CS/Math 240: Intro to Discrete Math 3/22/2011

Solutions to Homework 6

Instructor: Dieter van Melkebeek

Problem 1

We have the following relationships.

$$n^5 \log^2 n = \Theta(2^{5 \log n + \log\log n} \log(n^5)) \tag{1}$$
$$2^{5 \log n + \log\log n} \log(n^5) = O(4^{3 \log n}) \tag{2}$$
$$4^{3 \log n} = O(n^{\log n}) \tag{3}$$
$$n^{\log n} = 2^{(\log n)^2} \tag{4}$$
$$2^{(\log n)^2} = O(2^{(2^{\sqrt{\log n}})}) \tag{5}$$
$$2^{2^{\sqrt{\log n}}} = O(2^{\sqrt{n}}) \tag{6}$$
$$2^{\sqrt{n}} \sim n^5 2^{2 \log\log n} + 2^{\sqrt{n}} \tag{7}$$

We now prove them one by one, and argue that these results are the best possible. Before we do so, let's recall the following facts about logarithms and exponents. All logarithms are base 2.

$$(a^b)^c = a^{bc} \tag{8}$$
$$\log(m^c) = c \log m \tag{9}$$
$$2^{\log m} = m \tag{10}$$

Also note that the seven equations we seek to prove do not characterize the relationships of all pairs of expressions this problem asks you to compare. However, it is easy to extend our equations to compare the growths of any two of the eight functions we analyze in this problem. It is not always this easy, but it is in this case. Let's explain how to do it on an example.

Say f, g, h are functions, and suppose, for example, that we have $f(n) = O(g(n))$ and $g(n) = \Theta(h(n))$. Then there exist constants d_1, c_2 and d_2 such that $f(n)/g(n) \leq d_1$ and $c_2 \leq g(n)/h(n) \leq d_2$. We then see $g(n) \leq d_2 h(n)$, so $f(n) \leq d_1 g(n) \leq d_1 d_2 h(n)$, so $f(n)/h(n) \leq d_1 d_2$, and $f(n) = O(h(n))$. Since we have no lower bound on $f(n)/g(n)$, finding $f(n) = O(h(n))$ is the best we can get from the information given to us. We summarize what we can say in Table 1.

Intuitively, Table 1 makes sense. Note that if two functions are equal, then they are asymptotically equivalent too, but not necessarily the other way around (for example (7) cannot be changed to an equality). We also saw in Lecture 14 that when two functions f and g are asymptotically equivalent, then $f(n) = \Theta(g(n))$, but not necessarily the other way around. We also saw that $f(n) = \Theta(g(n))$ implies $f(n) = O(g(n))$, but the other implication doesn't need to hold.

Now we are ready to prove our seven relationships. The previous paragraphs then explain how to obtain all of the other relationships.

	$f(n) = g(n)$	$f(n) \sim g(n)$	$f(n) = \Theta(g(n))$	$f(n) = O(g(n))$
$g(n) = h(n)$	$f(n) = h(n)$	$f(n) \sim h(n)$	$f(n) = \Theta(h(n))$	$f(n) = O(h(n))$
$g(n) \sim h(n)$	$f(n) \sim h(n)$	$f(n) \sim h(n)$	$f(n) = \Theta(h(n))$	$f(n) = O(h(n))$
$g(n) = \Theta(h(n))$	$f(n) = \Theta(h(n))$	$f(n) = \Theta(h(n))$	$f(n) = \Theta(h(n))$	$f(n) = O(h(n))$
$g(n) = O(h(n))$	$f(n) = O(h(n))$	$f(n) = O(h(n))$	$f(n) = O(h(n))$	$f(n) = O(h(n))$

Table 1: What can we say about the relationship between f and h using the relationship between f and g and the relationship between g and h. For example, if $f(n) = O(g(n))$ and $g(n) = \Theta(h(n))$, we can conclude that $f(n) = O(h(n))$.

Proof of (1). We first rewrite the right-hand side of (1). Split the product in three parts: $2^{5\log n}$, $2^{\log\log n}$, and $\log(n^5)$. Now let's use some facts about powers and logarithms to rewrite them in a more convenient form.

Using (8) with $b = \log n$ and $c = 5$, we rewrite $2^{5\log n} = (2^{\log n})^5$. This is equal to n^5 by (10) with $m = n$.

We can also rewrite $2^{\log\log n} = \log n$ by (10) with $m = \log n$.

Finally, we can use (9) with $m = n$ and $c = 5$ to get $\log(n^5) = 5\log n$.

Now we multiply our three results together to get $2^{5\log n+\log\log n}\log(n^5) = 5n^5\log^2 n$.

Hence, we see that

$$\frac{1}{5} \leq \frac{n^5\log^2 n}{2^{5\log n+\log\log n}\log(n^5)} \leq \frac{1}{5} \tag{11}$$

which proves (1).

Note that $n^5\log^2 n \not\sim 2^{5\log n+\log\log n}\log(n^5)$ because (11) implies that the ratio of the two functions stays fixed at $1/5$ and, therefore, won't get arbitrarily close to one. We also have $n^5\log^2 n \neq 2^{5\log n+\log\log n}\log(n^5)$, and big Oh is not as tight as big Theta. Thus, our answer is the best possible one. □

Before we prove the other relationships, we give the following lemma that is generally applicable.

