【第四十一课】傅立叶变换与信息隐写术（一）

1.fft

```cpp
/*************************************************************************
	> File Name: 1.fft.cpp
	> Author: huguang
	> Mail: hug@haizeix.com
	> Created Time:
 ************************************************************************/

#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <queue>
#include <stack>
#include <algorithm>
#include <string>
#include <map>
#include <set>
#include <vector>
using namespace std;

#define PI acos(-1)

struct Complex {
    Complex(double r = 0, double i = 0) : r(r), i(i) {}
    double real() { return r; };
    Complex conj() { return Complex(r, -i); }
    Complex &operator/=(double n) { r /= n, i /= n;  return *this; }
    Complex operator+(const Complex &obj) {
        return Complex(r + obj.r, i + obj.i);
    }
    Complex operator*(const Complex &obj) {
        return Complex(r * obj.r - i * obj.i, r * obj.i + i * obj.r);
    }
    Complex operator-(const Complex &obj) {
        return Complex(r - obj.r, i - obj.i);
    }

    double r, i;
};

ostream &operator<<(ostream &out, const Complex &obj) {
    cout << obj.r << "+" << obj.i << "i";
    return out;
}

```

```cpp
struct FastFourierTransform {
    void __transform(vector<Complex> &a, int n, int type = 1) {
        if (n == 1) return ;
        int m = n / 2;

        // P0, P1
        vector<Complex> a0(m), a1(m);
        for (int i = 0; i < m; i++) a0[i] = a[i * 2], a1[i] = a[i * 2 + 1];
        __transform(a0, m, type);
        __transform(a1, m, type);

        // merge P0, P1
        Complex w(1, 0), wn(cos(2.0 * PI / n), type * sin(2.0 * PI / n));
        for (int k = 0; k < m; k++) {
            a[k] = a0[k] + w * a1[k];
            a[k + m] = a0[k] - w * a1[k];
            w = w * wn;
        }
        return ;
    }
    void dft(vector<Complex> &a, int n) {
        __transform(a, n);
        return ;
    }
    void idft(vector<Complex> &a, int n) {
        __transform(a, n, -1);
        for (int i = 0; i < n; i++) a[i] /= n;
        return ;
    }
};

int main() {
    int n, m;
    cin >> n >> m;
    int k = 1;
    while (k <= n + m + 1) k *= 2;
    vector<Complex> a(k), b(k), c(k);
    for (int i = 0; i <= n; i++) cin >> a[i].r;
    for (int i = 0; i <= m; i++) cin >> b[i].r;
    FastFourierTransform fft;
    fft.dft(a, k);
    fft.dft(b, k);
    cout << "A(x) value : ";
    for (int i = 0; i < k; i++) cout << a[i] << " ";
    cout << endl;

    cout << "B(x) value : ";
    for (int i = 0; i < k; i++) cout << b[i] << " ";
    cout << endl;
```

```
    for (int i = 0; i < k; i++) c[i] = a[i] * b[i];

    cout << "C(x) value : ";
    for (int i = 0; i < k; i++) cout << c[i] << " ";
    cout << endl;

    fft.idft(c, k);

    cout << "C(x) parameters : ";
    for (int i = 0; i < n + m + 1; i++) {
        cout << c[i].r << " " ;
    }
    cout << endl;
    return 0;
}
```

