Problem Set 1 Solutions

Exercise 1-1. Asymptotic Notation Properties

(a) False. $n = O(n^2)$ but $n^2 \neq O(n)$.

(b) False. Let $f(n) = n$ and $g(n) = n^2$.

(c) True. Since $f(n) = O(f(n))$, there exist constants c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0$. Taking the log of both sides, we have $\lg(f(n)) \leq \lg(cg(n)) = \lg(c) + \lg(g(n)) \leq c' \lg(g(n))$ for a sufficiently large constant c'.

(d) False. Let $f(n) = 2n$ and $g(n) = n$.

(e) False. Let $f(n) = 1/n$. (The statement is true, however, for $f(n) = \Omega(1)$, which covers most functions with which we will be working in this course.)

(f) False. True. Since $f(n) = O(f(n))$, there exist constants c and n_0 such that $f(n) \leq cg(n)$ for $n \geq n_0$. This implies that $g(n) \geq c' f(n)$ for $c' = 1/c$, so $g(n) = \Omega(f(n))$.

(g) False. Let $f(n) = 4^n$.

(h) True. Consider $f(n) + g(n)$ where $g(n) = o(f(n))$ and let c be a constant such that $g(n) < cf(n)$ for large enough n. Then $f(n) \leq f(n) + g(n) \leq (1 + c)f(n)$ for large enough n, so $f(n) + g(n) = O(f(n))$. Further, we have an obvious lower bound of $f(n) + g(n) = \Omega(f(n))$, so $f(n) + o(f(n)) = \Theta(f(n))$.

Exercise 1-2. Iterated Functions

(a) $f_c^*(n) = n$.

(b) $f_c^*(n) = \lg^* n$.

(c) $f_c^*(n) = \lceil \lg n \rceil$.

(d) $f_c^*(n) = \lceil \lg n \rceil$ - 1.

(e) $f_c^*(n) = \lceil \lg \lg n \rceil$.

(f) $f_c^*(n)$ is unbounded.

(g) $f_c^*(n) = \lceil \log_3 \lg n \rceil$.

(h) As an upper bound, $f_c^*(n) = O(\log n)$ since each iteration reduces n by at least a factor of 2 if $n \geq 4$, and $f_c^*(n) \leq 1$ for $2 \leq n < 4$.

Exercise 1-3. Finding Two Numbers Which Sum to x

The following algorithm solves the problem:

1.Sort the elements in S using mergesort.

2. Remove the last element from S. Let y be the value of the removed element.
3. If S is nonempty, look for $z = x - y$ in S using binary search.
4. If S contains such an element z, then STOP, since we have found y and z such that $x = y + z$. Otherwise, repeat Step 2.
5. If S is empty, then no two elements in S sum to x.

Notice that when we consider an element y_i of S during ith iteration, we don't need to look at the elements that have already been considered in previous iterations. Suppose there exists $y_j \in S$, such that $x = y_i + y_j$. If $j < i$, i.e. if y_j has been reached prior to y_i, then we would have found y_i when we were searching for $x - y_j$ during jth iteration and the algorithm would have terminated then.

Step 1 takes $\Theta(n \lg n)$ time. Step 2 takes $O(1)$ time. Step 3 requires at most $\lg n$ time. Steps 2–4 are repeated at most n times. Thus, the total running time of this algorithm is $\Theta(n \lg n)$. We can do a more precise analysis if we notice that Step 3 actually requires $\Theta(\lg(n-i))$ time at ith iteration. However, if we evaluate $\sum_{i=1}^{n-1} \lg(n-i)$, we get $\lg(n-1)!$, which is $\Theta(n \lg n)$. So the total running time is still $\Theta(n \lg n)$.

Problem 1-1. Asymptotic notation

The ranking is based on the following general facts:

- Exponential functions grow faster than polynomial functions, which grow faster than logarithmic functions.
- The base of a logarithm does not matter asymptotically, but the base of an exponential and the degree of a polynomial do matter.
- Stirling's approximation for $n!$ is useful for dealing with factorials asymptotically.

