1871. Jump Game VII

String]

Medium

Problem Description

Dynamic Programming Prefix Sum

You are given a binary string s that is 0-indexed, which means indexing starts from 0. Your task is to determine if you can move from the start position, index 0, to the last index of the binary string, s.length - 1, using a set of movement rules defined by

• The destination index j must have a value of '0' in the binary string (s[j] == '0').

• You can only jump to index j from index i if i + minJump <= j <= min(i + maxJump, s.length - 1).

two integers minJump and maxJump. The conditions for moving from index i to index j are as follows:

return true if reaching the last index is possible under the given conditions, otherwise return false.

You can only start your movement from index 0 if its value is '0', and you must end at the last index s.length - 1. The goal is to

Intuition

Sliding Window

reach s[i].

The essence of this problem is to find if there's a path from the beginning to the end of s while obeying the jump constraints. One intuitive approach to solve this problem will be using dynamic programming, where we calculate the possibility of reaching a particular index based on the possibilities of reaching previous indices.

The solution hinges on the following intuitions:

• We should track reachability from the start to each index with a boolean array dp where each index i of dp signifies whether it is possible to

• To efficiently determine if we can reach index i, we should use a prefix sum array pre_sum to quickly calculate the number of reachable indices

in the range [i - maxJump, i - minJump] without iterating through all of them every time.

- If any index j within the range [i maxJump, i minJump] is reachable (dp[j] == true) and s[i] == '0', then index i is also reachable. Thus, starting from index 0, we iteratively compute dp and pre_sum until we reach the end of the array or until it's clear that the
- Solution Approach

end is not reachable. The last element of the dp array gives us the answer to whether the end is reachable.

To implement the solution, we use <u>dynamic programming</u> combined with a <u>prefix sum</u> strategy. Here's a step-by-step breakdown

of how the solution is composed:

First, initialize two arrays dp and pre_sum with lengths equal to n and n + 1 respectively, where n is the length of the binary

string s. The array dp will store boolean values indicating whether each index i is reachable from the start while pre_sum

stores the prefix sums of dp. Set dp[0] to True because we start from index 0 and it's already reachable by definition.

If s[i] is '0', check if there's any reachable index within the window defined by the current index i minus minJump and

maxJump. To do this quickly, we calculated the <u>prefix sum</u> up to index i - minJump and subtract it from the prefix sum up to

index i - maxJump. If the resulting value is greater than 0, it means there is at least one reachable '0' we can jump from within

Similarly, set pre_sum[1] to 1 as we can reach the first element.

Iterate through each character in the binary string starting from index 1 as we have already initialized index 0. On each iteration, check if the current character s[i] is '0'. If it's not, we can't possibly jump to this index anyway, so we move on.

- the bounds. In this case, set dp[i] to True. Update the <u>prefix sum</u> array <u>pre_sum</u> accordingly by adding the value at dp[i] to the prefix sum up to index i. • Continue this process until you've gone through all characters of the binary string. The last element of the dp array, dp[n -
- The key algorithms and data structures used in this approach are: • **Dynamic Programming (DP):** Used to store the reachability state of each index as we progress through the string.

• Prefix Sum Array: Allows us to efficiently calculate the sum of elements in a range, which helps in quickly determining if a jump is possible

redundant calculations. This way, each i is checked only once, making the solution optimal in terms of time complexity.

1], gives us the final answer. If it's True, that means the end of the string is reachable; otherwise, it is unreachable.

Example Walkthrough Let's consider a small example to illustrate the solution. Suppose we have a binary string s = "0010110" and the given jump rules

By using the DP and Prefix Sum Array, we efficiently solve the problem by reusing the results of the sub-problems and avoiding

 We can start from index 0 since s[0] == '0', so dp[0] = true and pre_sum[1] = 1. **Step 1: Checking each index**

• Index 2 \rightarrow s[2] == '0' and it's within the minJump and maxJump from 0 (0 + 2 <= 2 <= 0 + 3). We find that pre_sum[2 - 1 + 1] -

 $pre_sum[2-3+1] = pre_sum[2] - pre_sum[0] = 1-0 = 1 > 0$, indicating that a previous index is reachable. So, dp[2] = true and

• Index 1 → s[1] == '0', but index 1 cannot be reached directly from 0 as minJump > 1, so dp[1] is not changed.

$pre_sum[3] = pre_sum[2] + dp[2] = 1 + 1 = 2.$ Index 3 → s[3] == '1', so it can't be reached because the position does not contain '0'.

Step 0: Initialization

are minJump = 2 and maxJump = 3.

