Tony Siu 916209430

1

Starting from i = floor(n/2)

i = 4

min_heapify(A,4) = [2,5,7,3,9,8,1,6,4]

recursive call

min_heapify(A,9) = [2,5,7,3,9,8,1,6,4]

i = 3

min_heapify(A,3) = [2,5,1,3,9,8,7,6,4]

recursive call

min_heapify(A,7) = [2,5,1,3,9,8,7,6,4]

i = 2

min_heapify(A,2) = [2,3,1,5,9,8,7,6,4]

recursive call

min_heapify(A,4) = [2,3,1,4,9,8,7,6,5]

recursive call

min_heapify(A,9) = [2,3,1,4,9,8,7,6,5]

i = 1

min_heapify(A,1) = [1,3,2,4,9,8,7,6,5]

2.

Heap Sort -> [h,e,g,d,b,a,f,c]

Quick Sort -> [d,c,a,e,b,f,g,h]

Bubble Sort -> [b,c,a,d,e,g,f,h]

Insertion Sort -> [a,d,h,e,b,g,f,c]

Merge Sort -> [a,d,e,h,b,c,f,g]

Selection Sort -> [a,d,b,e,f,c,g,h]

3

a)

step 1:
j = j + 1 -> j = 2
[5,8,3,2,8,5,3,6,8]
step 2:
i = 1,k = 9
Exchange A[2] and A[9],swap values 8 and 8
k = k - 1 -> k = 8
j = 2
[5,8,3,2,8,5,3,6,8]
step 3:
i = 1,j = 2
Exchange A[2] with A[8],swap values 8 and 6
k = k - 1 -> k = 7
[5,6,3,2,8,5,3,8,8]
step 4:
i = 1,j = 2
exchange A[2] with A[7], swap 6 and 3
k= k - 1; k = 6
[5,3,3,2,8,5,6,8,8]
step 5:
i = 1,j = 2,k = 6
exchange A[1] with A[2],swap values 5 and 3
i = i + 1; i = 2
j = j + 1; j = 3
[3,5,3,2,8,5,6,8,8]
step 6:
i = 2,j = 3,k = 6
Exchange A[2] with A[3], swap 5 and 3
i = 3
j = 4
[3,3,5,2,8,5,6,8,8]
step 7:
i = 3,j = 4,k= 6
exchange A[3] and A[4],
swap values 5 and 2
i = 4,j = 5
[3,3,2,5,8,5,6,8,8]
step 8:
i = 4,j = 5,k = 6
exchange A[5] and A[6],
swap values 8 and 5
[3,3,2,5,5,8,6,8,8]
step 9:
i = 4,j = 5,k =5
j = j + 1 -> j = 6
exit
[3,3,2,5,5,8,6,8,8]

b) The algorithm is related to quick sort , more specifically using the 3 way partitioning scheme

c) For 3 way partitioning;
T(n) = T(L) + T(E) + T(G) + Cn where L, E and G are the sizes of the less than, equal to, and greater than partitions and C is the constant. This simplifies to T(n) = c n log(n) in the best and average cases. This approach improves quicksort in prescence of duplicate keys for more balanced partitions

4.

a)

```
class Node:
    def __init__(self, data:int):
        self.data = data
        self.next = None

def reverse_linked_list(root:Node):
    if root is None:
        return None

    if root.next is None:
        return root

    new_root = reverse_linked_list(root.next)
    root.next.next = root
    root.next = None
    return new_root
```

I employed reversing a linked list recursively

1. If the root is None, return None.
2. If the root.next is None, return the root.
3. Recursively reverse the linked list starting from root.next.
by pointing the next root to itself
4. Set root.next.next to root.
5. Set root.next to None.
6. Return the new root.

b) The time taken for a recursive call is T(n - 1). The time for the 2 assignments is a constant k
This builds a recurrence relation T(n) = T(n - 1) + k. Unrolling the recurrence gives us
T(n) = k(n - 1) + c1 and simplifying it further gives us T(n) = kn + c.

5.

a)

Since the array is sorted, we can utilize binary search to efficiently find the occurrences of the target. The idea is to find the indices of the first and last occurrences of the target in the array. The number of occurrences is then the difference between these indices plus one. The below is my algorithm implementation.

```
def find_first_occurrence(array, length, target):
    low = 0
    high = length - 1
    first_occurrence = -1
    while low <= high:
        mid = (low + high) // 2
        if array[mid] < target:
            low = mid + 1
        elif array[mid] > target:
            high = mid - 1
        else:
            first_occurrence = mid
            high = mid - 1  # Continue searching to the left
    return first_occurrence

def find_last_occurrence(array, length, target):
    low = 0
    high = length - 1
    last_occurrence = -1
    while low <= high:
        mid = (low + high) // 2
        if array[mid] < target:
            low = mid + 1
        elif array[mid] > target:
            high = mid - 1
        else:
            last_occurrence = mid
            low = mid + 1  # Continue searching to the right
    return last_occurrence

def count(array, length, target):
    first = find_first_occurrence(array, length, target)
    if first == -1:
        return 0  # Target not found
    last = find_last_occurrence(array, length, target)
    return last - first + 1
```

5 b)
Best case time complexity is T(n) = log n. The best case complexity is determined y the time take perform binary search as it needs to find the true first or last occurrences and verify that.

Worst case time complexity is also T(n) = log n. The worst case occurs when the target is not present in the array. However, in this case binary search still examines the array fully before concluding that the target is absent.

Average case time complexity is also T(n) = log n. On average, the target may be located somwhere in the middle of the array and would conduct binary search to count the duplicates in the sorted array.