Problem Set 1 Solutions

Problem 1-1. Asymptotic notation

$$
\begin{array}{ccc}
2^{2^{n+1}} & & \\
n^n & & \\
(n+1)! & & \\
3(n!) & & \\
2^{2n} & & \\
e^n & & \\
n \cdot 2^n & & \\
2^n & & \\
(\frac{3}{2})^n & & \\
n^{\lg_8 n} & & \\
n^{2 \lg \lg n} & & \\
(\lg n)^{\lg n} & & \\
(\lg n)! & & \\
n^2 & n^2 + 2^{100} n & \sum_{k=1}^{n} k \\
n \lg n & & \\
\lfloor n \rfloor & 2^{\lg n} & n \\
\ln n & & \\
\sqrt{\lg n} & & \\
\ln \ln n & & \\
2^{\lg^* n} & & \\
\lg^*(\lg n) & & \\
\lg(\lg^* n) & & \\
100^{100} & 1 & (1/n)^{1/\lg n} \\
(\frac{2}{3})^n & &
\end{array}
$$

Problem 1-2. Recurrences

The following are the solutions and justification for the recurrences

(a) $T(n) = 6T(\frac{n}{3}) + n^3$
$T(n) = \Theta(n^3)$ by master method

(b) $T(n) = 6T(\frac{n}{3}) + n$
$T(n) = \Theta(n^{\log_3 6})$ by master method.

(c) $T(n) = 9T(\frac{n}{3}) + n^2$
$T(n) = \Theta(n^2 \log n)$ by master method.

(d) $T(n) = 8T(\frac{n}{2}) + n^3 \lg^2 n$
$T(n) = \Theta(n^3 \lg^3 n)$ by master method (extended) or by actually writing out the recursion tree and summing at every level.

(e) $T(n) = 10T(\frac{n}{3}) + n^2\sqrt{n}$
$T(n) = n^2\sqrt{n}$ by master method.

(f) $T(n) = T(\frac{n}{3}) + 2T(\frac{n}{4}) + n$
$T(n) = \Theta(n)$. Computed by writing out the recursion tree. At each level of the tree we have $n, (\frac{5}{6})n, (\frac{5}{6})^2 n, (\frac{5}{6})^i$ n. Except that some branches end earlier than others. We can upper bound this as

$$n \sum_{i=0}^{\infty} \left(\frac{5}{6}\right)^i \quad = \quad 6n$$

We also have a lower bound of n from the root of the tree. So we can figure out $\Theta(n)$ from this.

(g) $T(n) = T(n^{1/3}) + \lg n$
$T(n) = \Theta(\lg n)$
Let $n = 2^m$, giving $T(2^m) = T(2^{m/3}) + m$. Now rename $S(m) = T(2^m)$, and we have a new recurrence $S(m) = S(m/3) + m$ which is trivially solvable via the master method. So $S(m) = \Theta(m)$.
Going back, we have $T(n) = T(2^m) = S(m) = \Theta(m) = \Theta(\log n)$

(h) $T(n) = 3T(n-1) + n^3$
$T(n) = \Theta(3^n)$
We will prove this via the substitution method. By looking at it, we note that $T(x) > 3T(x-1)$, so at every stage, we are three times bigger. So our guess for a solution will include 3^n. Additionally, we're going to need to handle a n^3 element, so we'll make our guess as $T(n) \leq c3^n - dn^3$.

$$\begin{aligned}
T(n) &= 3T(n-1) + n^3 \\
&\leq 3(c3^{n-1} - d(n-1)^3) + n^3 \\
&= 3(c3^{n-1} - d(n^3 - o(n^3))) + n^3 \\
&= c3^n + (1-3d)n^3 + o(n^3) \\
&\leq c3^n - dn^3 \quad \text{for } d \text{ sufficiently large}
\end{aligned}$$

The boundary condition holds for $c = 1$ and d being some large constant. This proves $T(n) = O(3^n)$

Similarly a lower bound of $T(n) = \Omega(3^n)$ can be demonstrated as above, OR by a simple argument that at $T(n) > 3T(n-1)$, so we multiply whatever we had before by 3. $T(n) = \Omega(3^n)$

(i) $T(n) = T(\lg n) + 1$

$T(n) = \Theta(\lg^* n)$

This is easily seen by looking at what happens on each itration. Assume $T(1) = 1$, this gives us $T(2) = 2$, $T(4) = 3$, $T(16) = 4$, $T(2^{16}) = 5$, $T(2^{2^{16}}) = 6$, etc. This clearly follows a $lg^* n$ behavior, so we try $T(n) = c \lg^* n$

$$\begin{aligned} T(n) &= T(\lg n) + 1 \\ &= c \lg^*(\lg n) + 1 \\ &= c(\lg^* n - 1) + 1 \\ &= c \lg^* n - c + 1 \end{aligned}$$

If we let $c = 1$ the above holds. Base case at T(2) = 1 holds.

