EECS 203: Discrete Mathematics
Fall 2023
Homework 12 (Indiv. Ungraded, Groupwork Graded)

Due: No Due Date

No late homework accepted past midnight.

Number of Problems: 5+2 Total Points: 60+16

This homework should be submitted as a PDF through Gradescope (see instructions on course site for more details). We strongly prefer students to compose homework solutions using a word processor (Google Docs or MS Word), or ideally using LaTeX, but we will accept handwritten homework submissions scanned/photographed and converted to PDF. Note that submitted files must be less than 50 mb in size, but they really should be much smaller than this. No email or Piazza regrade requests will be accepted. For more detail on regrade requests, please refer to course policies.

- We will not grade homework problems that were not properly matched on Gradescope. Please submit early and often and double-check that you matched your pages to their question.
- Explanation or justification for yes/no, true/false, multiple choice questions, and the like: You must provide some sort of explanation or justification for these types of questions (so we know you didn't just guess), unless explicitly specified below. For example, simply answering "true" for a T/F question without providing an explanation will receive little or no credit.
- All problems require work to be shown. Questions that require a numerical answer will not be given credit if no work/explanation is shown.
- Honor Code: By submitting this homework, you agree that you are in compliance with the Engineering Honor Code and the Course Policies for 203, and that you are submitting your own work.
- Hyperlinks on Submitting Homework to Gradescope: Instructions or Youtube Video

Individual Portion

1. Divide and C$O(n)$quer [10 points]

Consider each of these algorithms. Write a recurrence relation for the approximate number of steps each algorithm takes in terms of N, then give a big-theta estimate for the run time of the algorithm.

(a) **procedure** NthPowerOfTwo(N)
    if $N = 0$
        return 1
    partial := NthPowerOfTwo($\lfloor N/2 \rfloor$)
    if N mod $2 = 1$
        return $2 \cdot \text{partial} \cdot \text{partial}$
    else
        return $\text{partial} \cdot \text{partial}$

(b) **procedure** CountBetween(a, b, with $a < b$)
    $N := b - a$
    if N = 1
        return 1
    halfway = $a + \lfloor N/2 \rfloor$
    return CountBetween(a, halfway) + CountBetween(halfway + 1, b)

Solution:

(a) The algorithm makes one recursive call with half of the input size, then does a mod operation, a comparison, and up to two multiplications. These are all constant with respect to the input size, so the recurrence is $T(N) = T\left(\frac{N}{2}\right) + \Theta(1)$. For the master theorem, we have $a = 1, b = 2, d = 0$, so we have $\frac{a}{b^d} = \frac{1}{1} = 1$. That means the run time is $\Theta(N^0 \log_2 N) = \Theta(\log N)$

(b) The algorithm makes 2 recursive calls, each with about half the input size, then takes constant time to combine them. Our recurrence is $T(N) = 2T\left(\frac{N}{2}\right) + \Theta(1)$. For the master theorem, we have $a = 2, b = 2, d = 0$, so $\frac{a}{b^d} = \frac{2}{1} > 1$. That means the run time is $\Theta(N^{\log_2(2)}) = \Theta(N)$

Grading Guidelines [10 points]
Parts a and b:
+2 correct recurrence relation
+2 applies the master theorem with the correct values a, b, and d. (Doesn't have to be labeled a, b, and d, but uses the right numbers)
+1 correct final complexity

2. Big-O-ver it [12 points]

Give the tightest big-O estimate for each of these functions. Show your steps to arrive at the final expression.

(a) $f(n) = (n^2 + 2^n)(n! + n^n) + (n^2 + n!)(2^n + n^n)$

(b) $g(n) = (\log n + \sqrt{n})(2^n + 3^n)$

Solution:

(a) We can either expand the terms, or multiply the fastest-growing term in each expression since after expanding, those terms will end up dominating. So we have that $f(n)$ is on the order of $2^n n^n + n! \cdot n^n$, and since $n!$ dominates 2^n, $f(n)$ is $O(n! \cdot n^n)$.

(b) Similar to (a), we have that $g(n)$ is $O(3^n\sqrt{n})$, since$\sqrt{n}$ and 3^n are the fastest-growing terms in each of their respective expressions.

