ICPC ZetaSquad Notebook (2019-20)

<< Ahmed Zaheer Dadarkar, Devansh Singh Rathore, Rakesh Kumar >>

Indian Institute of Technology, Palakkad

CONTENTS ::

1. GRAPH :

a. All Topological Orderings (Kahn algorithm similarity): O((n + m) ^ n)

```
vector<vector<int> > a; vector<int> topo;
int n, indeg[N];
void all_t_orders() {
        bool foundIndegZero = false;
        for(int u = 0;u < n;u++) {
                if(indeg[u] == 0) {
                        foundIndegZero = true;
                        indeg[u] --;
                        for(int v : a[u]) indeg[v] --;
                        topo.push_back(u);

                        all_t_orders();

                        indeg[u] ++;
                        for(int v : a[u]) indeg[v] ++;
                        topo.pop_back();
                }
        }
        if(!foundIndegZero){  printVec(topo); // We have a topo ordering }
}
```

b. Bipartite Check: O(n + m)

```
vector<vector<int> > a;
int color[N];
bool isBipartite;

void dfs(int u) {
        for(int v : a[u]) {
                if(color[v] == -1) {
                        color[v] = 1 - color[u];
                        dfs(v);
                }
                else if(color[v] == color[u]) isBipartite = false;
        }
}
void bip_check()
{
```

```
        isBipartite = true;
        for(int u = 0;u < n;u++) {
            if(color[u] == -1) { color[u] = 0; dfs(u); }
        }
}
```

c. Cycle check / Backedge detection in an undirected gph: O(n + m)

```
vector<vector<int> > a;
bool vis[N];
bool cycleFound;

void dfs(int u, int parent) {
        vis[u] = true;
        for(int v  : a[u]) {
            if(!vis[v]) dfs(v, u);    // tree edge
            else if(v != parent) cycleFound = true;  // (u, v) is a backedge
        }
}
void cycle_det() {
        cycleFound = false;
        for(int u = 0;u < n;u++) {
            if(!vis[u]) dfs(u, -1);
        }
}
```

d. Cycle check / Backedge detection in a directed gph: O(n + m)

```
vector<vector<int> > a;
int state[N];
bool cycleFound;

void dfs(int u) {
        state[u] = 1;
        for(int v : a[u]) {
            if(state[v] == 0) dfs(v);           // Tree Edge
            else if(state[v] == 1) cycleFound = true; // Back Edge
            // state value of 2 means Forward or Cross edge
        }
        state[u] = 2;
}
```

```
void cycle_det() {
    cycleFound = false;
    for(int u = 0;u < n;u++) if(state[u] == 0) dfs(u);
}
```

e. Articulation points and Bridges, Undirected: O(n + m)

```
vector<vector<int> > a; vector<pair<int, int> > br;
int dfs_low[N], dfs_num[N], numChildren, ct;
bool art_p[N]t;

void dfs(int u, int parent) {
    dfs_num[u] = dfs_low[u] = ct ++;
    for(int v : a[u]) {
        if(dfs_num[u] == -1) {
            if(parent == -1) numChildren ++;
            dfs(u);
            dfs_low[u] = min(dfs_low[u], dfs_low[v]);
            if(parent != -1 && dfs_low[v] >= dfs_num[u]) art_pt[u] = true;
            if(dfs_low[v] > dfs_num[u]) br.push_back({u, v});
        }
        else if(v != parent) dfs_low[u] = min(dfs_low[u], dfs_num[v]);
    }
}
void findArtBr() {
    ct = 0;
    for(int u = 0;u < n;u++) {
        if(dfs_num[u] == -1) {
            numChildren = 0;
            dfs(u, -1);
            if(numChildren > 1) art_pt[u] = true;
    }
}
```

