HW2 Solutions  
CSE 101, Spring 2002

Problem 1

Algorithm:
COUNTFEIT-FINDER(coins)
1  left ← first \([n/3]\) coins
2  right ← next \([n/3]\) coins
3  other ← remainder of coins
4  if left = right = 1
5    then Weigh(left, right)
6      if left rises
7        then return left
8      if right rises
9        then return right
10     else return other
11  Weigh(left, right)
12  if left rises
13    then COUNTFEIT-FINDER(left)
14  if right rises
15    then COUNTFEIT-FINDER(right)
16  else COUNTFEIT-FINDER(other)

Runtime Analysis:

Assuming the cost of partitioning up the coins is constant, the running time is \(T(n) = T([n/3]) + O(1)\). By the Master Method \(T(n) = \lg(n)\). The assumption on the cost of partitioning is reasonable given that you can arbitrarily pick which coins are in which groups.

Proof of Correctness

There are two reductions that can be made in one weighing: reducing to a group of half the size, achieved by dividing into two groups; and reducing to a third of the size, achieved by dividing into two groups. Obviously the second method, which is used by the algorithm is better. But what about dividing into more than three groups? This requires more than one weighing, but would the result be smaller than that of reducing by a factor of three at each weighing? The answer is no, to see this realize that when reducing to more than three groups, you still have to do pairwise comparisons within that group, with the possibility of maybe one three way comparison. However doing repeated pairwise comparisons on groups of the same size, yields poorer results than doing repeated pairwise comparisons on groups of shrinking size.
More formally, if you divide the coins into \( i \) equal sized groups, you will need to do \([i/2]\) comparisons to get a reduced size of \([n/i]\). However, if we did \([i/2]\) three-way comparisons recursively, we would get a reduced size of \([n/3^\text{2}]\). This again is clearly the better alternative.

**Problem 2**

This problem was done in the book, CLRS 66. (Whoops)

**Problem 3**

- \( T(n) = 3T(n/3) + n/2 \)
  
  This can be plugged into the master method:
  
  \[
  f(n) = n/2 = \Theta(n^{\log_33}), \text{ case 2 applies so } T(n) = \Theta(n^{\log_33} \lg n)
  \]

  For the next two recurrences the Master method does not apply. This is because in both problems, the \( f(n) \) is neither polynomially larger or smaller than then \( n^{k_{\log a}} \).

- \( T(n) = 2T(n/2) + \frac{n}{\lg n} \)
  
  The first step is to draw the recurrence tree (see Figure 1,)
  
  from this we can see that each level has a cost of \( \frac{n}{\lg n} \), where \( i \) is the level. Summing over all the levels gives us the cost of the recurrence:

  \[
  \sum_{i=0}^{\lg n-1} \frac{n}{\lg n - i} = n \sum_{i=0}^{\lg n-1} \frac{1}{\lg n - i}, \text{ let } a = \lg n
  \]

  \[
  = 2^a \sum_{i=0}^{a-1} \frac{1}{a - i}
  \]

  \[
  = 2^a \sum_{i=1}^{a} \frac{1}{i} \text{ this is a harmonic series}
  \]

  \[
  = 2^a \lg a \quad \text{ substitute back in fora}
  \]

  \[
  = n \lg \lg n
  \]

  \[
  = \Theta(n \lg \lg n)
  \]

- \( T(n) = 4T(n/2) + 5n^2 \lg \lg n \)

  Again, the first step is to draw the recurrence tree (see Figure 2,.) from this we can see that each level has a cost \( 5n^2 \lg \lg(n/2) \), where \( i \) is the level. Summing over all the levels
gives us the cost of the recurrence:
\[
\sum_{i=0}^{\log_2 n - 1} 5n^2 \log \left( \frac{n}{2i} \right) = \sum_{i=0}^{\log_2 n - 1} 5n^2 \log(n - i) \quad \text{let } a = \log n
\]
\[
= 5(2^a) \sum_{i=0}^{\log_2 n - 1} \log(a - i)
\]
\[
= 5(2^a) \sum_{i=1}^{\log_2 n} \log(i)
\]
\[
= 5(2^a)(\log 1 + \log 2 + \log 3 + \ldots + \log a)
\]
\[
= 5(2^a)(\log(1 \cdot 2 \cdot 3 \cdot \ldots \cdot a))
\]
\[
= 5(2^a) \log(a!)
\]
\[
= 5(2^a) \Theta(a \log a) \quad \text{CLRS 55, or from lecture}
\]
\[
= 5n^2 \Theta(\log n \log \log n)
\]
\[
= \Theta(n^2 \log n \log \log n)
\]

**Problem 4**

The input to this algorithm is an array \(A[1 \ldots n]\), and an integer \(x\).

