## 1. Template

### Main

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

#define MAX(a, b) (a > b) ? a : b
#define MIN(a, b) (a < b) ? a : b
#define int long long
#define vi vector<int>
#define pii pair<int, int>
#define vii vector<pii>

using namespace std;

void solve()
{
}

int32_t main()
{
   ios_base::sync_with_stdio(0);
   cin.tie(0);

   int t;
   cin >> t;

   for (int i = 0; i < t; i++)
   {
      solve();
   }

   return 0;
}
```

## 2. Graph

### Prim

```cpp
int spanningTreePrim(int V, vector<vector<int>> adj[])
{

   priority_queue<pair<int, int>> q;

   vector<bool> mask;
   mask.assign(V, false);
   mask[0] = true;

   int cost = 0;

   for (int i = 0; i < adj[0].size(); i++)
   {
      q.push({-adj[0][i][1], adj[0][i][0]});
   }

   while (q.size() != 0)
   {
      auto aux = q.top();
      q.pop();

      int k = aux.second;
      if (mask[k])
         continue;

      mask[k] = true;
      cost += abs(aux.first);

      for (int i = 0; i < adj[k].size(); i++)
      {
         if (!mask[adj[k][i][0]])
         {
            q.push({-adj[k][i][1], adj[k][i][0]});
         }
      }
   }

   return cost;
}
```

## Dfs Bfs

```
void dfs_g(int n, int c, vi adj[], vector<bool> &visited, vi &cc)
{
   visited[n] = true;
   cc[n] = c;

   for (int i = 0; i < adj[n].size(); i++)
   {
      if (!visited[adj[n][i]])
         dfs_g(adj[n][i], c, adj, visited, cc);
   }
}

void dfs_t(int n, int p, int d, vi adj[], vi &deep)
{
   deep[n] = d;

   for (int i = 0; i < adj[n].size(); i++)
   {
      if (p != adj[n][i])
         dfs_t(adj[n][i], n, d + 1, adj, deep);
   }
}

vi bfs(int node, int n, vi adj[])
{
   vi result(n);
   vector<bool> visited;
   visited.assign(n, false);

   queue<int> q;
   visited[node] = true;

   q.push(node);

   while (q.size() != 0)
   {
      int w = q.front();
      q.pop();

      for (int i = 0; i < adj[w].size(); i++)
      {
         if (!visited[adj[w][i]])
         {
            q.push(adj[w][i]);
            result[adj[w][i]] = result[w] + 1;
            visited[adj[w][i]] = true;
         }
      }
   }

   return result;
}
```

## Max Flow

```
template <typename flow_type>
struct dinic
{
   struct edge
   {
      size_t src, dst, rev;
      flow_type flow, cap;
   };

   int n;
   vector<vector<edge>> adj;

   dinic(int n) : n(n), adj(n), level(n), q(n), it(n) {}

   void add_edge(size_t src, size_t dst, flow_type cap, flow_type rcap = 0)
   {
      adj[src].push_back({src, dst, adj[dst].size(), 0, cap});
      if (src == dst)
         adj[src].back().rev++;
      adj[dst].push_back({dst, src, adj[src].size() - 1, 0, rcap});
   }

   vector<int> level, q, it;

   bool bfs(int source, int sink)
   {
      fill(level.begin(), level.end(), -1);
      for (int qf = level[q[0] = sink] = 0, qb = 1; qf < qb; ++qf)
      {
         sink = q[qf];
         for (edge &e : adj[sink])
         {
            edge &r = adj[e.dst][e.rev];
            if (r.flow < r.cap && level[e.dst] == -1)
               level[q[qb++] = e.dst] = 1 + level[sink];
         }
      }
      return level[source] != -1;
   }

   flow_type augment(int source, int sink, flow_type flow)
   {
      if (source == sink)
         return flow;
      for (; it[source] != adj[source].size(); ++it[source])
      {
         edge &e = adj[source][it[source]];
         if (e.flow < e.cap && level[e.dst] + 1 == level[source])
         {
            flow_type delta = augment(e.dst, sink, min(flow, e.cap - e.flow));
            if (delta > 0)
            {
               e.flow += delta;
               adj[e.dst][e.rev].flow -= delta;
               return delta;
            }
         }
      }
      return 0;
   }

   flow_type max_flow(int source, int sink)
   {
      for (int u = 0; u < n; ++u)
         for (edge &e : adj[u])
            e.flow = 0;
      flow_type flow = 0;
      flow_type oo = numeric_limits<flow_type>::max();

      while (bfs(source, sink))
      {
         fill(it.begin(), it.end(), 0);
         for (flow_type f; (f = augment(source, sink, oo)) > 0;)
            flow += f;

