CSCI 5454: PS1

Robert Werthman

1.

Let's say these algorithms solve an array sorting problem.

- Let algorithm A be bubblesort with a worst-case runtime of n^2.
- Let algorithm B be mergesort with a worst-case runtime of $n * log(n)$.
- Let C be the newly designed sorting algorithm with a worst-case runtime of $h(n)$.

In this case, $O(min(f(n), g(n)))$ will become $O(n * log(n))$ because it is the smaller of the two runtimes.
If $h(n)$ is $log(n)$ then $h(n)$ achieves the running time $O(min(f(n), g(n)))$ because $log(n)$ does not grow faster than $n * log(n)$ and is therefore bounded above by it.

Yes, you can achieve a running time exactly $min(f(n), g(n))$. Algorithm C would need to be designed in such a way that its running was equal to $min(f(n), g(n))$.

2.

Proposition/Claim: For any real constants a and b, where $b > 0$, the asymptotic relation $(n + a)^b = \Theta(n^b)$ is true.

Theorem: The asymptotic relation $(n + a)^b = \Theta(n^b)$ is true iff:

- There exists positive constants c_1, c_2, n_0 s uch that $0 \leq c_1(n^b) \leq (n + a)^b \leq c_2(n^b)$ for all $n \geq n_0$.

In order to prove the proposition above we must find some constants c_1, c_2, n_0 to satisfy the above bulleted sentence.

Proof:
First we want to find the floor and ceiling of $n + a$ so we can create an inequality similar to the one in the theorem above.

1. If $|a| \leq n$ then we can say that $n + a \leq n + |a| \leq 2n$ (Ceiling of $n + a$).
2. If $|a| \leq \frac{1}{2}n$ then we can say that $n + a \geq n - |a| \geq \frac{1}{2}n$ (Floor of $n + a$).

Now if $2|a| \leq n$ then we can combine the floor and ceilings into an compound inequality that holds true :

$$0 \leq \frac{1}{2}n \leq n + a \leq 2n$$

The only thing missing from this new equation is a power of b. Raising the new equation to a power of b gives:

$$0 \leq (\frac{1}{2}n)^b \leq (n + a)^b \leq (2n)^b \Rightarrow 0 \leq (\frac{1}{2})^b n^b \leq (n + a)^b \leq (2)^b n^b$$

Extracting the constants c_1, c_2, n_0 from this equation yields $c_1 = (\frac{1}{2})^b$, $c_2 = 2^b$, and $n_0 = 2|a|$ since $n \geq 2|a|$. These represent one solution.

3.

$f(n) = \Omega g(n)$ means that for all values to the right of some n_0 the value of $f(n)$ is on or above $cg(n)$.

$n!$	e^n	$(\frac{3}{2})^n$	$(lg\,n)!$	n^2	$n\,lg\,n$	$lg(n!)$	n	$(\sqrt{2})^{lg\,n}$	2^{lg*n}	$n^{1/lg\,n}$	1

Equivalence Classes

$lg(n!) = \Theta(n\,lg\,n)$
$n^{1/lg\,n} = \Theta(1)$

4.

a.

$T(n) = T(n-1) + n$, $T(1) = 1$
I will use a recurrence tree to solve this recurrence relation.

$$\begin{array}{c} n \\ | \\ n-1 \\ | \\ n-2 \\ | \\ 2 \\ | \\ \vdots \\ | \\ 1 \end{array}$$

The height of the tree is n and the cost at the root starts at n and decreases by 1 each level in the tree.
This means that the total cost of the tree is n.
So $T(n) = O(cost * depth) = O(n^2)$.

b.

$T(n) = 2T(n/2) + n^3$, $T(1) = 1$
I will use the master method to solve this recurrence relation.
$a = 2, b = 2, f(n) = n^3$
so $n^{\log_b a} = n^{\log_2 2} = n$
This tells us that the first 2 rules of the master theorem do not apply.

1. $f(n) \neq O(n^{1-\epsilon})$
2. $f(n) \neq \Theta(n)$

This leaves the 3rd rule of the master theorem as the solution.

