Codebook- Team Far_Behind
IIT Delhi, India

Ayush Ranjan, Naman Jain, Manish Tanwar

Contents

1 Syntax

1.1 Template

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

template<class T> ostream& operator<<(ostream &os, vector<T> V) {
 os << "[ "; for(auto v : V) os << v << " "; return os << "]";}
template<class L, class R> ostream& operator<<(ostream &os, pair<L,R> P) {
 return os << "(" << P.first << "," << P.second << ")";}

#define TRACE
#ifdef TRACE
#define trace(...) __f(#__VA_ARGS__, __VA_ARGS__)
template <typename Arg1>
void __f(const char* name, Arg1&& arg1){
 cout << name << " : " << arg1 << std::endl;
}
template <typename Arg1, typename... Args>
void __f(const char* names, Arg1&& arg1, Args&&... args){
 const char* comma = strchr(names + 1, ',');cout.write(names, comma - names) << " : " << arg1<<" | ";__f(comma+1, args...);
}
#else
#define trace(...) 1
#endif

#define ll long long
#define ld long double
#define vll vector<ll>
#define pll pair<ll,ll>
#define vpll vector<pll>
#define I insert
#define pb push_back
#define F first
#define S second
#define all(x) x.begin(),x.end()
#define endl "\n"
// const ll MAX=1e6+5;

// int mod=1e9+7;
```

```cpp
inline int mul(int a,int b){return (a*1ll*b)%mod;}
inline int add(int a,int b){a+=b;if(a>=mod)a-=mod;return a;}
inline int sub(int a,int b){a-=b;if(a<0)a+=mod;return a;}
inline int power(int a,int b){int rt=1;while(b>0){if(b&1)rt=mul(rt,a);a=mul(a,a);b>>=1;}return rt;}
inline int inv(int a){return power(a,mod-2);}
inline void modadd(int &a,int b){a+=b;if(a>=mod)a-=mod;}
int main(){
 ios_base::sync_with_stdio(false);cin.tie(0);cout.tie(0);cout<<setprecision(25);
}
```

```cpp
// clock
clock_t clk = clock();
clk = clock() - clk;
((ld)clk)/CLOCKS_PER_SEC
```

```cpp
// fastio
inline ll read() {
    ll n = 0; char c = getchar_unlocked();
    while (!('0' <= c && c <= '9')) c = getchar_unlocked();
    while ('0' <= c && c <= '9')
      n = n * 10 + c - '0', c = getchar_unlocked();
    return n;
}
inline void write(ll a){
    register char c; char snum[20]; ll i=0;
    do{
      snum[i++]=a%10+48;
      a=a/10;
    }
    while(a!=0); i--;
    while(i>=0)
      putchar_unlocked(snum[i--]);
    putchar_unlocked('\n');
}
using getline, use cin.ignore()
```

```cpp
// gp_hash_table
#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;
gp_hash_table<int, int> table; //cc_hash_table can also be used
//custom hash function
const int RANDOM = chrono::high_resolution_clock::now().time_since_epoch().count();
struct chash {
    int operator()(int x) { return hash<int>{}(x ^ RANDOM); }
};
gp_hash_table<int, int, chash> table;
//custom hash function for pair
struct chash {
    int operator()(pair<int,int> x) const { return x.first* 31 + x.second; }
};
```

```cpp
// random
mt19937 rng(chrono::steady_clock::now().time_since_epoch().count());
uniform_int_distribution<int> uid(l,r);
int x=uid(rng);
//mt19937_64 rng(chrono::steady_clock::now().time_since_epoch().count());
 // - for 64 bit unsigned numbers
vector<int> per(N);
for (int i = 0; i < N; i++)
    per[i] = i;
shuffle(per.begin(), per.end(), rng);
```

```cpp
// string splitting
// this splitting is better than custom function(w.r.t time)
string line = "Ge";
vector <string> tokens;
stringstream check1(line);
string ele;
// Tokenizing w.r.t. space ' '
while(getline(check1, ele, ' '))
tokens.push_back(ele);
```

1.2 C++ Sublime Build

```
{
  "cmd": ["bash", "-c", "g++ -std=c++11 -O3 '${file}' -o '${file_path}/${file_base_name}' && gnome-terminal -- bash -c '\"${file_path}/${file_base_name}\" < input.txt >output.txt' "],
  "file_regex": "^(..[^:]*):([0-9]+):?([0-9]+)?:? (.*)$",
  "working_dir": "${file_path}",
  "selector": "source.c++, source.cpp",
}
```

2 Data Structures

2.1 Fenwick

```cpp
/*All indices are 1 indexed*/
/*Range update and point query: maintain BIT of prefix sum of updates
 to add val in range [a,b] add val at a and -val at b
 value[a]=BITsum(a)+arr[a] where arr is constant*/
/***Range update and range query: maintain 2 BITs B1 and B2
```

```
**to add val in [a,b] add val at a and -val at b+1 in B1. Add val*(a-1) at a and -val*b at b+1
**sum[1,b]=B1sum(1,b)*b-B2sum(1,b)
**sum[a,b]=sum[1,b]-sum[1,a-1]*/
ll n;
ll fen[MAX_N];
void update(ll p,ll val){
 for(ll i = p;i <= n;i += i & -i)
  fen[i] += val;
}
ll sum(ll p){
 ll ans = 0;
 for(ll i = p;i;i -= i & -i)
  ans += fen[i];
 return ans;
}
```

3 Flows and Matching

3.1 Ford Fulkerson

```
// running time - O(f*m) (f -> flow routed)
const ll N = 3e3;
ll n; // number of vertices
ll capacity[N][N]; // adj matrix for capacity
vll adj[N]; // adj list of the corresponding undirected graph(***imp***)
// E = {1-2,2->3,3->2}, adj list should be => {1->2,2->1,2->3,3->2}
// *** vertices are 0-indexed ***

ll INF = (1e18);
ll snk, cnt; // cnt for vis, no need to initialize vis
vector<ll> par, vis;

ll dfs(ll u,ll curr_flow){
 vis[u] = cnt; if(u == snk) return curr_flow;
 if(adj[u].size() == 0) return 0;

 for(ll j=0;j<5;j++){ // random for good augmentation(**sometimes take time**)
        ll a = rand()%(adj[u].size());
        ll v = adj[u][a];
        if(vis[v] == cnt || capacity[u][v] == 0) continue;
        par[v] = u;
        ll f = dfs(v,min(curr_flow, capacity[u][v])); if(vis[snk] == cnt) return f;
    }
    for(auto v : adj[u]){
     if(vis[v] == cnt || capacity[u][v] == 0) continue;
     par[v] = u;
     ll f = dfs(v,min(curr_flow, capacity[u][v])); if(vis[snk] == cnt) return f;
    }
    return 0;
}

ll maxflow(ll s, ll t) {
    snk = t; ll flow = 0; cnt++;
    par = vll(n,-1); vis = vll(n,0);
    while(ll new_flow = dfs(s,INF)){
     flow += new_flow; cnt++;
     ll cur = t;
     while(cur != s){
      ll prev = par[cur];
      capacity[prev][cur] -= new_flow;
      capacity[cur][prev] += new_flow;
      cur = prev;
     }
    }
    return flow;
}
```

3.2 Dinic

