Identification of a Counterfeit Coin

Kurt Teichman
Department of Information and Computer Sciences
University of Hawaii at Manoa
Honolulu, Hawaii, U.S.A.
xxxxxxx@hawaii.edu

Abstract

This paper analyzes a technique used in finding a distinct counterfeit coin in a set of coins of the same denomination. If counterfeit the coin exists, this coin is either heavier or lighter than any of the other coins in the set. The set of coins can contain any permutation of coins while the set must contain at least 3 coins. The algorithm described in this paper presents an algorithm for a general \( n \) coin problem with \( n \geq 3 \). The means to evaluate the problem is accomplished by counting how many times a weighing function must be used to find the counterfeit coin. In this paper, we present an algorithm with the worst case time complexity is \( \Theta(\log n) \). This is achieved with a space complexity of \( \Theta(\log n) \).

1. Introduction

There are many variations of the counterfeit coin problem. Each of these variations is unique and involves known constraints of the counterfeit coin. These constraints can be described as follows:

- The coin is heavier than the rest of the coins in the set
- The coin is lighter than the rest of the coins in the set.

The counterfeit coin problem is often seen in examples of decision trees. One common version is the 8 coin problem [1]. Suppose there are seven coins and one counterfeit coin. The counterfeit coin weighs less than the other seven. How many weighings are necessary by use of a balance scale to determine which of the eight coins is the counterfeit one? One method to solving this is as follows:

- Partition the set of coins into sets to allow for the weighing of multiple coins at a time. In this case, the first partition should create 2 sets with 3 coins in each.
- Depending on the scale output, the user finds which partition has the counterfeit coin. It will either be in one of the 2 sets of 3 coins each, or it is either in the remaining set of 2 coins.
The partition is then split and weighed.

• The side of the scale that rises contains the counterfeit coin.

• This procedure is repeated until the user is left with 2 coins and thus one weighting to determine which was lighter.

Assuming the counterfeit item is the 8th element in the coin set, it would take at least 2 weightings to uncover the item by use of this method. As it can be seen, the sequence of weighings is a 3-ary tree. There are at least eight leaves in the decision three because there are 8 possible outcomes. Each possible outcome must be represented by at least 1 leaf. The largest number of weighings needed to determine the counterfeit coin is the height of the decision tree. To have a more thorough idea of how 2 came to be, one should know this corollary:

**Corollary 1**

If an $m$-ary tree of height $h$ has $l$ leaves, then $h \leq \lceil \log_m l \rceil$. If the $m$-ary tree is full and balanced, then $h = \lceil \log_m l \rceil$ [1].

Now it can be seen that $\lceil \log_3 8 \rceil = 2$. It should be noted that there are other methods that can be used to solve this puzzle, but this example gives the reader basic concepts to understanding the generalized implementation of divide-and-conquer used in this paper.

For this particular case considered by this paper, the coin constraint is unknown as it could be either of heavier or lighter than the other coins. This coin may not even exist in the set. Because of these rules, the simplest case requires the user to have at least 3 coins. With 3 coins, it requires 2 weightings of the balance scale to determine authenticity.

In this paper, we present a general algorithm for the identification of a possible counterfeit coin in a set of $n$ coins. In section 3, the algorithm’s key idea is explained followed by its pseudocode and a proof of its correctness.

## 2. Preliminaries

Before explaining our algorithm description and analysis, there are some preliminary remarks that must be made.

### 2.1. Notations

• *Typewriter* font is used to indicate an ADT except upon first mention.
2.2. Abstract Data Type

We have an abstract data type \textit{coinset} of which there will be one main instance, which is named \textit{main}. An instance of \textit{coinset} can have any finite number of coins in it (we describe the ADT \textit{coin} next). Instances of \textit{coinset} will also have the property of \textit{weight} (i.e. \texttt{main.weight} which denotes the sum weight of all the coins it contains. It should be noted that on each procedure call, \texttt{main} is partitioned into subsets of itself. These will be defined as other \textit{coinsets} denoted by \texttt{R}, \texttt{L}, and \texttt{O} (\texttt{R, L, O} \subseteq \texttt{main}). Each of these subsets contains elements unique in respect to the others. In this paper, it is assumed that terms partitions and \textit{coinsets} can be used interchangeably. It is also assumed that these particular partitions are assigned the respective reference range of a subset in \textit{main}.

