Assignment Solutions

1 Time Complexity Calculations

(a) Code Snippet 1

Outer loop: i = n / 2; i > 1; i /= 6 runs O(log n).

Middle loop: j = 2; j <= n; j *= 4 runs O(log n).

Inner loop: k = 0; k <= j; k *= 3 runs O(log j).

Combined complexity: O(log n) × O(log n) × O(log n) = O((log n)^3)

(b) Code Snippet 2

First set of loops: i = n; i > 1; i-- and j = 2; j <= n; j++ runs O(n × n) = O(n^2)

Second set of loops: Similar to (a), runs O((log n)^3)

Final complexity: O(n^2 + (log n)^3) = O(n^2)

2. Modifications to Binary Search

(a) Return First Index

```
def binary_search_first(arr, key):
    low, high, result = 0, len(arr) - 1, -1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == key:
            result = mid
            high = mid - 1
        elif arr[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return result
```

(b) Return First Index and Count

```
def binary_search_count(arr, key):
    def find_first(arr, key):
        low, high, result = 0, len(arr) - 1, -1
        while low <= high:
            mid = (low + high) // 2
            if arr[mid] == key:
                result = mid
                high = mid - 1
            elif arr[mid] < key:
                low = mid + 1
            else:
                high = mid - 1
        return result
```

```
def find_last(arr, key):

    low, high, result = 0, len(arr) - 1, -1

    while low <= high:

        mid = (low + high) // 2

        if arr[mid] == key:

            result = mid

            low = mid + 1

        elif arr[mid] < key:

            low = mid + 1

        else:

            high = mid - 1

        return result


first = find_first(arr, key)

last = find_last(arr, key)

if first == -1:

    return (-1, 0)

return (first, last - first + 1)
```

3. Binary Search Execution

Given,

Array with

Index: 0 1 2 3 4 5 6 7

Values: 23, 2, 19, 3, 7, 11, 5, 13

Target Value: T = 2

Step 1:

L = 0, R = 7

m = (L + R) // 2 = (0 + 7) // 2 = 3

A[m] = 7

Since A[m] > T, move to the left half: R = m - 1 = 2

Step 2:

L = 0, R = 2

m = (L + R) // 2 = (0 + 2) // 2 = 1

A[m] = 3

Since A[m] > T, move to the left half: R = m - 1 = 0

Step 3:

L = 0, R = 0

m = (L + R) // 2 = (0 + 0) // 2 = 0

A[m] = 2

Since A[m] = T, target is found at index 0

Therefore,

Sorted Array: [2, 3, 5, 7, 11, 13, 19, 23]

Index of T = 2: 0

The binary search will successfully find T = 2 in the sorted array.

4. Find Maximum in Wave Sequence

(i)
```
def find_wave_max(arr):
    low, high = 0, len(arr) - 1
    while low < high:
        mid = (low + high) // 2
        if arr[mid] > arr[mid + 1]:
            high = mid
        else:
            low = mid + 1
    return arr[low]
```

(ii) Time Complexity: O(log n).

At each step, calculate mid = (low + high) // 2, then it compares arr[mid] with its neighbors: If arr[mid] > arr[mid + 1], move left: high = mid. Otherwise, move right: low = mid + 1.

Each iteration eliminates half the range, reducing the search space exponentially.

The algorithm runs for at most $\log_2(n)$ iterations, where n is the array size.

Each iteration performs constant-time comparisons O(1).

Therefore, time Complexity: $O(\log_2 n)$.

5. Integer Square Root

(a) Linear Search

```
def linear_sqrt(key):
    result = -1 # Step 1: Initialize the result to -1
    for i in range(1, key + 1): # Step 2: Iterate through numbers from 1 to key
        if i * i <= key: # Step 3: Check if the square of i is less than or equal to key
            result = i # Step 4: Update result if condition is satisfied
    return result # Step 5: Return the largest valid i
```

(b) Binary Search

```
def binary_sqrt(key):
    low, high = 0, key # Step 1: Initialize the search range
    result = -1 # Step 2: Initialize the result to -1
    while low <= high: # Step 3: Repeat until the range is empty
        mid = (low + high) // 2 # Step 4: Find the midpoint of the range
        if mid * mid <= key: # Step 5: Check if mid^2 is less than or equal to key
            result = mid # Step 6: Update result to the current mid
            low = mid + 1 # Step 7: Narrow the search range to the right
        else:
            high = mid - 1 # Step 8: Narrow the search range to the left
    return result # Step 9: Return the largest valid mid
```

6. Sorting and Searching

(a) Sorting allows repeated searches in O(log N), making it better for multiple queries when M > N.

Sorting is preferred if multiple searches are required because:

Linear search: O(m×n) for m searches.

Sorting + Binary search: O(nlogn+mlogn). Sorting becomes efficient when m is large

(b) I can modify count sort by shifting all elements by adding the absolute value of the smallest element.

we can follow these steps to modify count sort for negative Integers

1.Find the smallest value (min_val).

2.Shift all elements by | min_val | to make them non-negative.

3.Apply Count Sort.

4.Shift the sorted elements back.

(c) Scale non-integer values to integers or handle them separately.

we can follow these steps to modify count ~~short~~ sort for the given list

1.Multiply all elements by 10^d(like, for decimals, multiply by 10).

2.Apply Count Sort.

3.Divide the sorted elements by 10^d to restore decimals.

(d) We can choose merge sort for better worst-case performance when memory is constrained.
We use Quick Sort if memory is limited, as it requires less space (O(logn)) compared to Merge Sort (O(n)).

(e) A sorted array, for example [1, 2, 3, 4, 5], causes worst-case $O(N^2)$ with a fixed pivot.

7. Sorting Jack's List:

```
def sort_jacks_list(arr):
    odd = sorted(arr[1::2]) # Step 1: Sort odd-indexed elements in increasing order
    even = sorted(arr[::2], reverse=True) # Step 2: Sort even-indexed elements in decreasing order
    result = [] # Step 3: Initialize an empty list to store the final sorted array
    for i in range(len(arr)): # Step 4: Iterate through the original array
        if i % 2 == 0: # Step 5: If index is even, add from the 'even' list
            result.append(even.pop(0)) # Pop the first element from 'even'
        else: # Step 6: If index is odd, add from the 'odd' list
            result.append(odd.pop(0)) # Pop the first element from 'odd'
    return result # Step 7: Return the sorted array
```

Now, for time complexity,
Sorting even and odd indexed elements separately takes linear time

Merging the two sorted parts into the original array also takes linear time.
Thus, the total time complexity is O(n)