DSA Youtube Learning:

Youtube video link

The Method - Revisited

Here's a systematic strategy we've applied for solving the problem:

1. State the problem clearly. Identify the input & output formats.
2. Come up with some example inputs & outputs. Try to cover all edge cases.
3. Come up with a correct solution for the problem. State it in plain English.
4. Implement the solution and test it using example inputs. Fix bugs, if any.
5. Analyze the algorithm's complexity and identify inefficiencies, if any.
6. Apply the right technique to overcome the inefficiency. Repeat steps 3 to 6.

- Linear Search
- Binary Search
- Big O Notation
- Problem 2:

You are given list of numbers, obtained by rotating a sorted list an unknown number of times. Write a function to determine the minimum number of times the original sorted list was rotated to obtain the given list. Your function should have the worst-case complexity of `O(log N)`, where N is the length of the list. You can assume that all the numbers in the list are unique.

Example: The list `[5, 6, 9, 0, 2, 3, 4]` was obtained by rotating the sorted list `[0, 2, 3, 4, 5, 6, 9]` 3 times.

We define "rotating a list" as removing the last element of the list and adding it before the first element. E.g. rotating the list `[3, 2, 4, 1]` produces `[1, 3, 2, 4]`.

"Sorted list" refers to a list where the elements are arranged in the increasing order e.g. `[1, 3, 5, 7]`

When a sorted list is rotated k times then the smallest number ends up at position k(counting from 0)

- Problem 3:

> **QUESTION 1**: As a senior backend engineer at Jovian, you are tasked with developing a fast in-memory data structure to manage profile information (username, name and email) for 100 million users. It should allow the following operations to be performed efficiently:
>
> 1. **Insert** the profile information for a new user.
> 2. **Find** the profile information of a user, given their username
> 3. **Update** the profile information of a user, given their usrname
> 4. **List** all the users of the platform, sorted by username
>
> You can assume that usernames are unique.

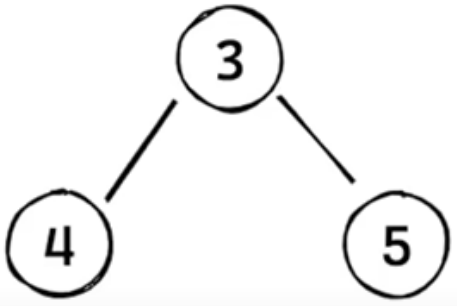
Exercise: What is the purpose of defining the functions `__str__` and `__repr__` within a class? How are the two functions different? Illustrate with some examples using the empty cells below.

Quiz 2: Which tree-based data structure is used to store the index in the Windows file system (NTFS). Who invented this data structure?

- Problem 4:

> **QUESTION 2**: Implement a binary tree using Python, and show its usage with some examples.

To begin, we'll create simple binary tree (without any of the additional properties) containing numbers as keys within nodes. Here's an example:

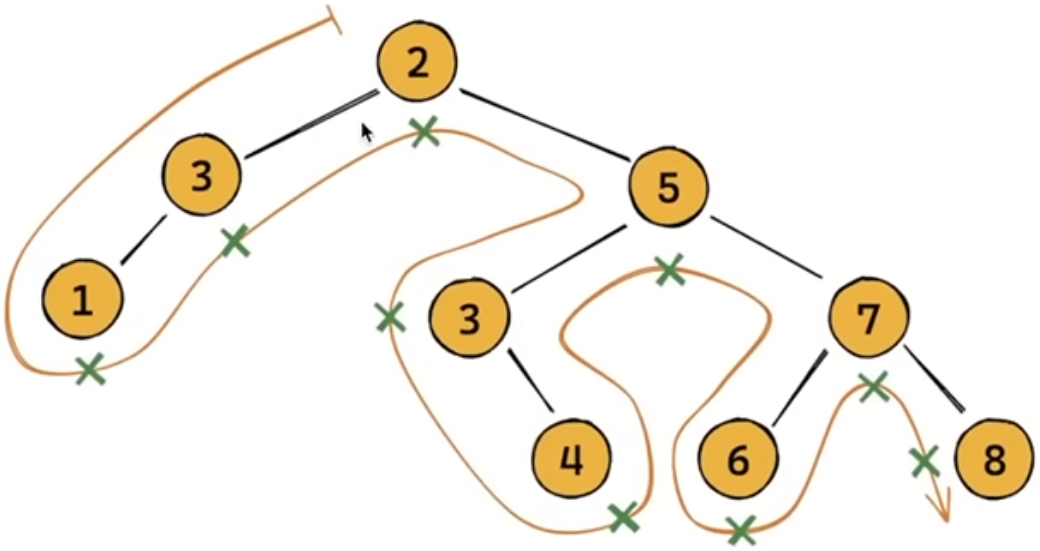


Here's a simple class representing a node within a binary tree.

