<https://leetcode.com/problems/minimum-number-of-taps-to-open-to-water-a-garden/description/>

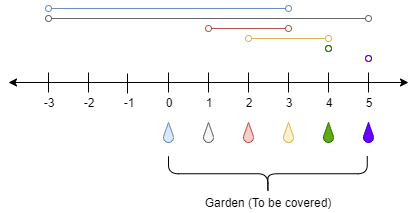
There is a one-dimensional garden on the x-axis. The garden starts at the point 0 and ends at the point n. (i.e., the length of the garden is n).

There are n + 1 taps located at points [0, 1, ..., n] in the garden.

Given an integer n and an integer array ranges of length n + 1 where ranges[i] (0-indexed) means the i-th tap can water the area [i - ranges[i], i + ranges[i]] if it was open.

Return **the minimum number of taps** that should be open to water the whole garden, If the garden cannot be watered return **-1**.

**Example 1:**



**Input:** n = 5, ranges = [3,4,1,1,0,0]

**Output:** 1

**Explanation:**

The tap at point 0 can cover the interval [-3,3]

The tap at point 1 can cover the interval [-3,5]

The tap at point 2 can cover the interval [1,3]

The tap at point 3 can cover the interval [2,4]

The tap at point 4 can cover the interval [4,4]

The tap at point 5 can cover the interval [5,5]

Opening Only the second tap will water the whole garden [0,5]

**Example 2:**

**Input:** n = 3, ranges = [0,0,0,0]

**Output:** -1

**Explanation:** Even if you activate all the four taps you cannot water the whole garden.

**Constraints:**

* 1 <= n <= 10^4
* ranges.length == n + 1
* 0 <= ranges[i] <= 100

**Attempt 1: 2023-12-14**

**The problem fully refer to L1024.Video Stitching and L45. Jump Game II**

**Solution 1: Native DFS (30 min,TLE 13/40)**

**Wrong Solution:**

**Test out by:**

Input n = 5 ranges = [4,1,1,1,1,1] Output = 3 Expected = 2

**The problem is happening on 'garden' array build process, initially, when i = 0 for garden[0] range should be 4, based on "i + ranges[i] = 0 + ranges[0] = 4", but when i = 1, since "i - ranges[i] = 1 - 1 = 0", then based on "2 \* ranges[i] = 2 \* ranges[1] = 2", it overwrites garden[0] = 4 to 2, which is wrong, in greedy model, we should keep the maximum range value for any 'garden' array position because we want to jump as far as possible to reach the end index more quickly, the solution is add "Math.max(garden[start], range)" to only keep max.**

class Solution {

    public int minTaps(int n, int[] ranges) {

        int[] garden = new int[n + 1];

        for(int i = 0; i < ranges.length; i++) {

            int start = i - ranges[i] < 0 ? 0 : i - ranges[i];

            int range = i - ranges[i] < 0 ? i + ranges[i] : 2 \* ranges[i];

            garden[start] = Math.max(garden[start], range);

        }

        int[] dp = new int[n + 1];

        Arrays.fill(dp, 10001);

        dp[0] = 0;

        for(int i = 1; i <= n; i++) {

            for(int j = 0; j < garden.length; j++) {

                if(j <= i && i <= j + garden[j]) {

                    dp[i] = Math.min(dp[i], dp[j] + 1);

                }

            }

        }

        return dp[n] == 10001 ? -1 : dp[n];

    }

}

**Correct Solution:**

class Solution {

    public int minTaps(int n, int[] ranges) {

        // Similar to L1024.Video Stitching / L45.Jump Game II

        // Build the 'garden' array where 'garden[i]' stores the maximum

        // range it can jump to

        // '+ 1' to include 0-index

        int[] garden = new int[n + 1];

        for(int i = 0; i < ranges.length; i++) {

            // Need to consider 'start' index as negative case, we only

            // calculate based on 0

            int start = i - ranges[i] < 0 ? 0 : i - ranges[i];

