CS577: Introduction to Algorithms (Spring 2018)	**Scribe:** Sam Vinitsky
HW1 Solutions	**Date:** 2/8/2018

1a) We find the defective coin via binary search. The main idea is that the testing procedure allows us to check whether an even-size set S of coins contains the defective coin. This is done by partitioning S into two sets, S_1 and S_2, of equal size, and checking whether S_1 and S_2 have the same weight. Their weights are different if and only if S contains the bad coin. We can write this algorithm out explicitly:

```
containsDefectiveCoin(S)
1. Split $S$ into two halves $S_1$ and $S_2$ (each with $n/2$ coins).
2. Weigh $S_1$ against $S_2$.
3. If they weigh the same, return false.
4. Otherwise return true.
```

From here, the rest of the algorithm is a standard binary search: given a collection C of 2^k, $k > 2$, coins, partition C into two sets A and B of equal size. Then test whether the bad coin is in A (using the above procedure); if it is, recurse on A; otherwise, recurse on B.

In the base case of a collection C of 2^2 ($k = 2$) coins, say $\{c_1, c_2, c_3, c_4\}$, we can just check whether $\{c_1, c_2\}$ and whether $\{c_1, c_3\}$ contain the bad coin. Depending on the results of these two tests, we can determine whether the coin is c_1, c_2, c_3, or c_4. We can write this algorithm out explicitly:

```
findDefectiveCoin($C$):
1. Let $n$ denote the number of coins in $C$.
2. If $n = 4$: // base case
   a) Suppose $C = \{c_1, c_2, c_3, c_4\}$
   b) Let $r =$ containsDefectiveCoin($\{c_1, c_2\}$)
   c) Let $r' =$ containsDefectiveCoin($\{c_1, c_3\}$)
      - If $r = r' =$ true, return $c_1$.
      - If $r = r' =$ false, return $c_4$.
      - If $r =$ false and $r' =$ true, return $c_3$.
      - If $r =$ true and $r' =$ false, return $c_2$.
3. Split $C$ into two halves, $A$ and $B$ (each with $n/2$ coins).
4. If containsDefectiveCoin($A$), then return findDefectiveCoin($A$).
5. Else, return findDefectiveCoin($B$).
```

1b) We can obtain the number of tests by solving the recurrence $T(k) = T(k-1)+1$ and $T(2) = 2$. We also provide an inductive proof.

Claim 1 *For all $k = 2, 3, \ldots$, and for all coin-sets C of size 2^k,* Find-Coin(C) *makes k tests.*

Proof: By induction on k:

- Base case: When $k = 2$, we are in the base case, where two tests are made no matter which coin is bad. Since the number of tests equals k, the claim holds.
- Recursive case: When $k > 2$, we fall into the recursive case. In this case, there is one explicit test, in addition to tests made by the recursive call. In the recursive call, the set passed to the recursive call has size $|C|/2 = 2^{k-1}$, and so the inductive hypothesis applies. From this, we conclude that the recursive call makes $k-1$ tests. Consequently, our algorithm performs k tests on input C.

■

1c)

Claim 2 *For all $k = 2, 3, \ldots$ and for all coin-sets C of size 2^k containing exactly one bad coin, our algorithm returns the defective coin in C.*

Proof: By induction on k:

- Base case ($k = 2$): When $k = 2$, we are in the base case, in which there are four coins, c_1, c_2, c_3, c_4. Suppose both $\{c_1, c_2\}$ and $\{c_1, c_3\}$ contain the bad coin. Then it must be c_1. Hence our algorithm will correctly return c_1. The remaining cases similarly follow by direct inspection.
- Recursive case ($k > 2$): When $k > 2$, we fall into the recursive case. Let A, B be the algorithm's partition of C into two halves of equal size. There is exactly one defective coin, c, so it is either in A or in B:
  - Case 1 ($c \in A$): If $c \in A$, containsDefectiveCoin(A) returns true and our algorithm recurses on S. Since A has size $|C|/2 = 2^{k-1}$ and since A contains exactly one defective coin, the assumptions in the inductive hypothesis hold. Therefore the conclusion of the inductive hypothesis holds, and so the recursive call correctly returns c.
  - Case 2 ($c \in B$): If $c \in B$, then containsDefectiveCoin(A) returns false and our algorithm recurses on B. As before, the inductive hypothesis applies, and so the recursive call returns c.