- Tree Traversal:

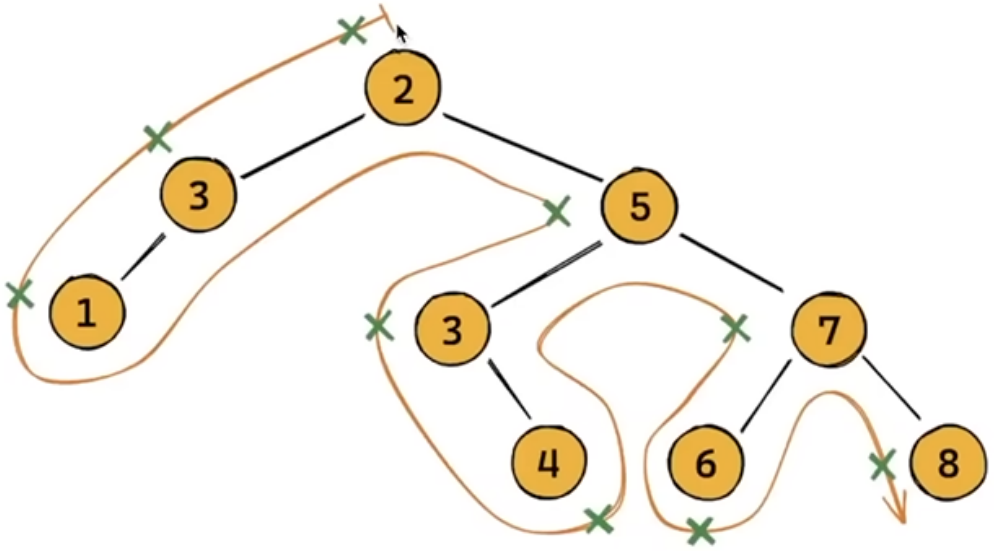
Inorder traversal

1. Traverse the left subtree recursively inorder.
2. Traverse the current node.
3. Traverse the right subtree recursively inorder.



Preorder traversal

1. Traverse the current node.
2. Traverse the left subtree recursively preorder.
3. Traverse the right subtree recursively preorder.



Post Order Trversal

- Binary Search Tree:

Binary Search Tree (BST)

A binary search tree or BST is a binary tree that satisfies the following conditions:

1. The left subtree of any node only contains nodes with keys less than the node's key
2. The right subtree of any node only contains nodes with keys greater than the node's key

It follows from the above conditions that every subtree of a binary search tree must also be a binary search tree.

> **QUESTION 8**: Write a function to check if a binary tree is a binary search tree (BST).

> **QUESTION 9**: Write a function to find the maximum key in a binary tree.

> **QUESTION 10**: Write a function to find the minimum key in a binary tree.

- Problem 5:

> **QUESTION 11**: Write a function to insert a new node into a BST.

We use the BST-property to perform insertion efficiently:

1. Starting from the root node, we compare the key to be inserted with the current node's key
2. If the key is smaller, we recursively insert it in the left subtree (if it exists) or attach it as as the left child if no left subtree exists.
3. If the key is larger, we recursively insert it in the right subtree (if it exists) or attach it as as the right child if no right subtree exists.

Here's a recursive implementation of `insert`.

In [ ]:
```
def insert(node, key, value):
    if node is None:
        node = BSTNode(key, value)
    elif key < node.key:
        node.left = insert(node.left, key, value)
        node.left.parent = node
    elif key > node.key:
        node.right = insert(node.right, key, value)
        node.right.parent = node
    return node
```

- Problem 6:

A Python-Friendly Treemap

We are now ready to return to our original problem statement.

> **QUESTION 1**: As a senior backend engineer at Jovian, you are tasked with developing a fast in-memory data structure to manage profile information (username, name and email) for 100 million users. It should allow the following operations to be performed efficiently:
>
> 1. **Insert** the profile information for a new user.
> 2. **Find** the profile information of a user, given their username
> 3. **Update** the profile information of a user, given their usrname
> 4. **List** all the users of the platform, sorted by username
>
> You can assume that usernames are unique.