            // Need to consider 'garden[start]' not as 2 \* ranges[i] case

            // because of 'start' index might be negative

            int range = i - ranges[i] < 0 ? i + ranges[i] : 2 \* ranges[i];

            // Test out by:

            // Input n = 5, ranges = [4,1,1,1,1,1]

            // Output = 3 Expected = 2

            // The problem is happening on 'garden' array build process, initially

            // when i = 0 for garden[0] range should be 4, based on "i + ranges[i]

            // = 0 + ranges[0] = 4", but when i = 1, since "i - ranges[i] = 1 - 1

            // = 0", then based on "2 \* ranges[i] = 2 \* ranges[1] = 2", it overwrites

            // garden[0] = 4 to 2, which is wrong, in greedy model, we should keep

            // the maximum range value for any 'garden' array position because we

            // want to jump as far as possible to reach the end index more quickly,

            // the solution is add "Math.max(garden[start], range)" to only keep max

            garden[start] = Math.max(garden[start], range);

        }

        int result = helper(garden, 0);

        return result == 10001 ? -1 : result;

    }

    // Return remain steps required to get last position of array

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

        // Able to reach, no more steps required

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

            return 0;

        }

        // Not able to reach, infinite steps remained

        // Note: actually not required, since minimum steps maximum value

        // set as 101, even return because of not approach last position

        // but already stuck on certain index won't increase the maximum value

        if(nums[index] == 0) {

            // In definition 1 <= n <= 10^4, return + 1 value when not

            // able process stopped because of no range at index

            return 10001;

        }

        int min = 10001;

        // 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(i < nums.length) {

                min = Math.min(min, 1 + helper(nums, i));

            }

        }

        return min;

    }

}

Time Complexity: O(N!)

At each index i we have N-i choices and we recursively explore each of them till end. So we require O(N\*(N-1)\*(N-2)...1) = O(N!).

Space Complexity: O(N)

**Solution 2: Top Down DP (Memoization) (10 min)**

class Solution {

    public int minTaps(int n, int[] ranges) {

        // Similar to L1024.Video Stitching / L45.Jump Game II

        // Build the 'garden' array where 'garden[i]' stores the maximum

        // range it can jump to

        // '+ 1' to include 0-index

        int[] garden = new int[n + 1];

        for(int i = 0; i < ranges.length; i++) {

            // Need to consider 'start' index as negative case, we only

            // calculate based on 0

            int start = i - ranges[i] < 0 ? 0 : i - ranges[i];

            // Need to consider 'garden[start]' not as 2 \* ranges[i] case

            // because of 'start' index might be negative

            int range = i - ranges[i] < 0 ? i + ranges[i] : 2 \* ranges[i];

            // Test out by:

            // Input n = 5, ranges = [4,1,1,1,1,1]

            // Output = 3 Expected = 2

            // The problem is happening on 'garden' array build process, initially

            // when i = 0 for garden[0] range should be 4, based on "i + ranges[i]

            // = 0 + ranges[0] = 4", but when i = 1, since "i - ranges[i] = 1 - 1

            // = 0", then based on "2 \* ranges[i] = 2 \* ranges[1] = 2", it overwrites

            // garden[0] = 4 to 2, which is wrong, in greedy model, we should keep

            // the maximum range value for any 'garden' array position because we

            // want to jump as far as possible to reach the end index more quickly,

            // the solution is add "Math.max(garden[start], range)" to only keep max

            garden[start] = Math.max(garden[start], range);

        }

        Integer[] memo = new Integer[garden.length];

        int result = helper(garden, 0, memo);

        return result == 10001 ? -1 : result;

    }

    // Return remain steps required to get last position of array

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

        // Able to reach, no more steps required

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

            return 0;

        }

        // Not able to reach, infinite steps remained

        // Note: actually not required, since minimum steps maximum value

        // set as 101, even return because of not approach last position

        // but already stuck on certain index won't increase the maximum value

        if(nums[index] == 0) {

            // In definition 1 <= n <= 10^4, return + 1 value when not

            // able process stopped because of no range at index

            return 10001;