f. Finding SCC - Tarjan: O(n + m)

```
vector<vector<int> > a;
int dfs_low[N], dfs_num[N], state[N], scc[N], scc_no;
stack<int> s;

void dfs(int u) {
```

```
        dfs_num[u] = dfs_low[u] = ct ++;
        state[u] = 1;
        s.push(s);
        for(int v : a[u]) {
                if(state[v] == 0) dfs(u);
                if(state[v] == 1) dfs_low[u] = min(dfs_low[u], dfs_low[v]);
        }
        if(dfs_low[u] == dfs_num[u]) {
                int v = -1;
                while(v != u) {
                        v = s.top();
                        s.pop();
                        state[v] = 2;
                        scc[v] = scc_no;
                }
                scc_no ++;
        }
}
void findSCC() {
        for(int u = 0;u < n;u++) if(state[u] == 0) dfs(u);
}
```

g. Prim's Algorithm MST: O(m*log(n))

```
priority_queue<pair<int, int>, vector<pair<int, int> > , greater<pair<int, int> > > pq;
vector<vector<pair<int, int> > > a;  // first: weight, second: vertex
bool taken[N];

void printMST() {
        int mst_wt = 0;
        pq.push({0, 0});
        while(!pq.empty()) {
                pair<int, int> u = pq.top();
                pq.pop();
                if(taken[u.second]) continue;
                taken[u.second] = true;
                mst_cost += u.first;
                for(pair v : a[u.second]) if(!taken[v.second]) pq.push(v);
        }
}
```

h. Floyd Warshall with path information: O(n ^ 3)

```
int n, adj[N][N], dist[N][N], p[N][N];
// adj is initialized with INF for edges which don't exist
void FW() {
        int k, i, j;
        for(i = 0;i < n;i++) {
                for(j = 0;j < n;j++) {
                        dist[i][j] = adj[i][j];
                        if(adj[i][j] != INF) p[i][j] = i;
        }}
        for(k = 0;k < n;k++) {
                for(i = 0;i < n;i++) {
                        for(j = 0;j < n;j++) {
                                if(dist[i][k] + dist[k][j] < dist[i][j])
                                {
                                        dist[i][j] = dist[i][k] + dist[k][j];
                                        p[i][j] = p[k][j];
}}}}}
void printPath(int s, int v) {
        if(v == s) {cout<<v<<" ";return; }
        printPath(s, p[s][v]); cout<<v<<" ";
}
```

i. Finding Diameter in a Tree: O(n)

```
vector<vector<pair<int, int> > > a;
int n, dist[N];
void dfs(int u) {
        for(pair<int, int> v : a[u]) {
                if(dist[v.second] == -1) { dist[v.second] = dist[u] + 1; dfs(v.second); }
}}
void findDiameterTree(){
        resetDist(); dfs(0);
        int s = 0, t = 0;
        for(int i = 1;i < n;i++) if(dist[i] > dist[s]) s = i;
        resetDist(); dfs(s);
        for(int i = 1;i < n;i++) if(dist[i] > dist[t]) t = i;
        int dia = dist[t];
```

```
}
```

j. Edmonds Karp: O(n*(m^2) + n^2)

```
vector<vector<int> > a;
queue<int> q;
int s, t, f, n, res[N][N];
bool vis[N];

void augment(int v, int minEdge) {
        if(v == s) { f = minEdge; return; }
        augment(p[v], min(minEdge, res[p[v]][v]);
        res[p[v]][v] -= f;
        res[v][p[v]] += f;
}
void EdmondsKarp() {
        int mf  = 0;
        while(true) {
                f = 0;
                resetBFS();
                vis[s] = true; q.push(s);
                while(!q.empty()) {
                        int u = q.front();
                        q.pop();
                        if(u == t) { augment(t, INF); break; }
                        for(int v : a[u]) {
                                if(res[u][v] > 0 && !vis[v]) {
                                        vis[v] = true;
                                        q.push(v);
                }}}
                if(f == 0) break;
                mf += f;
        }
        cout<<mf<<"\n";
}
```

