Six questions; each question is worth 20 points. There are 120 points for graduate students and 100 points for undergraduates. Graduate students need to solve all 6 questions. Undergraduates can avoid Q3 but would get bonus points for solving this problem. Each question is worth 20 points.

1. Time complexity upper and lower bounds are defined as follows:

   (a) **Time complexity upper bound** is a time threshold in which all instances of a problem can be solved.

   (b) **Time complexity lower bound** is a time threshold below which some instances of the problem cannot be solved.

   (c) These notions apply to both problems and algorithms.

Now assume algorithm $A$ solves problem $\Pi$. Also assume that $U(n)$ and $L(n)$ are a time upper bound and a time lower bound for problem $\Pi$, respectively. Let $t_A(n)$ and $s_A(n)$ be the time upper bound and the time lower bound for algorithm $A$, respectively. Are the following relationships necessarily correct? Please justify for your answers.

(a) $U(n) \leq t_A(n)$ ?

(b) $U(n) \geq s_A(n)$ ?

(c) $L(n) \geq t_A(n)$ ?

(d) $L(n) \leq s_A(n)$?
Answers:

Five points for each question. The student needs to explain the reason why he/she chose the answer. Correct reason is more important than the answer.

(a) No.

This is because $U(n)$ is just an upper bound for the problem $\Pi$ and it may be derived from the upper bound of different algorithm solving the same problem. So $U(n)$ could be either $> \text{ or } < t_A(n)$, the upper bound of algorithm $A$. For example, consider the sorting problem. Assume Algorithm $A$ is the INSERTION SORT that has the upper $O(n^2)$. But $U(n)$ could be the upper of another algorithm that runs in time $O(n \log_2 n)$ or time $O(n^3)$.

(b) No.

For the same reason as (a). $U(n)$ could be either $> \text{ or } < s_A(n)$, the lower bound of algorithm $A$. Also consider the Sorting problem and $A$ is the INSERTION SORT that has the lower bound $\Omega(n^2)$.

(c) No.

Actually $L(n) \leq t_A(n)$. That is, any algorithm’s running time to cover all instances cannot be faster than the proved lower bound $L(n)$ for the problem. Consider the problem is the sorting problem that has the lower bound $\Omega(n \log_2 n)$. Then another algorithm solving Sorting should not be faster than $\Omega(n \log_2 n)$.

(d) No.

While $L(n) \leq t_A(n)$, it is possible $L(n) > s_A(n)$, as it is possible that $s_A(n) < t_A(n)$.
2. Using the definition of Big-$O$ to prove or disprove

(a) If $T(n) = O(f(n))$ and $S(n) = O(f(n))$, $T(n) + S(n) = O(f(n))$.

(b) If $T(n) = O(f(n))$, then $T(n)f(n) \neq O(f(n))$.

**Answers:**

10 points for each question. Points are distributed to critical steps in the proof.

(a) Correct

Proof: Because $T(n) = O(f(n))$, by the definition of big-$O$, there must be constants $c_1$ and $k_1$ such that

$$T(n) \leq c_1 f(n) \text{ when } n \geq k_1$$  \hspace{1cm} (1)

Also because $S(n) = O(f(n))$, by the definition of big-$O$, there must be constants $c_2$ and $k_2$ such that

$$S(n) \leq c_2 f(n) \text{ when } n \geq k_2$$  \hspace{1cm} (2)

Combine (1) and (2) and choose $n \geq k = \max\{k_1, k_2\}$,

$$T(n) + S(n) \leq c_1 f(n) + c_2 f(n) = (c_1 + c_2)f(n) = cf(n)$$

where $c$ is defined as $c = c_1 + c_2$.

Therefore, $T(n) + S(n) = O(f(n))$, by the definition of big-$O$.

(b) Incorrect.

Here is a counter example to make $T(n)f(n) = O(f(n))$.

