);
    for(int i=2;i<=N;++i) lg[i]=lg[i>>1]+1;
  }

  // O(n log n)
  void build_idempotent(const vector<T>& a){
    n=(int)a.size();
    if(n==0){ st.clear(); return; }
    maxLog = 32 - __builtin_clz((unsigned)n);
    st.assign(maxLog, vector<T>(n));
    for(int i=0;i<n;++i) st[0][i]=a[i];
    for(int k=1;k<maxLog;++k){
      int half = 1 << (k-1);
      int len = 1 << k;
      for(int i=0;i+len<=n;++i){
        st[k][i] = op(st[k-1][i], st[k-1][i+half]);
      }
    }
  }

  // O(n log n)
  void build_disjoint(const vector<T>& a){
    n=(int)a.size();
    if(n==0){ st.clear(); return; }
    maxLog = 32 - __builtin_clz((unsigned)n);
    st.assign(maxLog, vector<T>(n));
    for(int i=0;i<n;++i) st[0][i]=a[i];
    for(int k=1;k<maxLog;++k){
      int len = 1 << k;
      int block = len << 1;
      for(int left=0; left<n; left+=block){
        int mid = min(left + len, n);
        int right = min(left + block, n);
        if(mid < right){
          st[k][mid] = a[mid];
          for(int i=mid+1;i<right;++i) st[k][i] =
op(st[k][i-1], a[i]);
        }
        if(mid-1 >= left){
          st[k][mid-1] = a[mid-1];
```

```
            for(int i=mid-2;i>=left;--i) st[k][i] = op(a[i],
st[k][i+1]);
        }
      }
    }
  }

public:
  // Constructor: O(n log n)
  SparseTableGeneral(const vector<T>& a,
SparseMode buildMode = SparseMode::IDEMPOTENT,
function<T(const T&,const T&)> operation =
function<T(const T&,const T&)>())
    : op(operation), mode(buildMode)
  {
    if(!op) op = [](const T& x, const T& y)->T{ return
(x<y)?x:y; };
    build_logs((int)a.size());
    if(mode==SparseMode::IDEMPOTENT)
build_idempotent(a); else build_disjoint(a);
  }

  // Rebuild with new array: O(n log n)
  void rebuild(const vector<T>& a, SparseMode
buildMode = SparseMode::IDEMPOTENT){
    mode = buildMode;
    build_logs((int)a.size());
    if(mode==SparseMode::IDEMPOTENT)
build_idempotent(a); else build_disjoint(a);
  }

  // Query [l, r], 0-based inclusive — O(1)
  T query(int l,int r) const{
    if(n==0) throw out_of_range("SparseTableGeneral:
empty table");
    if(l<0||r<0||l>=n||r>=n||l>r) throw
out_of_range("SparseTableGeneral::query - invalid
range");
    if(mode==SparseMode::IDEMPOTENT){
      int len = r - l + 1;
      int k = lg[len];
      return op(st[k][l], st[k][r - (1<<k) + 1]);
    } else {
      if(l==r) return st[0][l];
      int x = l ^ r;
      int k = 31 - __builtin_clz(x);
      return op(st[k][l], st[k][r]);
    }
  }

  // O(1)
  int size() const noexcept { return n; }

  // Debug print — O(n log n)
  void debug_print() const{
    if(n==0){ cout << "(empty)\n"; return; }
    cout << "mode = " <<
(mode==SparseMode::IDEMPOTENT ?
"IDEMPOTENT" : "DISJOINT") << "\n";
    for(int k=0;k<maxLog;++k){
      cout << "k=" << k << " (len=" << (1<<k) << "): ";
      for(int i=0;i<n;++i){
        if(mode==SparseMode::IDEMPOTENT){
          if(i + (1<<k) <= n) cout << st[k][i] << " ";
          else cout << "_ ";
        } else {
          cout << st[k][i] << " ";
        }
      }
      cout << "\n";
    }
  }
};
```

SQRT DCMP

```
}

inline namespace MY{
  class SqrtDecomposition{
  public:
    int n, blockSize, numBlocks;
    vector<long long> arr, blockValue, lazy;
    function<long long(const long long&, const long
long&)> op;
    long long identity;
    bool isSum;