k. Euler's Path: O(m)

```
list cyc; // we need list for fast insertion in the middle
void EulerTour(list::iterator i, int u) {
        for (int j = 0; j < (int)AdjList[u].size(); j++) {
```

```
            ii v = AdjList[u][j];
            if (v.second) { // if this edge can still be used/not removed
                v.second = 0; // make the weight of this edge to be 0 ('removed')
                for (int k = 0; k < (int)AdjList[v.first].size(); k++) {
                    ii uu = AdjList[v.first][k];  // remove bi-directional edge
                    if (uu.first == u && uu.second) { uu.second = 0; break; }
                }
                EulerTour(cyc.insert(i, u), v.first);
}}}
//inside int main()
    cyc.clear();
    EulerTour(cyc.begin(), A);  // cyc contains an Euler tour starting at A
    for (list::iterator it = cyc.begin(); it != cyc.end(); it++)
        printf("%d\n", *it); //Euler tour
```

l. MCBM: O(n*m)

```
//but Hopcroft Karp's algorithm that can solve the MCBM problem in O( √ V E)
vi match, vis;          // global variables

int Aug(int l) {        // return 1 if an augmenting path is found
    if (vis[l]) return 0;    // return 0 otherwise
    vis[l] = 1;
    for (int j = 0; j < (int)AdjList[l].size(); j++) {
        int r = AdjList[l][j];    // edge weight not needed -> vector AdjList
        if (match[r] == -1 || Aug(match[r])) {
            match[r] = l; return 1;        // found 1 matching
    }}
    return 0;       // no matching
}

// inside int main()
    // build unweighted bipartite graph with directed edge left->right set
    int MCBM = 0;
    match.assign(V, -1);   // V is the number of vertices in bipartite graph
    for (int l = 0; l < n; l++) {      // n = size of the left set
        vis.assign(n, 0);        // reset before each recursion
        MCBM += Aug(l);
    }
    printf("Found %d matchings\n", MCBM);
```

m. TSP: O((n^2)*(2^n))

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

struct tsp {
  using ll = long long;
  static const int N = 17;
  const ll inf = (ll) 1e18;
  ll dp[N][1 << N];
  int par[N][1 << N], start[N][1 << N], wgt[N][N];
  int n;

  ll solve() {
    //    for (int k = 0; k < n; ++k)    //Floyd-Warshall for shortest dist wgts (undirected)
    //       for (int i = 0; i < n; ++i) for (int j = 0; j < n; ++j)
    //          wgt[j][i] = wgt[i][j] = min(wgt[i][j], wgt[i][k] + wgt[k][j]);

    for (int mask = 0; mask < (1 << n); ++mask)
      for (int i = 0; i < n; ++i)
        dp[i][mask] = inf;

    for (int i = 0; i < n; ++i)
      dp[i][1 << i] = 0, par[i][1 << i] = -1, start[i][1 << i] = i;

    for (int mask = 1; mask < (1 << n); ++mask) {
      for (int pos = 0; pos < n; ++pos)
      {
        if (!(mask & (1 << pos))) continue;
        ll mn_val = inf;
        for (int prv = 0; prv < n; ++prv) {
          if (pos == prv || !(mask & (1 << prv)) || !wgt[prv][pos]) continue;
          ll guess = dp[prv][mask-(1<<pos)] + wgt[prv][pos];
          if (mn_val > guess) {
            mn_val = guess;
            par[pos][mask] = prv;
            start[pos][mask] = start[prv][mask-(1 << pos)];
          }}
        dp[pos][mask] = min(dp[pos][mask], mn_val);
```

```
    }}

  ll mn_cycle = inf;
  int pos = -1, mask = (1 << n) - 1;
  for (int i = 0; i < n; ++i) {
    ll go = dp[i][mask] + wgt[i][start[i][mask]];
    if (mn_cycle > go) {
      mn_cycle = go;
      pos = i;
    }}

  vector <int> cycle;
  while (pos != -1) {
    cycle.push_back(pos);
    int tmp = pos;
    pos = par[pos][mask];
    mask -= (1 << tmp);
  }
  for (int v : cycle) { cerr << v << ' '; } cerr << '\n';
  return mn_cycle;
}};
```

n. 2-SAT

