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

You are given an integer array nums. You are initially positioned at the array's **first index**, and each element in the array represents your maximum jump length at that position.

Return true *if you can reach the last index, or* false *otherwise*.

**Example 1:**

Input: nums = [2,3,1,1,4]

Output: true

Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.

**Example 2:**

Input: nums = [3,2,1,0,4]

Output: false

Explanation: You will always arrive at index 3 no matter what. Its maximum jump length is 0, which makes it impossible to reach the last index.

**Constraints:**

* 1 <= nums.length <= 104
* 0 <= nums[i] <= 105

**Attempt 1: 2023-01-20**

**Solution 1:  Native DFS (30 min, TLE)**

class Solution {

public boolean canJump(int[] nums) {

return helper(nums, 0);

}

private boolean helper(int[] nums, int index) {

// If I reach the last index, I should return true

if(index == nums.length - 1) {

return true;

}

// If at any point I reach an index with jump value = 0,

// I'll get stuck and hence will return a false

if(nums[index] == 0) {

return false;

}

// I can make jumps ranging from index + 1, till index + nums[index],

// and hence will run a loop to cover all those possbile jumps

for(int i = index + 1; i <= index + nums[index]; i++) {

// If true, it means taking this jump led me to the last index

if(i < nums.length && helper(nums, i)) {

return true;

}

}

// If I reach here, it means none of the jumps led me to the

// last index and hence returning, false.

return false;

}

}

Time Complexity : O(N^N) ~ exponential

Space Complexity: O(N)

**Solution 2:  Top Down DP (Memoization) (30 min)**

class Solution {

public boolean canJump(int[] nums) {

Boolean[] memo = new Boolean[nums.length + 1];

return helper(nums, 0, memo);

}

private boolean helper(int[] nums, int index, Boolean[] memo) {

// If I reach the last index, I should return true

if(index == nums.length - 1) {

return true;

}

// If memo already store result

if(memo[index] != null) {

return memo[index];

}

// If at any point I reach an index with jump value = 0,

// I'll get stuck and hence will return a false

if(nums[index] == 0) {

memo[index] = false;

return false;

}

// I can make jumps ranging from index + 1, till index + nums[index],

// and hence will run a loop to cover all those possbile jumps

for(int i = index + 1; i <= index + nums[index]; i++) {

// If true, it means taking this jump led me to the last index

if(i < nums.length && helper(nums, i, memo)) {

memo[i] = true;

return true;

}

}

// If I reach here, it means none of the jumps led me to the

// last index and hence returning, false.

memo[index] = false;

return false;

}

}

Time Complexity : O(N^2)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/jump-game/solutions/2375320/interview-scenario-recursion-memoization-dp-greedy/>

***let's get started***

Ask clarifying questions, examples, and scenarios. The more you ask, interviewer gets more and more intrigued, for good.

umm, at every index starting from the 0th, you can make jumps ranging from 1 till nums[index]. Greedily making a choice of jump won't work in subtle cases, for example, if from 0th index I take a jump to the index with maximum jump value, it might lead me to a postion with value of jumps equal to 0, and I'll be stuck, As I've got a lot of options to make a jump, I'd love to the explore the recurring idea behind this question(i.e. I can make a jump ranging from index(idx) to positions ranging from : (idx + 1 *till* idx + nums[idx]), and the same thing can be done from all these indexes. If by following any route I reach last index, I'll return true. Otherwise if none of the index can lead me to last index, I'll return a false.

