Beautiful Subsequence

Advanced Data Structure and Algorithm Analysis
Research Project 3

Author: XXX，XXX，XXX

Date:2020-04-23

Catalogue

Chapter 1: Introduction

1.1 Problem Description

In this project, we defined a sequence (with at least 2 elements) as “Beautiful Sequence” if we are given an integer *M* and the sequence contains 2 consecutive elements with difference no larger than *M*.

And Consider that a sequence has many subsequences. When given the actual composition of a sequence with *N* elements and a positive number *M*, our mission is to find the total number of beautiful subsequences that the sequence obtains.

It’s important to remember that since the answer might be too large, our result is supposed to be output after the computation of modulo 1000000007.

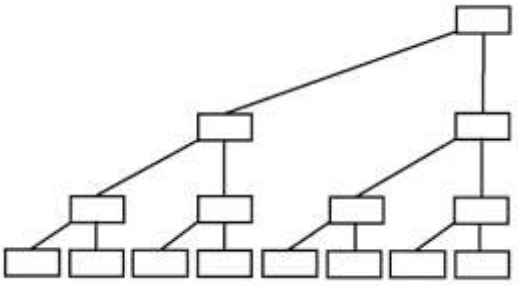
To complete this project, we should design algorithm to finish this problem and test our algorithm under different situation, including but not limited to the bonding situation and the big data situation.

1.2 Background of Data Structures

In the implementation of the algorithm, the **Binary Indexed Tree** (short for BIT) is used.

Binary Indexed Tree

Binary indexed tree is a data structure that can efficiently update elements and calculate prefix sums in a table of numbers. This structure was proposed by Boris Ryabko in 1989with a further modification published in 1992. It has subsequently became known under the name Fenwick tree after Peter Fenwick who has described this structure in his 1994 paper.



The Binary Indexed Tree achieves a much better balance between two operations: element update and prefix sum calculation when compared with a flat array of numbers. If we normally store the numbers, computing prefix sums and updating the array elements requires linear time. However, binary index tree allows both operations to be performed in a $O(logn)$ amortized time. This is achieved by representing the numbers as a tree, where the value of each node is the sum of the numbers in that subtree.

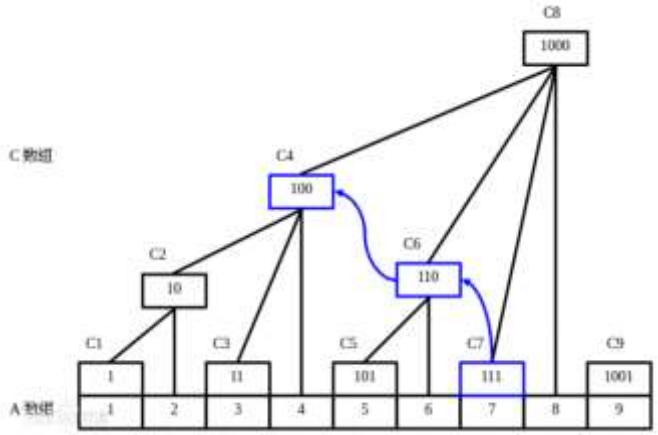
The main operations to set up the binary index tree is: *getsum, update* and *lowbit,* which will be detailed described later.

Chapter 2: Algorithm Specification

2.1 Implementation of BIT

For an array, if we need to get the sum of the prefixes of number indexed $1 \rightarrow m$, we can start from element 1 to m and sum the number simply. for n operations, the time complexity is $O(n^2)$. For value modification, we can directly find the number to be modified through indexing. The time complexity of n operations is $O(n)$. When the array size n is relatively large, the efficiency of prefix sum is low.

As shown in the figure, to accelerate the procedure, for an array of length n, array A stores the initial value of the array, and introduces an auxiliary array C (we build the binary index tree through array C).



We store the prefix sum in array C. As examples following:

C1 = A1
C2 = C1 + A2 = A1 + A2
C3 = A3
C4 = C2 + C3 + A4 = A1 + A2 + A3 + A4
C5 = A5
C6 = C5 + A6 = A5 + A6
C7 = A7
C8 = C4 + C6 + C7 + A8 = A1 + A2 + A3 + A4 + A5 + A6 + A7 + A8

Now, to get the prefix sum of a given number m, we can complete this operation in a clever way: for the m of the query, after converting it to binary, we consecutively operate -1 on the bit position of the last 1 of the binary string until number is 0. The prefix sum is the sum of *C[m]* where C is the binary index tree and m is the result every time. we get from the above procedure.