```
int n;
vector<vector<int>> g, gt;
vector<bool> used;
vector<int> order, comp;
vector<bool> assignment;
void dfs1(int v) {
   used[v] = true;
   for (int u : g[v]) {
      if (!used[u])
         dfs1(u);
   }
   order.push_back(v);
}
void dfs2(int v, int cl) {
   comp[v] = cl;
   for (int u : gt[v]) {
```

```
        if (comp[u] == -1)
            dfs2(u, cl);
    }
}
bool solve_2SAT() {
    used.assign(n, false);
    for (int i = 0; i < n; ++i) {
        if (!used[i])
            dfs1(i);
    }
    comp.assign(n, -1);
    for (int i = 0, j = 0; i < n; ++i) {
        int v = order[n - i - 1];
        if (comp[v] == -1)
            dfs2(v, j++);
    }
    assignment.assign(n / 2, false);
    for (int i = 0; i < n; i += 2) {
        if (comp[i] == comp[i + 1])
            return false;
        assignment[i / 2] = comp[i] > comp[i + 1];
    }
    return true; }
```

o. Heavy Light Decomposition

```
vector<int> parent, depth, heavy, head, pos;
int cur_pos;
int dfs(int v, vector<vector<int>> const& adj) {
    int size = 1;
    int max_c_size = 0;
    for (int c : adj[v]) {
        if (c != parent[v]) {
            parent[c] = v, depth[c] = depth[v] + 1;
            int c_size = dfs(c, adj);
            size += c_size;
            if (c_size > max_c_size)
                max_c_size = c_size, heavy[v] = c;
        }
    }
```

```
    return size;
}
int decompose(int v, int h, vector<vector<int>> const& adj) {
    head[v] = h, pos[v] = cur_pos++;
    if (heavy[v] != -1)
        decompose(heavy[v], h, adj);
    for (int c : adj[v]) {
        if (c != parent[v] && c != heavy[v])
            decompose(c, c, adj);
    }
}
void init(vector<vector<int>> const& adj) {
    int n = adj.size();
    parent = vector<int>(n);
    depth = vector<int>(n);
    heavy = vector<int>(n, -1);
    head = vector<int>(n);
    pos = vector<int>(n);
    cur_pos = 0;

    dfs(0, adj);
    decompose(0, 0, adj);
}
int query(int a, int b) {
    int res = 0;
    for (; head[a] != head[b]; b = parent[head[b]]) {
        if (depth[head[a]] > depth[head[b]])
            swap(a, b);
        int cur_heavy_path_max = segment_tree_query(pos[head[b]], pos[b]);
        res = max(res, cur_heavy_path_max);
    }
    if (depth[a] > depth[b])
        swap(a, b);
    int last_heavy_path_max = segment_tree_query(pos[a], pos[b]);
    res = max(res, last_heavy_path_max);
    return res;
}
```

p. Binary Lifting

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

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

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

    tout[v] = ++timer;
}

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

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

2. STRINGS:

a. Knuth-Morris-Pratt's (KMP) Algorithm: O(n + m)

```
void kmpPreprocess_Search(string T, string P, int n, int m)
```

```
{
        //Preprocess ==================>
         // b = back table, n = length of T, m = length of P
        int i=0, j=-1, b[n+100];
        b[0]=-1;
        while(i<m) {
                while(j>=0 && P[i]!=P[j]) j=b[j];
                i++;j++;
                b[i]=j;
        }

        //Search ================>
        i=0;j=0;
        while(i<n) {
                while(j>=0 && T[i]!=P[j]) j=b[j];
                i++;j++;
                if(j==m) {
                        cout<<"P found at index "<<i-j<<" in T\n";
                        j=b[j];
                }
        }
}
void kmp(string T, string P) {  //T is main string in which we search for string P
        int n=T.length(), m=P.length();
        kmpPreprocess_Search(T,P,n,m);
}
```

b. Z - function: O(n)

```
vector<int> z_function(string s) {
   int n = (int) s.length();
   vector<int> z(n);
   for (int i = 1, l = 0, r = 0; i < n; ++i) {
      if (i <= r)
         z[i] = min (r - i + 1, z[i - l]);
      while (i + z[i] < n && s[z[i]] == s[i + z[i]])
         ++z[i];
      if (i + z[i] - 1 > r)
         l = i, r = i + z[i] - 1;
   }
```