A \textit{coin} is another ADT of which there are \( n \) instances. It is assumed that all coins in this problem have identical property fields with the exception of \textit{coin.weight}. The counterfeit \textit{coin}, if it exists, has greater or lesser weight than the rest instances of \textit{coin}. All \textit{coin} instances are contained in \textit{main}.

2.3. Basic Concepts

Finding a counterfeit \textit{coin} is essentially a recursive algorithm that functions much like a recursive binary search [?]. By definition, binary search is a divide-and-conquer algorithm. With this in mind, finding a counterfeit \textit{coin} can be broken down into the following steps:

- Partition \textit{main} into 3 equal subsets of equal size, \texttt{L, R, O}, and remainder \textit{coin}(s) (if any)
- Compare the weights of the subsets to find which contains the counterfeit \textit{coin}
- If one of the subsets is of different weight than the other two, if possible, divide this subset into 3 equal subsets and repeat the same procedure.
- If it is not possible to make 3 equal subsets from the subset (i.e. subset contains 2 instances of \textit{coin}), then find a reference \textit{coin} from the \textit{main} to compare the 2 instances of the \textit{coin} with.
- If all of the partitions are of the same weight, check the remainder \textit{coin}(s) with a reference \textit{coin} from one of the 3 partitioned subsets.
- If found, return the counterfeit \textit{coin}. Otherwise return that there there exists no counterfeit \textit{coin} in \textit{main}.
2.4. Methods

The main procedure for this paper is \texttt{FindCoin(var main:coinset; first,last:integer):coin;}. This procedure returns, if found, a counterfeit \texttt{coin}. The procedure returns \texttt{null} if no counterfeit \texttt{coin} is found.

The weightings of the \texttt{coin} sets are done in constant time by simple comparison of their weight fields. Each weighting comparison can be assumed to be done in constant time as well.

To handle the possible final case of $n = 2$, let there also be a method \texttt{findreference(var main:coinset; first,mid:integer): integer;} which finds an index of an existing \texttt{coin} in \texttt{main} that is neither \texttt{first} nor \texttt{last}. This \texttt{coin} is assigned to \texttt{0}. \texttt{findreference} that, in theory, should accomplish this function in constant time.

To handle remainder \texttt{coins} after partitioning \texttt{main}, let there be a method \texttt{weighremaindercoins(var main:coinset; last,r:integer): coin;} that executes at most two weightings as there can be at most two remainder \texttt{coins}. This pseudocode arbitrarily chooses the index of the last known \texttt{coin} authentic \texttt{coin} to weigh against the remainder \texttt{coin}(s) from \texttt{main}. If found, this method returns the counterfeit \texttt{coin} or else it returns \texttt{null}. It should be noted that this method is only called just before the termination of \texttt{FindCoin}. Thus, the possible 2 weightings are considered as a constant.

2.5. Definitions

\textbf{Lemma 1:}

If \texttt{FindCoin(main, first, last)} is called with the problem size $n = (last - first + 1)$ for all $n \geq 3$, then it will return some coin $c_i$ ($first \leq i \leq last$) upon deduction of $c_i$ being counterfeit or \texttt{null} if no counterfeit \texttt{coin} is found.

\textbf{Theorem 1:}

Let $f$ be an increasing function that satisfies the recurrence relation

$$f(n) = af(n/b) + c$$

whenever $n$ is divisible by $b$, where $a \geq 1$, $b$ is an integer greater than 1, and $c$ is a positive real number. Then

$$f(n) \text{ is } \begin{cases} O(n^{\log_a a}) & \text{if } a > 1 \\ O(\log n) & \text{if } a = 1 \end{cases}$$
Furthermore, when \( n = b^k \), where \( k \) is a positive integer,

\[
f(n) = C_1 n \log_b a + C_2,
\]

where \( C_1 = f(1) - c(a - 1) \) and \( C_2 = -c(a - 1) \).