```
/*Time: O(m*n^2) and for any unit capacity network O(m * n^1/2)
TIme: O(min(fm,mn^2)) (f: flow routed)
(so for bipartite matching as well)
In practice it is pretty fast for any bipartite network
I/O:      n -> vertice; DinicFlow net(n);
          for(z : edges) net.addEdge(z.F,z.S,cap);
          max flow = maxFlow(s,t);
e=(u,v), e.flow represents the effective flow from u to v
(i.e f(u->v) - f(v->u)), vertices are 1-indexed
*** use int if possible(ll could be slow in dinic) *** */
struct edge {ll x, y, cap, flow;};
struct DinicFlow {
    // *** change inf accordingly *****
    const ll inf = (1e18);
    vector <edge> e;
    vector <ll> cur, d;
    vector < vector <ll> > adj;
    ll n, source, sink;
    DinicFlow() {}
    DinicFlow(ll v) {
        n = v;
        cur = vector <ll> (n + 1);
        d = vector <ll> (n + 1);
        adj = vector < vector <ll> > (n + 1);
    }
    void addEdge(ll from, ll to, ll cap) {
        edge e1 = {from, to, cap, 0};
        edge e2 = {to, from, 0, 0};
        adj[from].push_back(e.size()); e.push_back(e1);
        adj[to].push_back(e.size()); e.push_back(e2);
```

```
    }
    ll bfs() {
        queue <ll> q;
        for(ll i = 0; i <= n; ++i) d[i] = -1;
        q.push(source); d[source] = 0;
        while(!q.empty() and d[sink] < 0) {
            ll x = q.front(); q.pop();
            for(ll i = 0; i < (ll)adj[x].size(); ++i↩
                ) {
                ll id = adj[x][i], y = e[id].y;
                if(d[y] < 0 and e[id].flow < e[id].↩
                    cap) {
                    q.push(y); d[y] = d[x] + 1;
                }
            }
        }
        return d[sink] >= 0;
    }
    ll dfs(ll x, ll flow) {
        if(!flow) return 0;
        if(x == sink) return flow;
        for(;cur[x] < (ll)adj[x].size(); ++cur[x]) {
            ll id = adj[x][cur[x]], y = e[id].y;
            if(d[y] != d[x] + 1) continue;
            ll pushed = dfs(y, min(flow, e[id].cap -↩
                e[id].flow));
            if(pushed) {
                e[id].flow += pushed;
                e[id ^ 1].flow -= pushed;
                return pushed;
            }
        }
        return 0;
    }
    ll maxFlow(ll src, ll snk) {
        this->source = src; this->sink = snk;
        ll flow = 0;
        while(bfs()) {
            for(ll i = 0; i <= n; ++i) cur[i] = 0;
            while(ll pushed = dfs(source, inf)) {
                flow += pushed;
            }
        }
        return flow;
    }
};
```

3.3 MCMF

```
// MCMF Theory:
// 1.  If a network with negative costs had no ↩
   negative cycle it is possible to transform it ↩
   into one with nonnegative
//      costs. Using Cij_new(pi) = Cij_old + pi(i) - ↩
   pi(j), where pi(x) is shortest path from s to x ↩
   in network with an
//      added vertex s. The objective value remains ↩
   the same (z_new = z + constant). z(x) = sum(cij*↩
   xij)
//      (x->flow, c->cost, u->cap, r->residual cap).
// 2.  Residual Network: cji = -cij, rij = uij-xij, ↩
   rji = xij.
// 3.  Note: If edge (i,j),(j,i) both are there then↩
    residual graph will have four edges b/w i,j (↩
   pairs of parellel edges).
// 4.  let x* be a feasible soln, its optimal iff ↩
   residual network Gx* contains no negative cost ↩
   cycle.
// 5.  Cycle Cancelling algo => Complexity O(n*m^2*U↩
   *C) (C->max abs value of cost, U->max cap) (m*U*C↩
    iterations).
// 6.  Succesive shortest path algo => Complexity O(↩
   n^3 * B) / O(nmBlogn)(using heap in Dijkstra)(B ↩
   -> largest supply node).

//Works for negative costs, but does not work for ↩
   negative cycles
//Complexity: O(min(E^2 *V log V, E logV * flow))
// to use -> graph G(n), G.add_edge(u,v,cap,cost), G↩
   .min_cost_max_flow(s,t)
// ********* INF is used in both flow_type and ↩
   cost_type so change accordingly
const ll INF = 99999999;
// vertices are 0-indexed
struct graph {
  typedef ll flow_type; // **** flow type ****
  typedef ll cost_type; // **** cost type ****
  struct edge {
    int src, dst;
    flow_type capacity, flow;
    cost_type cost;
    size_t rev;
  };
  vector<edge> edges;
  void add_edge(int src, int dst, flow_type cap, ↩
     cost_type cost) {
    adj[src].push_back({src, dst, cap, 0, cost, adj[↩
       dst].size()});
    adj[dst].push_back({dst, src, 0, 0, -cost, adj[↩
       src].size()-1});
  }
  int n;
  vector<vector<edge>> adj;
  graph(int n) : n(n), adj(n) { }

  pair<flow_type, cost_type> min_cost_max_flow(int s↩
     , int t) {
    flow_type flow = 0;
    cost_type cost = 0;

    for (int u = 0; u < n; ++u) // initialize
      for (auto &e: adj[u]) e.flow = 0;