(j) $T(n) = T(\frac{n}{4}) + \sqrt{n}$

$T(n) = \sqrt{n}$ by master method

(k) $T(n) = T(\frac{n}{2} + \sqrt{n}) + 1$

$T(n) = \Theta(\lg n)$

We have no useful approach to solving this, so we try approximating and seeing if we can still prove an upper and lower bound. First we note that $T(n)$ is an increasing function. We also note that at $n = 16$, $n/2 + \sqrt{n} = 12$ and $3n/4 = 12$. For all values of $n > 16$, we have $3n/4 > n/2 + \sqrt{n}$. So, let $n_0 = 16$ and now we can try to solve for $T(n) \leq T(3n/4) + 1$. We must keep in mind that we are overshooting and will get an upper bound.

By master's theorem, the solution to the above is $T(n) = O(\lg n)$.

We now repeat, but underestimate the function, and state that $T(n) \geq T(n/2) + 1$. Again, the solution by master method is $\Omega(\lg n)$.

This leads us to conclude that $T(n) = \Theta(\lg n)$.

Problem 1-3. Product Finder

The main idea of this problem is to sort the array A and for each element of the array $A[p]$ search for $A[q] = x/A[p]$ using binary search. In order to print out the indices, we will need to augment that array with additional data that will contain the 'original' position. Given array $A[]$, we build a new array $B[]$ whose elements are pairs $(number, orig_position)$. This is done in time $\Theta(n)$ using the following simple code:

```
for i ← 1 to n
        B[i] ← (A[i], i)
```

This is just a copying routine which adds additional data (no proof necessary, but if one felt like it, the loop invariant would claim that for all $x < i$, $B[x] = (A[x], orig(x))$ where $o(x)$ is the original position of element $A[x]$ in array $A[]$.).

Now the functions are run on B, only looking at the numbers in B.

```
1   mergesort(B)
2   for i ← 1 to n
3           if ((B[i].num)^2 ≤ X) and binarysearch(X/(B[i].num))
4                   output(B[i].pos, binarysearch(X/B[i].num))
```

Note that this code only outputs the pairs once. Since it wasn't specified, it's ok to have $(3, 4)$ and $(4, 3)$ printed, but we chose not to do it.

To prove correctness, we need to prove the algorithm outputs the correct list. First, for each pair $(B[a], B[b])$ that is output has $B[a] * B[b] = X$. Also, each pair will be output only once. This is ensured by the condition $(B[i]^2 \leq X)$. Second, let $B[p]$ and $B[q]$ be two numbers such that $B[p] * B[q] = X$, and such that $B[p] \leq B[q]$. Step 1 permutes the array A. The array A remains unchanged after step 1. Thus, in step 2 there will be an i such that $i = p$. By the correctness of **binarysearch**, it will return $B[q] = X/B[p]$ and the pair $(B[p], B[q])$ will be output.

Running Time: Step 1 takes $\Theta(n \log n)$, and we do n binary searches which take $\Theta(\log n)$ each. Therefore, the total running time is $\Theta(n \log n)$.

Problem 1-4. Coinage

Ok. It turns out that with a simple divide and conquer approach, Long John can find the bad coin in time $\Theta(\log n)$. The basic idea is to compare two halves of the pile and figure out which half the bad coin is in, and repeat until we're down to one coin. Some issue arise when the size of the pile is odd, so we remove one coin.

```
pile ← all the coins
leftover ← empty
while pile has at least one coin
        if odd(pile)
                leftover ← one of the coins from pile
        leftpile ← half the coins in pile
        rightpile ← the other half
         balance(left <> right)
              if (leftpile = rightpile)
                      pile ← empty
```

```
            elseif (leftpile > rightpile)
                pile ← rightpile
            else
                pile ← leftpile
    return(leftover)
```

To prove that this algorithm does what we claim, we need a loop invariant. We will use the following invariant:

Loop Invariant: Either *pile* contains the bad coin, or *leftover* does. If *leftover* contains the coin, *pile* is empty.

Initialization: The loop invariant is clearly true at the start of the loop, since all the coins are in *pile*. This implies that the bad coin is in pile.

Maintenance: We assume that the invariant is true at the start of the loop. The bad coin is therefore either in *leftover* or in *pile*

- If the bad coin is in *leftover* then *pile* is empty (by assumption). The loop terminates immediately (*pile* doesn't have at least one coin). So the invariant is still true.
- If the bad coin is in *pile*, then pile has at least one coin, but the number of coins in pile can be even or odd
  - If the number is even, the coin will be in one of two smaller piles, and we will select it. At the end of the loop, *pile* will still include the bad coin.
  - If the number is odd, the coin will either be moved to *leftover* or be in one of two smaller piles. If it is moved to *leftover*, then the two piles will weigh the same, and at the end of the loop, *pile* will be empty and *leftover* will contain the coin. Otherwise, the coin will be in one of two smaller piles, which we will find and set *pile* to.

In all cases, the loop invariant remains true at the end of the loop.

Termination: The loop terminates when *pile* becomes empty. We know it must terminate because *pile* shrinks on every iteration of the loop, and it can't shrink forever. When it terminates, the loop invariant is still true, AND *pile* is empty. Since *pile* is empty, the invariant tells us that *leftover* must be the coin, and we are done.

Running Time: Now for the running time. In the worst case, we never get lucky (the bad coin is never set aside in the odd scenario) until the very end. Therefore, at each pass through the loop, the piles get half as big. This means that we repeatedly weigh half the previous number until we're down to one coin. This yields a number of weighings = $\Theta(\log n)$.