CS 317, Assignment 3

Answers

This assignment is all about divide and conquer algorithms, even if this is not explicitly stated in the questions below.

1. You are given a sorted sequence of integers $A_{1\ldots n}$ that has been circularly shifted k positions to the right. For example, $\langle 35, 42, 5, 15, 27, 29\rangle$ is a sorted sequence that has been circularly shifted $k = 2$ positions, while $\langle 27, 29, 35, 42, 5, 15\rangle$ has been shifted $k = 4$ positions.

   Design an algorithm that finds the index of the largest element in A in $O(\log n)$ time. Analyze the running time of your algorithm and provide a proof of correctness.

ANSWER:

For $k = 0$ the largest element is A_n, and $A_i \leq A_{i+1}$ for all $1 \leq i \leq n-1$. Consider now A in a circular fashion i.e., the element after A_n (that is, A_{n+1}) is A_1. Clearly, $A_n \geq A_{n+1}$. Overall all the elements in the circular array are followed by a larger element, with the exception of the largest (A_n) which is followed by a smaller element (the sole inversion). This property is not affected by a circular shift (which preserves the circular ordering) so it is valid for all k. We can thus find the largest element by looking for the inversion following it.

We can also find whether the sub-array $A_{l\ldots h-1}$ contains the maximum in constant time by checking whether $A_l > A_h$ (which implies an inversion). In all the following binary search-like algorithm will do:

```
algorithm FindMax(A, l, h):
    if l = h then return A_l
    else
        k ← (h − l)/2                                   (middle of the array)
        if A_k > A_{k+1} then return k                  (the middle is the maximum)
        if A_l > A_k then
            return FindMax(A, l, k − 1)                 (inversion in the first half means the maximum is there)
        else
            return FindMax(A, k + 1, h)                 (not in the first half so it must be in the second half)
```

The above argument readily establishes the correctness of the algorithm, but here is a formal proof by structural induction, for completeness:

For the base case, the algorithm correctly returns the sole element of the array as the maximum. For the inductive step, we note that the maximum is either A_k, in $A_{l\ldots k-1}$, or in $A_{k+1\ldots h}$. In the former case the index k is correctly returned since the condition of the first if is true

(the maximum is followed by an inversion). Otherwise if the inversion is located in $A_{l\ldots k-1}$ then $A_l > A_k$ and so we correctly call FindMax(A, l, $k-1$) which will return the index of the maximum value by the inductive hypothesis (we already know that the maximum will be somewhere in $A_{l\ldots k-1}$). Finally, if $A_l \leq A_k$ then we know that the maximum cannot be in $A_{l\ldots k-1}$. We also know that the maximum is not A_k (for it would have been returned earlier). Therefore the maximum can only be in $A_{k+1\ldots h}$, and the recursive call FindMax(A, $k+1$, h) will correctly find it for us by the inductive hypothesis.

The running time is just like the running time for binary search: we spend a constant time before the recursive call which is on half the original sequence, and so $T(n) = T(n/2) + 1$ which means that $T(n) = \Theta(\log n)$ (we already discussed that in class), as desired.

---

2. We want to sort a stack of n pancakes by size so that the largest pancake is at the bottom and the smallest at the top. For convenience we index the stack from 1 to n, with index 1 at the top and n at the bottom. We are allowed only two operations: FindLargest(n) returns the index of the largest pancake in an n-pancake stack, and Flip(k) flips the stack of pancakes $1 \ldots k$ so that the k-th pancake becomes the first, the $k-1$-st pancake becomes the second, and so on (just like inserting a spatula immediately below index k and using it to flip the stack of pancakes above it).

   (a) Design a linear-time algorithm to sort an arbitrary stack of n pancakes. Analyze the running time of your algorithm and provide a proof of correctness.

   Answer:

   We can bring the largest pancake at the top by finding it and then flipping the stack ending at the pancake we found. We can then bring the largest pancake in its proper place (at the bottom) by flipping the whole stack. What is left is to sort the top of the stack without the pancake we already placed in its proper place:

   algorithm FlipAll(n)**:**
   - **if** $n = 1$ **then return**
   - **else**
     - $k \leftarrow$ FindLargest(n)
     - Flip(k)
     - Flip(n)
     - FlipAll($n-1$)

   Correctness is easily established by structural induction (what else): For the base case we have a single pancake which is already sorted so the algorithm correctly does nothing. For the inductive step, k is the index of the largest pancake. We then flip the stack 1 to k and so pancake k (the largest) is now at the top. We then flip the whole stack with the obvious consequence that the top pancake (the largest) is now at the bottom. This means that the largest pancake is now in the right place (index n or bottom) in the stack. The remaining $n-1$ pancakes are then sorted by the recursive call (inductive hypothesis), for an overall sorted stack. Bring in the syrup!

