Homework No. 1  Solution Guides
CSCI 4470/6470 Algorithms, CS@UGA, Spring 2017
Due Tuesday January 24, 2017

Seven questions; each question is worth 20 points. There are 140 points for graduate students and 110 points for undergraduates.

NOTE: Graduate students need to solve all 7 questions. Undergraduates can avoid Q3(1) and Q5 but get bonus points for solving these two. Each question is worth 20 points.

1. 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 worst case time upper bound and the worst case time lower bound for algorithm $A$, respectively. What would you say about the following relations? Explain for each relation.
   (a) $U(n)$ vs $T_A(n)$.
   (b) $U(n)$ vs $S_A(n)$.
   (c) $L(n)$ vs $T_A(n)$.
   (d) $L(n)$ vs $S_A(n)$.

Answers:
(a) It needs to be clear that an upper bound of an algorithm, e.g., $T_A(n)$ for algorithm $A$ here, implies an upper bound for the problem, e.g., $\Pi$ here. But $U(n)$ is just another upper bound (possibly due to another algorithm solving $\Pi$). So $T_A(n)$ could be less than, equal to, or greater than $U(n)$. In other words, there is no relationship between $U(n)$ and $T_A(n)$. 

Examples: Π is the Sorting problem and A is INSERTION SORT so we have $T_A(n) = S_A(n) = O(n^2)$. But $U(n)$ could be either $O(n^4)$, $O(n^2)$, or $O(n \log_2 n)$, which are all upper bound for Sorting.

(b) Though $S_A(n) \leq T_A(n)$ for algorithm A, $S_A(n)$ does not imply a lower bound for the problem Π. So $S_A(n)$ is not comparable with $U(n)$ either. The examples in (a) also explain this situation.

(c) $L(n)$ asserts that regardless of algorithms, problem Π needs at least $L(n)$ amount of time to solve. This implies $T_A(n)$ cannot be less than $L(n)$. That is: $L(n) \leq T_A(n)$.

Examples: for the Sorting problem, we have lower bound proved $L(n) = \Omega(n \log_2 n)$. This means, any algorithm $A$ should have the upper bound $A_T(n) \geq L(n)$.

(d) Again, a lower bound for algorithm $A$ does not imply a lower bound for the problem Π.
2. Using the definition of Big-$O$ to prove

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

(b) If $T(n) = O(f(n))$, it is not necessary that $2^{T(n)} = O(2^{f(n)})$.

**Answers:**

The proofs have to be based on definition of Big-$O$ notation.

(a) Because $T(n) = O(f(n))$ and $S(n) = O(f(n))$, there are constants $c_1, c_2, k_1, k_2$ such that

$$ T(n) \leq c_1 f(n), \text{ when } n > k_1 $$

and

$$ S(n) \leq c_2 f(n), \text{ when } n > k_2 $$

Choose $c = c_1 + c_2$ and $k = \max\{k_1, k_2\}$ then

$$ T(n) + S(n) \leq c_1 f(n) + c_2 f(n) = (c_1 + c_2) f(n) = cf(n) \text{ when } n > k $$

By the definition of Big-$O$, $T(n) + S(n) = O(f(n))$.

(b) It suffices to give an example of $T(n)$ to justify this.

E.g., let $T(n) = 2n$ and $f(n) = n$. Then $2n = O(n)$. But

$$ 2^{2n} \neq O(2^n) $$

because we cannot find $c$ and $k$, such that $2^{2n} = 2^n \times 2^n \leq c2^n$ for all $n > k$. 
3. Consider the following summation of the harmonic sequence up to the 
nth term, for any \( n \geq 1 \),
\[
h(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}
\]

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

(b) 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:

(a) Proof:
\[
h(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \cdots + \frac{1}{n}
= 1 + \frac{1}{2} + \cdots + \frac{1}{\lfloor n/2 \rfloor} + \frac{1}{\lfloor n/2 \rfloor + 1} + \cdots + \frac{1}{n}
= h(\lfloor n/2 \rfloor) + \frac{1}{\lfloor n/2 \rfloor + 1} + \cdots + \frac{1}{n}
\leq h(\lfloor n/2 \rfloor) + \frac{1}{\lfloor n/2 \rfloor + 1} + \cdots + \frac{1}{\lfloor n/2 \rfloor + 1}
= h(\lfloor n/2 \rfloor) + \frac{1}{\lfloor n/2 \rfloor + 1} \times (\lfloor n/2 \rfloor + 1)
= h(\lfloor n/2 \rfloor) + 1
\]

(b) The substitution method is required for the proof. Note that, it would be okay to inequality \( h(n) \leq c \log_2 n + a \), with an additional constant term \( a \) on the right hand side of the inequality. This is fine as long as the substitution method is used to prove such an inequality.
Proof. Assume

\[ h\left(\left\lfloor \frac{n}{2} \right\rfloor \right) \leq c \log_2 \left\lfloor \frac{n}{2} \right\rfloor \]

Then by part (a), we have

\[ h(n) \leq h\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + 1 \leq c \log_2 \left\lfloor \frac{n}{2} \right\rfloor + 1 \leq c \log_2 \frac{n}{2} + 1 = c(\log_2 n - c) + 1 \]

\[ = c \log_2 n - c + 1 \leq \log_2 n \]

as long as \( c \geq 1 \).