      } // level[u] = -1 => source side of min cut
      return flow;
   }
};
```

## Articulation Point

```
vector<bool> visited;
vi t;
vi low;
vector<bool> art;

void dfs_art(vi adj[], int n, int p, int q)
{
   t[n] = q;
   low[n] = q++;
   visited[n] = true;

   int j = 0;

   for (int i = 0; i < adj[n].size(); i++)
   {
      if (!visited[adj[n][i]])
      {
         dfs_art(adj, adj[n][i], n, q);
         low[n] = min(low[adj[n][i]], low[n]);
         j++;

         if (low[adj[n][i]] >= t[n] && p != -1)
         {
            art[n] = true;
         }
      }
      else if (adj[n][i] != p)
      {
         low[n] = min(t[adj[n][i]], low[n]);
      }
   }

   if (p == -1)
   {
      art[n] = j >= 2;
   }
}

void articulationPoints(int V, vi adj[])
{
   visited.assign(V, false);
   t.assign(V, -1);
   low.assign(V, -1);
   art.assign(V, false);

   for (int i = 0; i < V; i++)
   {
      if (!visited[i])
      {
         dfs_art(adj, i, -1, 1);
      }
   }
}
```

## Dijsktra

```
int infinite = (int)1e9;

// O(V^2)
vector<int> dijkstra1(int V, vector<vector<int>> adj[], int S)
{
   vector<int> d;

   d.assign(V, infinite);
   d[S] = 0;

   vector<bool> mask;

   mask.assign(V, false);

   for (int i = 0; i < V; i++)
   {
      int m = infinite;
      int act = -1;

      for (int j = 0; j < V; j++)
      {
         if (mask[j])
            continue;

         if (m > d[j])
         {
            m = d[j];
            act = j;
         }
      }

      for (int j = 0; j < adj[act].size(); j++)
      {
         if (d[act] + adj[act][j][1] < d[adj[act][j][0]])
         {
            d[adj[act][j][0]] = d[act] + adj[act][j][1];
         }
      }

      mask[act] = true;
   }

   return d;
}

// O((V+E)log(E))
vi dijkstra2(int V, vii adj[], int S)
{
   vector<int> d;

   d.assign(V, infinite);
   d[S] = 0;

   priority_queue<pair<int, int>> q;
   q.push({d[S], S});

   while (!q.empty())
   {
      int act = q.top().second;
      int m = abs(q.top().first);
      q.pop();

      if (m > d[act])
         continue;

      for (int j = 0; j < adj[act].size(); j++)
      {
         if (d[act] + adj[act][j].second < d[adj[act][j].first])
         {
            d[adj[act][j].first] = d[act] + adj[act][j].second;
            q.push({-d[adj[act][j].first], adj[act][j].first});
         }
      }
   }

   return d;
}
```

## Bellman Ford

```
int infinite = (int)1e9;

vector<int> bellman_ford(int V, vector<vector<int>> &edges, int S)
{
   vector<int> d;
   d.assign(V, infinite);
   d[S] = 0;

   for (int i = 0; i < V - 1; i++)
   {
      for (int j = 0; j < edges.size(); j++)
      {
         if (d[edges[j][0]] + edges[j][2] < d[edges[j][1]])
         {
            d[edges[j][1]] = d[edges[j][0]] + edges[j][2];
         }
      }
   }

   for (int j = 0; j < edges.size(); j++)
   {
      if (d[edges[j][0]] + edges[j][2] < d[edges[j][1]])
      {
         vector<int> resp(1);
         resp[0] = -1;

         return resp;
      }
   }

   return d;
}
```

## Floyd Warshall

```
int infinite = (int)1e8;

void shortest_distance(vector<vector<int>> &matrix)
{
   for (int k = 0; k < matrix.size(); k++)
   {
      for (int i = 0; i < matrix.size(); i++)
      {
         for (int j = 0; j < matrix[0].size(); j++)
         {
            matrix[i][j] = min(matrix[i][j], matrix[i][k] + matrix[k][j]);
         }
      }
   }
}


void find_path_k(vector<vector<bool>> &matrix, int k)
{
   for (int x = 0; x < k; x++)
   {
      for (int i = 0; i < matrix.size(); i++)
      {
         for (int j = 0; j < matrix[0].size(); j++)
         {
            matrix[i][j] = matrix[i][j] || (matrix[i][x] && matrix[x][j]);
         }
      }
   }
}
```

## Lca