    vector<cost_type> p(n, 0);
```

```
auto rcost = [&](edge e) { return e.cost + p[e.↩
   src] - p[e.dst]; };
for (int iter = 0; ; ++iter) {
  vector<int> prev(n, -1); prev[s] = 0;
  vector<cost_type> dist(n, INF); dist[s] = 0;
  if (iter == 0) { // use Bellman-Ford to remove↩
      negative cost edges
    vector<int> count(n); count[s] = 1;
    queue<int> que;
    for (que.push(s); !que.empty(); ) {
      int u = que.front(); que.pop();
      count[u] = -count[u];
      for (auto &e: adj[u]) {
        if (e.capacity > e.flow && dist[e.dst] >↩
            dist[e.src] + rcost(e)) {
          dist[e.dst] = dist[e.src] + rcost(e);
          prev[e.dst] = e.rev;
          if (count[e.dst] <= 0) {
            count[e.dst] = -count[e.dst] + 1;
            que.push(e.dst);
          }
        }
      }
    }
    for(int i=0;i<n;i++) p[i] = dist[i]; // ↩
       added it
    continue;
  } else { // use Dijkstra
    typedef pair<cost_type, int> node;
    priority_queue<node, vector<node>, greater<↩
       node>> que;
    que.push({0, s});
    while (!que.empty()) {
      node a = que.top(); que.pop();
      if (a.S == t) break;
      if (dist[a.S] > a.F) continue;
      for (auto e: adj[a.S]) {
        if (e.capacity > e.flow && dist[e.dst] >↩
            a.F + rcost(e)) {
          dist[e.dst] = dist[e.src] + rcost(e);
          prev[e.dst] = e.rev;
          que.push({dist[e.dst], e.dst});
        }
      }
    }
  }
  if (prev[t] == -1) break;

  for (int u = 0; u < n; ++u)
    if (dist[u] < dist[t]) p[u] += dist[u] - ↩
       dist[t];

  function<flow_type(int,flow_type)> augment = ↩
     [&](int u, flow_type cur) {
    if (u == s) return cur;
    edge &r = adj[u][prev[u]], &e = adj[r.dst][r↩
       .rev];
    flow_type f = augment(e.src, min(e.capacity ↩
       - e.flow, cur));
        e.flow += f; r.flow -= f;
        return f;
      };
      flow_type f = augment(t, INF);
      flow += f;
      cost += f * (p[t] - p[s]);
    }
    return {flow, cost};
  }
};
```

4 Geometry

4.1 Convex Hull

```
pt firstpoint;
//for sorting points in ccw(counter clockwise) ↩
   direction w.r.t firstpoint (leftmost and ↩
   bottommost)
bool compare(pt x,pt y){
 ll o=orient(firstpoint,x,y);
 if(o==0)return lt(x.x+x.y,y.x+y.y);
 return o<0;
}
/*takes as input a vector of points containing input↩
    points and an empty vector for making hull
the points forming convex hull are pushed in vector ↩
   hull
returns hull containing minimum number of points in ↩
   ccw order
****remove EPS for making integer hull
*/
void make_hull(vector<pt>& poi,vector<pt>& hull)
{
 pair<ld,ld> bl={INF,INF};
 ll n=poi.size();ll ind;
 for(ll i=0;i<n;i++){
  pair<ld,ld> pp={poi[i].y,poi[i].x};
  if(pp<bl){
   ind=i;bl={poi[i].y,poi[i].x};
  }
 }
 swap(bl.F,bl.S);firstpoint=pt(bl.F,bl.S);
 vector<pt> cons;
 for(ll i=0;i<n;i++){
  if(i==ind)continue;cons.pb(poi[i]);
 }
 sort(cons.begin(),cons.end(),compare);
 hull.pb(firstpoint);ll m;
 for(auto z:cons){
  if(hull.size()<=1){hull.pb(z);continue;}
  pt pr,ppr;bool fl=true;
  while((m=hull.size())>=2){
   pr=hull[m-1];ppr=hull[m-2];
   ll ch=orient(ppr,pr,z);
```

```
    if(ch==-1){break;}
    else if(ch==1){hull.pop_back();continue;}
    else {
     ld d1,d2;
     d1=dist2(ppr,pr);d2=dist2(ppr,z);
     if(gt(d1,d2)){fl=false;break;}else {hull.pop_back();}
    }
   }
   if(fl){hull.push_back(z);}
  }
  return;
}
```

4.2 Convex Hull Trick

```
/*
maintains upper convex hull of lines ax+b and gives minimum value at a given x
to add line ax+b: sameoldcht.addline(a,b), to get min value at x: sameoldcht.getbest(x)
to get maximum value at x add -ax-b as lines instead of ax+b and use -sameoldcht.getbest(x)
*/
const int N = 1e5 + 5;
int n;
int a[N];
int b[N];
long long dp[N];
struct line{
    long long a , b;
    double xleft;
    bool type;
    line(long long _a , long long _b){
        a = _a;
        b = _b;
        type = 0;
    }
    bool operator < (const line &other) const{
        if(other.type){
            return xleft < other.xleft;
        }
        return a > other.a;
    }
};
double meet(line x , line y){
    return 1.0 * (y.b - x.b) / (x.a - y.a);
}
struct cht{
    set < line > hull;
    cht(){
        hull.clear();
    }
    typedef set < line > :: iterator ite;
    bool hasleft(ite node){
        return node != hull.begin();
    }
    bool hasright(ite node){
        return node != prev(hull.end());
    }
    void updateborder(ite node){
        if(hasright(node)){
            line temp = *next(node);
            hull.erase(temp);
            temp.xleft = meet(*node , temp);
            hull.insert(temp);
        }
        if(hasleft(node)){
            line temp = *node;
            temp.xleft = meet(*prev(node) , temp);
            hull.erase(node);
            hull.insert(temp);
        }
        else{
            line temp = *node;
            hull.erase(node);
            temp.xleft = -1e18;
            hull.insert(temp);
        }
    }
    bool useless(line left , line middle , line right){
        double x = meet(left , right);
        double y = x * middle.a + middle.b;
        double ly = left.a * x + left.b;
        return y > ly;
    }
    bool useless(ite node){
        if(hasleft(node) && hasright(node)){
            return useless(*prev(node) , *node , *next(node));
        }
        return 0;
    }
    void addline(long long a , long long b){
        line temp = line(a , b);
        auto it = hull.lower_bound(temp);
        if(it != hull.end() && it -> a == a){
            if(it -> b > b){
                hull.erase(it);
            }
            else{
                return;
            }
        }
        hull.insert(temp);
        it = hull.find(temp);
        if(useless(it)){
            hull.erase(it);
            return;
        }
        while(hasleft(it) && useless(prev(it))){
            hull.erase(prev(it));
        }
        while(hasright(it) && useless(next(it))){
            hull.erase(next(it));
```

