CS250 Fall 2003 - Midterm examination

Mathieu Blanchette

October 17th 2003, 13:35-14:30

Your name:

Your McGill ID number:

This is an open-book exam. Calculators and notes are allowed. Use the verso of a page if you need more space to answer. Unless noted otherwise, all logarithms are in base 2. This exam has 7 pages.

1 Running time analysis (20 points)

Indicate the running of each of the algorithms below using the simplest and most accurate big-Oh notation as a function of n. Assume that all arithmetic operations can be done in constant time. You don't need to show your steps or prove anything. The first algorithm is an example.

Algorithm	Running time in big-Oh notation
Algorithm Example(n) $x \leftarrow 0$ **for** $i \leftarrow 1$ **to** n **do** . $x \leftarrow x + 1$	**O(n)**
Algorithm exam1(n) $i \leftarrow 1$ **while** $i < n$ **do** . $i \leftarrow i + 100$	O(n). The number of iterations of the loop is n/100 which is O(n).
Algorithm exam2(n) $x \leftarrow 0$ **for** $i \leftarrow 1$ **to** n **do** . **for** $j \leftarrow 1$ **to** i **do** . $x \leftarrow x + 1$	O(n^2). The inner loop is executed i times, each time taking constant time. So the total running time is $c \cdot (1 + 2 + 3 + ... + n) = cn(n + 1)/2$
Algorithm exam3(n) $i \leftarrow 1$ $k \leftarrow 1$ **while** $i < n$ **do** . $i \leftarrow i + k$. $k \leftarrow k + 1$	$O(\sqrt{n})$. At each iteration, k increases by one. Thus it is a counter for the number of iterations. After k iterations, the value of i is $1 + 2 + 3 + ... + k = k(k + 1)/2$. The loop stops when $i \geq n$, i.e. when $k(k + 1)/2 \geq n$. Solving for k (which is the number of iterations), we get $k = -1/2 + -\sqrt{1/4 + 2n}$ which is $O(\sqrt{n})$.
Algorithm exam4(n) $k \leftarrow 1$ **for** $i \leftarrow 1$ **to** 1000 . **for** $j \leftarrow 1$ to i . $k \leftarrow (k + i - j) * (2 + i + j)$	O(1). The running time is independent of n.

2 (20 points)

(a) (15 points) Indicate, for each pair of functions (f, g) in the table below, whether $f(n)$ is $O(g(n))$, $\Omega(g(n))$ and $\Theta(g(n))$. Write either "yes" or "no" in each box. [Correct answer: 1 point, wrong answer = -0.5 point]. No justifications are necessary.

$\mathbf{f(n)}$	$\mathbf{g(n)}$	$\mathbf{f(n) \in O(g(n))}$	$\mathbf{f(n) \in \Omega(g(n))}$	$\mathbf{f(n) \in \Theta(g(n))}$
$n^{1.01} + n(\log n)^5$	$n^{1.01}$	Yes	Yes	Yes
$n^{1.01} + n(\log n)^5$	$n(\log n)^5$	No	Yes	No
$n^2 + n + 1$	$n \log n$	No	Yes	No
$n^{\log 3}$	$3^{\log n}$	Yes	Yes	Yes
$\log_2(n)$	$\log_3(n^2)$	Yes	Yes	Yes

(b) (5 points) Prove that $\log(n^2 + 10n + 100) \in O(\log(n))$ using only the definition of $O()$.
We need to show that there exist constants c and N such that $\log(n^2 + 10n + 100) \leq c \cdot \log(n)$ for all $n \geq N$. To do so, we need to start from one side of the inequality and derive the other side. Pick $N = 2$, and let's see what c will work. Assuming $n \geq N$, we get:

$$\begin{aligned}
\log(n^2 + 10n + 100) &\leq \log(n^2 + 10n \cdot n + 100 \cdot n \cdot n) \text{ since } n > 1 \\
&= \log(111n^2) \\
&= \log(111) + \log(n^2) \\
&= \log(111) + 2\log(n) \\
&\leq \log(111)\log(n) + 2\log(n) \text{ since } n \geq 2 \\
&= (2 + \log(111)) \cdot \log(n) \\
&= c \cdot \log(n) \text{ for } c = 2 + \log(111)
\end{aligned}$$

Thus, for $N = 2$ and $c = 2 + \log(111)$, we have that $\log(n^2 + 10n + 100) \leq c \cdot \log(n)$ for all $n \geq N$

Notice that many of you seem to think that it was sufficient to show that $\log(n^2+10n+100) \leq c \cdot \log(n)$ for *some* value of n. Writing something like: "Picking $n = 2$ and $c = \log(124)$, we get that $\log(n^2+10n+100) = \log(124) \leq c \cdot \log(n)$" is not a valid proof. Indeed, using this "technique", it is easy to "prove" things that are simply not true.