        }

        if(memo[index] != null) {

            return memo[index];

        }

        int min = 10001;

        // 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(i < nums.length) {

                min = Math.min(min, 1 + helper(nums, i, memo));

            }

        }

        return memo[index] = min;

    }

}

Time Complexity : O(N^2)

Space Complexity: O(N)

**Solution 3: Bottom Up DP (10 min)**

**Style 1: Traverse backward**

class Solution {

    public int minTaps(int n, int[] ranges) {

        // Similar to L1024.Video Stitching / L45.Jump Game II

        // Build the 'garden' array where 'garden[i]' stores the maximum

        // range it can jump to

        // '+ 1' to include 0-index

        int[] garden = new int[n + 1];

        for(int i = 0; i < ranges.length; i++) {

            // Need to consider 'start' index as negative case, we only

            // calculate based on 0

            int start = i - ranges[i] < 0 ? 0 : i - ranges[i];

            // Need to consider 'garden[start]' not as 2 \* ranges[i] case

            // because of 'start' index might be negative

            int range = i - ranges[i] < 0 ? i + ranges[i] : 2 \* ranges[i];

            // Test out by:

            // Input n = 5, ranges = [4,1,1,1,1,1]

            // Output = 3 Expected = 2

            // The problem is happening on 'garden' array build process, initially

            // when i = 0 for garden[0] range should be 4, based on "i + ranges[i]

            // = 0 + ranges[0] = 4", but when i = 1, since "i - ranges[i] = 1 - 1

            // = 0", then based on "2 \* ranges[i] = 2 \* ranges[1] = 2", it overwrites

            // garden[0] = 4 to 2, which is wrong, in greedy model, we should keep

            // the maximum range value for any 'garden' array position because we

            // want to jump as far as possible to reach the end index more quickly,

            // the solution is add "Math.max(garden[start], range)" to only keep max

            garden[start] = Math.max(garden[start], range);

        }

        int len = garden.length;

        // dp[i] means minimum steps can reach last position from index 'i'

        int[] dp = new int[len];

        Arrays.fill(dp, 10001);

        // Since no step required for last position, minimum step as 0

        dp[len - 1] = 0;

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

            for(int j = 1; j <= garden[i]; j++) {

                if(i + j < len) {

                    dp[i] = Math.min(dp[i], 1 + dp[i + j]);

                }

            }

        }

        return dp[0] == 10001 ? -1 : dp[0];

    }

}

Time Complexity : O(N^2)

Space Complexity: O(N)

**Style 2: Traverse forward**

class Solution {

    public int minTaps(int n, int[] ranges) {

        // Similar to L1024.Video Stitching / L45.Jump Game II

        // Build the 'garden' array where 'garden[i]' stores the maximum

        // range it can jump to

        // '+ 1' to include 0-index

        int[] garden = new int[n + 1];

        for(int i = 0; i < ranges.length; i++) {

            // Need to consider 'start' index as negative case, we only

            // calculate based on 0

            int start = i - ranges[i] < 0 ? 0 : i - ranges[i];

            // Need to consider 'garden[start]' not as 2 \* ranges[i] case

            // because of 'start' index might be negative

            int range = i - ranges[i] < 0 ? i + ranges[i] : 2 \* ranges[i];

            // Test out by:

            // Input n = 5, ranges = [4,1,1,1,1,1]

            // Output = 3 Expected = 2

            // The problem is happening on 'garden' array build process, initially

            // when i = 0 for garden[0] range should be 4, based on "i + ranges[i]

            // = 0 + ranges[0] = 4", but when i = 1, since "i - ranges[i] = 1 - 1

            // = 0", then based on "2 \* ranges[i] = 2 \* ranges[1] = 2", it overwrites

            // garden[0] = 4 to 2, which is wrong, in greedy model, we should keep

            // the maximum range value for any 'garden' array position because we

            // want to jump as far as possible to reach the end index more quickly,

            // the solution is add "Math.max(garden[start], range)" to only keep max

            garden[start] = Math.max(garden[start], range);

