Team Notebook

UITS-O(Struggle) - University of Information Technology and Sciences

March 9, 2023

Contents

1 Data Structures **2**
- 1.1 1D Segment Tree . . . 2
- 1.2 2D Segment Tree . . . 2
- 1.3 Disjoint Set union . . . 3
- 1.4 Lazy Segment Tree . . . 3
- 1.5 Monotonic Stack (Increasing) . . . 3
- 1.6 PBDS with Deletion in Multiset . . . 4
- 1.7 Policy Based Data Structure . . . 4
- 1.8 Sparse Table . . . 4

2 Graph Theory **5**
- 2.1 Bellman Ford . . . 5
- 2.2 Dijkstras . . . 5
- 2.3 Floyd Warshall . . . 6
- 2.4 Kruskals MST . . . 6
- 2.5 Lowest Common Ancestor . . . 6

3 Misc **6**
- 3.1 PBDS and Modular Arithmetic . . . 6

4 Number Theory **7**
- 4.1 Binary Exponentiation . . . 7
- 4.2 Binomial Coefficients nCr . . . 7
- 4.3 Catalan Number . . . 7
- 4.4 Chinese Remainder Theorem . . . 7
- 4.5 Euler Totient (precomputation) . . . 7
- 4.6 Euler Totient (single) . . . 7
- 4.7 Extended GCD . . . 8
- 4.8 Fermats Primality test . . . 8
- 4.9 Matrix Exponentiation . . . 8
- 4.10 Mobius Function . . . 8
- 4.11 Trivial nCr . . . 8
- 4.12 nCr Table . . . 9

5 Strings **9**
- 5.1 Aho Corasick . . . 9
- 5.2 Knuth Morris Pratt . . . 9
- 5.3 Manachers . . . 10
- 5.4 String Hashing (double) . . . 10
- 5.5 Trie . . . 11

1 Data Structures

1.1 1D Segment Tree

```cpp
/** 1. segment_tree<long long> seg_tree(a);
  * 2. seg_tree.build(1, 0, n - 1);
  * 3. Note : Make sure that the segment tree type and the
       vector type must mathc. E.g If, struct segment_tree <
            long long>
    then, vector must be vector<long long>
  * 4. Note : While using index for answer. Make sure to use
        them as (0 based)
  * 5. now, you're good to go.
 **/
template <typename T>
struct segment_tree {
  int n;
  vector<T> a, tree;
  /* Used to create the tree array */
  segment_tree (vector<T> cpy) {
    a = cpy;
    n = (int) a.size();
    tree.assign(n << 2, 0);
  };
  /* used to build the tree */
  void build (int node, int l, int r) {
    if (l == r) {
      tree[node] = a[l];
      return;
    }
    int mid = (l + r) >> 1;
    build(node << 1, l, mid);
    build((node << 1) + 1, mid + 1, r);
    tree[node] = tree[node << 1] + tree[(node << 1) + 1];
  }
  /* point sum or range sum */
  T sum (int node, int start, int end, int l, int r) {
    if (end < l or r < start) {
      return 0;
    }
    if (l <= start and end <= r) {
      return tree[node];
    }
    int mid = (start + end) >> 1;
    T left_sum = sum(node << 1, start, mid, l, r);
    T right_sum = sum((node << 1) + 1, mid + 1, end, l, r);
    return left_sum + right_sum;
  }
  /* point updating value / adding value */
  void update (int node, int start, int end, int id, T val)
      {
    if (start == end) {
      tree[node] = val;
      return;
    }
    int mid = (start + end) >> 1;
    if (id <= mid) {
      update(node << 1, start, mid, id, val);
    } else {
      update((node << 1) + 1, mid + 1, end, id, val);
    }
    tree[node] = tree[node << 1] + tree[(node << 1) + 1];
  }
};
```