Grading Guidelines [12 points]
Part a:
+2 has $n!$ in answer
+2 has n^n in answer
+1 multiples $n!$ by n^n
+1 no additional terms/multipliers in expression
Part b:
+2 has 3^n in answer
+2 has $\sqrt{n}$ in answer
+1 multiples 3^n by $\sqrt{n}$
+1 no additional terms/multipliers in expression

3. What's your running time? [12 points]

Give the tightest big-O estimate for the number of operations (where an operation is arithmetic, a comparison, or an assignment) used in each of the following algorithms:
(a)

```
procedure findMax(a_1, a_2, ..., a_N: real numbers)
    max := 0
    for i := 1 to N
        if a_i > max
            max = a_i
    return max
```

(b)

```
procedure sumOddIndices(a_1, a_2, ...a_N: real numbers)
```

```
    i := 1
    oddIndexSum := 0
    while i ≤ N
        oddIndexSum := oddIndexSum + a_i
        i := i + 2
    return oddIndexSum
```

(c)

```
procedure findMinPowerAboveN(N: positive integer)
    i := 1
    while i ≤ N
        i := i * 2
    return i
```

(d)

```
procedure findMaxDifference(a_1, a_2, ..., a_N: real numbers)
    maxDiff := 0
    for i := 1 to N
        for j := 1 to N
            if a_i − a_j > maxDiff
                maxDiff := a_i − a_j
    return maxDiff
```

(e)

```
procedure countElementsGreaterThanMean(a_1, a_2, ..., a_N: real numbers)
    sum := 0
    numGreaterThanMean := 0
    for i := 1 to N
        sum := sum + a_i
    mean := sum/N
    for j := 1 to N
        if a_j > mean
            numGreaterThanMean := numGreaterThanMean + 1
    return numGreaterThanMean
```

Solution:

(a) $O(N)$. For each element in the list, there is a constant amount of work being done (one comparison, sometimes one assignment). The number of operations ranges from roughly N to $2N$ (depending on how many times it enters the if statement, not counting loop variable arithmetic), which is $O(N)$.

(b) $O(N)$. There is a constant amount of work done for every other element in the list, which is $\frac{N}{2}$ elements, so multiplying this by a constant is still $O(N)$.

(c) $O(\log N)$. Even though this has the same loop bound as the previous problem and

the same amount of work within the loop, the way the loop variable is updated affects how many times the loop body is executed. Since the loop variable is multiplied by 2 each time, the number of times the loop is executed is how many times 1 needs to be multiplied by 2 to reach N, which is $\log_2 N$, or $O(\log N)$.

(d) $O(N^2)$. There are two nested loops, and within a single iteration of the outer loop, there is a constant amount work done for each element of the list, which is $O(N)$. Since the list executes N times, the overall complexity is $O(N^2)$.

(e) $O(N)$. Even though there are two loops here as in option (d), they are not nested. So the first loop does a constant amount of work for each element in the list which is $O(N)$, and the second loop also does a constant amount of work for each element in the list, which is $O(N)$. Since the second loop is not done for each iteration of the first loop, but rather done after the first loop is done, we add these instead of multiplying. So we get roughly $2N$ operations, which is still $O(N)$.

Grading Guidelines [12 points]
Part a:
+1 reports $O(N)$ as runtime
+1 states $O(N)$ comes from iteration over elements
Part b:
+1 reports $O(N)$ as runtime
+1 states $\frac{N}{2}$ operations from iteration process
+1 simplifies $\frac{N}{2}$ to $O(N)$
Part c:
+1 reports $O(\log N)$ as runtime
+1 states $\log N$ operations from iteration counter doubling
Part d:
+1 reports $O(N^2)$ as runtime
+1 states N^2 operations from nested looping
Part e:
+1 reports $O(N)$ as runtime
+1 states N operations from one loop and N operations from the other loop
+1 simplifies to $O(N)$

4. Think Quick! [17 points]

In the implementation of Quicksort, the basis for the most commonly used sorting algorithms, there's a function known as *partition*. Given a list of numbers, it takes the last number, then groups together all the smaller numbers and all the bigger numbers. There are much more efficient implementations of this function, but they have the same big-Θ time complexity as ours.

```
procedure partition(int[] [a_1, a_2, ..., a_n])
    middle := a_n
    smaller := []
    for i := 1 to n
        if a_i < middle
            insert a_i at the end of smaller
    bigger := []
    for i := 1 to n
        if a_i > middle
            insert a_i at the end of bigger
    return smaller, middle, bigger
```

(a) For simplicity, assume all numbers are distinct and adding an element to an array takes $\Theta(1)$ time. What is the big-Θ runtime of *partition*?

