Edition
1

LEETCODE

50 COMMON INTERVIEW QUESTIONS

Clean Code Handbook

LEETCODE

Clean Code Handbook

---

admin@leetcode.com

Chapter 0: Foreword

Hi, fellow LeetCoders:

First, I would like to express thank you for buying this eBook: "Clean Code Handbook: 50 Common Interview Questions". As the title suggested, this is the definitive guide to write clean and concise code for interview questions. You will learn how to write clean code. Then, you will ace the coding interviews.

This eBook is written to serve as the perfect companion for LeetCode Online Judge (OJ).

Each problem has a "Code it now" link that redirects to the OJ problem statement page. You can write the code and submit it to the OJ system, which you will get immediate feedback on whether your solution is correct. If the "Code it now" link says "Coming soon", it means the problem will be added very soon in the future, so stay tuned.

On the top right side of each problem are the Difficulty and Frequency ratings. There are three levels of difficulty: Easy, Medium, and Hard. Easy problems are problems that are easy to come up with ideas and the implementation should be pretty straightforward. Most interview questions fall into this level of difficulty.

On the other end, there are hard problems. Hard problems are usually algorithmic in nature and require more thoughts beforehand. There could be some coding questions that fall into Hard, but that is rare.

There are three frequency rating: Low, Medium, and High. The frequency of a problem being asked in a real interview is based on data collected from the user survey: "*Have you met this question in a real interview?*" This should give you an idea of what kind of questions is currently being asked in real interviews.

Each question may contain a section called "Example Questions Candidate Might Ask". These are examples of good questions to ask your interviewer. Asking clarifying questions is important and is a good chance to demonstrate your thought process.

Each question is accompanied with as many approaches as possible. Each approach begins with a runtime and space complexity summary so you can quickly compare between different approaches. The analysis of the runtime and space complexity is usually provided along with the solution. Analyzing complexity is frequently being asked in a technical interview, so you should definitely prepare for it.

You learn best when you solve a problem by yourself. If you get stuck, there are usually hints in the book to help you. If you are still stuck, read the analysis and try to write the code yourself in the Online Judge

Even though you might think a problem is *easy*, writing code that is concise and clean is ***not*** as easy as most people think. For example, if you are writing more than 30 lines of code during a coding interview, your code is probably not concise enough. Most code in this eBook fall between 20 — 30 lines of code.

1337c0d3r

Chapter 1: Array/String

1. Two Sum

Code it now: https://oj.leetcode.com/problems/two-sum/ Difficulty: Easy, Frequency: High

Question:

Given an array of integers, find two numbers such that they add up to a specific target number.

The function *twoSum* should return indices of the two numbers such that they add up to the target, where *index1* must be less than *index2*. Please note that your returned answers (both *index1* and *index2*) are not zero-based.

You may assume that each input would have ***exactly*** one solution.

Solution:

***O*(n^2) runtime, *O*(1) space – Brute force:**

The brute force approach is simple. Loop through each element *x* and find if there is another value that equals to *target* – *x*. As finding another value requires looping through the rest of array, its runtime complexity is $O(n^2)$.

***O*(*n*) runtime, *O*(*n*) space – Hash table:**

We could reduce the runtime complexity of looking up a value to *O*(1) using a hash map that maps a value to its index.

```
public int[] twoSum(int[] numbers, int target) {
   Map<Integer, Integer> map = new HashMap<>();
   for (int i = 0; i < numbers.length; i++) {
      int x = numbers[i];
      if (map.containsKey(target - x)) {
         return new int[] { map.get(target - x) + 1, i + 1 };
      }
      map.put(x, i);
   }
   throw new IllegalArgumentException("No two sum solution");
}
```

Follow up:

What if the given input is already sorted in ascending order? See Question [2. Two Sum II – Input array is sorted].

2. Two Sum II – Input array is sorted

Code it now: Coming soon! Difficulty: Medium, Frequency: N/A

Question:

Similar to Question [1. Two Sum], except that the input array is already sorted in ascending order.

