Chương 5. MỘT SỐ KỸ THUẬT CƠ BẢN

5.1. Kỹ thuật mảng thống kê

Kỹ thuật mảng thống kê đếm tần số xuất hiện của các giá trị trong một dãy, từ đó giúp các thao tác liên quan đến việc thống kê giá trị của dãy được thực hiện một cách hiệu quả.

Xét bài toán sau: Cho dãy gồm n số nguyên $a_1, a_2, \dots, a_n (0 \le a_i \le maxV; 1 \le n \le 10^6)$. Hãy thực hiện các yêu cầu

- Đếm số giá trị phân biệt của dãy.
- Cho biết giá trị có tần số xuất hiện nhiều nhất. Nếu có nhiều giá trị có cùng tần số lớn nhất thì cho biết giá trị nhỏ nhất trong số đó.
- Sắp xếp dãy có thứ tự tăng dần

Kỹ thuật mảng thống kê lưu tần số xuất hiện của giá trị x tại vị trí x trong mảng thống kê `cnt`, nói cách khác `cnt[x]` cho biết tần số xuất hiện của giá trị x trong dãy. Mảng thống kê `cnt` được xây dựng như sau

```
cnt[maxV+1] = {0};

for (int i = 1; i <= n; ++v)
    ++cnt[a[i]];
```

Độ phức tạp của thao tác: $O(n)$.

Như vậy với mảng thống kê `cnt[0]`, `cnt[1]`, …, `cnt[maxV]` ta có thể thực hiện các yêu cầu trên như sau

Đếm số giá trị phân biệt của dãy

```
int ans = 0;
for (int v = 0; v <= maxV; ++v)
    if (cnt[v] > 0) ++ans;
```

Độ phức tạp của thao tác: $O(maxV)$.

Tìm giá trị xuất hiện nhiều nhất

```
int maxfrq = 0, val;
for (int v = 0; v <= maxV; ++v)
    if (cnt[v] > maxfrq)
    {
        maxfrq = cnt[v];
        val = v;
```

```
    }
```

Độ phức tạp của thao tác: $O(maxV)$.

Sắp xếp dãy (phương pháp distributing sort)

```
for (int v = 0; v <= maxV; ++v)
    for (int i = 0; i < cnt[v]; ++i)
        cout<<v<<" ";
```

Độ phức tạp của thao tác: $O(n)$.

Phương pháp mảng thống kê thực hiện các yêu cầu với độ phức tạp tuyến tính. Kích thước của mảng `cnt` chính là độ lớn miền giá trị của x (giá trị phần tử dãy). Do đó nếu miền giá trị của các phần tử trong dãy quá lớn thì không thể áp dụng phương pháp này. Phương pháp mảng thống kê chỉ hiệu quả và khả thi khi độ lớn miền giá trị của các phần tử trong dãy cỡ 10^7.

5.2. Kỹ thuật 2 con trỏ

Kỹ thuật 2 con trỏ được áp dụng cho các bài toán xử lý trên dãy số nguyên. Kỹ thuật này sử dụng 2 biến i và j để quản lý một đoạn con gồm các phần tử nằm liên tiếp từ vị trí i đến vị trí j trong dãy hoặc xử lý 2 phần tử ở 2 vị trí i và j trong dãy. Con trỏ i và j có thể di chuyển trên cùng một dãy hoặc trên 2 dãy khác nhau. Kỹ thuật 2 con trỏ có những dạng thường gặp sau:

Hai con trỏ cùng chiều

Trong kỹ thuật này, 2 con trỏ i và j xuất phát từ đầu dãy và di chuyển về cuối dãy. Trong đó con trỏ j di chuyển nhanh hơn và đi trước, con trỏ i di chuyển chậm hơn và đi sau.

Bài toán 1: Cho dãy số nguyên gồm n phần tử thỏa điều kiện $a_1 \le a_2 \le \cdots \le a_n$. Hãy loại các giá trị trùng lắp ra khỏi dãy.

Cách 1: sử dụng kỹ thuật vét thông thường, độ phức tạp $O(n^2)$

```
void RemoveDup1(int a[], int& n)
{
    if (n == 0) return 0;
    for (int i = 1; i < n;)
    {
        if (a[i] == a[i-1])
        {
            for (int j = i+1; j < n; j++)
                a[j-1] = a[j];
            n--;
```

```
            }
            else i++;
        }
    }
```

Cách 2: Sử dụng kỹ thuật 2 con trỏ, độ phức tạp $O(n)$

```
void RemoveDup2(int a[], int& n)
{
    if (n == 0) return 0;
    int i = 0;
    for (int j = 1; j < n; j++)
    {
        if (a[i] != a[j])
            a[++i] = a[j];
    }
    n = i+1;
}
```

Bài toán 2: Cho 2 dãy số nguyên $a_1, a_2, \dots, a_n$ và $b_1, b_2, \dots, b_m$. Hãy kiểm tra dãy $\{a_i\}$ có phải là một dãy con gồm các phần tử không liên tiếp nhau của dãy $\{b_j\}$?

