Design and Analysis of Algorithms

Algorithm Design

Algorithm 1 groupAndRemoveNull (A[0..n-1])

```
//Converts the borrowedTools toolCollection into an array then loops
//through the array adding all non-null items into a temporary list and
//converting to list back into an array
//Input:   An array A[0..n-1] of tools
//Output:   A modified array A[] of tools with no null elements
temp[] ← null
for i←0 to n-1 do
    if A[i] != null then
        add A[i] to temp
    end if
end for
A ← temp
```

Algorithm 2 heapSort(A[0..n-1])

```
//Sorts array A into non decreasing order
and return B after three loops (top three tools)
//Input:   An array A[0..n-1] of tools
//Output:   An array B of the top three tools

B ← B[3] //create an array of 3 elements
Use heapBottomUp to create a heap of A
j ← 0
for i ← 0 to n-1 do
    Use maxKeyDelete to store A[0] in B[j] and move A[0] to the end
    j ← j + 1
    if j == 3 then //if there are three elements in j
        i ← n-1
    end if
end for
return B
```

Algorithm 3 heapBottomUp(A[0..n-1])

```
//Constructs a heap from elements of a given array
//by the bottom-up algorithm
//Input:  An array A[1..n] of tools
//Output:  A heap A[1..n]
for i← ⌊n/2⌋ downto 1 do
    k ← i
    v ← A[k]
    heap ← false
    while not heap and 2 * k ≤ n do
        j ← 2 * k
        if j < n then //there are two children
            if A[j] < A[j+1] then//compare array element borrowings
                j← j+1
            end if
        end if
        if v ≥ A[j] then //compare array element borrowings
            heap ← true
        else if A[k] ≥ A[j] then
            k ← j
        end if
    end while
    A[k]← v
end for
```

Algorithm 4 maxKeyDelete (A[0..n-1]), size, counter, B[0..2]

```
//Puts the first element of A in B[counter]
//swaps A[0] with A[n-1], decreases size and re heaps
//Input:  An array A[0..n-1] of tools, size of A (minus sorted elements)
//counter for tracking placement of elements, B[] store top 3 elements
//Output:  Stop after B[0..2] has been filled
B[counter] ← A[0]
temp ← A[size - 1]
n ← size - 1
//Heapify again
k ← 0
v ← A[0]
heap ← false
while not heap and (2 * k + 1) ≤ (n - 1) do
    j ← 2 * k + 1
    if j < (n-1) then //there are two children
        if A[j] < A[j+1] then//compare array element borrowings
            j← j+1
        end if
    end if
    if v ≥ A[j] then //compare array element borrowings
        heap ← true
    else
        A[k] ← A[j]
        k ← j
    end if
end while
A[k]← v
```

Algorithm Analysis

The 'Display Top Three' method is comprised of 4 different algorithms.

The first algorithm, **groupAndRemoveNull**, is a simple single for loop.
In worst case the algorithm has to loop through and assign all elements to the list:

$$C_{(worst)}(n) = \sum_{i=0}^{n-1} 2 = 2\sum_{i=0}^{n-1} 1 = 2(n-1) = 2n-2$$

Big O classes ignore multiplicative constants as they dont effect the time effiency of the algorithm in the big picture. Thus,

- Time efficiency - O(n), linear
- Space efficiency - temporary storage is needed

The second algorithm, **heapSort**, calls the third algorithm (heapBottomUp) and the fourth (maxKeyDelete). HeapSort performs the main purpose of the 'Display Top Three' method. The **heapBottomUp** is called once and has a time efficiency of O(n). The **maxKeyDelete** algorithm has a O(log n) efficiency. The **heapSort** calls **maxKeyDelete** n times (three in the case of this algorithm). Thus:

$$C_{(worst)}(nlogn) = \sum_{i=0}^{k} n = n * k = n * logn$$

- Time efficiency - O(nlogn)
- Space efficiency - temporary storage is not needed
- Not stable - if someone borrows a tool so that it match's the number of borrows of the previous Top Borrowed, the new Tool will swap with the original at the top of the list.

Both **groupAndRemoveNull** and **heapSort** are separate algorithms within the 'Display Top Three' method. As such the overall efficiency of this algorithm is O(nlogn) as it is set by the largest of the polynomial terms.