```
        }
        updateborder(it);
    }
    long long getbest(long long x){
        if(hull.empty()){
            return 1e18;
        }
        line query(0 , 0);
        query.xleft = x;
        query.type = 1;
        auto it = hull.lower_bound(query);
        it = prev(it);
        return it -> a * x + it -> b;
    }
};
cht sameoldcht;
int main()
{
    scanf("%d" , &n);
    for(int i = 1 ; i <= n ; ++i){
        scanf("%d" , a + i);
    }
    for(int i = 1 ; i <= n ; ++i){
        scanf("%d" , b + i);
    }
    sameoldcht.addline(b[1] , 0);
    for(int i = 2 ; i <= n ; ++i){
        dp[i] = sameoldcht.getbest(a[i]);
        sameoldcht.addline(b[i] , dp[i]);
    }
    printf("%lld\n" , dp[n]);
}
```

5 Trees

5.1 LCA

```
const int N = int(1e5)+10;
const int LOGN = 20;
set<int> g[N];
int level[N];
int DP[LOGN][N];
int n,m;
/*----------- Pre-Processing ------------*/
/* Code Cridits : Tanuj Khattar codeforces ↩
   submission */
void dfs0(int u)
{
 for(auto it=g[u].begin();it!=g[u].end();it++)
  if(*it!=DP[0][u])
  {
   DP[0][*it]=u;
   level[*it]=level[u]+1;
   dfs0(*it);
  }
}
void preprocess()
{
 level[0]=0;
 DP[0][0]=0;
 dfs0(0);
 for(int i=1;i<LOGN;i++)
  for(int j=0;j<n;j++)
   DP[i][j] = DP[i-1][DP[i-1][j]];
}
int lca(int a,int b)
{
 if(level[a]>level[b])swap(a,b);
 int d = level[b]-level[a];
 for(int i=0;i<LOGN;i++)
  if(d&(1<<i))
   b=DP[i][b];
 if(a==b)return a;
 for(int i=LOGN-1;i>=0;i--)
  if(DP[i][a]!=DP[i][b])
   a=DP[i][a],b=DP[i][b];
 return DP[0][a];
}
int dist(int u,int v)
{
 return level[u] + level[v] - 2*level[lca(u,v)];
}
```

5.2 Centroid Decompostion

```
/*
nx:maximum number of nodes
adj:adjacency list of tree,adj1: adjacency list of ↩
   centroid tree
par:parents of nodes in centroid tree,timstmp: ↩
   timestamps of nodes when they became centroids (↩
   helpful in comparing which of the two nodes ↩
   became centroid first)
ssize,vis:utility arrays for storing subtree size ↩
   and visit times in dfs
tim: utility for doing dfs (for deciding which nodes↩
    to visit)
cntrorder: centroids stored in order in which they ↩
   were formed
dist[nx]: vector of vectors with dist[i][0][j]=↩
   number of nodes at distance of k in subtree of i ↩
   in centroid tree and dist[i][j][k]=number of ↩
   nodes at distance k in jth child of i in centroid↩
    tree ***(use adj while doing dfs instead of adj1↩
   )***
dfs: find subtree sizes visiting nodes starting from↩
    root without visiting already formed centroids
dfs1: root- starting node, n- subtree size remaining↩
    after removing centroids -> returns centroid in ↩
   subtree of root
preprocess: stores all values in dist array
*/
```

```
const int nx=1e5;
vector<int> adj[nx],adj1[nx]; //adj is adjacency list of tree and adj1 is adjacency list for centroid tree
int par[nx],timstmp[nx],ssize[nx],vis[nx];//par is parent of each node in centroid tree,ssize is subtree size of each node in centroid tree,vis and timstmp are auxillary arrays for visit times in dfs- timstmp contains nonzero values only for centroids
int tim=1;
vector<int> cntrorder;//contains list of centroids generated (in order)
vector<vector<int> > dist[nx];
int dfs(int root)
{
 vis[root]=tim;
 int t=0;
 for(auto i:adj[root])
 {
  if(!timstmp[i]&&vis[i]<tim)
   t+=dfs(i);
 }
 ssize[root]=t+1;return t+1;
}
int dfs1(int root,int n)
{
 vis[root]=tim;pair<int,int> mxc={0,-1};bool poss=true;
 for(auto i:adj[root])
 {
  if(!timstmp[i]&&vis[i]<tim)
   poss&=(ssize[i]<=n/2),mxc=max(mxc,{ssize[i],i});
 }
 if(poss&&(n-ssize[root])<=n/2)return root;
 return dfs1(mxc.second,n);
}
int findc(int root)
{
 dfs(root);
 int n=ssize[root];tim++;
 return dfs1(root,n);
}
void cntrdecom(int root,int p)
{
 int cntr=findc(root);
 cntrorder.push_back(cntr);
 timstmp[cntr]=tim++;
 par[cntr]=p;
 if(p>=0)adj1[p].push_back(cntr);
 for(auto i:adj[cntr])
  if(!timstmp[i])
   cntrdecom(i,cntr);
}
void dfs2(int root,int nod,int j,int dst)
{
 if(dist[root][j].size()==dst)dist[root][j].push_back(0);
 vis[nod]=tim;
 dist[root][j][dst]+=1;
 for(auto i:adj[nod])
 {
  if((timstmp[i]<=timstmp[root])||(vis[i]==vis[nod]))continue;
  vis[i]=tim;dfs2(root,i,j,dst+1);
 }
}
void preprocess()
{
 for(int i=0;i<cntrorder.size();i++)
 {
  int root=cntrorder[i];
  vector<int> temp;
  dist[root].push_back(temp);
  temp.push_back(0);
  ++tim;
  dfs2(root,root,0,0);
  int cnt=0;
  for(int j=0;j<adj[root].size();j++)
  {
   int nod=adj[root][j];
   if(timstmp[nod]<timstmp[root])
    continue;
   dist[root].push_back(temp);
   ++tim;
   dfs2(root,nod,++cnt,1);
  }
 }
}
```

6 Maths

6.1 Chinese Remainder Theorem

```
#include<bits/stdc++.h>
using namespace std;
#define ll long long
/*solves system of equations x=rem[i]%mods[i] for any mod (need not be coprime)
intput:vector of remainders and moduli
output: pair of answer(x%lcm of modulo) and lcm of all the modulo (returns -1 if it is inconsistent)
*/
ll GCD(ll a, ll b) { return (b == 0) ? a : GCD(b, a % b); }
inline ll LCM(ll a, ll b) { return a / GCD(a, b) * b; }
inline ll normalize(ll x, ll mod) { x %= mod; if (x < 0) x += mod; return x; }
struct GCD_type { ll x, y, d; };
GCD_type ex_GCD(ll a, ll b)
{
    if (b == 0) return {1, 0, a};
    GCD_type pom = ex_GCD(b, a % b);
```