        }

        int len = garden.length;

        // dp[i] means minimum steps can reach last position from index 'i'

        int[] dp = new int[len];

        Arrays.fill(dp, 10001);

        // Since no step required for last position, minimum step as 0

        dp[0] = 0;

        for(int i = 0; i <= len - 2; i++) {

            for(int j = 1; j <= garden[i]; j++) {

                if(i + j < len) {

                    dp[i + j] = Math.min(dp[i + j], 1 + dp[i]);

                }

            }

        }

        return dp[len - 1] == 10001 ? -1 : dp[len - 1];

    }

}

Time Complexity : O(N^2)

Space Complexity: O(N)

**Solution 3: Another style DP (30 min)**

class Solution {

    public int minTaps(int n, int[] ranges) {

        int[] garden = new int[n + 1];

        for(int i = 0; i < ranges.length; i++) {

            int start = i - ranges[i] < 0 ? 0 : i - ranges[i];

            int range = i - ranges[i] < 0 ? i + ranges[i] : 2 \* ranges[i];

            garden[start] = Math.max(garden[start], range);

        }

        int[] dp = new int[n + 1];

        Arrays.fill(dp, 10001);

        dp[0] = 0;

        for(int i = 1; i <= n; i++) {

            for(int j = 0; j < garden.length; j++) {

                if(j <= i && i <= j + garden[j]) {

                    dp[i] = Math.min(dp[i], dp[j] + 1);

                }

            }

        }

        return dp[n] == 10001 ? -1 : dp[n];

    }

}

Time Complexity: O(NT)

Space Complexity: O(T)

**Refer to**

<https://grandyang.com/leetcode/1326/>

这道题说是在x轴上有一个长度为n的花园，共有等距离的 n+1 喷头，现在有一个 ranges 数组，说是第i个喷头可以覆盖的范围是 [i - ranges[i], i + ranges[i]]，问最少需要几个喷头可以覆盖整个花园。这道题跟之前那道 [Video Stitching](https://www.cnblogs.com/grandyang/p/14395758.html) 基本上没有太大的区别，就是场景换了一下，本质都是一样的，类似的还有 [Jump Game II](http://www.cnblogs.com/grandyang/p/4373533.html)。这道题可以使用贪婪算法 Greedy Algorithm 和动态规划 Dynamic Programming 来做，先来用贪婪算法，采用和 [Jump Game II](http://www.cnblogs.com/grandyang/p/4373533.html) 一样的思路。

这里将题目也看成跳跃游戏的场景，从位置0开始，看是否能跳到位置n。那么就需要知道每个点的跳力，于是来构建 jumps 数组。对于每个喷头，题目说了其覆盖范围是 [i - ranges[i], i + ranges[i]]，为了防止其越界，左边界点 left 就取 max(0, i - ranges[i])，右边界点 right 就取 min(n, i + ranges[i])，这样站在左边界点 left 上，其跳力就是 right - left，每次用这个来更新 jumps[left] 即可。有了 jumps 数组就可以开始计算最小跳跃数了，新建变量 farCanReach 表示当前最远能到达的位置，last 表示上次到达的最远位置。

然后开始遍历每个位置，若当前位置i大于 farCanReach，表示无法到达当前位置，即喷头无法覆盖整个花园，直接返回 -1。否则就用 i + jums[i] 来更新 farCanReach，因为当前的位置加上其跳力，就是其能到达的最远距离，若当前位置i等于 last 了，表示已经到达了上次跳跃的最远位置了，此时需要开始新的跳跃了，跳跃次数增1，last 更新为 farCanReach。以此类推，直到 for 循环推出后，判断若 farCanReach 大于等于n的话，表示可以到达末尾，返回结果 res，否则返回 -1 即可，参见代码如下：

