Hunter Morera
U79718512
Algorithms Homework 2

1. Use the master theorem to solve each of the following recurrences:

a. $T(n) = 3T(n/27) + 1$
$a = 3, b = 27, f(n) = 1$
$c = log_{27}(3) = 1/3$
We can now compare n^c to f(n) and we can see that $1 \leq n^{\frac{1}{3}} \; \forall \; n \geq 1$. This means we can say that $f(n) = O(n^{c-\epsilon})$ for some $\epsilon > 0$ and thus by the master theorem the recurrence $T(n) = \Theta(n^{\frac{1}{3}})$.

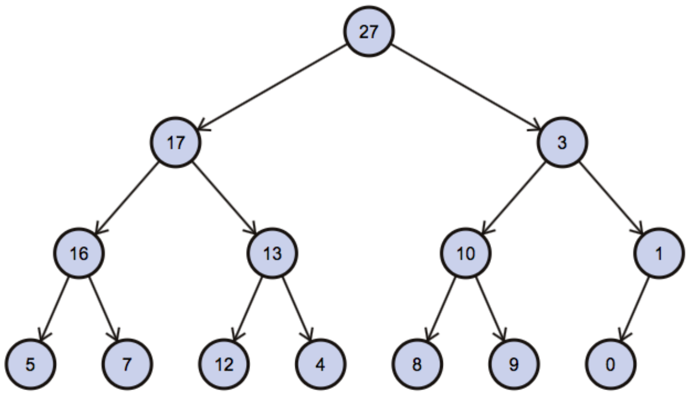
b. $T(n) = 7T(n/8) + lg(n)$
$a = 7, b = 8, f(n) = lg(n)$
$c = log_8(7) = 0.93$
We can now compare n^c to f(n) and we can see that $lg(n) \leq n^{0.93} \; \forall \; n \geq 0$. This means we can say that $f(n) = O(n^{c-\epsilon})$ for some $\epsilon > 0$ and thus by the master theorem the recurrence $T(n) = \Theta(n^{0.93})$.

c. $T(n) = 2T(n/4) + n$
$a = 2, b = 4, f(n) = n$
$c = log_4(2) = 0.5$
We can now compare n^c to f(n) and we can see that $n \geq n^{0.5} \; \forall \; n \geq 1$. This means that we can say $f(n) = \Omega(n^{c+\epsilon})$ for some $\epsilon > 0$. We now must show that $af(n/b) < f(n)$ for some large n.
$af(n/b) < f(n)$
$2 * (n/4) < n$ Lets use n = 100 and we get:
$2 * (100/4) < 100$
$50 < 100$
Therefore we can say that $af(n/b) < f(n)$ is true for some $n > 100$. Thus by the master theorem the recurrence $T(n) = \Theta(n)$.

d. $T(n) = 2T(n/4) + n^2$
$a = 2, b = 4, f(n) = n^2$
$c = log_4(2) = 0.5$
We can now compare n^c to f(n) and we can see that $n^2 \geq n^{0.5} \; \forall \; n \geq 1$. This means that we can say $f(n) = \Omega(n^{c+\epsilon})$ for some $\epsilon > 0$. We now must show that $af(n/b) < f(n)$ for some large n.
$af(n/b) < f(n)$
$2 * (n/4) < n^2$ Lets use n = 100 and we get:
$2 * (100/4) < 100^2$
$50 < 10000$
Therefore we can say that $af(n/b) < f(n)$ is true for some $n > 100$. Thus by the master theorem the recurrence $T(n) = \Theta(n^2)$.

e. $T(n) = 2T(n/4) + \sqrt{n}\, lg(n)$
$a = 2, b = 4, f(n) = \sqrt{n}\, lg(n)$
$c = log_4(2) = 0.5$
We can now compare n^c to f(n) and we can see that $\sqrt{n}\, lg(n) \geq \sqrt{n}$ however it is not polynomially different therefore we must say that $f(n) \in \Theta(n^c)$ therefore by the master theorem we can say that $f(n) \in \Theta(n^c lgn)$ therefore the recurrence would be $\Theta(\sqrt{n}\, lg^2 n)$

2.In order to show the operations of Max-Heapify(A,1) we must first draw the heap structure of array A = [27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0].



Applying Max-Heapify(A,1) would apply Max-Heapify to the node containing 27. As we can see the two child nodes of 27 are 17 and 3 which are both less than 27. Therefore no swaps will occur and the heap after the call of Max-Heapify(A,1) will remain exactly the same.

