Abraham Cardenas

CMPS 102 — Winter 2019 – Homework 2

1. (10 pts)

   Given: An array A containing n valid bank cards, we need to find a bank card that appears more than $\lfloor \frac{n}{2} \rfloor$ times. Also, we are restricted in only using the "equivalent tester" as an operation. We will give this "tester" the method name "Equiv-test".

   To distinguish if our algorithm found no frauds, we will return an "invalid bank card". This invalid bank card is a card that's not present in our array A.

```
function FIND-FRAUD($A$, $n$, $result$)                    ▷ $result$ is passed by reference.
    $card \leftarrow$ Find-fraud-recurse($A$, 0, $n$)
    if $card$ is a valid bank card then
        $result = card$
        return true
    else
        return false
    end if
end function
function FIND-FRAUD-RECURSE($A$, $\ell$, $u$)
    if $u - \ell = 0$ then
        return an invalid bank card
    else if ($u - \ell = 1$) or ($u - \ell = 2$ and Equiv-test($A[0]$, $A[1]$) = true) then
        return $A[0]$
    else
        $m \leftarrow \lfloor \frac{u+\ell}{2} \rfloor$
        $cardL \leftarrow$ Find-fraud-recurse($A$, $\ell$, $m$)
        if $cardL$ is a valid bank card then
            $found \leftarrow$ Check-card-majority($A$, $u$, $cardL$)
            if $found$ = true then
                return $cardL$
            end if
        end if
        $cardR \leftarrow$ Find-fraud-recurse($A$, $m + 1$, $u$)
        if $cardR$ is a valid bank card then
            $found \leftarrow$ Check-card-majority($A$, $u$, $cardR$)
            if $found$ = true then
                return $cardR$
            end if
        end if
        return an invalid bank card
    end if
end function
function CHECK-CARD-MAJORITY($A$, $n$, $card$)
```

```
35:     $count \leftarrow 0$
36:     $i \leftarrow 0$
37:     **while** $i < n$ **do**
38:         **if** Equiv-test($A[i]$, $card$) = true **then**
39:             $count \leftarrow count + 1$
40:         **end if**
41:         $i = i + 1$
42:     **end while**
43:     **if** $count > \lfloor \frac{n}{2} \rfloor$ **then**
44:         **return** true
45:     **else**
46:         **return** false
47:     **end if**
48: **end function**
```

Claim: Given an array A of size n, if the algorithm returns true, the bank card appended to $result$ will be an element in A (either from the left half side or right half side) that is present more than $\lfloor \frac{n}{2} \rfloor$ times. Otherwise, if the algorithm returns false, the algorithm sets $result$ to an invalid bank card since no fraud was found in A.

Induction Proof:

Base case: Our base case occurs when have an array A of size ≤ 2. When the array size is 0 (line 11), our algorithm returns an invalid bank card since by definition, an array of size 0 deos not contain elements and thus, contains no valid bank cards. When the array size is 1 (line 13) our algorithm returns the first element in the array. Similarly, when the array size is 2 and both elements in the array are the same (line 13), our algorithm returns the first element. In all cases, the algorithm clearly works.

Inductive Step: We have to prove that the algorithm works for any array A of size ≥ 3. Let's assume the algorithm works for any array of size $k - 1$. Let us prove the algorithm works for an array of size k.

We have the following cases:

(1) When $u - \ell = 0$, the algorithm clearly works.

(2) When $u - \ell = 1$ or $u - \ell = 2$, again, the algorithm clearly works.

(3) When $u - \ell \geq 3$, we first recurse through the left half side of the array $A[\ell \text{ .... } m]$. The size of this side of the array is $n = m - \ell = \lfloor \frac{\ell+u}{2} \rfloor - \ell$. If $\ell + u$ is odd, then $n = \frac{\ell+u-1}{2} - \ell = \frac{u-\ell-1}{2}$ which is smaller than $k = u - \ell$. Conversely, if $\ell + u$ is even, then $n = \frac{\ell+u}{2} - \ell = \frac{u-\ell}{2}$, which is smaller than $k = u - \ell$. Hence, the recursive call must be between 0 and $k - 1$, and is correct by the induction hypothesis.