Lemma 1. *Let $a, b, c \in \mathbb{R}$ be such that $a < c$. Then $m^a\log^b m = O(m^c)$, and $m^a\log^b m \neq \Omega(m^c)$.*

Proof of (2). We already know that $2^{5\log n+\log\log n}\log(n^5) = 5n^5\log^2 n$. Using (8) and (10) we can also rewrite $4^{3\log n}$ as

$$4^{3\log n} = \left(2^2\right)^{3\log n} = 2^{6\log n} = \left(2^{\log n}\right)^6 = n^6.$$

By Lemma 1 we have $5n^5\log^2 n = O(n^6)$ and we don't get a big Theta relationship between the two functions because Lemma 1 also tells us that $5n^5\log^2 n \neq \Omega(n^6)$. □

Proof of (3). Let's rewrite $4^{3\log n} = 2^{6\log n}$. We compare this to $n^{\log n}$, which we rewrite as $(2^{\log n})^{\log n} = 2^{\log^2 n}$. Now $6\log n = O(\log^2 n)$ and $6\log n \neq \Omega(\log^2 n)$ by Lemma 1 with $m = \log n$, $a = 1$, $b = 0$, and $c = 2$. Hence, the best we can hope for when comparing $4^{3\log n}$ and $n^{\log n}$ is (3). □

Proof of (4). We already saw in the proof of (3) that $n^{\log n} = 2^{(\log n)^2}$. □

Proof of (5). We compare the exponents $\log^2 n$ and $2^{\sqrt{\log n}}$. Let $m = \sqrt{\log n}$. Then our goal is to compare m^4 and 2^m. Observe that $\log(m^4) = 4 \log m$ and $\log(2^m) = m$. By Lemma 1, $4 \log m = O(m)$ and $4 \log m \neq \Omega(m)$, so $m^4 = O(2^m)$ and $m^4 \neq \Omega(2^m)$. Thus, $\log^2 n = O(2^{\sqrt{\log n}})$ and $\log^2 n \neq \Omega(2^{\sqrt{\log n}})$. It follows that $2^{\log^2 n} = O(2^{(2^{\sqrt{\log n}})})$, and this is the best we can get. □

Here we warn the reader that it is not true in general that if $f(n) = O(g(n))$, then $2^{f(n)} = O(2^{g(n)})$. For example, consider $f(n) = n$ and $g(n) = n/2$. Then $f(n) = O(g(n))$ because $f(n)/g(n) \leq 2$. But $2^{f(n)}/2^{g(n)} = 2^n/2^{n/2} = 2^{n/2}$, which is not bounded above by any constant. Thus, we see that $2^{f(n)} \neq O(2^{g(n)})$.

The constant d for which $f(n)/g(n) \leq d$ must be less than one if we want to conclude that $2^{f(n)} = O(2^{g(n)})$. Note that in the previous proof we had $\log^2 n \neq \Omega(2^{\log n})$, which suggests that we can drive the constant d as low as we want.

Proof of (6). Now our goal is to compare the exponents $2^{\sqrt{\log n}}$ and $\sqrt{n}$. We rewrite

$$\sqrt{n} = n^{1/2} = (2^{\log n})^{1/2} = 2^{(\log n)/2}$$

By Lemma 1 with $m = \log n$, $a = 1/2$, $b = 0$, and $c = 1$, we have $\sqrt{\log n} = O((\log n)/2)$, and $\sqrt{\log n} \neq \Omega((\log n)/2)$. We then get $2^{\sqrt{\log n}} = O(2^{(\log n)/2})$ and $2^{\sqrt{\log n}} \neq \Omega(2^{(\log n)/2})$, which implies that $2^{2^{\sqrt{\log n}}} = O(2^{(\log n)/2})$ and $2^{2^{\sqrt{\log n}}} \neq \Omega(2^{2^{(\log n)/2}})$. Now (6) follows. □

Proof of (7). Note that $n^5 2^{2 \log \log n} = n^5 \log^2 n$, and we know that $n^5 \log^2 n / 2^{\sqrt{n}}$ is not bounded below by any constant $\epsilon > 0$ (this claim follows from combining equations (1) through (6)). Now we realize that

$$1 - \epsilon < 1 \leq \frac{n^5 \log^2 n + 2^{\sqrt{n}}}{2^{\sqrt{n}}} = \frac{n^5 \log^2 n}{2^{\sqrt{n}}} + 1 \leq 1 + \epsilon,$$

so (7) follows. □

Problem 2

Recall the definition of Ω, Θ, O. Given $f : \mathbb{N} \to \mathbb{R}^+$ and $g : \mathbb{N} \to \mathbb{R}^+$, we say

- $f(n) = \Omega(g(n))$ if
$$(\exists c \in \mathbb{R}^+)(\exists N \in \mathbb{N})(\forall n \geq N) \;\; c \leq \frac{f(n)}{g(n)} \tag{12}$$
- $f(n) = O(g(n))$ if
$$(\exists d \in \mathbb{R}^+)(\exists N \in \mathbb{N})(\forall n \geq N) \;\; \frac{f(n)}{g(n)} \leq d \tag{13}$$
- $f(n) = \Theta g(n)$ if
$$(\exists c, d \in \mathbb{R}^+)(\exists N \in \mathbb{N})(\forall n \geq N) \;\; c \leq \frac{f(n)}{g(n)} \leq d \tag{14}$$

(a) In the given problem, we have to figure out if $f = \Omega(g)$ implies that $f^2 = \Omega(g)$. This means that if f is asymptotically bounded from below by some positive constant times g, then so is f^2. Intuitively, this need not be the case if f and g are small. This is because then $f^2(n)$ is less than $f(n)$ for all n, so it is possible that the lower bound for f does not hold for f^2. We need to exhibit an example where where the given statement does not hold true.

Let $f(n) = g(n) = \frac{1}{n}$. Then $f^2(n) = \frac{1}{n^2}$. We have that $f = \Omega(g)$. We claim that the statement $f^2 = \Omega(g)$ is false. We prove this by contradiction. Assume that the statement is true. This means that there exists $c \in \mathbb{R}^+, N \in \mathbb{N}$ such that for all $n \geq N$, $c \leq \frac{f^2(n)}{g(n)} = \frac{\frac{1}{n^2}}{\frac{1}{n}} = \frac{1}{n}$. This means that for all $n \geq N$, $nc \leq 1$. But we can always choose $n = \lceil \frac{1}{c} \rceil + 1$, in which case the inequality would not hold true. Hence, no such c, N can exist. Therefore $f^2 = \Omega(g)$ is not true.