We can create a generic class `TreeMap` which supports all the operations specified in the original problem statement in a python-friendly manner.

Hashing Functions

Hashing Function

A *hashing function* is used to convert strings and other non-numeric data types into numbers, which can then be used as list indices. For instance, if a hashing function converts the string `"Aakash"` into the number `4`, then the key-value pair `('Aakash', '7878787878')` will be stored at the position `4` within the data list.

Here's a simple algorithm for hashing, which can convert strings into numeric list indices.

1. Iterate over the string, character by character
2. Convert each character to a number using Python's built-in `ord` function.
3. Add the numbers for each character to obtain the hash for the entire string
4. Take the remainder of the result with the size of the data list

QUESTION 2: Complete the `get_index` function below which implements the hashing algorithm described above.

- Problem 7:

Your objective in this assignment is to implement a `HashTable` class which supports the following operations:

1. **Insert**: Insert a new key-value pair
2. **Find**: Find the value associated with a key
3. **Update**: Update the value associated with a key
4. **List**: List all the keys stored in the hash table

The `HashTable` class will have the following structure (note the function signatures):

In [ ]:
```
class HashTable:
    def insert(self, key, value):
        """Insert a new key-value pair"""
        pass

    def find(self, key):
        """Find the value associated with a key"""
        pass

    def update(self, key, value):
        """Change the value associated with a key"""
```

As you can see above, the value for the key `listen` was overwritten by the value for the key `silent`. Our hash table implementation is incomplete because it does not handle collisions correctly.

To handle collisions we'll use a technique called linear probing. Here's how it works:

1. While inserting a new key-value pair if the target index for a key is occupied by another key, then we try the next index, followed by the next and so on till we the closest empty location.
2. While finding a key-value pair, we apply the same strategy, but instead of searching for an empty location, we look for a location which contains a key-value pair with the matching key.
3. While updating a key-value pair, we apply the same strategy, but instead of searching for an empty location, we look for a location which contains a key-value pair with the matching key, and update its value.

We'll define a function called `get_valid_index`, which starts searching the data list from the index determined by the hashing function `get_index` and returns the first index which is either empty or contains a key-value pair matching the given key.

Sorting

- Problem 8:

> **QUESTION 1**: You're working on a new feature on Jovian called "Top Notebooks of the Week". Write a function to sort a list of notebooks in decreasing order of likes. Keep in mind that up to millions of notebooks can be created every week, so your function needs to be as efficient as possible.

The problem of sorting a list of objects comes up over and over in computer science and software development, and it's important to understand common approaches for sorting, and the trade-offs they offer. Before we solve the above problem, we'll solve a simplified version of the problem:

> **QUESTION 2**: Write a program to sort a list of numbers.

"Sorting" usually refers to "sorting in ascending order", unless specified otherwise.

3. Come up with a correct solution. State it in plain English. ¶

It's easy to come up with a correct solution. Here's one:

1. Iterate over the list of numbers, starting from the left
2. Compare each number with the number that follows it
3. If the number is greater than the one that follows it, swap the two elements
4. Repeat steps 1 to 3 till the list is sorted.

We need to repeat steps 1 to 3 at most `n-1` times to ensure that the array is sorted. Can you explain why? Hint: After one iteration, the largest number in the list.

This method is called **bubble sort**, as it causes smaller elements to *bubble* to the top and larger to *sink* to the bottom. Here's a visual representation of the process:

```
]: def bubble_sort(nums):
    # Create a copy of the list, to avoid changing it
    nums = list(nums)
    print('bubble_sort', nums)

    # 4. Repeat the process n-1 times
    for j in range(len(nums) - 1):
        print('Iteration', j)
        # 1. Iterate over the array (except last element)
        for i in range(len(nums) - 1):
            print('i' , i, nums[i], nums[i+1])
            # 2. Compare the number with
            if nums[i] > nums[i+1]:

                # 3. Swap the two elements
                nums[i], nums[i+1] = nums[i+1], nums[i]

    # Return the sorted list
    return nums
```

5. Analyze the algorithm's complexity and identify inefficiencies

The core operations in bubble sort are "compare" and "swap". To analyze the time complexity, we can simply count the total number of comparisons being made, since the total number of swaps will be less than or equal to the total number of comparisons (can you see why?).

```
for _ in range(len(nums) - 1):
    for i in range(len(nums) - 1):
        if nums[i] > nums[i+1]:
            nums[i], nums[i+1] = nums[i+1], nums[i]
```

There are two loops, each of length `n-1`, where `n` is the number of elements in `nums`. So the total number of comparisons is $(n-1) * (n-1)$ i.e. $(n-1)^2$ i.e. $n^2 - 2n + 1$.