3. The first operation in the heap sort algorithm is to call the Build-Max-Heap(A).

Build-Max-Heap(A = [5, 13, 2, 25, 7, 17, 20, 8, 4])

The Build-Max-Heap function will then call the Max-Heapify function on (A,i) for i values of 4,3,2,1. This is because $\lfloor length(A)/2 \rfloor$ is equal to 4. Below I will show the Array after each call of Max-Heapify.

Max-Heapify(A,4): A = [5, 13, 2, 25, 7, 17, 20, 8, 4] No swaps are made because 25 is larger than both of its child nodes.

Max-Heapify(A,3): A = [5, 13, 20, 25, 7, 17, 2, 8, 4] The value 20 and the value 2 were swapped on this call.

Max-Heapify(A,2): A = [5, 25, 20, 13, 7, 17, 2, 8, 4] The value 13 and 25 are swapped on this call.

Max-Heapify(A,1): A = [25, 13, 20, 8, 7, 17, 2, 5, 4] The value 5 and 25 were swapped, 5 and 13 were then swapped, and finally 5 and 8 were swapped on this call.

At this point we have finished the Build-Max-Heap function which returns back to the function heapsort. Heapsort will now execute a for loop from length of A to 2, inside the loop it will swap A[i] and A[1], then it will call Max-Heapify(A,1). Below I will show the array after each call of Max-Heapify.

i = 9, 25 and 4 were swapped before the function call, Max-Heapify(A,1): A = [20, 13, 17, 8, 7, 4, 2, 5, 25] The values 4 and 20, as well as 4 and 25 were swapped on this call.

i = 8, 20 and 5 were swapped before the function call, Max-Heapify(A,1): A = [17, 13, 5, 8, 7, 4, 2, 20, 25] The values 5 and 17 were swapped on this call.

i = 7, 17 and 25 were swapped before the function call, Max-Heapify(A,1): A = [13, 8, 5, 2, 7, 4, 17, 20, 25] The values 2 and 13, as well as 2 and 8 were swapped on this call.

i = 6, 13 and 4 were swapped before the function call, Max-Heapify(A,1): A = [8, 7, 5, 2, 4, 13, 17, 20, 25] The values 4 and 8, as well as 4 and 7 were swapped on this call.

i = 5, 8 and 4 were swapped before the function call, Max-Heapify(A,1): A = [7, 4, 5, 2, 8, 13, 17, 20, 25] The values 4 and 7 were swapped on this call.

i = 4, 2 and 7 were swapped before the function call, Max-Heapify(A,1): A = [5, 4, 2, 7, 8, 13, 17, 20, 25] The values 2 and 5 were swapped on this call.

i = 3, 5 and 2 were swapped before the function call, Max-Hepify(A,1): A = [4, 2, 5, 7, 8, 13, 17, 20, 25] The values 2 and 4 were swapped on this call.

i = 2, 4 and 2 were swapped before the function call, Max-Heapify(A,1): A = [2, 4, 5, 7, 8, 13, 17, 20, 25] No swaps were made on this function call.

Finally we have completed all of the steps of the heapsort function and we can see that we ended up with a sorted array of A = [2, 4, 5, 7, 8, 13, 17, 20, 25].

4. To prove that the recurrence $T(n) = 2T(0.5n - 3) + n \in \Omega(nlgn)$ we will use the substitution method. It follows that:

$T(n) \geq 2 * d * (\frac{n}{2} - 3) * lg(\frac{n}{2} - 3) + cn$

By the problem statement we know that we can not drop the -3 constant, however we can make a substitution of 3 with $\frac{n}{4}$ $\forall\, n \geq 12$ we can see that if we make this replacement we will always be subtracting a value of 3 or greater. As well we can drop the +cn term because it only causes the LHS to be larger for positive values of n. The problem then becomes:

$T(n) \geq 2 * d * (\frac{n}{2} - \frac{n}{4}) * lg(\frac{n}{2} - \frac{n}{4})\ \forall\, n \geq 12$

$T(n) \geq 2 * d * \frac{n}{4} * lg(\frac{n}{4})\ \forall\, n \geq 12$

$T(n) \geq 2 * d * \frac{n}{4} * [lg(n) - lg(4)]\ \forall\, n \geq 12$

$T(n) \geq 2 * d * \frac{n}{4} * [lg(n) - 2]\ \forall\, n \geq 12$

We can now replace the -2 constant with $\frac{1}{2}lg(n)\ \forall\, n \geq 16$ it follows that:

$T(n) \geq 2 * d * \frac{n}{4} * [lg(n) - \frac{1}{2}lg(n)]\ \forall\, n \geq 16$

$T(n) \geq 2 * d * \frac{n}{4} * \frac{1}{2} * lg(n)\ \forall\, n \geq 16$

$T(n) \geq \frac{d}{4} * n * lg(n)\ \forall\, n \geq 16$

$T(n) \geq \frac{d}{4} * nlg(n)\ \forall\, n \geq 16$

We have therefore proven that $T(n) = 2T(0.5n - 3) + n \in \Omega(nlgn)\ \forall\, n \geq 16\ and\ d > 0$

5. Answer the following questions about the Heapsort function.

a. The total number of swaps for heapsort on the array A = [5, 13, 2, 25, 7, 17, 20, 8, 4] is 23, 5 are from the Build-Max-Heap function and 18 are from the heapsort function calls, we can see this from all of the steps demonstrated in problem 2.

b. If you replace Max-Heapify(A,1) with Build-Max-Heap(A) the number of required swaps would remain the same at 18, however the number of calls to Max-Heapify will be more but only the first element in the array doesn't satisfy the heap condition, thus the number of swaps will remain the same, even if the number of comparisons goes up.

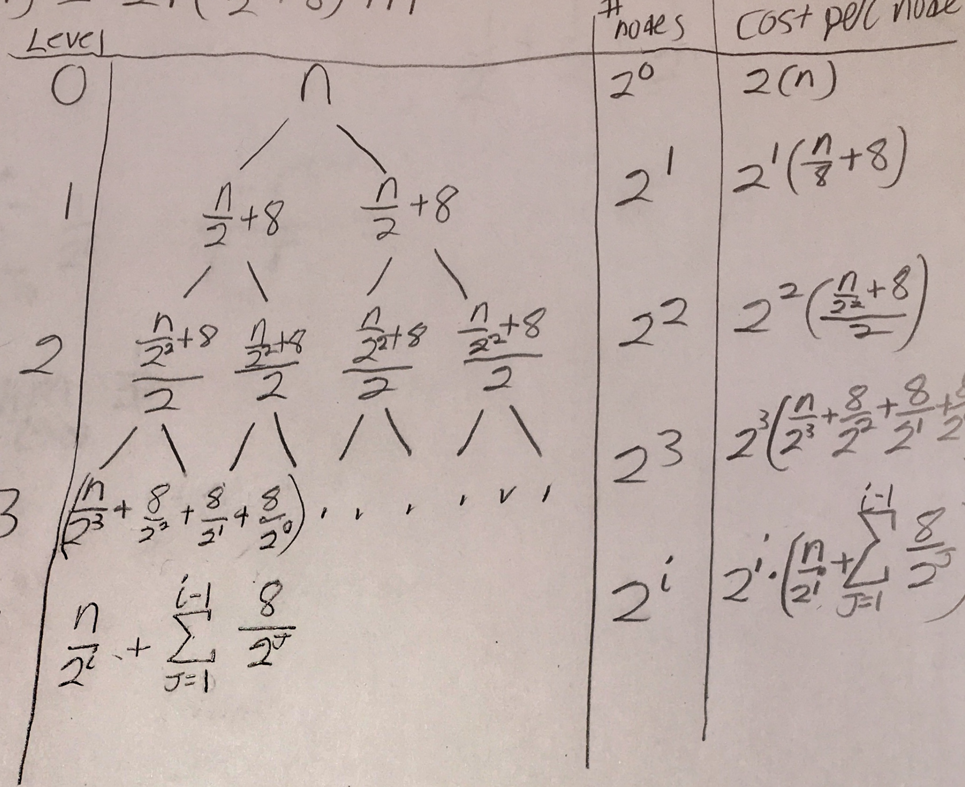
c. As was stated in the previous answer, the number of swap operations if we changed the heapsort code would remain the same, however, the number of actual comparisons that are done would increase. This increase in the number of comparisons does increase the upper bound of heapsort from $O(nlgn)$ to $O(n^2)$. This is because on each call of Max-Heapify(A,1) after the swap of A[1] and A[i], only the first element is the only one that doesn't satisfy the heap condition. However, if you simply called Build-Max-Heap instead you would make more unnecessary calls to Max-Heapify, thus increasing runtime to $O(n^2)$.