```
    return {pom.y, pom.x - a / b * pom.y, pom.d};
}
pair<ll,ll> CRT(vector<ll> &rem,vector<ll> &mods)
{
    ll n=rem.size();
    ll ans=rem[0];
    ll lcm=mods[0];
    for(ll i=1;i<n;i++)
    {
        auto pom=ex_GCD(lcm,mods[i]);
        ll x1=pom.x;
        ll d=pom.d;
        if((rem[i]-ans)%d!=0)return {-1,0};
        ans=normalize(ans+x1*(rem[i]-ans)/d%(mods[i]/d)*lcm,lcm*mods[i]/d);
        lcm=LCM(lcm,mods[i]); // you can save time by replacing above lcm * n[i] /d by lcm = lcm * n[i] / d
    }
    return {ans,lcm};
}
```

6.2 Discrete Log

```
// Discrete Log , Baby-Step Giant-Step , e-maxx
// The idea is to make two functions,
// f1(p) , f2(q) and find p,q s.t.
// f1(p) = f2(q) by storing all possible values of f1,
// and checking for q. In this case a^(x) = b (mod m) is
// solved by substituting x by p.n-q , where
// is choosen optimally , usually sqrt(m).

// credits : https://cp-algorithms.com/algebra/discrete-log.html
// returns a soln. for a^(x) = b (mod m)
// for given a,b,m . -1 if no. soln.
// complexity : O(sqrt(m).log(m))
// use unordered_map to remove log factor.
// IMP : works only if a,m are co-prime. But can be modified.
int solve (int a, int b, int m) {
    int n = (int) sqrt (m + .0) + 1;

    int an = 1;
    for (int i=0; i<n; ++i)
        an = (an * a) % m;

    map<int,int> vals;
    for (int i=1, cur=an; i<=n; ++i) {
        if (!vals.count(cur))
            vals[cur] = i;
        cur = (cur * an) % m;
    }

    for (int i=0, cur=b; i<=n; ++i) {
        if (vals.count(cur)) {
            int ans = vals[cur] * n - i;
            if (ans < m)
                return ans;
        }
        cur = (cur * a) % m;
    }
    return -1;
}
```

6.3 NTT

```
/*
Kevin's different Code: https://s3.amazonaws.com/codechef_shared/download/Solutions/JUNE15/tester/MOREFB.cpp
****There is no problem that FFT can solve while this NTT cannot
 Case1: If the answer would be small choose a small enough NTT prime modulus
 Case2: If the answer is large(> ~1e9) FFT would not work anyway due to precision issues
 In Case2 use NTT. If max_answer_size=n*(largest_coefficient^2)
 So use two or three modulus to solve it
****Compute a*b%mod if a%mod*b%mod would result in overflow in O(log(a)) time:
 ll mulmod(ll a, ll b, ll mod) {
     ll res = 0;
     while (a != 0) {
         if (a & 1) res = (res + b) % m;
         a >>= 1;
         b = (b << 1) % m;
     }
     return res;
 }
Fastest NTT (can also do polynomial multiplication if max coefficients are upto 1e18 using 2 modulus and CRT)
How to use:
P=A*B
Polynomial1 = A[0]+A[1]*x^1+A[2]*x^2+..+A[n-1]*x^n-1
Polynomial2 = B[0]+B[1]*x^1+B[2]*x^2+..+B[n-1]*x^n-1
P=multiply(A,B)
A and B are not passed by reference because they are changed in multiply function
For CRT after obtaining answer modulo two primes p1 and p2:
x = a1 mod p1, x = a2 mod p2 => x=((a1*(m2^-1)%m1)*m2+(a2*(m1^-1)%m2)*m1)%m1m2
*** Before each call to multiply:
 set base=1,roots={0,1},rev={0,1},max_base=x (such that if mod=c*(2^k)+1 then x<=k and 2^x is greater than equal to nearest power of 2 of 2*n)
 root=primitive_root^((mod-1)/(2^max_base))
 For P=A*A use square function
Some useful modulo and examples
```

```
mod1=463470593 = 1768*2^18+1 primitive root = 3 => max_base=18,root=3^1768
mod2=469762049 = 1792*2^18+1 primitive root = 3 => max_base=18,root=3^1792
(mod1^-1)%mod2=313174774 (mod2^-1)%mod1=154490124
Some prime modulus and primitive root
  635437057   11
     639631361   6
     645922817   3
     648019969   17
     666894337   5
     683671553   3
     710934529   17
     715128833   3
     740294657   3
     754974721   11
     786432001   7
     799014913   13
     824180737   5
     880803841   26
     897581057   3
     899678209   7
     918552577   5
     924844033   5
     935329793   3
     943718401   7
     950009857   7
     962592769   7
     975175681   17
     985661441   3
     998244353   3
*/
//x = a1 mod m1, x = a2 mod m2, invm2m1 = (m2^-1)%m1, invm1m2 = (m1^-1)%m2, gives x%m1*m2
#define chinese(a1,m1,invm2m1,a2,m2,invm1m2) ((a1 *1ll* invm2m1 % m1 * 1ll*m2 + a2 *1ll* invm1m2 % m2 * 1ll*m1) % (m1 *1ll* m2))
int mod;//reset mod everytime with required modulus
inline int mul(int a,int b){return (a*1ll*b)%mod;}
inline int add(int a,int b){a+=b;if(a>=mod)a-=mod;return a;}
inline int sub(int a,int b){a-=b;if(a<0)a+=mod;return a;}
inline int power(int a,int b){int rt=1;while(b>0){if(b&1)rt=mul(rt,a);a=mul(a,a);b>>=1;}return rt;}
inline int inv(int a){return power(a,mod-2);}
inline void modadd(int &a,int &b){a+=b;if(a>=mod)a-=mod;}