• pre_sum = [0, 0, 0, 0, 0, 0, 0]

• Index 4 → s[4] == '0'. We can jump from both indices 1 and 2 to 4. However, since dp[1] is false and dp[2] is true, we only consider the 2 index. Again, we find that $pre_sum[4 - 2 + 1] - pre_sum[4 - 3 + 1] = pre_sum[3] - pre_sum[2] = 2 - 1 = 1 > 0, so dp[4] = true and$ $pre_sum[5] = pre_sum[4] + dp[4] = 2 + 1 = 3.$

• Index 6 → s[6] == '0'. It's reachable from index 3, 4 with minJump and maxJump rules. But since dp[3] is false, we only consider the 4.

Checking the pre_sum, pre_sum[6 - 2 + 1] - pre_sum[6 - 3 + 1] = pre_sum[5] - pre_sum[4] = 3 - 2 = 1 > 0, so dp[6] = true and

Step 2: Final Result • Checking the last element in dp array, dp[6] = true. This indicates that it's indeed possible to jump to the last index using the given rules. Thus, the function should return true for this test case.

confirming the possibility to reach the end index.

• pre_sum = [0, 1, 1, 2, 2, 3, 3, 3]

class Solution:

 $pre_sum[7] = pre_sum[6] + dp[6] = 2 + 1 = 3.$

Index 5 → s[5] == '1', so it's not reachable.

without checking every individual index within the range each time.

• dp = [false, false, false, false, false, false]

The final dp and pre_sum look like this: dp = [true, false, true, false, true, false, true]

def canReach(self, arr: str, min_jump: int, max_jump: int) -> bool:

A dynamic programming list to keep track of the possibility to reach each index

Prefix sum array to keep track of number of reachable points in the string up to index i

We only need to consider '0' positions since '1' positions are not reachable

If the sum of reachable points between the bounds is greater than 0,

Update the prefix sum array with the reachability of the current index

if right bound >= left bound and prefix_sum[right_bound + 1] - prefix_sum[left_bound] > 0:

left bound = max(0, i - max jump) # Lower bound for the jump

right_bound = i - min_jump # Upper bound for the jump

Solution Implementation **Python**

We've successfully found a path that allows us to travel from index 0 to index 6, thereby adhering to the solution approach and

prefix_sum[1] = 1 # Loop through each point in the string

for i in range(1, length):

if arr[i] == '0':

return can_reach[length - 1]

int length = s.length();

// Get the length of the input string.

* @param max jump The maximum length of a jump.

int n = s.length(); // Length of the string

prefix sum[1] = 1; // Initialize prefix sum

for (int i = 1: i < n: i++)

dp[i] = 1;

if (s[i] == '0') {

return dp[n - 1] == 1;

// Example usage:

int main() {

* @return boolean indicating whether the end of the string can be reached.

std::vector<int> dp(n, 0); // DP array to store whether a position is reachable

// Iterate through the string to fill the DP array with reachable positions

// Update the prefix sum array with the current reachability status

// Function to determine if it's possible to reach the end of a string by jumping

// Each jump can only be made if the destination is a '0' character in the string.

// Iterate through the string to fill the DP array with reachable positions

// Update the prefix sum array with current reachability status

def canReach(self, arr: str, min_jump: int, max_jump: int) -> bool:

A dynamic programming list to keep track of the possibility to reach each index

Prefix sum array to keep track of number of reachable points in the string up to index i

We only need to consider '0' positions since '1' positions are not reachable

If the sum of reachable points between the bounds is greater than 0,

left bound = max(0, i - max jump) # Lower bound for the jump

right_bound = i - min_jump # Upper bound for the jump

Update the prefix sum array with the reachability of the current index

// Check if the current position is '0' and can potentially be jumped to

* @returns boolean indicating whether the end of the string can be reached.

const canReach = (s: string, minJump: number, maxJump: number): boolean => {

std::cout << (result ? "true" : "false") << std::endl; // Expected output: true or false

let dp: number[] = new Array(n).fill(0); // DP array to store whether a position is reachable

let prefixSum: number[] = new Array(n + 1).fill(0); // Array to store the prefix sum of the DP

let leftBound = Math.max(0, i - maxJump); // Left boundary for the window from which we can jump

// If there is at least one reachable position within the window, mark current position as reachable

let rightBound = i - minJump; // Right boundary for the window from which we can jump

if (leftBound <= rightBound && prefixSum[rightBound + 1] - prefixSum[leftBound] > 0) {

// Check if the current position is '0' and can potentially be jumped to

int left bound = std::max(0, i - max jump); // Left boundary for the window from which we can jump

// If there is at least one reachable position within the window, mark the current position as reachable

int right bound = i - min jump; // Right boundary for the window from which we can jump

if (left bound <= right bound && prefix sum[right bound + 1] - prefix sum[left bound] > 0) {

std::vector<int> prefix_sum(n + 1, 0); // Array to store the prefix sum of the DP

bool canReach(const std::string& s, int min jump, int max_jump) {

dp[0] = 1: // Starting position is always reachable

prefix_sum[i + 1] = prefix_sum[i] + dp[i];

