Uzhhorod National University

TeamName

p1, p2, p3

2023-09-27

Contest (1)

template.cpp

113 lines

```
#ifdef __APPLE__
#define _GLIBCXX_DEBUG
#define _GLIBCXX_DEBUG_PEDANTIC
#else
#pragma comment(linker, "/stack:200000000")
#pragma GCC optimize("Ofast")
#pragma GCC optimize("O3,unroll-loops")
#pragma GCC target("sse,sse2,sse3,ssse3,sse4")
#pragma GCC target("avx2,bmi,bmi2,popcnt,lzcnt")
#endif

#include <iostream>
#include <vector>
#include <algorithm>
#include <map>
#include <set>
#include <queue>
#include <deque>
#include <cmath>
#include <climits>

#ifdef __APPLE__
template <typename T>
class ordered_set : public std::set<T> {
public:
    auto find_by_order(size_t order) const {
        auto cur = this->begin();
        while (order--) {
            cur++;
        }
        return cur;
    }

    int order_of_key(const T &key) const {
        int cnt = 0;
        for (auto it = this->begin(); it != this->begin(); it
            ++, cnt++) {
            if (*it == key)
                return cnt;
        }
        return cnt;
    }
};
#else
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
template <typename T>
using ordered_set = tree<T, null_type, std::less<T>,
    rb_tree_tag, tree_order_statistics_node_update>;
#endif
using namespace std;

const int MOD = 998244353;
using ll = long long;
const ll INF = 1e18;

// #define int ll

template <typename T>
using graph = vector<vector<T>>;

template <typename T>
istream &operator>>(istream &in, vector<T> &a) {
    for (auto &i : a) {
        in >> i;
    }
    return in;
}

template <typename T>
ostream &operator<<(ostream &out, vector<T> &a) {
    for (auto &i : a) {
        out << i << " ";
    }
    return out;
}

int fast_pow(int a, int b, int mod) {
    if (b == 0)
        return 1;
    if (b % 2) {
        return (a * fast_pow(a, b - 1, mod)) % mod;
    }
    int k = fast_pow(a, b / 2, mod);
    return (k * k) % mod;
}

int fast_pow(int a, int b) {
    if (b == 0)
        return 1;
    if (b % 2) {
        return (a * fast_pow(a, b - 1));
    }
    int k = fast_pow(a, b / 2);
    return (k * k);
}

void solve() {

}

int32_t main(int32_t argc, const char *argv[]) {
    cin.tie(0);
    cout.tie(0);
    ios_base::sync_with_stdio(0);
    // insert code here...
    int tt = 1;
    // std::cin >> tt;
    while (tt--) {
        solve();
    }
    return 0;
}
```

.bashrc

3 lines

```
alias c='g++ -Wall -Wconversion -Wfatal-errors -g -std=c++20 \
  -fsanitize=undefined,address'
xmodmap -e 'clear lock' -e 'keycode 66=less greater' #caps = <>
```

.vimrc

6 lines

```
set cin aw ai is ts=4 sw=4 tm=50 nu noeb bg=dark ru cul
sy on   |   im jk <esc>   |   im kj <esc>   |   no ; :
" Select region and then type :Hash to hash your selection.
" Useful for verifying that there aren't mistypes.
ca Hash w !cpp -dD -P -fpreprocessed \| tr -d '[:space:]' \
 \| md5sum \| cut -c-6
```

hash.sh

3 lines

```
# Hashes a file, ignoring all whitespace and comments. Use for
# verifying that code was correctly typed.
cpp -dD -P -fpreprocessed | tr -d '[:space:]'| md5sum |cut -c-6
```

Algebra (2)

XorBasis.h

Description: Xor basis, all elements in the main set can be constructed using xor operation and elements in the basis
Time: insert per element - $\mathcal{O}(log(A_max))$

hash16 lines

```
array<int, 61> basis; // basis[i] -> element with smallest set
     bit equal to i
int sz; // Current size of the basis

bool insert_vector(int mask) {
  for (int i = 0; i <= 60; i++) {
    if ((mask & (1ll << i)) == 0)
      continue;
    if (!basis[i]) {
      basis[i] = mask;
      sz++;
      return true;
    }
    mask ^= basis[i];
  }
  return false;
}
```