I did not insist on total correctness, but perhaps now is a good time to mention it. The above proves partial correctness. For a complete proof we also need to consider termination, thus proving total correctness. In this respect we note that FlipAll always terminates since the argument decreases by 1 on each recursive call (from n to $n-1$) so it will eventually reach 1 which is the base case.

The number of flips is given by $T(1) = 0$ (no flips for a single pancake) and $T(n) = 2 + T(n-1)$ (two flips then the recursive call) or $T(n) - T(n-1) = 2$. The homogeneous part yields $r - 1 = 0$ and so $r_1 = 1$ and for the non-homogeneous part we have $b = 1$ and $d = 0$ that is, $(r-1)^1 = 0$ and so $r_2 = r_1 = 1$. The running time is therefore $T(n) = c_0 1^n + c_1 n 1^n = c_0 + c_1 n = \Theta(n)$, right where we wanted it to be.

(b) Discuss the optimality of your algorithm. (Hint: describe a stack of n pancakes that requires $\Omega(n)$ flips to sort.)

Answer:

We assemble our stack of pancakes by putting the largest at the top, the next largest at the bottom, and so on recursively until we run out of pancakes. For example, the set of pancakes $\{1, 2, 3, 4, 5, 6, 7\}$ (where the number denotes the size of that pancake) will end up arranged as the stack $\langle 7, 5, 3, 1, 2, 4, 6\rangle$ (with the top on the left).

We show that we need at least $n-1$ flips to sort this stack by induction over n: For a stack of height 1 we clearly need 0 $(1-1)$ flips. We now have a stack with the largest pancake at the top, so clearly not in the right place; therefore we need at least one flip to bring it to the said right place. If we flip the whole stack then we need to sort the remaining $n-1$ pancakes which takes at least $n-2$ flips by inductive hypothesis, so the overall number of flips is no less than $n-2+1 = n-1$, as desired. Suppose now that we do not flip the whole stack, only some prefix of k pancakes. The largest pancake is not at the bottom, so in all we will spend at least 2 flips to bring it there (the one we already performed and at least one more). The stack below the largest pancake as well as the stack at the top of said largest pancake have both the same recursive structure as the original stack, so they require at least $n-k-1$ and $k-2$ flips to sort, respectively by inductive hypothesis. This means that overall we need at least $2+n-k-1+k-2 = n-1$ flips to sort the whole stack, as desired.

This particular stack of pancakes requires $\Omega(n)$ time to sort, our algorithm runs in $\Theta(n)$ time, so our algorithm is optimal.

(c) Suppose now that one side of each pancake is burned. Describe a linear-time algorithm that sorts an arbitrary stack of n pancakes so that the burned side of each pancake faces down. Analyze the running time of your algorithm and provide a proof of correctness.

Answer:

I did not specify this explicitly in the question, but we certainly need some way to see whether a pancake is with the burned side up or down. For the simple algorithm below

I only need to check the top pancake, so I will minimally add to my available operations the predicate BurnedIsDown() which returns true iff the top pancake is burned side down.

This is then the simplest algorithm I can think of: We flip the largest pancake to the top, just like we did earlier. If that pancake is burned side up then we just flip the whole stack and we are done with that pancake (which is at the bottom with the burned side down). Otherwise we flip the top (largest) pancake, which thus ends up with the burned side up. Then we flip the whole stack; once more the largest pancake is in its proper place and with the proper side down. All is left is to recursively sort the rest of the pancakes.

algorithm FlipAllBurned(n)**:**

```
if n = 1 then return
else
    k ← FindLargest(n)
    Flip(k)
    if BurnedIsDown() then Flip(1)
    Flip(n)
    FlipAllBurned(n − 1)
```

This is a rather minor modification of FlipAll. The proof of correctness goes just like for FlipAll with very minor additions, so I am not going to reproduce it here (but you are encouraged to try it out). Asymptotically the running time is just like for FlipAll (we only add a constant-time step) that is, $\Theta(n)$.

