Previous Next

656. Coin Path

Aug. 6, 2017 | 9.7K views

Average Rating: 4.46 (11 votes)

Given an array `A` (index starts at `1`) consisting of N integers: A_1, A_2, ..., A_N and an integer `B`. The integer `B` denotes that from any place (suppose the index is `i`) in the array `A`, you can jump to any one of the place in the array `A` indexed `i+1`, `i+2`, ..., `i+B` if this place can be jumped to. Also, if you step on the index `i`, you have to pay A_i coins. If A_i is -1, it means you can't jump to the place indexed `i` in the array.

Now, you start from the place indexed `1` in the array `A`, and your aim is to reach the place indexed `N` using the minimum coins. You need to return the path of indexes (starting from 1 to N) in the array you should take to get to the place indexed `N` using minimum coins.

If there are multiple paths with the same cost, return the lexicographically smallest such path.

If it's not possible to reach the place indexed N then you need to return an empty array.

Example 1:

```
Input: [1,2,4,-1,2], 2
Output: [1,3,5]
```

Example 2:

```
Input: [1,2,4,-1,2], 1
Output: []
```

Note:

1. Path Pa_1, Pa_2, ..., Pa_n is lexicographically smaller than Pb_1, Pb_2, ..., Pb_m, if and only if at the first `i` where Pa_i and Pb_i differ, Pa_i < Pb_i; when no such `i` exists, then `n` < `m`.
2. A_1 >= 0. A_2, ..., A_N (if exist) will in the range of [-1, 100].
3. Length of A is in the range of [1, 1000].
4. B is in the range of [1, 100].

Solution

Approach #1 Brute Force[Time Limit Exceeded]

In this approach, we make use of a $next$ array of size n. Here, n refers to the size of the given A array. The array $nums$ is used such that $nums[i]$ is used to store the minimum number of coins needed to jump till the end of the array A, starting from the index i.

We start by filling the $next$ array with all -1's. Then, in order to fill this $next$ array, we make use of a recursive function `jump(A, B, i, next)` which fills the $next$ array starting from the index i onwards, given A as the coins array and B as the largest jump value.

With i as the current index, we can consider every possible index from $i+1$ to $i+B$ as the next place to be jumped to. For every such next index, j, if this place can be jumped to, we determine the cost of reaching the end of the array starting from the index i, and with j as the next index jumped from i, as $A[i]+jump(A,B,j,next)$. If this cost is lesser than the minimum cost required till now, for the same starting index, we can update the minimum cost and the value of $next[i]$ as well.

For every such function call, we also need to return this minimum cost.

At the end, we traverse over the $next$ array starting from the index 1. At every step, we add the current index to the res list to be returned and also jump/move to the index pointed by $next[i]$, since this refers to the next index for the minimum cost. We continue in the same manner till we reach the end of the array A.

Java Copy

```java
public class Solution {
    public List < Integer > cheapestJump(int[] A, int B) {
        int[] next = new int[A.length];
        Arrays.fill(next, -1);
        jump(A, B, 0, next);
        List < Integer > res = new ArrayList();
        int i;
        for (i = 0; i < A.length && next[i] > 0; i = next[i])
            res.add(i + 1);
        if (i == A.length - 1 && A[i]>= 0)
            res.add(A.length);
        else
            return new ArrayList < Integer > ();
        return res;
    }
    public long jump(int[] A, int B, int i, int[] next) {
        if (i == A.length - 1 && A[i] >= 0)
            return A[i];
        long min_cost = Integer.MAX_VALUE;
        for (int j = i + 1; j <= i + B && j < A.length; j++) {
            if (A[j] >= 0) {
                long cost = A[i] + jump(A, B, j, next);
                if (cost < min_cost) {
                    min_cost = cost;
                    next[i] = j;
                }
            }
```

Complexity Analysis

- Time complexity : $O(B^n)$. The size of the recursive tree can grow upto $O(b^n)$ in the worst case. This is because, we have B possible branches at every step. Here, B refers to the limit of the largest jump and n refers to the size of the given A array.
- Space complexity : $O(n)$. The depth of the recursive tree can grow upto n. $next$ array of size n is used.

Approach #2 Using Memoization [Accepted]

Algorithm

In the recursive solution just discussed, a lot of duplicate function calls are made, since we are considering the same index through multiple paths. To remove this redundancy, we can make use of memoization.

We keep a $memo$ array, such that $memo[i]$ is used to store the minimum cost of jumps to reach the end of the array A. Whenever the value for any index is calculated once, it is stored in its appropriate location. Thus, next time whenever the same function call is made, we can return the result directly from this $memo$ array, pruning the search space to a great extent.

