Homework Assignment No. 1 Answers

CSCI 4470/6470 Algorithms, CS@UGA, Spring 2018

Due Thursday January 25, 2018

Six questions; each question is worth 20 points. There are 120 points for graduate students and 100 points for undergraduates.

1. Using the definition of Big-$O$ to prove
   (a) If $T(n) = 200n \log_2 n + 10n$, then $T(n) = O(n^2)$;
   (b) If $T(n) = \frac{1}{200}n^2$, then $T(n) \neq O(n \log_2 n)$

Answers:

(a) The proof needs to find two constants $c > 0$ and $k > 0$, and prove
   
   $200n \log_2 n + 10n \leq cn^2$, when $n \geq k$

Sample proof:

Because $\log_2 n \leq n$ and $1 \leq n$, we have

\[ 200n \log_2 n + 10n \leq 200n \times n + 10n \times n = 210n^2 \tag{1} \]

So we can choose $c = 210$ and $k = 1$ to make the following hold, when $n \geq k$.

\[ 200n \log_2 n + 10n \leq cn^2 \]

That is, $200n \log_2 n + 10n = O(n^2)$
(b) A typical proof is to show the claim $\frac{1}{200}n^2 = O(n \log_2 n)$ would lead to a contradiction.

Sample proof:

Assume that $\frac{1}{200}n^2 = O(n \log_2 n)$.

By the big-O definition, here are constants $c > 0$ and $k > 0$ such that when $n \geq k$

$$\frac{1}{200}n^2 \leq cn \log_2 n$$

which is equivalent to

$$n \leq 200c \log_2 n$$

But when $n = 2^{20}c^2k = c^2k \times 2^{10}2^{10} > c^2k \times 1,000,000$ because it would lead to

$$c^2k \times 1,000,000 < c^2k2^{20} \leq 200c \log_2 (c^2k2^{20}) = c \times 200 \times (\log_2 c^2k + \log_2 2^{20})$$

$$= c \times 200 \times (\log_2 c^2 + \log_2 k + 20) = c \times 200 \times (2 \log_2 c + \log_2 k + 20)$$

$$\leq c \times 200 \times (2c + k + 20) \leq c \times 200 \times 2ck \times 20 \leq c^2k \times 8,000$$

which leads to

$$c^2k \times 1,000,000 \leq c^2k \times 8,000$$

a contradiction.
2. Using the definition of the Big-O definition to prove
   (a) If $T(n) = O(f(n))$ and $S(n) = O(f(n))$, then $T(n) + S(n) = O(f(n))$.
   (b) If $T(n) = O(f(n))$, it is not necessary that $2^{T(n)} = O(2^n)$.

Answers:

(a) Proof needs to use the big-O definition a number of times.

Proof:
   Since $T(n) = O(f(n))$, there are constant $c_1 > 0, k_1 > 0$ such that
   $$T(n) \leq c_1 f(n), \text{ when } n \geq k_1$$

   Similarly, $S(n) = O(f(n))$, there are constant $c_2 > 0, k_2 > 0$ such that
   $$T(n) \leq c_2 f(n), \text{ when } n \geq k_2$$

So when $n \geq \max\{k_1, k_2\}$
   $$T(n) + S(n) \leq c_1 f(n) + c_2 f(n) = (c_1 + c_2) f(n) = cf(n)$$

here $c$ is defined as $c_1 + c_2 > 0$.
By the definition of Big-O, $T(n) + S(n) = O(f(n))$. 

(b) A proof suffices to show a counter-example the $2^{T(n)} = O(2^{f(n)})$

**Proof.**

Let $f(n) = n$ and $T(n) = 2n$ so they satisfy

$$T(n) = O(f(n))$$

If $2^{T(n)} = O(2^{f(n)})$, then we would have

$$2^{2n} = O(2^n)$$

By the definition of Big-O, there must be constant $c > 0$ and $k > 0$,

$$2^{2n} \leq c2^n \text{ when } n \geq k$$

which leads to

$$2^{2n} = 2^n \times 2^n \leq c \times 2^n$$

But the above inequality is not true for any $n > \log_2 c$
3. Analyze the time upper bound $T(n)$ for the following algorithm BubbleSort, where $n$ represents the number of items in the list $A$.

