算法笔记

数据结构

并查集

并查集（Disjoint Set Union）是一种管理元素分组的数据结构，支持两种基本操作：查找元素属于哪个集合、合并两个集合。并查集通常使用路径压缩和按秩合并来优化性能。

实现步骤

1. **初始化**：每个元素最初都是自己的父节点，秩（rank）为0。
2. **查找**：通过递归查找元素的根节点，并在查找过程中进行路径压缩。
3. **合并**：将两个集合的根节点合并，按秩合并以保持树的平衡。

使用数组的实现

已知有n个元素并且用数 $0, 1, 2, \cdots, n-1$ 来表示。

```cpp
class DSU {
private:
    std::vector<int> parent;
    std::vector<int> rank;
public:
    DSU(int n) {
        parent.resize(n);
        rank.resize(n, 0);
        for (int i = 0; i < n; ++i) {
            parent[i] = i; // 每个元素的父节点初始化为自己
        }
    }
    // 查找操作，带路径压缩
    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]); // 路径压缩
        }
        return parent[x];
    }
    // 合并操作，带按秩合并
    void unite(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
```

```
            } else if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
    // 检查两个元素是否属于同一集合
    bool isSameSet(int x, int y) {
        return find(x) == find(y);
    }
};
```

> **秩的意义**
>
> 秩表示树的深度（或高度）的一个**上界**，而不是精确的深度。秩的主要作用是指导合并操作，使得树尽量保持平衡。因此在代码中，查询并压缩路径时，即使树的高度可能减小，秩也不会更新。秩始终是不减小的。

路径压缩的过程也可以使用栈辅助实现。

```
int find(int x) {
    std::stack<int> path; // 用于保存路径上的节点
    while (parent[x] != x) {
        path.push(x); // 将当前节点压入栈
        x = parent[x]; // 移动到父节点
    }
    while (!path.empty()) {
        parent[path.top()] = x; // 将栈中节点连接到根节点
        path.pop();
    }
    return x;
}
```

当路径较长时，递归调用可能使系统栈溢出，改用栈实现，从堆中申请空间，防止栈溢出。

使用哈希的模板实现

相对于数组，适用于元素范围未知或稀疏的场景。

```
template <typename T>
class DSU {
private:
    std::unordered_map<T, T> parent; // 父节点映射
    std::unordered_map<T, int> rank; // 秩映射
```

```
public:
    // 查找操作，带路径压缩
    const T& find(const T& x) {
        if (parent.find(x) == parent.end()) {
            parent[x] = x; // 如果元素不存在，初始化父节点为自己
            rank[x] = 0;   // 初始化秩为0
        }
        if (parent[x] != x) {
            parent[x] = find(parent[x]); // 路径压缩
        }
        return parent[x];
    }
    // 合并操作，带按秩合并
    void unite(const T& x, const T& y) {
        const T& rootX = find(x);
        const T& rootY = find(y);
        if (rootX != rootY) {
            if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
    // 检查两个元素是否属于同一集合
    bool isSameSet(const T& x, const T& y) {
        return find(x) == find(y);
    }
};
```

长整数加法

输入为两个长整数，保证是正数，输出和。输入以字符串形式给出。

通过 `std::string` 存储与逐位计算

适合处理中等长度的整数（例如几十位到几百位）。

```
std::string addLongIntegers(const std::string& num1, const std::string& num2) {
    // 反转字符串，方便从低位到高位计算
    std::string reversedNum1 = num1;
    std::string reversedNum2 = num2;
    std::reverse(reversedNum1.begin(), reversedNum1.end());
    std::reverse(reversedNum2.begin(), reversedNum2.end());
```

```
    std::string result;  // 结果字符串
    int carry = 0;  // 进位
    int maxLength = std::max(reversedNum1.length(), reversedNum2.length());

    for (int i = 0; i < maxLength; ++i) {
        // 获取当前位的数字，如果超出长度则补 0
        int digit1 = (i < reversedNum1.length()) ? (reversedNum1[i] - '0') : 0;
        int digit2 = (i < reversedNum2.length()) ? (reversedNum2[i] - '0') : 0;

        // 计算当前位的和
        int sum = digit1 + digit2 + carry;
        carry = sum / 10; // 更新进位
        result.push_back((sum % 10) + '0'); // 将当前位的结果存入
    }
    // 如果最后还有进位，添加到结果中
    if (carry) result.push_back(carry + '0');
    // 反转结果，恢复正常顺序
    std::reverse(result.begin(), result.end());
    // 去除前导零（如果结果不是 "0"）
    result.erase(0, result.find_first_not_of('0'));
    if (result.empty()) result = "0";
    return result;
}
```