The functions are ranked as follows, listed from left to right by row (there are only 29 entries below since n^2 was listed twice in the problem):

$(\frac{3}{4})^n$	1	$n^{1/\lg n}$	$\lg(\lg^* n)$	$\lg^*(\lg n)$	$2^{\lg^* n}$
$\ln \ln n$	$\sqrt{\lg n}$	$\ln n$	$n^{\log_6 5}$	n	$2^{\lg n}$
$n \lg n$	$\lg(n!)$	n^2	$\sum_{k=1}^n k$	$n^2 + n$	$n^{\lg \lg n}$
$(\lg n)^{\lg n}$	$(\frac{4}{3})^n$	2^n	$n2^n$	e^n	2^{2n}
$n!$	$(n+1)!$	n^n	2^{2^n}	$2^{2^{n+1}}$	

The equivalence classes determined by the Θ relationship are:
$\{1, n^{1/\lg n}\}$, $\{n, 2^{\lg n}\}$, $\{n \lg n, \lg(n!)\}$, $\{n^2, n^2 + n, \sum_{k=1}^n k\}$, and $\{n^{\lg \lg n}, (\lg n)^{\lg n}\}$.

Problem 1-2. Recurrences

(a) $T(n) = 6T(\frac{n}{2}) + n^3$.
By case (3) of the master method, $T(n) = \Theta(n^3)$.

(b) $T(n) = 9T(\frac{n}{3}) + n^2 \lg^3 n$
We show that $T(n) = \Theta(n^2 \log^4 n)$. In general, one may extend case (2) of the master method to say that if $f(n) = \Theta(n^{log_b a} \lg^k n)$, then $T(n) = \Theta(n^{log_b a} \lg^{k+1} n)$ (see Exercise 4.4-1 in the book). We prove the above bound on $T(n)$ using a recursion tree. Assume for simplicity that n is a power of three. The recursion tree will have $\log_3 n + 1$ levels, and the total amount of work on each successive level (starting from the top) will be $n^2 \lg^3(n)$, $n^2 \lg^3(n/3)$, $n^2 \lg^3(n/9)$, $\ldots$, $n^2 \lg^3 9$, $n^2 \lg^3 3$, $n^2 \lg^3 1$. Therefore, summing up all of the work in the tree, we have:

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log_3 n} n^2 \lg^3(n/3^i) \\
&= n^2 \sum_{i=0}^{\log_3 n} \lg^3 3^i \\
&= n^2 \sum_{i=0}^{\log_3 n} i^3 \lg^3 3 \\
&= n^2 \lg^3 3 \sum_{i=0}^{\log_3 n} i^3 \\
&= n^2 \lg^3 3 \cdot \Theta(\log^4 n) \\
&= \Theta(n^2 \log^4 n)
\end{aligned}$$

(c) $T(n) = T(n^{1/3}) + 1$
Make the substitution $m = 2^n$, so $T(m) = T(m/3) + 1$. The solution to this recurrence is $T(m) = \Theta(\log m)$ by case (1) of the master method. Substituting back, we have $T(n) = \Theta(\log \log n)$.

(d) $T(n) = 5T(\frac{n}{3}) + n\sqrt{n}$
By case (3) of the master method, $T(n) = \Theta(n\sqrt{n})$.

(e) $T(n) = T(\frac{n}{3}) + T(\frac{n}{2}) + n$
Using a recursion tree, the amount of work on each successive level (starting at the top), will be n, $(\frac{5}{6})n$, $(\frac{5}{6})^2 n$, $\ldots$. This series is upper-bounded by $n \sum_{i=0}^{\infty} (\frac{5}{6})^i = 6n$, so $T(n) = O(n)$. Notice also that $T(n)$ is lower-bounded by n, so $T(n) = \Omega(n)$. Therefore, $T(n) = \Theta(n)$.

(f) $T(n) = 2T(\frac{n}{8}) + \sqrt[3]{n}$
By case (2) of the master method, $T(n) = \Theta(\sqrt[3]{n} \log n)$.