int base = 1;
vector<int> roots = {0, 1};
vector<int> rev = {0, 1};
int max_base=18; //x such that 2^x|(mod-1) and 2^x>max answer size(=2*n)
int root=202376916;  //primitive root^((mod-1)/(2^max_base))
void ensure_base(int nbase) {
 if (nbase <= base) {
   return;
 }
 assert(nbase <= max_base);
 rev.resize(1 << nbase);
 for (int i = 0; i < (1 << nbase); i++) {
   rev[i] = (rev[i >> 1] >> 1) + ((i & 1) << (nbase - 1));
 }
 roots.resize(1 << nbase);
 while (base < nbase) {
   int z = power(root, 1 << (max_base - 1 - base));
   for (int i = 1 << (base - 1); i < (1 << base); i++) {
     roots[i << 1] = roots[i];
     roots[(i << 1) + 1] = mul(roots[i], z);
   }
   base++;
 }
}
void fft(vector<int> &a) {
 int n = (int) a.size();
 assert((n & (n - 1)) == 0);
 int zeros = __builtin_ctz(n);
 ensure_base(zeros);
 int shift = base - zeros;
 for (int i = 0; i < n; i++) {
   if (i < (rev[i] >> shift)) {
     swap(a[i], a[rev[i] >> shift]);
   }
 }
 for (int k = 1; k < n; k <<= 1) {
   for (int i = 0; i < n; i += 2 * k) {
     for (int j = 0; j < k; j++) {
       int x = a[i + j];
       int y = mul(a[i + j + k], roots[j + k]);
       a[i + j] = x + y - mod;
       if (a[i + j] < 0) a[i + j] += mod;
       a[i + j + k] = x - y + mod;
       if (a[i + j + k] >= mod) a[i + j + k] -= mod;
     }
   }
 }
}
vector<int> multiply(vector<int> a, vector<int> b, int eq = 0) {
 int need = (int) (a.size() + b.size() - 1);
 int nbase = 0;
 while ((1 << nbase) < need) nbase++;
 ensure_base(nbase);
 int sz = 1 << nbase;
 a.resize(sz);
 b.resize(sz);
 fft(a);
 if (eq) b = a; else fft(b);
 int inv_sz = inv(sz);
 for (int i = 0; i < sz; i++) {
   a[i] = mul(mul(a[i], b[i]), inv_sz);
 }
 reverse(a.begin() + 1, a.end());
 fft(a);
 a.resize(need);
 return a;
}
vector<int> square(vector<int> a) {
 return multiply(a, a, 1);
```

```
}
```

6.4 Langrange Interpolation

```
/* Input :
Degree of polynomial: k
Polynomial values at x=0,1,2,3,...,k
Output :
Polynomial value at x
Complexity: O(degree of polynomial)
Works only if the points are equally spaced
*/
ll lagrange(vll& v , int k, ll x,int mod)
{
    if(x <= k)
        return v[x];
    ll inn = 1;
    ll den = 1;
    for(int i = 1;i<=k;i++)
    {
        inn = (inn*(x - i))%mod;
        den = (den*(mod - i))%mod;
    }
    inn = (inn*inv(den % mod))%mod;
    ll ret = 0;
    for(int i = 0;i<=k;i++){
        ret = (ret + v[i]*inn)%mod;
        ll md1 = mod - ((x-i)*(k-i))%mod;
        ll md2 = ((i+1)*(x-i-1))%mod;
        if(i!=k)
            inn = (((inn*md1)%mod)*inv(md2 % mod))%↩
               mod;
    }
    return ret;
}
```

6.5 Matrix Struct

```
struct matrix{
    ld B[N][N], n;
    matrix(){n = N; memset(B,0,sizeof B);}
    matrix(int _n){
        n = _n; memset(B, 0, sizeof B);
    }
    void iden(){
     for(int i = 0; i < n; i++)
      B[i][i] = 1;
    }
    void operator += (matrix M){
        for(int i = 0; i < n; i++)
         for(int j = 0; j < n; j++)
          B[i][j] = add(B[i][j], M.B[i][j]);
    }
    void operator -= (matrix M){
        for(int i = 0; i < n; i++)
         for(int j = 0; j < n; j++)
          B[i][j] = sub(B[i][j], M.B[i][j]);
    }
    void operator *= (ld b){
        for(int i = 0; i < n; i++)
         for(int j = 0; j < n; j++)
          B[i][j] = mul(b, B[i][j]);
    }
    matrix operator - (matrix M){
        matrix ret = (*this);
        ret -= M; return ret;
    }
    matrix operator + (matrix M){
        matrix ret = (*this);
        ret += M; return ret;
    }
    matrix operator * (matrix M){
        matrix ret = matrix(n); memset(ret.B, 0, ↩
           sizeof ret.B);
        for(int i = 0; i < n; i++)
            for(int j = 0; j < n;j++)
                for(int k = 0; k < n; k++){
                    ret.B[i][j] = add(ret.B[i][j], ↩
                       mul(B[i][k], M.B[k][j]));
                }
        return ret;
    }
    matrix operator *= (matrix M){ *this = ((*this) ↩
       * M);}
    matrix operator * (int b){
        matrix ret = (*this); ret *= b; return ret;
    }
    vector<double> multiply(const vector<double> & v↩
       ) const{
     vector<double> ret(n);
     for(int i = 0; i < n; i++)
      for(int j = 0; j < n; j++){
       ret[i] += B[i][j] * v[j];
      }
     return ret;
    }
};
```

6.6 nCr(Non Prime Modulo)

```
// sandwich, jtnydv25 video
// calculates nCr, for small
// non-prime modulo, and (very) big n,r.
ll phimod;
vll pr,prn;vll fact;
ll power(ll a,ll x,ll mod){
 ll ans=1;
 while(x){
  if((1LL)&(x))ans=(ans*a)%mod;
```