1.2 2D Segment Tree

```cpp
/** 1. _2D_segment_tree<long long> _2D_seg_tree(a);
  * 2. _2D_seg_tree.buildx(1, 0, n - 1);
  * 3. Note : Make sure that the segment tree type and the
       vector type must mathc. E.g If, struct
            _2D_segment_tree <long long>
    then, vector must be vector<long long>
  * 4. Note : While using index for answer. Make sure to use
        them as (0 based)
  * 5. now, you're good to go.
 **/
template <typename T>
struct _2D_segment_tree {
  int n, m;
  vector<vector<T>> a, t;
  _2D_segment_tree (vector<vector<T>> a) {
    this -> a = a;
    this -> n = (int) a.size();
    this -> m = (int) a[0].size();
    t.assign(n << 2, vector<T>(m << 2));
  }
  void build_y (int vx, int lx, int rx, int vy, int ly, int
       ry) {
    if (ly == ry) {
      if (lx == rx) {
        t[vx][vy] = a[lx][ly];
      } else {
        t[vx][vy] = t[(vx << 1)][vy] + t[(vx << 1) + 1][vy];
      }
    } else {
      int my = (ly + ry) >> 1;
      build_y(vx, lx, rx, (vy << 1), ly, my);
      build_y(vx, lx, rx, (vy << 1) + 1, my + 1, ry);
      t[vx][vy] = t[vx][(vy << 1)] + t[vx][(vy << 1) + 1];
    }
  }
  /* Prepares the _2D segment tree
   * _2D_seg_tree.build_x(1, 0, n - 1); */
  void build_x (int vx, int lx, int rx) {
    if (lx != rx) {
      int mx = (lx + rx) >> 1;
      build_x((vx << 1), lx, mx);
      build_x((vx << 1) + 1, mx + 1, rx);
    }
    build_y(vx, lx, rx, 1, 0, m - 1);
  }
  T sum_y (int vx, int vy, int tly, int try_, int ly, int ry
      ) {
    if (ly > ry) {
      return (T) 0;
    }
    if (ly == tly && try_ == ry) {
      return t[vx][vy];
    }
    int tmy = (tly + try_) >> 1;
    return sum_y(vx, (vy << 1), tly, tmy, ly, min(ry, tmy))
       + sum_y(vx, (vy << 1) + 1, tmy + 1, try_, max(ly, tmy
            + 1), ry);
  }
  /* Returns the sum of a sub-matrix from,
   * [(left_x, left_y) top_left corner] to [(right_x,
       right_y) bottom_right corner]
   * _2D_seg_tree.sum_x(1, 0, n - 1, --left_x, --right_x, --
       left_y, --right_y) -> 0 based indexing */
  T sum_x (int vx, int tlx, int trx, int lx, int rx, int ly,
       int ry) {
    if (lx > rx) {
      return 0;
    }
    if (lx == tlx && trx == rx) {
      return sum_y(vx, 1, 0, m - 1, ly, ry);
    }
    int tmx = (tlx + trx) >> 1;
    return sum_x((vx << 1), tlx, tmx, lx, min(rx, tmx), ly,
        ry)
       + sum_x((vx << 1) + 1, tmx + 1, trx, max(lx, tmx + 1),
            rx, ly, ry);
  }
  void update_y (int vx, int lx, int rx, int vy, int ly, int
       ry, int x, int y, T new_val) {
    if (ly == ry) {
      if (lx == rx) {
```

```
      t[vx][vy] = new_val;
    } else {
      t[vx][vy] = t[(vx << 1)][vy] + t[(vx << 1) + 1][vy];
    }
  } else {
    int my = (ly + ry) >> 1;
    if (y <= my) {
      update_y(vx, lx, rx, (vy << 1), ly, my, x, y, new_val
          );
    } else {
      update_y(vx, lx, rx, (vy << 1) + 1, my + 1, ry, x, y,
          new_val);
    }
    t[vx][vy] = t[vx][(vy << 1)] + t[vx][(vy << 1) + 1];
  }
}
/* Updates a particular cell of the matrix - (x_axis,
    y_axis)
 * _2D_seg_tree.update_x(1, 0, n - 1, --left_x, --left_y,
     new_val); */
void update_x (int vx, int lx, int rx, int x, int y, T
    new_val) {
  if (lx != rx) {
    int mx = (lx + rx) >> 1;
    if (x <= mx) {
      update_x((vx << 1), lx, mx, x, y, new_val);
    } else {
      update_x((vx << 1) + 1, mx + 1, rx, x, y, new_val);
    }
  }
  update_y(vx, lx, rx, 1, 0, m - 1, x, y, new_val);
}
};
```

1.3 Disjoint Set union

```
/* Time complexity for each query: O(logN) */
/* Step 1: disjoint_set<int> dsu(n + 1); */
/* Step 2: dsu.make_set(u, v); */
vector<int> par, siz;
template<typename T>
struct disjoint_set {
  int n;
  T find_set(T v) {
    if (par[v] == v) {
      return v;
    }
    return par[v] = find_set(par[v]);
  }
  void init(T v) {
    par[v] = v;
    siz[v] = 1;
  }
  disjoint_set (int n) {
    this -> n = n;
    siz.assign(n + 1, 0);
    par.assign(n + 1, 0);
    for (int u = 1; u <= n; ++u) {
      init(u);
    }
  }
  void make_set(T a, T b) {
    a = find_set(a);
    b = find_set(b);
    if (a != b) {
      if (siz[a] < siz[b]) {
        swap(a, b);
      }
      par[b] = a;
      siz[a] += siz[b];
    }
  }
  T find_group_size(T a) {
    a = find_set(a);
    return siz[a];
  }
};
```

1.4 Lazy Segment Tree

```
/** 1. struct lazy_propagation <int64_t>lazy_prop(a);
  * 2. lazy_prop.build(1, 0, n - 1);
  * 3. now, you're good to go.
 **/
template <typename T>
struct lazy_propagation {
  struct info {
    T sum = 0, prop = 0;
  };
  int n;
  vector<T> a;
  vector<info> tree;
  lazy_propagation (vector<T> cpy) {
    a = cpy;
    n = (int) a.size();
    tree.resize(4 * n);
  };
  void build (int node, int l, int r) {
    if (l == r) {
      tree[node].sum = a[l];
      return;
    }
    int mid = (l + r) >> 1;
    build(node << 1, l, mid);
    build((node << 1) + 1, mid + 1, r);
    tree[node].sum = tree[node << 1].sum + tree[(node << 1) +
        1].sum;
  }
  T sum (int node, int start, int end, int l, int r, T carry
      = 0) {
    if (end < l or r < start) {
      return 0;
    }
    if (l <= start and end <= r) {
      return tree[node].sum + (((end - start) + 1) * carry);
    }
    int mid = (start + end) >> 1;
    T left_sum = sum(node << 1, start, mid, l, r, carry +
        tree[node].prop);
    T right_sum = sum((node << 1) + 1, mid + 1, end, l, r,
        carry + tree[node].prop);
    return left_sum + right_sum;
  }
  void update (int node, int start, int end, int l, int r, T
      val) {
    if (end < l or r < start) {
      return;
    }
    if (l <= start and end <= r) {
      tree[node].sum += ((end - start) + 1) * val;
      tree[node].prop += val;
      return;
    }
    int mid = (start + end) >> 1;
    update(node << 1, start, mid, l, r, val);
    update((node << 1) + 1, mid + 1, end, l, r, val);
    tree[node].sum = tree[node << 1].sum + tree[(node << 1) +
        1].sum + (((end - start) + 1) * tree[node].prop);
  }
};
```

