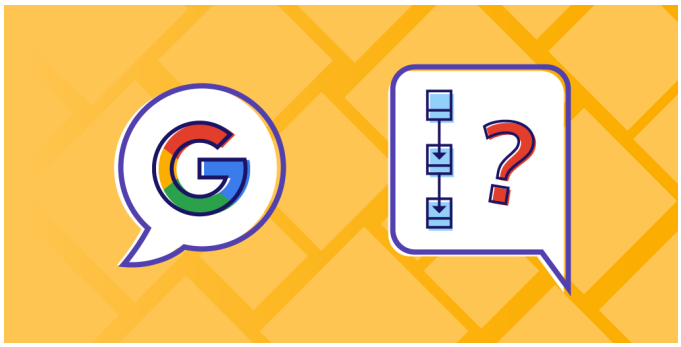
Google Coding Interview Questions: top 18 questions explained

Jul 31, 2020 - 17 min read

Aaron Xie



Google Coding Interview Series

- The definitive prep guide to the interview process
- Top Google coding questions explained

Landing a tech job at Google is no easy feat. Like other large tech giants, Google's software engineering interview process is comprehensive and difficult, requiring weeks of preparation and practice. So, how can you prepare for the coding interview? What interview questions should prepare for?

In our first part of this series, we talked about the Google interview process and explained how to succeed in behavioral interview. Today we're going to provide a part 2 to our Google Coding interview series, and we'll breakdown 18 common interview questions asked by Google recruiters.

Today we'll cover the following:

Google coding interview recap

The entire **Google interview** process takes around two months to complete and consists of five interviews in total. For the interviews, there will primarily be three question formats: system design, general analysis, and technical skills. To learn more about the interview process, check out our Google Coding Interview prep guide.

Prescreen: The first interview consists of a prescreening with a Google employee, which will last 45 - 60 minutes. In this interview, the interviewer will test you on your knowledge of data structures and algorithms.

On-site interviews: If you past the prescreen, you will be invited to an on-site interview, in which you will be interviewed by 4-6 employees for 45 minutes each. These interviews will more rigorously test your knowledge of data structures and algorithms. Typically, you will be writing code on a Google Doc or whiteboard.

Decision: Based on your performance, you will be scored on a scale between 1 and 4, with 3 being the minimum threshold to hire.

Overall, Google wants to test your technical and logical application of computer science. Here are some popular topics to review before your interview:

- Algorithms
- Sorting
- Data structure
- Graphs
- Recursion
- Object-oriented programming
- Big-O notation

18 common Google coding interview questions

1. Find the kth largest element in a number stream

Design a class to efficiently find the `Kth` largest element in a stream of numbers. The class should have the following two things:

- The constructor of the class should accept an integer array containing initial numbers from the stream and an integer `K`.
- The class should expose a function add(int num) which will store the given number and return the `Kth` largest number.

Example:

```
Input: [4, 1, 3, 12, 7, 14], K = 3
1. Calling add(6) should return '7'.
2. Calling add(13) should return '12'.
2. Calling add(4) should still return '12'.
```

Try solving this problem yourself:

```
class KthLargestNumberInStream:
  def __init__(self, nums, k):
    # TODO: Write your code here
    self.k = k

  def add(self, num):
    # TODO: Write your code here
    return -1


def main():

  kthLargestNumber = KthLargestNumberInStream([3, 1, 5, 12, 2, 11], 4)
  print("4th largest number is: " + str(kthLargestNumber.add(6)))
  print("4th largest number is: " + str(kthLargestNumber.add(13)))
  print("4th largest number is: " + str(kthLargestNumber.add(4)))


main()
```

Run

Solution

```
from heapq import *


class KthLargestNumberInStream:
  minHeap = []

  def __init__(self, nums, k):
    self.k = k
    # add the numbers in the min heap
    for num in nums:
      self.add(num)

  def add(self, num):
    # add the new number in the min heap
    heappush(self.minHeap, num)

    # if heap has more than 'k' numbers, remove one number
    if len(self.minHeap) > self.k:
      heappop(self.minHeap)

    # return the 'Kth largest number
    return self.minHeap[0]


def main():

  kthLargestNumber = KthLargestNumberInStream([3, 1, 5, 12, 2, 11], 4)
  print("4th largest number is: " + str(kthLargestNumber.add(6)))
  print("4th largest number is: " + str(kthLargestNumber.add(13)))
  print("4th largest number is: " + str(kthLargestNumber.add(4)))


main()
```

