Common Programming Test Questions

1 Data Structures and Types

1.1 Stacks and Queues

In computer science, a stack is an abstract data type that serves as a collection of elements, with two main principal operations - (i) push, which adds an element to the collection, and (ii) pop, which removes the most recently added element that was not yet removed.

In computer science, a queue is a collection of entities that are maintained in a sequence and can be modified by the addition of entities at one end of the sequence and the removal of entities from the other end of the sequence.

1.2 Linked Lists and Arrays

In computer science, a linked list is a linear collection of data elements whose order is not given by their physical placement in memory. Instead, each element points to the next. It is a data structure consisting of a collection of nodes which together represent a sequence. In its most basic form, each node contains: data, and a reference (in other words, a link) to the next node in the sequence.

In computer science, an array data structure, or simply an array, is a data structure consisting of a collection of elements (values or variables), each identified by at least one array index or key. An array is stored such that the position of each element can be computed from its index tuple by a mathematical formula.

1.3 Binary Trees

In computer science, a binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. A recursive definition using just set theory notions is that a (non-empty) binary tree is a tuple (L, S, R), where L and R are binary

trees or the empty set and S is a singleton set containing the root. Note that binary trees are acyclic. Traversing a binary tree can be done via either Depth-First Search (DFS) or Breadth-First Search (BFS).

2 Searching and Sorting Algorithms

This section covers optimal codes on binary searches, sorting arrays and finding medians.

2.1 Binary Search

The Python implementation of Binary Search algorithm below has $O(\log n)$ time complexity.

```
def binary_search(array, target):
    lower = 0
    upper = len(array)

    while lower < upper:
        x = lower + int((upper - lower)/2)
        val = array[x]

        if target == val:
            return x
        elif target > val:
            if lower == x:
                break
            lower = x
        elif target < val:
            upper = x
    return None
```

2.2 Merge-Sort Algorithm

This algorithm implementation in Python covers the Merge-Sort algorithm in $O(n \log n)$ time complexity.

```
def merge_sort(list_in):
    def merge(list_1, list_2):
        list_out = []
```

```
        n1 = len(list_1)
        n2 = len(list_2)
        while n1 > 0 and n2 > 0:
            if list_1[0] <= list_2[0]:
                list_out.append(list_1[0])
                list_1 = list_1[1:]
                n1 = len(list_1)
            else:
                list_out.append(list_2[0])
                list_2 = list_2[1:]
                n2 = len(list_2)

        if n1 > 0:
            list_out.extend(list_1)
        if n2 > 0:
            list_out.extend(list_2)
        return list_out

    n = len(list_in)
    if n == 1:
        return list_in
    else:
        n_mid  = int(n/2)
        list_l = list_in[:n_mid]
        list_r = list_in[n_mid:]

        # Recursively call the merge_sort function until  #
        # only one element if left. Then merge the sorted #
        # lists together to get the final sorted list.    #
        list_l = merge_sort(list_l)
        list_r = merge_sort(list_r)
        return merge(list_l, list_r)
```

