Algorithms

Assignment 3

Ashwin Sathiyanarayanan
002227609

Question 1:

```
MAX-HEAP-DELETE(A, i)
    if A.heap_size < 1
        error "heap underflow"

    if A.heap_size < i
        error "index i is larger than the heap size"

    A[i] = A[A.heap_size]
    A.heap_size = A.heap_size - 1
    MAX-HEAPIFY(A)
```

Explanation:

1. Initially, we check whether the heap is empty or not and if it is empty we return a heap underflow error.
2. Then we check if the index of the element to be deleted is greater than the heap size itself and if it is true, we return an index larger than heap size error.
3. Then we replace the element at the given index value with the last element from the heap. Then we decrement the heap size by 1.
4. Finally, since we have tampered with max heap property of the heap, we call the MAX-HEAPIFY function to maintain this property of the heap.

Question 2:

$T(n) = T(n-1) + \Theta(n)$

We have to prove $T(n) = \Theta(n^2)$.

(i) Proof for $T(n) = O(n^2)$

Guess: $T(n) = O(n^2)$

$$\begin{aligned} T(n) &= T(n-1) + \Theta(n) \\ &= c_1(n-1)^2 + c_2 n \\ &= c_1(n^2 + 1 - 2n) + c_2 n \\ &= c_1 n^2 + c_1 - 2c_1 n + c_2 n \\ &= c_1 n^2 - 2c_1 n + c_2 n + c_1 \end{aligned}$$

In order for T(n) to be $\leq c_1 n^2$,

$-2c_1 n + c_2 n + c_1 \leq 0$ *this should be true.*

or

$(-2c_1 + c_2)n + c_1 \leq 0$

Therefore,

for $n \geq 1$ *values,*

$(-2c_1 + c_2) \leq 0$, *this should be true.*

$-2c_1 + c_2 \leq 0$

$c_2 \leq 2c_1$

$c_1 \geq c_2/2$

Therefore, when $c_1 \geq c_2/2$,

$T(n) \leq cn^2$

$T(n) = O(n^2)$

(i) Proof for $T(n) = \Omega(n^2)$

$$T(n) = T(n-1) + \theta(n)$$
$$= c_1(n-1)^2 + c_2 n$$
$$= c_1(n^2 + 1 - 2n) + c_2 n$$
$$= c_1 n^2 + c_1 - 2c_1 n + c_2 n$$
$$= c_1 n^2 - 2c_1 n + c_2 n + c_1$$

In order for T(n) to be $\geq c_1 n^2$,

$-2c_1 n + c_2 n + c_1 \geq 0$ *this should be true.*

or

$(-2c_1 + c_2)n + c_1 \geq 0$

Therefore,

for valid values of n $(i.e)$ $n \geq 1$,

$(-2c_1 + c_2) \geq 0$, *this should be true.*

$-2c_1 + c_2 \geq 0$

$c_2 \geq 2c_1$

$c_1 \leq c_2/2$

Therefore, when $c_1 \leq c_2/2$,

$T(n) \geq cn^2$

$T(n) = \Omega(n^2)$

Since we have proved $T(n) = O(n^2)$ *and* $T(n) = \Omega(n^2)$,

from this we have proved $T(n) = \Theta(n^2)$.

Question 3:

```
PARTITION`(A, p, r)
        x = A[r]
        i = p - 1
        t = p - 1

        for j = p to r - 1
                if A[j] < x:
                        i = i + 1
                        t = t + 1
                        exchange A[i] with A[j]
                        if i != t:
                                Exchange A[t] with A[j]
                else if A[j] == x:
                        t = t + 1
                        exchange A[t] with A[j]

        exchange A[i + 1] with A[r]
        return i + 1, t
```

Explanation:

1. First, we select the last element of the subarray as the pivot element for this function.
2. Then we initialise the i and t values to p - 1. Here, at the end of this function, i and t represent the lower and upper bound of the subarray whose values of the elements are equal.
3. Then we loop through the subarray. During each iteration, we first check whether A[j] is less than the pivot element. If it is true, we increment the values of both the lower bound and upper bound by 1, and then we swap the elements in A[i](element in lower bound) and A[j](current element). Then we check if the lower bound and upper bound are not equal and if they are not equal, we swap the elements in A[t](element in upper bound) and A[j](current element).