Expressing this in the Big O notation, we can conclude that the time complexity of bubble sort is $O(n^2)$ (also known as quadratic complexity).

Insertion Sort

Before we look at explore more efficient sorting techniques, here's another simple sorting technique called insertion sort, where we keep the initial portion of the array sorted and insert the remaining elements one by one at the right position.

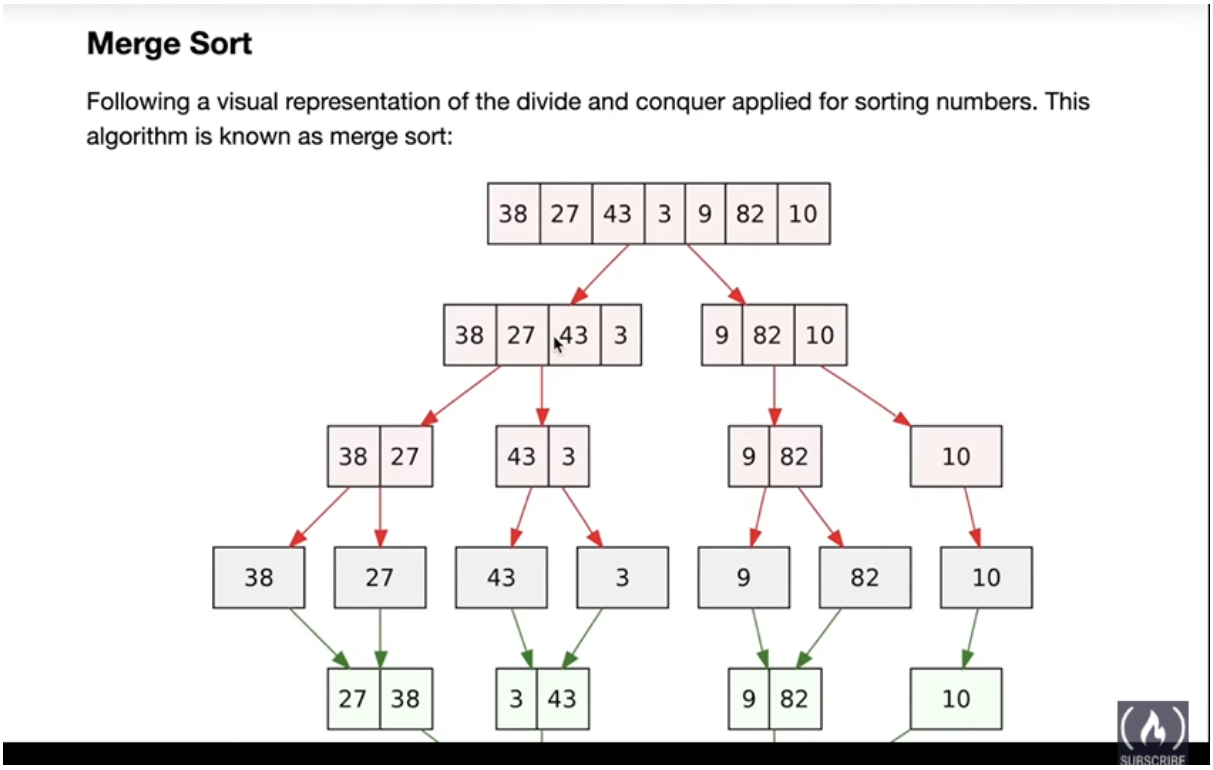
In [ ]:
```
def insertion_sort(nums):
    nums = list(nums)
    for i in range(len(nums)):
        cur = nums.pop(i)
        j = i-1
        while j >=0 and nums[j] > cur:
            j -= 1
        nums.insert(j+1, cur)
    return nums
```

In [ ]:
```
nums0, output0 = test0['input']['nums'], test0['output']

print('Input:', nums0)
print('Expected output:', output0)
result0 = insertion_sort(nums0)
print('Actual output:', result0)
```

Merge Sort

Following a visual representation of the divide and conquer applied for sorting numbers. This algorithm is known as merge sort:



Here's a step-by-step description for merge sort:

1. If the input list is empty or contains just one element, it is already sorted. Return it.
2. If not, divide the list of numbers into two roughly equal parts.
3. Sort each part recursively using the merge sort algorithm. You'll get back two sorted lists
4. Merge the two sorted lists to get a single sorted list

Can you guess how the "merge" operation step 4 works? Hint: Watch this animation: https://youtu.be/GW0USDwhBgo?t=28

> **QUESTION 3**: Write a function to merge two sorted arrays.