FFT.h

Description: FFT implementation
Time: (n + m) * log(n + m)

hash57 lines

```
vector<complex<double>> fft(vector<complex<double>> a, bool
     inverse = true) {
    int n = a.size();
    if (n == 1) {
        return {a[0]};
    }

    int k = n / 2;
    vector<complex<double>> par(k), nepar(k);
    for (int i = 0; i < n; i++) {
        if (i % 2 == 0)
            par[i / 2] = a[i];
        else
            nepar[i / 2] = a[i];
    }

    par = fft(par, inverse);
    nepar = fft(nepar, inverse);
    vector<complex<double>> ans(n);
    for (int i = 0; i < n; i++) {
        auto x = complex(cos(2 * i * PI / n), sin(2 * i * PI /
             n));
        if (inverse)
            x = complex(cos(2 * i * PI / n), sin(-2 * i * PI /
                 n));
```

```
        ans[i] = par[i % k] + nepar[i % k] * x;
    }
    return ans;
}

vector<int> mult(vector<int> a, vector<int> b) {
    int n = a.size();
    int m = b.size();
    int k = n + m - 1;

    int l = 1;
    while (l < k)
    {
        l *= 2;
    }
    vector<complex<double>> A(l), B(l);
    for (int i = 0; i < n; i++)
        A[i] = a[i];
    for (int j = 0; j < m; j++)
        B[j] = b[j];
    A = fft(A, 0);
    B = fft(B, 0);

    for (int i = 0; i < l; i++) {
        A[i] *= B[i];
    }
    A = fft(A, 1);

    vector<int> ans(l);
    for (int i = 0; i < l; i++) {
        A[i] /= l;
        ans[i] = int(A[i].real() + 0.5);
    }
    return ans;
}
```

NNT.h

Description: NNT implementation by modulo 998244353
Time: (n + m) * log(n + m)

hash67 lines

```
const int root = 31;
const int root_1 = fast_pow(root, MOD - 2, MOD);
const int root_pw = 1 << 23;

inline int inverse(int n, int mod) {
  return fast_pow(n, mod - 2, mod);
}

inline void nnt(vector<int> &a, bool invert, int mod) {
    int n = a.size();

    for (int i = 1, j = 0; i < n; i++) {
        int bit = n >> 1;
        for (; j & bit; bit >>= 1)
            j ^= bit;
        j ^= bit;

        if (i < j)
            swap(a[i], a[j]);
    }

    for (int len = 2; len <= n; len <<= 1) {
        int wlen = invert ? root_1 : root;
        for (int i = len; i < root_pw; i <<= 1)
            wlen = (int)(1LL * wlen * wlen % mod);

        for (int i = 0; i < n; i += len) {
            int w = 1;
            for (int j = 0; j < len / 2; j++) {
                int u = a[i+j], v = (int)(1LL * a[i+j+len/2] *
                    w % mod);
                a[i+j] = u + v < mod ? u + v : u + v - mod;
                a[i+j+len/2] = u - v >= 0 ? u - v : u - v + mod
                    ;
                w = (int)(1LL * w * wlen % mod);
            }
        }
    }

    if (invert) {
        int n_1 = inverse(n, mod);
        for (int & x : a)
            x = (int)(1LL * x * n_1 % mod);
    }
}

inline vector<int> multiply(vector<int> left, vector<int> right
    ) {
  int n = left.size(), m = right.size();
  int k = n + m - 1;
  int l = 1;
  while (l < k)
    l <<= 1;

  vector<int> A(l), B(l);
  for (int i = 0; i < n; i++) {
    A[i] = left[i];
  }
  for (int i = 0; i < m; i++) {
    B[i] = right[i];
  }

  nnt(A, false, MOD);
  nnt(B, false, MOD);
  for (int i = 0; i < B.size(); i++) {
    A[i] = (1ll * A[i] * B[i]) % MOD;
  }
  nnt(A, true, MOD);
  return A;
}
```