```
return z; }
```

c. Trie: O(n*log(n))

```
#define MAX_N 100010       // second approach: O(n log n)
char T[MAX_N];       // the input string, up to 100K characters
int n;   // the length of input string
int RA[MAX_N], tempRA[MAX_N];         // rank array and temporary rank array
int SA[MAX_N], tempSA[MAX_N];         // suffix array and temporary suffix array
int c[MAX_N];         // for counting/radix sort

void countingSort(int k) {      // O(n)
        int i, sum, maxi = max(300, n);         // up to 255 ASCII chars or length of n
        memset(c, 0, sizeof c);          // clear frequency table
        for (i = 0; i < n; i++)          // count the frequency of each integer rank
                c[i + k < n ? RA[i + k] : 0]++;
        for (i = sum = 0; i < maxi; i++) { int t = c[i]; c[i] = sum; sum += t; }
        for (i = 0; i < n; i++)          // shuffle the suffix array if necessary
                tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] = SA[i];
        for (i = 0; i < n; i++)          // update the suffix array SA
                SA[i] = tempSA[i];
}
void constructSA() {            // this version can go up to 100000 characters
        int i, k, r;
        for (i = 0; i < n; i++) RA[i] = T[i];     // initial rankings
        for (i = 0; i < n; i++) SA[i] = i;        // initial SA: {0, 1, 2, ..., n-1}
        for (k = 1; k < n; k <<= 1) {   // repeat sorting process log n times
                countingSort(k);        // actually radix sort: sort based on the second item
                countingSort(0);        // then (stable) sort based on the first item
                tempRA[SA[0]] = r = 0;        // re-ranking; start from rank r = 0
                for (i = 1; i < n; i++)             // compare adjacent suffixes
                        tempRA[SA[i]] = (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k]
                == RA[SA[i-1]+k]) ? r : ++r;
                        // if same pair => same rank r; otherwise, increase r
                for (i = 0; i < n; i++) RA[i] = tempRA[i];      // update the rank array RA
                if (RA[SA[n-1]] == n-1) break;        // nice optimization trick
} }

int main() {
        n = (int)strlen(gets(T));         // input T as per normal, without the '$'
```

```
        T[n++] = '$';          // add terminating character
        constructSA();
        for (int i = 0; i < n; i++) printf("%2d\t%s\n", SA[i], T + SA[i]);
} // return 0;
```

Applications of Suffix Array ⇒

String Matching: O(m*log(n)) →

```
vector<int> sort_cyclic_shifts(string const& s) {
    int n = s.size();
    const int alphabet = 256;
    vector<int> p(n), c(n), cnt(max(alphabet, n), 0);
    for (int i = 0; i < n; i++)
        cnt[s[i]]++;
    for (int i = 1; i < alphabet; i++)
        cnt[i] += cnt[i-1];
    for (int i = 0; i < n; i++)
        p[--cnt[s[i]]] = i;
    c[p[0]] = 0;
    int classes = 1;
    for (int i = 1; i < n; i++) {
        if (s[p[i]] != s[p[i-1]])
            classes++;
        c[p[i]] = classes - 1;
    }

    vector<int> pn(n), cn(n);
    for (int h = 0; (1 << h) < n; ++h) {
        for (int i = 0; i < n; i++) {
            pn[i] = p[i] - (1 << h);
            if (pn[i] < 0)
                pn[i] += n;
        }
        fill(cnt.begin(), cnt.begin() + classes, 0);
        for (int i = 0; i < n; i++)
            cnt[c[pn[i]]]++;
        for (int i = 1; i < classes; i++)
            cnt[i] += cnt[i-1];
        for (int i = n-1; i >= 0; i--)
```

```
            p[--cnt[c[pn[i]]]] = pn[i];
        cn[p[0]] = 0;
        classes = 1;
        for (int i = 1; i < n; i++) {
            pair<int, int> cur = {c[p[i]], c[(p[i] + (1 << h)) % n]};
            pair<int, int> 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;
}
```

Finding the Longest Common Prefix: O(n) →