通过 `vector<int>` 分段保存

适合处理较长的整数（例如几百位到几千位）。

```
class LongInt {
public:
    static constexpr int M = 1e9;  // M进制分段，每段9位
    vector<int> vd;  // vd逆向保存结果
    LongInt() {}
    LongInt(int a) {
        while (a > 0) {
            vd.push_back(a % M);
            a /= M;
        }
    }
    LongInt(long long a) {
        while (a > 0) {
            vd.push_back(a % M);
            a /= M;
        }
    }
    LongInt(const string& num) {
        for (int i = num.size(); i > 0; i -= 9) {
            int start = max(0, i - 9);
```

```
            string chunkStr = num.substr(start, i - start);
            vd.push_back(stoi(chunkStr));
        }
    }

    string getString() {
        string result;
        for (int i = vd.size() - 1; i >= 0; --i) {
            string chunkStr = to_string(vd[i]);
            if (i != vd.size() - 1) {
                // 补前导零，确保每段都是 9 位
                chunkStr.insert(0, 9 - chunkStr.size(), '0');
            }
            result += chunkStr;
        }
        return result;
    }
    void output() {
        printf("%d", vd.back());
        for (int i = vd.size() - 2; i >= 0; --i) {
            printf("%09d", vd[i]);
        }
    }

    LongInt operator+(const LongInt& other) {
        LongInt result;
        int carry = 0; // 进位
        int maxChunks = max(vd.size(), other.vd.size());
        for (int i = 0; i < maxChunks; ++i) {
            int chunk1 = (i < vd.size()) ? vd[i] : 0;
            int chunk2 = (i < other.vd.size()) ? other.vd[i] : 0;

            int sum = chunk1 + chunk2 + carry;
            carry = sum / M; // 每段最多 9 位，进位是 sum / 10^9
            result.vd.push_back(sum % M);
        }
        // 如果最后还有进位，添加到结果中
        if (carry) {
            result.vd.push_back(carry);
        }
        return result;
    }
};
// 例子
int main() {
    LongInt a("1234567853264901234567890"), b("98765432109876543210");
    LongInt c = a + b;
    cout << c.getString() << endl;
    c.output();
```

```
    return 0;
}
```

单调队列

队列中的元素保持单调递增或递减的顺序（可以是非严格）。通常用于解决滑动窗口问题，在$O(1)$的时间内获取窗口内最小值或最大值。
例题：[洛谷1886]

思路

使用双端队列实现。为了保持队列的单调性，每次`push`元素时，检查队尾元素是否能与被插入元素满足单调性，若不满足，持续弹出队尾元素，最后再插入该元素。
队首始终保存着队列的最值。
在解决滑动窗口问题时，已知窗口大小为k，可以知道窗口外的上一个元素数值，然后检查队首元素，若相等则弹出。队列中始终最多保存着k个元素。

单调递减队列与滑动窗口最大值

```
class MonotonicQueue {
private:
    deque<int> dq;
public:
    void push(int value) {
        // 维护单调递减队列
        while (!dq.empty() && dq.back() < value) {
            dq.pop_back();
        }
        dq.push_back(value);
    }
    void pop(int value) {
        // 只有当队首元素等于当前要弹出的元素时才弹出
        if (!dq.empty() && dq.front() == value) dq.pop_front();
    }
    int max() { return dq.front(); }
};

vector<int> maxSlidingWindow(vector<int>& nums, int k) {
    MonotonicQueue mq;
    vector<int> result;
    for (int i = 0; i < nums.size(); ++i) {
        if (i < k - 1) {
            mq.push(nums[i]);
        } else {
            mq.push(nums[i]);
            result.push_back(mq.max());
            mq.pop(nums[i - k + 1]);
```

```
        }
    }
    return result;
}
```

这里在每次 `push` 元素并读取最大值后后直接调用 `pop`， 使得队列中最多保存着$k-1$个元素，下一次循环时直接 `push`。

单调递增队列与滑动窗口最小值

```
class MonotonicQueue {
private:
    deque<int> dq;
public:
    void push(int value) {
        // 移除队列中所有大于当前元素的元素
        while (!dq.empty() && dq.back() > value) {
            dq.pop_back();
        }
        dq.push_back(value); // 将当前元素插入队列
    }
    void pop(int value) {
        // 只有当队首元素等于当前要弹出的元素时才弹出
        if (!dq.empty() && dq.front() == value) dq.pop_front();
    }
    int min() { return dq.front(); }
};

vector<int> minSlidingWindow(vector<int>& nums, int k) {
    MonotonicQueue mq;
    vector<int> result;
    for (int i = 0; i < nums.size(); ++i) {
        if (i < k - 1) {  // 先填充窗口的前 k-1 个元素
            mq.push(nums[i]);
        } else {
            mq.push(nums[i]);  // 窗口已满
            result.push_back(mq.min());
            // 弹出窗口最左侧的元素
            mq.pop(nums[i - k + 1]);
        }
    }
    return result;
}
```