Algorithm BubbleSort($A$);
1. $n = \text{length}(A)$;
2. $\text{swapped} = \text{true}$;
3. $m = n - 1$;
4. While ($\text{swapped} = \text{true}$) do
5. $\text{swapped} = \text{false}$
6. for $i = 1$ to $m$ do
7. if $A[i-1] > A[i]$ then
8. $\text{swap}(A[i-1], A[i])$
9. $\text{swapped} = \text{true}$
10. $m = m - 1$
11. return (A)

Answers:

It suffices to count total number of steps that the algorithm requires in the worst cases.

Assume the length of $A$ to be $n$ and the time complexity is $T(n)$.

$$T(n) \leq c_1 + c_2 + c_3 + c_4 \times (n+1) + c_5 \times n + c_6 \times n^2 + (c_7 + c_8 + c_9) \times n \times (n - 1) + c_{10} \times n + c_{11}$$

Line 6 is executed at most $c_6 \times n^2$ times because the for loop is executed $\leq n(m + 1)$ times, where $n$ is maximum number of times the body of while loop is executed. And $n(m + 1) \leq n(n - 1 + 1) = n^2$. Arguments for lines 7, 8, 9 are similar.

Thus, for some constants $c, d, e$

$$T(n) \leq c + dn + en^2$$

Thus $T(n) = O(n^2)$. 
4. Binary search on a sorted list of \( n \) numbers has the time complexity \( T(n) \) that satisfies the recurrence

\[
T(n) = T\left(\frac{n}{2}\right) + c
\]

where \( c \geq 1 \) is a constant. Use the substitution method (i.e., induction) to prove that \( T(n) = O(\log_2 n) \).

Answers:

A proof has to use the substitution method (i.e., induction).

\( n \) can be assumed to be a power of 2.

The proof is to show, there are constants \( a > 0, k > 0 \) such that

\[
T(n) \leq a \log_2 n, \text{ when } n \geq k
\]

Sample proof:

(1) **Base case:** we assume \( T(2) = d \) for some constant \( d \), because a binary search uses constant time when the list is of size 2.

   to show
   \[
   T(2) = d \leq a \log_2 n \text{ for } n \geq 2
   \]

   it suffices to choose \( a \geq d \).

(2) **Assumption:**

\[
T\left(\frac{n}{2}\right) \leq a \log_2 \frac{n}{2}
\]

(3) **Induction:** use the recurrence

\[
T(n) = T\left(\frac{n}{2}\right) + c
\]

plug in the assumption to get

\[
T(n) = T\left(\frac{n}{2}\right) + c \leq a \log_2 \frac{n}{2} + c = a \log_2 n - a \log_2 2 + c = a \log_2 n + c - a \leq a \log_2 n
\]

where the last inequality \( \leq \) requires \( a \geq c \).

So we choose \( a = \max\{c, d\} \) and \( k = 2 \), proving the claim.
5. Consider a variant of MERGE SORT that splits an unsorted list of $n$ items into two
sublists of lengths $\frac{n}{3}$ and $\frac{2n}{3}$, recursively sorts the two sublists, respectively, and merges
the sublists into one after they are sorted.

Let $T(n)$ be the time upper bound of this algorithm. Give a recurrence that is satisfied by
$T(n)$ and based on the recurrence to prove that $T(n) = O(n \log_2 n)$ using the recursive
tree method.

Answers:

A proof needs to formulate a recurrence for $T$ and use recursive tree method
to prove $T(n) = O(n \log_2 n)$.