1.5 Monotonic Stack (Increasing)

```
// C++ code to implement the approach
#include <bits/stdc++.h>
using namespace std;
```

```cpp
// Function to build Monotonic
// increasing stack
void increasingStack(int arr[], int N) {
    // Initialise stack
    stack<int> stk;

    for (int i = 0; i < N; i++) {

        // Either stack is empty or
        // all bigger nums are popped off
        while (stk.size() > 0 && stk.top() > arr[i]) {
            stk.pop();
        }
        stk.push(arr[i]);
    }

    int N2 = stk.size();
    int ans[N2] = { 0 };
    int j = N2 - 1;

    // Empty Stack
    while (!stk.empty()) {
        ans[j] = stk.top();
        stk.pop();
        j--;
    }

    // Displaying the original array
    cout << "The Array: ";
    for (int i = 0; i < N; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;

    // Displaying Monotonic increasing stack
    cout << "The Stack: ";
    for (int i = 0; i < N2; i++) {
        cout << ans[i] << " ";
    }
    cout << endl;
}

// Driver code
int main() {
    int arr[] = { 1, 4, 5, 3, 12, 10 };
    int N = sizeof(arr) / sizeof(arr[0]);

    // Function Call
    increasingStack(arr, N);
    return 0;
}
```

1.6 PBDS with Deletion in Multiset

```cpp
/* Necessary includes */
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using   namespace __gnu_pbds;

/* The data structure */
typedef tree<ll, null_type, less< ll >, rb_tree_tag,
     tree_order_statistics_node_update> ordered_set;

/* Functionalities */
ordered_set s; //declaring pbds
s.insert(x); //taking input
s.find(x)==s.end() // search for a present or not
s.order_of_key(x) //postion of x in sorted set
*s.find_by_order(r); // value presnt at index r
s1.insert({x,cnt++}); //insert in multiset
s1.erase(s1.lower_bound({x,-1})); //erase in multiset
s1.find_by_order(x)->first //value of index x in multiset
```

1.7 Policy Based Data Structure

```cpp
/** 1. Firstly, place the header files and namespace and set
      the data type and comparator.
 * 2. ordered_set X;
 * 3. X.insert(8);
 * 4. *X.find_by_order(1)
 *       Note : finds the kth largest or the kth smallest
      element (counting from zero)
 *           i.e. The element at the position i (powerful)
 * 5. X.order_of_key(3)
 *       Note : finds the number of items in a set that are
      strictly smaller than our item
 *           i.e. The position of the current element (
      powerful)
 * Note : This will exactly work like set, multiset, map [
      also can use their functionalities.
 *   -> Not possible : erasing elements with their value (in
       multiset)
 */
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
typedef tree <
  int, // type
  null_type, // use mapped_type for map
  less<int>, // comparator (less/greater) & type [less_equal
       for multiset]
  rb_tree_tag,
  tree_order_statistics_node_update
> ordered_set;
// Returns the position if found, else returns size
auto pbds_lower_bound = [&] (int el) {
  return (int) ms.order_of_key(el);
};
// Returns the position if found, else returns size
auto pbds_upper_bound = [&] (int el) {
  return (int) ms.order_of_key(el + 1);
};
```

1.8 Sparse Table

```cpp
/* Used for answering queries, but can answer R(Min/Max)Q in
      O(1) */
/* Step 1: sparse_table<int> st(a, N); */
/* Step 2: cout << st.min_query(l, r) << '\n'; */
template <typename T>
struct sparse_table {
  int N;
  int n, k;
  vector<T> a;
  vector<T> logs;
  vector<vector<T>> st;
  void gen_logs () {
    logs[1] = 0;
    for (int i = 2; i <= N; i++) {
      logs[i] = logs[i/2] + 1;
    }
  }
  // builds the table
  void proc () {
    st[0] = a;
    for (int i = 1; i <= k; ++i) {
      for (int j = 0; j + (1 << i) - 1 < n; ++j) {
        /* Change this line according the question */
        st[i][j] = min(st[i - 1][j], st[i - 1][j + (1 << (i -
              1))]);
      }
    }
  }
  // builds the structure
  sparse_table (vector<T> a, int N) {
```

```
    this -> a = a;
    this -> N = N;
    n = (int) a.size();
    logs.assign(N + 1, 0);
    gen_logs();
    k = logs[n];
    st.assign(k + 1, vector<T>(n + 1, 0));
    proc();
  }
  /* This function is used for only min/max query O(1); */
  T min_query (int l, int r) {
    int p = logs[r - l + 1];
    return min(st[p][l], st[p][r - (1 << p) + 1]);
  }
  /* This function is used for the rest of the queries - in
      O(log(n)) */
  T sum_query (int l, int r) {
    T sum = 0;
    for (int i = k; i >= 0; --i) {
      if ((1 << i) <= r - l + 1) {
        sum += st[i][l];
        l += 1 << i;
      }
    }
    return sum;
  }
};
```