  In both cases, our algorithm returns c, so Claim 2 is proven.

■

2a) Consider calling SlowSort on $A = [4, 3, 2, 1]$: (without lines 6-8 of Gather)

- Step 2 of SlowSort will sort $[4, 3]$ to be $[3, 4]$. Now $A = [3, 4, 2, 1]$
- Step 3 of SlowSort will sort $[2, 1]$ to be $[1, 2]$. Now $A = [3, 4, 1, 2]$
- Step 4 of SlowSort will call Gather on $A = [3, 4, 1, 2]$.
  - Step 9 of Gather will recurse on $A' = [3, 4]$. The recursive call will find $A'[1] < A'[2]$, and thus will leave A' in place. So after step 9, $A = [3, 4, 1, 2]$ still.
  - Step 10 of Gather will recurse on $A' = [1, 2]$. The recursive call will find $A'[1] < A'[2]$, and thus will leave A' in place. So after step 10, $A = [3, 4, 1, 2]$.
  - Step 11 of Gather will recurse on $A' = [4, 1]$. The recursive call will find $A'[1] > A'[2]$. It will then swap 4 and 1. So after step 11, $A = [3, 1, 4, 2]$ (since we swapped 1 and 4).
  - Therefore after Gather runs, we have $A = [3, 1, 4, 2]$
- After SlowSort calls Gather it does nothing to A, which was $A = [3, 1, 4, 2]$.
- Therefore at the end of SlowSort, we have $A = [3, 1, 4, 2]$ which is not sorted!

2b) Consider calling SlowSort on $A = [3, 4, 1, 2]$: (with lines 10 and 11 of Gather swapped)

- Step 2 of SlowSort will sort $[3, 4]$ to be $[3, 4]$. Now $A = [3, 4, 1, 2]$ still.
- Step 3 of SlowSort will sort $[1, 2]$ to be $[1, 2]$. Now $A = [3, 4, 1, 2]$ still.
- Step 4 of SlowSort will call Gather on $A = [3, 4, 1, 2]$.
  - Steps 6-8 will swap 4 and 1, resulting in $A = [3, 1, 4, 2]$.
  - Step 9 of Gather will recurse on $A' = [3, 1]$. The recursive call will find $A'[1] > A'[2]$, and thus it will swap 3 and 1. So after step 9, $A = [1, 3, 4, 2]$.
  - Step 11 of Gather will recurse on $A' = [3, 4]$. The recursive call will find $A'[1] < A'[2]$, and thus it will do nothing. This means that after step 11, $A = [1, 3, 4, 2]$ still. (Step 11 is called before step 10 by assumption)
  - Step 10 of Gather will recurse on $A' = [4, 2]$. The recursive call will find $A'[1] > A'[2]$, and thus it will swap 4 and 2. So after step 10, $A = [1, 3, 2, 4]$.
  - Therefore after Gather runs, we have $A = [1, 3, 2, 4]$
- After SlowSort calls Gather it does nothing to A, which was $A = [1, 3, 2, 4]$.
- Therefore at the end of SlowSort, we have $A = [1, 3, 2, 4]$ which is not sorted!

2c) Looking the structure of SlowSort, we see Gather essentially performs the same role as the Merge subroutine of MergeSort. We will prove the following claim:

Claim 3 *If n is a power of 2, and A is an list such that $A[1 \dots \frac{n}{2}]$ and $A[\frac{n}{2}+1 \dots n]$ are both sorted, then* GATHER(A) *will return a sorted list.*

Proof: We proceed by induction of $k = \log_2 n$.