Solution:

Of course we could still apply the [Hash table] approach, but it costs us $O(n)$ extra space, plus it does not make use of the fact that the input is already sorted.

$O(n \log n)$ runtime, $O(1)$ space – Binary search:

For each element x, we could look up if $target - x$ exists in $O(\log n)$ time by applying binary search over the sorted array. Total runtime complexity is $O(n \log n)$.

```
public int[] twoSum(int[] numbers, int target) {
   // Assume input is already sorted.
   for (int i = 0; i < numbers.length; i++) {
      int j = bsearch(numbers, target - numbers[i],  i + 1);
      if (j != -1) {
         return new int[] { i + 1, j + 1 };
      }
   }
   throw new IllegalArgumentException("No two sum solution");
}

private int bsearch(int[] A, int key, int start) {
   int L = start, R = A.length - 1;
   while (L < R) {
      int M = (L + R) / 2;
      if (A[M] < key) {
         L = M + 1;
      } else {
         R = M;
      }
   }
   return (L == R && A[L] == key) ? L : -1;
}
```

$O(n)$ runtime, $O(1)$ space – Two pointers:

Let's assume we have two indices pointing to the i^{th} and j^{th} elements, A_i and A_j respectively. The sum of A_i and A_j could only fall into one of these three possibilities:

i. $A_i + A_j >$ target. Increasing i isn't going to help us, as it makes the sum even bigger. Therefore we should decrement j.

ii. $A_i + A_j <$ target. Decreasing j isn't going to help us, as it makes the sum even smaller. Therefore we should increment i.

iii. $A_i + A_j ==$ target. We have found the answer.

```java
public int[] twoSum(int[] numbers, int target) {
   // Assume input is already sorted.
   int i = 0, j = numbers.length - 1;
   while (i < j) {
      int sum = numbers[i] + numbers[j];
      if (sum < target) {
         i++;
      } else if (sum > target) {
         j--;
      } else {
         return new int[] { i + 1, j + 1 };
      }
   }
   throw new IllegalArgumentException("No two sum solution");
}
```

3. Two Sum III – Data structure design

Code it now: Coming soon! Difficulty: Easy, Frequency: N/A

Question:

Design and implement a *TwoSum* class. It should support the following operations: *add* and *find*.

add(*input*) – Add the number *input* to an internal data structure.
find(*value*) – Find if there exists any pair of numbers which sum is equal to the *value*.

For example,

add(1); *add*(3); *add*(5); *find*(4) → true; *find*(7) → false

Solution:

***add* – *O*(*n*) runtime, *find* – *O*(1) runtime, *O*(n^2) space – Store pair sums in hash table:**

We could store all possible pair sums into a hash table. The extra space needed is in the order of $O(n^2)$. You would also need an extra $O(n)$ space to store the list of added numbers. Each *add* operation essentially go through the list and form new pair sums that go into the hash table. The *find* operation involves a single hash table lookup in $O(1)$ runtime.

This method is useful if the number of *find* operations far exceeds the number of *add* operations.

***add* – *O*(*n*) runtime, *find* – *O*(*n*) runtime, *O*(*n*) space – Binary search + Two pointers:**

Maintain a sorted array of numbers. First, we search for the correct insert position in $O(\log n)$ time using a modified binary search (See Question [48. Search Insert Position]). Each *add* operation takes $O(n)$ time because all elements after the inserted element need to be shifted. For *find* operation we could then apply the [Two pointers] approach in $O(n)$ runtime.

***add* – *O*(1) runtime, *find* – *O*(*n log n*) runtime, *O*(*n*) space – Sort + Two pointers:**

We could do a slight adjustment to the previous method. We store each input into an array without maintaining its sorted order. For *find* operation we first sort the array in $O(n \log n)$ time, then apply the [Two pointers] approach in $O(n)$ runtime.

