CS3230 : Design and Analysis of Algorithms (Fall 2014)

Tutorial Set #3

[For discussion during Week 5]

S-Problems are due: Friday, 05-Sep, before noon.

OUT: 02-Sep-2014 **Tutorials:** Tue & Wed, 09-10 Sep 2014

IMPORTANT: Read "Remarks about Homework" – also applies to tutorials.

Submit solutions to S-Problem(s) by deadline given above.
Prepare your answers to all the D-Problems in every tutorial set.

When preparing to present your answers,

- Think of a CLEAR EXPLANATION
- Illustrate with a good worked example;
- Describe the main ideas,
- Can you sketch why the solution works;
- Give analysis of running time, if appropriate
- Can you think of other (perhaps simpler) solutions?

Helpful Hints Series: Re-Using a Theorem and Software Code-Reuse

A theorem might be hard to prove, but once it is proven, it is always true. So, you can use the theorem to your advantage. There are two ways to re-use it:

(a) *re-use the result* of the theorem, and
(b) *re-use the method* used to prove the theorem.

In problem T3-R2(a), I am asking you to *re-use the method*. Can you see it? Re-using the method is easier for many people – since all you have to do is to just follow the method, and make small adjustments as necessary. In software development, this is like copying a "chunk of code" and using that "chunk of code" somewhere else (with suitable adjustment). Can you see it?

In problem T3-R4(a), I am asking you to *re-use the result*. Can you see that? Re-using the result is not as easy for many people. But, it is much more effective and trains *abstraction thinking*. In the long run, it helps you to think in bigger steps, better see the overall picture (instead of *only seeing* the nitty-gritty details.) In software development, this is like modularizing the "chunk of code" (and maybe adding in appropriate parameters) and calling the module somewhere else (with appropriate parameters). Can you see that this is the preferred was for software code reuse.

Theorem and Good-Quality Software:
Good quality software code is hard to develop, but once developed (written, implemented, and thoroughly tested), it always works. And all good codes are modular by design (with appropriate parameters). So you can re-use the code to your advantage. Two ways to re-use good code:

(a) *re-use the result* – namely call the module (with appropriate parameters), and
(b) *re-use the method* – develop *new module* using similar code (with suitable changes).

Routine Practice Problems *-- do not turn these in -- but make sure you know how to do them.*

R1. [Fun with TELESCOPING]
Solve this recurrence *using telescoping method*: (can you "see" any *telescope*?)
$A(n)= 2A(n/2) + 5n$ for $n > 1$, $A_1 = 5$

R2. (Exercise 7.1-1 on p. 171 of [CLRS]) [Understanding PARTITION in Quicksort]
Using Figure 7.1 as a model, illustrate the operation of PARTITION on the array
$A = [13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11]$.
(Use the *last element* as the pivot element. Of course, you swap it to the front *first*.)

R3. [Analysis of Randomized Quicksort: $(n+1)$ or (n) or $(n-1)$, does it matter?]
In the detailed analysis of Randomized Quicksort given in the lecture notes (slides 35-43), we used the recurrence $T(n) = T(k-1) + T(n-k) + (n+1)$, where the term $(n+1)$ was the time taken to partition the array $A[1..n]$.

(a) Suppose that we insist that the term in the recurrence should be $(n-1)$ instead, since we do $(n-1)$ comparisons in the partitioning subroutine given in the lecture notes (slide 4). Will this affect the result for $T(n) = 1.386\, n \lg n + O(n)$ (slide 43)?
Redo the entire analysis (slides 35-43) with $(n-1)$ instead of $(n+1)$. What is the difference?

(b) If you were to repeat **(a)** but now with the term $(n+1)$ replaced by (n), what do you *think* is the difference. Answer this *without* redoing the entire analysis.

S-Problems: (To do and submit by due date given in page 1)
Solve this S-problem(s) and submit for grading.

IMPORTANT: Write your NAME, Matric No, Tutorial Group in your Answer Sheet.

