# Problem 656: Coin Path

## Problem Information

**Difficulty:** Hard
**Acceptance Rate:** 0.00%
**Paid Only:** No

## Problem Description

You are given an integer array

coins

(

1-indexed

) of length

n

and an integer

maxJump

. You can jump to any index

i

of the array

coins

if

coins[i] != -1

and you have to pay

coins[i]

when you visit index

i

. In addition to that, if you are currently at index

i

, you can only jump to any index

i + k

where

i + k <= n

and

k

is a value in the range

[1, maxJump]

.

You are initially positioned at index

1

(

coins[1]

is not

-1

). You want to find the path that reaches index n with the minimum cost.

Return an integer array of the indices that you will visit in order so that you can reach index n with the minimum cost. If there are multiple paths with the same cost, return the

lexicographically smallest

such path. If it is not possible to reach index n, return an empty array.

A path

p1 = [Pa

1

, Pa

2

, ..., Pa

x

]

of length

x

is

lexicographically smaller

than

p2 = [Pb

1

, Pb

2

, ..., Pb

x

]

of length

y

, if and only if at the first

j

where

Pa

j

and

Pb

j

differ,

Pa

j

< Pb

j

; when no such

j

exists, then

x < y

.

Example 1:

Input:

coins = [1,2,4,-1,2], maxJump = 2

Output:

[1,3,5]

Example 2:

Input:

coins = [1,2,4,-1,2], maxJump = 1

Output:

[]

Constraints:

1 <= coins.length <= 1000

-1 <= coins[i] <= 100

coins[1] != -1

1 <= maxJump <= 100

## Code Snippets

**C++:**

```
class Solution {
public:
vector<int> cheapestJump(vector<int>& coins, int maxJump) {

}
};
```

**Java:**

```
class Solution {
public List<Integer> cheapestJump(int[] coins, int maxJump) {

}
}
```

**Python3:**

```
class Solution:
def cheapestJump(self, coins: List[int], maxJump: int) -> List[int]:
```

**Python:**

```
class Solution(object):
def cheapestJump(self, coins, maxJump):
"""
:type coins: List[int]
:type maxJump: int
:rtype: List[int]
"""
```

**JavaScript:**

```
/**
* @param {number[]} coins
* @param {number} maxJump
* @return {number[]}
*/
var cheapestJump = function(coins, maxJump) {

};
```

**TypeScript:**

```
function cheapestJump(coins: number[], maxJump: number): number[] {

};
```

**C#:**

```
public class Solution {
public IList<int> CheapestJump(int[] coins, int maxJump) {

}
}
```

**C:**

```
/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* cheapestJump(int* coins, int coinsSize, int maxJump, int* returnSize) {

}
```

**Go:**

```
func cheapestJump(coins []int, maxJump int) []int {

}
```

**Kotlin:**

```
class Solution {
fun cheapestJump(coins: IntArray, maxJump: Int): List<Int> {

}
}
```

**Swift:**

```
class Solution {
func cheapestJump(_ coins: [Int], _ maxJump: Int) -> [Int] {

}
}
```

**Rust:**

```
impl Solution {
pub fn cheapest_jump(coins: Vec<i32>, max_jump: i32) -> Vec<i32> {

}
}
```

**Ruby:**

```
# @param {Integer[]} coins
# @param {Integer} max_jump
# @return {Integer[]}
def cheapest_jump(coins, max_jump)

end
```

**PHP:**

```
class Solution {

/**
* @param Integer[] $coins
* @param Integer $maxJump
* @return Integer[]
*/
function cheapestJump($coins, $maxJump) {

}
```

```
}
```

**Dart:**

```
class Solution {
List<int> cheapestJump(List<int> coins, int maxJump) {

}
}
```

**Scala:**

```
object Solution {
def cheapestJump(coins: Array[Int], maxJump: Int): List[Int] = {

}
}
```

**Elixir:**

```
defmodule Solution do
@spec cheapest_jump(coins :: [integer], max_jump :: integer) :: [integer]
def cheapest_jump(coins, max_jump) do

end
end
```

**Erlang:**

```
-spec cheapest_jump(Coins :: [integer()], MaxJump :: integer()) ->
[integer()].
cheapest_jump(Coins, MaxJump) ->
.
```

