Best peak shape

3150102418 张倬豪

2017年5月12日

Context

1 Introduction

1.1 Concepts Description

1.1.1 Dynamic Programming

To define dynamic programming, we need to know that it is applied in multiple fields such as computer science, mathematics, management science, economics and bioinformatics, dynamic programming (also known as dynamic optimization) is a method for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions. The next time the same subproblem occurs, instead of recomputing its solution, one simply looks up the previously computed solution, thereby saving computation time at the expense of a (hopefully) modest expenditure in storage space. (Each of the subproblem solutions is indexed in some way, typically based on the values of its input parameters, so as to facilitate its lookup.) The technique of storing solutions to subproblems instead of recomputing them is called "memoization".

Dynamic programming algorithms are often used for optimization. A dynamic programming algorithm will examine the previously solved subproblems and will combine their solutions to give the best solution for the given problem. In comparison, a greedy algorithm treats the solution as some sequence of steps and picks the locally optimal choice at each step. Using a greedy algorithm does not guarantee an optimal solution, because picking locally optimal choices may result in a bad global solution, but it is often faster to calculate. Some greedy algorithms (such as Kruskal's or Prim's for minimum spanning trees) are however proven to lead to the optimal solution.

1.1.2 Longest Increasing Subsequence

In computer science, the longest increasing subsequence problem is to find a subsequence of a given sequence in which the subsequence's elements are in sorted order, lowest to highest, and in which the subsequence is as long as possible. This subsequence is not necessarily contiguous, or unique. Longest increasing subsequences are studied in the context of various disciplines related to mathematics, including algorithmics, random matrix theory, representation theory, and physics. The longest increasing subsequence problem is solvable in time $O(nlogn)$ by dynamic programming, where n denotes the length of the input sequence.

1.2 Project Description

This project specification is simple and clear. Given a sequence of numbers we need to find the longest and most symmetric subsequence that is increasing at some point i and decreasing since then.

PS: "symmetric" means the difference of the lengths of the increasing and the decreasing sub-sequences is minimized

- Input

For each case, input gives an integer N (3 <= N <= 104). Then N integers are given in the next line, separated by spaces. All the integers are in [-10000, 10000].

- Output

Print in a line the length of the best peak shape.

2 Algorithm Specification

2.1 Solution 1 (Quadratic complexity Algorithm)

2.1.1 Data Structure

In this project, data structure is rather simple and clear. It's algorithm that matters. So no more extra explanation will be stated. The comments below is clear enough.

```
1 int n, a[maxn]; // Store data
2 int res1[maxn] = {0}, res2[maxn] = {0}; // Store each side's longest
   ↪ increasing subsequence of each node
3 int max = 0, index = 0, sym; // Temp variable to find the best peak
```

2.1.2 Algorithm and Pseudo Code

This algorithm is easy to understand.

1. Get every position's LIS(longest increasing subsequence) and inverted LIS

For each position i, the LIS from the beginning to this node is represented as L_i. We can easily get this state transition equation:

$$L_i = max\{L_j + 1 | 0 \leq j < i, a[i] > a[j]\}$$

Understand this equation is easy, for every position i, L_i is the maximum of all the former position's $L_j + 1$ while $a[j]$ is less than $a[i]$. The time complexity can be easily understood as quadratic, the proof is as simple as bubble sort.

Because we are to find the best peak, one side of LIS is not enough. We need to find the inverted LIS like going down as well as the going up. We have this equation:

$$L_i = max\{L_j + 1 | i < j < n, a[i] > a[j]\}$$

```
1   res1[0] = res2[n - 1] j ← 0; // Initialize
2
3   for i ← 1 to n -1 do begin
4       for j ← 0 to i - 1 do begin
5           if a[j] < a[i] and res1[j] + 1 > res1[i] begin
6               res1[i] ← res1[j] + 1;
7           end
8       end
9   end
10
11  for i ← n - 2 downto 0 do begin
12      for j ← n - 1 downto i + 1 do begin
13          if a[j] < a[i] and res2[j] + 1 > res2[i] then begin
14              res2[i] ← res2[j] + 1;
15          end
16      end
17  end
```