Runtime complexity: $O(log(K))$

Space complexity: $O(K)$

To solve this problem, we will follow the common Top 'K' Elements pattern. So, we want to use a min-heap instead of a max-heap, which would be used for Kth smallest number. As we know, the root is the smallest element in the min heap. So, we can compare every number with root as we iterate through each number. If the number is bigger than root, we will swap the two. We will repeat this process until we have iterated through ever number.

2. Find `k` closest numbers

Given a sorted number array and two integers 'K' and 'X', find 'K' closest numbers to 'X' in the array. Return the numbers in the sorted order. 'X' is not necessarily present in the array.

Example:

```
Input: arr = [1,2,3,4,5], k = 4, x = 3
Output: [1,2,3,4]

Input: [2, 4, 5, 6, 9], k = 3, x = 6
Output: [4, 5, 6]
```

Try solving this problem yourself:

```
def find_closest_element(arr, K, X):
  result = []
  # TODO: Write your code here
  return result
```

Test Show Solution

Solution

Use the code widget above to see the solution to the coding question.

Runtime complexity: $O(K)$

Space complexity: $O(log(n-k))$

Once again this problem follows the Top 'K' Numbers pattern. Here is our approach to the problem:

- Since the array is already sorted, we can first find the number closest to `X` through binary search. Let's say that number is `Y`.
- The `K` closest numbers to `K` adjacent to `Y` in the array. We can search both sides of `Y` to find the closest numbers.
- Then, we can use a heap to efficiently search for the closest numbers. We can take `K` numbers in both directions of `Y` and push them in a min-heap sorted by their difference from X.
- Finally, we extract the top `K` numbers from the min-heap to find the required numbers.

3. Sum of two values

Given an array of integers and a value, determine if there are any two integers in the array whose sum is equal to the given value. Return true if the sum exists and return false if it does not.

Example

```
Given nums = [2, 7, 11, 15], target = 9,

Because nums[0] + nums[1] = 2 + 7 = 9,
return [0, 1].
```

Try it yourself

```
def find_sum_of_two(A, val):
 #TODO: Write - Your - Code
  return
```

Test Show Solution

Solution

Runtime complexity: $O(N)$

Space complexity: $O(N)$

In this solution, you can use the following algorithm to find a pair that add up to the target (say `val`).

- Scan the whole array once and store visited elements in a hash set. During scan, for every element `e` in the array, we check if `val - e` is present in the hash set i.e. `val - e` is already visited.
- If `val - e` is found in the hash set, it means there is a pair (`e`, `val - e`) in array whose sum is equal to the given `val`.
- If we have exhausted all elements in the array and didn't find any such pair, the function will return `false`.

4. Delete node with a given key

We are given the head of a linked list and a key. We have to delete the node that contains this given key. The following two examples elaborate on this problem further.

Example

```
Input: head = [4,5,1,9], key = 5
Output: [4,1,9]
Explanation: You are given the second node with value 5, the linked list shoul
```

Try it yourself

```
def delete_node(head, key):
  #TODO: Write - Your - Code
  return
```

Test | Show Solution

Solution

Runtime complexity: $O(n)$

Space complexity: $O(1)$

First, we have to find the key in the linked list. We'll keep two pointers, current and previous, as we iterate the linked list.

If the key is found in the linked list, then the current pointer would be pointing to the node containing the key to be deleted. The previous should be pointing to the node before the key node.

This can be done in a linear scan and we can simply update current and previous pointers as we iterate through the linked list.

5. Copy linked list with arbitrary pointer

You are given a linked list where the node has two pointers. The first is the regular `next` pointer. The second pointer is called `arbitrary_pointer` and it can point to any node in the linked list. Your job is to write code to make a deep copy of the given linked list. Here, deep copy means that any operations on the original list (inserting, modifying and removing) should not affect the copied list.