6. For the first problem of $T(n) = 2T(n/2) + sin(n)$ we can say that the f(n) part is $sin(n)$ which has the ability to fall between -1 and 1 mathematically speaking, however when we discuss the analysis of an algorithm the value must be a positive number as there is no such thing as negative run time. The $sin(n)$ here represents the conquer step of a divide and conquer algorithm, and we know that an algorithm cannot put together all of the solutions to a number of sub problems in negative time. Therefore we can say that $sin(n)$ would be bounded by constant time of 1. Thus we can do the comparison of 1 to $log_b(a)$ and we see that n^1 would dominate, meaning that $f(n) \in O(n^c)$ thus causing the recurrence to equal $\Theta(n)$ time.

As for the second problem of $T(n) = T(n/2) + nsin(n) + 2n$ We can first do some algebra to the problem to understand it more clearly, if we factor out an n from the last two terms we get $T(n) = T(n/2) + n(sin(n) + 2)$ again we can see here that the $sin(n) + 2$ part of the equation will have an asymptotic bound of 1 for the same reason as the previous problem. Thus we would be comparing an $f(n) = 1$ and a $log_b(a)$ of $log_2(1) = 0$ thus the $n^c = n^0 = 1$ any number n to the power of 0 would be 1, thus it would fall under the theta recurrence, meaning $f(n) \in \Theta(n^c)$ case and thus would be $\Theta(nlgn)$ time.

7. The use of the recursion tree method is shown on the next page:

$T(n) = 2T(\frac{n}{2}+8)+n$



Level		# nodes	cost per node
0	n	2^0	$2(n)$
1	$\frac{n}{2}+8 \quad \frac{n}{2}+8$	2^1	$2^1(\frac{n}{8}+8)$
2	$\frac{\frac{n}{2^2}+8}{2} \quad \frac{\frac{n}{2^2}+8}{2} \quad \frac{\frac{n}{2^2}+8}{2} \quad \frac{\frac{n}{2^2}+8}{2}$	2^2	$2^2\left(\frac{\frac{n}{2^2}+8}{2}\right)$
3	$\left(\frac{n}{2^3}+\frac{8}{2^2}+\frac{8}{2^1}+\frac{8}{2^0}\right) \ldots$	2^3	$2^3\left(\frac{n}{2^3}+\frac{8}{2^2}+\frac{8}{2^1}+\frac{8}{2^0}\right)$
i	$\frac{n}{2^i}+\sum_{j=1}^{i-1}\frac{8}{2^j}$	2^i	$2^i \cdot \left(\frac{n}{2^i}+\sum_{j=1}^{i-1}\frac{8}{2^j}\right)$

Recursion stops at $T(n)=1$ so height of tree can be calculated as $1=\frac{n}{2^i}=\log_2 n = i$ therefor height of tree is $\log_2 n$.

We can write the sumation of $\frac{8}{2^j}$ as $8\sum_{j=0}^{i-1}\frac{1}{2^j}$, as $i \to \infty$, the summation $=2$, we get $8 \cdot 2 = 16$

we then get $\frac{n}{2^i}+16$ on the ith level, we can ignore the constant & multiply by the number of nodes $2^i\left(\frac{n}{2^i}\right)=n$, thus each level of tree will cost n time & height of tree is $\log_2 n$.
Thus the reccurence will run $\Theta(n \lg n)$ time.

8. In order to prove the correctness of the Build-Max-Heap function we must start by determining the loop invariant.

Loop Invariant: At the start of every iteration of the for loop inside the Build-Max-Heap function, each node $i+1, i+2, ...n$ is a root of a Max-Heap.

We also know that each node $(\lfloor \frac{n}{2} \rfloor + 1), (\lfloor \frac{n}{2} \rfloor + 2), ...n$ is a leaf, thus being a root of a trivial Max-Heap. We know that $i = \lfloor \frac{n}{2} \rfloor$ prior to the first iteration of the for loop, this means that the loop invariant is initially true.

Any child of node i are at indexes higher than i, therefore by the loop invariant both children would be roots of Max-Heaps. If we assume that $i+1, i+2, ...n$ are all roots of Max-Heaps, Max-Heapify will make node i a Max-Heap root. When i is decremented after each iteration we re-establish the loop invariant.

Thus when $i = 0$ we know the for loop terminates, therefore we know that by the loop invariant each node, including node 1 is the root of a Max-Heap.
We can therefore say by induction that the Build-Max-Heap function does correctly take an array, build a Max-Heap, and then terminate, therefore the correctness of this algorithm has been proved.