While solving the problem, think out loud and let the interviewer know what you are thinking. recursion approach:

tc : O(N^N) ~ exponential

sc : O(N)

class Solution {

public:

bool canJump(vector<int>& nums) {

return create(nums, 0);

}

private:

bool create(vector<int>& nums, int idx) {

if(idx == nums.size() -1) return true; //if I reach the last index, I should return true;

if(nums[idx] == 0) return false; //if at any point I reach an index with jump value = 0

//,I'll get stuck and hence will return a false.

int reach = idx + nums[idx]; //the max jump that I can make

//I can make jumps ranging from idx + 1, till reach, and hence will run a loop

//to cover all those possbile jumps

for(int jump=idx + 1; jump <= reach; jump++) {

//if true, it means taking this jump led me to the last index.

if(jump < nums.size() && create(nums, jump))

return true;

}

//if I reach here, it means none of the jumps led me to the last index

//and hence returning, false.

return false;

}

};

There's no chance that this question will not give a T.L.E, it's exponential, it'll have many overlapping subproblems, and hence I can memoize it using a 1d dp array.

memoized :

tc : O(N\* N) -> for each index, I can have at max N jumps, hence O(N\* N).

sc : O(N) + O(N) -> stack space plus dp array size.

class Solution {

public:

bool canJump(vector<int>& nums) {

vector<int> dp(nums.size(), -1);

return create(nums, 0, dp);

}

private:

bool create(vector<int>& nums, int idx, vector<int>& dp) {

if(idx == nums.size() -1) return true;

if(nums[idx] == 0) return false;

if(dp[idx] != -1) return dp[idx]; //overlapping subproblems

int reach = idx + nums[idx];

for(int jump=idx + 1; jump <= reach; jump++) {

if(jump < nums.size() && create(nums, jump, dp))

return dp[idx] = true; //memoizing for particular index.

}

return dp[idx] = false; //memoizing for particular index.

}

};

**Solution 3:  Bottom Up DP (60 min)**

**Style 1: Traverse backward**

class Solution {

public boolean canJump(int[] nums) {

int n = nums.length;

boolean[] dp = new boolean[n];

// Only One Element , so we do not have to jump anywhere

dp[0] = true;

for(int i = 1; i < n; i++) {

// Here we are traversing backward

// We are checking for every position from last that whether we

// can reach that ith index with current position and jump or not

for(int j = i - 1; j >= 0; j--) {

// Here we are checking dp[j] to make sure that previously if we

// could reach the jth index or not, means that if we were not

// be able to reach the jth index then how can we jump from that

// index, so there will be no use of jumping from that index

if(dp[j] && j + nums[j] >= i) {

// Yes! we can reach the ith position from present jth position

dp[i] = true;

// If we can jump from the present jth position then no need of

// checking previous indexes as by checking dp[j] we have already

// made sure that we can reach the jth position easily and from

// there the ith position

break;

}

}

}

return dp[n - 1];

}

}

Time Complexity : O(N^2)

Space Complexity: O(N)

**Style 2: Traverse forward**

class Solution {

public boolean canJump(int[] nums) {

int n = nums.length;

boolean[] dp = new boolean[n];

// Only One Element , so we do not have to jump anywhere

dp[0] = true;

for(int i = 1; i < n; i++) {

// Here we can also traversing forward (but spend more time than backward)

// We are checking for every position from first that whether we

// can reach that ith index with current position and jump or not

for(int j = 0; j <= i; j++) {

// Here we are checking dp[j] to make sure that previously if we

// could reach the jth index or not, means that if we were not

// be able to reach the jth index then how can we jump from that

// index, so there will be no use of jumping from that index

if(dp[j] && j + nums[j] >= i) {

// Yes! we can reach the ith position from present jth position

dp[i] = true;

// If we can jump from the present jth position then no need of

// checking previous indexes as by checking dp[j] we have already

// made sure that we can reach the jth position easily and from

// there the ith position

break;

}

}

}

return dp[n - 1];

}

}

Time Complexity : O(N^2)

Space Complexity: O(N)

**Refer to**

<https://leetcode.com/problems/jump-game/solutions/2060972/c-easy-explanation-dp-tabulation-bottoms-up/>

**Solution 4:  Greedy (30 min)**

**Style 1: Traverse backward**

class Solution {

public boolean canJump(int[] nums) {

int n = nums.length;

int last\_index\_reachable = n - 1;

// Iterate backwards from second to last item until the first item

for(int i = n - 2; i >= 0; i--) {

// If this index has jump count which can reach to or beyond the

// last position, hence we just need to reach to this new index

if(i + nums[i] >= last\_index\_reachable) {

last\_index\_reachable = i;

}

}

return last\_index\_reachable == 0;

}

}

Time Complexity : O(N)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/jump-game/solutions/596454/python-simple-solution-with-thinking-process-runtime-o-n>

When I first see this problem, two things pop up in my mind:

* Maybe I can do some sort of DFS, BFS (with backtracking?) but there will be a lot of redundancies
* Then this begs for Dynamic Programming!