Thuật toán: Sử dụng kỹ thuật 2 con trỏ, độ phức tạp $O(m)$

```
bool CheckSub(int a[], int n, int b[], int m)
{
    for (int j = 0, i = 0; j < m; j++)
        if (a[i] == b[j])
        {
            i++;
            if (i == n) return true;
        }
    return false;
}
```

Bài toán 3: Cho dãy số nguyên dương $a_1, a_2, \dots, a_n$ và số nguyên dương S. Hãy chỉ ra một dãy con gồm các phần tử liên tiếp sao cho tổng của chúng đúng bằng S.

Nhận xét: một dãy con liên tiếp là dãy gồm các phần tử nằm từ vị trí i đến vị trí $j (1 \le i \le j \le n)$.

Cách 1: Sử dụng phương pháp vét cạn. Xét tất cả cặp vị trí $(i \le j)$ và kiểm tra tổng $a_i + a_{i+1} + \cdots + a_j$ có bằng S. Độ phức tạp $O(n^2)$

```
void FindSub1(int a[], int n, long long S, int &lo, int &hi)
{
    for (int i = 1; i <= n; ++i)
```

```
    {
        long long t = 0;
        for (int j = i; j <= n; ++j)
        {
            t = t + a[j];
            if (t == S)
            {
                lo = i, hi = j;
                return;
            }
        }
    }
}
```

Cách 2: Sử dụng kỹ thuật 2 con trỏ. Độ phức tạp $O(n)$

- Xét cặp vị trí $(i \le j)$, đặt $t = a_i + a_{i+1} + \cdots + a_j$.
- Giả sử $t < S$ và $t + a_{j+1} > S$, khi đó ta tăng con trỏ i đến khi $a_i + a_{i+1} + \cdots + a_j + a_{j+1} < S$
- Lặp lại thao tác trên cho đến khi tìm được dãy con có tổng đúng bằng S hoặc báo cáo không tìm thấy.

```
void FindSub2(int a[], int n, long long S, int &lo, int &hi)
{
    int i = 1, j = 0;
    lo = hi = -1;
    long long t = 0;
    while (i <= n && j <= n)
    {
        if (t + a[j+1] <= S)
            t = t + a[++j];
        else t = t – a[i++];
        if (t == S)
        {
            lo = i, hi = j;
            return;
        }
    }
}
```

Hai con trỏ ngược chiều

Trong kỹ thuật này, con trỏ i xuất phát từ đầu dãy và di chuyển về cuối dãy, con trỏ j xuất phát từ cuối dãy và di chuyển theo chiều ngược lại với con trỏ i.

Bài toán 1: Cho chuỗi kí tự, đảo ngược thứ tự các kí tự của chuỗi

Thuật toán: sử dụng kỹ thuật 2 con trỏ, độ phức tạp $O(n)$

```
void Reverse(string s)
{
    int i = 0, j = s.size() -1;
    while (i < j)
    {
        swap(s[i], s[j]);
        i++, j--;
    }
}
```

Bài toán 2: Cho chuỗi s chỉ gồm các kí tự chữ cái in thường. Hãy kiểm tra xem s có phải là chuỗi Palindrome hay không? Chuỗi rỗng được xem là chuỗi Palindrome

Thuật toán: Sử dụng kỹ thuật 2 con trỏ, độ phức tạp $O(n)$

```
bool isPalindrome(string s)
{
    int len = s.size();
    if (len == 0) return true;
    int i = 0, j = len-1;
    while (i < j)
    {
        if (s[i] != s[j])
            return false;
        i++, j--;
    }
    return true;
}
```

Bài toán 3: Cho dãy gồm n số nguyên thỏa $a_1 \leq a_2 \leq \cdots \leq a_n$ và số nguyên x. Hãy tìm vị trí 2 phần tử khác nhau của dãy có tổng đúng bằng x.

Thuật toán: Sử dụng kỹ thuật 2 con trỏ, độ phức tạp $O(n)$

```
void TwoSum(int a[], int n, int x, int &i, int &j)
{
    i = 0, j = n-1;
    while (i < j)
    {
        if (a[i] + a[j] == x)
            return;
        if (a[i] + a[j] < x) i++;
        else j--;
    }
    if (i >= n) i = -1;
```

```
    if (j < 0) j = -1;
}
```

Bài toán 4: Cho dãy gồm n số nguyên $a_1, a_2, \dots, a_n$ và số nguyên k. Hãy thực hiện xoay dãy k bước, mỗi bước đưa phần tử ở cuối dãy về đầu dãy. Ví dụ dãy $n = 7$ phần tử $\{1,2,3,4,5,6,7\}$ và $k = 3$, dãy biến đổi qua 3 bước như sau $\{1,2,3,4,5,6,7\} \xrightarrow{1} \{7,1,2,3,4,5,6\} \xrightarrow{2} \{6,7,1,2,3,4,5\} \xrightarrow{3} \{5,6,7,1,2,3,4\}$.