```
  a=(a*a)%mod;x>>=1LL;
 }
 return ans;
}
// prime factorization of x.
// pr-> prime ; prn -> it's exponent
void getprime(ll x){
  pr.clear();prn.clear();
  ll i,j,k;
  for(i=2;(i*i)<=x;i++){
   k=0;while((x%i)==0){k++;x/=i;}
   if(k>0){pr.pb(i);prn.pb(k);}
  }
  if(x!=1){pr.pb(x);prn.pb(1);}
  return;
}
// factorials are calculated ignoring
//  multiples of p.
void primeproc(ll p,ll pe){   // p , p^e
  ll i,d;
  fact.clear();fact.pb(1);d=1;
  for(i=1;i<pe;i++){
   if(i%p){fact.pb((fact[i-1]*i)%pe);}
   else {fact.pb(fact[i-1]);}
  }
  return;
}
// again note this has ignored multiples of p
ll Bigfact(ll n,ll mod){
 ll a,b,c,d,i,j,k;
 a=n/mod;a%=phimod;a=power(fact[mod-1],a,mod);
 b=n%mod;a=(a*fact[b])%mod;
 return a;
}
// Chinese Remainder Thm.
vll crtval,crtmod;
ll crt(vll &val,vll &mod){
 ll a,b,c,d,i,j,k;b=1;
 for(ll z:mod)b*=z;
 ll ans=0;
 for(i=0;i<mod.size();i++){
  a=mod[i];c=b/a;
  d=power(c,(((a/pr[i])*(pr[i]-1))-1),a);
  c=(c*d)%b;c=(c*val[i])%b;ans=(ans+c)%b;
 }
 return ans;
}
// calculate for prime powers and
// take crt. For each prime power,
// first ignore multiples of p,
// and then do recursively, calculating
// the powers of p separately.
ll Bigncr(ll n,ll r,ll mod){
 ll a,b,c,d,i,j,k;ll p,pe;
 getprime(mod);ll Fnum=1;ll Fden;
 crtval.clear();crtmod.clear();
 for(i=0;i<pr.size();i++){
  Fnum=1;Fden=1;
  p=pr[i];pe=power(p,prn[i],1e17);
  primeproc(p,pe);
  a=1;d=0;
  phimod=(pe*(p-1LL))/p;
  ll n1=n,r1=r,nr=n-r;
  while(n1){
   Fnum=(Fnum*(Bigfact(n1,pe)))%pe;
   Fden=(Fden*(Bigfact(r1,pe)))%pe;
   Fden=(Fden*(Bigfact(nr,pe)))%pe;
   d+=n1-(r1+nr);
   n1/=p;r1/=p;nr/=p;
  }
  Fnum=(Fnum*(power(Fden,(phimod-1LL),pe)))%pe;
  if(d>=prn[i])Fnum=0;
  else Fnum=(Fnum*(power(p,d,pe)))%pe;
  crtmod.pb(pe);crtval.pb(Fnum);
 }
 // you can just iterate instead of crt
 // for(i=0;i<mod;i++){
 //  bool cg=true;
 //  for(j=0;j<crtmod.size();j++){
 //   if(i%crtmod[j]!=crtval[j])cg=false;
 //  }
 //  if(cg)return i;
 // }
 return crt(crtval,crtmod);
}
```

6.7 Primitive Root Generator

```
/*To find generator of U(p),we check for all
  g in [1,p]. But only for powers of the
  form phi(p)/p_j, where p_j is a prime factor of
  phi(p). Note that p is not prime here.

  Existence , if one of these :
  1. p = 1,2,4
  2. p = q^k , where q -> odd prime.
  3. p = 2.(q^k) , where q-> odd prime

  Note that a.g^(phi(p)) = 1 (mod p)
            b.there are phi(phi(p)) generators if ↩
               exists.
*/
// Finds "a" generator of U(p),
// multiplicative group of integers mod p.
// here calc_phi returns the toitent function for p
// Complexity : O(Ans.log(phi(p)).log(p)) + time for↩
    factorizing phi(p).
// By some theorem, Ans = O((log(p))^6). Should be ↩
   fast generally.
int generator (int p) {
    vector<int> fact;
    int phi = calc_phi(p),  n = phi;
    for (int i=2; i*i<=n; ++i)
        if (n % i == 0) {
```

```
            fact.push_back (i);
            while (n % i == 0)
                n /= i;
        }
    if (n > 1)
        fact.push_back (n);

    for (int res=2; res<=p; ++res) {
        if(gcd(res,p)!=1)continue;
        bool ok = true;
        for (size_t i=0; i<fact.size() && ok; ++i)
            ok &= powmod (res, phi / fact[i], p) != ↩
               1;
        if (ok)  return res;
    }
    return -1;
}
```

7 Strings

7.1 Hashing Theory

```
If order not imp. and count/frequency imp. use this ↩
   as hash fn:-
( (a_1+h)^k + (a_2+h)^k + (a_3+h)^k + (a_4+h)^k ) % ↩
   p. Select : h,k,p
Alternate:
((x)^(a_1)+(x)^(a_2)+...+(x)^(a_k))%mod where x and ↩
   mod are fixed and a_1...a_k is an unordered set
```

7.2 Manacher

```
// Same idea as Z_algo, Time : O(n)
// [l,r] represents : boundaries of rightmost ↩
   detected subpalindrom(with max r)
// takes string s and returns a vector of lengths of↩
    odd length palindrom
// centered around that char(e.g abac for 'b' ↩
   returns 2(not 3))
vll manacher_odd(string s){
    ll n = s.length(); vll d1(n);
    for(ll i = 0, l = 0, r = -1;i<n;i++){
        d1[i] = 1;
        if(i <= r){
            d1[i] = min(r-i+1,d1[l+r-i]); // use ↩
               prev val
        }
        while(i+d1[i] < n && i-d1[i] >= 0 && s[i+d1[↩
           i]] == s[i-d1[i]]) d1[i]++; // trivial ↩
           matching
        if(r < i+d1[i]-1) l=i-d1[i]+1, r=i+d1[i]-1; ↩
           // update r
    }
    return d1;
}
// takes string s and returns vector of lengths of ↩
   even length ...
// (it's centered around the right middle char, bb ↩
   is centered around the later 'b')
vll manacher_even(string s){
    ll n = s.length(); vll d2(n);
    for(ll i = 0, l = 0, r = -1;i<n;i++){
        d2[i] = 0;
        if(i <= r){
            d2[i] = min(r-i+1,d2[l+r+1-i]);
        }
        while(i+d2[i] < n && i-d2[i]-1 >= 0 && s[i+↩
           d2[i]] == s[i-d2[i]-1]) d2[i]++;
        if(d2[i] > 0 && r < i+d2[i]-1) l=i-d2[i], r=↩
           i+d2[i]-1;
    }
    return d2;
}
// Other mtd : To do both things in one pass, add ↩
   special char e.g string "abc" => "$a$b$c$"
```

7.3 Suffix Array

```
//code credits - https://cp-algorithms.com/string/↩
   suffix-array.html
/*Theory :-
Sorted array of suffixes = sorted array of cyclic ↩
   shifts of string+$.
We consider a prefix of len. 2^k of the cyclic, in ↩
   the kth iteration.
And find the sorted order, using values for (k-1)th ↩
   iteration and
kind of radix sort. Could be thought as some kind of↩
    binary lifting.
String of len. 2^k -> combination of 2 strings of ↩
   len. 2^(k-1), whose
order we know. Just radix sort on pair for next ↩
   iteration.
Time :- O(nlog(n) + alphabet)
Applications :-
Finding the smallest cyclic shift;Finding a ↩
   substring in a string;
Comparing two substrings of a string;Longest common ↩
   prefix of two substrings;
Number of different substrings.
*/
// return list of indices(permutation of indices ↩
   which are in sorted order)
vector<ll> sort_cyclic_shifts(string const& s) {
    ll n = s.size();
    const ll alphabet = 256;
    //********* change the alphabet size accordingly↩
        and indexing ******************
```