A proof of Theorem 1 is given in Section 7.3 (pp.477-478) of [2].

3. Algorithm Design

3.1. Key Idea

The key idea of our algorithm is to find a counterfeit coin with the least number of uses of the weighing function. To be more specific, the key idea is to reduce the number of coins needed to be checked and weighed by a factor of 3 with each recursive call. It should be noted that the weighing function’s use is assumed whenever the weights of two coinsets are compared.

The first step is comparing the weight of \( L \) with \( R \). If these sets are not equal in weight, then \( L \) is compared with \( O \). If they are equal in weight, then \( R \) contains the counterfeit. However, if \( L \) and \( O \) are not equal in weight, then \( L \) contains the counterfeit. Thus, regardless of the outcome of the weighing of \( L \) and \( R \), \( O \) is always weighed against \( L \) to give final determination of the coinset in question. As a result of this, there are at least 2 weighings per recursive call. The coinset containing the counterfeit coin has its first and last indices passed in with main as arguments for the next recursive call of the FindCoin procedure.

If \( n \) is not a multiple of 3, remainder coin(s) exist that must be checked if \( L \), \( R \), and \( O \) are equal to each other. Each of these checks requires a reference coin to be selected. In the following pseudocode, the coin used is typically \( c_{last} \).

It should be noted that there is a possible case of having a partition of 2 coins being compared in the final recursive call. Since there is no way to determine which of the coins is counterfeit unless there are 3 or more coins, we also need a reference coin to allow for a deductive standard when comparing sets. Any coin in main, other than the 2 coins in question, can be used as a reference coin.

3.2. Pseudocode

```plaintext
1 procedure FindCoin(var main:coinset; first, last:integer): coin;
```
// Input:
// A reference to Coinset main containing coins;
// The integer assigned to a first index of the coinset
// The integer assigned to a last index of the coinset
// Assumptions:
// (1) coinset main contains more than 2 coins
// (2) first ≥ 1
// (3) last ≤ n
// Output:
// Finds a counterfeit coin \( c_i \) (1 ≤ i ≤ n) out of n coins, if it exists
// If the counterfeit coin does not exist, null is returned
// Initial Assertion: A counterfeit coin may exist
{
    int n ← last - first + 1;
    int p ← \lfloor \frac{n}{3} \rfloor - 1;
    int r;
    // Check for case where n = 2 and assign r accordingly
    if (n ≠ 2) then
        r ← n (mod 3)
    else
        r ← 0;
    int mid ← first + p + 1;
    int last ← last - r;
    coin c ← null;
    coinset L ← \{main.c_j | first ≤ j ≤ first + p\};
    coinset R ← \{main.c_k | mid ≤ k ≤ mid + p\};
    coinset O;
    // Check for case where n = 2 and assign O accordingly
    if (n ≠ 2) then
        O ← \{main.c_l | mid + p + 1 ≤ l ≤ last\}
    else
O ← \{main.c_l = findreference(main, first, mid)\};

// Base Case:
if (n ≤ 5) {
    if (L.weight ≠ R.weight) then { // Weigh(L,R)
        if (L.weight ≠ O.weight) then // Weigh(L,O)
            return L.c_j
        else
            return R.c_k
    }

    else if (L.weight ≠ O.weight) then { // Weigh(L,O)
        return O.c_l
    }

    else if (r ≠ 0) then {
        // If all three coinsets are equal in weight, compare remainder
        // coin(s) with the coin of the last index (2 ≥ weightings)
        return weighremaindercoins(main, last, r)
    }
}

// General Case:
if (L.weight ≠ R.weight) then { // Weigh(L,R)
    if (L.weight ≠ O.weight) then // Weigh(L,O)
        c ← FindCoin(main, first, first+p)
    else
        c ← FindCoin(main, mid, mid+p)
}

else if (L.weight ≠ O.weight) then { // Weigh(L,O)
    c ← FindCoin(main, mid+p+1, last)
}

else if (r ≠ 0) then {
    // If all three coinsets are equal in weight, compare remainder
    // coin(s) with the coin of the last index (2 ≥ weightings)
    c ← weighremaindercoins(main, last, r);
}