Now we show the base case, for \( k = 2 \), for

\[ h(2) = 1 + 1/2 = 1.5 \leq c \log_2 2 \]

to hold, we choose \( c = 2 \).

This proves the claim. That is, there is \( c = 2 \) and \( k = 2 \),

\[ h(n) \leq c \log_2 n \]

when \( n \geq k \)
4. Find an upper bound (the big-$O$ notation) for the following recurrence:

\[ T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + n, \quad T(1) = 2 \]

(a) Use the substitution method (i.e., guess and check) to prove.
(b) Use the recursive tree method to prove the upper bound.

**Answers:**

(a) The substitution method should be used correctly. It needs the basis case, assumption and induction.

\[ T(n) \] is of the linear time complexity but you may guess $n \log_2 n$ or even $n^2$. This is okay. The important thing is to use the substitution method correctly while the guessing of an upper bound is not very critical.

**Guess:** $T(n) \leq cn$ for some $c$ to be determined later.

**Proof:**

Basis: For $T(1) = 2 \leq cn = c \times 1$ to hold, we need $c \geq 2$.

Assume: $T(\lfloor \frac{n}{4} \rfloor) \leq c\lfloor \frac{n}{4} \rfloor$

Substitution:

\[
T(n) = 3T(\lfloor \frac{n}{4} \rfloor) + n \\
\leq 3c\lfloor \frac{n}{4} \rfloor + n \\
\leq 3c\frac{n}{4} + n \\
= \left(\frac{3}{4}c + 1\right)n \\
\leq cn
\]

In order to the last inequality to hold, we need $c \geq 4$. So we have prove

\[ T(n) \leq cn \]

for $c = 4$ and $n \geq 1$.

(b) Make sure the recursive tree is drawn/unfolded correctly and the final summation of time added up correctly.
Proof:

\[ T(n) \]

\[ T(\lfloor \frac{n}{4} \rfloor) \quad T(\lfloor \frac{n}{4} \rfloor) \quad T(\lfloor \frac{n}{4} \rfloor) \]

\[ T(\lfloor \frac{n}{4^2} \rfloor) T(\lfloor \frac{n}{4^2} \rfloor) T(\lfloor \frac{n}{4^2} \rfloor) T(\lfloor \frac{n}{4^2} \rfloor) \quad T(\lfloor \frac{n}{4^3} \rfloor) T(\lfloor \frac{n}{4^3} \rfloor) T(\lfloor \frac{n}{4^3} \rfloor) T(\lfloor \frac{n}{4^3} \rfloor) \]

\[ \ldots \]

\[ T(\lfloor \frac{n}{4^k} \rfloor) \quad \ldots \quad T(\lfloor \frac{n}{4^k} \rfloor) \quad \ldots \quad T(\lfloor \frac{n}{4^k} \rfloor) \]

for some \( k \) such that \( \frac{n}{4^k} = 1 \). That is \( k = \log_4 n \)

So \( T(n) \) is sum of the items in all leaves and all items in the rightmost column. That is

\[ T(n) = 3^k \times T(\lfloor \frac{n}{4^k} \rfloor) + \left( n + \frac{3}{4} n + \left( \frac{3}{4} \right)^2 n + \cdots + \left( \frac{3}{4} \right)^k n \right) \]

\[ = 3^k \times \left( 2 + \frac{3}{4} n + \left( \frac{3}{4} \right)^2 n + \cdots + \left( \frac{3}{4} \right)^k n \right) \]

\[ = 2 \times 3^{\log_4 n} + n(1 + \frac{3}{4} + \left( \frac{3}{4} \right)^2 + \cdots + \left( \frac{3}{4} \right)^k) \]

\[ = 2 \times 3^{\log_4 n} + n \left( \frac{1 - (\frac{3}{4})^{k-1}}{1 - \frac{3}{4}} \right) \]

\[ \leq 2 \times 4^{\log_4 n} + n \left( \frac{1}{4} \right) \]

\[ = 2n + 4n \]

\[ = 6n \]
5. **(Graduate students only, bonus for undergraduates)** Suppose the recurrence in Question 4 is changed to

\[ T(n) = 3T\left(\left\lfloor \frac{n}{4} \right\rfloor + 2\right) + n, \quad T(1) = 2 \]

Prove that the upper bound you have derived for Question 4 still holds here.

**Answer:**

A proof can use the substitution method, or recursive tree. We can also use the following observation:

\[ T(n) = 3T\left(\left\lfloor \frac{n}{4} \right\rfloor + 2\right) + n \leq 3T\left(\left\lfloor \frac{n}{3.5} \right\rfloor \right) + n = 3T\left(\left\lfloor \frac{2n}{7} \right\rfloor \right) + n \]

which makes it easier to use the substitution method to prove an upper bound.

