CSCI-UA.0310-004/005 Basic Algorithms February 18, 2015

Solutions to Problem 1 of Homework 3 (10 (+6) Points)

Name: Jason Yao *Due: Wednesday, February 18*

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 in 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.)

Solution:

```
makeMoniesWrapper(A)
  RETURN makeMonies(A, 1, A.length)
END makeMoniesWrapper

makeMonies(A, i, j)
  // Base case: lines crossed, must buy before selling
  if (j < i)
    RETURN 0
  endif
  mid = i + (j-i)/2
  maxLeft = makeMonies(A, i, mid)
  maxRight = makeMonies(A, mid + 1, j)
  α = min(A[1:mid])
```

```
  β = max(A[mid + 1: j])

  RETURN (maxLeft, maxRight, α, β)
END makeMonies
```

The conquer step takes $f(n) = \Theta(n)$, since it needs to find α and β, which should take 2n time, or $\Theta(n)$.

The conquer step works correctly because if you buy and then sell within the first half of array A, then maxLeft and maxRight will find the solution and return it.

If instead the min in in the first half, and the max is in the second half, then α and β that was found will lead to the answer in the form $\beta - \alpha$

□

(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.

Solution: Runtime of my algorithm:

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

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

$T(n) = \Theta(n \log n)$

Runtime of the solution to the Stock Profit problem in the book

$T(n) = \Theta(n \log n)$, so we have matched the efficiency of the book's solution to the Stock Profit problem □

(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?)

Solution:

```
makeMonies(A)
bestMin, bestMax, α = 1
β = A.length/2

For i = 1 to n
   bestMin = min(A[bestMin], A[i])
   bestMax = max(A[bestMax], A[i])
   if (i ≤ n/2 )
      α = min(A[α], A[i])
      β = max(A[β], A[i + A.length/2])
   endif
endfor

RETURN (bestMin, bestMax, α, β)
END makeMonies
```

□

CSCI-UA.0310-004/005 Basic Algorithms February 18, 2015

Solutions to Problem 2 of Homework 3 (10 Points)

Name: Jason Yao *Due: Wednesday, February 18*

A *local minimum* of an array $A[1], \ldots, A[n]$ is an index $i \in \{1, \ldots, n\}$ such that either (a) $i = 1$ and $A[1] \leq A[2]$; or (b) $i = n$ and $A[n] \leq A[n-1]$; or (c) $1 < i < n$ and $A[i] \leq A[i-1]$ and $A[i] \leq A[i+1]$. Note that every array has at least (and possibly more than) one local minimum, since the "global" minumum of the entire array is also a local minimum. Design an $O(\log n)$ divide-and-conquer algorithm to find some local minimum of a given (unsorted) array A of size n.
(**Hint**: Think of binary search for inspiration.)

Solution:

```
findLocalMinWrapper(A)
  RETURN findLocalMin(A, 1, A.length, A.length)
END findLocalMinWrapper

findLocalMin(A, i, j, n)
  // Base case 1: Lines crossed, returns null value
  if (i == j)
    RETURN 0
  endif
  mid = i + (j-i)/2
  //Base case 2: Edge cases
  if (
    ((mid == 1) && (A[mid] ≤ A[mid + 1])) ||
    ((mid == n) && (A[mid] ≤ A[mid - 1])))
    RETURN mid
  endif
  // Base case 3: Local min found
  if (
    (A[mid] ≤ A[mid - 1]) &&
    (A[mid] ≤ A[mid + 1])
    RETURN mid
  endif
  // Recursive Steps
  p = findLocalMin(A, i, mid, n)
  if (p != 0)
    RETURN p
  endif
  q = findLocalMin(A, mid + 1, j, n)
  if (q != 0)
    RETURN q
  endif
```

```
  else
    RETURN 0
  endelse
END findLocalMin
```

□

CSCI-UA.0310-004/005 Basic Algorithms February 18, 2015

Solutions to Problem 3 of Homework 3 (10 Points)

Name: Jason Yao *Due: Wednesday, February 18*

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 := ...
4     If A[i] = ... Return ...
5     If A[i] < ... Return IdFind(A, ..., ...)
6     Return IdFind(A, ..., ...)
```

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

Solution:

```
1 IdFind(A, j, k)
2     If j > k Return FALSE
3     Set i := j + (k−j)/2
4     If A[i] = i Return TRUE
5     If A[i] < i Return IdFind(A, i + 1, k)
6     Return IdFind(A, j, i)
```

□

(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".)

Solution:

Input: Array A of elements in sorted order

Output: boolean- TRUE or FALSE

Correctness: Proving I → O

Part 1: Output check

Case proofs:

If there is no element found, returns false

If there an element is found, returns true

If the element is lower, call the method again with the lower half of the sub-array, returns a boolean

If the element is higher, call the method again with the upper half of the sub array, returns a boolean

Part 2: Algorithm Terminates

As all four potential cases returns a value, the algorithm thus terminates correctly, as it returns a boolean in each case

Running Time:

$T(n) = 2T(\frac{n}{2}) + 1$

$T(n) = \Theta(\log n)$ □

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

Solution: The algorithm does not work if there is repeated elements in A, since the duplicated element in A would cause a shift towards the right in all subsequent elements, such that eventually it would return false.

i.e. the array A = 1,2,2,5,6,7, in which the algorithm would return false, since it would finish searching the right side, and then return false, even though i = 2 would be the normally correct answer □

CSCI-UA.0310-004/005 Basic Algorithms February 18, 2015

Solutions to Problem 4 of Homework 3 (10 Points)

Name: Jason Yao *Due: Wednesday, February 18*

We say that an array A is *c-nice*, where c is a constant (think 100), if for all $1 \leq i, j \leq n$, such that $j - i \geq c$, we have that $A[i] \leq A[j]$. For example, 1-nice array is already sorted. In this problem we will sort such c-nice A using Insertion Sort and QuickSort, and compare the results.

(a) (4 Points) In asymptotic notation (remember, c is a constant) what is the worst-case running time of Insertion Sort on a c-nice array? Be sure to justify your answer.

Solution:

As C is a relatively small number in an n-element array A, that means that there are a greater number of indicies i and j that are already in the correct relative position, indicating that Insertion Sort will not have to do many swaps, since it's mostly sorted.

Thus the worst case running time of insertion sort in this instance is in linear time, or $T_n = \Theta(n)$ □

(b) (5 Points) In asymptotic notation (remember, c is a constant) what is the worst-case running time of QuickSort on a c-nice array? Be sure to justify your answer.

Solution:

As C is a relatively small number in an n-element array A, that means that there are a greater number of indicies i and j that are already in the correct relative position, indicating that Quicksort will have to go through a basically pre-sorted array, causing it's worst case to occur, and for it to sort it in $T_n = \Theta(n^2)$ □

(c) (1 Points) Which algorithm would you prefer?

Solution:

Due to the fact that the array A is near pre-sorted, and is thus Insertion Sort's best case scenario, and Quicksort's worst case scenario, I would prefer to use Insertion Sort in this instance. □