1. Use the definition of “big-O” to prove that $479 n^7 \log(n) + 25n^4 + 54$ is $O(n^7 \log(n))$.

**Solution:** For all $n \geq 2$, $479 n^7 \log(n) + 25n^4 + 54 \leq 479 n^7 \log(n) + 25 n^7 \log(n) + 54 n^7 \log(n) = 558n^7 \log(n)$. Take $n_0 = 2$ and $c = 558$. We have that $479 n^7 \log(n) + 25n^4 + 54 \leq 479 n^7 \log(n) + 25 n^7 \log(n) + 54 n^7 \log(n) \leq cn^7 \log(n)$ for all $n \geq n_0$.

2. Use the definition of “big-O” to prove that $n^{3.5}$ is not $O(n^3)$.

**Solution:** Argue by contradiction. Suppose $n^{3.5}$ is $O(n^3)$. Then $\exists c, n_0$ s.t. $n^{3.5} \leq cn^3$ for all $n \geq n_0$. This implies $n^{0.5} \leq c$ for all $n \geq n_0$. Take $m = 1 + \max\{n_0, e^2\}$. Clearly $m > c$ and $m > n_0$, so we have a contradiction. Therefore $n^{3.5}$ is not $O(n^3)$.

3. You do not have to prove anything for this problem, just give examples, as specified.

(a) Find an example of non-negative functions $d(n)$, $f(n)$, $e(n)$, $g(n)$, such that $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, but $d(n) - e(n)$ is not $O(f(n) - g(n))$.

**Solution:** $d(n) = 2n$, $f(n) = n$, $e(n) = n$, $g(n) = n$.

(b) Find an example of non-negative functions $d(n)$, $f(n)$, $e(n)$, $g(n)$, such that $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$, and $d(n) - e(n)$ is $O(f(n) - g(n))$.

**Solution:** $d(n) = 2n$, $f(n) = n^2$, $e(n) = n$, $g(n) = n$.

4. Order functions

$2n \log(n^2), 5n^6, 2^{2013}, 2.5^n, 2.2^n, 2n^{6.5}, \log(n^{10}), 4 \cdot \log(n), 2^{100}, n^{1.03}, 70n, n \log n, 8n^6 + 5n^2$

by their asymptotic growth rate, in non-decreasing order. Indicate by circling those functions that are “big-Theta” of each other.

**Solution:**

$2^{100}, 2^{2013}, 4 \cdot \log(n), \log(n^{10}), 70n, 2n \log(n^2), n \log n, n^{1.03}, 8n^6 + 5n^2, 5n^6, 2n^{6.5}, 2.2^n, 2.5^n$.

The functions with the same asymptotic growth are underlined.
5. Give the best asymptotic (“big-Oh”) characterization of the best and worst case time complexity of the algorithm \( \text{Count}(A, B, n) \). Explain how you computed the complexity.

\[
\begin{align*}
\text{Algorithm } \text{Count}(A, B, n) \\
\text{Input: } & \text{Arrays } A \text{ and } B \text{ of size } n, \text{ where } n \text{ is even.} \\
& A, B \text{ store integers.} \\
i & \leftarrow 0 \\
sum & \leftarrow 0 \\
\text{while } i < \frac{n}{2} \text{ do} \\
& \quad \text{if } A[i + \frac{n}{2}] < 0 \text{ then} \\
& \quad \quad \text{for } j \leftarrow i + \frac{n}{2} \text{ to } n \text{ do} \\
& \quad \quad \quad \text{sum} \leftarrow \text{sum} + B[j] \\
& \quad i \leftarrow i + 1 \\
\text{return } sum
\end{align*}
\]

**Solution:** I put primitive operations on the right hand side. First column gives the best case, second column the worst case. Best case happens when array A holds only positive integers, in which case the “for” loop is never executed. Adding up the best case we get \( O(n) \) time complexity, and adding up the worst case we get \( O(n^2) \) time complexity.

6. Let \( A \) be an array storing positive integers. Write in pseudocode an algorithm that rearranges the elements of \( A \) so that the odd elements appear before the even elements. For example, if the input to the algorithm is an array \( A = \{4, 5, 2, 9\} \), and example of a valid output is \( A = \{5, 9, 4, 2\} \). Other outputs, for example \( A = \{5, 9, 2, 2\} \), are also valid, as long as the odd elements appear before the even ones. Compute the time complexity of your algorithm in the worst case. Explain how you computed complexity. Most likely, your complexity will be not linear. At the end of the course, we will learn a linear time algorithm for this problem.

**Solution:** The algorithm below has complexity \( O(n^2) \). Note that if the problem allowed extra data structures for storage, a linear time algorithm to do rearrangement is trivial: traverse the array, store odd elements in one linked list, even elements in another linked list, then rewrite the array first with the elements from the odd linked list, then with the elements from the even linked list. Below, I show pseudo-code for a quadratic time algorithm that does not use extra storage. At the end of the course, we will learn a linear time algorithm that rearranges elements without extra storage.
**Algorithm:** Rearrange(A, n)

**Input:** Array A of positive integers, size n

for \( i = 0 \) to \( n-1 \)  
if \( A[i] \) is even  
\( j = i + 1 \)  
while \( j < n \) and \( A[j] \) is even  
\( j = j + 1 \)  
if \( j < n \)  
\( \text{temp} = A[j] \)  
\( A[i] = \text{temp} \)