3 Proofs by induction (15 points)

Let $T(n)$ be defined as follows:

$$T(n) = \begin{cases} 1 \text{ if } n = 1 \\ 3T(\lceil n/2 \rceil) + n \text{ if } n > 1 \end{cases}$$

Use mathematical induction on k to prove that for any $k \geq 0$,

$$T(2^k) = 3^{k+1} - 2^{k+1}$$

Watch your mathematical formalism!

We prove that if $T(n)$ is defined recursively as above, then $T(2^k) = 3^{k+1} - 2^{k+1}$ for any $k \geq 0$. We do so by induction on k.
Base case:
For $k = 0$, we have $T(2^k) = T(1) = 1$ by definition. Indeed, $3^{k+1} - 2^{k+1} = 3^1 - 2^1 = 1$.
Induction step:
Induction hypothesis: Assume that $T(2^b) = 3^{b+1} - 2^{b+1}$ for some $b \geq 0$.
To be proved: This implies that $T(2^{b+1}) = 3^{(b+1)+1} - 2^{(b+1)+1}$
Proof:

$$\begin{aligned} T(2^{b+1}) &= 3T(\lceil (2^{b+1}/2 \rceil) + 2^{b+1} \text{ by definition of T()} \\ &= 3T(2^b) + 2^{b+1} \\ &= 3(3^{b+1} - 2^{b+1}) - 2^{b+1} \text{by I.H.} \\ &= 3^{b+2} + (-3+1)2^{b+1} \\ &= 3^{b+2} - 22^{b+1} \\ &= 3^{b+2} - 2^{b+2} \end{aligned}$$

. Thus, the statement is true for any $k \geq 0$.

4 Analysis of recursive algorithms(15 points)

Consider the following variant of the mergeSort algorithm seen in class.

Algorithm slowMergeSort($A, start, stop$)
Input: An array A of numbers. Indices $start$ and $stop$.
Output: The array $A[start...stop]$ is sorted.
if ($start < stop$) **then**
. $mid \leftarrow start$ /*In the original code, this was $mid \leftarrow \lfloor (start + stop)/2 \rfloor$*/
. slowMergeSort($A, start, mid$)
. slowMergeSort($A, mid + 1, stop$)
. merge($A, start, mid, stop$)

Let $T(n)$ be the number of primitive operations taken by slowMergeSort to sort an array of $n = stop - start + 1$ elements.

a) (5 points) Write a recurrence for $T(n)$, assuming that the number of primitive operations taken by merge($A, start, mid, stop$) is $10 \cdot (stop - start + 1) = 10n$.

Solution: When $n = 1$, the condition ($start < stop$) is checked but fails and then nothing else is done, so we get $T(1) = 1$. When $n > 1$, the number of primitive operations performed are : 1 for "if ($start < stop$)", then 1 for "mid = start", 1 + T(1) for the first recursive call, 1 + 1 + T(n-1) for the second recursive call, 1 + 10 n for the call to merge. Thus we get

$$\begin{aligned} T(n) &= 1 \text{ if n=1} \\ &= T(1) + T(n-1) + 6 + 10n = T(n-1) + 10n + 7 \text{ if } n > 1 \end{aligned}$$

b) (10 points) Guess an explicit formula for $T(n)$. You don't need to prove your result, but show your steps.

$$\begin{aligned} T(n) &= 7 + T(n-1) + 10n \\ &= 7 + (7 + T(n-2) + 10(n-1)) + 10n \\ &= 14 + T(n-2) + 10(n + (n-1)) \\ &... \\ &= (n-1)7 + T(1) + 10\sum_{i=2}^{n} i \\ &= 7n - 7 + 1 + 10(n(n+1)/2 - 1) \\ &= 5n^2 + 12n - 16 \end{aligned}$$

5 Recursive algorithms(20 points)

A king owns a collection of n gold coins, stored in an array $A[0...n-1]$. All gold coins have exactly the same weight except for exactly one that is actually filled with lead and is thus heavier than the normal gold coins. The king asks you to determine quickly what is the index of the lead coin. You own a scale that allows you to compare the total weight of any two regions $A[i...j]$ and $A[k...l]$ of the array. Let us represent this scale by a method compare(A, i, j, k, l):

$$\text{compare}(A,i,j,k,l) \text{ returns } \begin{cases} \text{-1 if } A[i...j] \text{ has a total weight smaller than that of } A[k...l] \\ +1 \text{ if } A[i...j] \text{ has a total weight larger than that of } A[k...l] \\ 0 \text{ if the two weights are equal} \end{cases}$$

Question: Using the *compare* method, write a recursive algorithm that returns the index of the lead coin. Your algorithm should make $O(\log n)$ calls to the compare method (but you don't need to prove this). Your algorithm has to work for any value of $n > 0$.

