CS3230 : Design and Analysis of Algorithms (Spring 2015)

Tutorial Set #3

(D&C, Randomized Quicksort, Lower-Bound-on-Comparison-based Sorts)

[For discussion during Week 5]

OUT: 05-Feb-2015 **Tutorials:** Mon & Fri, 09 & 13 Feb 2015

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

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-R1, D2, you can *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-R2, R3, you can *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 way 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. Increasing Growth Rates [modified from Spring 2014 Mid-Term Test]
Rank the following functions in *increasing order of growth*; that is, if function $f(n)$ is *immediately* before function $g(n)$ in your list, then it should be the case that $f(n)$ is $O(g(n))$.

$g_1(n) = 2^{(2\lg n)}$ $g_2(n) = 312n^2(\lg\lg n)$

$g_3(n) = \sum_{k=1}^{n} \sqrt{n} \cdot (3k+2)$ $g_4(n) = \sum_{k=1}^{n} 8n(\lg k)$

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

R3. [More practice with Master Theorem] Solve the following recurrence with Master Thm.
(i) $R(n) = 4R(n/16) + 25n^{0.5}(\lg n)^2$
(ii) $S(n) = 9S(n/3) + 5n^3$

R4. [Very Special Case Quicksort] (motivated by a Question to me after lecture)
Consider the very special case of Quicksort, in which we choose a pivot that is guaranteed to be not the smallest and not the largest element. For this very special case, what is the worst-case performance? Is it nearer to $\Theta(n \lg n)$ or nearer to $\Theta(n^2)$.
(*Hint:* What is the worst case split? Form a recurrence for $T(n)$, and then solve the recurrence for the running 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 with 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. (Like Exercise 7.1-1 on p. 171 of [CLRS]) [Understanding PARTITION]
Using the example in lecture notes (L4-Quicksort-Slides-5-16) as a model, illustrate the operation on the array $A[1..12]$ shown below, when you call PARTITION(A, 1, 12).

$A[1..12] = [10, 19, 9, 5, 12, 8, 7, 14, 21, 2, 6, 11]$.

(Use the *first element* as the pivot element.)

Note: You **MUST** use the PARTITION algorithm given in the lecture notes (L4-Quicksort-Slides-3-4). For your reference, it is shown below.

```
1.  Algorithm PARTITON (A, p, q)      // Given A[p..q]
2.  {   x ← A[p];
3.      i ← p;
4.      for j:= p+1 to q
5.          do if (A[j] ≤ x) then
6.                  then {i:= i+1
7.                        exchange A[i] ↔ A[j] }
8.      exchange A[p] ↔ A[i]
9.      return i
10. }
```

D3. [Fordy Quicksort, modified from problem by Ken Sung, 2013]
A CS3230 student, called Fordy said that he has an algorithm that, given an unsorted array $A[1..n]$, can find a "*forties pivot*" in time $R(n) = \Theta(6n)$. Using his magic "*forties pivot*", he can guarantee that *each* of the two partitions will have at least X% of the n elements, where X is a value between 40 to 45.

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

D4. [Alternative Decision Tree]
In the decision tree (for sorting 3 elements) shown in the lectures (Slide L04.3-Lower Bound on Sorting, Slides 5-6), we compare 1:2 (namely, a_1:a_2) at the root of the tree.

Give an alternative decision tree that compares 1:3 (namely, a_1:a_3) at the root node of the tree and also have *height* 3 (meaning that the worst-case time to sort is still 3 comparisons).

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. [How often does the maximum get updated?]
Consider the following simple code for finding the maximum of n numbers $A[1..n]$.

```
1. Algorithm Find-Max (A, n)
2. { Max-sf := –INFTY;
3.   for k:= 1 to n do {
4.     if (A[k] > Max-sf) then
5.       Max-sf := A[k];  endif
6.   } }
```

We already know that the running time is $\Theta(n)$, in fact it takes exactly n comparisons. We are now interested in the question "How many times is *Max-sf* updated in Line 5"?
In the worst-case, the answer is n. In the best-case, *Max-sf* is updated only ONCE.

In general, given a *random* permutation of the n numbers (let's assume the numbers are just $\{1,2,3,\ldots,n\}$), how many times (*on average*) is the variable *Max-sf* updated?

[**HINT:** The answer is ***not*** $n/2$. Can you build a recurrence and solve it?]

A6. [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 $O(n)$.