CSE 21 Fall 2018
Homework 2
Due: Sunday, October 14, 2018 at 10:00pm

INSTRUCTIONS

Homework should be done in groups of **one to three** people. You are free to change group members at any time throughout the quarter. Problems should be solved together, not divided up between partners. Homework must be submitted through **Gradescope** by **a single representative**. Submissions must be received by **10:00pm** on the due date, and there are no exceptions to this rule.

You will be able to look at your scanned work before submitting it. Please ensure that your submission is **legible** (neatly written and not too faint) or your homework may not be graded.

Students should consult their textbook, class notes, lecture slides, instructors, TAs, and tutors when they need help with homework. Students should not look for answers to homework problems in other texts or sources, including the Internet. You may ask questions about the homework in office hours, but **not on Piazza**.

Your assignments in this class will be evaluated not only on the correctness of your answers, but on your ability to present your ideas clearly and logically. You should **always explain** how you arrived at your conclusions and **justify your answers** with mathematically sound reasoning. Whether you use formal proof techniques or write a more informal argument for why something is true, your answers should always be well-supported. Your goal should be to **convince the reader** that your results and methods are sound.

For questions that require pseudocode, you can follow the same format as the textbook, or you can write pseudocode in your own style, as long as you specify what your notation means. For example, are you using "=" to mean assignment or to check equality? You are welcome to use any algorithm from class as a subroutine in your pseudocode. For example, if you want to sort list A using InsertionSort, you can call InsertionSort(A) instead of writing out the pseudocode for InsertionSort.

REQUIRED READING Rosen Sections 3.2 and 3.3

KEY CONCEPTS: Linear Search, Binary Search; Asymptotic notation, including the definitions of O, Θ, and Ω; best-case, worst-case, average-case; analyzing the run-time of algorithms

---

1. A list of n distinct integers $a_1, a_2, \ldots, a_n$ is called a *mountain list* if the elements reading from left to right first increase and then decrease. The location of the *peak* is the value i where a_i is greatest. For example, the list $1, 2, 3, 7, 6$ is a mountain list with a peak at 4. We also consider a list of increasing numbers to be a mountain list with a peak at n, and a list of decreasing numbers to be a mountain list with a peak at 1.

   (a) (5 points) Give pseudocode for an algorithm based on linear search that takes as input a mountain list $a_1, a_2, \ldots, a_n$ and returns the location of the peak. Show that your algorithm does no more than $n - 1$ comparisons on any input of size n.
   Solution:

```
procedure LinearPeak($a_1, a_2, \ldots, a_n$: mountain list)
1.  for $i := 1$ to $n - 1$
2.      if $(a_i > a_{i+1})$
3.          return $i$
4.  return $n$
```

   This algorithm does one comparison in each iteration of the for loop. On an input of size n, the for loop iterates $n - 1$ times (from $i = 1$ to $n - 1$) if it never returns in line 3, otherwise it iterates less than $n - 1$ times . Therefore, the algorithm does no more than $n - 1$ comparisons on any input of size n.

   (b) (5 points) Give an algorithm based on binary search that takes as input a mountain list $a_1, a_2, \ldots, a_n$ and returns the location of the peak. Your algorithm should be able to find the peak in any list of size 16 with no more than 4 comparisons (in general, up to k comparisons for a list of size 2^k.) You should provide pseudocode and an English description of your algorithm, but you do not need to prove that it meets the requirements for the number of comparisons.
   Solution:

```
procedure BinaryPeak($a_1, a_2, \ldots, a_n$: mountain list)
1.  $i := 1$
2.  $j := n$
3.  while $(i < j)$
4.      $m := \lfloor \frac{i+j}{2} \rfloor$
5.      if $(a_m < a_{m+1})$
6.          $i := m + 1$
7.      else
8.          $j := m$
9.  return $j$
```

   This algorithm looks at the middle element and the next element after that, and uses their values to determine whether the peak is in the left half or the right half of the list. If the middle element is smaller than the next element, then we know the peak is to the right of the middle element. Othwerise, we know that the peak is to the left of the middle element, or possibly the middle element itself. It continues to search the same way in either the left half or the right half of the list. The part of the list being searched gets smaller each time until there is only one element left, at which point the peak is located.

2. (5 points total - 1 points for each part) State the big-Θ class (in simplest form) that each of the following functions belongs to. Justify your answer in each case.