But my gut feeling was saying that this problem has to have a simpler approach.

So, here is my thinking process:

Base case: last index can trivially reach to last index.

**Q1**: How can I reach to the last index (I will call it last\_position) from a preceding index?

If I have a preceding index idx in nums which has jump count jump which satisfies idx+jump >= last\_position, I know that this idx is good enough to be treated as the last index because all I need to do now is to get to that idx. I am going to treat this new idx as a new last\_position.

I ask **Q1** again.

So now, here are two important things:

If we have indices which are like **sinkholes**, those with 0 as jump and every other preceding index can only jump to that sinkhole, our last\_position will not be updated anymore because idx+jump >= last\_position will not be satisfied at that sinkhole and every other preceding index cannot satisfy the idx+jump >= last\_position condition since their jumps are not big enough.

E.g. nums=[3,2,1,0,4] # Here 0 is a sinkhole becuase all preceding indices can only jump to the sinkhole

If we have **barriers**, those indices with 0 as jump, but the preceding indices contain jumps which can go beyond those barriers, idx+jump >= last\_position will be satisfied and last\_position will be updated.

E.g. nums=[3,2,2,0,4] # Here 0 is just a barrier since the index before that 0 can jump \*over\* that barrier

Finally ask this question when we have finished looping

Is the last position index of 0? (i.e, have we reached to the beginning while doing the process of jumping and updating the last\_position?)

If we have sinkholes in nums, our last\_position will not be 0. Thus, False will be returned.

1. class Solution:

2. def canJump(self, nums: List[int]) -> bool:

3. last\_position = len(nums)-1

4.

5. for i in range(len(nums)-2,-1,-1): # Iterate backwards from second to last item until the first item

6. if (i + nums[i]) >= last\_position: # If this index has jump count which can reach to or beyond the last position

7. last\_position = i # Since we just need to reach to this new index

8. return last\_position == 0

**Style 2: Traverse forward**

class Solution {

public boolean canJump(int[] nums) {

int n = nums.length;

int max\_reachable = 0;

// Iterate forwards from first item to the last item

for(int i = 0; i < n; i++) {

// max\_reachable >= i -> great, carry on, otherwise, cannot progress

if(max\_reachable < i) {

return false;

}

// now as you can reach this index, it's time to update your reach

// as at every index, you're getting a new jump length.

max\_reachable = Math.max(max\_reachable, i + nums[i]);

}

return true;

}

}

Time Complexity : O(N)

Space Complexity: O(1)

**Refer to**

<https://leetcode.com/problems/jump-game/solutions/2375320/interview-scenario-recursion-memoization-dp-greedy/>

**Bonus**, I can also think of a solution that kinda resembles Kadane's algorithm. For every index, I'm checking the max reach I can have till that element, if that reach is less than the value of my index, that means I can never reach this particular index and my answer should be false.

tc : O(N)

sc : O(1)

The comments that I've made show my entire thought process that I'd be delivering to the interviewer.

class Solution {

public:

bool canJump(vector<int>& nums) {

//it shows at max what index can I reach.

//initially I can only reach index 0, hence reach = 0

int reach = 0;

for(int idx = 0; idx < nums.size(); idx++) {

//at every index I'll check if my reach was atleast able to

//reach that particular index.

//reach >= idx -> great, carry on. Otherwise,

if(reach < idx) return false;

//now as you can reach this index, it's time to update your reach

//as at every index, you're getting a new jump length.

reach = max(reach, idx + nums[idx]);

}

//this means that you reached till the end of the array, wohooo!!

return true;

}

};