- Base Case: If $n = 2$, we have two cases:
  - $A[1] < A[2]$: we do nothing, and output A. This is correct, since A was already sorted.
  - $A[1] > A[2]$: we swap $A[1]$ and $A[2]$ in line 3. Now A is sorted.
- Inductive Hypothesis: Given an list A length $n = 2^{k-1}$, with $A[1 \dots \frac{n}{4}]$ and $A[\frac{n}{4} + 1 \dots \frac{n}{2}]$ already sorted, GATHER(A) outputs A in sorted order.
- Inductive Step: Consider running GATHER(A) for some list A of length $n = 2^k$. We will consider the "quarters" of A:

$$\begin{aligned} Q_1 &= A[1 \dots \frac{n}{4}] \\ Q_2 &= A[\frac{n}{4} + 1 \dots \frac{n}{2}] \\ Q_3 &= A[\frac{n}{2} + 1 \dots \frac{3n}{4}] \\ Q_4 &= A[\frac{3n}{4} + 1 \dots n] \end{aligned}$$

  Let $L_1 \circ L_2$ denote the concatenation of the lists L_1 and L_2. Using our new notation, we can write $A = Q_1 \circ Q_2 \circ Q_3 \circ Q_4$. Note that each of the quarters is sorted.

  Consider the execution of GATHER on a list $A = Q_1 \circ Q_2 \circ Q_3 \circ Q_4$ of length n. After steps 6-8 , the list will be $Q_1 \circ Q_3 \circ Q_2 \circ Q_4$. Step 9 recurses on $Q_1 \circ Q_3$, which it sorts by the inductive hypothesis (since this is a list with $\frac{n}{2} = 2^{k-1}$ elements, and Q_1 and Q_3 are sorted). Step 10 recurses on $Q_2 \circ Q_4$, which it sorts by the inductive hypothesis (since this is a list with $\frac{n}{2} = 2^{k-1}$ elements, and Q_2 and Q_4 are sorted). Therefore after steps 9 and 10, the list will be $Q_1' \circ Q_2' \circ Q_3' \circ Q_4'$.

  Observe that $Q_1' \circ Q_2'$ is a sorting of $Q_1 \circ Q_3$. Then Q_1' contains the smallest $\frac{n}{4}$ elements of A in sorted order, since all of these elements had to be in Q_1 or Q_3. Likewise, Q_4' contains the largest $\frac{n}{4}$ elements of A in sorted order. This means that that the middle $\frac{n}{2}$ elements of A (by size) are in $Q_2' \circ Q_3'$.

  Step 11 recurses on $Q_2' \circ Q_3'$, which it sorts by the inductive hypothesis (since Q_2' and Q_3' are each sorted). After step 11, the list will be $Q_1' \circ Q_2'' \circ Q_3'' \circ Q_4$. Note that $Q_2'' \circ Q_3''$ is a sorting of the middle $\frac{n}{2}$ elements of A. Now Q_1' contains the smallest $\frac{n}{4}$ elements in sorted order, $Q_2'' \circ Q_4''$ contains the middle $\frac{n}{4}$ elements in sorted order, and Q_4' contains the largest $\frac{n}{4}$ elements in sorted order. Therefore $Q_1' \circ Q_2'' \circ Q_3'' \circ Q_4$ is sorted. Thus we have a sorted list at the end of GATHER.

■

We can now use this claim to prove (by induction) the correctness of SLOWSORT as follows:

Claim 4 *If n is a power of 2, then* SLOWSORT$(A[1 \ldots n])$ *returns A in sorted order.*

Proof: We proceed by induction of $k = \log_2 n$.

- Base Case: If $n = 1$, then SLOWSORT just returns A, which is by definition in sorted order.
- Inductive Hypothesis: Given an list A of length $n = 2^{k-1}$, SLOWSORT(A) outputs A in sorted order.
- Inductive Step: Consider the execution of SLOWSORT on some list A of length n. By the inductive hypothesis, the recursive calls in lines 2 and 3 return $A[1 \ldots \frac{n}{2}]$ and $A[\frac{n}{2}+1 \ldots n]$ in sorted order. This means that the input to GATHER is a list of length n with both the first and second halves sorted internally. By Claim 3, we know that in this case GATHER(A) will return A in sorted order. But SLOWSORT(A) just outputs the result of GATHER(A), and thus also outputs A in sorted order for all lists of length n. Thus by induction, SLOWSORT(A) sorts A for lists of any length.