Java Copy

```java
public class Solution {
    public List < Integer > cheapestJump(int[] A, int B) {
        int[] next = new int[A.length];
        Arrays.fill(next, -1);
        long[] memo = new long[A.length];
        jump(A, B, 0, next, memo);
        List < Integer > res = new ArrayList();
        int i;
        for (i = 0; i < A.length && next[i] > 0; i = next[i])
            res.add(i + 1);
        if (i == A.length - 1 && A[i] >= 0)
            res.add(A.length);
        else
            return new ArrayList < Integer > ();
        return res;
    }
    public long jump(int[] A, int B, int i, int[] next, long[] memo) {
        if (memo[i] > 0)
            return memo[i];
        if (i == A.length - 1 && A[i] >= 0)
            return A[i];
        long min_cost = Integer.MAX_VALUE;
        for (int j = i + 1; j <= i + B && j < A.length; j++) {
            if (A[j] >= 0) {
                long cost = A[i] + jump(A, B, j, next, memo);
                if (cost < min_cost) {
                    min_cost = cost;
```

Complexity Analysis

- Time complexity : $O(nB)$. $memo$ array of size n is filled only once. We also do a traversal over the $next$ array, which will go upto B steps. Here, n refers to the number of nodes in the given tree.
- Space complexity : $O(n)$. The depth of the recursive tree can grow upto n. $next$ array of size n is used.

Approach #3 Using Dynamic Programming [Accepted]

Algorithm

From the solutions discussed above, we can observe that the cost of jumping till the end of the array A starting from the index i is only dependent on the elements following the index i and not the ones before it. This inspires us to make use of Dynamic Programming to solve the current problem.

We again make use of a $next$ array to store the next jump locations. We also make use of a dp with the same size as that of the given A array. $dp[i]$ is used to store the minimum cost of jumping till the end of the array A, starting from the index i. We start with the last index as the current index and proceed backwards for filling the $next$ and dp array.

With i as the current index, we consider all the next possible positions from $i+1$, $i+2$,..., $i+B$, and determine the position, j, which leads to a minimum cost of reaching the end of A, which is given by $A[i]+dp[j]$. We update $next[i]$ with this corresponding index. We also update $dp[i]$ with the minimum cost, to be used by the previous indices' cost calculations.

At the end, we again jump over the indices as per the $next$ array and put these indices in the res array to be returned.

Java Copy

```java
1  public class Solution {
2      public List < Integer > cheapestJump(int[] A, int B) {
3          int[] next = new int[A.length];
4          long[] dp = new long[A.length];
5          Arrays.fill(next, -1);
6          List < Integer > res = new ArrayList();
7          for (int i = A.length - 2; i >= 0; i--) {
8              long min_cost = Integer.MAX_VALUE;
9              for (int j = i + 1; j <= i + B && j < A.length; j++) {
10                 if (A[j] >= 0) {
11                     long cost = A[i] + dp[j];
12                     if (cost < min_cost) {
13                         min_cost = cost;
14                         next[i] = j;
15                     }
16                 }
17             }
18             dp[i] = min_cost;
19         }
20         int i;
21         for (i = 0; i < A.length && next[i] > 0; i = next[i])
22             res.add(i + 1);
23         if (i == A.length - 1 && A[i] >= 0)
24             res.add(A.length);
25         else
26             return new ArrayList < Integer > ();
27         return res;
```

Complexity Analysis

- Time complexity : $O(nB)$. We need to consider all the possible B positions for every current index considered in the A array. Here, A refers to the number of elements in A.
- Space complexity : $O(n)$. dp and $next$ array of size n are used.

Analysis written by: @vinod23

Rate this article:

Previous Next

Comments: 15

Sort By

Type comment here... (Markdown is supported)

Preview Post

whack-a-mole ★38 December 3, 2017 3:18 AM

There's also a linear solution using a sliding window.

You need a queue that allows you these operations: push(), pop() and getMin(). This can be done in amortized O(1) time.

Read More

3 Share Reply

SHOW 1 REPLY

Ark-kun ★69 February 8, 2018 5:13 PM Report

This can be solved a bit better using the Dijkstra path-finding. Go back from finish (for lexicographical ordering) and only explore the cheapest total paths.
It helps in cases like the following:
[1,9,9,9,9,9,9,9,9,9,1,9,9,9,9,9,9,9,9,9,9,1,9,9,9,9,9,9,9,9,9,9,1,9,9,9,9,9,9,9,9,9,9,1,9,9,9,9,9,9,9,9,9,9,1]
10

Read More

1 Share Reply

SHOW 1 REPLY

KnightY ★60 August 9, 2017 9:50 PM

Why second approach has O(n^2) time complexity instead of O(nB)? *It looks to me that for every node you try at most B possible locations to jump at, which should yield to O(n*B) time complexity. And also I think DP and recursive+memorization should have the same time complexity, these two are different ways of coding, in my opinion. Please correct me if I'm wrong.

1 Share Reply

hello_world_cn ★240 September 30, 2018 1:34 PM

Approach 3, Line 23: `A[i] >= 0` is not necessary.

0 Share Reply

vinod23 ★425 August 9, 2017 10:36 PM

@KnightY you are right. Thanks for catching that. I have corrected it.

0 Share Reply

KnightY ★60 August 9, 2017 9:58 PM

And Thanks for sharing your answers and thoughts!

0 Share Reply

vinod23 ★425 August 8, 2017 10:44 AM

@nlharish @kk636 @kk636 @bnslakk I have fixed the codes. Please have a look. Thanks.

0 Share Reply

SHOW 1 REPLY

vinod23 ★425 August 8, 2017 10:38 AM

@zestypanda I have changed it to O(n^2). Thanks.

0 Share Reply

nlharish ★0 August 8, 2017 5:01 AM

Need to update the code to A[j] >= 0 in all places.

0 Share Reply

MockCode ★26 August 7, 2017 11:27 PM

The code in the last approach can't pass.

0 Share Reply

‹ 1 2 ›