fft_bug

```
/*************************************************************************
  > File Name: 1.fft.cpp
  > Author: huguang
  > Mail: hug@haizeix.com
  > Created Time:
 ************************************************************************/

#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <queue>
#include <stack>
#include <algorithm>
#include <string>
#include <map>
#include <set>
#include <vector>
using namespace std;

#define PI acos(-1)

struct Complex {
    Complex(double r = 0, double i = 0) : r(r), i(i) {}
    double real() { return r; };
    Complex conj() { return Complex(r, -i); }
    Complex &operator/=(double n) { r /= n, i /= n;  return *this; }
    Complex operator+(const Complex &obj) {
        return Complex(r + obj.r, i + obj.i);
    }
    Complex operator*(const Complex &obj) {
```

```
        return Complex(r * obj.r - i * obj.i, r * obj.i + i * obj.r);
    }
    Complex operator-(const Complex &obj) {
        return Complex(r - obj.r, i - obj.i);
    }

    double r, i;
};

ostream &operator<<(ostream &out, const Complex &obj) {
    cout << obj.r << "+" << obj.i << "i";
    return out;
}

struct FastFourierTransform {
    void __transform(vector<Complex> &a, int n, int type = 1) {
        if (n == 1) return ;
        int m = n / 2;

        // P0, P1
        vector<Complex> a0(m), a1(m);
        for (int i = 0; i < m; i++) a0[i] = a[i * 2], a1[i] = a[i * 2 + 1];
        __transform(a0, m, type);
        __transform(a1, m, type);

        // merge P0, P1
        Complex w(1, 0), wn(cos(2.0 * PI / n), type * sin(2.0 * PI / n));
        for (int k = 0; k < m; k++) {
            a[k] = a0[k] + w * a1[k];
            a[k + m] = a0[k] - w * a1[k];
            w = w * wn;
        }
        return ;
    }
    void dft(vector<Complex> &a, int n) {
        __transform(a, n);
        return ;
    }
    void idft(vector<Complex> &a, int n) {
        __transform(a, n, -1);
        for (int i = 0; i < n; i++) a[i] /= n;
        return ;
    }
};

int main() {
    int n, m;
    cin >> n >> m;
    int k = 1;
```

```
    while (k <= n + m + 1) k *= 2;
    vector<Complex> a(k), b(k), c(k);
    for (int i = 0; i <= n; i++) cin >> a[i].r;
    for (int i = 0; i <= m; i++) cin >> b[i].r;
    FastFourierTransform fft;
    fft.dft(a, k);
    fft.dft(b, k);

    for (int i = 0; i < k; i++) {
        cout << a[i] << " ";
    }
    cout << endl;

    for (int i = 0; i < k; i++) {
        cout << b[i] << " ";
    }
    cout << endl;

    for (int i = 0; i < n + m + 1; i++) c[i] = a[i] * b[i];
    fft.idft(c, k);
    for (int i = 0; i < n + m + 1; i++) {
        cout << c[i].r << " " ;
    }
    cout << endl;
    return 0;
}
```

923. 三数之和的多种可能

给定一个整数数组 `arr` ，以及一个整数 `target` 作为目标值，返回满足 `i < j < k` 且 `arr[i] + arr[j] + arr[k] == target` 的元组 `i, j, k` 的数量。

由于结果会非常大，请返回 `109 + 7` 的模。

示例 1：

```
输入：arr = [1,1,2,2,3,3,4,4,5,5], target = 8
输出：20
```

```
class Solution {
public:
    const int mod_num = (int)(1e9 + 7);
    int twoSumMulti(vector<int> &arr, int l, int r, int target) {
        int ans = 0;
        while (l < r) {
            if (arr[l] + arr[r] < target) l += 1;
            else if (arr[l] + arr[r] > target) r -= 1;
            else {
                if (arr[l] == arr[r]) {
```

```
                    int n = r - l + 1;
                    ans += n * (n - 1) / 2;
                    ans %= mod_num;
                    break;
                }
                int lcnt = 1, rcnt = 1;
                while (arr[l + 1] == arr[l]) {
                    lcnt += 1;
                    l += 1;
                }
                while (arr[r - 1] == arr[r]) {
                    rcnt += 1;
                    r -= 1;
                }
                ans += lcnt * rcnt;
                ans %= mod_num;
                l += 1, r -= 1;
            }
        }
        return ans;
    }
    int threeSumMulti(vector<int>& arr, int target) {
        sort(arr.begin(), arr.end());
        int n = arr.size(), ans = 0;
        for (int i = 0, I = n - 2; i < I; i++) {
            ans += twoSumMulti(arr, i + 1, n - 1, target - arr[i]);
            ans %= mod_num;
        }
        return ans;
    }
};
```