2 Graph Theory

2.1 Bellman Ford

```
/* Write this in main function */
/* Time complexity: O(V * E)
 * In case, E = V^2 then, O(V^3) */
/* Add edges both ways for undirected graph */
/* g.push_back({u, v, w}); */
/* g.push_back({v, u, w}); */
int cyc_node = -1;
const int inf = (int) 1e9;
vector<int> dist(n + 1, inf);
vector<int> par(n + 1, -1);
auto check_neg_cycle = [&] () {
  if (cyc_node == -1) {
    return printf("No neg cycle\n"), 0;
  }
  int u = cyc_node;
  for (int i = 1; i <= n; ++i) {
    u = par[u];
  }
  vector<int> path;
  for (int cur = u; ; cur = par[cur]) {
    path.push_back(cur);
    if (1 < (int) path.size() and cur == u) {
   break;
    }
  }
  reverse(path.begin(), path.end());
  for (auto p : path) {
    printf("%d ", p);
  }
  printf("\n");
  return 0;
};
auto bellman_ford = [&] (int src) {
  dist[src] = 0;
  for (int i = 1; i <= n; ++i) {
    cyc_node = -1;
    for (int j = 0; j < m; ++j) {
   int u = g[j][0];
   int v = g[j][1];
   int w = g[j][2];
   if (dist[u] < inf) {
     if (dist[u] + w < dist[v]) {
    dist[v] = max(-inf, dist[u] + w);
    par[v] = u;
    cyc_node = v;
     }
   }
    }
  }
  // remove this if not needed
  check_neg_cycle();
};
bellman_ford(1);
```

2.2 Dijkstras

```
/* Used to extract the shortest path from source(u) to
     destination(v) */
/* Time complexity: O(V + E log V) */
/* Step 1: dijkstra<int> dij(g, n, --src); */
/* Step 2: dij.proc_tab(); */
/* Step 3: cout << dij.get_dist(--v) << '\n'; */
/* Step 4: auto path = dij.get_path(v); */
template <typename T>
struct dijkstra {
int n;
int src;
// Change this (inf) according to the question
const T inf = (T) 1e16;
vector<int> par, seen;
vector<T> dist;
vector<vector<array<int, 2>>> g;
dijkstra (vector<vector<array<int, 2>>> g, int n, int src)
      {
  this -> g = g;
  this -> n = n;
  this -> src = src;
  // Remove this (par) if not needed
  par.assign(n, -1);
  seen.assign(n, false);
  dist.assign(n, inf);
}
// Processes the distance table
void proc_tab () {
  multiset<array<T, 2>> ms;
  dist[src] = 0;
  ms.insert({0, src});
  while (!ms.empty()) {
    auto u = *ms.begin();
    ms.erase(ms.begin());
    if (!seen[u[1]]) {
      seen[u[1]] = true;
      for (auto ch : g[u[1]]) {
        if (dist[u[1]] + ch[1] < dist[ch[0]]) {
          dist[ch[0]] = dist[u[1]] + ch[1];
          /* Here saving the previous node as parent if
               this is giving less cost */
          par[ch[0]] = u[1];
          ms.insert({dist[ch[0]], ch[0]});
        }
      }
    }
  }
}
// Returns the shortest distance from source to
     destination
T get_dist (int dest) {
  return dist[dest];
}
// Returns the shortest path from source to destination
vector<int> get_path (int dest) {
  vector<int> path;
  for (int v = dest; v != -1; v = par[v]) {
    path.push_back(v + 1);
  }
```

```
    reverse(path.begin(), path.end());
    return path;
  }
};
```

2.3 Floyd Warshall

```
/* Time complexity: O(n ^ 3)
 * Computes the all pair shortest paths */
auto floyd_warshall = [&] () {
  /* In order to work with this algorithm, the graph needs
     to be represented in adjacency matrix form. */
  /* init the (d) array, if there doesn't exists a path b/w
     u-v then set them to infinity */
  const int inf = (int) 1e9;
  for (int k = 1; k <= n; ++k) {
    for (int i = 1; i <= n; ++i) {
      for (int j = 1; j <= n; ++j) {
        /* if there exists both of these path or not */
        if (d[i][k] < inf and d[k][j] < inf) {
          d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
        }
      }
    }
  }
};
```

2.4 Kruskals MST

```
/* Make sure to write this algorithm in main function */
/* Before this, make sure to write the dsu algo */
/* Time complexity: O(mlog(m)) - only for sorting, others
    done in constant time */
disjoint_set<int> dsu(n + 1);
auto kruskals = [&] () {
  long long min_cost = 0;
  /* The edges must be sorted in asc according to their
     weights */
  sort(p.begin(), p.end());
  for (int i = 0; i < m; ++i) {
    /* p[i][0] = cost, p[i][1] = u, p[i][2] = v; */
    if (dsu.find_set(p[i][1]) != dsu.find_set(p[i][2])) {
      min_cost += p[i][0];
      dsu.make_set(p[i][1], p[i][2]);
    }
  }
  return min_cost;
};
```