```
        vector<ll> p(n), c(n), cnt(max(alphabet, n), 0);
    // p -> sorted order of 1-len prefix of each cyclic shift index.
    // c -> class of a index
    // pn -> same as p for kth iteration . ||ly cn.
    for (ll i = 0; i < n; i++)
        cnt[s[i]]++;
    for (ll i = 1; i < alphabet; i++)
        cnt[i] += cnt[i-1];
    for (ll i = 0; i < n; i++)
        p[--cnt[s[i]]] = i;
    c[p[0]] = 0;
    ll classes = 1;
    for (ll i = 1; i < n; i++) {
        if (s[p[i]] != s[p[i-1]])
            classes++;
        c[p[i]] = classes - 1;
    }
        vector<ll> pn(n), cn(n);
    for (ll h = 0; (1 << h) < n; ++h) {
        for (ll i = 0; i < n; i++) {  // sorting w.r.t second part.
            pn[i] = p[i] - (1 << h);
            if (pn[i] < 0)
                pn[i] += n;
        }
        fill(cnt.begin(), cnt.begin() + classes, 0);
        for (ll i = 0; i < n; i++)
            cnt[c[pn[i]]]++;
        for (ll i = 1; i < classes; i++)
            cnt[i] += cnt[i-1];
        for (ll i = n-1; i >= 0; i--)
            p[--cnt[c[pn[i]]]] = pn[i];    // sorting w.r.t first (more significant) part.
        cn[p[0]] = 0;
        classes = 1;
        for (ll i = 1; i < n; i++) {  // determining new classes in sorted array.
            pair<ll, ll> cur = {c[p[i]], c[(p[i] + (1 << h)) % n]};
            pair<ll, ll> prev = {c[p[i-1]], c[(p[i-1] + (1 << h)) % n]};
            if (cur != prev)
                ++classes;
            cn[p[i]] = classes - 1;
        }
        c.swap(cn);
    }
    return p;
}

vector<ll> suffix_array_construction(string s) {
    s += "$";
    vector<ll> sorted_shifts = sort_cyclic_shifts(s);
    sorted_shifts.erase(sorted_shifts.begin());
    return sorted_shifts;
}
```

```
// For comparing two substring of length l starting at i,j.
// k - 2^k > l/2. check the first 2^k part, if equal,
// check last 2^k part. c[k] is the c in kth iter of S.A construction.
int compare(int i, int j, int l, int k) {
    pair<int, int> a = {c[k][i], c[k][(i+l-(1 << k))%n]};
    pair<int, int> b = {c[k][j], c[k][(j+l-(1 << k))%n]};
    return a == b ? 0 : a < b ? -1 : 1;
}
/*
Kasai's Algo for LCP construction :
Longest Common Prefix for consecutive suffixes in suffix array.
lcp[i]=length of lcp of ith and (i+1)th suffix in the susffix array.
1. Consider suffixes in decreasing order of length.
2. Let p = s[i....n]. It will be somewhere in the S.A.We determine its lcp = k.
3. Then lcp of q=s[(i+1)....n] will be atleast k-1. Why?
4. Remove the first char of p and its successor in the S.A. These are suffixes with lcp k-1.
5. But note that these 2 may not be consecutive in S.A. But however lcp of strings in
   b/w have to be also atleast k-1.
*/
vector<ll> lcp_construction(string const& s, vector<ll> const& p) {
    ll n = s.size();
    vector<ll> rank(n, 0);
    for (ll i = 0; i < n; i++)
        rank[p[i]] = i;
    ll k = 0;
    vector<ll> lcp(n-1, 0);
    for (ll i = 0; i < n; i++) {
        if (rank[i] == n - 1) {
            k = 0;
            continue;
        }
        ll j = p[rank[i] + 1];
        while (i + k < n && j + k < n && s[i+k] == s[j+k])
            k++;
        lcp[rank[i]] = k;
        if (k)
            k--;
    }
    return lcp;
}
```

7.4 Trie

```
const ll AS = 26; // alphabet size
ll go[MAX][AS];
ll cnt[MAX];ll cn=0;
// cn -> index of next new node
// convert all strings to vll
ll newNode() {
 for(ll i=0;i<AS;i++)
  go[cn][i]=-1;
 return cn++;
}
// call newNode once *******
// before adding anything **
void addTrie(vll &x) {
 ll v = 0;
 cnt[v]++;
 for(ll i=0;i<x.size();i++){
  ll y=x[i];
  if(go[v][y]==-1)
   go[v][y]=newNode();
  v=go[v][y];
  cnt[v]++;
 }
}
// returns count of substrings with prefix x
ll getcount(vll &x){
 ll v=0;
 for(i=0;i<x.size();i++){
  ll y=x[i];
  if(go[v][y]==-1)
   go[v][y]=newNode();
  v=go[v][y];
 }
 return cnt[v];
}
```

7.5 Z-algorithm

```
// [l,r] -> indices of the rightmost segment match
// (the detected segment that ends rightmost(with ↩
   max r))
// 2 cases -> 1st. i <= r : z[i] is atleast min(r-i↩
   +1,z[i-l]), then match trivially
// 2nd. o.w compute z[i] with trivial matching
// update l,r
// Time : O(n)(asy. behavior), Proof : each ↩
   iteration of inner while loop make r pointer ↩
   advance to right,
// Applications:    1) Search substring(text t,↩
   pattern p) s = p + '$' + t.
// 3) String compression(s = t+t+...+t, then find |t↩
   |)
// 2) Number of distinct substrings (in O(n^2))
// (useful when appending or deleting characters ↩
   online from the end or beginning)
```

```
vector<ll> z_function(string s) {
    ll n = (ll) s.length();
    vector<ll> z(n);
    for (ll i = 1, l = 0, r = 0; i < n; ++i) {
        if (i <= r)
            z[i] = min (r - i + 1, z[i - l]); // use↩
                previous z val
        while (i + z[i] < n && s[z[i]] == s[i + z[i↩
           ]]) // trivial matching
            ++z[i];
        if (i + z[i] - 1 > r)
            l = i, r = i + z[i] - 1; // update ↩
               rightmost segment matched
    }
    return z;
}
```