Project 3 Beautiful Subsequence

Group 24

Date: 2020-04-21

Project 3 Beautiful Subsequence

1. Introduction

1.1 Description

Given a sequence S, we are required to find the number of all subsequences which contain 2 neighbors with difference no larger than M, where M is an input integer.

We are going to use *dynamic programming* ro solve this problem.

2. Algorithm Specification

2.1 Data Structure Specification

- `input[MAX]`

  The array to store the input sequence.

- `dp[MAX]`

Before diving into the algorithm, let's look at the data structure we use to perform dynamic programming.

`dp[]` is an array that store the state information. Specifically, `dp[i]` is the number of subsequences ending at index i such that `input[i]` is the last element of the subsequence.

2.2 Algorithm Specification

- State Definition

  As mentioned above, we define the state as `dp[i]`, which indicate the number of subsequences ending at index i such that `input[i]` is the last element of the subsequence. And the final result would be $\sum_i dp[i]$.

- State Conversion Equation

  To compute `dp[i]`, we need to traverse each j $(0 \leq j < i)$. For each j, we're going to find the number of *beautiful* subsequences consist of element whose indices can only be chosen from the set $\{0, 1, \ldots, j, i\}$. And `dp[i]` would be the sum of all the j.

  For each j, we can divide this into two conditions

  - $abs(input[i] - input[j]) \leq M$

    In this case, the i^{th} and j^{th} elements have already satisfying the definition of *beautiful*, thus any number whose index in $\{0, 1, \ldots, j-1\}$ can be chosen.So we can get $dp[i] = dp[i] + 2^j$

  - $otherwise$

    Otherwise, for each valid case in `dp[j]`, we can append `input[i]` to the end of the subsequence. So the recurrence would be $dp[i] = dp[i] + dp[j]$

  As a result, state conversion equation would be

$$dp[i] = \begin{cases} dp[i] + 2^j & \text{abs(input[i]-input[j])} \leq M \\ dp[i] + dp[j] & \text{otherwise} \end{cases}$$

  - Base case: $dp[i] = 0$ for each i

- Aftering acquiring the recurrence relation, it is easy to implement the code.

```
for(int i = 0; i < N; i++)  // initialization the base case
    dp[i] = 0;
```

```
for(int i = 1; i < N; i++)
    for(int j = 0; j < i; j++){
        if(abs(input[i] - input[j]) <= M)    // case1
            dp[i] += pow(2, j);
        else                                // case2
            dp[i] += dp[j];
    }

for(int i = 0, result = 0; i < N; i++){     // get the result
    result += dp[i];
    result %= DIVISOR;
}
```

3. Testing Results

- Sample

  input:

  > 4 2
  >
  > 5 3 8 6

  output:

  > 8

- Minimum case

  input:

  > 2 3
  >
  > 1 7

  output:

  > 0

- Min N

  input:

  > 0 2

  output:

  > 0

- Max N

  input:

  > 100001 0
  >
  > 0 1 2 3 4 5 ... 100000

  output:

  > 0

4. Analysis and Comments

4.1 Time Complexity

The initialization and finding the result takes $O(N)$. The dynamic programming process takes $O(1+2+\cdots+N)=O(N^2)$. As a result, this algorithm takes $O(N^2)$ time.

4.2 Space Complexity

We only use two 1-D arrays, so the space complexity is $O(N)$.

4.3 Comments

- At my first thought, I wanted to use the bitmask to implement the DP, but the input size may be too big(we need 10^5 bits in the maximum case, which is not feasible), so we turn to another approach.

5. Author List

Programmer: WangRui

Tester: LiYalin

Writer: LiYalin OuyangHaodong

6. Declaration

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

7. Appendix

7.1 Source code in C

> You can open the code.c in IDE for a better view

```c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

#define MAX 100001
#define DIVISOR 1000000007

int main()
{
    int N, M;
    int i, j, input[MAX], dp[MAX];
    long count = 0;

    scanf("%d %d", &N, &M);
    for (i = 0; i < N; i++) {
        scanf("%d", &input[i]);
        dp[i] = 0;                          // initialize the dp[] array
```

```
    }

    for (i = 1; i < N; i++) {
        for (j = 0; j < i; j++) {
            if (abs(input[i] - input[j]) <= M) // case1: already valid
                dp[i] += 1<<j;
            else                                // case2: otherwise
                dp[i] += dp[j];
        }
    }

    for (i = 0; i < N; i++) {                   // get the result
        count += dp[i];
        count %= DIVISOR;
    }

    printf("%ld\n", count);

    return 0;
}
```