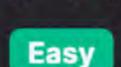
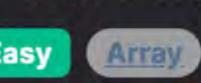
744. Find Smallest Letter Greater Than Target





Problem Description

Binary Search

Leetcode Link

The problem presents an array letters of sorted characters in non-decreasing order and a target character. The task is to find the smallest character in the array that is lexicographically greater than the target character. If no such character exists in the array, the requirement is to return the first character in letters. Lexicographically greater means the character should come after the target character in the alphabet. It is guaranteed that the array contains at least two different characters.

Intuition

to quickly eliminate half of the search space at each step. We search for the point where we can find the first character in letters that is greater than target. Here's a step-by-step intuition:

The intuitive approach to solving this problem is to use a binary search because the characters are sorted, which makes it possible

2. While the left pointer is less than the right pointer, we repeatedly divide the search space in half. We calculate a middle index

1. Initialize two pointers, left and right, at the beginning and at the end (plus one) of the array respectively.

- mid by averaging left and right. 3. We compare the character at the middle index with the target using their ASCII values (through the ord function).
- 4. If the character at the middle index is greater than the target character, we have to keep searching in the left part of the array
- to find the smallest character larger than target, so we move the right pointer to mid. 5. If the character in the middle is less than or equal to target, we have to search in the right part of the array, so we increment the
- 6. When the left and right pointers meet, they point at the smallest character greater than the target if such a character exists in the array. If such character does not exist, left will be equal to len(letters), meaning we need to wrap around to the start of
- the array. 7. To address the wrapping, we use modulus operation left % len(letters). This ensures that if left is beyond the last index, we return the first character in the array.
- This approach efficiently narrows down the search space using the properties of sorted arrays and direct character comparisons, allowing us to find the answer in logarithmic time complexity, which is much faster than linearly scanning through the array.

Solution Approach

The implementation uses a binary search algorithm, which is efficient for sorted arrays. The binary search algorithm repeatedly divides the search space in half, gradually narrowing down the section of the array where the answer could be, thereby reducing the

left pointer to mid + 1.

number of comparisons that need to be made. Here's how the provided code works: 1. Initialization: left is set to 0 (the start of the array), and right is set to len(letters), which is one past the end of the array.

2. Binary Search Loop:

• The condition while left < right ensures that the loop runs until left and right meet.

• mid = (left + right) >> 1 finds the middle index. The >> 1 is a bit shift to the right by one position, effectively dividing the sum by 2, but faster.

3. Character Comparison:

if ord(letters[mid]) > ord(target): If the character at the mid index in letters is lexicographically greater than target,

search to the left by updating right = mid.

- else: statement means the mid character is not greater than target, so we search to the right by updating left = mid + 1. 4. Finding the Answer:
 - After the loop exits, left is the index of the smallest character that is lexicographically greater than the target. If such a character doesn't exist, left will be equal to len(letters), indicating that we have searched the entire array without finding

a character greater than target.

5. Return Statement: o return letters[left % len(letters)] ensures we return the correct character: • If left is less than len(letters), it means we've found a character that is greater than target, and we return that

character. • If left is equal to len(letters), the modulus operation causes it to wrap around to 0, returning the first character of the

- array, which is the required behavior when a greater character isn't found.
- In terms of data structures, only the input array letters is used, and pointers (indices) are manipulated to traverse the array. No additional data structures are required for this algorithm, which keeps the space complexity low.
- significantly more efficient than a linear search, which would have a time complexity of O(n).

Example Walkthrough Let's apply the solution approach to a small example:

By implementing the binary search pattern, the time complexity is $O(\log n)$, where n is the length of the input array. This is

Assume we have the sorted array letters = ["c", "f", "j"] and the target character target = "a".

Now, let's walk through the binary search algorithm step by step:

2. Binary Search Loop:

1. Initialization: We set left to 0 and right to 3 (the length of letters is 3).

 Begin the loop since left < right (0 < 3). Calculate mid which is (0 + 3) >> 1, yielding 1.

At index 1, the character is "f". We compare "f" with target which is "a".

3. Character Comparison:

 Since ord("f") (102) is greater than ord("a") (97), we set right to mid, which is now 1. 4. Continue Binary Search Loop:

The loop condition still holds as left < right (0 < 1).

Calculate mid again, which is (0 + 1) >> 1, yielding 0.

 Compare character at index 0, "c", with target. Since ord("c") (99) is greater than ord("a") (97), we set right to mid, which is now 0.

5. Loop Ends: The loop now terminates since left is not less than right (both are 0).

modulus operation would wrap around to return the first element in the array, "c".

if ord(letters[mid]) > ord(target):

Otherwise, look to the right half

char nextGreatestLetter(vector<char>& letters, char target) {

int mid = left + (right - left) / 2; // Prevent potential overflow.

