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 $\log(f(n)) \leq \log(cg(n)) = \log(c) + \log(g(n)) \leq c' \log(g(n))$ 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) $f_c^*(n) = [\lg n]$.
(d) $f_c^*(n) = [\lg n] - 1$.
(e) $f_c^*(n) = [\lg [\lg n]]$.
(f) $f_c^*(n)$ is unbounded.
(g) $f_c^*(n) = [\log_3 \lg n]$.
(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 $i$th 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 $j$th 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 $i$th 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):

\[
\begin{array}{cccccccc}
(\frac{3}{4})^n & 1 & n^{1/\lg n} & \lg(\lg n) & n & n^{1/\lg n} & \lg(n) & 2^{\lg n} \\
\ln \ln n & \sqrt{\lg n} & \ln n & n \log_2 5 & n & n^{1/\lg n} & \lg(n) & 2^{\lg n} \\
n \lg n & \lg(n!) & n & \sum_{k=1}^{n} k & n^2 + n & n \lg(n) & n \lg \lg n & 2^n \\
(\lg n)^{\lg n} & (\frac{4}{3})^n & 2^n & n^{2^2} & e^n & 2^n & 2^{n+1} & 2^n \\
n! & (n + 1)! & n^n & 2^n & 2^{2n} & 2^{2n+1} & \\
\end{array}
\]

The equivalence classes determined by the $\Theta$ 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 $\{n^{\lg n}, (\lg n)^{\lg n}\}$. 

Problem 1-2. Recurrences

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

(b) \( T(n) = 9T\left(\frac{n}{3}\right) + n^2 \log^3 n \)

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 \( f(n) = \Theta(n^{\log_3 a} \log^k n) \), then \( T(n) = \Theta(n^{\log_3 a} \log^{k+1} n) \) (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 \( n^2 \log^3(n), n^2 \log^3(n/3), n^2 \log^3(n/9), \ldots, n^2 \log^3 9, n^2 \log^3 3, n^2 \log^3 1 \). Therefore, summing up all of the work in the tree, we have:

\[
T(n) = \sum_{i=0}^{\log_3 n} n^2 \log^3(n/3^i) \\
= n^2 \sum_{i=0}^{\log_3 n} \log^3 3^i \\
= n^2 \sum_{i=0}^{\log_3 n} i^3 \log^3 3 \\
= n^2 \log^3 3 \sum_{i=0}^{\log_3 n} i^3 \\
= n^2 \log^3 3 \cdot \Theta(\log^4 n) \\
= \Theta(n^2 \log^4 n)
\]

(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) \( T(n) = 5T\left(\frac{n}{5}\right) + n\sqrt{n} \)

By case (3) of the master method, \( T(n) = \Theta(n\sqrt{n}) \).

(e) \( T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{n}{5}\right) + n \)

Using a recursion tree, the amount of work on each successive level (starting at the top), will be \( n, \left(\frac{5}{6}\right)n, \left(\frac{5}{6}\right)^2 n, \ldots \) This series is upper-bounded by \( n \sum_{i=0}^{\infty} \left(\frac{5}{6}\right)^i = 6n \), 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) \( T(n) = 2T\left(\frac{n}{8}\right) + \sqrt{n} \)

By case (2) of the master method, \( T(n) = \Theta(\sqrt{n} \log n) \).

(g) \( T(n) = T\left(\frac{n}{2}\right) + \lg n + 1 \)

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

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

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)**

1. for $j \leftarrow 2$ to $\text{length}[A]$
2. do $key \leftarrow A[j]$
3. $i \leftarrow j - 1$
4. while $i > 0$ and $A[i] > A[i + 1]$
5. do $A[i + 1] \leftarrow A[i]$
6. $A[i + 1] \leftarrow key$

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 $A[1 \ldots \lceil n/2 \rceil]$
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 of 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 $i$th element of $L$ to the $j$th 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 $\text{Index}[i]$. Whenever the algorithm performs a comparison on two equal array elements $A[i] = A[j]$, we will have it compare instead the values of $\text{Index}[i]$ and $\text{Index}[j]$. 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.