Example

```
Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
Output: [[7,null],[13,0],[11,4],[10,2],[1,0]]
```

Try it yourself

```
def deep_copy_arbitrary_pointer(head):
   #TODO: Write - Your - Code
  return None
```

Test

Solution

Runtime complexity: $O(N)$

Space complexity: $O(N)$

Here is how the solution algorithm works to find a paid that adds up to the target.

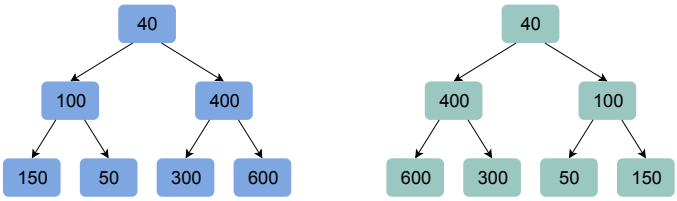
For this problem, we will use a map to track the arbitrary nodes pointed by the original list. Then, we create a deep copy of the original linked list in two passes.

- For the first pass, we create a copy of the original linked list. When create the new copy, use the same values for data and arbitrary_pointer in the copied list. Furthermore, it's important to keep updating the map with entries where the key is the address to the old node and the value is the address of the new node.
- Once we've created the copy, again we'll pass the copied linked list and update the arbitrary pointers to the new address created in the first pass.

6. Mirror binary tree nodes

Given the root node of a binary tree, swap the 'left' and 'right' children for each node. The below example shows how the mirrored binary tree should look like.

Example



Try it yourself

```
def mirror_tree(root):
  #TODO: Write - Your - Code
  return root
```

Test

Solution

Runtime complexity: $O(N)$

Space complexity: $O(N)$

This problem is quite straightforward. For this algorithm, we will utilize a post order traversal of the binary tree. For each node, we will swap its left and right child.

7. Check if two binary trees are identical

Given the roots of two binary trees, determine if these trees are identical or not. Identical trees have the same layout and data at each node.

Example

```
Input:     1         1
          / \       / \
         2   3     2   3

        [1,2,3],   [1,2,3]

Output: true

Input:     1         1
          / \       / \
         2   1     1   2

        [1,2,1],   [1,1,2]

Output: false
```

Try it yourself

```
def are_identical(root1, root2):
  if root1 == None and root2 == None:
    return True

  if root1 != None and root2 != None:
    return (root1.data == root2.data and
              are_identical(root1.left, root2.left) and
              are_identical(root1.right, root2.right))

  return False

arr1 = [100,50,200,25,125,350]
arr2 = [1,2,10,50,180,199]
root1 = create_BST(arr1)
display_level_order(root1)
root2 = create_BST(arr2)

display_level_order(root2)
if(are_identical(root1, root2)):
  print("The trees are identical")
else:
  print("The trees are not identical")
```

Test | Show Solution

Solution

Runtime complexity: $O(N)$

Space complexity: $O(N)$

This problem can be tackled using recursion. The base case of the recursive solution would be when both nodes being compared are null or one of them is null.

Two trees being compared are identical if:

- data on their roots is the same
- the left sub-tree of ‘A’ is identify to the left sub-tree of ‘B’
- the right sub-tree of ‘A’ is identical to the right sub-tree of ‘B’

Using recursion, we can solve this problem through a depth-first traversal on both trees simultaneously while comparing the data at each node.

8. String segmentation

You are given a dictionary of words and a large input string. You have to find out whether the input string can be completely segmented into the words of a given dictionary. The following two examples elaborate on the problem further.

Example

```
Input: s = "applepenapple", wordDict = ["apple", "pen"]
Output: true
Explanation: Return true because "applepenapple" can be segmented as "apple pe
             Note that you are allowed to reuse a dictionary word.
```

Try it yourself

```
def can_segment_string(s, dictionary):
  #TODO: Write - Your - Code
  return False
```

Test Show Solution

Solution

Runtime complexity: $O(2^N)$ (without memoization)