2.5 Lowest Common Ancestor

```
/* Time complexity: Build O(nlog(n)), Query O(log(n)); */
/* Step 1: binary_lifting bl(n, m, g); */
/* Step 2: bl.lca(u, v); */
/* Step 3: bl.get_dist(u, v); */
struct binary_lifting {
  int n, m;
  vector<int> lvl;
  vector<vector<int>> g;
  vector<vector<int>> par;
  void dfs (int v, int l, int p) {
    lvl[v] = l;
    par[v][0] = p;
    for (auto ch : g[v]) {
      if (ch != p) {
        dfs(ch, l + 1, v);
      }
    }
  }
  void init () {
    dfs(1, 0, -1);
    const int logn = __lg(n);
    for (int i = 1; i <= logn; ++i) {
      for (int j = 1; j <= n; ++j) {
        if (par[j][i - 1] != -1) {
          int p = par[j][i - 1];
          par[j][i] = par[p][i - 1];
        }
      }
    }
  }
  // builds the structure
  binary_lifting (int n, int m, vector<vector<int>> g) {
    this -> n = n;
    this -> m = m;
    this -> g = g;
    lvl.assign(n + 1, 0);
    const int logn = __lg(n);
    par.assign(n + 1, vector<int>(logn + 1, -1));
    init();
  }
  // Returns the lowest common ancestor of two nodes
  int lca (int u, int v) {
    if (lvl[v] < lvl[u]) {
      swap(v, u);
    }
    int d = lvl[v] - lvl[u];
    while (d) {
      int logd = __lg(d);
      v = par[v][logd];
      d -= (1 << logd);
    }
    if (u == v) {
      return u;
    }
    int logn = __lg(n);
    for (int i = logn; i >= 0; --i) {
      if (par[u][i] != -1 and par[u][i] != par[v][i]) {
        u = par[u][i];
        v = par[v][i];
      }
    }
    return par[u][0];
  }
  // Returns the distance between two nodes
  int get_dist (int u, int v) {
    int com_ances = lca(u, v);
    return lvl[u] + lvl[v] - (lvl[com_ances] << 1);
  }
};
```

3 Misc

3.1 PBDS and Modular Arithmetic

```
//Policy based data-structure

#include <ext/pb_ds/assoc_container.hpp>
using namespace __gnu_pbds;

typedef tree< long long, null_type, less_equal<long long>,
    rb_tree_tag, tree_order_statistics_node_update >
    ordered_set;
//change ll to any data type
//less_equal for multiset increasing order
//less for set increasing order
//greater_equal for multiset decreasing order
//greater for set decreasing order

//cout<<*X.find_by_order(1)<<endl; // iterator to the k-th
    largest element
//cout<<X.order_of_key(-5)<<endl; // number of items in a
    set that are strictly smaller than our item
```

```cpp
//Number theory related
const int MOD = 1e9+7;
int gcd ( int a, int b ) { return __gcd ( a, b ); }
int lcm ( int a, int b ) { return a * ( b / gcd ( a, b ) );
    }
inline void normal(int &a) { a %= MOD; (a < 0) && (a += MOD)
    ; }
inline int modMul(int a, int b) { a %= MOD, b %= MOD; normal
    (a), normal(b); return (a*b)%MOD; }
inline int modAdd(int a, int b) { a %= MOD, b %= MOD; normal
    (a), normal(b); return (a+b)%MOD; }
inline int modSub(int a, int b) { a %= MOD, b %= MOD; normal
    (a), normal(b); a -= b; normal(a); return a; }
inline int modPow(int b, int p) { int r = 1; while(p) { if(p
    &1) r = modMul(r, b); b = modMul(b, b); p >>= 1; }
    return r; }
inline int modInverse(int a) { return modPow(a, MOD-2); }
inline int modDiv(int a, int b) { return modMul(a,
    modInverse(b)); }
```

4 Number Theory

4.1 Binary Exponentiation

```cpp
/* Time complexity: O(power)
 * Step 1: binary_expo<int>(base, power, mod); */
template<typename T, typename X>
T binary_expo (T val, T power, X m) {
  T output = 1;
  while (power) {
    if (power & 1) {
      output = T((output * 1LL * val) % m);
    }
    val = (val * 1LL * val) % m;
    power >>= 1;
  }
  return output;
}
```

4.2 Binomial Coefficients nCr

```cpp
/* Time complexity: log(MOD - 2) */
/* Step 1: bin_coeff<int> bcoef(MAXN, MOD); */
/* Step 2: bcoef.nCr(n, r); */
template <typename T>
struct bin_coeff {
  T n, m;
  vector<T> fact;
  void gen_fact () {
    fact[0] = fact[1] = 1;
    for (int i = 2; i <= n; ++i) {
      fact[i] = (1LL * fact[i - 1] * i) % m;
    }
  }
  bin_coeff (T n, T m) {
    this -> n = n;
    this -> m = m;
    fact.resize(n + 1);
    gen_fact();
  }
  T inv (T val, T power) {
    T output = 1;
    while (power) {
      if (power & 1) {
        output = T((output * 1LL * val) % m);
      }
      val = (val * 1LL * val) % m;
      power >>= 1;
    }
    return output;
  }
  T nCr (T N, T R) {
    return (fact[N] * 1LL * inv((fact[R] * 1LL * fact[N - R])
        % m, m - 2)) % m;
  }
};
```