// between the positions specified by minJump and maxJump.

const n: number = s.length; // Length of the string

dp[0] = 1; // Starting position is always reachable

prefixSum[i + 1] = prefixSum[i] + dp[i];

// Return if the last position is reachable

Length of the input string

can_reach = [False] * length

 $prefix_sum = [0] * (length + 1)$

for i in range(1, length):

if arr[i] == '0':

return can_reach[length - 1]

The starting point is always reachable

Loop through each point in the string

Initially only the first point is reachable

Calculate the range of jumps

can reach[i] = True

then the current point is reachable

prefix_sum[i + 1] = prefix_sum[i] + int(can_reach[i])

2. A single for-loop on the index i ranges from 1 to n-1, resulting in O(n) iterations.

Check if the last point is reachable, return the result

prefixSum[1] = 1; // Initialize prefix sum

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

if (s.charAt(i) === '0') {

dp[i] = 1;

return dp[n - 1] === 1;

length = len(arr)

can_reach[0] = True

 $prefix_sum[1] = 1$

* @param s The string representing safe(0) and unsafe(1) positions.

// Return if the last position is reachable

bool result = canReach("011010", 2, 3);

* @param minJump The minimum length of a iump.

* @param maxJump The maximum length of a jump.

Length of the input string

can reach = [False] * length

 $prefix_sum = [0] * (length + 1)$

The starting point is always reachable

Initially only the first point is reachable

Calculate the range of jumps

can reach[i] = True

then the current point is reachable

prefix_sum[i + 1] = prefix_sum[i] + int(can_reach[i])

Check if the last point is reachable, return the result

public boolean canReach(String s, int minJump, int maxJump) {

length = len(arr)

can reach[0] = True

Java

class Solution {

```
// Create a dynamic programming (dp) array to hold the reachability of each position.
        boolean[] isReachable = new boolean[length];
       // Always start at position 0.
        isReachable[0] = true;
       // Use a prefix sum array to keep a running sum of reachable positions.
        int[] prefixSum = new int[length + 1];
        prefixSum[1] = 1;
       // Iterate through the string starting at position 1 since we're always starting at position 0.
        for (int i = 1; i < length; ++i) {</pre>
            // Check if the current position has a '0' and therefore can be landed on.
            if (s.charAt(i) == '0') {
                // Calculate the maximum left (furthest back we can jump from current position).
                int left = Math.max(0, i - maxJump);
                // Calculate the minimum right (closest jump we can make to current position).
                int right = i - minJump;
                // Ensure that right is not less than left and that there is at least one true
                // within the range in the prefix sum array to jump to current position.
                if (right >= left && prefixSum[right + 1] - prefixSum[left] > 0) {
                    // Mark the current position as reachable.
                    isReachable[i] = true;
            // Update the prefix sum array with the sum up to the current position.
            prefixSum[i + 1] = prefixSum[i] + (isReachable[i] ? 1 : 0);
       // Return if the last position in the string is reachable.
        return isReachable[length - 1];
C++
#include <vector>
#include <string>
/**
* Function to determine if it's possible to reach the end of a string by jumping
* between the positions specified by min jump and max jump.
* Each jump can only be made if the destination is a '0' character in the string.
* @param s The string representing safe ('0') and unsafe ('1') positions.
* @param min jump The minimum length of a jump.
```

return 0; */ **TypeScript**

/**

```
// Example usage:
// const result: boolean = canReach("011010", 2, 3);
// console.log(result); // Expected output: true or false
```

class Solution:

Time and Space Complexity The given Python code implements a dynamic programming solution to determine if it is possible to reach the end of a string given certain jumping rules.

The time complexity of the algorithm can be analyzed based on loop operations and array manipulations:

if right bound >= left bound and prefix_sum[right_bound + 1] - prefix_sum[left_bound] > 0:

3. Inside the for-loop, the check pre_sum[r + 1] - pre_sum[l] > 0 is a constant time operation, O(1), as it involves accessing elements of the prefix sum array and subtraction.

Time Complexity

string.

4. The update of the pre_sum array with pre_sum[i + 1] = pre_sum[i] + dp[i] is also a constant time operation, executed once each loop iteration.

1. Initialization of the dp and pre_sum arrays with False and 0 respectively, which takes 0(n) time each, where n is the length of the input

Combining these observations, the overall time complexity can be considered as 0(n) because the for-loop dominates the execution time.

Space Complexity

The space complexity is determined by the space required to store the dynamic programming states and additional constructs: 1. The dp array that stores Boolean values representing if a position i can be reached, has a size of n, contributing 0(n) to the space complexity.

Hence, the overall space complexity of this solution is O(n), which is a sum of space used by dp and pre_sum.

2. The pre_sum array helps keep track of the prefix sums of dp, also requiring 0(n) space.

In conclusion, the given code has a time complexity of O(n) and a space complexity of O(n).