Acm Algorithm Templates

Epoche

Contents

1 Introduction 3
2 Planing 3
3 File Header 3
4 Graph 3
4.1 Storage 4
4.2 Traverse 4
4.3 Shortest path 4
5 Math 5
5.1 sieve prime number 5
5.2 Eluer function 5
5.3 Fast power 6
5.4 GCD 6
5.5 Matrix 6
6 Data Structure 6
6.1 Monotonic queue 7
6.2 Sparse Table (Rrange Maximum/Minimum Query) 7
6.3 Disjoint union 7
6.4 Fenwick Tree 8
6.5 Segment Tree 9
6.5.1 Classic 9
6.5.2 Lazy tag 10
7 Misc 11
7.1 Discrete 11
7.2 Leap year 12
7.3 STL 12
7.4 Sum of Prefix and Difference 12
7.5 Fast IO 12
7.6 CPU Information 13

1 Introduction

Personal Acm Templates

2 Planing

Problem Set https://www.luogu.com.cn/training/9391

- 矩阵
  1. 乘法 ✓
  2. 逆矩阵
- 树状数组
  1. 权值树状数组
  2. 单点修改，区间查询，区间修改，单点查询
- 线段树
- 图
  1. Bellman-ford
- 最小生成树 (Minimal spanning tree)
  1. Kruskal
  2. prim
- 素数筛
  1. 埃氏筛
  2. 欧拉筛

3 File Header

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

typedef long long i64;
```

Or

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <cmath>
#include <queue>
#include <stack>
#include <set>
#include <functional>

// #define rs ranges
#define fst first
#define snd second

typedef long long i64;
```

Optimize

```cpp
#pragma GCC optimize(2)
```

4 Graph

4.1 Storage

adjacency list

```cpp
struct Graph {
    int N;
    std::vector<std::vector<std::pair<int, int>>> adj;

    Graph(int n) {
        N = n;
        adj.resize(n);
    }

    void add(int u, int v, int d) {
        adj[u - 1].emplace_back(v - 1, d);
    }
};
```

4.2 Traverse

DFS

```cpp
void dfs(int u) {
    std::vector<int> vis(N, 1);

    auto recursive = [&](auto self, auto cur) -> void {
        vis[cur] = 0;

        for (auto [v, w] : adj[cur]) {
            if (vis[v] == 0) continue;
            self(self, v);
        }
    };

    recursive(recursive, u);
}
```

BFS

```cpp
void bfs(int origin) {
    std::vector<int> vis(N, 1);
    std::queue<int> q;
    q.emplace(origin);

    while (!q.empty()) {
        auto u = q.front();
        q.pop();
        vis[u] = 0;

        for (auto [v, _] : adj[u]) {
            if (vis[v] == 0) continue;
            q.emplace(v);
        }
    }
}
```

4.3 Shortest path

Heap optimized Dijkstra

Storage the minus edge weight, so that we don't need write the declaration: `std::greater<int>`.

```cpp
void dijkstra() {
    std::vector<int> dis(N, -1);
    std::priority_queue<std::pair<int, int>> h;
    h.emplace(0, 0);

    while (!h.empty()) {
        auto [d, u] = h.top();
        h.pop();

        if (dis[u] != -1) continue;
        dis[u] = -d;

        for (auto [v, w] : adj[u]) {
            h.emplace(d - w, v);
        }
    }
}
```

5 Math

5.1 sieve prime number

Liner sieve

```cpp
void sieveLiner (int n) {
    for (int i = 2; i <= n; i++) {
        if (st[i] == 0) pri[cnt++] = i;
        for (int j = 0; pri[j] <= n / i; j++) {
            st[pri[j] * i] = true;
            if (i % pri[j] == 0) break;
        }
    }
}
```

trial division determin prime number

```cpp
bool isPrime (int n) {
    if (n < 2) return false;
    for (int i = 2; i <= n / i; i++)
        if (n % i == 0) return false;
    return true;
}
```

5.2 Eluer function

```cpp
int euler_phi(int n) {
    int ans = n;
    for (int i = 2; i * i <= n; i++) {
        if (n % i == 0) {
            ans = ans / i * (i - 1);
            while (n % i == 0) n /= i;
        }
    }
    if (n > 1) ans = ans / n * (n - 1);
    return ans;
}
```

