<https://leetcode.com/problems/jump-game-iii/>

Given an array of non-negative integers arr, you are initially positioned at start index of the array. When you are at index i, you can jump to i + arr[i] or i - arr[i], check if you can reach to **any** index with value 0.

Notice that you can not jump outside of the array at any time.

**Example 1:**

Input: arr = [4,2,3,0,3,1,2], start = 5

Output: true

Explanation:

All possible ways to reach at index 3 with value 0 are:

index 5 -> index 4 -> index 1 -> index 3

index 5 -> index 6 -> index 4 -> index 1 -> index 3

**Example 2:**

Input: arr = [4,2,3,0,3,1,2], start = 0

Output: true

Explanation:

One possible way to reach at index 3 with value 0 is:

index 0 -> index 4 -> index 1 -> index 3

**Example 3:**

Input: arr = [3,0,2,1,2], start = 2

Output: false

Explanation: There is no way to reach at index 1 with value 0.

**Constraints:**

1 <= arr.length <= 5 \* 104

0 <= arr[i] < arr.length

0 <= start < arr.length

**Attempt 1: 2023-01-04**

**Solution 1:  Recursive traversal (10 min)**

**Basic ideas**

**Refer to**

<https://leetcode.com/problems/jump-game-iii/solutions/1618905/java-c-dfs-bfs-detailed-explanation/>

**Intuition:** Since you are given a starting index and you want to reach to the index where value is 0. So don't think much just start with your starting index and keep jumping to the index you can jump from the curr index until unless you reach index with value 0.

Now what should be the terminating conditions:

**1.)** if the curr index becomes out of range (i.e., 0>curIndex>=arr.length) then you cant reach any other index now, so no need to go further from there.

**2.)** If you reached the same index again, that means now you are stuck in a cycle, you will keep coming to and for to this index so gain don't go further.

So if any of the two condition written above comes before reaching to the index having value 0 , directly leave checking further from there and check other direction because now its sure that you can't reach to your target index at least from this index.

**Style 1: Additional O(n) space required for 'visited' array to tag we don't go back to same index again since it won't create new result even take any further steps**

class Solution {

public boolean canReach(int[] arr, int start) {

boolean[] visited = new boolean[arr.length];

return helper(arr, start, visited);

}

private boolean helper(int[] arr, int cur, boolean[] visited) {

// Since 0 <= arr[i] < arr.length, no need consider negative arr[cur]

if(cur >= arr.length || cur < 0 || visited[cur]) {

return false;

}

if(arr[cur] == 0) {

return true;

}

visited[cur] = true;

boolean forward = helper(arr, cur + arr[cur], visited);

boolean backward = helper(arr, cur - arr[cur], visited);

return forward || backward;

}

}

Time Complexity: O(n)

Space Complexity: O(n)

**Refer to**

<https://leetcode.com/problems/jump-game-iii/solutions/463798/java-python-3-bfs-and-dfs-codes-w-analysis/>

public boolean canReach(int[] arr, int start) {

return canReach(arr, start, new HashSet<>());

}

private boolean canReach(int[] arr, int pos, Set<Integer> seen) {

if (0 <= pos && pos < arr.length && seen.add(pos)) {

return arr[pos] == 0 ||

canReach(arr, pos + arr[pos], seen) ||

canReach(arr, pos - arr[pos], seen);

}

return false;

}

**Style 2: No additional O(n) space required to tag already visited index, just by flip the index value to negative or assign to negative value like -1 directly**

class Solution {

public boolean canReach(int[] arr, int start) {

boolean[] visited = new boolean[arr.length];

return helper(arr, start);

}

private boolean helper(int[] arr, int cur) {

// Since 0 <= arr[i] < arr.length, no need consider negative arr[cur]

if(cur >= arr.length || cur < 0 || arr[cur] < 0) {

return false;

}

if(arr[cur] == 0) {

return true;

}

// We can update arr[cur] to negative value either before using it in DFS

// since below either 'forward' or 'backward' branch will take one value,

// no matter if we flip negative -> positive or positive -> negative before

// the usage or we can use arr[cur] = -1;

arr[cur] = -arr[cur];

boolean forward = helper(arr, cur + arr[cur]);

boolean backward = helper(arr, cur - arr[cur]);

// We cannot update to negative value after using arr[cur], because the purpose

// to assign negative value to arr[cur] is to block further process in next

// level recursion by checking assign arr[cur] to negative value and check

// arr[cur] < 0 initially when a new level recursion begins, to simulate the same

