CS222 Homework 1

Exercises for Algorithm Design and Analysis by Li Jiang, 2016 Autumn Semester

5140309507 林禹臣 yuchenlin@sjtu.edu.cn

1. Given an array of non-negative integers, you are initially positioned at the first index of the array.

   Each element in the array represents your maximum jump length at that position.

   Determine if you are able to reach the last index.

   For example:

   A = [2,3,1,1,4], return true.

   A = [3,2,1,0,4], return false.

   Input:

   int A[]: the input array.

   int N: length of A.

   Output:

   return true or false.

   Solution. I have two solutions here (Algorithm 1 and Algorithm 2) and their time complexity are both $O(n)$.

   The first one [1] is based on the greedy idea that we jump to the most promising position step by step.

   To illustrate this idea, I would like to introduce the function I used to measure the ability of every index, which is defined as follows:

   $$Ability(index) = index + A[index]$$

   The ability value of a index just means the farthest position it can reach. Thus, every position has a value to represent its ability to reach the end. When we need to decide where to go, we can just simply jump forward to the one with largest ability value.

   About Correctness

   To evaluate the correctness of this algorithm, we just need to check if it is correct when it returns true or false respectively.

   If all possible next positions of a certain position have smaller or equal ability value, we call it a *Stop*. For example, if A is [3,2,0,0...], then we have the ability array as [3,3,2,3,...]. The first position's all next possible positions have smaller or equal ability value as its, and thus it is a *Stop* according to my definition.

   If there is a sequence of increasing indexes and the ability of the last one is larger than the end, we call it a *Correct Path*. (Note: This path does not include the end.)

   Proposition 1: There is no *Stop* in all Correct Paths.

   Now, let us prove that there is no *Stop* in all Correct Paths. By contradiction, if there is a path that contains such an *Stop* position whose index is i. Say, its last successor is $j = i + A[i]$.

Because i is a *Stop*, we have $i + A[i] \geq j + A[j]$. We can conclude that and $A[j] = 0$. We know that the next position k on this Correct Path must locate between $i + 1$ and j. Due to the fact that A is a *Stop*, we have $k + A[k] <= i + A[i]$. If k is j, it has no next position because its ability is just k, which cannot satisfy the definition of the positions on Correct Path, because $ability(k) = k + 0 = k < end$. If k is not j, we will find that any possible next position of k has the same situation like k, so we can transfer k to j. Proposition 1 is correct.

Therefore, our algorithm is correct when it returns $false$.

Proposition 2: The path that Algorithm 1 gets is a shortest Correct Path.

By contradiction, if we have a shorter path than the path Algorithm 1 gives, there must be a time when it did not choose the next possible position which has the largest ability. Thus, it must have a successor which have larger ability than original path. However, it is impossible. Because our algorithm compares all the positions and always choose the one with largest ability.

Therefore, our algorithm is correct when it returns $true$.

About Time Complexity

If we compare the ability of every next possible positions, the time complexity of this algorithm will not be $O(n)$ for each position may have some same successions and we do some repeating work. Fortunately, we can just search from the last most able index every time. It is because the ability value on the correct path must be strictly increasing.

Therefore, this algorithm can be executed in $O(n)$ time.

Algorithm 1: Jump-Forward-Greedy Algorithm

```
   Input: int A[],int N
   Output: bool Abeled
 1 end = N − 1;
 2 currentIndex = 0;
 3 currentAbility = 0 + A[0];
 4 lastAbility = 0;
 5 while cuurentAbility < end do
 6 |   nextIndex = currentIndex;
 7 |   nextAbility = currentAbility;
 8 |   for each i ∈ [lastAbility + 1, currentAbility] do
 9 |   |   if A[i] + i > nextAbility then
10 |   |   |   nextIndex = i;
11 |   |   |   nextAbility = A[i] + i;
12 |   lastAbility = currentAbility;
13 |   if nextIndex > currentIndex then
14 |   |   currentIndex = nextIndex;
15 |   |   currentAbility = nextAbility;
16 |   else
17 |   |   return false;
18 return true;
```

The second solution[2] is based on the similar idea of the first one. The only difference is that we do not jump anymore. To illustrate how it works, let us define something. A *scope* means

a sequence of continuous indexes from 0 such as $[0, 1, 2, 3, ..., n]$. A *scope* is reachable when it can jump several times to reach its end.