**Racket:**

```
(define/contract (cheapest-jump coins maxJump)
(-> (listof exact-integer?) exact-integer? (listof exact-integer?))
)
```

## Solutions

### C++ Solution:

```
/*
* Problem: Coin Path
* Difficulty: Hard
* Tags: array, graph, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public:
vector<int> cheapestJump(vector<int>& coins, int maxJump) {

}
};
```

### Java Solution:

```
/**
* Problem: Coin Path
* Difficulty: Hard
* Tags: array, graph, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

class Solution {
public List<Integer> cheapestJump(int[] coins, int maxJump) {

}
}
```

### Python3 Solution:

```
"""
Problem: Coin Path
```

```
Difficulty: Hard
Tags: array, graph, dp

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(n) or O(n * m) for DP table
"""

class Solution:
def cheapestJump(self, coins: List[int], maxJump: int) -> List[int]:
# TODO: Implement optimized solution
pass
```

### Python Solution:

```
class Solution(object):
def cheapestJump(self, coins, maxJump):
"""
:type coins: List[int]
:type maxJump: int
:rtype: List[int]
"""
```

### JavaScript Solution:

```
/**
* Problem: Coin Path
* Difficulty: Hard
* Tags: array, graph, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
* @param {number[]} coins
* @param {number} maxJump
* @return {number[]}
*/
var cheapestJump = function(coins, maxJump) {
```

```
};
```

### TypeScript Solution:

```
/**
* Problem: Coin Path
* Difficulty: Hard
* Tags: array, graph, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

function cheapestJump(coins: number[], maxJump: number): number[] {

};
```

### C# Solution:

```
/*
* Problem: Coin Path
* Difficulty: Hard
* Tags: array, graph, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

public class Solution {
public IList<int> CheapestJump(int[] coins, int maxJump) {

}
}
```

### C Solution:

```
/*
* Problem: Coin Path
```

```
* Difficulty: Hard
* Tags: array, graph, dp
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) or O(n * m) for DP table
*/

/**
* Note: The returned array must be malloced, assume caller calls free().
*/
int* cheapestJump(int* coins, int coinsSize, int maxJump, int* returnSize) {

}
```

**Go Solution:**

```
// Problem: Coin Path
// Difficulty: Hard
// Tags: array, graph, dp
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

func cheapestJump(coins []int, maxJump int) []int {

}
```

**Kotlin Solution:**

```
class Solution {
fun cheapestJump(coins: IntArray, maxJump: Int): List<Int> {

}
}
```

**Swift Solution:**

```
class Solution {
func cheapestJump(_ coins: [Int], _ maxJump: Int) -> [Int] {
```

```
}
}
```

### Rust Solution:

```
// Problem: Coin Path
// Difficulty: Hard
// Tags: array, graph, dp
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) or O(n * m) for DP table

impl Solution {
pub fn cheapest_jump(coins: Vec<i32>, max_jump: i32) -> Vec<i32> {

}
}
```

### Ruby Solution:

```
# @param {Integer[]} coins
# @param {Integer} max_jump
# @return {Integer[]}
def cheapest_jump(coins, max_jump)

end
```

### PHP Solution:

```
class Solution {

/**
* @param Integer[] $coins
* @param Integer $maxJump
* @return Integer[]
*/
function cheapestJump($coins, $maxJump) {

}
```

```
}
```

**Dart Solution:**

```
class Solution {
List<int> cheapestJump(List<int> coins, int maxJump) {

}
}
```

**Scala Solution:**

```
object Solution {
def cheapestJump(coins: Array[Int], maxJump: Int): List[Int] = {

}
}
```

**Elixir Solution:**

```
defmodule Solution do
@spec cheapest_jump(coins :: [integer], max_jump :: integer) :: [integer]
def cheapest_jump(coins, max_jump) do

end
end
```

**Erlang Solution:**

```
-spec cheapest_jump(Coins :: [integer()], MaxJump :: integer()) ->
[integer()].
cheapest_jump(Coins, MaxJump) ->
.
```

**Racket Solution:**

```
(define/contract (cheapest-jump coins maxJump)
(-> (listof exact-integer?) exact-integer? (listof exact-integer?))
)
```