1963. 使字符串平衡的最小交换次数

给你一个字符串 `s` ，**下标从 0 开始** ，且长度为偶数 `n` 。字符串 **恰好** 由 `n / 2` 个开括号 `'['` 和 `n / 2` 个闭括号 `']'` 组成。

只有能满足下述所有条件的字符串才能称为 **平衡字符串** ：

- 字符串是一个空字符串，或者
- 字符串可以记作 `AB` ，其中 `A` 和 `B` 都是 **平衡字符串** ，或者
- 字符串可以写成 `[C]` ，其中 `C` 是一个 **平衡字符串** 。

你可以交换 **任意** 两个下标所对应的括号 **任意** 次数。

返回使 `s` **变成** 平衡字符串** 所需要的 **最小** 交换次数。

示例 1：

```
输入: s = "][]["
输出: 1
解释: 交换下标 0 和下标 3 对应的括号，可以使字符串变成平衡字符串。
最终字符串变成 "[[]]" 。
```

```cpp
class Solution {
public:
    int minSwaps(string s) {
        int ans = 0, l = 0, r = s.size() - 1, lcnt = 0, rcnt = 0;
        lcnt += (s[l] == '[' ? 1 : -1);
        rcnt += (s[r] == ']' ? 1 : -1);
        while (l < r) {
            while (l < r && lcnt >= 0) l += 1, lcnt += (s[l] == '[' ? 1 : -1);
            while (l < r && rcnt >= 0) r -= 1, rcnt += (s[r] == ']' ? 1 : -1);
            if (l >= r) break;
            ans += 1;
            lcnt += 2, rcnt += 2;
        }
        return ans;
    }
};
```

1984. 学生分数的最小差值

给你一个 **下标从 0 开始** 的整数数组 `nums` ，其中 `nums[i]` 表示第 `i` 名学生的分数。另给你一个整数 `k` 。

从数组中选出任意 `k` 名学生的分数，使这 `k` 个分数间 **最高分** 和 **最低分** 的 **差值** 达到 **最小化** 。

返回可能的 **最小差值** 。

示例 1：

```
输入: nums = [90], k = 1
输出: 0
解释: 选出 1 名学生的分数，仅有 1 种方法:
- [90] 最高分和最低分之间的差值是 90 - 90 = 0
可能的最小差值是 0
```

```cpp
class Solution {
public:
    int minimumDifference(vector<int>& nums, int k) {
        sort(nums.begin(), nums.end());
        int ans = INT_MAX;
        for(int i = k - 1 ,n = nums.size(); i < n; i++){
            ans = min(ans,nums[i] - nums[i - k + 1]);
        }
        return ans;
    }
};
```

1981. 最小化目标值与所选元素的差

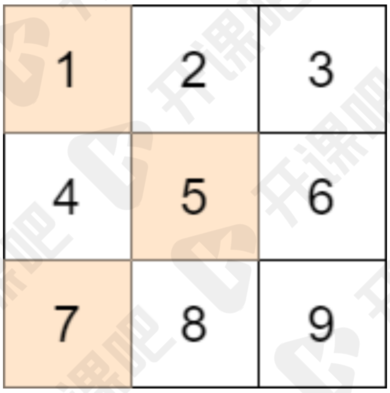
给你一个大小为 `m x n` 的整数矩阵 `mat` 和一个整数 `target` 。