// Final Assertion: A counterfeit coin does or does not exist
return c
3.3. Correctness Proof

We apply the method of induction to prove the correctness of the recursive algorithm and the upholding of Lemma 1. As an assumption of this counterfeit coin problem, main contains at least 3 instances of coin. The base case of this algorithm, however, occurs when \( n \leq 5 \). To be more specific, the base cases of \( \{ n \mid 3 \leq n \leq 5 \} \) must be evaluated. Since \( n \leq 5 \), line 36 is true and there always is an output: a counterfeit coin or null if there does not exist a counterfeit. Thus Lemma 1 is satisfied and the post condition is true.

We shall expand on one more case before doing the inductive step, and that is when \( n = 6 \). Note that line 36 is now false and we proceed with the general case. If the 3 instances of coinset are of equal size, since \( c \) is already initialized to null at the start, null is returned. However, if a counterfeit coin is detected, FindCoin invokes a recursive call to find a counterfeit coin on a partition of size 2. This call triggers the assignment of \( O \) to a set containing a reference coin on line 34. Since line 36 is now true, a base case will be evaluated as normal with \( L \), \( R \), and \( O \) assigned to one coin a piece. Since we have already proved that the base case holds, Lemma 1 is satisfied and the post condition is true.

Now, for \( n > 5 \), assume that FindCoin satisfies Lemma 1 on problem instances of size \( k \) such that \( 6 \leq k < n \) and first and last are any indices such that \( k = last - first + 1 \). Since \( n > 5 \), line 35 is false and the general case is evaluated. Depending on the position and existence of the counterfeit coin, if and else if decision statements on lines 51, 56, and 58 can lead to the evaluation of 3 partitions, \( L \), \( R \), and \( O \).

- \( L : first + p - first \leq (n - 1) \)
- \( R : mid + p - mid \leq (n - 1) \)
- \( O : last - mid + p + 1 \leq (n - 1) \)

Thus, the inductive hypothesis is applied to all 3 recursive calls. So if any of lines 51, 56, and 58 are true, the preconditions of FindCoin are also satisfied and we can assume the subsequent call accomplishes the objective. The else if statement on line 51 dealing with the remainder coins leads to a constant number of weightings depending on a value of \( r \) with \( r \geq 2 \). If weighremaindercoins returns a coin we are done. If weighremaindercoins returns null, then the current \( c \) is assigned null and \( c \) is returned from this particular call. Thus, there were no counterfeit coins existing in main. Therefore, this is also a correct output satisfying the postcondition and we are done.
4. Computational Complexity Analysis

4.1. Time Complexity

The time complexity of our algorithm for finding a counterfeit coin is analyzed in terms of the number of times at which the comparing of weights between two coinsets occurs for a problem of size \( n \) coins. The best case occurs when \( n \) is a multiple of 3 and no counterfeit coin exists. When this happens, only 2 weightings occur and thus FindCoin has a best-case performance of \( \Theta(1) \).

The worst case occurs when \( n \) is not a multiple of 3 and the remainder is 2. For example, if \( n = 8 \) and the counterfeit coin was the 8th coin, there would be 4 weighings used to find that coin. The FindCoin algorithm effectively thirds the problem for each recursive call. To evaluate this further, it is necessary to show \( n \) coins in terms of \( 3^k \). Taking the logarithm of both sides gives us \( k = \log_3 n \). Since there are at least 2 weightings at every level of the decision tree, with the last level having a chance at 2 extra weighings when dealing with remainders, the total weighing comparisons is summarized as \( T(n) \leq 2 \log_3 n + 2 \). This would evaluate to \( T(n) = \Theta(\log n) \).