Data structures (3)

SegmentTree.h

Description: Segment tree from below.
Time: update - $\mathcal{O}(\log N)$, get - $\mathcal{O}(\log N)$

hash36 lines

```
struct SegmentTree {
  static const int N = (1 << 20);
  array<int, N> tree;
  SegmentTree() {
    tree.fill(0);
  }

  void update(int pos, int val) {
    pos += N;
    tree[pos] = val;
    pos >>= 1;
    while (pos > 0) {
      tree[pos] = tree[pos << 1] + tree[(pos << 1) | 1];
      pos >>= 1;
    }
  }

  int get_sum(int l, int r) {
    l += N;
    r += N;
    int ans = 0;
    while (l < r) {
      if (l & 1) {
        ans += tree[l++];
      }
      if ((r & 1) == 0) {
        ans += tree[r--];
      }

      l >>= 1;
      r >>= 1;
    }
    return ans;
  }
};
```

SegmentTreeWithPromises.h

Description: Zero-indexed sum-tree with update on segment. Bounds are inclusive to the left and to the right.
Time: update - $\mathcal{O}(\log N)$, get - $\mathcal{O}(\log N)$

<vector>, <iostream>, <array> hash65 lines

```
struct segment_tree {
  static const int N = 1e5 + 100;
  static const int NONE = -1;

  struct node{
    int mn = INT_MAX;
    int mx = INT_MIN;
    int promise = NONE;
  };
  array<node, 4 * N> tree;

  void update(int l, int r, int val) {
    update(1, 0, N - 1, l, r, val);
  }

  node get(int l, int r) {
    return get(1, 0, N - 1, l, r);
  }

  void push(int v, int l, int r) {
    if (tree[v].promise == NONE)
      return;
    tree[v].mn = tree[v].mx = tree[v].promise;
    if (l != r) {
      tree[2 * v].promise = tree[v].promise;
      tree[2 * v + 1].promise = tree[v].promise;
    }
    tree[v].promise = NONE;
  }

  void update(int v, int tl, int tr, int l, int r, int value) {
    push(v, tl, tr);
    if (l > tr || tl > r)
      return;
    if (l <= tl && tr <= r) {
      tree[v].promise = value;
      push(v, tl, tr);
      return;
    }
    int mid = (tl + tr) / 2;
    update(2 * v, tl, mid, l, r, value);
    update(2 * v + 1, mid + 1, tr, l, r, value);

    tree[v].mx = max(tree[2 * v].mx, tree[2 * v + 1].mx);
    tree[v].mn = min(tree[2 * v].mn, tree[2 * v + 1].mn);
  }
```

```
  node get(int v, int tl, int tr, int l, int r) {
    push(v, tl, tr);
    if (l > tr || tl > r)
      return node();
    if (l <= tl && tr <= r) {
      return tree[v];
    }
    int mid = (tl + tr) / 2;
    auto left = get(2 * v, tl, mid, l, r);
    auto right = get(2 * v + 1, mid + 1, tr, l, r);

    return node {
      .mn = min(left.mn, right.mn),
      .mx = max(left.mx, right.mx),
      .promise = NONE
    };
  }
};
```

LiChaoTree.h

Description: Li-Chao tree, online convex hull for maximizing f(x) = k * x + b

Time: add - $\mathcal{O}(\log N)$, get - $\mathcal{O}(\log N)$ hash76 lines

```
struct line
{
  int k = 0;
  int b = -INF;

    int f(int x) const {
        return k * x + b;
    }
};

struct li_chao_tree {
  const int MX = 1e9 + 100;
  struct node
  {
    line ln;
    node* left = nullptr;
    node* right = nullptr;
  };

  node* new_node() {
    const int N = 100000;
    static node* block;
    static int count = N;

    if (count == N) {
      block= new node[N];
      count = 0;
    }
    return (block + count++);
  };

  node* root = new_node();

  int get(int x) {
    return get(root, 0, MX, x);
  }
    void add(line ln) {
        return add(root, 0, MX, ln);
    }

  int get(node*& v, int l, int r, int x) {
    if (!v) {
      return -INF;
    }
    int ans = v->ln.f(x);
    if (r == l) {
      return ans;
    }
    int mid = (r + l) / 2;
    if (x <= mid) {
      return max(ans, get(v->left, l, mid, x));
    } else {
          return max(ans, get(v->right, mid + 1, r, x));
      }
  }

    void add(node*& v, int l, int r, line ln) {
        if (!v) {
            v = new_node();
        }
        int m = (r + l) / 2;
        bool left = v->ln.f(l) < ln.f(l);
    bool md = v->ln.f(m) < ln.f(m);
        if (md)
            swap(v->ln, ln);
        if (l == r) {
            return;
        }
        if (left != md) {
            add(v->left, l, m, ln);
        } else {
            add(v->right, m + 1, r, ln);
        }
    }
};
```