Let $T(n) = 10$ be constant function and $f(n^2)$. Therefore, $T(n) = O(n^2)$. But

$$T(n)f(n) = 10 \times n^2 = 10n^2 = O(n^2)$$

which makes $T(n)f(n) = O(f(n))$ true.
3. (Graduate students only, bonus for undergraduates) Consider the following summation of the harmonic sequence up to the \( n \)th term, for any \( n \geq 1 \),
\[
h(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}
\]

(1) Prove that we have the following recurrence for \( h(n) \). (No math induction is needed in your proof.)
\[
h(n) \leq h(\lfloor n/2 \rfloor) + 1
\]

(2) Use the substitution method to prove that, there are constant \( c > 0, k > 0 \) such that
\[
h(n) \leq c \log_2 n
\]
for all \( n \geq k \).

**Answers:**

10 points for each question. Points are distribute to critical steps in the proof.

(1) Proof. (The student should not use induction to prove.)

\[
h(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}
\]
\[
= (1 + \frac{1}{2} + \cdots + \frac{1}{\lfloor n/2 \rfloor}) + (\frac{1}{\lfloor n/2 \rfloor + 1} + \cdots + \frac{1}{n})
\]
\[
= h(\lfloor n/2 \rfloor) + (\frac{1}{\lfloor n/2 \rfloor + 1} + \cdots + \frac{1}{n})
\]
\[
\leq h(\lfloor n/2 \rfloor) + (\frac{1}{\lfloor n/2 \rfloor + 1} + \cdots + (\frac{1}{\lfloor n/2 \rfloor + 1}))
\]
\[
= h(\lfloor n/2 \rfloor) + \frac{1}{\lfloor n/2 \rfloor + 1} \times \lfloor n/2 \rfloor
\]
\[
\leq h(\lfloor n/2 \rfloor) + \frac{1}{\lfloor n/2 \rfloor + 1} \times (\lfloor n/2 \rfloor + 1)
\]
\[
= h(\lfloor n/2 \rfloor) + 1
\]
So, \( h(n) \leq h(\lfloor n/2 \rfloor) + 1 \)
(2) To use the substitution method to prove \( h(n) \leq c \log_2 n \), for all \( n \geq k \), it is necessary to show the inequality by identifying the constants \( c \) and \( k \).

**Base case:** Because when \( n = 1 \), \( h(n) = 1 \) but \( c \log_2 n = 0 \). To make sure the inequality holds for a base case, we choose the base case to be \( n = 2 \), so

\[
h(2) = 1 + \frac{1}{2} \leq c \log_2 2 = c
\]

But for the inequality to hold, we need \( c \geq 1.5 \).

**Assumption:** \( h(\lfloor n/2 \rfloor) \leq c \log_2 \lfloor n/2 \rfloor \).

**Induction:** We will use the assumption to substitute the right hand side of the known recurrence

\[
h(n) \leq h(\lfloor n/2 \rfloor) + 1
\]

We have

\[
h(n) \leq h(\lfloor n/2 \rfloor) + 1 \\
\leq c \log_2 \lfloor n/2 \rfloor + 1 \\
\leq c \log_2 n/2 + 1 \\
= c \log_2 n - c + 1
\]

because \( c \geq 1.5 \), last term \( \leq c \log_2 n \), i.e., \( h(n) \leq c \log_2 n \), when \( n \geq k = 2 \).
4. Find an upper bound (the big-$O$ notation) for the following recurrence:

$$T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + \frac{n}{2}, \quad T(1) = 1$$

Use the substitution method (i.e., guess and check) to prove your upper bound.

**Answer:**

First we guess $T(n) = O(n \log_2 n)$. (Note: Other upper bound guessed may be okay, such as $O(n^2)$. So this question is mainly to test the student’s ability to use the substitution method to do a complexity proof.)

Then we need to prove $T(n) = O(n \log_2 n)$. That is, we need to show there are constants $c$ and $k$ such that

$$T(n) \leq cn \log_2 n$$

when $n \geq k$.

**Base case:**

Since when $n = 1$, $T(n) = 1$, yet $cn \log_2 n = 0$, we would rather choose $n = 2$ as the base case. In particular, using the recurrence, we have $T(2) = 2T(\frac{2}{2}) + \frac{2}{2} = 2T(1) + 1 = 3$.

To let $T(2) = 3 \leq c2 \log_2 2$, we only need to choose $c \geq 3/2$.

