Algorithm Plan

Anagram(s, t): An Anagram is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.
1. return sorted(s) == sorted(t)
Time complexity O(n logn)
Space complexity: O(n)
————————————————————

Api testing(requests, JSON): Python requests library for CRUD operations.
```
import requests
# Data to be sent in the POST request (can be a dictionary, JSON, etc.)
headers={“Content-type”: “application/json”}
data = {
    'key1': 'value1',
    'key2': 'value2'
}
response = requests.post(url, data=data, headers=headers)
```
————————————————————

All Numbers unique(list): Write a program which takes a sequence of numbers and check if all numbers are unique.
```
# 1. Convert the list to a set using set() function.
# 2. If the length of the set is equal to the length of the list, return True.
# 3. Otherwise, return False.
  if len(list) == len(set(list)):
     return True
  else:
     return False
```
————————————————————

121. Best Time to Buy and Sell Stock(prices_list): List of prices of stock, want to maximize profit selling in future, return maximum profit or zero if you can’t.
1. keep track of the minimum price found so far and the maximum profit found so far while iterating through the list of prices.
2. For each price, you calculate the profit that could be made by selling at that price (assuming you bought at the minimum price found so far), and update the maximum profit if this profit is higher.
```
def maxProfit(self, prices) -> int:
    min_price = float('inf')
    max_profit = 0
    for price in prices:
        min_price = min(min_price, price)
        profit = price - min_price
        max_profit = max(max_profit, profit)
    return max_profit
```
————————————————————

Binary_gap(n): number 9 has binary representation 1001 and contains a binary gap of length 2. number 32 has binary representation 100000 and has no binary gaps. Find the largest binary gap.
1. Convert number to binary and strip first 2 characters -> bin(n)[2:]
2. Keep 2 counters:
current_gap_length=0 and longest_gap_length=0
3. Iterate over the bits
   4. If bit==0 -> current_gap_length +=1
   5. If bit==1 ->
      longest_gap_length = max(longest_gap_length, current_gap_length)
      current_gap_length = 0 (reset the counter)
6. After iterating over all the bits return longest_gap_length
————————————————————

Binary_search(list, target): Find target in an ordered list. Return index
```
max = len(list) - 1
min = 0
while max >= min:
    mid = (max + min) // 2
    if target == list[mid]:
        return mid
    elif target > list[mid]:
        min = mid + 1
    else:
        max = mid - 1
return -1 >>>> Time complexity O(log n) - Space complexity O(1)
```

```
def binary_search_recursive(arr, target, left, right):
    if left > right:
        return -1
    mid = (left + right) // 2
    if arr[mid] == target:
        return mid
    elif arr[mid] < target:
        binary_search_recursive(arr, target, mid + 1, right)
    else:
        binary_search_recursive(arr, target, left, mid - 1)
```
time O(log n) Memory O(log n) < call stack in recursion
————————————————————

are_symbols_balanced(string): # A function that takes a string of symbol pairs as a parameter. The function should return True if the symbols are balanced, or False if they are not balanced
```
# Symbol string: {[()]} ([]{}()) ((()))) balanced
# Symbol string: (([{]) [](]){] unbalanced
def are_symbols_balanced(symbols):
    brackets = { dict keys=closing, values opening
        "}": "{",
        "]": "[",
        ")": "("
    }
    stack = []
    for symbol in symbols:
        if symbol in brackets: # is it a closing symbol?
            if not stack:
                return False
            top = stack.pop()  # get the OPENING symbol from stack
            if top != brackets[symbol]: # Opening sym, dict[closing] = Op.
                return False
        else:  # it is an opening symbol
            stack.append(symbol)
    return not stack
```
time: O(n), space O(n) due to stack
————————————————————

```
def add_all_previous_numbers(numbers): DTEX interview June 2024
    output = []
    output.append(numbers[0])
    for idx in range(1, len(numbers)):
        current_sum = numbers[idx]
        for idy in range(idx):
            current_sum = current_sum + numbers[idy]
        output.append(current_sum)
    return output
```
————————————————————

```
def dictionary_can_form_string_recursive(dictionary, string):
    string = string.replace(" ", "")  # Remove spaces if present
    def can_form(start):
        if start == len(string):  # Base case:
            return True
        for end in range(start + 1, len(string) + 1):
            if string[start:end] in dictionary and can_form(end):
                return True
        return False
    return can_form(0)
```
————————————————————

```
def dictionary_can_form_str_recursive_memoization(dictionary, string):
    dictionary_set = set(dictionary)  # list to a set for O(1) lookups
    memo = {}  # Use memoization
    def can_form(start):
        if start == len(string):
            return True
        if start in memo:
            return memo[start]
        for end in range(start + 1, len(string) + 1):
            if string[start:end] in dictionary_set and can_form(end):
                memo[start] = True
                return True
        memo[start] = False
        return False
    return can_form(0)
```