(b) The given statement is true and we give a proof for the statement. Since $f = O(g)$, this means that there exist $d \in \mathbb{R}^+$ and $N \in \mathbb{N}$ such that $f(n) \leq d.g(n)$ for all $n \geq N$. Adding $g(n)$ to both sides of the inequality, we get $f(n) + g(n) \leq d.g(n) + g(n) = (d + 1)g(n)$.

Now, f is function from $\mathbb{N}$ to $\mathbb{R}^+$. This means that $0 \leq f(n)$ for all $n \in \mathbb{N}$. Adding $g(n)$ to both sides of the inequation, $g(n) \leq f(n) + g(n)$.

Hence, we have $g(n) \leq f(n)+g(n) \leq (d+1)g(n)$ for all $n \geq N$, which means that $f+g = \Theta(g)$. Therefore the given statement is true.

Problem 3

(a) We will give two different solutions to this problem.

Algebraic Solution: In this solution, we will solve the given summation algebraically. The summation is over the terms of a sequence that starts with 1 and where each next term in the sequence is obtained by multiplying the previous term by a fixed constant r. Such a sequence is called a *geometric sequence*.

Consider

$$S = 1 + r + \ldots + r^k$$

Multiplying both the sides by the common ratio r, we get

$$rS = r + r^2 + \ldots + r^k + r^{k+1}$$

Subtracting the two expressions,

$$\begin{aligned}(1-r)S &= 1 - r^{k+1} \\ S &= \frac{1-r^{k+1}}{1-r} = \frac{r^{k+1}-1}{r-1}.\end{aligned}$$

Solution using Induction: Next we provide a proof by induction on k. We would like to prove that $P(k) : \sum_{i=0}^{k} r^i = \frac{r^{k+1}-1}{r-1}$ holds for all integers $k \geq 0$.

The base case is $k = 0$. LHS $= \sum_{i=0}^{0} r^i = r^0 = 1$. RHS $= \frac{r^{0+1}-1}{r-1} = \frac{r-1}{r-1} = 1$. LHS = RHS, so the base case holds.

Now, suppose $P(k)$ holds true for some integer $k \geq 0$. We would like to prove that $P(k+1)$ follows, i.e. $\sum_{i=0}^{k+1} r^i = \frac{r^{k+2}-1}{r-1}$. Applying the induction hypothesis on the LHS, we get

$$\begin{aligned}\sum_{i=0}^{k+1} r^i &= \left(\sum_{i=0}^{k} r^i\right) + r^{k+1} = \frac{r^{k+1}-1}{r-1} + r^{k+1} \\ &= \frac{r^{k+1}-1}{r-1} + \frac{r^{k+1}(r-1)}{r-1} = \frac{r^{k+1}-1+r^{k+1}(r-1)}{r-1} \\ &= \frac{r^{k+2}-1}{r-1}\end{aligned}$$

This completes the induction step of the proof. Hence, we have shown that $P(k)$ holds for all integers $k \geq 0$.

(b) We have to derive a constant c such that n^c is asymptotically both an upper bound and a lower bound on $f(n)$ up to constant factors. Let's try to analyze $f(n)$ using the underlying recursion tree. We saw how to do this in Lecture 15.

Let us first consider the case where $n = 3^k$ for some integer $k \geq 0$ The recursion tree for evaluation of $f(n)$ can be seen in figure 1. In this tree, an instance with parameter $n > 1$

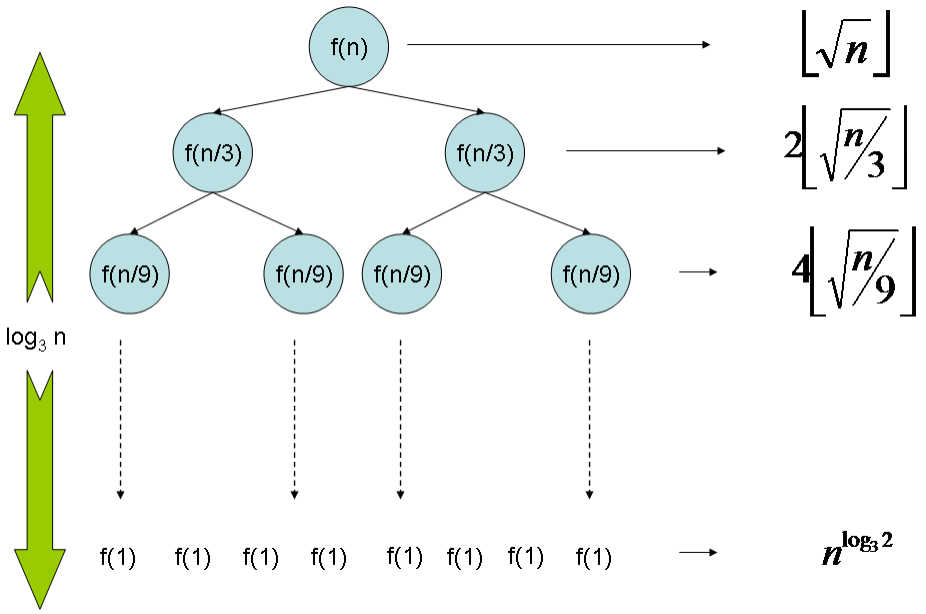

Figure 1: Recursion Tree

makes 2 calls to instances with parameter $n/3$ and, in addition, takes $\lfloor\sqrt{n}\rfloor$ work. If we count the levels from 0, then at level i there are 2^i instances, each with parameter 3^{k-i}. the leaves of the tree correspond to parameter value 1, and all appear at level k.

In order to compute $f(n)$, we aggregate the amount of work attributed locally to the calls at each level of recursion. The top level($i = 0$) contributes $\lfloor\sqrt{n}\rfloor$ to the value of $f(n)$. On the next level, the two calls each contribute $\lfloor\sqrt{n/3}\rfloor$ to the value of $f(n)$, for a total of $2\lfloor\sqrt{n/3}\rfloor$ at that level. Continuing this way, at the i^{th} level with $i < k$, there are 2^i calls, each contributing $\lfloor\sqrt{n/3^i}\rfloor$ to the value of $f(n)$, for a total of $2^i\lfloor\sqrt{n/3^i}\rfloor$. At the bottom level, $i = k$, we have 2^k contributions of $f(1) = 1$ each, for a total of 2^k. Since $k = \log_3 n$, we have that $2^k = 2^{\log_3 n} = 3^{\log_3 2 \cdot \log_3 n} = (3^{log_3 n})^{\log_3 2} = n^{\log_3 2}$.

Adding up the local contributions over all levels of the recursion tree, we have

$$f(n) = \lfloor\sqrt{n}\rfloor + 2\lfloor\sqrt{n/3}\rfloor + 4\lfloor\sqrt{n/9}\rfloor + \cdots + 2^{k-1}\lfloor\sqrt{3}\rfloor + 2^k$$

Now $\sqrt{n} - 1 < \lfloor\sqrt{n}\rfloor \leq \sqrt{n}$. We will use these inequalities to figure out the bounds. Let us first derive an upper bound.

$$\begin{aligned} f(n) &= \lfloor\sqrt{n}\rfloor + 2\lfloor\sqrt{n/3}\rfloor + 4\lfloor\sqrt{n/9}\rfloor + \cdots + 2^{k-1}\lfloor\sqrt{3}\rfloor + 2^k \\ &\leq \sqrt{n} + 2\sqrt{n/3} + 4\sqrt{n/9} + \cdots + 2^{k-1}\lfloor\sqrt{3}\rfloor + 2^k \\ &= \sqrt{n} + 2\sqrt{n/3} + 4\sqrt{n/9} + \cdots + 2^k\sqrt{n/3^k} \\ &= \sqrt{n}(1 + 2/\sqrt{3} + 4/\sqrt{9} + \cdots + 2^k/\sqrt{3^k}) \end{aligned}$$

If you notice carefully, the expression obtained in the last step involves a a geometric sum with common ratio $r = 2/\sqrt{3}$. We use the result obtained in 3(a) to evaluate this expression. Hence,

we have that $f(n) \geq g(n)$ where

$$\begin{aligned} g(n) &= \sqrt{n} \cdot \sum_{i=0}^{k} (2/\sqrt{3})^i \\ &= \sqrt{n} \frac{(2/\sqrt{3})^{k+1} - 1}{2/\sqrt{3} - 1} \\ &= \sqrt{n} \frac{(2/\sqrt{3})(2/\sqrt{3})^{k} - 1}{2/\sqrt{3} - 1} \end{aligned}$$

Now, we will simplify the term $2/\sqrt{3}^k$. We know that $k = \log_3 n$. Hence,

$$(2/\sqrt{3})^k = (2/\sqrt{3})^{\log_3 n} = 3^{\log_3(2/\sqrt{3}) \cdot \log_3 n} = (3^{\log_3 n})^{\log_3(2/\sqrt{3})} = n^{\log_3(2/\sqrt{3})} = n^{\log_3 2 - \frac{1}{2}}.$$

Hence, $g(n) = \sqrt{n} \frac{(2/\sqrt{3}) n^{\log_3 2 - \frac{1}{2}} - 1}{2/\sqrt{3} - 1} = \frac{(2/\sqrt{3}) n^{\log_3 2} - \sqrt{n}}{2/\sqrt{3} - 1} = O(n^{\log_3 2})$. Thus, $g(n) = O(n^{\log_3 2})$.

We now derive the same lower bound up to a constant factor. We have that $f(n) \geq h(n)$ where

$$\begin{aligned} h(n) &= \sqrt{n} - 1 + 2(\sqrt{n/3} - 1) + 4(\sqrt{n/9} - 1) + \cdots + 2^{k-1}(\sqrt{3} - 1) + 2^k \\ &= g(n) - (1 + 2 + 4 + \cdots + 2^{k-1}) \end{aligned}$$

What we subtract from $g(n)$ is a geometric sum with ratio 2, and equals $2^k - 1 = n^{\log_3 2} - 1$. Using our expression for $g(n)$, we have

$$h(n) = (\frac{2/\sqrt{3}}{2/\sqrt{3} - 1} - 1) n^{\log_3 2} - \frac{2/\sqrt{3}}{2/\sqrt{3} - 1} \sqrt{n} = \Omega(n^{\log_3 2}).$$

Thus, we have shown that there exist positive reals a and b such that for all sufficiently large n of the form $n = 3^k$ where k is a nonnegative integer, $an^c \leq f(n) \leq bn^c$ for $c = \log_3 2$.

For an arbitrary positive integer n, there always exists a (unique) nonnegative integer k such that $3^k \leq n < 3^{k+1}$. Using the recurrence for $f(n)$, one can show by induction that if $n_1 \leq n_2$ then $f(n_1) \leq f(n_2)$. If follows that $f(3^k) \leq f(n) \leq f(3^{k+1})$. Using our bounds for $f(n)$ when n is a power of 3, we obtain that $a(n/3)^c \leq f(n) \leq b(3n)^c$ for all sufficiently large integers n. This shows that $f(n) = \Theta(n^c)$ where $c = \log_3 2$.

Problem 4

Part a

We start by an example to understand the behavior of Quicksort's while loop. We know from the description of the algorithm that the element $A[e]$ should be used as a pivot for the other elements in the array, and that after the pivot step, the array has been rearranged into two parts (one with all elements larger than $A[e]$, and the other with all elements less than or equal).

Working through the example array [54213], we see that after the while loop we have:

$[5_i \quad 4 \quad 2 \quad 1 \quad 3_{j,p}]$
$[5_i \quad 4 \quad 2 \quad 1_j \quad 3_p] \qquad A[i] > 3 \rightarrow j \leftarrow j - 1$
$[1_i \quad 4 \quad 2 \quad 5_j \quad 3_p] \qquad swap(i, j)$
$[1 \quad 4_i \quad 2 \quad 5_j \quad 3_p] \qquad A[i] < 3 \rightarrow i \leftarrow i + 1$
$[1 \quad 4_i \quad 2_j \quad 5 \quad 3_p] \qquad A[i] > 3 \rightarrow j \leftarrow j - 1$
$[1 \quad 2_i \quad 4_j \quad 5 \quad 3_p] \qquad swap(i, j)$
$[1 \quad 2 \quad 4_{i,j} \quad 5 \quad 3_p] \qquad A[i] < 3 \rightarrow i \leftarrow i + 1$
$Exit\ WHILE$
$[1 \quad 2 \quad 4 \quad 5 \quad 3]$

Where the range $A[b...i-1]$ are all $\leq A[e]$, and the range $A[j...e-1]$ are $> A[e]$.

We now use our observations of QuickSort's behavior, along with our knowledge of the desired output of the while loop (from the program description) to state our loop invariant. Our invariant is as follows:

Invariant A: *$b \leq i \leq j \leq e$, and $A[e]$ is at least as large as all elements $A[b...i-1]$, and less than all elements $A[j...e-1]$*

We see that at P(0), $b = i < j = e$, and the two ranges are empty arrays; so the invariant holds. Now, we assume P(k) and prove P(k + 1).

$b \leq i$ and $j \leq e$ hold since i only increments within the loop (and likewise j only decrements. We know at P(k), $i < j$, since from P(k) $i \leq j$, and the while loop would have exited had $i = j$ at P(k). Since the distance between i and j only ever decreases by one in a given iteration, we have $i \leq j$ in P(k+1). So the first half of our invariant holds.

We will prove the second half of the invariant through a proof by cases.

Case One: $A[i] > p$. Let i_k and j_k refer to the initial values of i and j during this iteration. In this case, the range $A[j_k...e-1]$ increases to $A[j_k - 1...e-1]$, while the range $A[b...i-1]$ is unchanged (and none of the elements in this range affected by the swap). The elements $A[j_k...e-1]$ are all $> A[e]$ from P(k). After the swap, the element $A[j_k - 1] \leftarrow A[i]$; so $A[j_k - 1] > A[e]$ from the if condition. Thus, when $j \leftarrow j_k - 1$, all elements in the range $A[j...e-1]$ are $> A[e]$, and the invariant holds.

Case Two: $A[i] \leq p$. In this case, the range $A[b...i_k - 1]$ increases to $A[b...i_k]$, while the range $A[j...e-1]$ is unchanged. The elements $A[b...i_k - 1]$ are all $\leq A[e]$ from P(k). From the condition, $A[i_k] \leq p$; so all elements in the range $A[b...i_k]$ are $\leq A[e]$. When $i \leftarrow i_k + 1$, all elements in the range $A[b...i-1]$ are $\leq A[e]$, and the invariant holds.

So, our overall invariant holds. We note that it is important to show a stronger argument in order to prove correctness for our algorithm; namely that the resulting array A following the loop

is the same as our original input array. Our main concern is that our swap commands are outside of the range of the array $A[b...e]$. To ensure that this is not the case, we introduce the following:

Invariant A_2: *The array A is a permutation of the original input array; that is, there are no elements in A not present in the original array, and vice versa*

The proof for this invariant follows directly from Invariant A: since $b \leq i \leq j \leq e$, and the swap command occurs on $A[i]$ and $A[j]$, we see that no swaps will occur out of bounds.

Part b

In order to fill in the missing line of code, we again look at the behavior of Quicksort - we know from the description of the algorithm that after the pivot step, Quicksort recursively sorts the two parts of the array (one with all elements larger than $A[e]$, and the other with all elements less than or equal). Given this, we know that the recursive calls on lines (10) and (11) are supposed to refer to these two parts of the array.

However, we see that without line (9), the second recursive call, $Quicksort(A, j+1, e)$ is called on an incorrect range. Invariant A states that after the while loop, the range $A[b...i-1]$ are all $\leq A[e]$, and the range $A[j...e-1]$ are $> A[e]$. But then the QuickSort$(A, j+1, e)$ incorrecly sorts $A[e]$ again. Further, $A[j]$ is not in the correct position of the array (since it is potentially greater than $A[e]$), but is never sorted again.

We can see that to get line (11) to refer to the correct range, it is necessary to **(9) swap $A[j]$ and $A[e]$**. The important thing to note is the purpose of this line; ostensibly, if we simply wanted the recursive calls to modify the correct range, we could have altered line (11).

By swapping the elements $A[e]$ and $A[j]$, we place our pivot value in a location in the array where $A[e]$ is greater than or equal to all the elements before it, and less than all the elements after it. This means that the element $A[e]$ is in its correct sorted position! This will be an important observation for our proof of correctness.

Proof of Correctness

We first prove partial correctness for QuickSort. In our base case, an array of size 1, $b = e$, so the array is returned (which is correct, as a singleton array is trivially sorted). For larger arrays, our proof of partial correctness follows directly from Invariant A.

As we have stated earlier, after the while loop, all elements $A[b...i-1] \leq A[e]$, and all elements from $A[j...e-1]$ are $> A[e]$. From our ranges of i and j from the invariant, we know that the while loop can only exit when $i = j$; so $A[b...i-1]$ and $A[j...e-1]$ make up the full array (excluding $A[e]$). We then know that, after our swap in line (9), all elements from $A[b...i-1] = A[b...j-1]$ are $\leq A[j]$, and all elements from $A[j+1...e]$ are $> A[j]$; so A[j] is in sorted position. We assume our recursive calls sort the ranges $A[b...j-1]$ and $A[j+1...e]$; this requires that $A[b...j-1]$ and $A[j+1...e]$ match the input specifications of QuickSort. For the array $A[b...j-1]$ to match the input specifications, $b \leq j-1+1$; this holds from Invariant A. Similiarly, for $A[j+1...e]$, $j+1 \leq e+1$ is met from our invariant that $j \leq e$.

From Invariant A, all elements $A[b...j-1] \leq A[e] < A[j...e-1]$. After the swap, then, all elements $A[b...j-1] \leq A[j] < A[j+1...e]$. We have already argued that A[j] is properly sorted above. So when the ranges $A[b...j-1]$, $A[j+1...e]$ are sorted by our recursive calls, our entire array A is sorted correctly as well.

To prove termination, we must show that the while loop terminates, and that our recursive calls will reach our base case. We see that the while loop terminates, since the value $j - i$ is initially nonnegative and decreases by 1 at every iteration of the loop; when it becomes equal to 0, the loop terminates. To show that our recursive calls will reach our base case, we note that the ranges $A[b...i-1]$ and $A[j+1...e]$ both have size less than the original range $A[b...e]$ since they do not include $A[j]$ (In fact, the sum of their ranges will be less than the original range, but this is not necessary). So each recursive call will be on an array of decreasing size, until the base case is reached ($j - i \leq 0$). So the program terminates, and the program is correct.

Part c

First, we will introduce a variable $n = e - b + 1$; n is the size of the range $A[b...e]$. This will simplify our arguments in the following sections.

We note that if no swaps occur in the while iteration of an array $A[b...e]$, the recursive calls on the array occur in the range range $A[b...i-1] = A[b...e-1]$, and $A[j+1...e] = A[e+1...e]$. The second recursive call immediately returns and adds no calls of (3), while the first call iterates on an array whose size is $e - 1 + b + 1$, or 1 less than the original array. This case would give a total number of iterations of (3) equal to $n + T(n-1)$ (the while loop will iterate $(n-1)$ times and then make a final check to exit). We recall that if this recurrence starts at P(1) = 1, it will give a total number of iterations = $\frac{(n)(n+1)}{2}$. However, since an array of size 1 will return immediately at line (1), with no calls of the while loop, we have P(1) = 0; P(2), however, will equal 2, and the recurrence holds from that point. So our actual recurrence is a sum of terms from 2 to n, or = $\frac{(n)(n+1)}{2} - 1$.

To show that this is actually the maximum number of iterations possible, we must show that there is no other situation which gives more iterations of (3), and we must show that there is an array which fits the recurrence.

We see that any fully sorted array will fit the recurrence stated above; no swaps will occur on this array during any iteration of the while loop, or any recursive call. Thus $T(n) \geq \frac{(n)(n+1)}{2} - 1$. It remains to show that this recurrence is the maximum number of iterations possible for QuickSort, in other words, $T(n) \leq \frac{(n)(n+1)}{2} - 1$.

We show that our reccurrence equation holds for T(0) and T(1); both an empty array and a single array have no iterations of (3). Now, we assume that the maximum number of iterations of (3) on all array from size 2 up to size k follow the pattern $T(k) = \frac{(k)(k+1)}{2} - 1$. We will prove that the maximum iterations of (3) on an array of size k + 1 is $\frac{(k+1)(k+2)}{2} - 1$.

For an array of size k + 1, there will be k iterations of the while loop before the recursive calls. These calls will be on arrays of size $k - i$ and i, respectively, with $k \geq i \geq 0$. So each of these calls will take at most T(k-i) and T(i) iterations of (3) altogether. If $i = 0$ or $k - i = 0$, the total number of iterations will be T(0) + T(k) + k+1, or at most $\frac{(k)(k+1)}{2} - 1 + k + 1 = \frac{(k+1)(k+2)}{2} - 1$. If $i = 1$ or $k - i = 1$, the total number of iterations will be T(1) + T(k-1) + k+1, which will be strictly less than this value (since T(1) = 0, and T(k-1) < T(k)). So in both of these cases, the maximum value reached is $\frac{(k+1)(k+2)}{2} - 1$.

We now look at the remaining cases, $2 \leq i \leq k - 2$. In these cases, we know that the maximum value T(i) and T(k-i) will reach match the formula given in our inductive hypothesis. This gives a total number of iterations equal to $\frac{(k-i)(k-i+1)}{2} - 1 + \frac{(i)(i+1)}{2} - 1 = \frac{k^2 - 2ki + k + 2i^2}{2} - 2 = \frac{(k)(k+1) + 2i(i-k)}{2} - 2 = \frac{(k)(k+1) - 2i(k-i)}{2} - 2$. The value $i(k-i)$ is a quadratic whose second degree term has a negative

coefficient, and can be shown to be positive whenver $2 \leq i \leq k-2$. So in these cases, we have $< \frac{(k)(k+1)}{2}$ iterations maximum from these recursive calls. When added to the $k+1$ iterations of the while loop at this step, we have a value less than the desired maximum; this shows that in fact the upper bound on the number of iterations of line (3) is achieved when $i = 0$, with that bound being $\frac{(k+1)(k+2)}{2} - 1$.