4. If A[j] is not less than the pivot element, we check if A[j](current element) is equal to the pivot element. If it is true, we increment the value of the upper bound of the duplicate elements, and then we swap the elements in A[t](element in upper bound) and A[j](current element).
5. Finally, at the end of this for loop, we swap the pivot element with the next element to the lower bound of the range of the duplicate elements.
6. Therefore, finally, i + 1 represents the q value and t represents the t value, where elements of A[q: t] are equal, elements of A[p: q-1] are less than A[q], and elements of A[t+1: r] are greater than A[q].

Question 4:

Using Theorem 8.1 in the textbook,

If l is the number of reachable nodes and h is the height of the tree,
$n! \leq l \leq 2^h$

1. *if we consider at least half of the* $n!$ *inputs of length* n,
   $n!/2 \leq n! \leq l \leq 2^h$
   Therefore,
   $n!/2 \leq 2^h$
   Taking Log on both sides
   $log(n!/2) \leq log(2^h)$
   $log(n!) - log(2) \leq h.log(2)$
   $log(n!) - 1 \leq h$

   Using Stirling's approximation, we know that
   $log(n!) = \Theta(nlogn)$

   Therefore,
   $\Theta(nlogn) - 1 = h$
   we can neglect 1 *as it is a lower order term.*
   $h = \Theta(nlogn)$

   from the equation, we can infer that there is no comparison sort for at least half of the $n!$ *inputs of length* n *whose running time is linear.*

2. *if we consider a fraction of* $1/n$ *of the inputs of length* n,
   $n!/n \leq n! \leq l \leq 2^h$
   Therefore,
   $n!/n \leq 2^h$
   Taking log on both sides,
   $log(n!/n) \leq log(2^h)$

$log(n!) - log(n) \leq h.log(2)$

$Using\ Stirling's\ approximation,\ we\ know\ that$
$log(n!) = \Theta(nlogn)$

$h = \Theta(nlogn) - logn$
$we\ can\ neglect\ logn\ as\ it\ is\ a\ lower\ order\ term.$

Therefore,
$h = \Theta(nlogn)$

$from\ the\ equation,\ we\ can\ infer\ that\ there\ is\ no\ comparison\ sort\ for\ a\ fraction\ of\ 1/n\ of\ the\ inputs\ of\ length\ n\ whose\ running\ time\ is\ linear.$

3. $if\ we\ consider\ a\ fraction\ 1/2n\ of\ the\ input\ of\ length\ n,$
   $n!/2^n \leq n! \leq l \leq 2^h$
   $Therefore,$
   $n!/2^n \leq 2^h$
   $Taking\ log\ on\ both\ sides,$
   $log(n!/2^n) \leq log(2^h)$
   $log(n!) - log(2^n) \leq h.log(2)$
   $log(n!) - n.log(2) \leq h.log(2)$
   $log(n!) - n \leq h$
   $Using\ Stirling's\ approximation,\ we\ know\ that$
   $log(n!) = \Theta(nlogn)$
   $h = \Theta(nlogn) - n$
   $we\ can\ neglect\ n\ as\ it\ is\ a\ lower\ order\ term.$
   Therefore,
   $h = \Theta(nlogn)$

$from\ the\ equation,\ we\ can\ infer\ that\ there\ is\ no\ comparison\ sort\ for\ a\ fraction\ 1/2n\ of\ the\ input\ of\ length\ n\ whose\ running\ time\ is\ linear.$

Question 5:

It is easily possible to craft an algorithm that answers any query about how many of the n integers fall into a range [a:b] in O(1) time. We follow the steps as follows:

1. Initially, we must create an auxiliary array of the size k where k is the maximum possible value in the input array. This auxiliary array will be used as a counting array which is used to keep track of the number of recurrences of the elements in the input array. The indices of the counting array will be used to address the element in the input array and the value in each index of the counting array will be the number of recurrences of that index(value) in the input array.
2. Now, we have to iterate through our counting array and initialize the value of all indices to zero initially.
3. Now, we have to write a for loop to iterate through the input array. During each iteration, we have to update the count of each occurrence of the element we encounter in the input array, into the value of that index in our counting array. This is done like: $countingarray[inputarray[i]] \mathrel{+}= 1;$
4. At the end of this loop, our counting array will be populated with the number of occurrences of each element in the input array using the index-value mapping property.
5. Now, we apply the concept of cumulative addition (or) running total to our counting array. So now we iterate through our counting array starting from the second index and during each iteration, we have to update the current index's value as the summation of the current index's value and the previous index's value, like: $countingarray[i] = countingarray[i] + countingarray[i-1]$
6. At the end of this loop, our counting array will be a cumulative summation array of the count of the number of recurrences of the elements in our input array.
7. Now this is pretty much the array we need. Now if we need to find how many of the n integers fall into the range [a….b], provided $0 \leq a, b \leq k$, all we need to do is to find the value of the counting array at index b subtracted by the value of the counting array at index

a-1, (i.e) $number\ of\ integers\ that\ fall\ in\ the\ range\ [a.... b] = countingarray[b] - countingarray[a - 1]$

8. The whole algorithm takes time $\Theta(n + k)$. The first loop to initialize the counting array with 0s will take $\Theta(k)$ time as the size of the counting array is k. The second loop which iterates through the input array to get the values of the number of occurrences of each element runs in $\Theta(n)$ time as the size of the input array is n. Then the third loop which is used to calculate the running total of the counting array runs in $\Theta(k)$ time as the size of the counting array is k. Therefore the overall running time of the algorithm is $\Theta(n + k)$.
9. Now, when it comes to actually finding the number of elements in the required range of a to b, this runs in $\Theta(1)$ because we just get the value at index b of the counting array and subtract it from the value at index a-1 of the counting array, each of which takes only $\Theta(1)$ as the time taken to access an array element is $\Theta(1)$.

Question 6:

Stable Algorithms:

- Insertion Sort
- Counting Sort

Unstable Algorithms:

- Merge Sort
- Quick Sort

Converting any comparison sort to be stable:

1. We can convert any comparison sort to be stable by considering not only the elements but also their index value at the input array for comparison. This ensures that not only the value of the input array is used for comparison but also their indices.
2. This can be implemented as follows:
    a. We can use a tuple inside of an array to map the value of the element along with its index position.
    b. While comparing, we first compare if the value1 <= value2. If this is false, then move on to the next
    c. comparison. If it is true, then check if the index of value1 < index of value 2. If it is true, place value1 before value 2, and if it is false, place value 2 before value1.
    d. The time complexity of this algorithm remains the same as that of the sorting algorithm we use because the only extra thing that we do is to compare the indices of the values using an if statement which takes only $\Theta(1)$ time.
    e. When it comes to space complexity, it doubles by the initial value as here we are storing the value along with its index values instead of just the element. Therefore, if initially it would have taken $O(n)$, now it would take $O(2n)$.

Question 7:

We can sort n integers in the range of 0 to $n^3 - 1$ in O(n) time using radix sort. We can represent each number in the array in the form of base-n. Then we can use radix sort along with counting sort to sort the numbers.

First, we have to find the number of digits d required to represent the n integers.

In an n-ary number system, each digit can be of any value in the range from 0 to $n^3 - 1$. So if we have d digits, the total number of digits d will be $log_k n^3$.

$$d = log_n n^3$$
$$d = 3.log_n n$$

$$log_n n = 1$$

$$d = 3$$

Therefore, the number of digits required is 3.

Now, we know that radix sort sorts in

$$\Theta(d(n + k))$$
$$substituting\ the\ values\ of\ d\ and\ k$$
$$\Theta(3(n + n))$$
$$\Theta(3(2n))$$
$$\Theta(6n)$$
$$\Theta(n)$$

$Therefore,\ using\ radix\ sort\ and\ to\ sort\ base_n\ numbers,\ we\ can\ sort\ n\ integers\ in\ the\ range\ of\ n^3 - 1\ in\ \Theta(n)$