Graphs (4)

Dinitz.h

Description: Dinitz algorithm, finds max flow in network

Time: $\mathcal{O}(V^2 * E)$ hash84 lines

```
struct Edge {
  int from, to;
  int cap, flow;
  Edge(int from_, int to_, int cap_): from(from_), to(to_), cap
      (cap_), flow(0) {}

  int other(int v) const {
    if (v == from)
      return to;
    return from;
  }
  int capacity(int v) const {
    if (v == from)
      return (cap - flow);
    return flow;
  }
  void add(int df, int v) {
    if (v == from) {
      flow += df;
    } else {
      flow -= df;
    }
  }
};

vector<int> dinitz_bfs(int v, const graph<Edge*>& g) {
  int n = g.size();
  vector<int> dist(n, n + 100);
  dist[v] = 0;
  queue<int> q;
  q.push(v);
  while (!q.empty()) {
    int v = q.front();
    q.pop();

    for (auto& e : g[v]) {
      int to = e->other(v);
      if (!e->capacity(v))
        continue;
      if (dist[to] > n) {
        dist[to] = dist[v] + 1;
        q.push(to);
      }
    }
  }
  return dist;
}
vector<bool> blocked;
vector<int> dist;
int dinitz_dfs(int v, int F, graph<Edge*>& g, int t) {
  if (v == t || F == 0)
    return F;
  bool all_blocked = true;
  int pushed = 0;
  for (auto& e : g[v]) {
    int to = e->other(v);
    if (dist[to] != dist[v] + 1)
      continue;

    if (e->capacity(v) && !blocked[to]) {
      int df = dinitz_dfs(to, min(F, e->capacity(v)), g, t);
      e->add(df, v);
      pushed += df;
      F -= df;
    }

    if (!blocked[to] && e->capacity(v))
      all_blocked = false;
  }

  if (all_blocked)
    blocked[v] = true;
  return pushed;
}

while (true) {
  dist = dinitz_bfs(s, g);

  if (dist[t] > dist.size())
    break;

  blocked.assign(dist.size(), false);
  dinitz_dfs(s, INF, g, t);
}
```

Kuhn.h

Description: Fast pair matching algorithm

Time: $\mathcal{O}(n * (n + m))$ hash38 lines

```
graph<int> g;
vector<int> mt, used, rev_mt;
bool dfs(int v) {
    if (used[v] == 1)
        return false;
    used[v] = 1;
    for (auto to : g[v]) {
        if (mt[to] == -1) {
            rev_mt[v] = to;
            mt[to] = v;
```

```
            return true;
        }
    }
    for (auto to : g[v]) {
        if (dfs(mt[to])) {
            rev_mt[v] = to;
            mt[to] = v;
            return true;
        }
    }
    return false;
}

void pair_matching() {
    for (int it = 0; ; it++) {
        bool finded = false;
        used.assign(n, 0);
        for (int i = 0; i < n; i++) {
            if (rev_mt[i] == -1 && dfs(i)) {
                cnt++;
                finded = true;
            }
        }
        if (!finded) {
            break;
        }
    }
}
```