■

2d)

Let $T(n)$ denote the number of comparisons made by GATHER on an list of size n. In the base case of $n = 2$, we do one comparison. Therefore $T(2) = 1$. If $n > 2$, then we recursively call GATHER three times on inputs of size $n/2$. Therefore $T(n) = 3 \cdot T(n/2)$. Summarizing, we have:

$$\begin{aligned} T(n) &= 3 \cdot T(n/2) \quad \text{if } n > 2 \\ T(2) &= 1 \end{aligned}$$

Let $S(n)$ denote the number of comparisons made by SLOWSORT on list of size n. In the base case of $n = 1$, SLOWSORT does no comparisons. Therefore $S(1) = 0$. If $n > 1$, then SLOWSORT recursively calls itself twice on inputs of size $n/2$ (this is $2 \cdot S(n/2)$) comparisons), and then calls GATHER on an list of size n (this is $T(n)$ comparisons). Therefore $S(n) = 2 \cdot S(n/2) + T(n)$. Summarizing this, we have:

$$\begin{aligned} S(n) &= 2 \cdot S(n/2) + T(n) \\ S(1) &= 0 \end{aligned}$$

We now solve both recurrences, starting with $T(n)$. By unrolling the recurrence, we get:

$$\begin{aligned} T(n) &= 3 \cdot T(n/2) \\ T(n) &= 3 \cdot (3 \cdot T(n/4)) && = 3^2 \cdot T(n/4) \\ T(n) &= 3^2 \cdot (3 \cdot T(n/8)) && = 3^3 \cdot T(n/8) \\ &\cdots \\ T(n) &= 3^i \cdot T(n/2^i) \end{aligned}$$

where the last line comes from generalizing the pattern (each time we unroll, we multiply the right-hand-side by 3 and divide n by 2). We stop when we reach the base case of 2, meaning we need to find a k such that $n/2^k = 2$. This occurs when $n = 2^k \cdot 2 = 2^{k+1}$, so $k = \log n - 1$. Then

$$\begin{aligned} T(n) &= 3^{\log n - 1} \cdot T(2) \\ &= \frac{1}{3} \cdot 3^{\log n} \\ &= \frac{1}{3} \cdot n^{\log 3} \end{aligned}$$

Thus $T(n) \in O(n^{\log 3})$. Note that $\log 3 \approx 1.58$.

We can now solve the recurrence for $S(n)$. Recall that

$$\begin{aligned} &S(n) = 2 \cdot S(n/2) + T(n) \\ &S(1) = 0 \end{aligned}$$

By unrolling the recursion we get, for every $i \in \{0, 1, 2, \ldots, \log n\}$

$$S(n) = 2^i \cdot S(n/2^i) + \big(\ T(n) + 2T(n/2) + \cdots + 2^{i-1}T(n/2^{i-1})\ \big)$$

in particular, setting $i = \log n$ gives

$$\begin{aligned} S(n) &= 2^{\log n} S(n/2^{\log n}) + \Big(\ T(n) + 2T(n/2) + \cdots + 2^{(\log n)-1} T(n/2^{(\log n)-1})\ \Big) \\ &= nS(1) + \sum_{j=0}^{(\log n)-1} \frac{1}{3} \cdot \left(\frac{n}{2^j}\right)^{\log 3} \cdot 2^j \\ &= 0 + \frac{1}{3} \cdot n^{\log 3} \cdot \sum_{j=0}^{(\log n)-1} \left(\frac{2}{2^{\log 3}}\right)^j \end{aligned}$$

where the last sum is just a geometric series with ratio $r = \frac{2}{2^{\log 3}}$. This means the sum is bounded above by some constant (in particular, $\frac{1}{1-(2/2^{\log 3})} = 3$). Therefore, the total number of comparisons made by SlowSort is in $O(n^{\log 3})$, just like what we found for Gather.