Cách 1: Sử dụng phương pháp vét cạn, độ phức tạp $O(n * k)$

```
void rotate1(int a[], int n, int k)
{
    for (int i = 1; i <= k; i++)
    {
        int x = a[n-1];
        for (int j = n-1; j > 0; j--)
            a[j] = a[j-1];
        a[0] = x;
    }
}
```

Cách 2: Sử dụng mảng phụ, độ phức tạp $O(n)$

Ý tưởng: khi thực hiện một phép xoay phải dãy thì mỗi phần tử của dãy đều bị dịch tới 1 vị trí, do đó khi thực hiện k phép xoay phải thì phần tử thứ i của dãy bị dịch tới vị trí $i + k$.

```
void rotate2(int a[], int n, int k)
{
    vector<int> b(n);
    for (int i = 0; i < n; i++) b[(i+k) % n] = a[i];

    for (int i = 0; i < n; i++) a[i] = b[i];
}
```

Cách 3: Sử dụng kỹ thuật đảo mảng, độ phức tạp $O(n)$

```
void reverse(int a[], int l, int r)
{
    int i = l, j = r;
    while (i < j)
        swap(a[i++], a[j--]);
}

void rotate3(int a[], int n, int k)
{
    k = k % n;
```

```
    reverse(a, 0, n-k-1);
    reverse(a, n-k, n-1);
    reverse(a, 0, n-1);
}
```

5.3. Kỹ thuật mảng hằng

Một trong những ứng dụng hiệu quả của mảng hằng là hạn chế các điều kiện rẽ nhánh làm tinh gọn chương trình. Xét một số bài toán sau

Bài toán 1: Cho số nguyên $n(1 \le n \le 12)$. Viết hàm in ra màn hình tên tiếng Anh (viết tắt) của tháng n.

So sánh 2 đoạn chương trình minh họa dưới để thấy sự khác biệt giữa việc sử dụng câu lệnh rẽ nhánh (đoạn chương trình 1) và sử dụng mảng hằng (đoạn chương trình 2).

Đoạn chương trình 1: Sử dụng câu lệnh rẽ nhánh

```
void PrintMonth(int n)
{
    if (n == 1) cout<<"Jan";
    else if (n == 2) cout<<"Feb";
    else if (n == 3) cout<<"Mar";
    else if (n == 4) cout<<"Apr";
    else if (n == 5) cout<<"May";
    else if (n == 6) cout<<"Jun";
    else if (n == 7) cout<<"Jul";
    else if (n == 8) cout<<"Aug";
    else if (n == 9) cout<<"Sep";
    else if (n == 10) cout<<"Oct";
    else if (n == 11) cout<<"Nov";
    else cout<<"Dec";
}
```

Đoạn chương trình 2: Sử dụng mảng hằng

```
//khai báo mảng hằng month
string month[13] = {"", "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
void PrintMonth(int n)
{
    cout<<month[n];
}
```

Bài toán 2: Cho bảng số nguyên A kích thước $m \times n(1 \le m,n \le 10^3)$. Hãy đếm số ô trong bảng thỏa tính chất giá trị tại ô đó lớn hơn giá trị tất cả ô chung cạnh và chung đỉnh xung quanh.

Đoạn chương trình 1: Sử dụng câu lệnh rẽ nhánh

```
int Count(int m, int n)
```

```
{
    int ans = 0;
    for (int i = 1; i <= m; ++i)
        for (int j = 1; j <= n; ++j)
        {
            if (A[i][j] > A[i][j-1] && A[i][j] > A[i-1][j-1] &&
                A[i][j] > A[i-1][j] && A[i][j] > A[i-1][j+1] &&
                A[i][j] > A[i][j+1] && A[i][j] > A[i+1][j+1] &&
                A[i][j] > A[i+1][j] && A[i][j] > A[i+1][j-1]) ++ans;
        }
    return ans;
}
```

Đoạn chương trình 2: Sử dụng mảng hằng

```
int dx[8] = {0, -1, -1, -1, 0, 1, 1, 1}; //khai báo mảng hằng dx
int dy[8] = {-1, -1, 0, 1, 1, 1, 0, -1}; //khai báo mảng hằng dy
int Count(int m, int n)
{
    int ans = 0;
    for (int i = 1; i <= m; ++i)
        for (int j = 1; j <= n; ++j)
        {
            int cnt = 0;
            for (int k = 0; k < 8; ++k)
                if (A[i][j] > A[i + dx[k]][j + dy[k]]) ++cnt;
            if (cnt == 8) ++ans;
        }
    return ans;
}
```

Nhận xét: Kỹ thuật mảng hằng giúp rút gọn các điều kiện được xét trong câu lệnh rẽ nhánh hoặc tránh việc xét quá nhiều câu lệnh rẽ nhánh. Điều này giúp cho chương trình giảm thiểu các lỗi khi xét các điều kiện phức hợp và trong một số trường hợp có thể giúp chương trình chạy nhanh hơn.