2.3 Median of Two Sorted Arrays

This algorithm implementation in Python covers finding the median of two sorted arrays of sizes m and n in $O(\log(n+m))$ time complexity. We should make use of the Binary search property for optimality.

```
def median_two_sorted_arrays(arr_A, arr_B):
```

```
"""
Find the index i and j such that for sorted arrays A
and B, we have:
A[0], A[1], ... , A[i-1] | A[i], A[i+1], ... , A[n-1]
B[0], B[1], ... , B[j-1] | B[j], B[j+1], ... , B[m-1]
"""

# Initialise i and j to be n/2 and m/2 respectively. #
n = len(arr_A)
m = len(arr_B)
if n == 0 or m == 0:
    raise ValueError

if n == 1 and m == 1:
    return (arr_A[0] + arr_B[0]) / 2
if n == 1 and m > 1:
    # Slot A into B using Binary Search. #
    val_A = arr_A[0]
    lower = 0
    upper = m
    new_array = [0] * (m+1)
    while (upper-lower) != 1:
        idx_p = lower + int((upper - lower)/2)
        if val_A > arr_B[idx_p]:
            lower = idx_p
        elif val_A < arr_B[idx_p]:
            upper = idx_p
        elif val_A == arr_B[idx_p]:
            break

    new_array[:idx_p] = arr_B[:idx_p]
    new_array[idx_p]  = val_A
    new_array[(idx_p+1):] = arr_B[idx_p:]

    mid_pt  = int((m+1)/2)
    mid_val = new_array[mid_pt]
    return mid_val
if m == 1 and n > 1:
    return median_two_sorted_arrays(arr_B, arr_A)
```

```
# Edge cases. #
if arr_A[0] >= arr_B[m-1] or arr_B[0] >= arr_A[n-1]:
    if arr_A[0] >= arr_B[m-1]:
        new_array = arr_B + arr_A
    if arr_B[0] >= arr_A[n-1]:
        new_array = arr_A + arr_B

    if (m+n) % 2 == 0:
        mid_pt = int((m+n)/2)
        return (new_array[mid_pt-1] + new_array[mid_pt]) / 2
    else:
        mid_pt = int((m+n)/2)
        return new_array[mid_pt]

if (m % 2 == 0) and (n % 2 == 0):
    idx_A = int(n/2)
    idx_B = int(m/2)
else:
    idx_A = int(n/2)
    idx_B = int(m/2) + 1

# Check for edge case. #
if idx_A == (n-1) and \
    arr_B[idx_B-1] >= arr_A[idx_A]:
    idx_A = int(n/2)
    idx_B = int(m/2)

if idx_B > (m-1):
    idx_B = int(m/2)

# Condition A[i-1] > B[j] and B[j-1] > A[i]. #
while True:
    A_left = arr_A[idx_A-1]
    B_left = arr_B[idx_B-1]
    A_right = arr_A[idx_A]
    B_right = arr_B[idx_B]

    if A_left >= B_right:
        idx_A -= 1
        idx_B += 1
```

```
        if B_left >= A_right:
            idx_A += 1
            idx_B -= 1

        if A_left < B_right and B_left < A_right:
            break

    max_left  = max(A_left, B_left)
    min_right = min(A_right, B_right)
    if (m+n) % 2 == 0:
        return (max_left + min_right) / 2
    else:
        n_median = int((m+n)/2) + 1
        nA_right = n - idx_A
        nB_right = m - idx_B
        if (nA_right + nB_right) >= n_median:
            return min_right
        else:
            return max_left
    return None
```

3 Miscellaneous

3.1 Reversing an Array

To reverse an array, use `tmp_list[::-1]` where `tmp_list` is a Python list.

3.2 Palindromes

A palindrome is a word, number, phrase, or other sequence of characters which reads the same backward as forward, such as madam, racecar.

```
def check_palindrome(tmp_input):
    if type(tmp_input) != list:
        tmp_input = list(tmp_input)
    return tmp_input == tmp_input[::-1]
```

3.3 Longest Common Substring

```
def longest_common_substring(
    string_1, string_2, check_palindrome=True):
```

```
    tmp_lcs = ""

    n1 = len(string_1)
    n2 = len(string_2)
    for m in range(n1):
        for n in range(n2):
            tmp_index = 0
            tmp_match = ""

            idx1 = m + tmp_index
            idx2 = n + tmp_index
            while idx1 < n1 and idx2 < n2 and \
                string_1[idx1] == string_2[idx2]:
                tmp_match += string_1[idx1]
                tmp_index += 1

                idx1 = m + tmp_index
                idx2 = n + tmp_index
            if len(tmp_match) > len(tmp_lcs):
                if check_palindrome:
                    if tmp_match[::-1] == tmp_match:
                        tmp_lcs = tmp_match
                else:
                    tmp_lcs = tmp_match

    if len(tmp_lcs) == 1:
        return None
    else:
        return tmp_lcs
```

3.4 Longest Subsequence

In mathematics, a subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

```
def longest_subsequence(string_in):
    tmp_longest = []
    subsequence = []
    tmp_sub_set = set([])
```

```
for n_char in range(len(string_in)):
    if string_in[n_char] not in tmp_sub_set:
        subsequence.append(string_in[n_char])

        if len(subsequence) > len(tmp_longest):
            tmp_longest = subsequence
            tmp_sub_set = set(tmp_longest)
    else:

        subsequence = [string_in[n_char]]
return "".join(tmp_longest)
```