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) $\cong$ 2T(n/2) + 2n = O(n log n) from class.

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

□

2 Book 4.12: Video Streams

Background

Suppose you have n video streams that need to be sent, one after another, over a communication link. Stream i consists of a total of b_i bits that need to be sent, at a constant rate over a period of t_i seconds. You cannot send two streams at the same time, so you need to determine a *schedule* for the streams: an order in which to send them. Whichever order you choose, there cannot be any delays between the end of one stream and the start of the next. Suppose your schedule starts at time 0 (and therefore ends at time $\sum_{i=1}^{n} t_i$, whichever order you choose). We assume that all the values b_i and t_i are positive integers.

Now, because you're just one user, the link does not want you taking up too much bandwidth, so it imposes the following constraint, using a fixed parameter r.

> *(∗) For each natural number $t > 0$, the total number of bits you send over the time interval from 0 to t cannot exceed $r \cdot t$.*

Note that this constraint is only imposed for time intervals that start at 0, *not* for time intervals that start at any other value. We say that a schedule is *valid* if it satisfies the constraint $(*)$ imposed by the link.

Questions:

Given a set of n streams, each specified by its number of bits b_i and its time duration t_i, as well as the link parameter r, determine whether $\exists$ a valid schedule.

2.0.1 Question 1:

$$\textit{Claim: } \exists \textit{ a valid schedule } \iff \textit{ each stream } i \textit{ satisfies } b_i \leq r \cdot t_i.$$

Decide whether you think the claim is true or false, and give a proof of either the claim or its negation.

2.0.2 Question 2:

Give an algorithm that takes a set of n streams, each specified by its number of bits b_i and its time duration t_i, as well as the link parameter r, and determines whether there exists a valid schedule. The running time of your algorithm should be polynomial in n.

2.1 Question 1:

False.
Proof by example, giving the case: $s_1 = (1,1)$, $s_2 = (2,1)$ and $r = 1$. We see here that this is clearly an acceptable order as there is no space and for $\underline{t=1} \implies b_1 \leq r \cdot t$ i.e. $1 \leq 1 \cdot 1$ so $*$ holds. Similarly for $\underline{t=2} \implies b_2 \leq r \cdot t$ i.e. $2 \leq 1 \cdot 2$ so $*$ holds. However this will fail the case since for s_2, $b_2 \nleq r \cdot t_2$ i.e. $2 \nleq 1 \cdot 1 \implies$ fails the case. $\square$

2.2 Question 2:

2.2.1 Mathematical Formulation

Given an input of n video streams, $s_1 = (b_1, t_1), s_2 = (b_2, t_1), ..., s_n = (b_n, t_n)$ where b_i = the load of stream i and t_i = the duration for how long it takes to transmit all data. We will say that in the correct order, there cannot be any delays between the end of one stream and the start of the next. i.e. time starts t=1 and goes to $\sum_{i=1}^{n} t_i$. We will denote the current time as T_i for when each step i begins where $T_1 = 1$. Additionally there is a rule that $b_i \leq r \cdot T_i$ for each stream i where r is a fixed parameter.

2.2.2 Solution

Important Confusing Data Structures:

- VideoStream[n] **streams** : Holds the video streams storing them as objects able to retrieve their (d_i, t_i)

The main functionality of this is to sort the video streams by d_i/t_i, and then check to see if the condition is met. If at any point it is not, then there does not exist a valid arrangement.

Algorithm 2 Method

```
procedure sortStreams(streams, r)
    sort(streams, Comparator())
    T ← 1
    for s_i ∈ streams do
        if b_i > r · T then
            return null
        T+ = t_i
    return streams
procedure compare(d_a, t_a, d_b, t_b)
    pos_a ← d_a/t_a
    pos_b ← d_b/t_b
    return pos_a − pos_b
```

2.2.3 Correctness

Proposition 4.
Propose that by sorting via d_i/t_i we gaurantee $\exists$ a valid ordered input.

Proof.
The motivation behind this is using the equality $d_i \leq r \cdot T_i \implies d_i/T_i \leq r$ must also hold. Therefore, we see that the i^{th} stream is depenent on the ratio between the load and the current time. $\therefore$ We develop our rational this way by saying that if we order them then in the order of their ratio, then those with lower b_i compared to t_i would appear first, i.e. $d_1/t_1 \leq d_2/t_2 \leq ... \leq d_n/t_n$. If $\exists$ a valid output then this will give us a valid output. This is due to the fact that if you were to rearrange any part, then the ratios would be inverted and then streams would be worse off. □

2.2.4 Analysis

Let N be the total number of streams being considered.

Proposition 5. *The space complexity of this algorithm is* ***O(N)***

Proof.
This is due to the fact that all of our data is stored within the object array of size N and two integers T *and* r.

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

□

Proposition 6. *The time complexity of this algorithm is* ***O(N·log(N))***

Proof. This is the case because our algorithm takes $(N \cdot log(N))$ to sort based off of this comparator and then linearly searches through, (N), doing the single check statement (1)

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

□