Eluer theorem

$$a^b = \begin{cases} a^{b \bmod \varphi(n)} & \gcd(a,b) = 1 \\ a^b & \gcd(a,b) \neq 1, b < \varphi(n) \\ a^{b \bmod \varphi(n) + \varphi(n)} & \gcd(a,b) \neq 1, b \geq \varphi(n) \end{cases} \pmod n$$

5.3 Fast power

```cpp
constexpr i64 MODP = 1000000007;
constexpr i64 power (i64 a, i64 b, i64 p = MODP) {
    i64 res = 1;
    for (; b; b >>= 1, a = a * a % p)
        if (b & 1) res = res * a % p;
    return res;
}
```

5.4 GCD

```cpp
i64 gcd(i64 a, i64 b) {
    return b == 0 ? a : gcd(b, a % b);
}
```

5.5 Matrix

```cpp
constexpr i64 MOD = 1000000007;
struct Matrix {
    int row, column;
    std::vector<std::vector<int>> data;

    Matrix() {}
    Matrix(int n, int m) {
        init(n, m);
    }

    void init(int n, int m) {
        row = n, column = m;
        data.assign(n, std::vector<int>(m));
    }

    std::vector<int> operator[](const int &x) {
        return data[x];
    }

    Matrix operator*(const Matrix &other) const {
        Matrix res(row, other.column);
        int r = row, c = other.column, l = column;

        for (int i = 0; i < r; i++) {
            for (int k = 0; k < l; k++) {
                int cur = data[i][k];
                for (int j = 0; j < c; j++) {
                    res[i][j] += cur * other.data[k][j];
                    res[i][j] %= MOD;
                }
            }
        }

        return res;
    }
};
```

6 Data Structure

6.1 Monotonic queue

```
std::deque<int> mq;
for (int i = 0; i < n; i++) {
    if (mq.front() < i - len + 1) mq.pop_front();
    while (not mq.empty() && mq.back() > a[i]) mq.pop_back();

    mq.emplace_back(a[i]);
}
```

6.2 Sparse Table (Rrange Maximum/Minimum Query)

```
template<class T>
struct SparseTable {
    int n;
    std::vector<std::vector<T>> st;

    using optFunction = std::function<T(const T &, const T &)>;
    optFunction opt;
    static T defaultOpt(const T &a, const T &b) {
        return std::max(a, b);
    }

    SparseTable(const std::vector<T> &_init, optFunction _opt = defaultOpt) {
        opt = _opt;
        init(_init);
    }

    void init(auto &_init) {
        n = _init.size();
        int cap = std::log2(n) + 1;
        st.assign(n, std::vector<T>(cap));

        for (int i = 0; i < n; i++) {
            st[i][0] = _init[i];
        }

        for (int j = 1; j < cap; j++) {
            int cur = 1 << (j - 1);
            for (int i = 0; i + cur < n; i++) {
                st[i][j] = opt(st[i][j - 1], st[i + cur][j - 1]);
            }
        }
    }

    T query(int l, int r) {
        int k = std::log2(r - l + 1);
        return opt(st[l][k], st[r - (1 << k) + 1][k]);
    }

    T query(int l, int r, int k) {
        return opt(st[l][k], st[r - (1 << k) + 1][k]);
    }
};
```

6.3 Disjoint union

```
struct DisjointSet {
    std::vector<int> f, _size;

    DisjointSet() {}
    DisjointSet(int n) {
```

```
        init(n);
    }

    void init(int n) {
        f.resize(n);
        std::iota(f.begin(), f.end(), 0);
        _size.assign(n, 1);
    }

    int find(int x) {
        return f[x] == x ? x : f[x] = find(f[x]);
    }

    bool merge(int u, int v) {
        int fu = find(u), fv = find(v);
        if (fu == fv) return false;

        f[fu] = fv;
        _size[fv] += _size[fu];
        return true;
    }

    bool same(int u, int v) {
        return find(u) == find(v);
    }

    int size(int x) {
        return _size[find(x)];
    }
};
```