2. Find the best peak according to the above results.

This is also simple. We only need to travel the sequence in linear time once. We use variables max, $index$, sym to store the attributes of the current best peak. When we find a better one, we renew them. One thing we need to pay attention to is we need to leave out the positions where $res1[i]$ or $res2[i]$ equals 0, because they are not a peak at all. The are strictly increasing or decreasing.

```
1   sym = n - 1; // The maximum possible value of sym
2   for i ← 1 to n - 1 do begin
```

```
        if res1[i] and res2[i] are positive, res1[i] + res2[i] > max or
    ↪  res1[i] + res2[i] = max and sym > abs(res1[i] - res2[i]) then begin
            max  res1[i] + res2[i];
            sym ← abs(res1[i] - res2[i]);
            index ← i;
        end
    end
```

2.1.3 Proof of correctness

We can understand this algorithm in a direct way like mathematical induction. For each position, the former position's LIS has been all calculated correctly. So we can get the new position's LIS by travelling the previous numbers and find the longest LIS. The further test on PTA can also attribute to the correctness of this algorithm.

2.2 Solution 2 (NlogN complexity Algorithm)

2.2.1 Data Structure

```
int n, a[maxn], d1[maxn], d2[maxn], i, len1 = 0, len2 = 0, newIndex; // d1,
 ↪  d2, len1 and len2 are stated below.
int res1[maxn] = {0}, res2[maxn] = {0}, max = 0, index 0, sym;
```

2.2.2 Algorithm and Pseudo Code

This is a much more efficient algorithm. The time complexity can reach to $O(NlogN)$. To do this we need to alloc new arrays and use binary search algorithm. We use array $d[]$. $d[len]$ means the last and least element of LIS whose length is len. 1. First, $len \leftarrow 0$ and $d[len] = a[0]$ 2. Then, we need to calculate as this way:

For each $a[i]$, When $a[i] > d[len]$, we have $d[len++] = a[i]$. Otherwise, we need to locate j, which satisfies $d[j-1] < a[i] < d[j]$, and renew $d[j]$ to $a[i]$.

At the same time, we need to store res1[i] as well to find the best peak. When $a[i] > d[len]$, we have $res[i] = len$. Otherwise, we need to locate $j(newIndex)$, which satisfies $d[j-1] < a[i] < d[j]$, and $res[i] = j$.

Now we get the same $res[]$ as solution 1. The following steps are exactly the same as solution 1.

Now we can see, for every position i, we may need to perform a binary search, which can be done in the following pseudo code:

```
int find(int d[], int start, int end, int element) {//find the first pos >=
 ↪ element  guarantee d[end] >= element
    if d[start] >= element then return start;
    while start < end - 1 begin
        int mid ← (start + end) / 2;
        if (d[mid] < element)
            start ← mid;
        else
            end ← mid;
    end
    return end
}
```

The major algorithm can be represented as:

```
    d1[len1] ← a[0];
    for i ← 1 to n - 1 do begin
        if a[i] > d1[len1] then begin
            len1++;
            d1[len1] ← a[i];
            res1[i] ← len1;
        end
        else begin
            newIndex ← find(d1, 0, len1, a[i]);
            if newIndex <> -1 then begin
                d1[newIndex] ← a[i];
                res1[i] ← newIndex;
            end
            else res1[i] ← 0;
        end
    end

    d2[len2] ← a[n - 1];
    for i ← n - 2 downto 0 do begin
        if a[i] > d2[len2] then begin
            len2++;
            d2[len2] ← a[i];
            res2[i] ← len2;
```

```
24          end
25          else end
26              newIndex ← find(d2, 0, len2, a[i]);
27              if newIndex <> -1 begin
28                  d2[newIndex] ← a[i];
29                  res2[i] ← newIndex;
30              end
31              else res2[i] ← 0;
32          end
33      end
```

2.2.3 Proof of correctness

We can understand this algorithm in a direct way like mathematical induction, too. The generation of $res[]$ is nearly the same as solution 1 except for the binary search optimization.