Try to explain how the merge operation works in your own words below:

1. ???
2. ???
3. ???
4. ???

10. Apply the right technique to overcome the inefficiency. Repeat Steps 3 to 6.

The fact that merge sort requires allocating additional space as large as the input itself makes it somewhat slow in practice because memory allocation is far more expensive than comparisons or swapping.

Quicksort

To overcome the space inefficiencies of merge sort, we'll study another divide-and-conquer based sorting algorithm called **quicksort**, which works as follows:

1. If the list is empty or has just one element, return it. It's already sorted.
2. Pick a random element from the list. This element is called a *pivot*.
3. Reorder the list so that all elements with values less than or equal to the pivot come before the pivot, while all elements with values greater than the pivot come after it. This operation is called *partitioning*.
4. The pivot element divides the array into two parts which can be sorted independently by making a recursive call to quicksort.

Recursion and Dynamic Programming

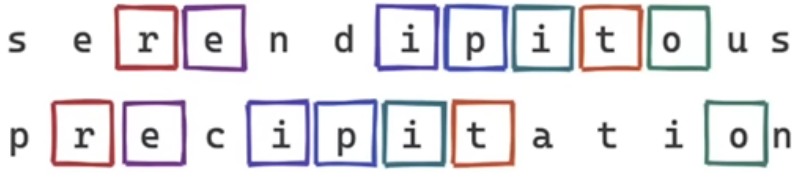
- Problem 9:

Problem Statement

QUESTION 1: Write a function to find the length of the **longest common subsequence** between two sequences. E.g. Given the strings "serendipitous" and "precipitation", the longest common subsequence is "reipito" and its length is 7.

A "sequence" is a group of items with a deterministic ordering. Lists, tuples and ranges are some common sequence types in Python.

A "subsequence" is a sequence obtained by deleting zero or more elements from another sequence. For example, "edpt" is a subsequence of "serendipitous".



- Problem 10:

0-1 Knapsack Problem

Problem statement

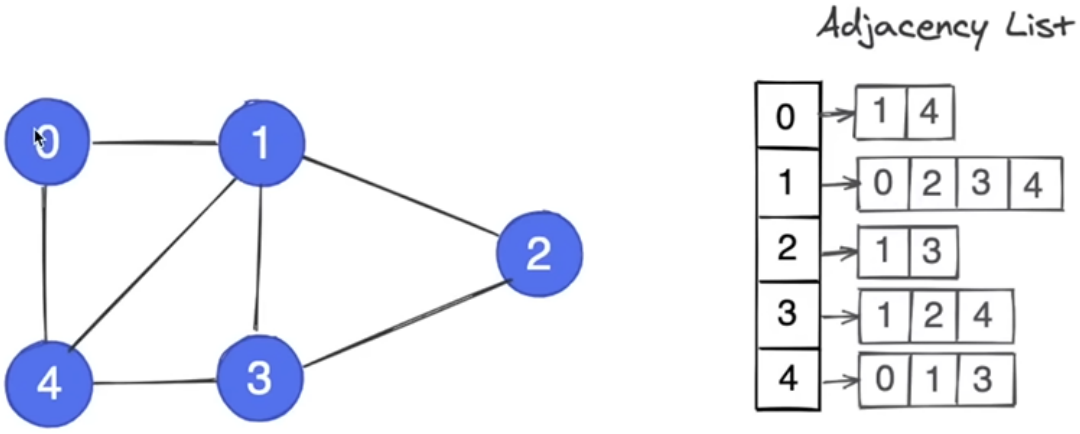
> You're in charge of selecting a football (soccer) team from a large pool of players. Each player has a cost, and a rating. You have a limited budget. What is the highest total rating of a team that fits within your budget. Assume that there's no minimum or maximum team size.