```
class SparseTable
{

private:
   vector<vi> lookup;

   vi arr;

   int rmq(int a, int b)
   {
      if (arr[a] <= arr[b])
         return a;

      return b;
   }

   int operation(int a, int b)
   {
      return rmq(a, b);
   }

   void build_sparse_table()
   {
      int n = arr.size();

      for (int i = 0; i < n; i++)
         lookup[i][0] = i;

      for (int j = 1; (1 << j) <= n; j++)
      {
         for (int i = 0; i <= n - (1 << j); i++)
            lookup[i][j] = operation(lookup[i][j - 1], lookup[i + (1 << (j - 1))][j - 1]);
      }
   }

public:
   SparseTable(vi &a)
   {
      int q = (int)log2(a.size());

      arr.assign(a.size(), 0);
      lookup.assign(a.size(), vi(q + 1));

      for (int i = 0; i < a.size(); i++)
         arr[i] = a[i];

      build_sparse_table();
   }

   int query(int l, int r)
   {
      int q = (int)log2(r - l + 1);

      return operation(lookup[l][q],
                  lookup[r - (1 << q) + 1][q]);
   }

   int get(int i) { return arr[i]; }
};

void dfs(int n, int p, int d, vi adj[], vi &deep, vi &arr)
{
   deep[n] = d;
   arr.push_back(n);

   for (int i = 0; i < adj[n].size(); i++)
   {
      if (p != adj[n][i])
      {
         dfs(adj[n][i], n, d + 1, adj, deep, arr);
         arr.push_back(n);
      }
   }
}

vi lca(int root, int n, vi adj[], vii &querys)
{
   vi deep(n);
   vi first(n);
   vi last(n);
   vi arr;
   vi resp(querys.size());

   dfs(root, -1, 0, adj, deep, arr);

   for (int i = 0; i < arr.size(); i++)
      last[arr[i]] = i;

   for (int i = arr.size() - 1; i >= 0; i--)
      first[arr[i]] = i;
   vi arr_deep(arr.size());

   for (int i = 0; i < arr_deep.size(); i++)
      arr_deep[i] = deep[arr[i]];

   auto s = SparseTable(arr_deep);

   for (int i = 0; i < querys.size(); i++)
   {
      int l = first[querys[i].first];
      int r = last[querys[i].second];

      if (l > r)
      {
         r = last[querys[i].first];
         l = first[querys[i].second];
      }

      int q = s.query(l, r);
      resp[i] = arr[q];
   }

   return resp;
}
```

## Topological Sort

```
vector<int> topoSort(int V, vector<int> adj[])
{
   vector<int> in(V);
   vector<int> resp;

   for (int i = 0; i < V; i++)
   {
      for (int j = 0; j < adj[i].size(); j++)
      {
         in[adj[i][j]]++;
      }
   }

   queue<int> q;

   for (int i = 0; i < V; i++)
   {
      if (in[i] == 0)
         q.push(i);
   }

   while (q.size() != 0)
   {
      int n = q.front();
      q.pop();

      for (int i = 0; i < adj[n].size(); i++)
      {
         in[adj[n][i]]--;

         if (in[adj[n][i]] == 0)
            q.push(adj[n][i]);
      }

      resp.push_back(n);
   }

   return resp;
}
```

## Kruskal

```
class ufds
{
private:
   vector<int> p, rank, sizeSet;
   int disjoinSet;

public:
   ufds(int n)
   {
      p.assign(n, 0);
      rank.assign(n, 0);
      sizeSet.assign(n, 1);
      disjoinSet = n;
      for (int i = 0; i < n; i++)
      {
         p[i] = i;
      }
   }

   int find(int n)
   {
      if (n == p[n])
         return n;
      p[n] = find(p[n]);
      return p[n];
   }

   bool isSameSet(int i, int j) { return find(i) == find(j); }

   void unionSet(int i, int j)
   {
      if (!isSameSet(i, j))
      {
         disjoinSet--;
         int x = find(i);
         int y = find(j);
         if (rank[x] > rank[y])
         {
            p[y] = x;
            sizeSet[x] += sizeSet[y];
         }
         else
         {
            p[x] = y;
            sizeSet[y] += sizeSet[x];
            if (rank[x] == rank[y])
               rank[y]++;
         }
      }
   }

   int numDisjoinset() { return disjoinSet; }

   int sizeofSet(int i) { return sizeSet[find(i)]; }
};