    // O(n)
    SqrtDecomposition(const vector<long long>& a,
                      function<long long(const long long&,
const long long&)> operation,
                      long long identityElem,
                      bool isSumQuery=false)
      : arr(a), op(operation), identity(identityElem),
isSum(isSumQuery){
      n = arr.size();
      blockSize = max(1, (int)sqrt(n));
      numBlocks = (n + blockSize - 1) / blockSize;
      blockValue.assign(numBlocks, identity);
      lazy.assign(numBlocks, 0);
      for(int b=0; b<numBlocks; b++) rebuildBlock(b);
    }

    // O(√n)
    void pointUpdate(int i, long long val){
      int b = i / blockSize;
      arr[i] = val - lazy[b];
      rebuildBlock(b);
    }

    // O(√n)
    void rangeAdd(int l, int r, long long delta){
      if(!isSum) return;
      int bL = l / blockSize, bR = r / blockSize;
      if(bL == bR){
        for(int i=l; i<=r; i++) arr[i] += delta;
        rebuildBlock(bL);
      }else{
        for(int i=l; i<(bL+1)*blockSize; i++) arr[i] +=
delta;
        rebuildBlock(bL);
        for(int b=bL+1; b<bR; b++) lazy[b] += delta;
        for(int i=bR*blockSize; i<=r; i++) arr[i] += delta;
        rebuildBlock(bR);
      }
    }

    // O(√n)
    long long query(int l, int r){
      int bL = l / blockSize, bR = r / blockSize;
      long long res = identity;
      if(bL == bR){
        for(int i=l; i<=r; i++) res = op(res, arr[i] +
lazy[bL]);
      }else{
        for(int i=l; i<(bL+1)*blockSize; i++) res =
op(res, arr[i] + lazy[bL]);
        for(int b=bL+1; b<bR; b++){
          if(isSum) res += blockValue[b] +
lazy[b]*blockSize;
          else for(int i=b*blockSize; i<min(n,
(b+1)*blockSize); i++) res = op(res, arr[i] + lazy[b]);
        }
        for(int i=bR*blockSize; i<=r; i++) res = op(res,
arr[i] + lazy[bR]);
      }
      return res;
    }

  private:
    // O(√n)
    void rebuildBlock(int b){
      int l = b * blockSize;
      int r = min(n, (b + 1) * blockSize);
      blockValue[b] = identity;
      for(int i=l; i<r; i++) blockValue[b] =
op(blockValue[b], arr[i]);
    }
  };
}
```

MOS SQRT

```
inline namespace MY{
  template<typename T, typename AnswerType=long
long>
  class MosAlgorithm{
  public:
    int n, blockSize;
    const vector<T>& arr;
    vector<AnswerType> answers;
    AnswerType currentAnswer=0;
    vector<int> freq;

    struct Query{
      int l, r, idx;
      Query(int l,int r,int idx):l(l),r(r),idx(idx){}
    };
    vector<Query> queries;

    // O(n)
    MosAlgorithm(const vector<T>&
a):n(a.size()),arr(a){
      blockSize=max(1,(int)sqrt(n));
      freq.assign(1e6+5,0);
    }

    // O(1)
    void addQuery(int l,int r,int idx){
      queries.emplace_back(l,r,idx);
    }

    // O(1)
    void add(int idx){
      int x=arr[idx];
      freq[x]++;
      if(freq[x]==1) currentAnswer++;
    }

    // O(1)
    void remove(int idx){
      int x=arr[idx];
      freq[x]--;
      if(freq[x]==0) currentAnswer--;
    }

    // O((n+q)*√n)
    vector<AnswerType> process(){
      int q=queries.size();
      answers.assign(q,0);
      sort(queries.begin(),queries.end(),[&](const
Query& a,const Query& b){
        int blockA=a.l/blockSize, blockB=b.l/blockSize;
        if(blockA!=blockB) return blockA<blockB;
        return (blockA&1)?(a.r>b.r):(a.r<b.r);
      });
      int L=0,R=-1;
```

```
        for(auto &qq:queries){
            while(L>qq.l) add(--L);
            while(R<qq.r) add(++R);
            while(L<qq.l) remove(L++);
            while(R>qq.r) remove(R--);
            answers[qq.idx]=currentAnswer;
        }
        return answers;
    }
};
}
```

MOS UPD