Graph Algorithms

BFS, DFS, and Shortest Path

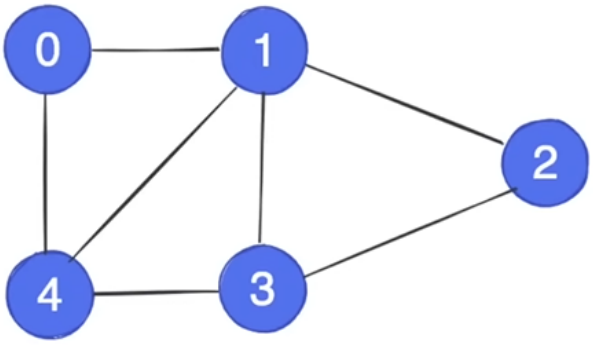
- Problem 11:

Adjacency Lists



> **Question**: Create a class to represent a graph as an adjacency list in Python

- Problem 12:



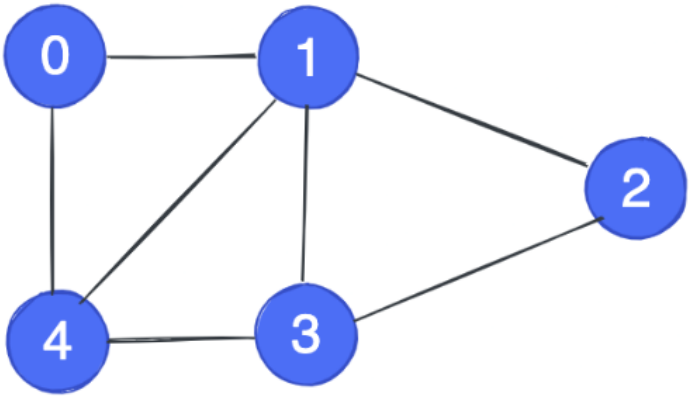
Adjacency Matrix

	0	1	2	3	4
0	0	1	0	0	1
1	1	0	1	1	1
2	0	1	0	1	0
3	0	1	1	0	1
4	1	1	0	1	0

> **Question**: Represent a graph as an adjacency matrix in Python

BFS

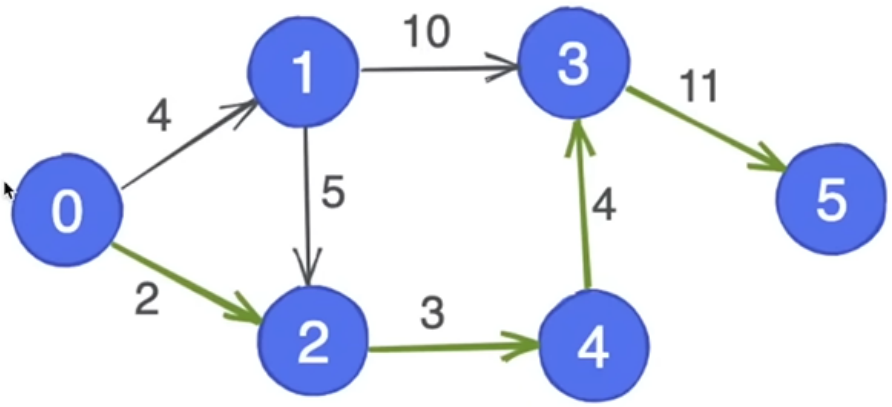
> **Question**: Implement breadth-first search given a source node in a graph using Python.



- Problem 13:

Shortest Paths

> **Question**: Write a function to find the length of the shortest path between two nodes in a weighted directed graph.



Dijkstra's algorithm (Wikipedia):

Dijkstra's Algorithm

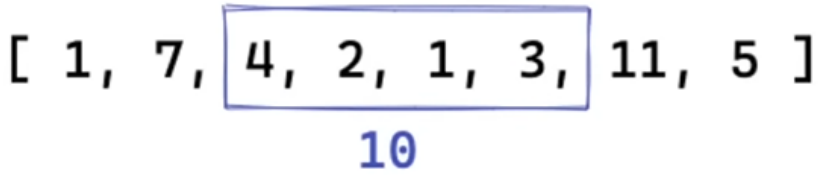
Sample Interview Problem:

- Problem 14:

Subarray with Given Sum

The following question was asked during a coding interview for Amazon:

> You are given an array of numbers (non-negative). Find a continuous subarray of the list which adds up to a given sum.



- Problem 15:

Minimum Edit Distance

The following interview was asked during a coding interview at Google:

> Given two strings A and B, find the minimum number of steps required to convert A to B. (each operation is counted as 1 step.) You have the following 3 operations permitted on a word:
>
> - Insert a character
> - Delete a character
> - Replace a character