But, in this algorithm we need to know that there is a difference between LIS in range (0, i) and LIS in range (0, i) while ends up with i. For example, sequence {1, 2, 3, 4, 3} has res[4] = 4, but we need to get LIS that ends with a[4], which is 2. In this problem, 2 is clearly the answer.

The further test on PTA can also attribute to the correctness of this algorithm.

3 Testing results

3.1 Testing cases

1. Test 1

n	20
a	1 3 0 8 5 -2 29 20 20 4 10 4 7 25 18 6 17 16 2 -1

2. Test 2

n	5
a	-1 3 8 10 20

3. Test 3

n	3
a	1 1 1

4. Test 4(Leave out 10000 numbers, source file in the folder)

n	10000
a	.....

3.2 Results

1. Test 1

   10 14 25

2. Test 2

   No peak shape

3. Test 3

   No peak shape

4. Test 4(Leave out 10000 numbers)

   273 5295 9969

   PS: As shown in the picture below, my program passed all the tests in PTA.

4 Analysis and comments

4.1 Time complexity

Time complexity analysis of this project is quite easy. For solution 1, it is clear that for each number, $O(n)$ time will be required to search the LIS and renew it. Thus, the whole time complexity is $O(n^2)$. The process to find the best peak is linear time. So it is clear that whole time is $O(n)$.

评测结果

时间	结果	得分	题目	编译器	用时（ms）	内存（MB）	用户
2017-05-11 15:31	答案正确	35	5-1	gcc	449	2	zju3150102418

测试点结果

测试点	结果	得分/满分	用时（ms）	内存（MB）
测试点1	答案正确	19/19	19	1
测试点2	答案正确	1/1	3	1
测试点3	答案正确	1/1	139	1
测试点4	答案正确	1/1	130	1
测试点5	答案正确	2/2	8	1
测试点6	答案正确	10/10	449	2
测试点7	答案正确	1/1	3	1

评测结果

时间	结果	得分	题目	编译器	用时（ms）	内存（MB）	用户
2017-05-03 15:04	答案正确	35	5-1	gcc	14	2	zju3150102418

测试点结果

测试点	结果	得分/满分	用时（ms）	内存（MB）
测试点1	答案正确	19/19	7	1
测试点2	答案正确	1/1	2	1
测试点3	答案正确	1/1	4	2
测试点4	答案正确	1/1	5	1
测试点5	答案正确	2/2	14	1
测试点6	答案正确	10/10	5	1
测试点7	答案正确	1/1	2	1

For solution 2, it is also clear that for each number, %O(logn)$ time will be needed to complete the binary search. The rest of the analysis is the same as solution 1. So the whole time complexity is $O(nlogn)$.

4.2 Space complexity

It's too simple. Clearly they are all $O(n)$.

4.3 Comments

These two algorithms both have their advantages and disadvantages. Solution 1 is easy to understand and implement, but the time cost is a little high. Solution 2 is more difficult, but more efficient.

Actually, $O(nlogn)$ algorithm has other ways to implement. Binary search is only one way. We can also implement it in Binary Indexed Tree(BIT), which can be implemented in the further study.