// Function to find sum of weights of edges of the Minimum Spanning Tree.
int spanningTreeKruskal(int V, vector<vector<int>> adj[])
{
   ufds dsu(V);

   vector<pair<int, pair<int, int>>> a;

   for (int i = 0; i < V; i++)
   {
      for (int j = 0; j < adj[i].size(); j++)
      {
         a.push_back({adj[i][j][1], {i, adj[i][j][0]}});
      }
   }

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

   int cost = 0;

   for (int i = 0; i < a.size(); i++)
   {
      if (!dsu.isSameSet(a[i].second.first, a[i].second.second))
      {
         cost += a[i].first;

         dsu.unionSet(a[i].second.first, a[i].second.second);
      }
   }

   return cost;
}
```

## Bridge Edges

```
vector<bool> visited;
vector<int> t;
vector<int> low;
set<pair<int, int>> bridges;

void dfs_bridges(vector<int> adj[], int n, int p, int q)
{
   t[n] = q;
   low[n] = q++;
   visited[n] = true;
   int j = 0;

   for (int i = 0; i < adj[n].size(); i++)
   {
      if (!visited[adj[n][i]])
      {
         dfs_bridges(adj, adj[n][i], n, q);
         low[n] = min(low[adj[n][i]], low[n]);
         j++;
      }
      else if (adj[n][i] != p)
      {
         low[n] = min(t[adj[n][i]], low[n]);
      }
   }

   if (t[n] == low[n] && p != -1)
   {
      bridges.insert({min(n, p), max(n, p)});
   }
}

set<pair<int, int>> bridge_edges(int V, vector<int> adj[])
{
   visited.assign(V, false);
   t.assign(V, -1);
   low.assign(V, -1);
   bridges = set<pair<int, int>>();

   for (int i = 0; i < V; i++)
   {
      if (!visited[i])
      {
         dfs_bridges(adj, i, -1, 1);
      }
   }

   return bridges;
}
```

## Scc Tarjans

```
stack<int> q;
vector<bool> mask;
vector<int> cc_list;

void g_transp(int V, vector<int> adj[], vector<int> new_adj[])
{
   for (int i = 0; i < V; i++)
   {
      for (int j = 0; j < adj[i].size(); j++)
      {
         new_adj[adj[i][j]].push_back(i);
      }
   }
}

void dfs_visit(int n, vector<int> adj[], int cc)
{
   mask[n] = true;

   for (int i = 0; i < adj[n].size(); i++)
   {
      if (!mask[adj[n][i]])
         dfs_visit(adj[n][i], adj, cc);
   }

   if (cc == -1)
      q.push(n);
   else
   {
      cc_list[n] = cc;
   }
}

void tarjans(int V, vector<int> adj[])
{
   vector<int> new_adj[V];
   g_transp(V, adj, new_adj);

   mask.assign(V, false);
   cc_list.assign(V, -1);

   for (int i = 0; i < V; i++)
   {
      if (mask[i])
         continue;
      dfs_visit(i, adj, -1);
   }

   for (int i = 0; i < V; i++)
      mask[i] = false;

   int ind = 0;

   while (q.size() != 0)
   {
      int act = q.top();
      q.pop();

      if (!mask[act])
      {
         dfs_visit(act, new_adj, ind);
         ind++;
      }
   }
}
```

## 3. Algorithms

### Kmp Pf

```
vi prefix_function(string p)
{
   vi pf(p.size());

   pf[0] = 0;
   int k = 0;

   for (int i = 1; i < p.size(); i++)
   {
      while (k > 0 && p[k] != p[i])
         k = pf[k - 1];

      if (p[k] == p[i])
         k++;

      pf[i] = k;
   }

   return pf;
}

vi kmp(string t, string p)
{
   vi result;
   vi pf = prefix_function(p);
   int k = 0;

   for (int i = 0; i < t.size(); i++)
   {
      while (k > 0 && p[k] != t[i])
         k = pf[k - 1];

      if (p[k] == t[i])
         k++;

      if (k == p.size())
      {
         result.push_back(i - (p.size() - 1));
         k = pf[k - 1];
      }
   }

   return result;
}
```

## 4. DataStructure

### Segment Tree

```
class SegmentTree
{
private:
   vi values;

