CSCI-GA.1170-003/004 Fundamental Algorithms February 17, 2019

Problem Set 3

Lecturer: Yevgeniy Dodis *Due: Thursday, February 21*

Problem 3-1 (Stock Profit) 10 (+6) Points

Sometimes, computing "extra" information can lead to more efficient divide-and-conquer algorithms. As an example, we will improve on the solution to the problem of maximizing the profit from investing in a stock (page 68-74).

Suppose you are given an array A of n integers such that entry $A[i]$ is the value of a particular stock at time interval i. The goal is to find the time interval (i, j) such that your profit is maximized by buying at time i and selling at time j. For example, if the stock prices were monotone increasing, then $(1, n)$ would be the interval with the maximal profit $(A[n] - A[1])$. More formally, the current formulation of the problem has the following input/output specification:

Input: Array A of length n.

Output: Indices $i \leq j$ maximizing $(A[j] - A[i])$.

(a) (6 Points) Suppose your change the input/output specification of the stock problem to also compute the largest and the smallest stock prices:

Input: Array A of length n.

Output: Indices $i \leq j$ maximizing $(A[j] - A[i])$, and indices α, β such that $A[\alpha]$ is a minimum of A and $A[\beta]$ is a maximum of A.

Design a divide-and-conquer algorithm for this modified problem. Make sure you try to design the most efficient "conquer" step, and argue why it works. How long is your conquer step? (**Hint**: When computing optimal i and j, think whether the "midpoint" $n/2$ is less than i, greater than j or in between i and j.)

(b) (4 Points) Formally analyze the runtime of your algorithm and compare it with the runtime of the solution for the Stock Profit problem in the book.

(c) (**Extra Credit**; 6 Points) Design a direct, non-recursive algorithm for the Stock Profit problem which runs in time $O(n)$. Write its pseudocode. Ideally, you should have a single "**for** $i = 1$ **to** n" loop, and inside the loop you should maintain a few "useful counters". Formally argue the correctness of your algorithm.
(**Hint**: Scan the array left to right and maintain its running minimum and the best solution found so far. Under which conditions would the best current solution be improved when scanning the next array element?)

Problem 3-2 (Identity Element in an Array) 10 Points

Let A be an array with n distinct integer elements in sorted order. Consider the following algorithm IdFind(A, j, k) that finds an $i \in \{j \dots k\}$ such that $A[i] = i$, or returns **FALSE** if no such element i exists.

```
1 IdFind($A, j, k$)
2     if $j > k$ return FALSE
3     Set $i := \dots$
4     if $A[i] = \dots$ return $\dots$
5     if $A[i] < \dots$ return IdFind($A, \dots, \dots$)
6     return IdFind($A, \dots, \dots$)
```

(a) (3 points) Fill in the blanks (denoted ...) to complete the above algorithm.

(b) (5 points) *Prove* correctness and analyze the running time of the algorithm.
(Notice the emphasis on *Prove*, you can't just say "my algorithm works because it works".)

(c) (2 points) Does the algorithm work if the elements of A are not distinct? Why or why not?

Problem 3-3 (Tower of Hanoi) 10 (+6) points

The Tower of Hanoi is a well known mathematical puzzle. It consists of three rods, and a number n of disks of different sizes which can slide onto any rod. The puzzle starts with all disks stacked up on the 1st rod in order of increasing size with the smallest on top. The objective of the puzzle is to move all the disks to the 3rd rod, while obeying the following rules.

- Only one disk is moved at a time
- Each move consists of taking one disk from top of a rod, and moving it on top of the stack on another rod
- No disk may be placed on top of a smaller disk.

A recursive algorithm that solves this problem is as follows: We first move the top $n-1$ disks from rod 1 to rod 2. Then we move the largest disk from rod 1 to rod 3 and then move the $n-1$ smaller disks from rod 2 to rod 3. Using the symmetry between the rods, the number of steps that this algorithm takes is given by the recurrence

$$T(n) = 2T(n-1) + 1 \ ,$$

which can be solved to get $T(n) = 2^n - 1$.

(a) (5 points) Show that the above algorithm is optimal, i.e., there does not exist a strategy that solves the Tower of Hanoi puzzle in less than $2^n - 1$ steps.

(b) (5 points) Suppose the moves are restricted further such that you are only allowed to move disks to and from rod 2. Give an algorithm that solves the puzzle in $O(3^n)$ steps.

(c) (6 points(**Extra credit**)) Suppose the moves are restricted such that you are only allowed to move from rod 1 to rod 2, rod 2 to rod 3, and from rod 3 to rod 1. Give an algorithm that solves the puzzle in $O((1+\sqrt{3})^n)$ steps.

Problem 3-4 (Slow Sorting) 15 points

Consider the following recursive sorting algorithm. Note that $\lceil x \rceil$ is the smallest integer greater than or equal to x, while $\lfloor x \rfloor$ is the largest integer less than or equal to x.

```
SLOW-SORT($A[1 \ldots n]$)
    if $n < 8$
        INSERTION-SORT($A[1 \ldots n]$)
    else
        SLOW-SORT($A[1 \ldots \lceil 5n/8 \rceil]$)
        SLOW-SORT($A[\lfloor 3n/8 \rfloor + 1 \ldots n]$)
        SLOW-SORT($A[1 \ldots \lceil 5n/8 \rceil]$)
```

(a) (3 points) Give a counter-example on a list of size 8 whose elements are from the set $\{1, 2, \ldots, 8\}$ to prove that the above algorithm won't work.

(b) (8 points) However, it turns out that the algorithm is close to being correct. Prove that the followind modified version of SLOW-SORT does in fact work.

```
SLOW-SORT($A[1 \ldots n]$)
    if $n < 8$
        INSERTION-SORT($A[1 \ldots n]$)
    else
        SLOW-SORT($A[1 \ldots \lceil 2n/3 \rceil]$)
        SLOW-SORT($A[\lfloor n/3 \rfloor + 1 \ldots n]$)
        SLOW-SORT($A[1 \ldots \lceil 2n/3 \rceil]$)
```

In your proof, describe and justify three invariants that hold after each of the recursive calls. As a hint, the last invariant should be, the entire array is sorted. Additionally, explain why your proof does not work for the original algorithm.

(c) (4 points) Give and solve the recurrence relation to bound the running time of the correct version of SLOW-SORT. How does this compare to other sorting algorithms that we have seen in class?