TwoSat.h

Description: 2-sat implementation
Time: $\mathcal{O}(n)$ hash71 lines

```
struct TwoSat {
    graph<int> g, rev;
    vector<int> used, order, comp, ans;
    int n;

    TwoSat(int _n) {
        this->n = _n;
        g.assign(2 * n, {});
        rev.assign(2 * n, {});
    }
    void add_edge(int u, int v) {
        g[u].push_back(v);
        rev[v].push_back(u);
    }

    void add_clause_or(int a, bool val_a, int b, bool val_b) {
        add_edge(a + val_a * n, b + !val_b * n);
        add_edge(b + val_b * n, a + !val_a * n);
    }

    void add_clause_xor(int a, bool val_a, int b, bool val_b) {
        add_clause_or(a, val_a, b, val_b);
        add_clause_or(a, !val_a, b, !val_b);
    }

    void add_clause_and(int a, bool val_a, int b, bool val_b) {
        add_clause_xor(a, !val_a, b, val_b);
    }

    void top_sort(int v) {
        used[v] = 1;
        for (auto to : g[v]) {
            if (!used[to])
                top_sort(to);
        }
        order.push_back(v);
    }

    void compress(int v, int id) {
        comp[v] = id;
        for (auto to : rev[v]) {
            if (comp[to] == -1)
                compress(to, id);
        }
    }

    bool satisfiable() {
        order.clear();
        used.assign(2 * n, 0);
        comp.assign(2 * n, -1);
        ans.assign(n, 0);

        for (int i = 0; i < 2 * n; i++) {
            if (!used[i])
                top_sort(i);
        }
        reverse(order.begin(), order.end());
        int id = 0;
        for (auto v : order) {
            if (comp[v] == -1)
                compress(v, id++);
        }

        for (int i = 0; i < n; i++) {
            if (comp[i] == comp[i + n])
                return false;
            ans[i] = (comp[i + n] < comp[i]);
        }
        return true;
    }
};
```

Strings (5)

ZFunction.h

Description: Z-functions, z[i] equal to the length of largest common prefix of string s and suffix of s starting at i.
Time: $\mathcal{O}(N)$, N - size of string s hash16 lines

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

SuffixArray.h

Description: Suffix array will contain integers that represent the starting indexes of the all the suffixes of a given string, after the aforementioned suffixes are sorted.
Time: $\mathcal{O}(N * log2(N))$, N - size of string s hash69 lines

```
vector<int> suffix_arrays(string s) {
  s = s + "$";
  int n = s.size();

  vector<int> p(n);
  vector<vector<int>> c(20, vector<int>(n));
  int alphabet = 256;

  auto set_classes = [&](int k) {
    int classes = 0;
    c[k][p[0]] = classes++;
    for (int i = 1; i < n; i++) {
      auto cur = pair{c[k - 1][p[i]], c[k - 1][(p[i] + (1<<(k
          -1))) % n]};
      auto prev = pair{c[k - 1][p[i - 1]], c[k - 1][(p[i - 1] +
          (1<<(k-1))) % n]};
      if (cur == prev) {
        c[k][p[i]] = c[k][p[i - 1]];
      } else {
        c[k][p[i]] = classes++;
      }
    }
  };

  auto init_base = [&]() {
    vector<int> cnt(alphabet);
    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 = n - 1; i >= 0; i--) {
      p[cnt[s[i]] - 1] = i;
      cnt[s[i]]--;

    }
    int classes = 0;
    c[0][p[0]] = classes++;
    for (int i = 1; i < n; i++) {
      if (s[p[i]] == s[p[i - 1]]) {
        c[0][p[i]] = c[0][p[i - 1]];
      } else {
        c[0][p[i]] = classes++;
      }
    }
  };

  init_base();

  for (int k = 0; (1<<k) < n; k++) {
    vector<int> pn(n), cnt(n);
    for (int i = 0; i < n; i++) {
      pn[i] = (p[i] - (1<<k) + n) % n;
      cnt[c[k][pn[i]]]++;
    }

    for (int i = 1; i < n; i++)
      cnt[i] += cnt[i - 1];

    for (int i = n - 1; i >= 0; i--) {
      p[cnt[c[k][pn[i]]] - 1] = pn[i];
      cnt[c[k][pn[i]]]--;
    }

    set_classes(k + 1);
  }

  p.erase(p.begin());
  return p;
}
```