   vi p_values;
   int n;

   int left(int p) { return p << 1; };

   int right(int p) { return (p << 1) + 1; }

   int simple_node(int index) { return values[index]; }

   int prop(int x, int y) { return x + y; }

   void build(int p, int l, int r)
   {
      if (l == r)
      {
         p_values[p] = simple_node(l);
         return;
      }

      build(left(p), l, (l + r) / 2);
      build(right(p), (l + r) / 2 + 1, r);

      p_values[p] = prop(p_values[left(p)], p_values[right(p)]);
   }

   void set(int p, int l, int r, int i, int v)
   {
      if (l == r)
      {
         values[l] = v;
         p_values[p] = simple_node(l);
         return;
      }

      if (i <= (l + r) / 2)
         set(left(p), l, (l + r) / 2, i, v);
      else
         set(right(p), (l + r) / 2 + 1, r, i, v);

      p_values[p] = prop(p_values[left(p)], p_values[right(p)]);
   }

   int query(int p, int l, int r, int lq, int rq)
   {
      if (lq <= l && r <= rq)
         return p_values[p];

      int l1 = l, r1 = (l + r) / 2;
      int l2 = (l + r) / 2 + 1, r2 = r;

      if (l1 > rq || lq > r1)
         return query(right(p), l2, r2, lq, rq);
      if (l2 > rq || lq > r2)
         return query(left(p), l1, r1, lq, rq);

      int lt = query(left(p), l1, r1, lq, rq);
      int rt = query(right(p), l2, r2, lq, rq);

      return prop(lt, rt);
   }

public:
   SegmentTree(vi &a)
   {
      values = a;
      n = a.size();
      p_values.assign(4 * n, 0);
      build(1, 0, n - 1);
   }

   int query(int i, int j) { return query(1, 0, n - 1, i, j); }

   void set(int i, int v) { set(1, 0, n - 1, i, v); }

   int get(int i) { return values[i]; }
};
```

## Sparse Table

```cpp
class SparseTable
{

private:
   vector<vi> lookup;

   vi arr;

   int operation(int a, int b)
   {
      if (arr[a] <= arr[b])
         return a;

      return b;
   }

   int simple_node(int i) { return i; }

   void build_sparse_table()
   {
      int n = arr.size();

      for (int i = 0; i < n; i++)
         lookup[i][0] = simple_node(i);

      for (int j = 1; (1 << j) <= n; j++)
      {
         for (int i = 0; i <= n - (1 << j); i++)
            lookup[i][j] = operation(lookup[i][j - 1],
                                     lookup[i + (1 << (j - 1))][j - 1]);
      }
   }

public:
   SparseTable(vi &a)
   {
      int q = (int)log2(a.size());

      arr.assign(a.size(), 0);
      lookup.assign(a.size(), vi(q + 1));

      for (int i = 0; i < a.size(); i++)
         arr[i] = a[i];

      build_sparse_table();
   }

   int query(int l, int r)
   {
      int q = (int)log2(r - l + 1);

      return operation(lookup[l][q], lookup[r - (1 << q) + 1][q]);
   }

   int get(int i) { return arr[i]; }
};
```

## Heap

```cpp
class Heap
{

private:
   vi p;
   int _size;

   int left(int i) { return 2 * i + 1; }
   int right(int i) { return 2 * i + 2; }
   int father(int i) { return (i - 1) / 2; }
   bool is_leaft(int i) { return i >= _size / 2; }

   void heapify_down_min(int i)
   {
      if (is_leaft(i))
         return;
      if (p[i] > p[left(i)] || p[i] > p[right(i)])
      {
         int item = p[left(i)] > p[right(i)] ? right(i) : left(i);

         int aux = p[i];
         p[i] = p[item];
         p[item] = aux;

         heapify_down_min(item);
      }
   }

   void heapify_down_max(int i)
   {
      if (is_leaft(i))
         return;
      if (p[i] < p[left(i)] || p[i] < p[right(i)])
      {
         int item = p[left(i)] < p[right(i)] ? right(i) : left(i);

         int aux = p[i];
         p[i] = p[item];
         p[item] = aux;

         heapify_down_max(item);
      }
   }

   void heapify_up_min(int i)
   {
      if (i == 0)
         return;
      if (p[i] < p[father(i)])
      {
         int aux = p[i];
         p[i] = p[father(i)];
         p[father(i)] = aux;