6.4 Fenwick Tree

```
template<typename T>
struct Fenwick {
    int n;
    std::vector<T> a;

    Fenwick(int n = 0) {
        init(n);
    }

    void init(int n) {
        this->n = n;
        a.assign(n, T());
    }

    void add(int x, T v) {
        for (int i = x + 1; i <= n; i += i & -i) {
            a[i - 1] += v;
        }
    }

    T sum(int x) {
        auto ans = T();
        for (int i = x; i > 0; i -= i & -i) {
            ans += a[i - 1];
        }
        return ans;
    }

```

```
    T rangeSum(int l, int r) {
        return sum(r) - sum(l);
    }
};
```

6.5 Segment Tree

6.5.1 Classic

```
template<class Info>
struct SegmentTree {
    int n;
    std::vector<Info> info;

    SegmentTree() : n(0) {}
    SegmentTree(int n, Info _info = Info()) {
        init(std::vector<Info>(n, _info));
    }

    template<class T>
    SegmentTree(std::vector<T> _init) {
        init(_init);
    }
    template<class T>
    void init(std::vector<T> _init) {
        n = _init.size();
        info.assign(4 << std::__lg(n), Info());

        auto build = [&_init, this](auto self, int p, int l, int r) -> void {
            if (r - l == 1) {
                info[p] = Info(_init[l]);
                return;
            }

            int mid = (l + r) / 2;
            self(self, p * 2, l, mid);
            self(self, p * 2 + 1, mid, r);
            pull(p);
        };
        build(build, 1, 0, n);
    }

    void pull(int p) {
        info[p] = info[p * 2] + info[p * 2 + 1];
    }

    void update(int p, int l, int r, int x, const Info &v) {
        if (r - l == 1) {
            info[p] = v;
            return;
        }

        int mid = (l + r) / 2;
        if (x < mid) {
            update(p * 2, l, mid, x, v);
        } else {
            update(p * 2 + 1, mid, r, x, v);
        }
        pull(p);
    }
    void update(int x, const Info &v) {
        update(1, 0, n, x, v);
    }
```

```

    Info rangeQuery(int p, int s, int e, int l, int r) {
        if (s >= r or e <= l) {
            return Info();
        }

        if (s >= l and e <= r) {
            return info[p];
        }

        int mid = (s + e) / 2;
        return rangeQuery(p * 2, s, mid, l, r) + rangeQuery(p * 2 + 1, mid, e, l, r);
    }
    Info rangeQuery(int l, int r) {
        return rangeQuery(1, 0, n, l, r);
    }
};
```

6.5.2 Lazy tag

```
template<class Info, class Tag>
struct LazySegmentTree {
    int n;
    std::vector<Info> info;
    std::vector<Tag> tag;

    LazySegmentTree() : n(0) {}
    LazySegmentTree(int n_, Info v_ = Info()) {
        init(n_, v_);
    }
    template<class T>
    LazySegmentTree(std::vector<T> init_) {
        init(init_);
    }
    void init(int n_, Info v_ = Info()) {
        init(std::vector(n_, v_));
    }
    template<class T>
    void init(std::vector<T> init_) {
        n = init_.size();
        info.assign(4 << std::__lg(n), Info());
        tag.assign(4 << std::__lg(n), Tag());
        std::function<void(int, int, int)> build = [&](int p, int l, int r) {
            if (r - l == 1) {
                info[p] = init_[l];
                return;
            }
            int m = (l + r) / 2;
            build(2 * p, l, m);
            build(2 * p + 1, m, r);
            pull(p);
        };
        build(1, 0, n);
    }

    void pull(int p) {
        info[p] = info[2 * p] + info[2 * p + 1];
    }
    void apply(int p, const Tag &v) {
        info[p].apply(v);
        tag[p].apply(v);
    }
    void push(int p) {
```

```
        apply(2 * p, tag[p]);
        apply(2 * p + 1, tag[p]);
        tag[p] = Tag();
    }

    void modify(int p, int l, int r, int x, const Info &v) {
        if (r - l == 1) {
            info[p] = v;
            return;
        }
        int m = (l + r) / 2;
        push(p);
        if (x < m) {
            modify(2 * p, l, m, x, v);
        } else {
            modify(2 * p + 1, m, r, x, v);
        }
        pull(p);
    }
    void modify(int p, const Info &v) {
        modify(1, 0, n, p, v);
    }

    Info rangeQuery(int p, int l, int r, int x, int y) {
        if (l >= y || r <= x) {
            return Info();
        }
        if (l >= x && r <= y) {
            return info[p];
        }
        int m = (l + r) / 2;
        push(p);
        return rangeQuery(2 * p, l, m, x, y) + rangeQuery(2 * p + 1, m, r, x, y);
    }
    Info rangeQuery(int l, int r) {
        return rangeQuery(1, 0, n, l, r);
    }

    void rangeApply(int p, int l, int r, int x, int y, const Tag &v) {
        if (l >= y || r <= x) {
            return;
        }
        if (l >= x && r <= y) {
            apply(p, v);
            return;
        }
        int m = (l + r) / 2;
        push(p);
        rangeApply(2 * p, l, m, x, y, v);
        rangeApply(2 * p + 1, m, r, x, y, v);
        pull(p);
    }
    void rangeApply(int l, int r, const Tag &v) {
        return rangeApply(1, 0, n, l, r, v);
    }
};
```