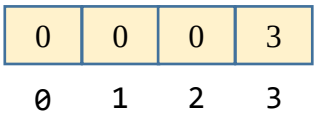
5.4. Kỹ thuật xử lý số nguyên lớn

Các kiểu dữ liệu số nguyên trong đa số các ngôn ngữ lập trình hiện tại cho phép thực hiện tính toán với miền giá trị lên đến $2^{63} - 1$. Tuy nhiên một số bài toán lập trình cần thực hiện các thao tác tính toán trên các số nguyên với số lượng chữ số lên đến hàng ngàn chữ số hoặc hơn. Các số nguyên như vậy được gọi là các số nguyên lớn.

Ngôn ngữ C/C++ không hỗ trợ xử lý tính toán trên các số nguyên có miền giá trị lớn hơn $2^{63} - 1$. Do đó, ta cần phải cài đặt các thao tác tính toán cơ bản trên các số nguyên lớn để phục vụ giải các bài toán liên quan.

Có nhiều cách để biểu diễn một số nguyên lớn, ta có thể coi số nguyên lớn như là một dãy các chữ số, do đó có thể sử dụng xâu kí tự, mảng một chiều hoặc danh sách liêt kết để biểu diễn một số nguyên lớn. Tài liệu này sử dụng mảng một chiều để biểu diễn một số nguyên lớn vì tính đơn giản trong cài đặt.

Nhận xét rằng các phép toán cộng, trừ, nhân trên số nguyên đều thực hiện từ hàng đơn vị trở đi. Do đó, để thuận tiện cho việc cài đặt, số nguyên lớn được biểu diễn thành mảng với các chữ số có thứ tự ngược lại với giá trị thực sự của nó. Chẳng hạn số nguyên 3000 được biểu diễn như sau:



Khai báo số nguyên lớn

```
typedef vector<int> bigint;
```

Các phép toán trên số nguyên lớn được cài đặt theo cách tính toán bằng tay thông thường. Dưới đây ta xét một số phép toán cơ bản đối với số nguyên lớn không dấu.

Phép cộng 2 số nguyên lớn

```
bigint operator + (bigint a, bigint b)
{
    bigint res;
    int i = 0, j = 0, cr = 0;
    while (i < a.size() || j < b.size()) {
        if (i < a.size())
            cr = cr + a[i++];
        if (j < b.size())
            cr = cr + b[j++];

        res.push_back(cr%10);
        cr = cr / 10;
    }
    if (cr > 0) res.push_back(cr);
    return res;
}
```

Phép trừ 2 số nguyên lớn

```
//điều kiện a >= b
bigint operator - (bigint a, bigint b)
{
    bigint res;
    int i = 0, j = 0, cr = 0;
    while (i < a.size() || j < b.size())
    {
        int c = a[i++] - cr;
        if (j < b.size()) c = c - b[j++];
        if (c < 0)
        {
            c = c + 10;
            cr = 1;
        }
        else cr = 0;
        res.push_back(c);
    }
    while (res.back() == 0) res.pop_back(); //Loại bỏ chữ số 0 vô nghĩa
    return res;
}
```

Phép nhân nguyên số lớn với số nguyên có 1 chữ số

```
bigint operator * (bigint a, int b)
{
    bigint res;
    int cr = 0;
    for (int i = 0; i < a.size(); ++i)
    {
        cr = cr + a[i]*b;
        res.push_back(cr % 10);
        cr = cr / 10;
    }
    if (cr > 0) res.push_back(cr);
    return res;
}
```

Phép nhân 2 số nguyên lớn

```
//hàm cho kết quả của a*10^n
bigint mul10(bigint a, int n)
{
    bigint res(a.size()+n, 0);
```

```
    for (int i = 0, j = n; i < a.size(); ++i, ++j)
        res[j] = a[i];
    return res;
}

bigint operator * (bigint a, bigint b)
{
    bigint res;
    for (int i = 0; i < b.size(); ++i)
    {
        bigint c = a*b[i];
        c = mul10(c, i);
        res = res + c;
    }
    return res;
}
```