CS 577 Homework 2

Alex Pizzuto

Question One: 5.3

Our goal is to, given a stack of n cards, decide if at least $n/2$ of them are equivalent in $\mathcal{O}(n \log n)$ time. First, we make the observation that if we do have one value that makes up greater than half of the entries, that if we divide our entries into two evenly sized piles, that at least one of those piles will have at least hald of the entries that are equal to the majority value. Thus, we design a divide and conquer approach. First, we will divide the pile into two evenly sized piles, then check recursively if in either pile there is a majority of one value. If either one of the searches on either half returns a majority value, then we check that value against all of the other cards (this step would only be $\mathcal{O}(n)$, but is necessary because just because there is a majority in one pile does not mean that there is a global majority). We will use the notation where a capital letter represents a set of cards, and lowercase indexed letters refer to the elements (and we start our indexing at zero). Thus, we have First, we verify correctness. If there exists a value that makes up a majority

```
FindMajorityCard A:
    if sizeOf A is 1 then
        return a_1;
    else
        x = FindMajorityCard(first half of A);
        y = FindMajorityCard(second half of A);
        if x or y is a real value then
            test x and y against the rest of the cards;
            return x or y if either is the majority value;
        else
            return None;
        end
    end
```

Algorithm 1: Find Majority Value Algorithm for determining if a class of objects has at least half of its elements in one equivalence class

of the pile, then it is a majority of either the first half or the second half (or possibly both) halves. Thus, the value is found, and the correct value is returned.

Now, we must analyze the running time. At each level, we make two recursive calls, one on the first half of the set we are looking at, one at the second half. Additionally, we might possibly have to do tests against all of the other cards if we have found a majority in one of the halves. Assuming the worst case (both halves have contenders for a majority value at each level), and calling c the

amount of computation time necessary to make comparisons, we get

$$\begin{aligned} T(n) &= 2T(n/2) + 2 \cdot c \cdot n \\ &\Rightarrow T(n) = \mathcal{O}(n \log n) \end{aligned}$$

Question Two

Our goal is to seat n patrons in a row of n seats where any seating causes the cascading reseating of adjacent seats, and we find an algorithm to place the customers in their seats such that only $\mathcal{O}(n \log n)$ seatings/reseatings are necessary. In general, we wish to avoid placing customers in adjacent seats until it is completely necessary. In the case of 3 seats, we would place the outer customers (1 and 3) and then finally the middle. Although the last seating will cause everybody to be reseated, it is the only level where we have reseatings. For example's sake, we also work out the next case of having 7 seats. Here, we want to place the very middle (4) last, but we also want to place the middles of either half (2 and 6) second last. So we would seat:

$$\begin{aligned} 1,\ 3,\ 5,\ 7 \rightarrow &\quad \text{4 seatings} \\ 2,\ 6 \rightarrow &\quad \text{6 seatings (each value causes 2 additional reseatings)} \\ 4 \rightarrow &\quad \text{7 seatings} \end{aligned}$$

It is clear, then, that we have a way to place the customers, and it amounts to the same way of a level order removal of a balanced binary tree. Although we present the closed form solution in terms of an iteration, it is based on a structuring similar to a heap, and are placing based off of location in subtrees, so it is an inherently divide and conquer algorithm. Thus, we can formalize the algorithm, where $\exists\ k \in \mathbb{Z}_{\geq 0}$ such that $n = 2^k - 1$:

for *i in 1 ... k* **do**
  for *j in 1 ...* 2^{k-i} **do**
    seat $j \cdot 2^i - 2^{i-1}$;
  end
end

Algorithm 2: Placement algorithm for putting n patrons in a row of n seats with as few reseatings as possible

We will now analyze the running time by performing an explicit sum. It is clear that during the i^{th} iteration of the outer loop, each seating requires 2^{i-1} reseatings, and there are 2^{k-i} placements at each level. Thus, we have

$$\begin{aligned} T(n) &= c \sum_{i=1}^{k} (2^{i-1} - 1) 2^{k-i} \\ &= c\Big(2^k - 1 + k2^k\Big) \\ &= c(n + n \log n) \\ &\Rightarrow T(n) = \mathcal{O}(n \log n) \end{aligned}$$

Question Three

Our goal is to find the maximal difference of all pairs in an array, where we must subtract the number on the right from the number on the left. One way to think about this problem is if we divide the array into two halves, then the greatest difference is either completely contained in the first half, completely contained in the second half, or obtained by taking a large number in the first half and a small number from the second half, so we can solve this problem recursively. We begin with the pseudo-code of the algorithm:

MaximalDifference *(A)*:
    if *size of A is 1* **then**
        return the only element in A;
    end
    leftResult = **MaximalDifference** *(First half of A)*:
    rightResult = **MaximalDifference** *(Second half of A)*:
    x = max(First half of A);
    y = min(Second half of A);
    crossTerm = x - y;
    return max(leftResult, rightResult, crossTerm);

Algorithm 3: MaximalDifference algorithm to find the largest ordered difference in an array

Correctness is ensured as all three possibilities for the placement of the elements of the maximal difference have been handled. We now analyze the complexity of this algorithm. At each level, we make 2 recursive calls to arrays of half the initial size. Additionally, we perform one max and min operation at each level as well as 2 $\mathcal{O}(1)$ operations. Thus, we have:

$$\begin{aligned} T(n) &= 2T(n/2) + 2c_1 n + 2c_2 \\ &= 2(2T(n/4) + 2c_1 \frac{n}{2} + 2c_2) + 2c_1 n + 2c_2 \\ &= 4T(n/4) + c_1 \cdot n(2+2) + (2+4)c_2 \\ &\vdots \\ &= \mathcal{O}(n) + \mathcal{O}(n \log n) \\ &\Rightarrow T(n) = \mathcal{O}(n \log n) \end{aligned}$$