   (a) $\log(n) + n\log(n)$

   Solution: $\Theta(n\log(n))$

   (b) $n\log(n^2) + n\log(n)$

   Solution: $\Theta(n\log(n))$

   (c) $\sqrt{n} + \sqrt[3]{n}$

   Solution: $\Theta(\sqrt{n})$

   (d) $\log(n) + \log(2n) + \log(3n)$

   Solution: $\Theta(\log(n))$

   (e) $3^{2n} + 2^{3n}$

   Solution: $\Theta(9^n)$

3. (10 points total - 2 points for each part (a)-(e), part (f) is optional)

In this problem, we wish to compare the time taken to perform an algorithm on an input of size n versus on an input of size $4n$. We saw that the exact time will vary with environmental factors such as hardware and software details, but the order of the algorithm tells us how we expect the time to scale when we increase the input size.

For each function below, suppose we are running an algorithm of that order (i.e. in that Θ class) on an input of size n then again on an input of size $4n$. By what multiplicative factor do we expect the total run-time of our algorithm to increase?

For example, an answer of 3 means that we expect our algorithm to take 3 times as long on an input of size $4n$ versus on an input of size n. Similarly, an answer of n means that we expect it to take n times as long on an input of size $4n$ versus on an input of size n.

(a) $2n$

Solution:

$$\frac{2(4n)}{2(n)} = \frac{8n}{2n} = 4$$

(b) n^3

Solution:

$$\frac{(4n)^3}{n^3} = \frac{64n^3}{n^3} = 64$$

(c) 2^{n^2}

Solution:

$$\frac{2^{(4n)^2}}{2^{n^2}} = \frac{2^{16n^2}}{2^{n^2}} = 2^{15n^2}$$

(d) $n!$

Solution:

$$\frac{(4n)!}{n!} = \frac{(4n) * (4n-1) \cdots * 2 * 1}{n * (n-1) * \cdots * 2 * 1} = 4n * (4n-1) * \cdots * (n+1)$$

(e) n^n

Solution:

$$\frac{(4n)^{4n}}{n^n} = \frac{4^{4n} * n^{4n}}{n^n} = 4^{4n} * n^{3n}$$

(f) (Optional) $\log_2 n$

Solution:

$$\frac{\log_2 4n}{\log_2 n} = \frac{(\log_2 n) + (\log_2 4)}{\log_2 n} = 1 + \frac{2}{\log_2 n}$$

This says that when we quadruple the input size to an algorithm that takes time $\Theta(\log_2 n)$, the amount of time required grows by a factor barely more than 1. Algorithms that take logarithmic time are considered to be very fast for this reason.

4. (20 points total - 2 points each) For each part, say whether the statement is true or false and justify your answer. **All logarithms are base 2 unless otherwise noted.**

   (a) $2^n \in \Theta(4^n)$
   Solution: FALSE. $4^n = 2^n * 2^n$ so 2^n differs from 4^n by a multiple of 2^n, which is not constant. Two functions are in the same Θ class only when they differ by a constant multiple. Alternatively, we can use the limit method to compute

$$\lim_{n\to\infty} \frac{4^n}{2^n} = \lim_{n\to\infty} 2^n = \infty,$$

   which means 4^n grows faster than 2^n.

   (b) $\log(n^2) + \log(10^{10}n^{10}) \in O(\log n)$
   Solution: TRUE. We can use log rules to simplify.

$$\begin{aligned}\log(n^2) + \log(10^{10}n^{10}) &= 2\log(n) + \log((10n)^{10})\\ &= 2\log(n) + 10\log(10n)\\ &= 2\log(n) + 10\log(10) + 10\log(n)\\ &= 12\log(n) + 10\log(10).\end{aligned}$$

   Note that $10\log(10)$ is just a constant, so this function is $O(\log n)$.

   (c) $\sqrt{2^n} \in O((\sqrt{2})^n)$
   Solution: TRUE. These functions have the same growth rate because they are actually the same function, just written in different ways.

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

   (d) $\ln n \in \Theta(\log_2 n)$
   Solution: TRUE. The change of base formula shows that logarithms with any base are related by a constant factor. The change of base formula in this case says $\ln n = \dfrac{\log_2 n}{\log_2 e} = \dfrac{1}{\log_2 e}\log_2 n$ and since we can ignore constants, we know $\ln n$ and $\log_2 n$ are in the same Θ class.

   (e) $\lfloor \dfrac{n+1}{2} \rfloor \in \Theta(n)$
   Solution: TRUE. We have $\frac{n+1}{2} - 1 \leq \lfloor \frac{n+1}{2} \rfloor \leq \frac{n+1}{2}$, so we can sandwich $\lfloor \frac{n+1}{2} \rfloor$ between two linear functions. Thus, $\lfloor \frac{n+1}{2} \rfloor$ grows at least as quickly as the linear function $\frac{n+1}{2} - 1$ and at most as quickly as the linear function $\frac{n+1}{2}$. Therefore it grows at the same rate as linear functions and is in $\Theta(n)$.