**Guess:** Given \( T(n) \leq 3T\left(\left\lfloor \frac{2n}{7} \right\rfloor \right) + n \), we guess \( T(n) \leq cn \) for some \( c \) to be determined later.

**Proof:**

Basis: For \( T(1) = 2 \leq cn = c \times 1 \) to hold, we need \( c \geq 2 \).

Assume: \( T\left(\left\lfloor \frac{2n}{7} \right\rfloor \right) \leq c\left\lfloor \frac{2n}{7} \right\rfloor \)

Substitution:

\[
T(n) = 3T\left(\left\lfloor \frac{2n}{7} \right\rfloor \right) + n \\
\leq 3c\left\lfloor \frac{2n}{7} \right\rfloor + n \\
\leq 6c \frac{n}{7} + n \\
= \left(\frac{6}{7}c + 1\right)n \\
\leq cn
\]

**In order to the last inequality to hold, we need** \( c \geq 7 \). So we have prove

\[ T(n) \leq cn \]

for \( c = 7 \) and \( n \geq 1 \).
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) Analyze your recursive algorithm: first give a recurrence for the time function of your algorithm and then prove that the time function is bounded by a quadratic function in \(n\).

(3) 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 \(O(n^2)\). Prove or disprove this suggestion.

**Answers:**

(1) (7 points)
Algorithm **Insertion-Sort**\((A)\)

1. for \(j = 2\) to \(\text{length}[A]\) do
2. \hspace{1cm} key = \(A[j]\)
3. \hspace{1cm} \{Insert \(A[j]\) into sorted \(A[1..j - 1]\)\}
4. \hspace{1cm} i = j - 1
5. while \(i > 0\) and \(A[i] > key\)
6. \hspace{1cm} do \(A[i + 1] = A[i]\)
7. \hspace{1cm} i = i - 1
8. \hspace{1cm} \(A[i + 1] = key\)

There are maybe various recursive versions for **Insertion Sort**, but only the outer for loop is needed to be rewritten as a recursion.

The recursion for the for loop can be written as follow:
Rec-Insertion Sort($A, j$)

1. if $j > 1$ call Rec-Insertion Sort($A, j - 1$)
2. same as the rest in Insertion Sort

... Of course, the algorithm needs to be called initially as Rec-Insertion Sort($A, n$)

(2) (10 points)

Let $T(j)$ be the time for Rec-Insertion Sort($A, j$), then

$$T(j) \leq T(j - 1) + cj, \ T(1) = d$$

where time $cn$ is used for the inner while loop.

This recurrence can be proved using the recursive tree method (unfolding) or the substitution method (induction). $T(j) = O(j^2)$. So $T(n) = O(n^2)$.

(3) (3 points)

The argument is not correct. Binary search can be efficient (to identify where the key can be inserted) only when the list is stored in an array (random-access). However, an array data structure makes it hard to shift data in $O(\log_2 n)$ time. One the other hand, a linked list data structure may achieve such efficiency in shifting data, but it does not allow binary search to be employed in $O(\log_2 n)$ time.
7. The HEAPSORT algorithm first builds the maximum heap with subroutine MAX-HEAPIFY, which is called \([n/2]\) times, from the non-leaf node with the highest index to the one with the lowest index (i.e., the root). Would the algorithm still work if the order of these calls to MAX-HEAPIFY is reversed, i.e., from the root to the the non-leaf node with the highest index? Justify your answer.

Answer:

Algorithm HeapSort(A)
1. Build-Max-Heap(A)
2. for \(i = \text{length}[A] \) downto 2
3. exchange \(A[1] \leftrightarrow A[i]\)
4. \(\text{heapsize}[A] = \text{heapsize}[A] - 1\)
5. Max-Heapify(A, 1)

Subroutine Build-Max-Heap(A)
1. \(\text{heapsize}[A] = \text{length}[A]\)
2. for \(i = \left\lfloor \frac{1}{2} \text{length}[A] \right\rfloor \) downto 1 \(\iff \text{here!}\)
3. Max-Heapify(A, i)

Subroutine Max-Heapify(A, i)
1. \(l = \text{LEFT}[i]\)
2. \(r = \text{RIGHT}[i]\)
3. if \((l \leq \text{heapsize}[A]) \text{ and } (A[l] > A[i])\)
4. then \(\text{largest} = l\)
5. \(\text{else} \text{largest} = i\)
6. if \((r \leq \text{heapsize}[A]) \text{ and } (A[r] > A[\text{largest}])\)
7. then \(\text{largest} = r\)
8. if \(\text{largest} \neq i\)
9. then exchange \(A[i] \leftrightarrow A[\text{largest}]\)
10. Max-Heapify(A, \text{largest})

The algorithm would not work if the order of these called to MAX-HEAPIFY is reversed. For this question, note that you should not regard the order to call MAX-HEAPIFY within BUILD-MAX-HEAP as the order of nodes that are treated within MAX-HEAPIFY.

Since MAX-HEAPIFY is top-down, if the root element is handled the first, then it may not avoid the situation of non-max value in the root.
Consider the following list stored in an array organized into a binary complete binary tree: