CS3230 : Design and Analysis of Algorithms (Spring 2015)

Homework Set #2

[Lectures 1 to 4]

OUT: 28-Jan-2015 **DUE:** Friday, 12-Feb-2015, before 12:00noon

IMPORTANT: Write your NAME, Matric No, Tut. Gp (G1-G7) in your Answer Sheet.

Solve all the S-Problems in every homework set.

- Start each of the problems on a *separate* page.
- Make sure your name and matric number is on each sheet.
- Write legibly. If we cannot read what you write, we cannot give points. In case you CANNOT write legibly, please type out your answers and print out hard copy.
- To hand the homework in, **staple them together** and drop them into the CS3230 dropbox outside my office (COM1 03-17) before the due date and time.

IMPORTANT:

You are advised to start on the homework *early*. For some problems, you need to let it incubate in your head before the solution will come to you. Also, start early so that there is time to consult the teaching staff during office hours. Some students might need some pointers regarding writing the proofs, others may need pointers on writing out the algorithm (idea, example, pseudo-code), while others will need to understand *more deeply* the material covered in class before they apply them to solving the homework problems. Use the office hours for these purposes!

It is always a good idea to email the teaching staff before going for office hours or if you need to schedule other timings to meet. Do it in advance.

HOW TO "Give an Algorithm":

When asked to "give an algorithm" to solve a problem, your write-up should *NOT* be just the code. (*This will receive very low marks*.) Instead, it should be a short essay. The first paragraph should summarize the problem and what your results are. The body of the essay should consist of the following:

- A description of the algorithm *in English* and, if helpful, pseudo-code.
- One *worked example or diagram* to show more precisely how your algorithm works.
- A *proof* (or indication) of the correctness of the algorithm
- An *analysis* of the running time of the algorithm.

Remember, your goal is to communicate. Full credit will be given only to correct solutions which are described clearly. Convoluted and obtuse descriptions will receive low marks.

In CS3230, you learn to develop high-level abstractions when describing algorithms. Try not to speak in ML/AL (machine/assembly language) or "for (j=0; j<n; j++) do". Instead give names to your sets (of objects or things or data structures), talk about Depth-First Search, Binary Search, traverse the graph, sort the set, use a priority queue, etc. You are no longer in CS1010, CS1020, CS2010 or CS2020. Speak with greater sophistication, and at a higher level of abstraction.

(**Note:** Each problem is worth 10 marks. So the problems are NOT equally weighted. So, adjust your strategy accordingly.)

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

R1. Using the most appropriate method, prove that
(a) $(\ln n)^5 = O(n^{0.2})$ **(b)** $2^n = O(n!)$
(*Hint:* Find good theorems to use to simplify your work. (Theorems are *magic potions*.))

Standard-Problems -- Solve these problems and turn them in by the due-date.

S1. Modified from Problem 3-3, pp 61-62 of [CLRS] [Sorting out order of growth rates]
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 $O(g(n))$.

$g_1(n) = n!$ $g_2(n) = 5n^2(\lg n)+14$ $g_3(n) = n^{\lg n}$ $g_4(n) = 4n^{0.5}$
$g_5(n) = 2^{2(\lg n)}$ $g_6(n) = 3(\lg n)^2$ $g_7(n) = 2^n$ $g_8(n) = n^2(\lg \lg n)$

To simplify notations, we write $f(n) << g(n)$ to mean $f(n) = o(g(n))$ and $f(n) \equiv g(n)$ to mean $f(n) = \Theta(g(n))$. For example, the four functions n^2, n, $(2013n^2 + n)$ and n^3 could be sorted in increasing order of growth as follows: ($n << n^2 \equiv (2013n^2 + n) << n^3$).

Do not turn in proofs for this problem, but you should do them anyway just for practice.

S2. [Really Solving Recurrences]
(a) Solve for $T(n)$, in Θ-notation, given that $T(n) = 4T(n/2) + \Theta(n^2\lg^3 n)$ for $n>1$, $T(1)=1$, by using the Master Theorem, given in the Lecture Notes.
[*Hint:* Carefully review lecture slides on Master Theorem.]

(b) Solve the recurrence $T(n) = 4T(n/2) + 5n$ for $n>1$, $T(1)=1$ by using the recursion tree expansion method or by telescoping method.

(c) Solve for $U(n)$, given the recurrence $U(n) = \left[\frac{2}{(n-1)}\sum_{k=1}^{n-1} U(k)\right] + 5n$.

S3. (Exercise 8.2-1 on p. 196 of [CLRS]) [Exercising Counting-Sort]
Using Figure 8.2 (or lecture note diagram) as a model, illustrate the operation of COUNTING-SORT on the array $A[1..12] = [3, 2, 1, 7, 1, 7, 2, 5, 1, 5, 7, 4]$. You can assume that the numbers in the array A are integers ranging from 1 to 7. Show your working.

S4. [Finding *the* Missing Integer] (From a past exam/quiz)
You are given an array A[1..*n*] of size *n*, where $n = (2^k - 1)$. You are told that the *unsorted* array *A*[1..*n*] contains all the integers from 0 to *n* (inclusive), *except one*, but you don't know the missing number. For *k*=3, an example is *A*[1..7] = [3, 0, 2, 6, 1, 4, 7]. For this example, the *missing number* is 5.

(a) Give a $\Theta(n)$ algorithm to find the missing integer.
[*Hint:* Use an additional array *B*[0..*n*] if necessary.]

(b) Suppose that you are *now* told that you *cannot directly access* the integers in *A* with a single query operation. (*So, your algorithm from part* (*a*) *don't work now*.)

The elements in *A* are represented in binary and stored as *k* bits, namely,
$(b_{k-1}\ b_{k-2}\ \ldots\ b_1\ b_0)$ where each b_j is a binary bit. (*Note:* $k = \lg n$)

You can make the following query operation:

BIT-QUERY(*j*, *i*) : "fetch the *j*th bit of *A*[*i*]" (in constant $\Theta(1)$ time)

which returns a 0 or a 1 in constant $\Theta(1)$ time.

To illustrate this, we show an example using the array *A*[1..7] given above and we give some sample queries. Remember, the array *A*[1..*n*] is hidden from you (not accessible).

Hidden Array:	**Sample Queries**
A[1] = 0 1 1	BIT-QUERY(0,2) = 0
A[2] = 0 0 0	
A[3] = 0 1 0	BIT-QUERY(1,3) = 1
A[4] = 1 1 0	
A[5] = 0 0 1	BIT-QUERY(2,6) = 1
A[6] = 1 0 0	
A[7] = 1 1 1	

Using this BIT-QUERY operation, and additional storage if needed, describe an algorithm for finding the *missing integer* with $\Theta(n)$ worst-case running time. Also illustrate how your algorithm works on the above example.

[*Note:* If you can use *k* queries to find *all the bit* of *A*[*i*] (for all the *i*), then *this step alone* will take $\Theta(nk) = \Theta(n \lg n)$ time, which is TOO SLOW. We want $\Theta(n)$ worst-case.]