This algorithm is once more optimal. Indeed, the original problem is a simplification of this problem, and so the lower bound $\Omega(n)$ established earlier for the original problem applies to this problem as well. Our algorithm FlipAllBurned also meets this lower bound, and thus is optimal.

Note in passing that even if they are asymptotically optimal, neither FlipAll nor FlipAllBurned are the last word in pancake sorting. These problems have received sustained attention and better algorithms have been developed; their running times are still $\Theta(n)$, but the constants hidden by the asymptotic notation can be and have been lowered in the past. Obviously though this is beyond the matter covered in the course and I will stop here since I am getting really hungry by now.

---

3. Let T be a binary tree with n vertices. Deleting any vertex v splits T into at most three subtrees, one rooted at the left child of v (if any), the second rooted at the right child of v (if any), and the third containing the parent of v (if any). We call v a central vertex if each of these smaller trees has at most $n/2$ vertices.

   (a) Show that every binary tree has a central vertex.

Answer:

To avoid tiresome repetitions, I will abuse the notation in what follows and refer to the tree rooted at a vertex v also as v. Which of the two I mean will be clear from the context.

Let $\textsc{Vertices}(x)$ denote the number of vertices in the tree rooted at x. Call our input tree T and let $n = \textsc{Vertices}(T)$. We now go down in T either left or right toward the child with the largest number of vertices, stopping at the first vertex c such that $\textsc{Vertices}(\textsc{Left}(c)) \leq n/2$ and $\textsc{Vertices}(\textsc{Right}(c)) \leq n/2$. We started with a tree with n vertices, each time we do down the number of vertices decreases (since we no longer count the current vertex let alone the vertices in the unexplored child), and we can continue down our path until we reach a leaf that is, a tree with 0 vertices. This means that we will always encounter on our path a vertex x such that $\textsc{Vertices}(x) \leq n/2$. The parent of x is then our c, which thus always exists.

Let us see now how the vertex c splits the tree. Let p be the parent of c. It must be that $\textsc{Vertices}(c) > n/2$, for otherwise we would have stopped at p. Indeed, $\textsc{Vertices}(c) \leq n/2$ means that the sibling of c also has less than $n/2$ vertices (c has more vertices than its sibling) and the stopping condition would have been met at p. Since c has more than $n/2$ vertices, it follows that the tree containing p has less than $n/2$ vertices. Additionally, $\textsc{Vertices}(\textsc{Left}(c)) \leq n/2$ and $\textsc{Vertices}(\textsc{Right}(c)) \leq n/2$ by definition. It follows that c is a central vertex.

(b) Describe an algorithm to find a central vertex in an arbitrary given binary tree. Analyze the running time of your algorithm and provide a proof of correctness.

Answer:

The previous question establishes the correctness of the following algorithm:

```
algorithm FindCenter(T):
    CountVertices(T)
    FindCenterRec(T, Vertices(T))
algorithm CountVertices(T):
    if Null(T) then return 0
    else
        n_l ← CountVertices(Left(T))
        n_r ← CountVertices(Right(T))
        Vertices(T) ← 1 + n_l + n_r
        return Vertices(T)
algorithm FindCenterRec(v,n):
    n_l ← Vertices(Left(v))
    n_r ← Vertices(Right(v))
    if n_l ≤ n/2 ∧ n_r ≤ n/2 then return v
    if n_l ≤ n_r then return FindCenterRec(Right(v), n)
    else return FindCenterRec(Left(v), n)
```

I am not going to bother with the correctness of CountVertices, it is just too basic.

FindCenter could have also been implemented iteratively, but as I said at the beginning of the handout this is all about divide and conquer (aka recursive) approaches, hence my recursive solution.

As for the running time, we note that CountVertices(T) takes $\Theta(n)$ time since we have one call to CountVertices for each vertex in the tree. Every recursive call to FindCenterRec happens one level down in the tree form the parent call, so there cannot be more calls to FindCenterRec than the height of the tree, which is smaller than n. Therefore the overall running time is $\Theta(n)$ (being dominated by the running time of CountVertices(T)).

Make sure you review the submission guidelines posted on the course's Web site before submitting. Note in particular (last warning) that the only acceptable ways to describe an algorithm are pseudo-code (preferred) or actual code. Textual descriptions in particular are not acceptable.