   (f) $\dfrac{n}{\ln n} \in \Omega((\ln n)^2)$
   Solution: TRUE. We can see that $\dfrac{n}{\ln n}$ grows faster than $(\ln n)^2$ using the limit method and l'Hopital's Rule.

$$\begin{aligned}
\lim_{n\to\infty} \frac{\frac{n}{\ln n}}{(\ln n)^2} &= \lim_{n\to\infty} \frac{n}{(\ln n)^3} \\
&= \lim_{n\to\infty} \frac{1}{3(\ln n)^2 \cdot \frac{1}{n}} \\
&= \lim_{n\to\infty} \frac{n}{3(\ln n)^2} \\
&= \lim_{n\to\infty} \frac{1}{6(\ln n) \cdot \frac{1}{n}} \\
&= \lim_{n\to\infty} \frac{n}{6\ln n} \\
&= \lim_{n\to\infty} \frac{1}{6 \cdot \frac{1}{n}} \\
&= \lim_{n\to\infty} \frac{n}{6} \\
&= \infty
\end{aligned}$$

Since $\dfrac{n}{\ln n}$ grows faster than $(\ln n)^2$, it is true that $\dfrac{n}{\ln n} \in \Omega((\ln n)^2)$.

(g) $2^n \in O(n!)$

Solution: TRUE. For all $n > 3$, we have $2^n < n!$. Thus, it is true that $2^n \in O(n!)$.

(h) $2^n \in O(3^n/n^2)$

Solution: TRUE. We shall prove that $2^n \in O(3^n/n^2)$ using the limit method by showing

$$L = \lim_{n\to\infty} \frac{2^n}{3^n/n^2} = \lim_{n\to\infty} \frac{n^2 2^n}{3^n} = 0.$$

Unfortunately, the above limit is difficult to evaluate directly so we first compute $\ln(L)$ as follows.

$$\begin{aligned}
\ln(L) &= \ln\left[\lim_{n\to\infty} \ln\left(\frac{n^2 2^n}{3^n}\right)\right] \\
&= \lim_{n\to\infty} \left[\ln\left(\frac{n^2 2^n}{3^n}\right)\right] \\
&= \lim_{n\to\infty} (2\ln(n) + n\ln(2) - n\ln(3)) \\
&= \lim_{n\to\infty} \left(2\ln(n) + n\ln\left(\frac{2}{3}\right)\right) \\
&= \lim_{n\to\infty} \left(2\ln(n) + n\ln\left(\frac{2}{3}\right)\right) \\
&= -\infty.
\end{aligned}$$

Therefore, $\lim\limits_{n\to\infty} \dfrac{n^2 2^n}{3^n} = 0$ which shows that $3^n/n^2$ grows faster than 2^n. Hence, $2^n \in O(3^n/n^2)$.

(i) If $\log f(n) \in \Theta(\log g(n))$, then $f(n) \in \Theta(g(n))$.

Solution: FALSE. As a counterexample, take $f(n) = n^2$ and $g(n) = n^3$. Then $\log(f(n)) = \log(n^2) = 2\log(n)$ and $\log(g(n)) = \log(n^3) = 3\log(n)$. This means $\log f(n) \in$

$\Theta(\log g(n))$ because they differ only in a constant multiple, however $f(n) \notin \Theta(g(n))$ because n^2 grows slower than n^3.

(j) If $f(n) \in \Theta(g(n))$, then $3^{f(n)} \in \Theta(3^{g(n)})$.

Solution: FALSE. As a counterexample, take $f(n) = n$ and $g(n) = 2n$. Then $3^{f(n)} = 3^n$ and $3^{g(n)} = 3^{2n} = 9^n$. This means $f(n) \in \Theta(g(n))$, however $3^{f(n)} \notin \Theta(3^{g(n)})$ because 3^n grows slower than 9^n.

5. (a) (5 points) Let $f(n)$ and $g(n)$ be positive valued increasing functions of n. Using the definition of Big-O, prove that:

$$f(n) + g(n) \in O(max(f(n), g(n))).$$

Proof: We know that $f(n) \leq max(f(n), g(n))$ and $g(n) \leq max(f(n), g(n))$, since the maximum of two numbers is at least as big as each individually. So by adding these, $f(n) + g(n) \leq 2max(f(n), g(n))$ for all $n > 1$. Thus, $f(n) + g(n) \in O(max(f(n), g(n)))$ by the definition of O, with $C = 2$ and $k = 1$.