         heapify_up_min(father(i));
      }
   }

   void heapify_up_max(int i)
   {
      if (i == 0)
         return;
      if (p[i] > p[father(i)])
      {
         int aux = p[i];
         p[i] = p[father(i)];
         p[father(i)] = aux;

         heapify_up_max(father(i));
      }
   }

   bool is_heap_min(int i)
   {
      if (is_leaft(i))
         return true;

      return p[left(i)] >= p[i] && p[right(i)] >= p[i];
   }

   bool is_heap_max(int i)
   {
      if (is_leaft(i))
         return true;

      return p[left(i)] <= p[i] && p[right(i)] <= p[i];
   }

   void build_heap_min()
   {
      for (int i = size() - 1; i >= 0; i++)
      {
         if (!is_heap_min(i))
            heapify_down_min(i);
      }
   }

   void build_heap_max()
   {

      for (int i = size() - 1; i >= 0; i--)
      {
         if (!is_heap_max(i))
         {
            heapify_down_max(i);
         }
      }
   }

public:
   Heap()
   {
      _size = 0;
   }

   Heap(vi &v)
   {
      p = v;
      _size = p.size();

      build_heap_max();
   }

   void push(int n)
```

```
  {
     if (_size == p.size())
        p.push_back(n);
     else
        p[_size] = n;

     _size++;

     heapify_up_max(_size - 1);
  }

  int top()
  {
     return p[0];
  }

  void pop()
  {
     p[0] = p[_size - 1];

     _size--;

     heapify_down_max(0);
  }

  int size()
  {
     return _size;
  }
};
```

## Abi

```
class Abi
{

private:
   vi p;
   int _size;

   int ls_one(int i) { return i & (-i); }

public:
   Abi(int n)
   {
      _size = n;
      p.assign(n + 1, 0);
   }

   int rsq(int k)
   {
      int sum = 0;

      for (int i = k; i > 0; i -= ls_one(i))
      {
         sum += p[i];
      }

      return sum;
   }

   int sum(int a, int b) { return rsq(b) - rsq(a - 1); }

   void adjust_sum(int k, int v)
   {
      for (int i = k; i < p.size(); i += ls_one(i))
         p[i] += v;
   }

   int size()
   {
      return _size;
   }
};
```

## Avl

```
struct avl
{
   int key;
   int height;
   int size;
   avl *left;
   avl *right;

   avl(int k)
   {
      key = k;
      height = 1;
      size = 1;
      left = NULL;
      right = NULL;
   }

   int getBalance()
   {
      int leftHeight = 0;
      int rightHeight = 0;

      if (left != NULL)
         leftHeight = left->height;

      if (right != NULL)
         rightHeight = right->height;

      return leftHeight - rightHeight;
   }

   void updateSize()
   {
      int leftSize = 0;
      int rightSize = 0;

      if (left != NULL)
         leftSize = left->size;

      if (right != NULL)
         rightSize = right->size;

      size = leftSize + rightSize + 1;
   }

   void updateHeight()
   {
      int leftHeight = 0;
      int rightHeight = 0;

      if (left != NULL)
         leftHeight = left->height;

      if (right != NULL)
         rightHeight = right->height;

      height = max(leftHeight, rightHeight) + 1;
   }

   avl *rotateLeft()
   {
      avl *newRoot = right;
      right = newRoot->left;
      newRoot->left = this;
      updateHeight();
      newRoot->updateHeight();
      return newRoot;
   }

   avl *rotateRight()
   {
      avl *newRoot = left;
      left = newRoot->right;
      newRoot->right = this;
      updateHeight();
      newRoot->updateHeight();
      return newRoot;
   }

   avl *balance()
   {
      updateHeight();
      updateSize();
      int balance = getBalance();

      if (balance == 2)
      {
         if (left->getBalance() < 0)
            left = left->rotateLeft();
         return rotateRight();
      }

      if (balance == -2)
      {
```

```
      if (right->getBalance() > 0)
         right = right->rotateRight();
      return rotateLeft();
   }

   return this;
}

avl *insert(int k)
{
   if (k < key)
   {
      if (left == NULL)
         left = new avl(k);
      else
         left = left->insert(k);
   }
   else
   {
      if (right == NULL)
         right = new avl(k);
      else
         right = right->insert(k);
   }

   return balance();
}

avl *findMin()
{
   if (left == NULL)
      return this;
   else
      return left->findMin();
}

avl *removeMin()
{
   if (left == NULL)
      return right;
   left = left->removeMin();
   return balance();
}