(a) If our recursive call on the left half side of the array returned a valid bank card $cardL$, then we loop through array A by first letting $count = 0$ and $i = 0$. Our $count$ will contain the amount of times $cardL$ has appeared in $A[0 \text{ .... } i]$. In the loop, if Equiv-test returns true at any point i ($A[i] = cardL$), our $count$ increases by one. Otherwise, we increment i by 1 until the loop terminates. At loop termination, $i = k - 1$ which means that $count$ will contain the amount of times $cardL$ has appeared in $A[0 \text{ .... } k - 1]$.

(b) If $count > \lfloor \frac{k}{2} \rfloor$, we return $cardL$ (found a fraud). Otherwise, we continue and check the remaining right half side elements of A.

(4) When $u - \ell \geq 3$ and when $cardL$ was not returned, we recurse through the right half side of the array $A[m+1 \;....\; u]$. The size of this half is $n = u - (m+1) = u - \lfloor \frac{\ell+u}{2} \rfloor - 1$. If $\ell + u$ is even, then $n = \frac{u-\ell}{2} - 1$, which is less than $k = u - \ell$. On the other hand, if $\ell + u$ is odd, then $n = u - \frac{(\ell+u-1)}{2} - 1 = \frac{u-\ell}{2} - \frac{1}{2}$, which is also less than $k = u - \ell$. Hence, the recursive call uses a smaller range of values within array A and is thus, correct by our induction hypothesis.

(a) If our recursive call on the right half side of the array returned a valid bank card $cardR$, we repeat the same process as (3a) but instead of using $cardL$, we use $cardR$. So, $count$ will contain the amount of times $cardR$ appears in $A[0 \;....\; k-1]$.

(b) If $count > \lfloor \frac{k}{2} \rfloor$, we return $cardR$ (found a fraud).

(5) We have recursed through $A[\ell \;....\; m] = A[0 \;....\; \lfloor \frac{u+\ell}{2} \rfloor] = A[0 \;....\; \lfloor \frac{k}{2} \rfloor]$ and $A[m+1 \;....\; u] = A[\lfloor \frac{k}{2} \rfloor + 1 \;....\; k-1]$. Combining both sides, we can see that we recursed through $A[0 \;....\; \lfloor \frac{k}{2} \rfloor, \lfloor \frac{k}{2} \rfloor + 1, \;....\; k-1] = A[0 \;....\; k-1]$. Since we recursed through the entire array, we can conclude that no fraud exists in the array. So, we return an invalid bank card (fraud not found).

From (3b) and (4b) we can conclude that the algorithm will either return a fraud that it finds on the left half side or right half side. And from (5), we can conclude that if no fraud exists in the array, the algorithm will return an invalid bank card.

Therefore, since the algorithm works in all cases, we can conclude that the algorithm will work on an array A of size k. $\square$

Recurrence relation:

From lines 1-16, our algorithm does constant amount of work. From lines 17-48, our algorithm does recursion 2 times on an array whose size is $\frac{1}{2}$ smaller than the original array. Also, we loop through array A at most, 2 times. Time taken looping twice is linear in time, so our recurrence relation is as follows:

$$\boxed{T(n) = 2T(\frac{n}{2}) + \Theta(n)}$$

Upper Bound:

Using the master theorem we can see that the upper bound would be:

$\Rightarrow$ $\quad T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n^1)$ where $a = 2$, $b = 2$, and $d = 1$

Since $2 = 2^1$,

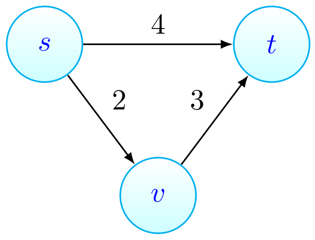
$\Rightarrow$ $\quad T(n) = \Theta(n^1 \log n) = \boxed{\Theta(n \log n)}$

2. (6 pts)

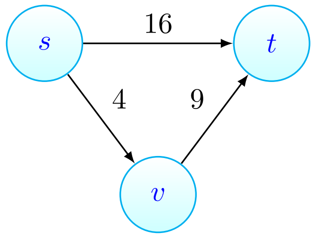
a) This statement is true because for example, Kruskal's algorithm first sorts the weighted edges in increasing order. Squaring all the weights would still result in the same set of sorted edges so the MST would not change.

b) This statement is false.