Another way of analyzing \( T(n) \) is to use Theorem 1 mentioned as a definition in this article. Since this algorithm fits the form of that recurrence relation, FindCoin be summarized as \( T(n) = T(n/3) + \Theta(1) \) where \( \Theta(1) \) represents a constant number of weighings. By applying Theorem 1, we are able to state \( T(n) = \Theta(\log n) \). Thus the worst case performance of FindCoin is \( \Theta(\log n) \).

4.2. Space Complexity

The space complexity of FindCoin can be defined by the bound of the summed complexities described. Since main is assumed to be given, it has a complexity of simply \( \Theta(1) \). Since it is assumed that partitioning main does not require any extra space on the stack, just pointer adjustment to the subset, one can imply that the space used by recursive FindCoin is proportional to logarithm of the number of coins contained in main. Mainly this will describe the depth of the search. This is described by \( \Theta(\log n) \). All the other variables: int first, int last, int mid, int r, int p, int n, and coin c each all have space complexities \( \Theta(1) \), respectively. When asymptotically comparing all the respective complexities, \( S(n) = \log_3 n + 8 \), one can trivially see that the space complexity for this algorithm bounded by \( \Theta(\log n) \).
4.3. Comparison with a Naïve Algorithm

The beginning of this algorithm checks the first 3 coins for a possible counterfeit. Since one needs at least 3 coins to determine a reference, these conditional statements accomplish both goals. At least, two weightings occur in lines 11-16. The next step involves the for loop executing n - 3 comparisons, that is if \( n > 3 \). It can also be assumed that \( i \) is being incremented \( n - 3 \) times as well. Thus the total time complexity for \( T(n) = 2(n - 3) + 2 \) is bounded by \( \Theta(n) \).

As for the space complexity, assume that the space that the main uses is again assumed to be provided and does not count towards the space complexity of this LinearFindCoin algorithm. Since the algorithm is iterative, only one stack is required to store the variables associated with it. There are 3 variables that are on the stack: int first, int last, and int i. Each variable has a space complexity of \( \Theta(1) \) so the total space complexity is \( \Theta(1) \). Therefore the space complexity of the LinearFindCoin algorithm is \( \Theta(1) \), \( S(n) = \Theta(1) \).

1 procedure LinearFindCoin (var main:coinset; first,last:integer):coin;
2     // Input:
3     // A reference to Coinset main containing coins;
4     // The integer assigned to a first index of the coinset
5     // The integer assigned to a last index of the coinset
6     // Assumptions:
7     // (1) coinset main contains > 2 coins
8     // Output:
9     // Finds a counterfeit coin \( c_i \) \( (1 \leq i \leq n) \) out of \( n \) coins, if it exists
10    // If the counterfeit coin does not exist, null is returned
11 { 12     if (main.cfirst.weight \( \neq \) main.cfirst+1.weight) then {
13         if (main.cfirst.weight \( \neq \) main.cfirst+2.weight) then
14             return main.cfirst
15         else
16             return main.cfirst+1
17     } else if (main.cfirst.weight \( \neq \) main.cfirst+2.weight) then {
18         return main.cfirst+2
19     };

October 30, 2010        DRAFT
// If the method gets this far, we know that none of the first 3 coins are counterfeit
// Use coin main.c\text{first} as a reference
// Loop invariant:
// 
// No counterfeit \( c_j \) has been found in \( first \leq j \leq i - 1 \)

for \( i \leftarrow 4 \) to last do {
  if (main.c\text{first}.\text{weight} \neq main.c_i.\text{weight}) then
    return main.c_i;
  \( i \leftarrow i + 1 \)
};
return null

5. Conclusion

In this paper, an algorithm for finding a counterfeit coin was presented to solve the problem of finding an unauthentic coin regardless of its location in a set of \( n \) coins. This recursive algorithm performs in a very straightforward manner. The correctness of the algorithm was rigorously proven, its computational complexities were analyzed, and it was compared to a naive algorithm. With a worst-case time complexity of \( \Theta(\log n) \) and a space complexity of \( \Theta(\log n) \), the algorithm proves to be an efficient solution for searching a coin set for a counterfeit coin.

References