从矩阵的 **每一行** 中选择一个整数，你的目标是 **最小化** 所有选中元素之 **和** 与目标值 `target` 的 **绝对差** 。

返回 **最小的绝对差** 。

`a` 和 `b` 两数字的 **绝对差** 是 `a - b` 的绝对值。

示例 1：




```
输入：mat = [[1,2,3],[4,5,6],[7,8,9]], target = 13
输出：0
解释：一种可能的最优选择方案是：
- 第一行选出 1
- 第二行选出 5
- 第三行选出 7
所选元素的和是 13 ，等于目标值，所以绝对差是 0 。
```

```cpp
class Solution {
public:
    int minimizeTheDifference(vector<vector<int>>& mat, int target) {
        int n = mat.size(), m = mat[0].size(), sum = 0;
        unordered_set<int> h[2];
        for (auto x : mat[0]) h[0].insert(x), sum = max(sum, x);
        for (int i = 1; i < n; i++) {
```

```
            int ind = i % 2, pre_ind = (i - 1) % 2;
            h[ind].clear();
            int max_num = 0;
            for (auto x : mat[i]) max_num = max(x, max_num);
            sum += max_num;
            for (int j = i + 1; j <= sum; j++) {
                for (auto x : mat[i]) {
                    if (h[pre_ind].find(j - x) == h[pre_ind].end()) continue;
                    h[ind].insert(j);
                    break;
                }
            }
        }
        int ans = INT_MAX;
        for (auto x : h[(n - 1) % 2]) ans = min(ans, abs(target - x));
        return ans;
    }
};
```

1987. 不同的好子序列数目

给你一个二进制字符串 `binary` 。 `binary` 的一个 **子序列** 如果是 **非空** 的且没有 **前导 0** （除非数字是 `"0"` 本身），那么它就是一个 **好** 的子序列。

请你找到 `binary` **不同好子序列** 的数目。

- 比方说，如果 `binary = "001"` ，那么所有 **好** 子序列为 `["0", "0", "1"]` ，所以 **不同** 的好子序列为 `"0"` 和 `"1"` 。 注意，子序列 `"00"` ， `"01"` 和 `"001"` 不是好的，因为它们有前导 0 。

请你返回 `binary` 中 **不同好子序列** 的数目。由于答案可能很大，请将它对 `109 + 7` **取余** 后返回。

一个 **子序列** 指的是从原数组中删除若干个（可以一个也不删除）元素后，不改变剩余元素顺序得到的序列。

示例 1：

```
输入：binary = "001"
输出：2
解释：好的二进制子序列为 ["0", "0", "1"] 。
不同的好子序列为 "0" 和 "1" 。
```

```
class Solution {
public:
    int numberOfUniqueGoodSubsequences(string binary) {
        int n = binary.size(), mod_num = (int)(1e9+7);
        int f[n + 1][2], flag = 0;
        f[n][0] = f[n][1] = 0;
        for (int i = n - 1; i >= 0; i--) {
            int j = (binary[i] - '0');
            if (j == 0) flag = 1;
            f[i][j] = f[i + 1][j] + f[i + 1][1 - j] + 1;
```

```
            f[i][1 - j] = f[i + 1][1 - j];
            f[i][j] %= mod_num;
            f[i][1 - j] %= mod_num;
        }
        return f[0][1] + flag;
    }
};
```

1911. 最大子序列交替和

一个下标从 **0** 开始的数组的 **交替和** 定义为 **偶数** 下标处元素之 **和** 减去 **奇数** 下标处元素之 **和** 。

- 比方说，数组 `[4,2,5,3]` 的交替和为 `(4 + 5) - (2 + 3) = 4` 。

给你一个数组 `nums` ，请你返回 `nums` 中任意子序列的 **最大交替和** （子序列的下标 **重新** 从 0 开始编号）。

一个数组的 **子序列** 是从原数组中删除一些元素后（也可能一个也不删除）剩余元素不改变顺序组成的数组。比方说， `[2,7,4]` 是 `[4,**2**,3,**7**,2,1,**4**]` 的一个子序列（加粗元素），但是 `[2,4,2]` 不是。

示例 1：

```
输入：nums = [4,2,5,3]
输出：7
解释：最优子序列为 [4,2,5] ，交替和为 (4 + 5) - 2 = 7 。
```

```
class Solution {
public:
    long long maxAlternatingSum(vector<int>& nums) {
        int n = nums.size();
        long long sub_max = INT_MIN, add_max = nums[0], a, b, ans = nums[0];
        for (int i = 1; i < n; i++) {
            a = max(sub_max + nums[i], (long long)nums[i]);
            b = add_max - nums[i];
            ans = max(ans, max(a, b));
            sub_max = max(sub_max, b);
            add_max = max(add_max, a);
        }
        return ans;
    }
};
```