(g) $T(n) = T(\frac{n}{2} + \lg n) + 1$
Since we are allowed to assume that $T(n) = O(1)$ for sufficiently small n, let's assume that $T(n) = O(1)$ for $n \leq 8$. If $n > 8$, then $\frac{n}{2} + \lg n < \frac{3n}{4}$, and since

$T(n)$ is a monotonically increasing function, we have $T(n) \leq T(\frac{3n}{4}) + 1$. Therefore, $T(n) = O(\log n)$. Additionally, since $T(n) \geq T(\frac{n}{2}) + 1$, we have $T(n) = \Omega(\log n)$. Hence, $T(n) = \Theta(\log n)$.

(h) $T(n) = 2T(n-1) + n^7$

The solution to this recurrence is $T(n) = O(2^n)$. We prove this by induction. Take as our inductive hypothesis $T(n) \leq c2^n - dn^7$ for constants c and d. By choosing c large enough, we can certainly satisfy the initial conditions on T. For our inductive step, we have

$$\begin{aligned}
T(n) &= 2T(n-1) + n^7 \\
&\leq 2 \cdot (c2^{n-1} - d(n-1)^7) + n^7 \\
&\leq 2 \cdot (c2^{n-1} - dn^7 + o(n^7)) + n^7 \\
&\leq c2^n + (1-2d)n^7 + o(n^7) \\
&\leq c2^n - dn^7 \quad \text{for } d \text{ sufficiently large}
\end{aligned}$$

There is an obvious lower bound of $T(n) = \Omega(2^n)$, so we have $T(n) = \Theta(2^n)$.

Problem 1-3. Inversions

(a) The five inversions are $(1,5)$, $(2,5)$, $(3,4)$, $(3,5)$, $(4,5)$.

(b) If an n-element set is sorted in reverse order, then each of the $\binom{n}{2}$ pairs of elements will be an inversion.

(c) Insertion sort runs in $\Theta(n+I)$ time, where I denotes the number of inversion initially present in the array being sorted. To see this, consider the pseudocode for insertion sort:

```
INSERTION-SORT(A)
for j ← 2 to length[A]
      do key ← A[j]
         i ← j − 1
         while i > 0 and A[i] > A[i + 1]
               do A[i + 1] ← A[i]
         A[i + 1] ← key
```

Everything except the while loop requires $\Theta(n)$ time. We now observe that every iteration of the while loop swaps an adjacent pair of out-of-order elements $A[i]$ and $A[i+1]$. This decreases the number of inversions in A by exactly one since $(i, i+1)$ will no longer be an inversion (the other inversions are not affected). Since there is no other means of decreasing the number of inversions of A, we see that the total number of iterations of the while loop over the entire course of the algorithm must be equal to I.

(d) To count the number of inversions in an array A, we modify merge sort so that it counts inversions as it sorts. Let L denote the lower half of the array $A[1 \ldots \lfloor n/2 \rfloor]$

and let R denote the upper half of the array $A[\lfloor n/2 \rfloor + 1 \ldots n]$. By induction, we can assume that our modified merge sort will be able to count the number inversions in L and the number of inversions in R. The only remaining task is that of counting inversions which consist of one element in L and one in R. This is done as follows, during the merge step. Every time we add an element from L to the merged array, we count the number of elements of R with which it will form inversions. More precisely, suppose that we're comparing the ith element of L to the jth element of R, and that $L[i] < R[j]$, so $L[i]$ is the next element to be added to the merged list. In this case, $L[i]$ will be in an inversion with each of the elements in $R[1 \ldots j-1]$, so we add $j-1$ to our running total inversion count. The extra counting doesn't affect the asymptotic running time of merge sort, so we can count the number of inversion of A in $\Theta(n \log n)$ time.

Problem 1-4. Stable Sorting

Take any sorting algorithm and its input array A. We will augment each array element $A[i]$ by storing with it the value of its initial index in the array, i. Let's denote this extra field as $Index[i]$. Whenever the algorithm performs a comparison on two equal array elements $A[i] = A[j]$, we will have it compare instead the values of $Index[i]$ and $Index[j]$. Therefore, if two elements have equal values, the one which started out closer to the beginning of the array will be deemed "less" than the other element. This approach requires $\Theta(n)$ extra storage space and $\Theta(n)$ extra running time, but since any sorting algorithm must spend $\Omega(n)$ time looking at all of its input elements, the extra running time is absorbed and has not impact on the overall asymptotic running time of the algorithm.