```
vector<int> lcp_construction(string const& s, vector<int> const& p) {
    int n = s.size();
    vector<int> rank(n, 0);
    for (int i = 0; i < n; i++)
        rank[p[i]] = i;

    int k = 0;
    vector<int> lcp(n-1, 0);
    for (int i = 0; i < n; i++) {
        if (rank[i] == n - 1) {
            k = 0;
            continue;
        }
        int 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;
}
```

Finding the Longest Repeated Substring: O(n) →(UVa)

If we have computed the Suffix Array in O(n log n) and the LCP between consecutive suffixes in Suffix Array order in O(n), then we can determine the length of the Longest Repeated Substring (LRS) of T in O(n).

The length of the longest repeated substring is just the highest number in the LCP array. In Table 6.5—left that corresponds to the Suffix Array and the LCP of T = ‘GATAGACA$’, the highest number is 2 at index i=7. The first 2 characters of the corresponding suffix SA[7] (suffix 0) is ‘GA’. This is the longest repeated substring in T.

Finding the Longest Common Substring: O(n) →(UVa)

Without loss of generality, let’s consider the case with only two strings. We use the same example as in the Suffix Tree section earlier: T1 = ‘GATAGACA$’ and T2 = ‘CATA#’. To solve the LCS problem using Suffix Array, first we have to concatenate both strings (note that the terminating characters of both strings must be different) to produce T = ‘GATAGACA$CATA#’. Then, we compute the Suffix and LCP array of T as shown in Figure 6.6.

Then, we go through consecutive suffixes in O(n). If two consecutive suffixes belong to different owner (can be easily checked17, for example we can test if suffix SA[i] belongs to T1 by testing if SA[i] < the length of T1), we look at the LCP array and see if the maximum LCP found so far can be increased. After one O(n) pass, we will be able to determine the Longest Common Substring. In Figure 6.6, this happens when i=7, as suffix SA[7] = suffix 1 = ‘ATAGACA$CATA#’ (owned by T1) and its previous suffix SA[6] = suffix 10 = ‘ATA#’ (owned by T2) have a common prefix of length 3 which is ‘ATA’. This is the LCS.

3. MATHEMATICS:

a. Power: O(log(n))

```
int power(int x, unsigned int y) {       //to calculate power(x, n)
   int temp;
   if( y == 0) return 1;

   temp = power(x, y/2);
   if (y%2 == 0) return temp*temp;
   else return x*temp*temp;
}
```

b. Recurrence relations

Catalan Number →Cat(m) = (2m ∗ (2m − 1)/(m ∗ (m + 1))) ∗ Cat(m-1)

Combination →C(n+1,k)=C(n,k)+C(n,k−1)

c. Linear Diophantine Equation: O(log(n))

```
int gcd(int a, int b, int &x, int &y) {
        if(a == 0) {
                x = 0; y = 1; return b;
        }
        int x1, y1;
        int g = gcd(a % b, b, x1, y1);
        x = y1 - (b / a) * x1; y = x1;
        return g;
}
void find_a_Solution(int a, int b, int c) {
        int x0, y0;
        int g = gcd(abs(a), abs(b), x0, y0);
        if(c % g) return;        // no solution

        x0 *= c / g;
        y0 *= c / g;
        if(a < 0) x0 = -x0;
        if(b < 0) y0 = -y0;
}
```

We can find all solutions this way: -

x = x0 + k * (b / g)

y = y0 - k * (a / g)

d. Chinese Remainder Theorem

CODE→

```
for (int i = 0; i < k; ++i) {
   x[i] = a[i];
   for (int j = 0; j < i; ++j) {
      x[i] = r[j][i] * (x[i] - x[j]);
      x[i] = x[i] % p[i];
      if (x[i] < 0) x[i] += p[i];
}}
```

4. POLYHASH:

```
int count_unique_substrings(string const& s) {
    int n = s.size();
    const int p = 31;
    const int m = 1e9 + 9;
    vector<long long> p_pow(n);
    p_pow[0] = 1;
    for (int i = 1; i < n; i++)
        p_pow[i] = (p_pow[i-1] * p) % m;

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

5. DATA STRUCTURES AND LIBRARIES:

a. Segment Tree

```
struct SegTree {
    static const int MXN = int(2e6) + 10;
    int A[MXN];
    long long st[4 * MXN], lz[4 * MXN];
    inline int mid (int s, int e) { return s + ((e - s) >> 1); }
    void create (int s, int e, int x) {
        lz[x] = 0;
        if (s == e) {
            st[x] = A[s];
```

```
            return;
        }
        create (s, mid (s, e), x + x + 1);
        create (mid (s, e) + 1, e, x + x + 2);
        st[x] = st[x + x + 1] + st[x + x + 2];
    }
    void lzupd (int s, int e, int x) {
        if (lz[x] != 0) {
            st[x] += lz[x] * (e - s + 1);
            if (s != e) {
                lz[x + x + 1] += lz[x];
                lz[x + x + 2] += lz[x];
            }
            lz[x] = 0;
        }}
    void upd (int s, int e, int qs, int qe, int x, int k) {
        lzupd (s, e, x);
        if (e < qs || s > qe) return;
        if (s >= qs && e <= qe) {
            st[x] += k * (e - s + 1);
            if (s != e) {
                lz[x + x + 1] += k;
                lz[x + x + 2] += k;
            }
            return;
        }
        upd (s, mid (s, e), qs, qe, x + x + 1, k);
        upd (mid (s, e) + 1, e, qs, qe, x + x + 2, k);
        st[x] = st[x + x + 1] + st[x + x + 2];
    }
    long long gets (int s, int e, int qs, int qe, int x) {
        lzupd (s, e, x);
        if (e < qs || s > qe) return 0;
        if (s >= qs && e <= qe) return st[x];
        long long lgets = gets (s, mid (s, e), qs, qe, x + x + 1);
        long long rgets = gets (mid (s, e) + 1, e, qs, qe, x + x + 2);
        return lgets + rgets;
    }
};
```

b. Fenwick tree: O(n)

```
class FenwickTree {
        private: vi ft;    // recall that vi is: typedef vector vi;
        public: FenwickTree(int n) { ft.assign(n + 1, 0); }     // init n + 1 zeroes
        int rsq(int b) { // returns RSQ(1, b)
                int sum = 0;
                for (; b; b -= LSOne(b)) sum += ft[b];
                return sum;
        }           // note: LSOne(S) (S & (-S))
        int rsq(int a, int b) {    // returns RSQ(a, b)
                return rsq(b) - (a == 1 ? 0 : rsq(a - 1));
        }          // adjusts value of the k-th element by v (v can be +ve/inc or -ve/dec)
        void adjust(int k, int v) {        // note: n = ft.size() - 1
                for (; k < (int)ft.size(); k += LSOne(k)) ft[k] += v; }
};
int main() {
        int f[] = { 2,4,5,5,6,6,6,7,7,8,9 };        // m = 11 scores
        FenwickTree ft(10);    // declare a Fenwick Tree for range [1..10]
        // insert these scores manually one by one into an empty Fenwick Tree
        for (int i = 0; i < 11; i++) ft.adjust(f[i], 1);     // this is O(k log n)
        printf("%d\n", ft.rsq(1, 1));    // 0 => ft[1] = 0
        printf("%d\n", ft.rsq(1, 2));    // 1 => ft[2] = 1
        printf("%d\n", ft.rsq(1, 6));    // 7 => ft[6] + ft[4] = 5 + 2 = 7
        printf("%d\n", ft.rsq(1, 10));  // 11 => ft[10] + ft[8] = 1 + 10 = 11
        printf("%d\n", ft.rsq(3, 6));    // 6 => rsq(1, 6) - rsq(1, 2) = 7 - 1
        ft.adjust(5, 2);                 // update demo
        printf("%d\n", ft.rsq(1, 10));  // now 13
} // return 0;
```

c. Sparse Table

```
int log[MAXN+1];
log[1] = 0;
for (int i = 2; i <= MAXN; i++) log[i] = log[i/2] + 1;

long long st[MAXN][K];
for (int i = 0; i < N; i++) st[i][0] = array[i];

for (int j = 1; j <= K; j++)
```