Algorithm findHeavy$(A, start, stop)$
Input: An array A of coins, one of which is heavier than the others. The heavier coin is guaranteed to be located between indices *start* and *stop* inclusively.
Output: The index of the heavier coin.

/*Write your pseudocode here*/
if $(start = stop)$ **then return** *start*
$n \leftarrow stop - start + 1$
$mid \leftarrow \lfloor (start + stop + 1)/2 \rfloor$
if (n mod $2 = 0$) **then**
. $comp \leftarrow$ compare$(A, start, mid - 1, mid, stop)$
. **if** $(comp = -1)$ **then return** findHeavy$(A, mid, stop)$
. **else return** findHeavy$(A, start, mid - 1)$
else
. $comp \leftarrow$compare$(A, start, mid - 1, mid, stop - 1)$
. **if** $(comp = -1)$ **then return** findHeavy$(A, mid, stop - 1)$
. **if** $(comp = +1)$ **then return** findHeavy$(A, start, mid - 1)$
. **else return** *stop*

6 Stacks (10 points)

The following pseudocode takes as argument an array A and returns true if and only if this array has a certain property.

Algorithm: CheckProperty(A, n)
Input: An array A of n elements
Output: Returns true if A has the property, and false otherwise
Stack $s \leftarrow$ new Stack();
for $i \leftarrow 0$ **to** $\lfloor n/2 \rfloor - 1$ **do**
. s.push(X[i])
for $i \leftarrow \lceil n/2 \rceil$ **to** $n - 1$ **do**
. **if** $(X[i] \neq$ s.top()) **then return** false;
. **else** s.pop()
return true

Question: What is that property?
Answer: The algorithm checks if the input is a palyndrome. That is, it checks if symmetric around its center: A[0]=A[n-1], A[1]=A[n-2],.... In general, A[i] = A[n-1-i] for all i between 0 and $\lfloor n/2 \rfloor$.

7 Bonus (5 points)

Are the statements below true or false? Write a one-sentence justification. To obtain any credit, the justification has to be clear and correct.

a) If an algorithm A_1 runs in time $\Theta(n^2)$ and an algorithm A_2 for the same problem runs in time $\Theta(n \log n)$, then for N large enough, A_2 will be faster than A_1 for *any* input of size at least N.
Answer: FALSE. A_2 will be faster than A_1 only for the worst-case input. There may be inputs on which A_1 is faster than A_2. For example, the insertionSort algorithm has worst-case running time $O(n^2)$ and the mergeSort algorithm has worst-case running time $O(n \log n)$. However, when run on an input that is already sorted, insertionSort will take time $O(n)$ and will be faster than mergeSort.

b) $f(n) \in O(g(n))$ if and only if $\log(f(n)) \in O(\log(g(n)))$.
Answer: FALSE. Choose $f(n) = n^2$ and $g(n) = n$. Then $f(n) \notin O(g(n))$, but $\log(f(n)) = 2log(n) = 2g(n)$ and so $\log(f(n)) \in O(\log(g(n))$.

c) If $f(n) \notin O(g(n))$, then $g(n) \in O(f(n))$.
Answer: FALSE. For example, if $f(n) = n \cdot |sin(n)|$, and $g(n) = n \cdot |cos(n)|$, then $f(n) \notin O(g(n))$ and $g(n) \notin O(f(n))$. That's because there are values of n for which $f(n)$ is about n times larger than $g(n)$, and other values of n for which $g(n)$ is about n times larger than $f(n)$. Notice however that if the functions $f(n)$ and $g(n)$ are monotonically increasing, then the statement above is true.

d) It is always possible to take a recursive algorithm and rewrite it as an iterative algorithm that has the same big-Oh running time.
Answer: TRUE. One can always rewrite a recursive algorithm into a non-recursive algorithm. To do so, use a stack to simulate what happens when a recursive call is made. That is, before doing a recursive call, save the value of all local variables onto the stack. When returning from a method, just restore the values of the local variables by popping them from the stack. The resulting algorithm will have the same complexity as the original recursive algorithm. However, it will be muhc more difficult to understand and debug.

e) When writing a proof by induction, proving the base case is not truly necessary and is usually done only to convince ourselves that the statement is true for at least one value of n. **Answer:** FALSE. The base case is absolutely required for the validity of the proof. Otherwise, one can prove crazy thing like $n = n + 10$ for all $n \geq 0$:
Base case:
Skipped...
Induction step:
Induction hypothesis: Assume that $b = b + 10$ for some $b \geq 0$.
To be proved: This implies that $(b + 1) = (b + 1) + 10$
Proof:

$$\begin{aligned} b+1 &= (b)+1 \\ &= (b+10)+1 \text{by the I.H.} \\ &= (b+1)+10 \end{aligned}$$

.

So we can do the induction step correctly, even though the statement to prove is completely false! Obviously, we would not be able to prove the base case. Conclusion: Without a base case, an induction proof is worth nothing.