1. Template

Main

```
#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. DataStructure

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;
   }
};
```

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]; }
};
```

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); }
};
```

Sparse Table

```
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]; }
};
```

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] = nullptr;
   }

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

      while (node->children[s[i] - ’a’] != nullptr && i < s.size())
      {
         node = node->children[s[i] - ’a’];
         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] - ’a’] == nullptr)
         {
            node->children[s[i] - ’a’] = new Trie(s[i]);
            q--;
         }

         node = node->children[s[i] - ’a’];
         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] - ’a’] == nullptr)
         {
            node->children[s[i] - ’a’] = new Trie(s[i]);
         }

         if (node->children[s[i] - ’a’]->cant_string_me == 1)
         {
            node->children[s[i] - ’a’] = nullptr;

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

         node = node->children[s[i] - ’a’];
         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] - ’a’];
      }
   }

   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 - ’a’]; }

   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)]; }
};
```