Problem Set 1

This problem set is due **in recitation** on **Friday, February 13**.

Reading: Chapters §1, 2.1-2.3, 3, 4, 28.2, 30.1, Akra-Bazzi Handout

There are **five** problems. Each problem is to be done on a **separate sheet** (or sheets) of paper. Mark the top of each sheet with your name, the course number, the problem number, your recitation section, the date, and the names of any students with whom you collaborated.

You will often be called upon to "give an algorithm" to solve a certain problem. Giving an algorithm entails:

1. A description of the algorithm in English and, if helpful, pseudocode.
2. A proof (or argument) of the correctness of the algorithm.
3. An analysis of the running time of the algorithm.

It is also suggested that you include at least one worked example or diagram to show more precisely how your algorithm works. Remember, your goal is to communicate. Graders will be instructed to take off points for convoluted and obtuse descriptions. If you cannot solve a problem, give a brief summary of any partial results.

Problem 1-1. Asymptotic Notation

Decide whether these statements are **True** or **False**. You must justify all your answers to receive full credit.

(a) $f(n) = \Omega(g(n)) \implies g(n) = O(f(n))$

Solution: True by definition.

(b) $f(n) = \omega(g(n)) \implies f(n) = \Omega(g(n))$

Solution: True by definition.

(c) $f(n) = O(g(n)) \land f(n) = \Omega(h(n)) \implies g(n) = \Theta(h(n))$

Solution: False. $f(n) = n^2 = O(n^3) = \Omega(n)$, but $n \neq \Theta(n^3)$.

(d) $o(g(n)) \cap \omega(g(n)) = \emptyset$

Solution: True. No function can be both asymptotically greater and smaller than $g(n)$.

(e) $f(n) = O(g(n)) \wedge g(n) = \Omega(f(n)) \implies f(n) = \Theta(g(n))$

Solution: False. $n = O(n^2) \wedge n^2 = \Omega(n)$ does not imply that $n = \Theta(n^2)$

Problem 1-2. More Asymptotic Notation

Rank the following functions by increasing order of growth; that is, find an arrangement $g_1, g_2, \ldots, g_{20}$ of the functions satisfying $g_1 = O(g_2)$, $g_2 = O(g_3)$, …, $g_{19} = O(g_{20})$. Partition your list into equivalence classes such that $f(n)$ and $g(n)$ are in the same class if and only if $f(n) = \Theta(g(n))$.

$\binom{n}{2}$	$n \log n$	$\sum_{k=1}^{n} \frac{1}{k}$	$8n^2$	$\log \sqrt{\log n}$
$n!$	$\log \log n$	$n^{\log n}$	$\log n!$	$4^{\log n}$
$\sum_{k=0}^{n} \binom{n}{k}$	$2^{\log^2 n}$	10^{100}	3^n	$\log n$
$(\sqrt{2})^{\log n}$	$(n-1)!$	$3n^3$	2^n	$5\sqrt{n}$

Solution:

The following are ordered asymptotically from smallest to largest, are as follows (two functions, f and g are on the same line if $f(n) = \Theta(g(n))$):

$$10^{100}$$

$$\log \log n \qquad \log \sqrt{\log n}$$

$$\log n \qquad \sum_{k=1}^{n} \frac{1}{k}$$

$$5\sqrt{n} \qquad (\sqrt{2})^{\log n}$$

$$\log n! \qquad n \log n$$

$$8n^2 \qquad 4^{\log n} \qquad \binom{n}{2}$$

$$3n^3$$

$$n^{\log n} \qquad 2^{\log^2 n}$$

$$2^n \qquad \sum_{k=0}^{n} \binom{n}{k}$$

$$3^n$$

$$(n-1)!$$

$$n!$$

Problem 1-3. Recurrence Relations

Solve the following recurrences. Give a Θ bound for each problem. If you are unable to find a Θ bound, provide as tight upper (O or o) and lower (Ω or ω) bounds as you can find. Justify your answers. You may assume that $T(1) = O(1)$.