```
inline namespace MY{
template<typename T, typename AnswerType=long long>
class MosAlgorithmWithUpdates{
public:
    struct Query{int l,r,t,idx;Query(int l,int r,int t,int idx):l(l),r(r),t(t),idx(idx){}};
    struct Update{int pos;T oldVal,newVal;Update(int pos,T oldVal,T newVal):pos(pos),oldVal(oldVal),newVal(newVal){}};

    int n,blockSize,L=0,R=-1,T=0;
    const vector<T>& originalArray;
    vector<T> arr;
    vector<Query> queries;
    vector<Update> updates;
    vector<AnswerType> answers;
    AnswerType currentAnswer=0;
    vector<int> freq;

    // O(n)
    explicit MosAlgorithmWithUpdates(const vector<T>& a):n(a.size()),originalArray(a),arr(a){
        blockSize=cbrt(max(1,n));
        freq.assign(1e6+5,0);
    }

    // O(1)
    void addQuery(int l,int r,int idx){
        queries.emplace_back(l,r,(int)updates.size(),idx);
    }

    // O(1)
    void addUpdate(int pos,T newValue){
        updates.emplace_back(pos,arr[pos],newValue);
        arr[pos]=newValue;
    }

    // O(1)
    void add(int idx){
        int x=arr[idx];
        freq[x]++;
        if(freq[x]==1) currentAnswer++;
    }

    // O(1)
    void remove(int idx){
        int x=arr[idx];
        freq[x]--;
        if(freq[x]==0) currentAnswer--;
    }

    // O(1)
    void applyUpdate(int updIdx,bool forward){
        int pos=updates[updIdx].pos;
        T from=forward?updates[updIdx].oldVal:updates[updIdx].newVal;
        T to=forward?updates[updIdx].newVal:updates[updIdx].oldVal;
        if(L<=pos && pos<=R){
            remove(pos);
            arr[pos]=to;
            add(pos);
        }else arr[pos]=to;
    }

    // O((n+q)*n^(2/3))
    vector<AnswerType> process(){
        arr=originalArray;
        int q=queries.size();
        answers.assign(q,0);
        sort(queries.begin(),queries.end(),[&](const Query&a,const Query&b){
            int blockA=a.l/blockSize,blockB=b.l/blockSize;
            if(blockA!=blockB) return blockA<blockB;
            int blockRA=a.r/blockSize,blockRB=b.r/blockSize;
            if(blockRA!=blockRB) return blockRA<blockRB;
            return a.t<b.t;
        });
        arr=originalArray;L=0;R=-1;T=0;
        for(auto &qq:queries){
            while(T<qq.t) applyUpdate(T++,true);
            while(T>qq.t) applyUpdate(--T,false);
            while(L>qq.l) add(--L);
            while(R<qq.r) add(++R);
            while(L<qq.l) remove(L++);
            while(R>qq.r) remove(R--);
            answers[qq.idx]=currentAnswer;
        }
        return answers;
    }
};
}
```

STR ALGOS

AHO CORASICK

```
inline namespace MY{
    class AC_NODE{
    public:
        unordered_map<char,AC_NODE*> children;
        AC_NODE *failure_link,*output_link;
        vector<int> pat_inds;
        char data;
        AC_NODE(char ch){ data=ch; failure_link=output_link=nullptr; }
    };

    class AC_TRIE{
        AC_NODE *root;
    public:
        AC_TRIE(){ root=new AC_NODE('\0'); }

        // O(total_nodes)
        void build_failure_output_links(){
            queue<AC_NODE*> bfsq;
            for(auto &p:root->children) bfsq.push(p.second);
            while(!bfsq.empty()){
                AC_NODE *fr=bfsq.front(); bfsq.pop();
                for(auto &p:fr->children){
                    AC_NODE *flink=fr->failure_link;
                    do{

if(flink->children.find(p.first)!=flink->children.end()){

p.second->failure_link=flink->children[p.first];
                            break;
                        }
                        flink=flink->failure_link;
                    } while(flink);
                    bfsq.push(p.second);
                }
                if(!fr->failure_link->pat_inds.empty())
fr->output_link=fr->failure_link;
                else
fr->output_link=fr->failure_link->output_link;
            }
        }