解法一：

    class Solution {

        public:

        int minTaps(int n, vector<int>& ranges) {

            vector<int> jumps(n + 1);

            for (int i = 0; i < n + 1; ++i) {

                int left = max(0, i - ranges[i]);

                int right = min(n, i + ranges[i]);

                jumps[left] = max(jumps[left], right - left);

            }

            int res = 0, last = 0, farCanReach = 0;

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

                if (i > farCanReach) return -1;

                farCanReach = max(farCanReach, i + jumps[i]);

                if (i == last) {

                    ++res;

                    last = farCanReach;

                }

            }

            return farCanReach >= n ? res : -1;

        }

    };

再来看动态规划的解法，建立一个一维数组 DP，其中 dp[i] 表示覆盖范围 [0, i] 区间需要的最少喷头个数，最终结果会保存在 dp[n] 中。dp 数组长度为 n+1，每个数字初始化为 n+1，但 dp[0] 要初始化为0。接下来求状态转移方程，使用和 [Video Stitching](https://www.cnblogs.com/grandyang/p/14395758.html) 中类似的方法，遍历 [0, n] 中的每一个喷头，然后计算其能覆盖范围的左右边界，注意还是要进行防越界处理，跟上面的解法相同。计算出喷头i的覆盖的范围 [left, right] 之后，遍历其范围内的每一个位置j，由于该区间内的每个点都是可以到达的，即相当于在 dp[left] 的基础上又增加了一个喷头，所以可以用 1 + dp[left] 来更新 dp[j]，这就是状态转移方程。最终循环退出后，判断 dp[n] 是否等于初始值 n+1，是的话返回 -1，否则返回 dp[n] 即可，参见代码如下：

解法二：

    class Solution {

        public:

        int minTaps(int n, vector<int>& ranges) {

            vector<int> dp(n + 1, n + 1);

            dp[0] = 0;

            for (int i = 0; i < n + 1; ++i) {

                int left = max(0, i - ranges[i]);

                int right = min(n, i + ranges[i]);

                for (int j = left; j <= right; ++j) {

                    dp[j] = min(dp[j], 1 + dp[left]);

                }

            }

            return (dp[n] == n + 1) ? -1 : dp[n];

        }

    };

**Refer to**

<https://leetcode.com/problems/minimum-number-of-taps-to-open-to-water-a-garden/solutions/712660/dp-pattern-to-solve-3-similar-problems/>

We can use the same DP pattern to solve the following 3 problems.

a.1326. Minimum Number of Taps to Open to Water a Garden

public int minTaps(int[] ranges) {

    int len = ranges.length;

    int[] dp = new int[len];

    Arrays.fill(dp, len + 1);

    dp[0] = 0;

    for (int i = 0; i < len; i++) {

        int start = Math.max(i - ranges[i], 0);

        int end = Math.min(i + ranges[i], len - 1);

        for (int j = start; j <= end; j++) {

            dp[j] = Math.min(dp[j], dp[start] + 1);

        }

    }

    return dp[len - 1] == len + 1 ? -1 : dp[len - 1];

}

b.45. Jump Game II

public int jump(int[] nums) {

    int len = nums.length;

    int[] dp = new int[len];

    Arrays.fill(dp, len + 1);

    dp[0] = 0;

    for (int i = 0; i < len; i++) {

        int start = i;

        int end = Math.min(i + nums[i], len - 1);

        for (int j = start; j <= end; j++) {

            dp[j] = Math.min(dp[j], dp[start] + 1);

        }

    }

    return dp[len - 1] == len + 1 ? -1 : dp[len - 1];

}

c.1024. Video Stitching

public int videoStitching(int[][] clips, int T) {

    int[] dp = new int[T + 1];

    Arrays.fill(dp, T + 2);

    dp[0] = 0;

    for (int i = 0; i <= T; i++) {

        for (int[] clip : clips) {

            int start = clip[0];

            int end = clip[1];

            if (i >= start && i <= end)

                dp[i] = Math.min(dp[i], dp[start] + 1);

        }

    }

    return dp[T] == T + 2 ? -1 : dp[T];

}