Uzhhorod National University

3yagoda

Vasyl Merenych, Dmytro Mayor, Roman Pitsura

2022-09-28

Contest (1)

template.cpp

88 lines

```cpp
// #pragma comment(linker, "/stack:200000000")
// #pragma GCC optimize("Ofast")
// #pragma GCC optimize("O3,unroll-loops")
// #pragma GCC target("sse,sse2,sse3,ssse3,sse4,avx2")

// #define _GLIBCXX_DEBUG
// #define _GLIBCXX_DEBUG_PEDANTIC

#include <iostream>
#include <vector>
#include <algorithm>
#include <map>
#include <set>
#include <queue>
#include <deque>
#include <cmath>
#include <climits>
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>

using namespace __gnu_pbds;
using namespace std;

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

// #define int ll

template<typename T>
using ordered_set = tree<T,null_type,less<T>,rb_tree_tag,
    tree_order_statistics_node_update>;

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
```

Data structures (2)

SegmentTreeNode.h

Description: Customizable segment tree node.
Time: update - $\mathcal{O}(\log N)$, get - $\mathcal{O}(\log N)$

ed0532, 116 lines

```cpp
template<class T, T default_value, T (*merge_value)(const T&,
    const T&), void (*update_value)(T&, const T&)>
class Node {
protected:
    using NodeT = Node<T, default_value, merge_value,
        update_value>;
    using ll = long long;

    Node* left = nullptr;
    Node* right = nullptr;

    T value;

    Node* my_copy() {
        Node* result = new NodeT();
        result->left = this->left;
        result->right = this->right;
        result->value = this->value;
        return result;
    }

    Node* get_left_node() {
        if(left == nullptr)
            left = new NodeT();
        return left;
    }

    Node* get_right_node() {
        if(right == nullptr)
            right = new NodeT();
        return right;
    }

    const T& get_left_val() const {
        if(left == nullptr)
            return default_value;
        return left->value;
    }

    const T& get_right_val() {
        if(right == nullptr)
            return default_value;
        return right->value;
    }

    void recalculate() {
        if(!left && !right) {
            this->value = default_value;
            return;
        }
        if(!left) {
            this->value = merge_value(default_value,
                get_right_node()->value);
            return;
        }
        if(!right) {
            this->value = merge_value(get_left_node()->value,
                default_value);
            return;
        }
        this->value = merge_value(get_left_node()->value,
            get_right_node()->value);
    }

    void upd_acreate_shadow(ll l, ll r, ll pos, T new_value) {
        if (l == r) {
            update_value(this->value, move(new_value));
            return;
        }
        ll m = (l + r) / 2;
        if(pos <= m) {
            if(this->left) {
                this->left = this->left->upd_acreate(l, m, pos,
                    move(new_value));
            } else {
                this->left = new NodeT();
                this->left->upd(l, m, pos, move(new_value));
            }
        } else {
            if(this->right) {
```

```
                this->right = this->right->upd_acreate(m + 1, r
                    , pos, move(new_value));
            } else {
                this->right = new NodeT();
                this->right->upd(m + 1, r, pos, move(new_value)
                    );
            }
        }
        recalculate();
    }

public:
    explicit Node(): value(default_value) {}

    void upd(ll l, ll r, ll pos, T new_value) {
        if(l == r) {
            update_value(this->value, move(new_value));
            return;
        }
        ll m = (l + r) / 2;
        (pos <= m ? get_left_node()->upd(l, m, pos, move(
            new_value)) : get_right_node()->upd(m + 1, r, pos,
             move(new_value)));
        recalculate();
    }

    Node* upd_acreate(ll l, ll r, ll pos, T new_value) {
        Node* result = my_copy();
        result->upd_acreate_shadow(l, r, pos, new_value);
        return result;
    }

    T get(ll l, ll r, ll L, ll R) {
        if(r < L || R < l)
            return default_value;
        if(L <= l && r <= R)
            return this->value;
        ll m = (l + r) / 2;

        if(!left && !right) return this->value;
        if(!left) return right->get(m + 1, r, L, R);
        if(!right) return left->get(l, m, L, R);

        return merge_value(get_left_node()->get(l, m, L, R),
            get_right_node()->get(m + 1, r, L, R));
    }
};
```

SegmentTree.h

Description: Customizable segment tree
Time: update - $\mathcal{O}(\log N)$, get - $\mathcal{O}(\log N)$

<SegmentTreeNode.h> 412028, 19 lines

```
template<class T, T default_value, T (*merge_value)(const T&,
    const T&), void (*update_value)(T&, const T&)>
class Tree {
private:
    ll l, r;
    Node<T, default_value, merge_value, update_value>* root =
        nullptr;
public:
    Tree(ll l, ll r): l(l), r(r), root(new Node<T,
        default_value, merge_value, update_value>()) {}

    void upd(ll pos, T value) {
        root->upd(l, r, pos, move(value));
    }

    T get(ll L, ll R) {
        return root->get(l, r, L, R);
    }
};

template<class T, T default_value>
using SumTree = Tree<T, default_value, sum<T>, add_value<T>>;
```

PersistentSegmentTree.h

Description: Persistent segment tree
Time: update - $\mathcal{O}(\log N)$, get - $\mathcal{O}(\log N)$

<SegmentTreeNode.h> 04748a, 22 lines

```
template<class T, T default_value, T (*merge_value)(const T&,
    const T&), void (*update_value)(T&, const T&)>
class PersistentTree {
private:
    ll l, r;
    vec<Node<T, default_value, merge_value, update_value>*>
        roots;
public:
    PersistentTree(ll l, ll r): l(l), r(r) {
        roots.push_back(new Node<T, default_value, merge_value,
            update_value>());
    }

    size_t update(ll pos, T value) {
        roots.push_back(roots[roots.size() - 1]->upd_acreate(l,
            r, pos, move(value)));
        return roots.size() - 1;
    }

    T get(ll tree_id, ll L, ll R) {
        return roots[tree_id]->get(l, r, L, R);
    }
};

template<class T, T default_value>
using PersistentSumTree = PersistentTree<T, default_value, max<
    T>, set_value<T>>;
```

Strings (3)

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

0c3dd4, 16 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

f6127b, 69 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 = 0; i < alphabet; 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
```