7 Misc

7.1 Discrete

```
auto discrete = [](std::vector<int> a) {
    std::vector<int> n(a);
    std::sort(n.begin(), n.end());

```

```
    for (int i = 0; i < a.size(); i++) {
        a[i] = std::lower_bound(n.begin(), n.end(), a[i]) - n.begin();
    }
};
```

7.2 Leap year

Leap years are **evenly divisible by 4**. The most recent leap year was 2020 and the next leap year will be 2024. However, any year that is **evenly divided by 100 would not be a leap year unless it is evenly divided by 400**. This is why 1600, 2000, and 2400 are leap years, while 1700, 1800, 1900, 2100, 2200, and 2300 are common years, even though they are all divisible by 4.

```
bool isLeapYear (int year) {
    return (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0);
}
```

7.3 STL

```
std::vector<T,Allocator>::assign

void assign(size_type count, const T& value);

template<class InputIt>
void assign(InputIt first, InputIt last);
```

7.4 Sum of Prefix and Difference

2D

```
void build(std::vector<std::vector<int>> sum, auto a, int n, int m) {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            sum[i][j] = sum[i][j - 1] + a[i - 1][j - 1] -
                        sum[i - 1][j - 1] + sum[i - 1][j];
        }
    }
}

int query(auto sum, int a, int b, int c, int d) {
    return sum[c][d] + sum[a - 1][b - 1] - sum[a - 1][d] - sum[c][b - 1];
}
```

```
void build(auto dif, auto a, auto n, auto m) {
    for (int i = n; i > 0; i--) {
        for (int j = m; j > 0; j--) {
            dif[i][j] = a[i][j] - dif[i - 1][j] -
                        dif[i][j - 1] + dif[i - 1][j - 1];
        }
    }
}
void modi(auto dif, int x1, int x2, int y1, int y2, int x) {
    dif[x1][y1] += x, dif[x2 + 1][y2 + 1] += x;
    dif[x1][y2 + 1] -= x, dif[x2 + 1][y1] -= x;
}
```

7.5 Fast IO

```
namespace io {
  struct read {
```

```
        static constexpr int M = 1 << 23;
        char buf[M], *S = buf, *P = buf, c, l;

        inline char gc() {
            return (S == P && (P = (S = buf) + fread(buf, 1, M, stdin), S == P) ?
EOF : *S++);
        }

        template<typename T> read &operator>>(T &x) {
            for (c = 0; !isdigit(c); c = gc()) {
                l = c;
            }

            for (x = 0; isdigit(c); c = gc()) {
                x = x * 10 + c - '0';
            }

            return x = (l ^ 45) ? x : -x, *this;
        }
    } in;
}
```

7.6 CPU Information

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

using u32 = uint32_t;

static void cpu(u32 X, u32 Y, u32 msg[4]) {
    __cpuid_count(X, Y, msg[0], msg[1], msg[2], msg[3]);
}

int main() {
    u32 data[4];
    char msg[50];

    for (int i = 0; i < 3; ++i) {
        cpu(0x80000002 + i, 0, data);

        for (int j = 0; j < 4; ++j) {
            reinterpret_cast<u32 *>(msg)[i * 4 + j] = data[j];
        }
    }

    std::cout << msg;
}
```