S1. Increasing Growth Rates [from Spring 2014 Mid-Term Test]

(a) Rank the following functions in *ascending order of growth*; that is, if function $g(n)$ *immediately* follows function $f(n)$ in your list, then it should be the case that $f(n)$ is $O(g(n))$.

$g_1(n) = 2^{(2\lg n)}$ $\quad$ $g_2(n) = n^2(\lg\lg n)$

$g_3(n) = \sum_{k=1}^{n} \sqrt{n} \cdot (3k+2)$ $\quad$ $g_4(n) = 3n \sum_{k=1}^{n} (\lg k)$

Show your steps for the non-trivial cases (such as for g_1, g_3, g_4).

(b) [Finding a Target Pair]
You are given an unsorted array $A[1..n]$ of real numbers and a *target* X. You want to find a *target pair*, namely, two numbers in the array A that adds up to the target X. Mathematically, we want two elements $A[i]$ and $A[j]$, (with $i \neq j$) for which $A[i]+A[j] = X$. Give an algorithm for finding a target pair (if it exists) in $O(n \lg n)$ worst-case time.

D-Problems: Solve these D-problems and prepare to discuss them in tutorial class. You may be called upon to present your solution *or your best attempt at a solution*. Your solution presentation does NOT need to be fully correct, given your best attempt. The TA will help clarify and correct any issues or errors.

D1. [Merging Multiple Lists Efficiently]
You are given k sorted arrays, with n elements each, and you want to suitably combine them into a single sorted array (with kn elements).

(a) One strategy to do this is to use a 2-way merge to merge the first two sorted arrays, then merge the result with the third sorted array, then merge in the fourth, and so on. What is the time complexity of this algorithm, in terms of k and n?

(b) Design a more *efficient* algorithm for solving this problem. What is the time complexity of your algorithm, in terms of k and n?

D2. [Weighing Problem – Contributed by Tan Lee Eng, 1991]

(a) You are given 27 coconuts labeled 1, 2, . . . , 27, and *only* a beam balance. Among these 27 coconuts, 26 of them have exactly the *same* weight, but one of them is *lighter*. We want an algorithm to identify the lighter coconut using *only* the beam balance (which gives three possible outcomes: left-side is heavy, both-sides balanced, right-side is heavy).

(b) Before solving this, how about solving the smaller 3 coconuts problem. Give a simple algorithm for doing this. Specify which coconuts to put onto the beam balance and what to do for each possible outcome of the weighing.

(c) Now, give an algorithm to solve the original 27 coconuts problem, namely to identify the lighter coconut using the least number of weighing with the beam balance.

(d) [**Question:** What, if anything, does this problem have to do with Analysis of Algorithms? Or, why is Prof. Leong asking this weird question in CS3230?] (Short answers please.)

D3. [Teeny Quicksort, modified from problem by Ken Sung, 2013]
A CS3230 student, called Teeny (who is also a teenager) said that she has an algorithm that, given an unsorted array $A[1..n]$, can find a "*teenager pivot*" in time $P(n) = \Theta(5n)$. Using her magic "*teenager pivot*", she can guarantee that *each* of the two partitions will have at least between 13% to 19% of the n elements.

Let $TQ(n)$ be the *worst-case performance* of Ms. Teeny's Quicksort. Give a recurrence relation for $TQ(n)$. Solve for $TQ(n)$ and give it in Θ-notations.

Advanced Problems – Try these for challenge and fun. There is no deadline for A-problems. *Turn in your attempts DIRECTLY to Prof. Leong. Do not combine it with your HW solutions.*)

A5. [Finding a Frequent Element]
Given an array of n numbers $A[1..n]$, we define a *frequent* element to be the element that appears *more than* $\lfloor n/2 \rfloor$ times in the array $A[1..n]$. Design an $O(n)$ algorithm for find the *frequent* element in $A[1..n]$, or determine that it does not exist. Prove that your algorithm is correct and runs in time $O(n)$.