There is another algorithm, called Quickselect, that uses *partition*. It finds the kth smallest element in an array. In other words, it gets the element that would be at index k if the array was sorted. In most cases, its big-Θ runtime is faster than many expect.

```
procedure quickselect(int[] [a_1, a_2, ..., a_n], int k)
    smaller, middle, bigger := partition([a_1, a_2, ..., a_n])
    if k ≤ numElements(smaller)
        // kth smallest element is smaller than middle
        return quickselect(smaller, k)
    else if k = numElements(smaller) + 1
        // kth smallest element is middle
        return middle
    else
        // kth smallest element is bigger than middle
        return quickselect(bigger, k − numElements(smaller) − 1
```

(b) For simplicity, assume *smaller* and *bigger* have $\frac{n}{2}$ elements[1] and *numElements* takes $\Theta(1)$ time. What is the big-Θ runtime of *quickselect*?

Solution:

(a) The bodies of both for loops run in $\Theta(1)$, and each for loop runs n times. Thus, *partition* runs in $\Theta(n)$ time.

[1]This is tricky to analyze because *smaller* and *bigger* aren't always $\frac{n}{2}$. However, they "usually" are, depending on exactly what we mean by "usually."

(b) We can find this using the Master Theorem. Let $T(n)$ be the runtime of *quickselect*. The algorithm starts by calling *partition*, which takes $\Theta(n)$. It then chooses between recursion into *smaller* or *bigger*, or the base case. We assume that both of those arrays have $\frac{n}{2}$ elements. So, in both recursive cases, this takes $T\left(\frac{n}{2}\right)$ time. Thus, $T(n) = T\left(\frac{n}{2}\right) + \Theta(n)$.

As defined in the Master Theorem, $a = 1$, $b = 2$, and $d = 1$. We can calculate $\frac{a}{b^d} = \frac{1}{2^1} = \frac{1}{2} < 1$, so we use the first case of the theorem. This tells us that $T(n)$ is $\Theta(n^d) = \Theta(n)$. So, Quickselect runs in linear time.

Grading Guidelines [17 points]
Part a:
+2 identifies that the body of the loops runs in $\Theta(1)$
+2 identifies that each loop runs $\Theta(n)$ times
+1 shows that the algorithm runs in $\Theta(n)$ time
Part b:
+2 identifies the Master Theorem as a useful tool and attempts to find a recurrence to apply it to
+2 identifies the $\Theta(n)$ term from *partition* (or other term if solution to part a differed)
+2 identifies the $T\left(\frac{n}{2}\right)$ term from recursion
+2 no incorrect terms in the recurrence (eg. two $T\left(\frac{n}{2}\right)$ instead of just one, but constant factors for the if statement checks are good)
+2 correct a, b, d for the Master Theorem (can be called anything, but need to identify the right numbers in the context of the theorem)
+2 correct application of the Master Theorem

Group Portion

1. Grade Groupwork 11

Using the solutions and Grading Guidelines, assess your Groupwork 11:

- Construct a table to assess your previous weeks work, below is an example of such a table.
- Please include your original groupwork submission in this grading submission, marking up the original submission for revisions
- For each rubric item, indicate whether or not your/your group's submission achieved this rubric item.
- For rubric items that weren't achieved, please visually indicate, whether with a sentence, annotation, mark-up, correction, or similar why your submission did not receive this item.
- You do not have to "redo" the problem correctly for the grading portion, however this is recommended.
- If your group submitted the same groupwork for the previous week, you may grade the same thing together. If not, we want you to give a grade of your version, which means submitting this groupwork assignment separately. Again, you could discuss this together or on your own.
- For extra credit, write yourself positive remark(s) associated to your work.

	(i)	(ii)	(iii)	(iv)	(v)	(vi)	(vii)	(viii)	(ix)	(x)	(xi)	Total:
Problem 2 (15)												/10
Problem 3 (20)												/8
Total:												/18