avl *remove(int k)
{
   if (k < key)
      left = left->remove(k);
   else if (k > key)
      right = right->remove(k);
   else
   {
      avl *leftChild = left;
      avl *rightChild = right;

      delete this;

      if (rightChild == NULL)
         return leftChild;

      avl *min = rightChild->findMin();
      min->right = rightChild->removeMin();
      min->left = leftChild;
      return min->balance();
   }

   return balance();
}

int getRank(int k)
{
   if (k < key)
   {
      if (left == NULL)
         return 0;
      else
         return left->getRank(k);
   }
   else if (k > key)
   {
      if (right == NULL)
         return 1 + left->size;
      else
         return 1 + left->size + right->getRank(k);
   }
   else
      return left->size;
}

int getKth(int k)
{
   if (k < left->size)
      return left->getKth(k);
   else if (k > left->size)
```

```
            return right->getKth(k - left->size - 1);
        else
            return key;
    }

    ~avl()
    {
        if (left != NULL)
            delete left;

        if (right != NULL)
            delete right;
    }
};
```

## Trie

```
class Trie
{
private:
    int cant_string;
    int cant_string_me;
    int cant_node;
    char value;
    Trie *children[alphabet];

public:
    Trie(char a)
    {
        cant_string = 0;
        cant_node = 1;
        cant_string_me = 0;
        value = a;

        for (int i = 0; i < alphabet; i++)
            children[i] = NULL;
    }

    pair<Trie *, int> search(string s)
    {
        Trie *node = this;
        int i = 0;

        while (i < s.size() && node->children[s[i] - first_char] != NULL)
        {
            node = node->children[s[i] - first_char];

            i++;
        }

        return {node, i};
    }

    void insert(string s)
    {
        int q = s.size() - search(s).second;

        Trie *node = this;

        for (int i = 0; i < s.size(); i++)
        {
            node->cant_node += q;

            if (node->children[s[i] - first_char] == NULL)
            {
                node->children[s[i] - first_char] = new Trie(s[i]);
                q--;
            }

            node = node->children[s[i] - first_char];
            node->cant_string_me++;
        }

        node->cant_string++;
    }

    void eliminate(string s)
    {
        if (!contains(s))
            return;

        Trie *node = this;
        int q = 0;

        for (int i = 0; i < s.size(); i++)
        {
            if (node->children[s[i] - first_char] == NULL)
            {
                node->children[s[i] - first_char] = new Trie(s[i]);
```

```
         }

         if (node->children[s[i] - first_char]->cant_string_me == 1)
         {
            node->children[s[i] - first_char] = NULL;

            q = s.size() - i;
            break;
         }

         node = node->children[s[i] - first_char];
         node->cant_string_me--;

         if (i == s.size() - 1)
            node->cant_string--;
      }

      node = this;

      for (int i = 0; i < s.size() - q + 1; i++)
      {
         node->cant_node -= q;
         node = node->children[s[i] - first_char];
      }
   }

   bool contains(string s)
   {
      auto q = search(s);
      return q.second == s.size() && q.first->cant_string >= 1;
   }

   int cant_words_me() { return cant_string_me; }

   int cant_words() { return cant_string; }

   Trie *get(char a) { return children[a - first_char]; }

   int size() { return cant_node; }
};
```

## Ufds

```
class ufds
{
private:
   vector<int> p, rank, sizeSet;
   int disjoinSet;

public:
   ufds(int n)
   {
      p.assign(n, 0);
      rank.assign(n, 0);
      sizeSet.assign(n, 1);
      disjoinSet = n;
      for (int i = 0; i < n; i++)
      {
         p[i] = i;
      }
   }

   int find(int n)
   {
      if (n == p[n])
         return n;
      p[n] = find(p[n]);
      return p[n];
   }

   bool isSameSet(int i, int j) { return find(i) == find(j); }

   void unionSet(int i, int j)
   {
      if (!isSameSet(i, j))
      {
         disjoinSet--;
         int x = find(i);
         int y = find(j);
         if (rank[x] > rank[y])
         {
            p[y] = x;
            sizeSet[x] += sizeSet[y];
         }
         else
         {
            p[x] = y;
            sizeSet[y] += sizeSet[x];
            if (rank[x] == rank[y])
            rank[y]++;
        }
    }
}
```