(a) $T(n) = T(\sqrt{n}) + 1$

Solution: Use a change of variables and iteration. Letting $m = \log n$, we have $T(m) = T(m/2) + c$. By iteration, this solves to $T(m) = \Theta(\log m) = \Theta(\log \log n)$.

(b) $T(n) = 2T(n-1) + 1$

Solution: Use substitution or iteration. Assume $T(n-1) = c(2^{n-1} - 1)$. Plugging this into the recurrence give you: $T(n) = 2c(2^{n-1} - 1) + c = c(2^n - 1) = \Theta(2^n)$.

(c) $T(n) = 2T(n/3) + 1$

Solution: Case 1 of the Master Method: $f(n) = c = O(n^{\log_3 2 - \epsilon})$, so $T(n) = \Theta(n^{\log_3 2})$.

(d) $T(n) = 49T(n/25) + (\sqrt{n})^3 \log n$

Solution: Case 3 of the Master Method: $f(n) = n^{3/2} \log n = \Omega(n^{\log_5 7}$, so $T(n) = \Theta((\sqrt{n})^3 \log n)$.

(e) $T(n) = 9T(n/3) + n^2 \log n$

Solution: Use Recursion Tree, Akra-Bazzi or Generalized Case 2 of Master Method: $f(n) = n^{\log_3 9} \log n = n^2 \log n$ so $T(n) = \Theta(n^2 \log^2 n)$.

(f) $T(n) = 8T(n/2) + n^3$

Solution: Case 2 of the Master Method: $f(n) = \Theta(n^{\log_2 8})$, so $T(n) = \Theta(n^3 \log n)$.

Problem 1-4. Divide and Conquer Multiplication

(a) Show how to multiply two linear polynomials $ax + b$ and $cx + d$ using only three multiplications. (Hint: One of the multiplications is $(a + b) \cdot (c + d)$.)

Solution: Multiply $e = a \cdot c$, $g = b \cdot d$ and $h = (a+b) \cdot (c+d)$. Letting $f = h - e - g$, the product of the two polynomials is is $ex^2 + fx + g$.

(b) Give a divide-and-conquer algorithm for multiplying two polynomials of degree-bound n that runs in time $\Theta(n^{\log 3})$.

Solution: Divide the coefficients of each n-degree polynomial $P(x)$ and $Q(x)$ into a high half and a low half. That is, let $P(x) = x^{n/2} \cdot P^h(x) + P^l(x)$ and $Q(x) = x^{n/2} \cdot Q^h(x) + Q^l(x)$. Use the technique from part (a) to multiply P and Q using three multiplications of their high and low halves. Recursively multiply each half using the same technique. The recursive base case is simply multiplying two polynomials of the form $ax + b$ and $cx + d$.

The results from the recursive calls are combined using addition and subtraction. Assuming that adding coefficients is a $O(1)$-time operation, the work to combine the coefficients from recursion is $O(n)$. The recurrence relation for this algorithm is $T(n) = 3T(n/2) + O(n)$. By case 1 of the Master Method, the runtime of this algorithm is $\Theta(n^{\log 3})$.

(c) Show that two n-bit integers can be multiplied in $O(n^{\log 3})$ steps, where each step operates on at most a constant number of 1-bit values.

Solution: Treat a n-bit integer $b_n \ldots b_1 b_0$ as an n-degree polynomial of the form $b_n x^n + \ldots + b_1 x + b_0$. Then simply use the multiplication procedure of part (b) to multiply two n-bit integers represented as polynomials in $O(n^{\log 3})$ time.

Problem 1-5. Finding a Pair that Sums to x

Give a $\Theta(n \log n)$ algorithm which, given a set S of n real numbers and another real number x, determines whether or not there exists two elements in S whose sum is exactly x.

Solution: Sort S using an $\Theta(n \log n)$-time sorting algorithm. For each element s_i, search the sorted array for $x - s_i$ using an $O(\log n)$ binary search. Since there will be at most n searches, this will take time $\Theta(n \log n)$.

If there exists a pair $s_i + s_j = x$, then on the ith iteration the binary search will find s_j and the algorithm will correctly halt.If no such pair exists, the algorithm will search for every value $x - s_i$, fail to find any, and correctly halt.