Space complexity: $O(N^2)$

This problem can be tackled by segmenting the input strign at every possible index to see if the string can be completely segmented into words in the dictionary. We can use an algorithm as follows:

```
n = length of input string
for i = 0 to n - 1
  first_word = substring (input string from index [0, i] )
  second_word = substring (input string from index [i + 1, n - 1] )
  if dictionary has first_word
    if second_word is in dictionary OR second_word is of zero length, then ret
    recursively call this method with second_word as input and return true if
```

9. Find all palindrome substrings

Given a string find all non-single letter substrings that are palindromes. For instance:

Example

```
Input: "abc"
Output: 3
Explanation: Three palindromic strings: "a", "b", "c".
```

Try it yourself

```
def find_all_palindrome_substrings(input):
  # TODO: Write - Your - Code
  return -1
```

Test

Solution

Runtime complexity: $O(N^2)$ (without memoization)

Space complexity: $O(1)$

We will iterate through each letter in the input string. For each letter, we can find palindromes by expanding to the left and right will we check for even and odd length palindromes. If there are no palindromes, move to the next letter.

We find palindromes by checking if the left and right character are equal. If they are, we print out the palindrome substring.

10. Largest sum subarray

In the array below, the largest sum subarray starts at index 3 and ends at 6, and with the largest sum being 12.

Example

```
Input: [-2,1,-3,4,-1,2,1,-5,4],
Output: 6
Explanation: [4,-1,2,1] has the largest sum = 6.
```

Try it yourself

```
def max_sub_array_of_size_k(k, arr):
  # TODO: Write your code here
  return -1
```

Test | Show Solution

Solution

Runtime complexity: $O(N)$

Space complexity: $O(1)$

We will solve this problem by implementing Kadane's algorithm. The algorithm works by scanning an entire array and at each position find the maximum sum of the subarray ending there. This is achieved by keeping a current_max for the current array index and a global_max. The algorithm is as follows:

```
current_max = A[0]
global_max = A[0]
for i = 1 -> size of A
    if current_max is less than 0
        then current_max = A[i]
    otherwise
        current_max = current_max + A[i]
    if global_max is less than current_max
        then global_max = current_max
```

11. Determine if the number is valid

Given an input string, determine if it makes a valid number or not. For simplicity, assume that white spaces are not present in the input.

4.325 is a valid number.

1.1.1 is NOT a valid number.

222 is a valid number.

22. is NOT a valid number.

0.1 is a valid number.

22.22. is NOT a valid number.

Try it yourself

```
def is_number_valid(s):
  #TODO: Write - Your - Code
  return False
```

Test | Show Solution

Solution

Runtime complexity: $O(N)$

Space complexity: $O(1)$

To check if a number is valid, we'll use the state machine below. The initial state is start, and we'll process each character to identify the next state. If the state ever ends up at unknown or in a decimal, the number is not valid.

add state machine visual

12. Print balanced brace combinations

Print all braces combinations for a given value `n` so that they are balanced. See the example below.

```
For example, given n = 3, a solution set is:

[
  "((()))",
  "(()())",
  "(())()",
  "()(())",
  "()()()"
]
```

Try it yourself

```
def print_all_braces(n):
  #TODO: Write - Your - Code

  return
```

Test | Show Solution

Solution

Runtime complexity: $O(2^N)$

Space complexity: $O(N)$

The solution algorithm works by maintaining counts of `left_braces` and `right_braces`. The algorithm can be seen below.

```
left_braces count: 0
right_braces count: 0

if left_braces count is less than n:
  add left_braces and recurse further
if right_braces count is less than left_braces count:
  add right_braces and recurse further
stop recursing when left_braces and right_braces counts are both equal to n
```

13. LRU

Least Recently Used (LRU) is a common caching strategy. It defines the policy to evict elements from the cache to make room for new elements when the cache is full, meaning it discards the least recently used items first.