The pseudo code is as following:

Here the *lowbit* function is to get the position of the last 1 in the binary string.

```
1.  // C is the binary index tree storing the prefix sum.
2.  // m is the index to be checked.
3.  lowbit( m ){
4.      return m & (-m);
5.  }
6.
7.  getSum( C, m ){
8.      ans := 0;
9.      while m do
10.         ans += C[m];
11.         m -= lowbit(m);
12.     end
13.     return sum;
14. }
```

The other operation about BIT is to update the value in array C. We can inversely design the algorithm: to update the value of position X, we convert X to binary, and continue to add 1 to the position of the last 1 of binary until the maximum value n of array subscript is reached. The pseudo code is as following:

```
1.  // A is the origin array.
2.  // C is the binary index tree storing the prefix sum.
3.  // X is the index to be checked.
4.  // value is to be update.
5.  lowbit( m ){
6.      return m & (-m);
7.  }
8.
9.  update( X, value ){
10.     A[x] += value;
11.     while x <= n do
12.         C[x] += value;
13.         x += lowbit(x);
14.     end
```

```
15. }
```

2.2 Algorithm of Beautiful Subsequence

First we consider the opposite of thinking. It's difficult to obtain the number of subsequence that has a neighbor with difference less than m directly. However, it's easier to obtain the number of subsequence that every neighbors have a difference bigger than m. So we can define such a subsequence as "Ugly Subsequence". So that:

$$Result = NUM(Subsequece) - NUM(Ugly\ \ Subsequence)$$

We use a dynamic programming to obtain the number of ugly subsequences.

At the beginning, we sort the array in ascending way into a *inc* array. For each number X in the original array, we first find the bond index *upper* and *lower* that fit the formula in the *inc* array:

$$inc[lower] \leq X - m < X + m \leq inc[upper]$$

Here we use a binary search algorithm to maintain the time complexity in $O(logn)$.

Then the dynamic programming will be run n times. In each time, we record the number of subsequences that is suit for our conditions when adding a new element into consideration using the binary index tree. Also, we update the binary index tree as well.

The dynamic programming formula can be written as:

$$\begin{cases} dp[0] = 0, & i = 0 \\ dp[i] = dp[i-1] + tip, & 0 < i \leq n \end{cases}$$

where tip is the newly added number of suitable subsequences in each step when adding a new element into consideration.

At last we calculate the total number of subsequences with 2 or more than 2 elements according to the original sequence as:

$$Total\ \ number = 2^n - n - 1$$

Where n is the number of elements. We use the total number to minus the number of ugly subsequences and get the result.

The pseudo code is as following:

```
1. // A is the origin array.
2. // inc is the array sorted into a ascending way.
3. // m is the given number.
4. // n is the number of elements in array.
```

```
// mod is the given number for modulo.
// dp is the array for dynamic programming.
BeautifulSubsequence( A, m, n ){
    inc := sort(A);
    for i←0 to n do
        self:= find(A[i])
        upper := find(A[i] + m);
        lower := find(A[i] - m - 1);
        // find is a function to find the position in inc array.
        tip := getSum(n) - getSum(upper) + getSum(lower);
        // getSum is the operation in BIT.
        tip := (tip % mod + mod) % mod;
        // Project demand that the result should modulo a given number.
        update(self, tip + 1);
        // update is the operation in BIT. Write into the BIT
        dp[i] := (dp[i-1] + tip) % mod;
    end
    return N - dp[n]
    // N is the number of all subsequences with 2 or more elements.
}
```

Chapter 3: Testing Results

3.1 Testing environment:

Compiled under Window10, 64bits, by CodeBlocks

3.2 Testing Example and Results:

Test	Input	Output
1	10 10 11 55 44 77 22 88 33 66 99 110	0
2	20 1000 11 22 33 44 55 66 77 88 99 100 111 222 333 444 555 666 777 888 999 1000	1048555
3	6 3 1 5 4 7 2 8	48
4	30 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	1073741793
5	30 3000 23911 9079 50847 91041 14927 96209 33216 51294 52601 51921 79793 43877 31191 92681 82961 30245 79862 92561 34967 21181 347 16801 97218 88401 14609 3416 16841 67541 20111 90118	702286650