        // O(len(word))
        void insert(const string &word,const int &ind){
            AC_NODE* ptr=root;
            for(const char &c:word){
                if(ptr->children.find(c)==ptr->children.end())
                    ptr->children[c]=new AC_NODE(c);
                ptr=ptr->children[c];
                ptr->failure_link=root;
            }
            ptr->pat_inds.push_back(ind);
        }

        // O(total_length)
        void buildTrie(const vector<string>&vec){
            for(int i=0;i<(int)vec.size();i++) insert(vec[i],i);
        }

        // O(text_length+total_matches)
        vector<vector<int>> acocc(const string&text,const vector<string>&pat){
            int sz=pat.size(),n=text.size();
            vector<vector<int>> occ(sz);
            AC_NODE *ptr=root;
            for(int i=0;i<n;i++){
                char ch=text[i];
                while(ptr!=root &&
ptr->children.find(ch)==ptr->children.end())
                    ptr=ptr->failure_link;
                if(ptr->children.find(ch)!=ptr->children.end())
                    ptr=ptr->children[ch];
                for(int idx:ptr->pat_inds)

occ[idx].push_back(i-(int)pat[idx].length()+1);
                AC_NODE *mol=ptr->output_link;
                while(mol){
                    for(int idx:mol->pat_inds)

occ[idx].push_back(i-(int)pat[idx].length()+1);
                    mol=mol->output_link;
                }
            }
            return occ;
        }

        // O(total_nodes)
        void freeNode(AC_NODE* node){
            for(auto &p:node->children) freeNode(p.second);
            delete node;
        }

        ~AC_TRIE(){ freeNode(root); }
    };
}
```

KMP

```
inline namespace MY{
    class KMP{
```

```
  public:
    // O(m)
    static vector<int> buildLPS(const string &pat){
      int m=pat.length(),j=0,i=1;
      vector<int>LPS(m,0);
      while(i<m){
        if(pat[i]==pat[j]) LPS[i]=++j,i++;
        else if(j==0) LPS[i++]=0;
        else j=LPS[j-1];
      }
      return LPS;
    }

    // O(n+m)
    static vector<int> kmpocc(const string &text,const string &pat){
      int n=text.length(),m=pat.length(),i=0,j=0;
      if(n<m||m==0) return {};
      vector<int>occ,LPS=buildLPS(pat);
      while(i<n){
        if(text[i]==pat[j]){
          i++; j++;
          if(j==m) occ.push_back(i-m),j=LPS[j-1];
        }else if(j==0) i++;
        else j=LPS[j-1];
      }
      return occ;
    }
  };
}
```

MANACHERS

```
inline namespace MY{
  class Manachers{
  public:
    // O(n)
    static string transform(const string &str){
      string s="#";
      for(const char &c:str) s.push_back(c),s.push_back('#');
      return s;
    }

    // O(n)
    static string lonPalindrome(const string &str){
      if(str.empty()) return "";
      string t=transform(str); int n=t.size(),l=0,r=0,center=0,maxLen=0;
      vector<int>p(n,0);
      for(int i=1;i<n;i++){
        int k;
        if(i>r) k=0;
        else{
          int j=l+(r-i);
          if(j-p[j]>l){ p[i]=p[j]; continue; }
          else k=r-i;
        }
        while(i-k>=0&&i+k<n&&t[i-k]==t[i+k]) k++;
        k--; l=i-k; r=i+k; p[i]=k;
        if(p[i]>maxLen) maxLen=p[i],center=i;
      }
      int start=(center-maxLen)/2;
      return str.substr(start,maxLen);
    }
  };
}
```

RABIN KARP

```
inline namespace MY{
  class RABINKARP{
    static const int RK_RADIX_1=31,RK_RADIX_2=53;
    static const int RK_MOD_1=1e9+7,RK_MOD_2=1e9+9;
  public:
    // O(m)
    static pair<int,int> hashPair(const string &str){
      long long h1=0,h2=0;
      for(char c:str){
        h1=((__int128)h1*RK_RADIX_1+(unsigned char)c)%RK_MOD_1;
        h2=((__int128)h2*RK_RADIX_2+(unsigned char)c)%RK_MOD_2;
      }
      return {h1,h2};
    }

