Assignment 1
John Abo
30088517

1. .

a) Since the output must be a permutation of the input, we also need to prove that A' contains the same element as A. (We already know that it is in ascending order)

b) Loop invariant:

For each iteration, the smallest element in $A[i..n]$ cannot be at a position greater than j.

Initialization:

Before entering the inner for loop, $j = A.length$, where $A.length$ is the last position in the array. Therefore, the loop invariant holds prior to the first iteration of the loop.

Maintenance:

The only change occurring in the loop is that the smallest element is being moved towards the position $A[i]$, meaning it is either decrementing or staying the same position. The position of the smallest element can then never be greater than j. Decrementing j preserves the loop invariant, as the lowest element should at least be in $j - 1$ prior to decrementing.

Termination:

We see that the condition for the for loop is that $j < i + 1$, so we must have $j = i$ after the loop terminates. Substituting that into the loop invariant we get:

The smallest element in subarray $A[i..n]$ cannot be in a position be in a position greater than i. Since i is the first position, we can conclude that the that the first element of the array is now the smallest as well.

c) Loop invariant:

The subarray $A[1..i - 1]$ should be sorted and contain the smallest elements of $A[1..n]$.

Initialization:

When $i = 1$ before entering the subarray $A[1..i - 1]$ is empty and is trivially sorted. Therefore, the loop invariant holds prior to the first iteration of the loop.

Maintenance:

The loop ensures the smallest element that isn't already an element of $A[1..i - 1]$ is moved to $A[i]$, where $A[i]$ is in the next value that should be added to $A[1..i - 1]$. Therefore when i is incremented, $A[1..i - 1]$ will be trivially sorted and contain the smallest elements of $A[1..n]$, thus the loop invariant holds.

Termination:

We see the condition for the for loop to end is that $i > A.length - 1$, so we must have $i = A.length$ after the loop terminates. Substituting that into the loop invariant we get:

The subarray $A[1..A.length - 1]$ should be sorted and contain the smallest elements of $A[1..n]$. The last element $A[A.length]$ should then be the largest element. We can then conclude that the entire array sorted.

d) The outer loop is run at $n-1$ times, while the inner loop is run $n-i$ times, meaning the worst case of this implementation of bubble sort is $O(n^2)$. This is the same as insertion sort.

2.

a) *Highest* Growth Rate

$2^{2^{n+1}}$		
2^{2^n}		$f(n+1) > f(n)$ for this function
$(n+1)!$		
$n!$		Same reasoning as above
$n2^n$		
e^n		
2^n		In this case $e > 2$
$\left(\frac{3}{2}\right)^n$		In this case $2 > \frac{3}{2}$
$(\lg(n))!$		
$n^{\lg(\lg(n))}$	$\lg(n)^{\lg(n)}$	These functions are equal
n^3		
n^2	$4^{\lg(n)}$	These functions are equal
$n\lg(n)$		
$\lg(n!)$		Because $n^n > n!$
$2^{\lg(n)}$	n	These functions are equal
$\left(\sqrt{2}\right)^{\lg(n)}$		
$2^{\sqrt{2\lg(n)}}$		In this case $2 > \sqrt{2}$
$\lg^2(n)$		
$\lg(n)$		$f(n)^2 > f(n)$ for this function
$\sqrt{\lg(n)}$		
$\ln(\ln(n))$		
$2^{\lg^*(n)}$		
$\lg^*(n)$	$\lg^*(\lg(n))$	These functions are equal
$\lg(\lg^*(n))$		
1	$n^{1/\lg(n)}$	These functions are equal, constant growth rate is the lowest

Lowest Growth Rate

b) A function $f(n)$ that is neither $O(g_i(n))$ or $\Omega(g_i(n))$ for all functions $g_i(n)$ in part a) would be $(g_i(n)!)^{\sin(n)}$, where it would be 0 when $\sin(n) = 0$ and $g_i(n)!$ when $\sin(n) = 1$. This prevents the function $g_i(n)$ from being either greater or less than for all $n > n_0$, and is thus $f(n)$ neither $O\big(g_i(n)\big)$ nor $\Omega\big(g_i(n)\big)$

3. We're given: $T(n) = 2T\left(\frac{n}{4}\right) + \sqrt{n}$

a) We see that:
$a = 2$
$b = 4$
$f(n) = \sqrt{n}$
The polynomial is:

$$n^{\log_b a} = n^{\log_4 2} = n^{1/2} = \sqrt{n}$$

Observe that $f(n)$ and the polynomial have the same order of growth, so we get that:

$$T(n) = \Theta\left(n^{\log_4 2}\lg(n)\right) = \Theta(\sqrt{n}\lg(n))$$

b) Conjecture:

Guess the solution will be in the form:

$$T(n) = O(\sqrt{n}\lg(n))$$

Meaning there exists $n_0 > 0$ and $c > 0$ such that for all $n \geq n_0$ we have:

$$T(n) \leq c \cdot \sqrt{n}\lg(n)$$

Basis:

We know $T(1) = C$ for some constant C. We can consider $n_0 = 4$

$$T(4) = 2T\left(\frac{4}{4}\right) + \sqrt{4} = 2C + 4 \leq c \cdot 2\lg(4) = c \cdot 4$$

So long as we choose $C \geq 0$ and $c \geq 1$

Induction Step:

Supposed for every $m < n$, we have:

$$T(m) \leq c \cdot \sqrt{m}\lg(m)$$

We then see that:

$$\begin{aligned} T(n) &= 2T\left(\frac{n}{4}\right) + \sqrt{n} \\ &\leq 2 \cdot c \cdot \sqrt{\frac{n}{4}}\lg\left(\frac{n}{4}\right) + \sqrt{n} \\ &= 2 \cdot c \cdot \frac{\sqrt{n}}{2}(\lg(n) - \lg(4)) + \sqrt{n} \\ &= c\sqrt{n}\lg(n) - 2c\sqrt{n} + \sqrt{n} \\ &\leq c\sqrt{n}\lg(n) \end{aligned}$$

This holds for all $c > 0$

From the basis, we chose $n_0 = 4$ and $c \geq 1$. The induction step works for all $c > 0$, so we choose $n_0 = 4$ and $c = 1$. This gives us

$$T(n) \leq c \cdot g(n_0)$$

For all $n \geq 4$ and $c = 1$. We then get

$$T(n) = O(g(n))$$

4. Code is included with the submission.

5. $RIGHT(i)$ and $LEFT(i)$ are algorithms given in class to get position of right and left child. The singly list is a dynamic set and $Insert$ adds element to the head of the linked list while $Extract$ receives a specified value then removes that value from the list. $Insert$ and $Extract$ is assumed to take constant time.

$Largest - Elements(H, k)$:

Input: Where $k \leq n$ and H is a max heap

Output: An Array $A[1..k]$ containing the k largest elements of H

//This will hold the positions of nodes being compared which can vary in number

Create singly linked list $searching$

//If k is 0 or negative, then it should return the empty array as there is no value to add

If $k < 1$ then return A;

$A[1] = H[1]$

//Adds nodes to be searched to $searching$ list

$searching.Insert(RIGHT(1))$;

$searching.Insert(LEFT(1))$;

$i = 2$

While $i \leq k$ do

//The following line removes the next maximum from max-heap because it is searching for the maximum among all the children nodes in $searching$. This takes linear time, as the list must be traversed, so get $T(n) = O(k)$ for the next step.

$Extract$ position from $searching$ with largest key and set to $largest$

$searching.Insert(RIGHT(largest))$;

$searching.Insert(LEFT(largest))$;

$A[i] = H[largest]$;

$i = i + 1$;

Return A;

Loop Invariant: $searching$ contains the children of the keys that are an element of $A[1..i-1]$ and $H[1..n]$, so that A contains the $i - 1$ largest elements of array H.

Initialization: $searching$ is inserted with the children of $A[1..1]$ which is $H[1]$. Thus, they are elements of both arrays that would satisfy the loop invariant, as the $H[1]$ is the maximum of array H.

Maintenance: The next largest element that should follow $A[i-1]$ should be at a position that is part of the list $searching$, which will be found and set to $largest$. $A[i]$ will be set to that next largest element, so when i is incremented, the loop invariant holds.

Termination: We see the condition for the loop is $i \leq k$ so $i = k + 1$ when the loop ends. Substituting this back into the loop invariant we get:

$searching$ contains the children of the keys that are an element of $A[1..k]$ and $H[1..n]$, so that A contains the k largest elements of array H. Thus the array $A[1..k]$ contains the k largest elements of $H[1..n]$.

Worst case scenario:

The most elements that $searching$ will contain is $2 \times k$ when i nodes have been searched and both of the children are added to the list. Since this search must be done k times, we can conclude that $T(n) = O(k^2)$.