Example

```
LRUCache cache = new LRUCache( 2 /* capacity */ );

cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // returns 1
cache.put(3, 3);    // evicts key 2
cache.get(2);       // returns -1 (not found)
cache.put(4, 4);    // evicts key 1
cache.get(1);       // returns -1 (not found)
cache.get(3);       // returns 3
cache.get(4);       // returns 4
```

Try it yourself

```
# Linked list operations
# insert_at_tail(self, key, data)
# remove(self, data)
# remove_node(self, node)
# remove_head(self)
# remove_tail(self)
# get_head(self)
# get_tail(self)

class LRUCache:

  def __init__(self, capacity):
    self.capacity = capacity
    self.cache = {}
    self.cache_vals = LinkedList()

  def Set(self, key, value):
    # TODO: Write - Your - Code
    return

  def get(self, key):
    # TODO: Write - Your - Code
      return -1

  def get_cache(self):
    res = ""
    node = self.cache_vals.head
    while node:
      res += "(" + str(node.key) + "," + str(node.data) +  ")"
      node = node.next
    return res
```

Test | Show Solution

Solution

Runtime complexity:

get (hashset): $O(N)$

set (hashset): $O(1)$

deletion at head when adding a new element (linked list): $O(1)$

search for deleting and adding to tail (linked list): $O(N)$

Complexity: $O(N)$

Space complexity: $O(N)$

To implement an LRU cache we use two data structures: a hashmap and a doubly linked list. A doubly linked list helps in maintaining the eviction order and a hashmap helps with O(1) lookup of cached keys. Here goes the algorithm for LRU cache.

```
If the element exists in hashmap
    move the accessed element to the tail of the linked list
Otherwise,
    if eviction is needed i.e. cache is already full
        Remove the head element from doubly linked list and delete its hashmap
    Add the new element at the tail of linked list and in hashmap
Get from Cache and Return
```

Note that the doubly linked list is used to keep track of the most recently accessed elements. The element at the tail of the doubly linked list is the most recently accessed element. All newly inserted elements (in put) go the tail of the list. Similarly, any element accessed (in get operation) goes to the tail of the list.

14. Find Low/High Index

Given an array of integers `nums` sorted in ascending order, find the starting and ending position of a given `target` value.

If the target is not found in the array, return `[-1, -1]`. See the example below.

```
Input: nums = [5,7,7,8,8,10], target = 8
Output: [3,4]
```

Try it yourself

```
def find_low_high_index(arr, key):
 #TODO: Write - Your - Code
  return [-1, 1]
```

Test | Show Solution

Solution

Runtime complexity: $O(LOG(N))$

Space complexity: $O(1)$

Scanning the array in a linear fashion would be highly inefficient because the array size could be in the millions. Instead, we will use binary search twice: once to find the low index and once to find the high index.

Here's the algorithm to find the low index:

- At each step, consider the array between low and high indices and calculate the mid index.
- If the element at mid is greater or equal to the key, the high becomes `mid - 1`. Index at low remains the same.
- If the element at mid is less than the key, low becomes `mid + 1`.
- When low is greater than high, low would be pointing to the first occurrence of the key. If the element at low does not match the key, return -1.

For high index, we can use a similar algorithm with slight changes.

- Switch the low index to `mid + 1` when element at `mid` index is less than or equal to the key.
- Switch the high index to `mid - 1` when the element at mid is greater than the key.

15. Merge overlapping intervals

Given a collection of intervals, merge all overlapping intervals. See the example below.

```
Example 1:

Input: [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlaps, merge them into [1,6].

Example 2:
Input: [[1,4],[4,5]]
Output: [[1,5]]
Explanation: Intervals [1,4] and [4,5] are considered overlapping.
```

Try it yourself

```
def merge(intervals):
  merged = []
  # TODO: Write your code here
  return merged
```

Test | Show Solution

Solution

Runtime complexity: $O(N)$

Space complexity: $O(N)$

This problem can be solved utilizing a simple linear search algorithm, since we already know that inputs are sorted by starting timestamps. Here's the approach we will take:

- List of input intervals is given, and we'll keep merged intervals in the output list.

For each interval,

- If the input interval is overlapping with the last interval in the output list, we'll merge the two intervals and update the last interval of the output list with the merged interval.
- Otherwise, we'll add an input interval to the output list.

16. Path sum

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum. See below for an example.

```
Given the below binary tree and sum = 22,

      5
     / \
    4   8
   /   / \
  11  13  4
 /  \      \
7    2      1
return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.
```

Try it yourself

```
def has_path(root, sum):
  # TODO: Write your code here
  return False
```

Test | Show Solution

Solution

Runtime complexity: $O(N)$

Space complexity: $O(N)$

This problem can be solved utilizing a simple linear search algorithm, since we already know that inputs are sorted by starting timestamps. Here's the approach we will take:

- List of input intervals is given, and we'll keep merged intervals in the output list.