\textsc{findsum}(\(A, x\))

1. \(S \leftarrow \text{mergesort}(A)\)
2. \(i \leftarrow 1; j \leftarrow n\)
3. \textbf{while} \(i \neq j\)
4. \textbf{do if} \(S[i] + S[j] = x\)
5. \quad \textbf{then return} \(S[i], S[j]\)
6. \quad \textbf{if} \(S[i], S[j] > x\)
7. \quad \textbf{then} \(j \leftarrow j - 1\)
8. \quad \textbf{else} \(i \leftarrow i + 1\)
9. \textbf{return} \text{\textit{nil}}

Solution: We use induction. The induction hypothesis is: We know how to solve the problem for a sorted set of \(n\) elements.

The base case \((n = 2)\) is trivial. For the induction step, assume that \(S\) is sorted in increasing order. Consider \(S[1] \text{ and } S[n]\) and let \(y = S[1] + S[n]\). If \(y = x\), then we are done.

If \(y > x\), then clearly \(S[n]\) cannot be part of the solution because \(S[n] + S[i] > x\) for all \(i\). Therefore, we can eliminate \(S[n]\) from consideration and solve the remaining problem by induction. If \(y < x\), then by similar argument, we can eliminate \(S[1]\).

Note that sorting takes \(O(n \log n)\) time. Thus after the first step, the algorithm will have a runtime of at least \(O(n \log n)\), the while loop then iterates through the sorted array exactly once, leading to an additional cost of \(O(n)\). \(O(n \log n + n) = O(n \log n)\), so we are done.

**Problem 5**

- **Sorting 4 numbers**

For this case, simply running merge-sort will produce a sorted array in 5 comparisons.
Briefly, merge-sort would have two recursive calls of size 2, each of these calls would require one comparison. Merging two sorted arrays of size two would take another 3 comparisons. Total amount of comparisons is 5.

• **Sorting 5 numbers**

The five elements are $a, b, c, d, e$.

We begin this algorithm by finding a partial ordering of the first four elements:

- Compare $a$ with $d$, and $b$ with $c$.[2 comparisons]
- Compare the larger values from the two previous comparisons. Assuming that $a > b$ and $b > c$, we would compare $a$ with $b$. The larger of these two would then completely dominate the two values from the other comparison. This gives us a partial ordering of $a > b > c$ and $a > d$. If the relationships were different, there would still be the same partial ordering, but the arrangement of the elements would differ.[1 comparison]

For simplicity in the next part, let’s assume that the partial ordering is $a > b > c$ and $a > d$.

- Compare $b$ with $e$. Note that we are comparing $e$ with whatever the middle element is of the partial ordering.[1 comparison]
- We now have two cases:
  1. If $e > b$, then compare $e$ with $a$.
  2. If $e < b$, then compare $e$ with $c$

[1 comparison]

- In the worst case we now have a ordered set of four elements, in which $a$ is the largest element. (Why is this the worst case?) This means that we have to insert $d$ into a sorted set of three elements. This step will take two more comparisons.[2 comparisons]

Adding all the []’s we get 7 comparisons. We know that the lower bound for comparison sort is $\lceil \lg n! \rceil$, for 5 elements $\lceil \lg 5! \rceil = \lceil \lg 120 \rceil = 7$. So we know that our method must be optimal since we can do no better.
Figure 1

\[ \begin{align*}
    i=0 & : \frac{n}{\log(n)} \quad \ldots \quad \frac{n}{\log(n/2)} \\
    i=1 & : \frac{n}{2 \log(n/2)} \quad \ldots \quad \frac{n}{2 \log(n/2)} \\
    i=2 & : \frac{n}{4 \log(n/4)} \quad \ldots \quad \frac{n}{4 \log(n/4)} \\
    i=3 & : \frac{n}{8 \log(n/8)} \quad \ldots \quad \frac{n}{8 \log(n/8)} \\
    \end{align*} \]

Figure 2

\[ \begin{align*}
    i=0 & : 5n^2 \log(\log(n)) \\
    i=1 & : 5n^2 \log(\log(n/2)) \\
    i=2 & : 5n^2 \log(\log(n/4)) \\
    i=3 & : 5n^2 \log(\log(n/8)) \\
    i=4 & : 5n^2 \log(\log(n/16)) \\
    \end{align*} \]