***add* – *O*(1) runtime, *find* – *O*(*n*) runtime, *O*(*n*) space – Store input in hash table:**

A simpler approach is to store each input into a hash table with the input as key and its count as value. To find if a pair sum exists, just iterate through the hash table in $O(n)$ runtime. Make sure you are able to handle duplicates correctly.

```java
public class TwoSum {
  private Map<Integer, Integer> table = new HashMap<>();

  public void add(int input) {
    int count = table.containsKey(input) ? table.get(input) : 0;
    table.put(input, count + 1);
  }

  public boolean find(int val) {
    for (Map.Entry<Integer, Integer> entry : table.entrySet()) {
      int num = entry.getKey();
      int y = val - num;
      if (y == num) {
        // For duplicates, ensure there are at least two individual numbers.
        if (entry.getValue() >= 2) return true;
      } else if (table.containsKey(y)) {
        return true;
      }
    }
    return false;
  }
}
```

4. Valid Palindrome

Code it now: https://oj.leetcode.com/problems/valid-palindrome/ Difficulty: Easy, Frequency: Medium

Question:

Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring cases.

For example,

"A man, a plan, a canal: Panama" is a palindrome.

"race a car" is *not* a palindrome.

Example Questions Candidate Might Ask:

Q: What about an empty string? Is it a valid palindrome?
A: For the purpose of this problem, we define empty string as valid palindrome.

Solution:

O(*n*) runtime, *O*(1) space:

The idea is simple, have two pointers – one at the head while the other one at the tail. Move them towards each other until they meet while skipping non-alphanumeric characters.

Consider the case where given string contains only non-alphanumeric characters. This is a valid palindrome because the empty string is also a valid palindrome.

```
public boolean isPalindrome(String s) {
   int i = 0, j = s.length() - 1;
   while (i < j) {
      while (i < j && !Character.isLetterOrDigit(s.charAt(i))) i++;
      while (i < j && !Character.isLetterOrDigit(s.charAt(j))) j--;
      if (Character.toLowerCase(s.charAt(i))
            != Character.toLowerCase(s.charAt(j))) {
         return false;
      }
      i++; j--;
   }
   return true;
}
```

5. Implement strstr()

Code it now: https://oj.leetcode.com/problems/implement-strstr/ Difficulty: Easy, Frequency: High

Question:

Implement *strstr*(). Returns the index of the first occurrence of *needle* in *haystack*, or –1 if *needle* is not part of *haystack*.

Solution:

***O*(*nm*) runtime, *O*(1) space – Brute force:**

There are known efficient algorithms such as Rabin-Karp algorithm, KMP algorithm, or the Boyer-Moore algorithm. Since these algorithms are usually studied in an advanced algorithms class, it is sufficient to solve it using the most direct method in an interview – The *brute force method*.

The brute force method is straightforward to implement. We scan the *needle* with the *haystack* from its first position and start matching all subsequent letters one by one. If one of the letters does not match, we start over again with the next position in the *haystack*.

Assume that *n* = length of *haystack* and *m* = length of *needle*, then the runtime complexity is *O*(*nm*).

Have you considered these scenarios?

i. *needle* or *haystack* is empty. If *needle* is empty, always return 0. If *haystack* is empty, then there will always be no match (return –1) unless *needle* is also empty which 0 is returned.

ii. *needle*'s length is greater than *haystack*'s length. Should always return –1.

iii. *needle* is located at the end of *haystack*. For example, "aaaba" and "ba". Catch possible off-by-one errors.

iv. *needle* occur multiple times in *haystack*. For example, "mississippi" and "issi". It should return index 2 as the *first* match of "issi".

v. Imagine two very long strings of equal lengths = *n*, *haystack* = "aaa…aa" and *needle* = "aaa…ab". You should not do more than *n* character comparisons, or else your code will get Time Limit Exceeded in OJ.

Below is a clean implementation – no special if statements for all the above scenarios.

```
public int strStr(String haystack, String needle) {
   for (int i = 0; ; i++) {
      for (int j = 0; ; j++) {
         if (j == needle.length()) return i;
         if (i + j == haystack.length()) return -1;
         if (needle.charAt(j) != haystack.charAt(i + j)) break;
      }
   }
}
```