(b) (5 points) Let $f(n)$ and $g(n)$ be positive valued increasing functions of n. Using the definition of Big-O, prove that:

$$\text{If } f(n) \in O(g(n)) \text{ and } g(n) \in O(h(n)) \text{ then } f(n) \in O(h(n)).$$

Proof: The definition of $f(n) \in O(g(n))$ says that we can find constants C_1 and k_1 such that

$$|f(n)| \leq C_1|g(n)|$$

whenever $n > k_1$.
Similarly, the definition of $g(n) \in O(h(n))$ says that we can find constants C_2 and k_2 such that

$$|g(n)| \leq C_2|h(n)|$$

whenever $n > k_2$.
Let $C_3 = C_1C_2$ and let $k_3 = \max(k_1, k_2)$.
Since $|f(n)| \leq C_1|g(n)|$ for all $n \geq k_1$ and $|g(n)| \leq C_2|h(n)|$ for all $n \geq k_2$ then certainly $|f(n)| \leq C_1|g(n)|$ and $|g(n)| \leq C_2|h(n)|$ for all $n \geq \max(k_1, k_2)$.
Then by plugging the second inequality into the first inequality, we get that

$$|f(n)| \leq C_1 (C_2|h(n)|) = C_1C_2|h(n)|$$

for all $n \geq \max(k_1, k_2)$. Thus, $|f(n)| \leq C_3|h(n)|$ for all $n \geq k_3$ which implies that $f(n) \in O(h(n))$.

(c) (5 points) Let $a(n)$ and $b(n)$ be functions from the nonnegative integers to the positive real numbers. We say that $a(n) \in \Omega(b(n))$ if there exist positive constants C and k such that $a(n) \geq C \cdot b(n)$ for all $n > k$.
Now let $f(n)$, $g(n)$, and $h(n)$ be functions from the nonnegative integers to the positive real numbers. Prove the following transitive property from the above definition of Ω:

$$\text{If } f(n) \in \Omega(g(n)) \text{ and } g(n) \in \Omega(h(n)) \text{ then } f(n) \in \Omega(h(n)).$$

Solution: Similar to the argument of the previous problem. Alternatively, you can also convert the Big-Ω notation into Big-O and use the transitive property of Big-O.

6. The following algorithm takes as input a list $h_1, h_2, \ldots, h_n$ of n homework scores, where $n \geq 2$. Each homework score is a real number from 0 to 100. The algorithm computes the average of the homework scores, where the lowest score is dropped (not included in the calculation.)

procedure HWAvg($h_1, h_2, \ldots, h_n$: a list of $n \geq 2$ homework scores between 0 and 100)

```
1.  $m := 1$
2.  **for** $i := 2$ to $n$
3.      **if** $h_i < h_m$ **then** $m := i$
4.  $total := 0$
5.  **for** $j := 1$ to $n$
6.      **if** $j \neq m$ **then** $total := total + h_j$
7.  **return** $total/(n-1)$
```

(a) (4 points) What is the worst-case runtime of the HWAvg algorithm in Θ notation? Justify your answer by referring to the pseudocode.

Solution: The algorithm takes time $\Theta(n)$ in the worst case. For any input, the algorithm will do the basic operations in lines 1, 4, and 7 each once. This is time $\Theta(1)$. It also does the constant operation in line 3 exactly $n-1$ times, so the first loop takes time $\Theta(n)$. Similarly, it does the constant operation in line 6 exactly n times, so the second loop takes time $\Theta(n)$. Since the time is the maximum of the parts done in sequence, and we have only done constant and linear operations, the overall time of the algorithm on any input is $\Theta(n)$.

(b) (6 points) Give pseudocode for a different algorithm, SimplerHWAvg, that uses only one loop to solve the same problem, and analyze its worst-case runtime in Θ notation. What effect does using only one loop have on the algorithm's efficiency?

Solution: To solve the problem with only one loop, we can sum the homework scores while finding the minimum, then remove the lowest score from the total before finding the average.

procedure SimplerHWAvg($h_1, h_2, \ldots, h_n$: a list of $n \geq 2$ homework scores between 0 and 100)

```
1.  $m := 1$
2.  $total := h_1$
3.  **for** $i := 2$ to $n$
4.      **if** $h_i < h_m$ **then** $m := i$
5.      $total := total + h_i$
6.  **return** $(total - h_m)/(n-1)$
```

This algorithm takes time $\Theta(n)$ in the worst case. For any input, the algorithm will do the basic operations in lines 1 and 2 and 7, which is time $\Theta(1)$. It also does the constant operations in lines 4 and 5 exactly $n-1$ times, so the for loop takes time $\Theta(n)$, making the overall time of the algorithm on any input $\Theta(n)$. This means that even though the pseudocode for SimplerHWAvg is more succinct than the pseudocode for HWAvg, using only one loop instead of two does not have a notable effect on the algorithm's efficiency.