Techniques (A)

techniques.txt
159 lines

```
Recursion
Divide and conquer
  Finding interesting points in N log N
Algorithm analysis
  Master theorem
  Amortized time complexity
Greedy algorithm
  Scheduling
  Max contiguous subvector sum
  Invariants
  Huffman encoding
Graph theory
  Dynamic graphs (extra book-keeping)
  Breadth first search
  Depth first search
  * Normal trees / DFS trees
  Dijkstra's algorithm
  MST: Prim's algorithm
  Bellman-Ford
  Konig's theorem and vertex cover
  Min-cost max flow
  Lovasz toggle
  Matrix tree theorem
  Maximal matching, general graphs
  Hopcroft-Karp
  Hall's marriage theorem
  Graphical sequences
  Floyd-Warshall
  Euler cycles
  Flow networks
  * Augmenting paths
  * Edmonds-Karp
  Bipartite matching
  Min. path cover
  Topological sorting
  Strongly connected components
  2-SAT
  Cut vertices, cut-edges and biconnected components
  Edge coloring
  * Trees
  Vertex coloring
  * Bipartite graphs (=> trees)
  * 3^n (special case of set cover)
  Diameter and centroid
  K'th shortest path
  Shortest cycle
Dynamic programming
  Knapsack
  Coin change
  Longest common subsequence
  Longest increasing subsequence
  Number of paths in a dag
  Shortest path in a dag
  Dynprog over intervals
  Dynprog over subsets
  Dynprog over probabilities
  Dynprog over trees
  3^n set cover
  Divide and conquer
  Knuth optimization
  Convex hull optimizations
  RMQ (sparse table a.k.a 2^k-jumps)
  Bitonic cycle
  Log partitioning (loop over most restricted)
Combinatorics
  Computation of binomial coefficients
  Pigeon-hole principle
  Inclusion/exclusion
  Catalan number
  Pick's theorem
Number theory
  Integer parts
  Divisibility
  Euclidean algorithm
  Modular arithmetic
  * Modular multiplication
  * Modular inverses
  * Modular exponentiation by squaring
  Chinese remainder theorem
  Fermat's little theorem
  Euler's theorem
  Phi function
  Frobenius number
  Quadratic reciprocity
  Pollard-Rho
  Miller-Rabin
  Hensel lifting
  Vieta root jumping
Game theory
  Combinatorial games
  Game trees
  Mini-max
  Nim
  Games on graphs
  Games on graphs with loops
  Grundy numbers
  Bipartite games without repetition
  General games without repetition
  Alpha-beta pruning
Probability theory
Optimization
  Binary search
  Ternary search
  Unimodality and convex functions
  Binary search on derivative
Numerical methods
  Numeric integration
  Newton's method
  Root-finding with binary/ternary search
  Golden section search
Matrices
  Gaussian elimination
  Exponentiation by squaring
Sorting
  Radix sort
Geometry
  Coordinates and vectors
  * Cross product
  * Scalar product
  Convex hull
  Polygon cut
  Closest pair
  Coordinate-compression
  Quadtrees
  KD-trees
  All segment-segment intersection
Sweeping
  Discretization (convert to events and sweep)
  Angle sweeping
  Line sweeping
  Discrete second derivatives
Strings
  Longest common substring
  Palindrome subsequences
  Knuth-Morris-Pratt
  Tries
  Rolling polynomial hashes
  Suffix array
  Suffix tree
  Aho-Corasick
  Manacher's algorithm
  Letter position lists
Combinatorial search
  Meet in the middle
  Brute-force with pruning
  Best-first (A*)
  Bidirectional search
  Iterative deepening DFS / A*
Data structures
  LCA (2^k-jumps in trees in general)
  Pull/push-technique on trees
  Heavy-light decomposition
  Centroid decomposition
  Lazy propagation
  Self-balancing trees
  Convex hull trick (wcipeg.com/wiki/Convex_hull_trick)
  Monotone queues / monotone stacks / sliding queues
  Sliding queue using 2 stacks
  Persistent segment tree
```