Now we have shown that the maximum number of iterations of (3) for QuickSort on an array of size n is $\frac{(n)(n+1)}{2} - 1$; and we have given an example array that satisfies this formula. So $T(n) \leq \frac{(n)(n+1)}{2} - 1$ and $T(n) \geq \frac{(n)(n+1)}{2} - 1$, or $T(n) = \frac{(n)(n+1)}{2} - 1$. Our proof is complete.

Problem 5

This question deals with the following algorithm.

Function CountPairsInError(A, b, e)

Input: $b, m, e \in \mathbb{N}$, $b \leq e+1$, and $A[b..e]$ is an array of integers
Output: The number of pairs in error in $A[b..e]$.
Side Effect: $A[b..e]$ is sorted in nondecreasing order.

```
(1)  if b ≥ e then return 0
(2)  m ← ⌊(b + e)/2⌋
(3)  c ← CountPairsInError(A, b, m) + CountPairsInError(A, m + 1, e)
(4)  B[b..e] ← A[b..e]
(5)  i ← b; j ← m + 1; k ← b
(6)  while i ≤ m and j ≤ e do
(7)      if B[i] ≤ B[j] then A[k] ← B[i]; i ← i + 1; c ← c + j − m − 1
(8)                     else A[k] ← B[j]; j ← j + 1
(9)      k ← k + 1
(10) if i ≤ m then
(11)     A[k..e] ← B[i..m]
(12)     c ← c + (m − i + 1) · (e − m)
(13) return c
```

Throughout the solution, we will make references to the mergesort algorithm, so we also list it below.

Function MergeSort(A, b, e)

Input: $b, m, e \in \mathbb{N}$, $b \leq e+1$, and $A[b..e]$ is an array of integers
Side Effect: $A[b..e]$ is sorted in nondecreasing order.

```
(1)  if b ≥ e then return
(2)  m ← ⌊(b + e)/2⌋
(3)  MergeSort(A, b, m)
(4)  MergeSort(A, m + 1, e)
(5)  B[b..e] ← A[b..e]
(6)  i ← b; j ← m + 1; k ← b
(7)  while i ≤ m and j ≤ e do
(8)      if B[i] ≤ B[j] then A[k] ← B[i]; i ← i + 1
(9)                     else A[k] ← B[j]; j ← j + 1
(10)     k ← k + 1
(11) if i ≤ m then A[k..e] ← B[i..m]
```

Part a

We start by proving partial correctness. First observe that `CountPairsInError` rearranges the contents of the array A in exactly the same way as `MergeSort`. Notice that the code that is in `CountPairsInError` but not in `MergeSort` does not modify the array in any way. Thus, since the behavior of `CountPairsInError` on the array A is exactly the same as that of `MergeSort`, it follows that `CountPairsInError` sorts the array $A[b..e]$, assuming it terminates.

Now we show that `CountPairsInError` correctly computes the number of pairs in error.

For the base case, we have $b \geq e$. In that case $A[b..e]$ consists of at most one element, so there can't be any pairs in error. Therefore, returning zero in this case is the correct behavior.

Now suppose $b < e$. Then there are three ways of getting a pair that is in error. We can have (i) both indices in $\{b, \ldots, m\}$, (ii) both indices in $\{m+1, \ldots, e\}$, or (iii) one index in $\{b, \ldots, m\}$ and one index in $\{m+1, \ldots, e\}$.

Pairs in error that satisfy conditions (i) or (ii) are accounted for in the recursive calls. Since $b \leq e$, we have $b = 2b/2 = \lfloor 2b/2 \rfloor \leq \lfloor (b+e)/2 \rfloor$. This means that $b \leq m$, so the preconditions of the first recursive call to `CountPairsInError` are satisfied, and this call correctly counts the number of errors from pairs of type (i). Next, since $b < e$, we have $e = 2e/2 > (e+b)/2 \geq \lfloor (e+b)/2 \rfloor = m$, so $m < e$, which means that $m + 1 \leq e$. Thus, the preconditions of the second recursive call to `CountPairsInError` are also satisfied, and this call correctly computes the number of pairs in error of type (ii).

To complete the proof of partial correctness, we need to show that the rest of the code correctly computes the contribution of the pairs satisfying (iii). We rewrite the remaining code as the function below. Note that the number of pairs in error of type (iii) in the original array A[b..e] is exactly the same as the total number of pairs in error in A[b..m] after the parts A[b..m] and A[m+1..e] have been sorted. For that reason we call the function `CountPairsInErrorInSortedParts`.

Function CountPairsInErrorInSortedParts(A, b, m, e)

Input: $b, m, e \in \mathbb{N}$, $b \leq m < e$
Input: A - an array $A[b..e]$ of integers where $A[b..m]$ and $A[m+1..e]$ are sorted in nondecreasing order.
Output: The number of pairs in error in $A[b..e]$
Side Effect: $A[b..e]$ is sorted in nondecreasing order.

```
(1)  c ← 0
(2)  B[b..e] ← A[b..e]
(3)  i ← b; j ← m + 1; k ← b
(4)  while i ≤ m and j ≤ e do
(5)      if B[i] ≤ B[j] then A[k] ← B[i]; i ← i + 1; c ← c + j − m − 1
(6)                     else A[k] ← B[j]; j ← j + 1
(7)      k ← k + 1
(8)  if i ≤ m then
(9)      A[k..e] ← B[i..m]
(10)     c ← c + (m − i + 1) · (e − m)
(11) return c
```

With this function in hand, we can rewrite `CountPairsInError` as follows.

Function CountPairsInErrorTwo(A, b, e)

Input: $b, m, e \in \mathbb{N}$, $b \leq e+1$, and $A[b..e]$ is an array of integers
Output: The number of pairs in error in $A[b..e]$.
Side Effect: $A[b..e]$ is sorted in nondecreasing order.