// If the middle letter is strictly greater than the target,

// move the right pointer to mid, as we want the smallest letter

// Initialize the pointers for the binary search.

right = mid

left = mid + 1

else:

6. Finding the Answer: The left is 0, and left is less than len(letters).

returned as the smallest lexicographically greater character than target. In this example, the desired character is found within two iterations of the binary search loop, demonstrating the efficiency of the

If the middle letter is greater than the target, look to the left half

def nextGreatestLetter(self, letters: List[str], target: str) -> str: # Initialize the left and right pointers to the start and end of the list respectively left, right = 0, len(letters) # Use binary search to find the position of the next greatest letter while left < right: # Find the middle index mid = (left + right) // 2 # the '>> 1' is a bitwise operation equivalent to integer division by 2

7. Return Statement: We return letters [left % len(letters)], which is letters [0]. The character at index 0 is "c", so "c" is

binary search algorithm in quickly locating the answer. The result, "c", is indeed the smallest character in the array that is

lexicographically greater than the target "a". If the target had been "k", the search would conclude with left being 3, and the

17 # The modulo operation ensures wrapping around if the target letter is greater than any letter in the list 18 19 return letters[left % len(letters)] 20

Java Solution

10

11

12

13

14

15 16

Python Solution

class Solution:

```
class Solution {
       public char nextGreatestLetter(char[] letters, char target) {
           // Initialize the start and end pointers for binary search
           int start = 0;
           int end = letters.length;
           // Perform binary search to find the smallest letter greater than target
           while (start < end) {</pre>
               // Calculate the mid point to split the search into halves
9
               int mid = (start + end) >>> 1; // Using unsigned shift for safe mid calculation
11
12
               // If the middle letter is greater than the target
13
               if (letters[mid] > target) {
                   // We have a new possible candidate for next greatest letter (inclusive)
14
                   // and we need to search to the left of mid (exclusive)
15
                   end = mid;
16
17
               } else {
18
                   // If mid letter is less than or equal to the target,
19
                   // we need to search to the right of mid (exclusive)
                   start = mid + 1;
20
21
22
23
24
           // After the search, start is the least index where letters[index] > target,
           // since the array is circular, we use modulo operator to wrap around the index
25
26
           return letters[start % letters.length];
27
28 }
29
```

20 } else { // If the middle letter is less than or equal to the target, 21 22 // move the left pointer past mid. 23 left = mid + 1;

C++ Solution

1 #include <vector>

class Solution {

int left = 0;

int right = letters.size();

// Find the middle index.

// that is greater than the target.

// Since the list is circular, if we go past the end,

if (letters[mid] > target) {

right = mid;

// Perform binary search.

while (left < right) {</pre>

public:

6

10

11

12

13

14

15

16

17

18

19

24

25

26

27

```
28
           // we return the first element (modulo operation).
29
           return letters[left % letters.size()];
30
31 };
32
Typescript Solution
   function nextGreatestLetter(letters: string[], target: string): string {
       // Initialize the number of elements in the 'letters' array.
       const numLetters = letters.length;
       // Set the initial search interval between the start and end of the array.
       let leftIndex = 0;
       let rightIndex = numLetters;
       // Perform a binary search to find the smallest letter greater than the target.
 8
       while (leftIndex < rightIndex) {</pre>
 9
           // Calculate the middle index.
10
           let middleIndex = leftIndex + ((rightIndex - leftIndex) >>> 1); // Same as Math.floor((left + right) / 2)
11
12
13
           // If the letter at the middle index is greater than the target,
           // it could be a potential answer, so move the right index to the middle.
14
15
           // Otherwise, move the left index to one position after the middle.
           if (letters[middleIndex] > target) {
16
               rightIndex = middleIndex;
17
           } else {
                leftIndex = middleIndex + 1;
19
20
21
22
23
       // Since the letters wrap around, we use the modulo operator (%) with the number of letters.
       // This ensures we get a valid index if the 'leftIndex' goes beyond the array bounds.
24
       return letters[leftIndex % numLetters];
25
```

Time and Space Complexity

26 } 27

narrow down the search space. Since the search space is divided by two in every step, the maximum number of steps is proportional

Space Complexity The space complexity of the code is 0(1). This is because the space required does not scale with the size of the input list. The code

same.

Time Complexity The provided code performs a binary search on the sorted list of letters to find the smallest letter in the list that is larger than the target character. The time complexity of binary search is O(log n), where n is the number of elements in the input list. This is because, with each comparison, it effectively halves the size of the search space. Initially, the search space is the entire letters list. With each iteration of the loop, either the left or right index is adjusted to to log2(n). Therefore, the time complexity is O(log n).

uses a constant amount of extra space for variables such as left, right, and mid. There are no additional data structures or

recursive calls that would increase the space complexity. Regardless of the list size, the amount of memory required remains the