In this algorithm, we just keep narrowing a reachable *scope* to a smaller reachable *scope*. If the *scope* [0] is reachable, then we return *true*. Otherwise, we return $false$.

The method of narrowing a *scope* is to check if there is a position p in it which has the *ability* to reach end of this *scope*. If there is, then we can just narrow down the *scope* to a smaller *scope* $[0, 1, 2, ..., p]$.

The time complexity is obviously $O(n)$ and the correctness is due to its equivalence to the first solution.

The reason why this solution is more easy to implement is that it does not care about the exact shortest path from the start to the end. It is all about checking and narrowing reachable *scopes* from the end to the start, while the first algorithm is based on jumping. Although their direction are different, the time complexity of them are both $O(n)$.

Algorithm 2: Narrowing-Reachable-Scope Algorithm

Input: int $A[]$,int N
Output: bool *Abeled*

```
1 currentScopeEnd = N − 1;
2 for each i from N − 1 to 0 do
3     if A[i] + i >= currentScopeEnd then
4         currentScopeEnd = i;
5 return currentScopeEnd == 0;
```

□

2. Given an array of non-negative integers, you are initially positioned at the first index of the array.

   Each element in the array represents your maximum jump length at that position.

   Your goal is to reach the last index in the minimum number of jumps.

   For example:

   Given array A = [2,3,1,1,4]

   The minimum number of jumps to reach the last index is 2. (Jump 1 step from index 0 to 1, then 3 steps to the last index.)

   Note:

   You can assume that you can always reach the last index.

   Input:

   int A[]: the input array.

   int N: length of A.

   Output:

   return minimum number of jumps.

Solution. To solve this problem, we just need to do a little modification on Algorithm 1. The proof of *Proposition 2* tells us how to find the shortest correct path. Thus, we just need to count the number of loops and plus one (remember that the end is not included into the correct path). Evidently, its time complexity is $O(n)$ as well. See Algorithm 3.

Algorithm 3: Count Length of Shortest Correct Path

```
Input: int A[],int N
Output: int len
end = N − 1;
if end == 0 then
    return 0;
currentIndex = 0;
currentAbility = 0 + A[0];
lastAbility = 0;
len = 0;
while cuurentAbility < end do
    nextIndex = currentIndex;
    nextAbility = currentAbility;
    for each i ∈ [lastAbility + 1, currentAbility] do
        if A[i] + i > nextAbility then
            nextIndex = i;
            nextAbility = A[i] + i;
    lastAbility = currentAbility;
    currentIndex = nextIndex;
    currentAbility = nextAbility;
    len + +;
return len + 1;
```

□

3. There are N children standing in a line. Each child is assigned a rating value.

   You are giving candies to these children subjected to the following requirements:

   a. Each child must have at least one candy.

   b. Children with a higher rating get more candies than their neighbors.(assume no equal rating neighbors)

   What is the minimum candies you must give?

   Input:

   int A[]: the input array of rating values.

   int N: length of A,(number of children).

   Output:

   return minimum number of candies you must give.

Solution. My intuition of solving this problem is to find all (local) maximum values of the ratings and then use them to determine the number of candies. It is practical but I found that it is quite tricky to implement this idea into clear and readable code.

Thus, I changed my idea a little.

First of all, I give one candy to the first kid for every kid should have at least one candy.

Because we are stingy about candies, every time I give a child in this direction A one more candy than his left neighbor B when A has higher rating than B. Otherwise I just give A one candy. (This is our greedy idea.)

It may annoy many kids because some of them should have more candies for they have higher ratings than their right neighbors. Thus, I find the last but one kid and ask if he is unhappy. If he feels unfair, I give him one more candy than his right neighbor. And one by one from the opposite direction. Meanwhile, I keep the happiness of the kids already feel fair.

Finally, all kids are happy now. Obviously, the time complexity is $O(n)$.

Algorithm 4: Give Candies Forward Then Backward to Unhappy Kids

Input: int $A[]$,int N
Output: int *minimum*

```
1  candies = initiate a N-size array of int;
2  candies[0] = 1;
3  for each i from 1 to (N − 1) do
4      if A[i] > A[i − 1] then
5          candies[i] = candies[i − 1] + 1;
6      else
7          candies[i] = 1;
8  for each i from (N − 2) to 0 do
9      if A[i] > A[i + 1] and candies[i] < (candies[i + 1] + 1) then
10         candies[i] = candies[i + 1] + 1;
11 return the sum of candies;
```

□