    // O(n+m)
    static vector<int> rkocc(const string &text,const string &pat){
      int n=text.size(),m=pat.size();
      if(n<m||m==0) return {};
      vector<int>occ;
      auto patHash=hashPair(pat);
      auto txtHash=hashPair(text.substr(0,m));
      long long pow1=1,pow2=1;
      for(int i=1;i<m;i++){
        pow1=((__int128)pow1*RK_RADIX_1)%RK_MOD_1;
        pow2=((__int128)pow2*RK_RADIX_2)%RK_MOD_2;
      }
      if(txtHash==patHash) occ.push_back(0);
      for(int i=1;i<=n-m;i++){
        txtHash.first=(txtHash.first-(__int128)(unsigned char)text[i-1]*pow1)%RK_MOD_1;
        txtHash.second=(txtHash.second-(__int128)(unsigned char)text[i-1]*pow2)%RK_MOD_2;
        if(txtHash.first<0) txtHash.first+=RK_MOD_1;
        if(txtHash.second<0) txtHash.second+=RK_MOD_2;
        txtHash.first=((__int128)txtHash.first*RK_RADIX_1+(unsigned char)text[i+m-1])%RK_MOD_1;
        txtHash.second=((__int128)txtHash.second*RK_RADIX_2+(unsigned char)text[i+m-1])%RK_MOD_2;
        if(txtHash==patHash) occ.push_back(i);
      }
      return occ;
    }
  };
}
```

RK_HASHFUNC

```
inline namespace MY{
  class HashFunc{
    const int MOD1=1e9+7,MOD2=1e9+9,P1=131,P2=257;
    vector<int> hash1,hash2,pow1,pow2;

  public:
    // O(n)
    HashFunc(const string &s){
      int n=s.size(); if(!n) return;
      hash1.assign(n,0); hash2.assign(n,0);
      pow1.assign(n,1); pow2.assign(n,1);
      hash1[0]=(unsigned char)s[0];
      hash2[0]=(unsigned char)s[0];
      for(int i=1;i<n;i++){
        pow1[i]=(1LL*pow1[i-1]*P1)%MOD1;
        pow2[i]=(1LL*pow2[i-1]*P2)%MOD2;
        hash1[i]=((1LL*hash1[i-1]*P1)+(unsigned char)s[i])%MOD1;
        hash2[i]=((1LL*hash2[i-1]*P2)+(unsigned char)s[i])%MOD2;
      }
    }

    // O(1)
    pair<int,int> getHash(int l,int r,bool isOneIdx){
      l=(isOneIdx?l-1:l);
      r=(isOneIdx?r-1:r);
      int h1=hash1[r],h2=hash2[r];
      if(l>0){
        int len=r-l+1;
        h1=(h1-(1LL*hash1[l-1]*pow1[len])%MOD1+MOD1)%MOD1;
        h2=(h2-(1LL*hash2[l-1]*pow2[len])%MOD2+MOD2)%MOD2;
      }
      return {h1,h2};
    }
  };
}
```

Z ALGO

```
inline namespace MY{
  class ZAlgo{
  public:
    // O(1)
    static char charAt(const string &pattern,const string &text,int idx){
      int m=pattern.size();
      if(idx<m) return pattern[idx];
      if(idx==m) return '$';
      return text[idx-m-1];
    }

    // O(n+m)
    static vector<int> zocc(const string &text,const string &pattern){
      int m=pattern.size(),n=text.size();
      vector<int> occ; if(!m||n<m) return occ;

      vector<int> Z(m,0);
      int L=0,R=0;
      for(int i=1;i<m;i++){
        if(i<=R) Z[i]=min(R-i+1,Z[i-L]);
        while(i+Z[i]<m&&pattern[Z[i]]==pattern[i+Z[i]]) Z[i]++;
        if(i+Z[i]-1>R) L=i,R=i+Z[i]-1;
      }

      L=R=-1;
      for(int i=0;i<n;i++){
        int k=0;
        if(i<=R) k=min(R-i+1,Z[i-L]);
        while(k<m&&i+k<n&&pattern[k]==text[i+k]) k++;
        if(i+k-1>R) L=i,R=i+k-1;
        if(k==m) occ.push_back(i);
      }
      return occ;
    }
  };
}
```