5 Appendix

5.1 Source Code

5.1.1 Solution 1

```
1  //
2  //  main.c
3  //  adspj4
4  //
5  //  Created by 张倬豪 on 2017/5/3.
6  //  Copyright © 2017年 Icarus. All rights reserved.
7  //
8
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <math.h>
12
13 #define maxn 100010
14
15 int find(int d[], int start, int end, int element) { //find the first pos >=
    ↪ element  guarantee d[end] >= element
```

```
16      if (d[start] >= element) return start; //if already bigger than element
 ↪  we want to find
17      while (start < end - 1) {
18          int mid = (start + end) >> 1; //get the middle position
19          if (d[mid] < element) start = mid;
20          else end = mid;
21      }
22      return end;
23  }
24
25  int main(int argc, const char * argv[]) {
26      int n, a[maxn], d1[maxn], d2[maxn], i, len1 = 0, len2 = 0, newIndex;
 ↪  //d1, d2 means last and least element of LIS whose length is len.
27      int res1[maxn] = {0}, res2[maxn] = {0}, max = 0, index = 0, sym;
28      scanf("%d", &n);
29      for (i = 0; i < n; i++) {
30          scanf("%d", &a[i]);
31      }
32      d1[len1] = a[0]; //initialize
33      for (i = 1; i < n; i++) {
34          if (a[i] > d1[len1]) { //when a[i] > d[len]
35              len1++; //new element in d
36              d1[len1] = a[i];
37              res1[i] = len1; //new element in res
38          }
39          else { //when a[i] <= d[len]
40              newIndex = find(d1, 0, len1, a[i]); //get the new index where
 ↪  d[j-1] < a[i] < d[j]
41              if (newIndex != -1) { //if it exists
42                  d1[newIndex] = a[i]; //renew d[newIndex] to a[i]
43                  res1[i] = newIndex;
44              }
45              else res1[i] = 0;
46          }
47      }
48
49      d2[len2] = a[n - 1]; //exactly the same but inverted
50      for (i = n - 2; i >= 0; i--) {
51          if (a[i] > d2[len2]) {
52              len2++;
53              d2[len2] = a[i];
```

```
            res2[i] = len2;
        }
        else {
            newIndex = find(d2, 0, len2, a[i]);
            if (newIndex != -1) {
                d2[newIndex] = a[i];
                res2[i] = newIndex;
            }
            else res2[i] = 0;
        }
    }
    sym = n - 1;
    for (i = 1; i < n; i++) {
        if (res1[i] > 0 && res2[i] >0 && (((res1[i] + res2[i]) > max) ||
 ↪  ((res1[i] + res2[i] == max) && (abs(res1[i] - res2[i]) <= sym)))) { //if
 ↪  new peak is better
            max = res1[i] + res2[i]; //renew the current best one
            sym = abs(res1[i] - res2[i]);
            index = i;
        }
    }
    if (max) printf("%d %d %d", max + 1, index + 1, a[index]); //don't forget
 ↪  to plus 1 because it didn't count itself
    else printf("No peak shape");
    return 0;
}
```

5.1.2 Solution 2

```
//
//  main.c
//  adspj4
//
//  Created by 张倬豪 on 2017/5/3.
//  Copyright © 2017年 Icarus. All rights reserved.
//

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

```

```
#define maxn 100010

int main(int argc, const char * argv[]) {
    int n, a[maxn], d1[maxn], d2[maxn], i, j, len1 = 0, len2 = 0, newIndex;
    //d1, d2 means last and least element of LIS whose length is len.
    int res1[maxn] = {0}, res2[maxn] = {0}, max = 0, index = 0, sym;
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d", &a[i]);
    }
    res1[0] = res2[n - 1] = 0;

    for (i = 1; i < n; i++) {
        for (j = 0; j < i; j++) { //for each number search former numbers
            if (a[j] < a[i] && res1[j] + 1 > res1[i]) { //if longer
    subsequence is found
                res1[i] = res1[j] + 1; //renew it
            }
        }
    }

    for (i = n - 2; i >= 0; i--) {
        for (j = n - 1; j > i; j--) { //exactly the same but inverted
            if (a[j] < a[i] && res2[j] + 1 > res2[i]) {
                res2[i] = res2[j] + 1;
            }
        }
    }

    sym = n - 1;
    for (i = 1; i < n; i++) {
        if (res1[i] > 0 && res2[i] >0 && (((res1[i] + res2[i]) > max) ||
    ((res1[i] + res2[i] == max) && (abs(res1[i] - res2[i]) <= sym)))) { //if
    new peak is better
            max = res1[i] + res2[i]; //renew the current best one
            sym = abs(res1[i] - res2[i]);
            index = i;
        }
    }
    if (max) printf("%d %d %d", max + 1, index + 1, a[index]); //don't forget
    to plus 1 because it didn't count itself
```

```
49      else printf("No peak shape");
50      return 0;
51  }
```

5.2 Reference

1. Wikipedia - Dynamic Programming
2. Wikipedia - Longest Increasing Subsequence

5.3 Declaration

We hereby declare that all the work done in this project titled "Binary Search Trees" is of my independent effort.