```
    int numDisjoinset() { return disjoinSet; }

    int sizeofSet(int i) { return sizeSet[find(i)]; }
};
```

## Segment Tree Lazy

```
class SegmentTreeLazy
{
private:
   vi values;
   vector<bool> lazy;
   vi l_values;
   vi p_values;
   int n;

   int left(int p) { return p << 1; };

   int right(int p) { return (p << 1) + 1; }

   int simple_node(int index) { return values[index]; }

   int prop(int x, int y) { return x + y; }

   int prop_lazy(int x, int y) { return x + y; }

   int prop_lazy_up(int x, int y, int s) { return x + y * s; }

   void update_lazy(int p, int l, int r)
   {
      if (l == r)
      {
         values[l] = prop_lazy(values[l], l_values[p]);
      }

      p_values[p] = prop_lazy_up(p_values[p], l_values[p], r - l + 1);
   }

   void propagate_lazy(int p, int l, int r)
   {
      lazy[p] = false;

      if (l == r)
         return;

      l_values[left(p)] = lazy[left(p)]
                        ? prop_lazy(l_values[left(p)], l_values[p])
                        : l_values[p];
      l_values[right(p)] = lazy[right(p)]
                         ? prop_lazy(l_values[right(p)], l_values[p])
                         : l_values[p];

      lazy[left(p)] = true;
      lazy[right(p)] = true;
   }

   void build(int p, int l, int r)
   {
      if (l == r)
      {
         p_values[p] = simple_node(l);
         return;
      }

      build(left(p), l, (l + r) / 2);
      build(right(p), (l + r) / 2 + 1, r);

      p_values[p] = prop(p_values[left(p)], p_values[right(p)]);
   }

   void set(int p, int l, int r, int i, int v)
   {
      if (lazy[p])
      {
         update_lazy(p, l, r);
         propagate_lazy(p, l, r);
      }

      if (l == r)
      {
         values[l] = v;
         p_values[p] = simple_node(l);
         return;
      }

      if (i <= (l + r) / 2)
         set(left(p), l, (l + r) / 2, i, v);
      else
         set(right(p), (l + r) / 2 + 1, r, i, v);

      p_values[p] = prop(p_values[left(p)], p_values[right(p)]);
   }

   int query(int p, int l, int r, int lq, int rq)
   {
      if (lazy[p])
      {
         update_lazy(p, l, r);
         propagate_lazy(p, l, r);
      }

      if (lq <= l && r <= rq)
         return p_values[p];

      int l1 = l, r1 = (l + r) / 2;
      int l2 = (l + r) / 2 + 1, r2 = r;

      if (l1 > rq || lq > r1)
         return query(right(p), l2, r2, lq, rq);
      if (l2 > rq || lq > r2)
         return query(left(p), l1, r1, lq, rq);

      int lt = query(left(p), l1, r1, lq, rq);
      int rt = query(right(p), l2, r2, lq, rq);

      return prop(lt, rt);
   }

   void set_rank(int p, int l, int r, int lq, int rq, int value)
   {
      if (lazy[p])
      {
         update_lazy(p, l, r);
         propagate_lazy(p, l, r);
      }

      if (l > rq || lq > r)
         return;

      if (lq <= l && r <= rq)
      {
         lazy[p] = true;
         l_values[p] = value;
         update_lazy(p, l, r);
         propagate_lazy(p, l, r);
         return;
      }

      set_rank(left(p), l, (l + r) / 2, lq, rq, value);
      set_rank(right(p), (l + r) / 2 + 1, r, lq, rq, value);

      p_values[p] = prop(p_values[left(p)], p_values[right(p)]);
   }

   int get(int p, int l, int r, int i)
   {
      if (lazy[p])
      {
         update_lazy(p, l, r);
         propagate_lazy(p, l, r);
      }

      if (l == r)
         return values[i];

      if (i <= (l + r) / 2)
         return get(left(p), l, (l + r) / 2, i);

      return get(right(p), (l + r) / 2 + 1, r, i);
   }

public:
   SegmentTreeLazy(vi &a)
   {
      values = a;
      n = a.size();
      p_values.assign(4 * n, 0);
      lazy.assign(4 * n, false);
      l_values.assign(4 * n, 0);
      build(1, 0, n - 1);
   }

   int query(int i, int j) { return query(1, 0, n - 1, i, j); }

   void set(int i, int v) { set(1, 0, n - 1, i, v); }

   void set_rank(int i, int j, int v) { set_rank(1, 0, n - 1, i, j, v); }
```

```
  int get(int i) { return get(1, 0, n - 1, i); }
```

```
        };
```