7. Suppose we are given a list of real numbers, and we want to know whether the list contains any duplicate elements. The following two algorithms solve this problem, returning *true* if some pair of list elements are the same, and *false* if not.

```
procedure MatchExistsA($a_1, a_2, \ldots, a_n$: a list of real numbers with $n \geq 1$)
1.  for $i := 2$ to $n$
2.    for $j := 1$ to $i - 1$
3.      if $a_j == a_i$ then
4.        return true
5.  return false
```

```
procedure MatchExistsB($a_1, a_2, \ldots, a_n$: a list of real numbers with $n \geq 1$)
1.  BubbleSort($a_1, a_2, \ldots, a_n$)
2.  for $i := 1$ to $n - 1$
3.    if $a_i == a_{i+1}$ then
4.        return true
5.  return false
```

(a) (5 points) Describe all worst-case inputs to each algorithm. Find the worst-case running time of each algorithm using Θ notation. Justify your answers by referring to the pseudocode. Which algorithm is more efficient in the worst case?

Solution: The worst case for MatchExistsA occurs if the duplicates are the last two elements of the list, or if there are no duplicates. In these cases, the nested for loops at lines 1 and 2 execute fully, without returning early from line 4. The order of this algorithm in the worst case is $\Theta(n^2)$ because we do constant operations in lines 3 and 4, and these lines run $i-1$ times for each value of i between 2 and n. That is, the total number of times we do these constant operations is $1+2+\cdots+(n-1) = \frac{n(n-1)}{2}$, which is $\Theta(n^2)$.

The worst case for MatchExistsB occurs when the only duplicates are the largest two elements of the list, or if there are no duplicates. In MatchExistsB, the list is sorted using BubbleSort, which takes time $\Theta(n^2)$ in the worst case (Rosen pg. 221, Example 5). After sorting, the list is searched by comparing adjacent pairs of elements. Since this consists of basic operations (lines 3 and 4) inside a loop that runs $n-1$ times (line 2), the running time of lines 2 through 4 is $\Theta(n)$. Since the order of an algorithm comes from the maximum of the parts done in sequence, MatchExistsB is $\Theta(max(n^2, n)) = \Theta(n^2)$.

Therefore, in the worst case, both MatchExistsA and MatchExistsB have running time $\Theta(n^2)$. Hence they are about equally efficient.

(b) (5 points) Describe all best-case inputs to each algorithm. Find the best-case running time of each algorithm using Θ notation. Justify your answers by referring to the pseudocode. Which algorithm is more efficient in the best case?

Solution: The best case for MatchExistsA occurs if the duplicates are the first two elements in the list. In this case, the running time of MatchExistsA is $\Theta(1)$ since it will find the duplicates in the first iteration itself at line 3 when $i = 2$ and $j = 1$, and so it takes a constant amount of time not dependent on the input size n.

The best case for MatchExistsB occurs if the duplicates are the smallest two elements in the list. BubbleSort in line 1 will always occur, and it takes time $\Theta(n^2)$ even in the best case, because it does the same number of comparisons regardless of the input. From the code below (Rosen pg. 197), we can see that the inner loop does $n-i$ comparisons for each value of i between 1 and $n-1$. Therefore, the total number of comparisons for BubbleSort is $(n-1)+(n-2)+\cdots+2+1 = \frac{n(n-1)}{2}$, which is $\Theta(n^2)$. After sorting, the duplicates will be identified in the first iteration at line 3 when $i = 1$, so this part will only take a constant amount of time, $\Theta(1)$ in the best case. Since these parts are done in sequence, the overall time of the algorithm is the maximum of the two, or $\Theta(n^2)$. Since MatchExistsA had a runtime of $\Theta(1)$ in the best case, it is more efficient than MatchExistsB in the best case.

```
procedure BubbleSort($a_1, a_2, \ldots, a_n$: a list of real numbers with $n \geq 2$)
1.  for $i := 1$ to $n - 1$
2.      for $j := 1$ to $n - i$
3.          if $a_j > a_{j+1}$ then interchange $a_j$ and $a_{j+1}$
```