(1) recurrence

$$T(n) = T(\frac{n}{3}) + T(\frac{2n}{3}) + n + c$$

and $T(1) = d$

(where $n + c$ can be replaced with $cn$ or simply $n$.
also the student can assume $n$ is a power of 3).

(2) proof by the recursive tree method

• the tree can be in the format introduced in the lecture note or given in the book;
• the proof should be aware of that the paths to the leaves are not of the same length,
  with the leftmost the shortest (size get reduced to $\frac{1}{3}$) and rightmost the longest
  (size get reduced to $\frac{2}{3}$) at each phase of recursion.

the total steps should be

$$T(n) = n \text{ terms} + c \text{ terms at all layers} + \text{ all } T(1)’s \text{ at all leaves}$$

A simpler strategy to get an upper bound as follows:

(a) Every path is of length bounded by $k = \log_{3/2} n$ assuming $(\frac{2}{3})^k n = 1$.

Thus and there are $k$ layers;

(b) layer $i$ contributes quantity $n + 2^{i-1} c$

(c) the total number of leaves $\leq 2^k$, each contributing the quantity $T(1)$.

(d) 

$$T(n) \leq 2^k T(1) + n \times k + \sum_{i=1}^{k} 2^{i-1} c = 2^k d + n \log_{\frac{3}{2}} n + (2^k - 1)c$$
however, because

\[
2^k = 2^{\log_{1.5} n} = 2^{\frac{\log_2 n}{\log_2 1.5}} = (2^{\log_2 n})^{\frac{1}{\log_2 1.5}} = n^{\frac{1}{\log_2 1.5}} = n^{\log_{1.5} 2} < n^{1.8}
\]

\[
T(n) = O(n^{1.8})
\]

If you can achieve the above, you got 100% credit.

Alternatively, a tighter upper bound \(O(n \log_2 n)\) can be achieved by a careful analysis on the recursive tree. There are two quantities in the previous analysis can be substantially reduced.

(a) contribution of \(c\) terms from each level: The contribution is \(2^{i-1}c\) at layer \(i\) until level \(i > i_0 = \log_3 n\). After that the contribution is \(2/3\) of \(2c\). So the total contribution of \(c\) terms is

\[
\sum_{i=1}^{i_0} 2^{i-1}c + \sum_{i=i_0+1}^{k} \left(\frac{4}{3}\right)^{i-i_0}c = (2^{i_0} - 1)c + \sum_{i=1}^{k-i_0} \left(\frac{4}{3}\right)^i c \leq 2^{\log_3 n}c + 3^{\frac{4}{3} \log_{1.5} n} = O(n)
\]

(b) number of leaves \(T(1)\)'s: is also proportional to the contribution of \(c\)'s. Therefore, \(= n\). One perspective on the number of \(T(1)\)'s is to see that each element belongs eventually to a list of size 1. So there are \(n\) \(T(1)\)'s.

So the total time is, for some \(d\),

\[
T(n) \leq k \times dn = O(n \log_{1.5} n) = O(n \log_2 n)
\]

since \(k = \log_{1.5} n\).

A drawing of the recursive tree will be given in the class.
6. (Graduate students only, bonus for undergraduates) We know that every loop can be replaced with a recursion solution. Rewrite the algorithm Insertion Sort as a recursive algorithm such that no loop statements (such as for and while) appear in the algorithm.

Answers:

- A sample solution:

```c
Rec-InsertionSort(A, i) /* recursively sorted list A[1..i] */
{
  if (i==1)
    return(A);
  else
    Rec-InsertionSort(A, i-1);
    Insert(A, i, A[i]);
    return(A)
}

Insert(A, k, x) /* recursively insert x to sorted list A[1..k-1]
{
  if (k==1) OR (x >= A[k-1])
    A[k] = x;
    return (A)
  else
    A[k] = A[k-1];
    Insert(A, k-1, x);
}
```