4.3 Catalan Number

```cpp
long long catalan[n + 1];

// Initialize first two values in table
catalan[0] = catalan[1] = 1;

// Fill entries in catalan[] using recursive formula
for (int i = 2; i <= n; i++) {
  catalan[i] = 0;
  for (int j = 0; j < i; j++) {
    catalan[i] += catalan[j] * catalan[i - j - 1];
  }
}
```

4.4 Chinese Remainder Theorem

```cpp
struct Congruence {
    long long a, m;
};

long long chinese_remainder_theorem(vector<Congruence> const
    & congruences) {
  long long M = 1;
  for (auto const& congruence : congruences) {
    M *= congruence.m;
  }

  long long solution = 0;
  for (auto const& congruence : congruences) {
    long long a_i = congruence.a;
    long long M_i = M / congruence.m;
    long long N_i = mod_inv(M_i, congruence.m);
    solution = (solution + a_i * M_i % M * N_i) % M;
  }
  return solution;
}
```

4.5 Euler Totient (precomputation)

```cpp
/* Time complexity: O(n*log*log(n)) */
void phi_1_to_n(int n) {
  vector<int> phi(n + 1);
  for (int i = 0; i <= n; i++) {
    phi[i] = i;
  }
  for (int i = 2; i <= n; i++) {
    if (phi[i] == i) {
      for (int j = i; j <= n; j += i) {
        phi[j] -= phi[j] / i;
      }
    }
  }
}
```

4.6 Euler Totient (single)

```cpp
/* Time complexity: O(sqrt(n))
 * Returns phi(n) */
int phi (int n) {
  int result = n;
  for (int i = 2; i * i <= n; i++) {
    if (n % i == 0) {
```

```
      while (n % i == 0) {
        n /= i;
      }
      result -= result / i;
    }
  }
  if (n > 1) {
    result -= result / n;
  }
  return result;
}
```

4.7 Extended GCD

```
int gcd(int a, int b, int& x, int& y) {
  if (b == 0) {
      x = 1;
      y = 0;
      return a;
  }
  int x1, y1;
  int d = gcd(b, a % b, x1, y1);
  x = y1;
  y = x1 - y1 * (a / b);
  return d;
}
```

4.8 Fermats Primality test

```
/* Time complexity: O(k*log(n))
 * Where, (k) is number of iterations and
 * (n) is the number to check for primality */
template <typename T>
bool fermat (T n, int iter=5) {
  if (n < 4) {
    return n == 2 or n == 3;
  }
  for (int i = 0; i < iter; i++) {
    T a = 2 + rand() % (n - 3);
    if (binary_expo<T>(a, n - 1, n) != 1) {
      return false;
    }
  }
  return true;
}
```

4.9 Matrix Exponentiation

```
// C++ program to find value of f(n) where f(n)
// is defined as,
//    F(n) = F(n-1) + F(n-2) + F(n-3), n >= 3
// Base Cases :
//    F(0) = 0, F(1) = 1, F(2) = 1
// Time Complexity: O(logN)
// Step 1: findNthTerm(n)

// A utility function to multiply two matrices
// a[][] and b[][]. Multiplication result is
// stored back in b[][]
void multiply(int a[3][3], int b[3][3])
{
    // Creating an auxiliary matrix to store elements
    // of the multiplication matrix
    int mul[3][3];
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 3; j++)
        {
            mul[i][j] = 0;
            for (int k = 0; k < 3; k++)
                mul[i][j] += a[i][k]*b[k][j];
        }
    }

    // storing the multiplication result in a[][]
    for (int i=0; i<3; i++)
        for (int j=0; j<3; j++)
            a[i][j] = mul[i][j]; // Updating our matrix
}

// Function to compute F raise to power n-2.
int power(int F[3][3], int n)
{
    int M[3][3] = {{1,1,1}, {1,0,0}, {0,1,0}};

    // Multiply it with initial values i.e with
    // F(0) = 0, F(1) = 1, F(2) = 1
    if (n==1)
        return F[0][0] + F[0][1];

    power(F, n/2);

    multiply(F, F);

    if (n%2 != 0)
        multiply(F, M);

    // Multiply it with initial values i.e with
    // F(0) = 0, F(1) = 1, F(2) = 1
    return F[0][0] + F[0][1] ;
}

// Return n'th term of a series defined using below
// recurrence relation.
// f(n) is defined as
//    f(n) = f(n-1) + f(n-2) + f(n-3), n>=3
// Base Cases :
//    f(0) = 0, f(1) = 1, f(2) = 1
int findNthTerm(int n)
{

    int F[3][3] = {{1,1,1}, {1,0,0}, {0,1,0}} ;

    //Base cases
    if(n==0)
        return 0;
    if(n==1 || n==2)
        return 1;

    return power(F, n-2);
}
```

4.10 Mobius Function

```
for (i = 0; i < MU_MAX;i++) {
  mu[i] = 1;
}

for (i = 2; i <= sqroot; i++) {
  if (mu[i] == 1) {
    // for each factor found, swap (+) and (-)
    for (j = i; j <= MU_MAX; j += i) {
      mu[j] *= (-1LL);
    }
    // square factor = 0
    for (j = i * i; j <= MU_MAX; j += i * i) {
      mu[j] = 0;
    }
  }
}
```

4.11 Trivial nCr

```cpp
/* Used when there's no mod used
 * Time complexity: O(k)
 * Step 1: ncr_triv<int>(n, r) */
template <typename T>
T ncr_triv (T n, T k) {
  T ncr = 1;
  if (n - k < k) {
    k = n - k;
  }
  for (T i = 0; i < k; ++i) {
    ncr *= (n - i);
    ncr /= (i + 1);
  }
  return ncr;
}
```

4.12 nCr Table

```cpp
long long ncr[maxn][maxn] = {0};
const int mod = (int) 1e9 + 7;
void init () {
  ncr[0][0] = 1;
  for (int i = 1; i < maxn; i++) {
    ncr[i][0] = 1;
    for (int j = 1; j < i + 1; j++) {
      ncr[i][j] = (ncr[i - 1][j - 1] + ncr[i - 1][j]) % mod;
    }
  }
}
```

5 Strings

5.1 Aho Corasick

```cpp
const int K = 26;

struct Vertex {
  int next[K];
  bool leaf = false;
  int p = -1;
  char pch;
  int link = -1;
  int go[K];

  Vertex(int p=-1, char ch='$') : p(p), pch(ch) {
    fill(begin(next), end(next), -1);
    fill(begin(go), end(go), -1);
  }
};

vector<Vertex> t(1);

void add_string(string const& s) {
  int v = 0;
  for (char ch : s) {
    int c = ch - 'a';
    if (t[v].next[c] == -1) {
      t[v].next[c] = t.size();
      t.emplace_back(v, ch);
    }
    v = t[v].next[c];
  }
  t[v].leaf = true;
}

int go(int v, char ch);

int get_link(int v) {
  if (t[v].link == -1) {
    if (v == 0 || t[v].p == 0) {
      t[v].link = 0;
    } else {
      t[v].link = go(get_link(t[v].p), t[v].pch);
    }
  }
  return t[v].link;
}

int go(int v, char ch) {
  int c = ch - 'a';
  if (t[v].go[c] == -1) {
    if (t[v].next[c] != -1) {
      t[v].go[c] = t[v].next[c];
    } else {
      t[v].go[c] = v == 0 ? 0 : go(get_link(v), ch);
    }
  }
  return t[v].go[c];
}
```

5.2 Knuth Morris Pratt

```cpp
/* Total complexity: O(n + m) */
/* Application (string problems) : */
/*  1. Used to extract matched positions */
/*  2. Used to know if we have a match or not */
/* Here, tab[j] denotes the length of the prefix which
      mathces with the suffix corresponding to index (j) */
/* Step 1: kmp<int> km(full_string, pattern_to_search_for);
    */
/* Step 2: auto ids = km.pos(); */
template <typename T>
struct kmp {
  int n, m;
  string s, t;
  vector<T> tab;
  // Creating the prefix length table
  void proc () {
    int i = 0;
    for (int j = 1; j < m;) {
      if (t[i] == t[j]) {
        tab[j] = i + 1;
        i += 1, j += 1;
      } else {
        if (i) {
          i = tab[i - 1];
        } else {
          j += 1;
        }
      }
    }
  }
  // initializing everything
  kmp (string s, string t) {
    this -> s = s;
    this -> t = t;
    n = (T) s.size();
    m = (T) t.size();
    tab.assign(m, 0);
    proc();
  }
  // Returns all the starting positions where we have a
      match
  // If we have a match we continue,
  // Otherwise, we look in the previous index of the table
      to save time.
  vector<T> pos () {
    int i = 0;
    int j = 0;
    vector<T> ids;
    while (i < n) {
      if (s[i] == t[j]) {
        i += 1, j += 1;
      } else {
```

```
      if (j) {
        j = tab[j - 1];
      } else {
        i += 1;
      }
    }
    // If pattern found take the index
    if (j == m) {
      ids.push_back(i - m);
      j = tab[j - 1];
    }
  }
  return ids;
 }
};
```

5.3 Manachers

```
/* Time complexity: O(N) */
/* While solving this problem always try to solve this on
    the basis of the generated answer it is returning */
/* Which means, always try to solve on the basis of
    converted string -> #a#b#a# */
/* Return the length of a palindrome from left side,
    defining (i) as the middle of that palindrome*/
/* manachers<int> man(s); */
/* auto ans = man.ret_ans(); */
template <typename T>
struct manachers {
  int n;
  vector<int> p;
  void manac_odd (string s) {
    n = (int) s.size();
    s = "(" + s + ")";
    p.assign(n + 2, 0);
    int l = 1, r = 1;
    for (int i = 1; i <= n; ++i) {
      p[i] = max(0, min(r - i, p[l + (r - i)]));
      while (s[i - p[i]] == s[i + p[i]]) {
        p[i] += 1;
      }
      if (r < i + p[i]) {
        l = i - p[i];
        r = i + p[i];
      }
    }
  }
  manachers (string t) {
    string s = "";
    for (auto c : t) {
      s += string("#") + c;
    }
    manac_odd(s + "#");
  }
  vector<T> ret_ans () {
    return vector<T>(p.begin() + 1, p.end() - 1);
  }
};
```

5.4 String Hashing (double)

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

const int mod = (int) 1e9 + 7;
int add_mod (int a, int b) {
  int res = (a + b) % mod;
  res += (res < 0 ? mod : 0);
  return res;
}

int sub_mod (int a, int b) {
  int res = (a - b) % mod;
  res += (res < 0 ? mod : 0);
  return res;
}

int mult_mod (int a, int b) {
  int res = (a * 1LL * b) % mod;
  res += (res < 0 ? mod : 0);
  return res;
}

template<typename T, typename X>
T binary_expo (T val, T power, X m) {
  T output = 1;
  while (power) {
    if (power & 1) {
      output = T((output * 1LL * val) % m);
    }
    val = (val * 1LL * val) % m;
    power >>= 1;
  }
  return output;
}

int main () {
  /* ios::sync_with_stdio(false); */
/* cin.tie(0); */

string s;
cin >> s;

/* This block of code completely double hashes the string
     S */
int p1 = 31, p2 = 53;
int n = (int) s.size();
vector<int> pref_hash1(n);
vector<int> pref_hash2(n);
pref_hash1[0] = (s[0] - 'a') + 1;
pref_hash2[0] = (s[0] - 'a') + 1;
/* The inverse array is needed to substract the substring'
     s hash */
vector<int> p_pow1(n), inv1(n);
vector<int> p_pow2(n), inv2(n);
p_pow1[0] = inv1[0] = 1;
p_pow2[0] = inv2[0] = 1;
for (int i = 1; i < n; i++) {
  p_pow1[i] = (p_pow1[i - 1] * 1LL * p1) % mod;
  p_pow2[i] = (p_pow2[i - 1] * 1LL * p2) % mod;
  inv1[i] = binary_expo<int>(p_pow1[i], mod - 2, mod);
  inv2[i] = binary_expo<int>(p_pow2[i], mod - 2, mod);
  pref_hash1[i] = add_mod(pref_hash1[i - 1], mult_mod((s[i]
       - 'a' + 1), p_pow1[i]));
  pref_hash2[i] = add_mod(pref_hash2[i - 1], mult_mod((s[i]
       - 'a' + 1), p_pow2[i]));
}

/* This function returns the hash-1 of the substring of
     string s
 * Moreover, this function also uses 0 based indesing */
auto substring_hash1 = [&] (int l, int r) {
  int res = pref_hash1[r];
  if (0 < l) {
    res -= pref_hash1[l - 1];
  }
  res = mult_mod(res, inv1[l]);
  return res;
};
/* This function returns the hash-1 of the substring of
     string s
 * Moreover, this function also uses 0 based indesing */
auto substring_hash2 = [&] (int l, int r) {
  int res = pref_hash2[r];
  if (0 < l) {
    res -= pref_hash2[l - 1];
  }
  res = mult_mod(res, inv2[l]);
```

```cpp
    return res;
  };

  /* This block of code quering for each substring hash*/
  int q;
  cin >> q;
  while (q--) {
    int l, r;
    cin >> l >> r;
    --l, --r;
    cout << substring_hash1(l, r) << '\n';
    cout << substring_hash2(l, r) << '\n';
  }
  return 0;
}
```

5.5 Trie

```cpp
/* Time complexity (construction): */
/*   * number of nodes. Which depends on matched prefix. */
/*   * the more prefix-match, the better. */
/* Time complexity (per query): */
/*   * length of the asked string */
/* Step 1: trie tri; */
/* Step 2: tri.insert(s); */
/* Step 3: cout << (tri.search(t) ? "YES\n" : "NO\n") << '\n
    '; */
/* Step 4: tri.del(); */
/* Note: useful for searching a string is present or not */
struct trie {
  struct node {
    bool endmark;
 /* Change the size to (10) if working with digits */
    node* next[26];
    node () {
      endmark = false;
   /* Change the limit to 10 if working with digits */
      for (int i = 0; i < 26; ++i) {
        next[i] = NULL;
      }
    }
  } * root;
  /* trie tri; */
  trie () {
    root = new node();
  }
  /* tri.insert(s); */
  /* inserts a string the the trie */
  void insert (string s) {
    node* curr = root;
    for (auto ch : s) {
      /* change 'a' according to the problem statement */
      int id = ch - 'a';
      if (curr -> next[id] == NULL) {
        curr -> next[id] = new node();
      }
      curr = curr -> next[id];
    }
    curr -> endmark = true;
  }
  /* return if a string is present in the list or not */
  /* cout << (tri.search(t) ? "YES\n" : "NO\n") << '\n'; */
  bool search (string s) {
    node* curr = root;
    for (auto ch : s) {
      /* change 'a' according to the problem statement */
      int id = ch - 'a';
      if (curr -> next[id] == NULL) {
        return false;
      }
      curr = curr -> next[id];
    }
    return curr -> endmark;
  }
  void del_node (node* curr) {
 /* Change the limit to 10 if working with digits */
    for (int i = 0; i < 26; ++i) {
      if (curr -> next[i]) {
        del_node(curr -> next[i]);
      }
    }
    delete(curr);
  }
  /* tri.del(); */
  /* deletes, all the nodes. Useful in reducing memory */
  void del () {
    del_node(root);
  }
};
```