(1) **if** $b \geq e$ **then return** 0
(2) $m \leftarrow \lfloor (b+e)/2 \rfloor$
(3) $c \leftarrow$ `CountPairsInErrorTwo`(A, b, m) + `CountPairsInErrorTwo`$(A, m+1, e)$
(4) $c \leftarrow c +$ `CountPairsInErrorInSortedParts`(A, b, m, e)
(5) **return** c

Recall the following two invariants we proved when proving correctness of the `MergeSortedParts` function in the proof of correctness of `MergeSort`.

Invariant 1. *After each iteration of the loop, $A[b..k-1]$ is sorted and consists of elements of $B[b..i-1]$ and $B[m+1..j-1]$. Furthermore, all elements in $A[b..k-1]$ are at most as large as all elements in $B[i..m]$ and $B[j..e]$.*

Invariant 2. *After every iteration of the loop, $b \leq i \leq m+1 \leq j \leq e+1$ and $k = i + j - (m+1)$.*

Observe that all errors in B involve exactly one element $B[i]$ with $b \leq i \leq m$. Each such element gets placed into A exactly once: either during some iteration of the loop, or after the loop terminates. We focus on the point when $B[i]$ is added to A, and show that in the iteration of the loop (or after the loop terminates), c gets increased by the number of pairs in error involving i.

We use the idea from the previous paragraph in the proof of an additional loop invariant. Before we state the invariant, we need some more notation. Let $B[b..i-1]B[m+1..e]$ be the array formed by putting together $B[b..i-1]$ into one array with $B[m+1..e]$, where the first part is $B[b..i-1]$ and the second part is $B[m+1..e]$.

Invariant 3. *After every iteration of the loop, c equals the number of pairs in error in the array $B[b..i-1]B[m+1..e]$.*

Proof. For the base case, $i = b$, so $B[b..i-1]$ is the empty array, and $B[b..i-1]B[m+1..e] = B[m+1..e]$ is sorted. There are no pairs in error in a sorted array, so $c = 0$ is the correct value.

Now assume that after some iteration of the loop, c counts the number of pairs in error in $B[b..i-1]B[m+1..e]$. Consider the next iteration of the loop. If $i = m+1$ or $j = e+1$, there isn't going to be another iteration of the loop, so assume that $i \leq m$ and $j \leq e$.

In this situation, Invariant 1 implies that all elements of $B[m+1..j-1]$ are less than $B[i]$. Since $B[b..m]$ is sorted, and since $b \leq i \leq m$ by Invariant 2, all elements of $B[b..i-1]$ are also less than $B[i]$. Finally, $B[i] \leq B[j]$, $m+1 \leq j \leq e$, and $B[m+1..e]$ is sorted, so all elements of $B[j..e]$ are at least $B[i]$. Thus, position i is in error only with positions $m+1$ through $j-1$, which is a total of $(j-1)-(m+1)-1 = j-m-1$ positions, all of which are positions in $B[b..i]B[m+1..e]$.

All other pairs in error in $B[b..i]B[m+1..e]$ come from the subarray $A[b..i-1]B[m+1..e]$, and we know that before this iteration of the loop, c was the count of those. Because every pair of indices in error in $B[b..i]B[m+1..e]$ involves exactly one index from $\{b, \ldots, i\}$, c counts the number of pairs in error in $B[b..i]B[m+1..e]$ after adding $j-m-1$ to c on line 5. Thus, the invariant is maintained after this iteration of the loop. □

Invariant 3 implies that once the loop is over, c counts the number of pairs in error in the array $B[b..i-1]B[m+1..e]$.

If the loop terminated because $i = m+1$, then note $B[b..i-1]B[m+1..e] = B[b..e]$, so c actually counts the number of pairs in error in $B[b..e]$. Furthermore, in this case, the condition on line 8 is false, so the body of the if statement doesn't execute, and the algorithm returns the correct value.

If the loop terminated because $j = e+1$, then an argument similar to the one done in the proof of Invariant 3 shows that for each ℓ in $\{i, \ldots, m\}$, the number of pairs in error involving ℓ in $B[b..\ell]B[m+1..e]$ is $e-(m+1)+1 = e-m$. We can view the body of the if statement on line 8 as the following loop.

Algorithm 1: Modified body of the if statement on line 8

```
(1) ℓ ← i
(2) while ℓ ≤ m do
(3)     A[k] ← B[ℓ]
(4)     c ← c + e − m
(5)     ℓ ← ℓ + 1
(6)     k ← k + 1
```

We can extend the argument from the proof of Invariant 3 to this loop and show that after every iteration, c counts the number of pairs in error in $B[b..\ell]B[m+1..e]$.

After this loop terminates, we have $\ell = m+1$, so c counts the number of pairs in error in $B[b..e]$, and `CountPairsInErrorInSortedParts` returns the correct value right after this loop terminates. This completes the proof of partial correctness.

The argument that `CountPairsInErrorInSortedParts` terminates is exactly the same as the proof of termination of `MergeSort` in Lecture 15.

Part b

The recursion tree of `CountPairsInError` is the same as the one for `MergeSort`. Also, the running time of `CountPairsInErrorInSortedParts` is $O(n)$ where $n = e-b+1$ just like in the analysis of the `MergeSortedParts` because the only additional operations `CountPairsInErrorInSortedParts` makes in addition to what `MergeSortedParts` does are at most n updates to the value of c. Thus, the running time of `CountPairsInErrorInSortedParts` is $O(n)$ just like the running time of `MergeSortedParts`.

We can use the same argument as we used in Lecture 15 to show that the contribution of each level of the recursion tree towards the running time of `CountPairsInError` with an array of size n is $O(n)$. Since there are a total of $O(\log n)$ levels in the recursion tree, this gives us a running time of $O(n \log n)$ as desired.