For each interval,

- If the input interval is overlapping with the last interval in the output list, we'll merge the two intervals and update the last interval of the output list with the merged interval.
- Otherwise, we'll add an input interval to the output list.

17. Find the missing number

We are given an array containing 'n' distinct numbers taken from the range 0 to 'n'. Since the array has only 'n' numbers out of the total 'n+1' numbers, find the missing number. See the example below.

```
Example 1:
Input: [4, 0, 3, 1]
Output: 2

Example 2:
Input: [8, 3, 5, 2, 4, 6, 0, 1]
Output: 7
```

Try it yourself

```
def find_missing(input):
 #TODO: Write - Your - Code
  return
```

Test | Show Solution

Solution

Runtime complexity: $O(N)$

Space complexity: $O(1)$ Let's solve this problem with a linear, $O(N)$, solution using arithmetic series sum formula.

Here is the algorithm:

1. Find the sum 'sum_of_elements' of all the numbers in the array. This would require a linear scan, O(n).
2. Then find the sum 'expected_sum' of first 'n' numbers using the arithmetic series sum formula i.e. (n * (n + 1)) / 2.
3. The difference between these i.e. 'expected_sum - sum_of_elements', is the missing number in the array.

18. Reverse a linked list

Reverse a singly linked list. See the example below.

```
Input: 1->2->3->4->5->NULL
Output: 5->4->3->2->1->NULL
```

Try it yourself

```
def reverse(head):
  reversed_list = head
  #TODO: Write - Your - Code
  return reversed_list
```

Test

Solution

Runtime complexity: $O(N)$

Space complexity: $O(1)$

Let's take a look at the iterative approach.

If the linked list has 0 or 1 nodes, then the current list can be returned as is. If there are two or more nodes, then the iterative approach starts with two pointers:

- `reversed_list`: A pointer to already reversed linked list.
- `list_to_do`: A pointer to the remaining list.

Then, we set `reversed_list->next` to `NULL`. This becomes the last node of the reversed linked list.

For each iteration, the `list_to_do` pointer moves forward. The current node becomes the new head of the reversed linked list. The loop terminates when we hit `NULL`.

More practice and preparation

Congratulations on finishing these problems! To pass the coding interview, it's important that you continue practice. To prepare for the Google Coding Interview, you should brush up on dynamic programming, backtracking, recursion, greedy algorithms, and divide & conquer. Here are some more common coding interview questions to practice.

19. Determine if the sum of three integers is equal to the given value
20. Intersection point of two linked lists
21. Move zeros to the left
22. Add two integers
23. Merge two sorted linked lists
24. Convert binary tree to doubly linked list
25. level order traversal of binary tree
26. Reverse words in a sentence
29. Find all possible subsets
30. Clone a directed graph
31. Serialize / deserialize binary tree
32. Search rotated array
33. Set columns and rows as zeros
34. Connect all siblings
35. Find all sum combinations
36. Clone a directed graph
37. Closest meeting point
38. Search for the given key in a 2d matrix

27. Find maximum single sell profit
28. Calculate the power of a number

Wrapping up and resources

Cracking the Google coding interview simply comes down to repetition and practice. It's important that you continue practice more interview questions after reading this article. Beyond that, make sure you have a good idea of the structure and process of the Google interview, which you can learn more through Google Coding Interview Part 1.

Tackling these coding challenges comes from a strong foundation of understanding data structures and algorithms, as well as when to apply them. Now, it's time to practice, practice, and practice!

Here are some essential resources for you to continue your preparation for the Google coding interview.

Courses designed for you

- Grokking the Coding Interview: Patterns for Coding Questions
- Grokking the Behavioral Interview

Essential articles to read

- Cracking the Google Coding Interview: the definitive prep guide
- Ace the top 15 Java Algorithms Questions for Coding Interviews
- 5 tried and true techniques to prepare for a coding interview

WRITTEN BY
Aaron Xie