Counter-example:



The shortest path from s to t is the edge (s, t). If we square all the weights then we will have the following graph.



The shortest path from s to t is now the path from edge (s, v) to (v, t).

3. (9 pts)

Given: Two sequences of events S_1 and S_2, we need to find if S_1 is a subsequence of S_2. We will accomplish this by looping through each sequence until we've looped through all of S_1 and/or all of S_2.

```
function IS-SUBSEQUENCE($S_1$, $S_2$, $m$, $n$)
    if $m = 0$ or $n = 0$ then
        return false
    end if
    $i \leftarrow 0$
    $j \leftarrow 0$
    while $j < m$ and $i < n$ do
        if $S_1[j] = S_2[i]$ then
            $j = j + 1$
        end if
        $i = i + 1$
    end while
    if j = m then
        return true
    else
        return false
    end if
```

```
18:  end function
```

Claim: The algorithm either returns true or false depending if S_1 is a subsequence of S_2.

Induction Proof:

Base case: The base case occurs when the size of S_1 is 0 or when the size of S_2 is 0. In both cases, it's clear that S_1 is not a subsequence of S_2 so the algorithm returns false.

Inductive Step: We need to show that the algorithm works for any sequence S_1 of size ≥ 1 and any sequence S_2 of size ≥ 1. Let's assume the algorithm works for any sequence S_1 of size $\leq k_1 - 1$ and any sequence S_2 of size $\leq k_2 - 1$. Let's prove the algorithm works for a sequence S_1 of size k_1 and a sequence S_2 of size k_2.

Before iterating, the algorithm initializes both j and i to 0. Both are used to index S_1 and S_2 respectively.

(I.) The algorithm only increments j when finding a matching event in S_2 (line 8). So, the subsequence $S_1[0 \text{ .... } j-1]$ will contain the events that are matched in $S_2[0 \text{ .... } i-1]$. Since each event happened after the next, each event will be matched in chronological order since S_1 and S_2 are indexed starting from the first event (0^{th} event). During each iteration, we repeat this process and increment i while doing so.

From the induction hypothesis, we can assume that the loop terminates and that we have the following cases:

(1) When $j = k_1$ and $i \leq k_2$, we can use (I.) to see that we looped through $S_1[0 \text{ .... } j-1] = S_1[0 \text{ .... } k_1 - 1]$ and $S_2[0 \text{ .... } i-1]$ where $i \leq k_2$. We can conlude that because all events of S_1 were matched in part of or all of S_2, S_1 must be a subsequence of S_2 so we return true.

(2) When $j < k_1$ and $i = k_2$, we can use (I.) to see that we looped through at most, $S_1[0 \text{ .... } j-1] = S_1[0 \text{ .... } (k_1-1)-1] = S_1[0 \text{ .... } k_1-2]$ and $S_2[0 \text{ .... } i-1] = S_2[0 \text{ .... } k_2-1]$. We can conclude that we looped through all of S_2 and, at most, matched every event in S_1 except the last event. And because we didn't find every event of S_1 in S_2, we can conclude that S_1 is not a subsequence of S_2 so we return false.

Using (1) and (2), we can conclude that the algorithm will either return true or false if S_1 is a subsequence of S_2. Therefore, since the algorithm works in all cases, we can conclude that the algorithm will work on a sequence S_1 of size k_1 and a sequence S_2 of size k_2. $\square$

Recurrence relation:

From lines 2-6 and 13-17, we only do a constant amount of work. From lines 7-12, we loop through all of S_1 and/or S_2. We can express this as the following recurrence:

$$\boxed{T(m,n) = T(m-1,n-1) + T(m,n-1)}$$

Upper Bound:

In the worst-case, we would have to traverse through all of S_1 and all of S_2, so our upper bound would be:

$$\boxed{O(n+m)}$$

Sources

https://www.geeksforgeeks.org/majority-element/
https://en.wikipedia.org/wiki/Kruskal%27s_algorithm