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

6. DP

a. Divide and Conquer

```
vector<long long> dp_before(n), dp_cur(n);
// compute dp_cur[l], ... dp_cur[r] (inclusive)
void compute(int l, int r, int optl, int optr)
{
    if (l > r) return;
    int mid = (l + r) >> 1;
    pair<long long, int> best = {INF, -1};
    for (int k = optl; k <= min(mid, optr); k++) {
        best = min(best, {dp_before[k] + C(k, mid), k});
    }
    dp_cur[mid] = best.first;
    int opt = best.second;
    compute(l, mid - 1, optl, opt);
    compute(mid + 1, r, opt, optr);
}
```

b. Paraquet

```
vector < vector<long long> > d;
void calc (int x = 0, int y = 0, int mask = 0, int next_mask = 0 {
    if (x == n) return;
    if (y >= m) d[x+1][next_mask] += d[x][mask];
    else
    {
        int my_mask = 1 << y;
        if (mask & my_mask)
            calc (x, y+1, mask, next_mask);
        else
        {
            calc (x, y+1, mask, next_mask | my_mask);
            if (y+1 < m && ! (mask & my_mask) && ! (mask & (my_mask << 1)))
                calc (x, y+2, mask, next_mask);
        }
   }}
int main()
```

```
{
   cin >> n >> m;
   d.resize (n+1, vector<long long> (1<<m));
   d[0][0] = 1;
   for (int x=0; x<n; ++x) for (int mask=0; mask<(1<<m); ++mask) calc (x, 0, mask, 0);
   cout << d[n][0];
}
```

c. Tree DP

```
list <int> *adj;
vector <long long> *suffix, *prefix;
long long *dp, *ans;
int n, m;
void dfs1(int u, int p) {
        prefix[u].push_back(1);
        int cnt = 0;
        long long prod = 1;
        for (int nxt : adj[u]) {
                if (nxt != p) {
                        dfs1(nxt, u);
                        prod = prod * (dp[nxt] + 1) % m;
                        prefix[u].push_back(prefix[u][cnt] * (dp[nxt] + 1) % m);
                }
                else prefix[u].push_back(prefix[u][cnt]);
                cnt++;
        }
        cnt = 0;
        suffix[u].push_back(1);
        for (auto nxt = adj[u].rbegin(); nxt != adj[u].rend(); ++nxt) {
                if (*nxt != p) suffix[u].push_back(suffix[u][cnt] * (dp[*nxt] + 1) % m);
                else suffix[u].push_back(suffix[u][cnt]);
                cnt++;
        }
        dp[u] = prod;
}
void dfs2(int u, int p, long long val) {
        int cnt = 0;
        long long temp;
        for (int nxt : adj[u]) {
```

```
            if (nxt != p) {
                int len = suffix[u].size();
                temp = suffix[u][len - cnt - 2] * prefix[u][cnt] % m;
                temp = temp * val % m;
                ans[nxt] = dp[nxt] * (temp + 1) % m;
                //cout<<temp<<" "<<u<<" "<<nxt<<endl;
                dfs2(nxt, u, (temp + 1) % m);
            }
            cnt++;
}}
```

d. Longest Increasing Subsequence: O(n*logn)

```
int lis(vector<int> const& a) {
    int n = a.size();
    const int INF = 1e9;
    vector<int> d(n+1, INF);
    d[0] = -INF;
    for (int i = 0; i < n; i++) {
        int j = upper_bound(d.begin(), d.end(), a[i]) - d.begin();
        if (d[j-1] < a[i] && a[i] < d[j]) d[j] = a[i];
    }
    int ans = 0;
    for (int i = 0; i <= n; i++) if (d[i] < INF) ans = i;
    return ans;
}
```
