Stephen Chambers
smx227
CS858

Assignment 2 Writeup

1.

My program is working correctly. The plots are attached.

2.

It seems there is a tradeoff between time complexity and accuracy. min2_scan gets very similar results to the heap in terms of performance, but pales in comparison in terms of time it takes to run. The other two algorithms do not grade well on performance, but have a better time complexity than the heap. However, the difference in performance between seq and seq_sort and the heap is far greater than the difference in tiem run. Therefore, using a heap is the best choice in almost all cases. I can conclude that efficient use of data structures can drastically speed up algorithms.

3 (a)

Match is called within contains 1001^2 times.
Match is called within index-helper 1001*1000 times.
Match is called within remove 1001 * 1000 times.

Therefore, match is called 1001^2 + 1001*1000 + 1001*1000 times.

3 (b)

The time complexity is $O(n^2)$.

3 (c)

```
prev = NULL
cur = list
while(cur != null)
   if match(cur.data, x)
      if prev != NULL
         prev.next = cur.next
   prev = cur
   cur = cur.next
```

The time complexity of this algorithm is O(n), as it scans the list linearly and modifies next pointers to remove each elements it finds.

4 (3-2).

A	B	O	o	Ω	w	θ
$lg^k n$	n^ϵ	yes	yes	no	no	no
n^k	c^n	yes	yes	no	no	no
$\sqrt{n}$	$n^{sin(n)}$	no	no	no	no	no
2^n	$2^{n/2}$	no	no	yes	yes	no
$n^{log(c)}$	$c^{log(n)}$	yes	no	yes	no	yes
$log(n!)$	$log(n^n)$	yes	no	yes	no	yes

5 (6-1-3).

In a max heap, the following property holds, called the **max-heap-property**. This property states that:

A[parent(i)] >= A[i]

According to this property, the value of nodes starting from the leaf **must** either stay the same or increase as they get to the root. Therefore, the root will have the maximum value for that subtree.

6 (6-1-4).

In a max heap, the smallest element must reside in a leaf.

7 (6-5-5).

Invariant: The subarray A[1..A.heap-size] satisfies the max-heap property, except that there may be one violation: A[i] may be larger than A[PARENT(i)].

Proof:

Initialization: We are given that the max-heap property holds at the time HEAP-INCREASE-KEY is called. Additionally, the only change in the heap before the loop is in line 3, where A[i] is set to the new key. Therefore, the only possible violation in this heap is that A[i] may be larger than A[PARENT(i)].

Maintenance: During the loop, the only swapping in the heap that occurs is with A[i] and its parent. We also know that the rest of the heap satisfies the max-heap property, so the only possible violation after this swap is that A[PARENT(i)] may be larger than A[PARENT(PARENT(i))]. Therefore, i is then set to be PARENT(i), so by

the next iteration of this loop, the only possible violation is that A[i] may be larger than A[PARENT(i)].

Termination: There are only two possible ways for this loop to terminate. The first is if *i* is less than 1, meaning it is the root. The root has no parents, and is the maximum value in the heap. We also know that the rest of the heap satisfies the max-heap-property. Therefore, the invariant holds during this termination condition.

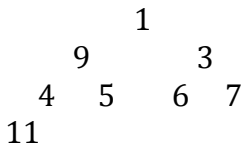
The second termination condition is if A[PARENT(i)] > A[i]. If this is true, that means that the max-heap-property is satisfied by A[i] and A[PARENT(i)]. We also know that the max-heap-property is satisfied in every other element in the heap. Therefore, the invariant holds after all possible termination conditions.

8 (6-5-9).

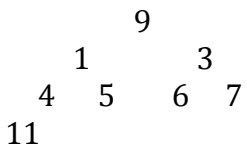
```
lists[]                                 assume k sorted lists
heap[]                                  assume heap
output[]                                assume output array

/* Initialize the heap to the first element of each k list */
for i = 1 to K
  heap.insert(lists[i].get(0))              O(log(k))

index = 0
count = 0
while empty != K
        for i = 1 to K
           index++
           min = heap.pop()                     O(log(k))
           output[count] = min                  O(n)
           count++
           if index < lists.length
              heap.insert(lists[i].get(index))      O(log(k))
           else
              empty++
```

9 (6-2-6).

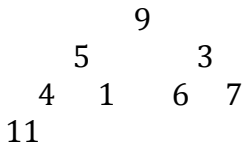
Consider the example heap below, with MAX-HEAPIFY called on the root.



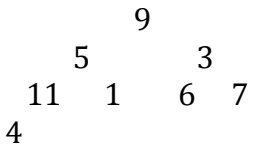
After the first step, the heap would like the following:



Now 1 is not the largest between it and its kids, so the MAX-HEAPIFY is called recursively to get the following:



Now 4 is not the largest between it and its kids, so the MAX-HEAPIFY is called recursively to get the following:



We have proved that in the worst case, MAX-HEAPFIY needed to be called for every node in the path from the root to the leaf of that subtree. It needed to be called log(n) times.

Therefore, the worst case time complexity is $\Omega \log(n)$, because for any $c_1 > 0$, the following holds:

$$\log(n) \leq \log(n) * c_1$$

10 (6-4-2).

Proof:

Initialization: At the start of the for loop, i is initialized to A.length. Therefore, the subarray A[1..i] is a max-heap contains the I smallest element of A[1..n]. Additionally, A[i+1..n] contains the n-1 largest elements of A[1..n], because it contains *all* of the elements from A[1..n]. The invariant holds at loop initialization.

Maintenance: During the loop, the only swapping operation exchanges A[1] with A[i]. Additionally, the heap-size is decreased by 1. Finally, the value at the root of the heap is pushed down through the MAX-HEAPIFY method. Therefore, the subarray A[1..i] is a max-heap contains the i smallest element of A[1..n], because the largest element was swapped with A[i], and the element that was at A[i] was pushed down. Additionally, A[i+1..n] contains the n-1 largest elements of A[1..n], because the largest element from the previous iteration was swapped down to A[i], and n was decreased by 1. The invariant holds during the loop.

Termination: The loop terminates when i = 2. Because the invariant was guaranteed to hold during initialization and maintence, when i=2, the first element will be the smallest element in the subarray [1..n], and all other elements will be larger. Or, more formally, A[i+1..n] contains the n-1 largest elements of A[1..n].

11 (6-1 (b)).

In the worst case, HEAP-INCREASE-KEY will have a complexity of log(n), because an element at the leaf may have to be increased until it becomes the root. HEAP-INCREASE-KEY is called n times by BUILD-MAX-HEAP', through MAX-HEAP-INSERT. Therefore, the worst case time complexity of this algorithm is nlog(n).
Now we need to prove that nlog(n) = $\theta nlog(n)$

Proof:

c_1 * nlog(n) <= nlog(n) <= c_2 * nlog(n)

For every c1 >0 and c2 > 0, the equation above will hold. Therefore, the worst case time complexity of BUILD-MAX-HEAP' is $\theta nlog(n)$.

12.
I thought this assignment was difficult for the amount of time we had to do it. I would have easily preferred the first assignment to be due in five days, and have this one take a full week.