算法

快速幂

基本原理：

$$a^b = \begin{cases} a \cdot a^{b-1}, & b\text{为奇数} \\ \left(a^{\frac{b}{2}}\right)^2, & b\text{为偶数}. \end{cases}$$

例如求2^{13}的过程如下：

1. $a^{13} = a \times a^{12}$;
2. $a^{12} = \left(a^6\right)^2$;
3. $a^6 = \left(a^3\right)^2$;
4. $a^3 = a \times a^2$;
5. $a^2 = (a)^2$;
6. $a = a \times 1$.

算法时间复杂度为$O(\log n)$。

递归实现

```
// 计算 a^b mod m，m为大质数
int binaryPow(int a, int b, int m) {
        if (b == 0) return 1;  // a^0 = 1
        if (b & 1) {  // b & 1 等价于 b % 2 == 1，判断b的奇偶性
                return a * binaryPow(a, b - 1, m) % m;
        } else {
                int mul = binaryPow(a, b / 2, m);
                return mul * mul % m;
        }
}
```

注意如果 `b` 为偶数时直接 `return binaryPow(a, b/2, m) * binaryPow(a, b/2, m)` 这样时间复杂度仍然是$O(n)$。
考虑上面a^{13}的例子，可以将任意正整数分解为一系列2的幂之和且分解唯一，如$13 = 8 + 4 + 1 = 2^3 + 2^2 + 2^0 = 1101_{(2)}$，因此$a^{13} = a^8 \times a^4 \times a^1$。

迭代实现

```
// 计算 a^b mod m，m为大质数
int binaryPow(int a, int b, int m) {
        int ans = 1;
        while (b > 0) {
                if (b & 1) {  // b的二进制末尾为1
                        ans = ans * a % m;
                }
                a = a * a % m;
```

```
            b >>= 1;  // b右移一位
        }
        return ans;
}
```

素数

埃拉托斯特尼筛法

给定n，输出不大于n的所有质数。

```
vector<int> prime(int n) {
    vector<int> ans;  // 保存结果
    vector<int> flag(n + 1, 0);  // 标记数组，0为质数，1为合数。arr[0~1]不使用
    for (int i = 2; i < n + 1); ++i) {  // 从最小的质数2开始遍历
        if (!flag[i]) {  // i为质数
            ans.push_back(i);
            for (int j = i * 2; j < n + 1; j += i) flag[j] = 1;  // 标记i的所有倍数
        }
    }
    return ans;
}
```

如果只需要处理标记数组，最外层 `i` 的遍历为 `0 ~ sqrt(n)` 即可。

质因数分解

给定n，输出质因式分解的结果（质因数和对应的指数）。
思路：

1. 计算n的所有质因数（若n为合数，最大质因数为n/2；若n为质数，则最大质因数为n)；
2. 依次用n除以质因数，每次可以整除时累加指数数组一次。

```
vector<int> p = prime(n);  // 不大于n的所有质数
vector<int> a(p.size());  // 各质数对应的指数数组
for (int i = 0; i < p.size(); ++i) {
    while (n % p[i] == 0) {
        n /= p[i];
        ++a[i];
    }
}
```

求$n!$的质因数分解

基本思路：从$2 \sim n$依次求其质因数分解，累加指数。
优化思路：对于质数2，考虑$2 \sim n$中的数，所有偶数都包含1个质因数2，而所有4的倍数都额外包含1个质因数2，所有8的倍数又额外包含一个质因数2……因此对于$n!$，质因数p的指数为：

$$\left[\frac{n}{p}\right] + \left[\frac{n}{p^2}\right] + \left[\frac{n}{p^3}\right] + \left[\frac{n}{p^4}\right] + \cdots + \left[\frac{n}{p^k}\right]$$

其中$[\cdot]$为取整函数，直到$n < p^k$。

```
vector<int> p = prime(n);  // 不大于n的所有质数
vector<int> a(p.size());  // 各质数对应的指数数组
for (int i = 0; i < p.size(); ++i) {
    for (int wk = n; wk > 0;) {
        wk /= p[i];
        a[i] += wk;
    }
}
```