1 Book 5.3: Bank Cards

Background:

Suppose you're consulting for a bank that's concerned about fraud detection, and they come to you with the following problem. They have a collenction of n bank cards that they've confiscated, suspecting them of being used in fraud. Each bank card is a small plastic object, containing a magnetic stripe with some encrypted data, and it corresponds to a unique account in the bank. Each account can have many bank cards corresponding to it, and we'll say that two bank cards are *equivalent* if they correspond to the same account. It's very difficult to read the account number off a bank card directly, but the bank has a high-tech "equivalence tester" that takes two bank cards and, after performing some computations, determines whether they are equivalent.

Question:

Among the collection of n cards, is there a set of more than $n/2$ of them that are all equivalent to one another? Assume that the only feasible operations you can do with the cards are to pick two of them and plug them in to the equivalence tester. Show how to decide the answer to their question with only $(n \cdot log(n))$ invocations of the equivalence tester.

1.1 Mathematical Formulation

Given an input of n bank cards, determine if $\exists$ a subset M where all cards $c_i \in M$ are identical and $n/2 < |M|$. s.t. $c_i = c_j$.

1.2 Solution

Important Confusing Data Structures:

- BankCard[n] **bankCards** : Store each Bank Card in each slot. We note that we are not able to sort this or do anything but access and swap positions i.e. can only access.

We will begin by making the observation that if $\exists$ a subset M where all cards $c_i \in M$ are identical and $n/2 < m = |M|$, then in one of the halves $bankCards[0..(n/2)]$ or $bankCards[(n/2+1)..n]$ $\exists$ at least $m/2$ identical cards. This algorithm will implement this recursively in order to determine whether or not the afformentioned statement holds. $Note : for\ the\ sake\ of\ this\ algorithm\ I\ will\ use\ BankCard[\,]\ to\ pass\ through\ but\ the\ actual\ algorithm\ would\ use\ a\ global\ array\ and\ pass\ the\ corresponding\ indices$

Algorithm 1 BankCard

```
procedure Equivalence(BankCard[] bankCards)
    n ← sizeOf(bankCards)
    m ← n/2
    if n == 1 then
        return bankCard[0]
    else n == 2
        if bankCards[0].equals(bankCards[1]) then
            return bankCard[0]
    bankCards1, bankCards2 ← bankCards[0..m], bankCards[(m+1)..n]
    card1, card2 ← Equivalence(bankCards1), Equivalence(bankCards2)
    if card1 is a card then
        test card1 against all other cards in bankCards and numberEqual++ when
equal
        if numberEqual == GOAL_SIZE then
            return card1
    if card2 is a card then
        test card2 against all other cards in bankCards and numberEqual++ when
equal
        if numberEqual == GOAL_SIZE then
            return card2
    return bankCards
```

If at the end of this method no card has been returned $\implies$ there is no matching of size n/2

1.3 Correctness

Proposition 1.
Propose that the algorithm will determine if $\exists$ *a subset* M *where all cards* $c_i \in M$ *are identical and* $n/2 < m = |M|$.

Proof.
This holds because if more than $n/2$ cards are equivalent, one of the halves $bankCards[0..(n/2)]$ or $bankCards[(n/2+1)..n]$ $\exists$ more than $m/2$ identical cards. $\therefore$ one of the two recursive calls must return a card equivalent to the whole set's majority equivalence $\therefore$ this algorithm compares all returned cards to the whole set, so then the majority equivalence will be found. □

1.4 Analysis

For the following analysis, we will say that..

Proposition 2. *The space complexity of this algorithm is* ***O(N)***

Proof.
This is due to the fact that all of our data is stored in the array containing all the cards. Everything else is pointers which are being initialized recursively so at most $3 \cdot log(N)$ of them will be active at a time.

Giving us a space complexity of **O(N)**

□

Proposition 3. *The time complexity of this algorithm is* ***O(N·log(N))***

Proof. This is the case because our algorithm calls the *Equivalence* method, where N cards = T(N), as two recursive calls (each of size n/2) and at most 2n tests for the two returned cards at each level. So T(n) ≅ 2T(n/2) + 2n = O(n log n) from class.

Giving us a time complexity of **O(N·log(N))**

□