Captures of testing results are as follows:

No.1

No.2

```
20 1000
11 22 33 44 55 66 77 88 99 100 111 222 333 444 555 666 777 888 999 1000
1048555
```

No.3

```
6 3
1 5 4 7 2 8
48
```

No.4

```
30 2
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1073741793

Process returned 0 (0x0)   execution time : 6.439 s
Press any key to continue.
```

No.5

```
30 3000
23911 9079 50847 91041 14927 96209 33216 51294 52601 51921 79793 43877 31
47 16801 97218 88401 14609 3416 16841 67541 20111 90118
702286650
```

3.2 Testing Analysis:

1. The differ between the two data in group 1 is at least 11, and the result should be 0;
2. The differ between the two data in group 2 is less than 1000, the result should be the number of subsets minus the number of empty sets and the number of elementary subsets, 220-21 = 1048555;
3. In the third group, 64 subsets of data are removed from element prime subsets and empty subsets, and 57 subsets are analyzed one by one. So the result is 48;
4. The 4th group are consisted of 3000 1. The result should be 1073741793
5. The 5th group is generated randomly, which is too large for theoretical analysis; All data in group 4 are 1, and the result should be 702286650;
6. When the number of elements increasing and n >= 40, the program will overflow. So bigger data is not tested here.

Chapter 4: Analysis and Comments

4.1 Result Analysis

It can be concluded form the result that our algorithm can successfully get the result of problem.

And through the increasing of data amount, the running time increasing as well. We can obtain time complexity as $O(logn)$.

4.2 Complexity Analysis

Now we theoretically analyze the time and space complexity of our algorithm.

In the index part we use binary search and get the position of a certain number in the inc array which cost $O(logn)$. And the dynamic programming need n times to loop. So the total time complexity is $O(nlogn)$**,** where n is the scale of the problem.

Since we use an array of n position to store the original array, an array of n position to compute dynamic programming, and an array to set up the binary index tree as well, the total space complexity is $O(n)$, where n is the scale of the problem.

Appendix: Source Code (in C++)

```cpp
#include<iostream>
#include<cstdio>
#include<cstring>
#include<cmath>
#include<algorithm>
#define lowbit(x) x&(-x)
using namespace std;

const int mod = 1e9 + 7;
const int maxn = 1e5 + 10;
int n,m;
int result;
int dp[maxn];
int inc[maxn];
int tree[maxn];
int a[maxn];

int count(int x)
{
    return pow(2, x) - x - 1;
}

int flag(int x)
{
  int min = 1;
  int max = n;
  int mid;
  while(min <= max){
    mid = (min + max) / 2;
    if(inc[mid] > x) max = mid - 1;
    else  min = mid + 1;
  }
  return min - 1;
}
```

```
35.
36. void update(int x, int dx)
37. {
38.     while(x < maxn)
39.     {
40.         tree[x] = (tree[x] + dx) % mod;
41.         x += lowbit(x);
42.     }
43. }
44.
45. int getsum(int x)
46. {
47.     int sum = 0;
48.     while(x > 0)
49.     {
50.         sum = (sum + tree[x]) % mod;
51.         x -= lowbit(x);
52.     }
53.     return sum;
54. }
55.
56. int main()
57. {
58.     int i;
59.     int self, upper, lower;
60.     int tip;
61.     cin >> n >> m;
62.     for(i = 1;i <= n;i++){
63.         cin >> a[i];
64.         inc[i] = a[i];
65.     }
66.     sort(inc + 1, inc + 1 + n);
67.     dp[0] = 0;
68.     for(i = 1;i <= n;i++){
69.         self = flag(a[i]);
70.         upper = flag(a[i] + m);
```

```
        lower = flag(a[i] - m - 1);
        tip = getsum(n) - getsum(upper) + getsum(lower);
        tip = (tip % mod + mod) % mod;
        update(self, tip + 1);
        dp[i] = (dp[i-1] + tip) % mod;
    }
    result = count(n) - dp[n];
    cout << result << endl;
    return 0;
}
```

Declaration

We hereby declare that all the work done in this project titled Beautiful Subsequence is of our independent effort as a group.

Duty Assignments:

Programmer: XXX

Tester: XXX

Report Writer: XXX