**Assumption:** $T(\lfloor \frac{n}{2} \rfloor) \leq c\lfloor \frac{n}{2} \rfloor \log_2 \lfloor \frac{n}{2} \rfloor$

**Induction:** we substitute $T(\lfloor \frac{n}{2} \rfloor)$ in the recurrence with the assumed upper bound:

$$T(n) = 2T(\lfloor \frac{n}{2} \rfloor) + \frac{n}{2}$$

$$\leq 2c\lfloor \frac{n}{2} \rfloor \log_2 \lfloor \frac{n}{2} \rfloor + \frac{n}{2}$$

$$\leq 2c \frac{n}{2} \log_2 \frac{n}{2} + \frac{n}{2}$$

$$= cn \log_2 n - cn + \frac{n}{2}$$

(5)

because $c \geq 3/2$, term $-cn + \frac{n}{2}$ is negative. So $T(n) \leq cn \log_2 n$, as long as $n \geq k = 2$. 


5. Let function $T(n) = 4T\left(\frac{n}{2}\right) + n$, where $T(1) = 1$. Use the recursive tree method to derive an upper bound for $T(n)$.

**Answer:**

The student needs to draw a recursive tree for $T(n)$. He/she can use either the notation introduced in the textbook or the notation in the lecture note. The critical issue if the student can unfold the recurrence and sum up the times on the tree.

After concluding a complexity upper bound from the recursive tree, the student does NOT need to use substitution method to prove the bound.

Partial credits are given to overall structure of the recursive tree construction; mistakes made in the last few steps of calculation should be a minor issue.

The following solution uses the notation given in the lecture note.

\[
T(n) = 4^k T\left(\frac{n}{2^k}\right) + n \times (1 + \frac{4}{2} + \left(\frac{4}{2}\right)^2 + \left(\frac{4}{2}\right)^3 + \ldots + \left(\frac{4}{2}\right)^k)
= 4^k \times 1 + n \times (1 + 2 + 2^2 + 2^3 + \ldots + 2^k)
= 4^k \log_2 n + n \times (2^{k+1} - 1)
= 2^{2\log_2 n} + n \times (2^{\log_2 n+1} - 1)
= 2^{\log_2 n^2} + n \times 2^{\log_2 n} \times 2 - n
= n^2 + 2n \times n - n
\leq 3n^3
= O(n^2)
\]
6. The **Insertion Sort** algorithm discussed in the class (also in the textbook) can be written as recursive algorithms, by changing either or both of the loops in **Insertion Sort** to recursive calls. This exercise asks you to rewrite **Insertion Sort** to a recursive algorithm, namely **Rec-Insertion Sort**, by replacing only the outer **for** loop with a recursive call.

You would need to use the following recursion scheme for the recursive insertion sort. Let \((A, n)\) be an array with \(n\) elements to be sorted. The algorithm first sorts the prefix \(n-1\) items and then inserts the last item \(A[n]\) into the sorted prefix.

(1) Give such a recursive algorithm for insertion sort.

(2) Someone has suggested that the steps for the inner **while** loop could be done more efficiently since the prefix sublist has already been sorted. For example, one could use a “binary search” to quickly identify where the new item can be inserted, which could be done in \(O(\log_2 n)\) time for each new item. So the total time would be \(O(n \log_2 n)\) instead of \(O(n^2)\). Show that the time complexity would actually remain the same as \(O(n^2)\).

**Answers:**

(1) Roughly the recursive algorithm looks like:

\[
\text{RecInsertionSort}(A, n)
\]

\[
\quad \text{If } n = 1 \text{ return ( )}
\]

\[
\quad \text{else Call RecInsertionSort}(A, n - 1);
\]

\[
\quad i = n - 1;
\]

\[
\quad \text{key} = A[n];
\]

\[
\quad \text{while } i > 0 \text{ and } A[i] > \text{key} \text{ do}
\]

\[
\quad \quad A[i + 1] = A[i]
\]

\[
\quad \quad i = i - 1
\]

\[
\quad \quad A[i + 1] = \text{key}
\]

(2) The proof should argue with the following logic:

(a) Binary search would only gain a speed-up if the data structure is implemented with an array. However, because of using array, the insertion process has to shift the elements downward also, with additional linear time cost in the worst case.

(b) On the other hand, if the data structure is a linked list, it would be difficult to use binary search to gain a speed up.