3. $f(n) = n^3 = \Omega(n^{1+\epsilon})$ if $\epsilon = 1$. And $2f(n/2) \leq cf(n) \Rightarrow 2(n/2)^3 \leq cn^3$ if $c = \frac{1}{2}$ and $n \geq 1$.

Therefore, $T(n) = \Theta(n^3)$.

5.

a.

```
Data: Nearly sorted array of size n integers
Result: Completely sorted array
for j = 2 to A.length do
    key = A[j];
    i = j - 1;
    while i > 0 and A[i] > key do
        A[i+1] = A[i];
        i = i - 1;
    end
    A[i+1] = key;
end
```

Algorithm 1: Insertion-Sort(A)

Analysis: In order to figure out the running time of Insertion Sort we need to add up the cost of each statement in the algorithm.

- If the array is of size n then the statement **for j = 2 to A.length** will execute n times with a cost of c_1.

- The statements **key = A[j]** (inserting into an array) and **i=j-1** (setting a variable) will execute $n-1$ times each with a cost of c_2 and c_3 respectively.

- Since k elements are unsorted in this array than any unsorted element is no more than k places away from its sorted position. This means that the statement **while i > 0 and A[i] > key** could be executed in the worst case $\sum_{j=2}^{n} k$ times with a cost of c_4.

- The statements **A[i+1] = A[i]** (inserting into an array) and **i = i + 1** (setting a variable) are executed $\sum_{j=2}^{n} k-1$ times with a cost of c_5 and c_6 respectively.

- Finally, the statement **A[i+1] = key** (inserting into an array) is executed $n-1$ times with a cost of c_7.

Therefore, the equation for the runtime, $T(n)$, of insertion-sort is:

$$\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4(\sum_{j=2}^{n} k) + c_5(\sum_{j=2}^{n} k-1) + c_6(\sum_{j=2}^{n} k-1) + c_7(n-1) \\
&= c_1 n + c_2(n-1) + c_3(n-1) + c_4(k(n-1)) + c_5(\sum_{j=2}^{n} k-1) + c_6(\sum_{j=2}^{n} k-1) + c_7(n-1)
\end{aligned}$$

Since $k < n$ further reduction of $T(n)$ would yield a linear function of n so we can say the runtime would turn out to be $O(n)$.

b.

The sorting algorithm I suggest to get a $O(n)$ runtime is Counting Sort.

```
Data: A is the input array of length n
Data: B is the sorted array of length n
Data: k is the highest integer in A
let C[0..k] be a new array
for i = 0 to k do
  | C[i] = 0
end
for j = 1 to A.length do
  | C[A[j]] = C[A[j] + 1
end
for i = 1 to k do
  | C[A[j]] = C[i] + C[i − 1]
end
for j = A.length downto 1 do
  | B[C[A[j]]] = A[j]
  | C[A[j]] = C[A[j]] − 1
end
```

Algorithm 2: Counting-Sort(A, B, k)

Analysis:

- Initializing $C[0..k]$ takes $k+1$ time to execute and costs c_0.
- The statement **for i = 0 to k** take $k+1$ times to execute and cost c_1.
- The statements **for j = 1 to A.length** and **j = A.length downto 1** take n times to execute and cost c_3 and c_4 respectively.
- The statement **i = 1 to k** takes k times to execute and costs c_2.

The equation for the runtime, $T(n)$, of Counting Sort is:

$$T(n) = c_0(k+1) + c_1(k+1) + c_3 n + c_4 n + c_2 k \dots$$

Reducing $T(n)$ further would show that the runtime of Counting Sort is a linear function of n that runs in a linear time of $O(k+n)$. If $k = O(n)$ then the running time is $\Theta(n)$.

c.

(b) doesn't contradict the $\Omega(n\, log\, n)$ lower bound given on page 59 of the textbook because the algorithm is not a comparison sorting algorithm.
It has been proven that any comparison sort must make $\Omega n\, log\, n)$ comparisons in the worst case to sort n elements. Since counting sort is not a comparison sorting algorithm its runtime is not bounded by $\Omega(n\, log\, n)$.