// effect as visited array, but if we assign arr[cur] to negative value after

// already hit the next level recursion, it has chance to go into dead loop

// and cause stack overflow, e.g Input {3,0,2,1,2}, 2

//arr[cur] = -arr[cur];

return forward || backward;

}

}

Time Complexity: O(n)

Space Complexity: O(n) -> stack recursion still use O(n) but additional space is only O(1)

**Why we cannot assign arr[cur] to negative value after helper recursion call ?**

**We cannot update to negative value after using arr[cur], because the purpose to assign negative value to arr[cur] is to block further process in next level recursion by checking assign arr[cur] to negative value and check arr[cur] < 0 initially when a new level recursion begins, to simulate the same effect as visited array, but if we assign arr[cur] to negative value after already hit the next level recursion, it has chance to go into dead loop and cause stack overflow**

e.g Input {3,0,2,1,2}, 2

**Refer to**

<https://leetcode.com/problems/jump-game-iii/solutions/465602/java-c-python-1-line-recursion/>

public boolean canReach(int[] A, int i) {

return 0 <= i && i < A.length && A[i] >= 0 && ((A[i] = -A[i]) == 0 || canReach(A, i + A[i]) || canReach(A, i - A[i]));

}

Check 0 <= i < A.length

flip the checked number to negative A[i] = -A[i]

If A[i] == 0, get it and return true

Continue to check canReach(A, i + A[i]) and canReach(A, i - A[i])

## **Complexity**

Time O(N), as each number will be flipper at most once.

Space O(N) for recursion.

**Solution 2:  Iterative traversal (10 min)**

**Style 1: Additional O(n) space required for 'visited' array to tag we don't go back to same index again since it won't create new result even take any further steps**

class Solution {

public boolean canReach(int[] arr, int start) {

boolean[] visited = new boolean[arr.length];

Queue<Integer> q = new LinkedList<Integer>();

q.offer(start);

while(!q.isEmpty()) {

int cur = q.poll();

if(arr[cur] == 0) {

return true;

}

if(!visited[cur]) {

visited[cur] = true;

if(cur + arr[cur] < arr.length) {

q.offer(cur + arr[cur]);

}

if(cur - arr[cur] >= 0) {

q.offer(cur - arr[cur]);

}

}

}

return false;

}

}

Time Complexity: O(n)

Space Complexity: O(n)

**Style 2: No additional O(n) space required to tag already visited index, just by flip the index value to negative or assign to negative value like -1 directly**

class Solution {

public boolean canReach(int[] arr, int start) {

Queue<Integer> q = new LinkedList<Integer>();

q.offer(start);

while(!q.isEmpty()) {

int cur = q.poll();

if(arr[cur] == 0) {

return true;

}

if(arr[cur] > 0) {

// Not like DFS solution above have to modify arr[cur] to negative

// value before calling next level recursion, in BFS we don't change

// the arr[cur] to negative before using it since the comparison

// condition as "if(cur + arr[cur] < arr.length)" and "if(cur - arr[cur] >= 0)"

// will impact if we modify arr[cur] first

//arr[cur] = -arr[cur];

if(cur + arr[cur] < arr.length) {

q.offer(cur + arr[cur]);

}

if(cur - arr[cur] >= 0) {

q.offer(cur - arr[cur]);

}

// Make sure we modify original arrary element after use it

// and we can also use arr[cur] = -1;

arr[cur] = -arr[cur];

}

}

return false;

}

}

Time Complexity: O(n)

Space Complexity: O(n) -> iterative traversal use queue still use O(n) but additional space is only O(1)

**Why arr[cur] can be modified to negative value before actual using it in DFS but not able to do in BFS ?**

**Not like DFS solution we can modify arr[cur] to negative either before or after using it, because two branch will surely touch both original value of arr[cur] and negative arr[cur] at same recursion level, so modify it to negative doesn't change the next level recursion process, but brilliantly block the used index by checking if it is negative value at the beginning in next recursion.**

arr[cur] = -arr[cur];

boolean forward = helper(arr, cur + arr[cur]); -> either line will take positive or negative value of arr[cur]

boolean backward = helper(arr, cur - arr[cur]); -> either line will take positive or negative value of arr[cur]

=====================================

and check at the beginning in next recursion as below:

if(cur >= arr.length || cur < 0 || arr[cur] < 0) {

return false;

}