CS140 - Assignment 2

Due: Sunday, Feb. 4 at 10pm

Munir Vafai, Collins Kariuki

Feb 3, 2024

This problem set should be done with a *different* partner than you worked with last week. If you would like help finding a partner, reach out ASAP (not later than Thursday).

You must use LaTeX to format your solution; one person should upload the pdf

1. **[5 points]** Algorithm A has a running time described by the recurrence $T(n) = 7T(n/2) + n^2$. A competing algorithm B has a running time described by the recurrence $T(n) = aT(n/4) + n^2$. What is the largest integer value for a such that B is asymptotically faster than A? Explain your answer. (Hint: use the master method.)

   Using the master method on algorithm A, we find: $a = 7$, $b = 2$, and $f(n) = n^2$. Now we have to determine which case our $f(n)$ falls under:

   - $f(n)$ is $O(n^{\log_b a-\varepsilon})$ for some $\varepsilon > 0$,
   - $f(n)$ is $\Theta(n^{\log_b a})$,
   - $f(n)$ is $\Omega(n^{\log_b a+\varepsilon})$ for some $\varepsilon > 0$, and if $af(n/b) \leq kf(n)$ for some constant $k < 1$ and sufficiently large n, then $T(n) = \Theta(f(n))$.

   $\log_b a = \log_2 7$ and we find that $f(n)$ comfortably falls under case 1, n^2 is $O(n^{\log_2 7-\varepsilon})$ for some $\varepsilon > 0$ and the recurrence relation is $\Theta(n^{\log_2 7})$.

   Using the master method on algorithm B: $a = a$, $b = 4$, and $f(n) = n^2$.

   Using the master method on algorithm A, we find: $a = a$, $b = 4$, and $f(n) = n^2$. Now we have to determine which case our $f(n)$ falls under. $\log_b a = \log_4 a$ and thus assuming $f(n)$ falls under case 2, we have to compare $\Theta(n^{\log_4 a})$ (for B) with $\Theta(n^{\log_2 7})$ (for A).

Comparing the exponents we find:

$$\begin{aligned}
\log_2 7 &\geq \log_4 a \\
\log_2 7 &\geq \frac{\log_2 a}{\log_2 4} \\
\log_2 7 &\geq \frac{\log_2 a}{2} \\
2\log_2 7 &\geq \log_2 a \\
\log_2 a &\leq 2\log_2 7 \\
2^{\log_2 a} &\leq 2^{2\log_2 7} \\
a &\leq (2^{\log_2 7})^2 \\
a &\leq 7^2 \\
a &\leq 49
\end{aligned}$$

2. **[10 points]** 3-part sort

Consider the following sorting algorithm: First sort the first two-thirds of the elements in the array. Next sort the last two thirds of the array. Finally, sort the first two thirds again. Notice that this algorithm does not allocate any extra memory; all the sorting is done inside array A. Here's the code:

```
ThreeSort ($A$,$i$,$j$)
    **if** $A[i] > A[j]$
        **swap** $A[i]$ and $A[j]$
    **if** $i + 1 \geq j$
        **return**
    $k = \lfloor (j - i + 1)/3 \rfloor$.
    ThreeSort($A, i, j - k$)        **Comment:** Sort first two-thirds.
    ThreeSort($A, i + k, j$)        **Comment:** Sort last two thirds.
    ThreeSort($A, i, j - k$)        **Comment:** Sort first two-thirds again!
```

(a) Give an informal but convincing explanation (not a rigorous proof by induction) of why the approach of sorting the first two-thirds of the array, then sorting the last two-thirds of the array, and then sorting again the first two-thirds of the array yields a sorted array. A few well-chosen sentences should suffice here.

When we sort the first 2/3 of the array, the largest elements of that sub array will end up in the middle third of the total array. Then, when we sort the last 2/3 of the array, the final third is guaranteed to be in the final sorted order (i.e., containing the largest elements). Then, sorting the first 2/3 again will result in a fully sorted list.

(b) Find a recurrence relation for the worst-case running time of ThreeSort.

$T(n) = 3T(\frac{2}{3}n) + n$

(c) Solve the recurrence relation.

$a = 3, b = \frac{3}{2}, f(n) = n$

$n^{log_b a} = n^{\log_{\frac{3}{2}} 3}$

In this case, $n = O(n^{\log_{\frac{3}{2}} 3 - \epsilon})$ so T(n) is $\theta(n^{\log_{\frac{3}{2}} 3})$.

(d) How does the worst-case running time of ThreeSort compare with the worst-case running times of Insertion Sort, Selection Sort, and Mergesort?

Insertion sort and selection sort have a worst-case running time of $O(n^2)$. Merge sort has a a worst-case running time of $O(n \log n)$.

$\theta(n^{\log_{\frac{3}{2}} 3})$ is roughly $\theta(n^{2.7})$, so it is worse than all of them.

3. **[7 points]** Halfsies

In some situations, there is not a natural ordering to the data so we can't sort it, but we can check equality (e.g. images). Given an array of elements A, we would like to determine if there exists a value that occurs in more than half of the entries of the array. If so, return that value, otherwise, return *null*. Assume you can only check equality of elements in the array which takes time $O(1)$.

Describe a divide-and-conquer algorithm whose running time is asymptotically better than $\Theta(n^2)$. Provide pseudocode and/or a clear English description of your algorithm and state the worst case running time of your solution (in terms of Θ or O where appropriate) *with justification.*

In this algorithm, we recursively split the array, finding the majority label:

```
findMajority($A$, $start$, $end$):
    if $start == end$
        return $A[start]$
    $mid = (start + end)/2$
    $leftMajority$ = findMajority($A$, $start$, $mid$)
    $rightMajority$ = findMajority($A$, $mid + 1$, $end$)
    if $leftMajority == rightMajority$
            return $leftMajority$
    $leftCount$ = countOccurrences($A$, $start$, $end$, $leftMajority$)
    $rightCount$ = countOccurrences($A$, $start$, $end$, $rightMajority$)
    $majorityCount = (end - start + 1)/2$
    if $leftCount > majorityCount$
            return $leftMajority$
    if $rightCount > majorityCount$
            return $rightMajority$
    return null

countOccurrences($A$, $start$, $end$, $element$):
    $count = 0$
    for $i = start$ to $end$
            if $A[i] == element$
            $count += 1$
    return $count$
```

The divide step cuts the array in half, resulting in a depth of log n. At each level of recursion, we perform two counts over the array, which takes O(n) in the worst case.

Therefore, the running time is $\theta(n \log n)$.

4. **[10 points]** Given a *sorted* list of unique integers A[1...n], determine if an entry exists such that $A[i] = i$. If an entry exists, return the index, otherwise, return *null*. (Hint: You can do better than $O(n)$. Think divide-and-conquer.) Write **pseudocode** to solve this problem and and *state the worse case running time* (in terms of Θ or O where appropriate). You will be graded on the efficiency of your solutions.

```
findPoint($A$, $start$, $end$):
    **if** $end < start$:
        **return** null
    $mid = \frac{start+end}{2}$
    **if** $A[mid] == mid$:
        **return** $mid$
    **elif** $A[mid] < mid$:
        findPoint($A$, $mid + 1$, $end$)
    **elif** $A[mid] > mid$:
        findPoint($A$, $start$, $mid - 1$)
```

The worst-case running time is $\Theta(\log\ n)$. This is because, similar to binary search, each recursive call reduces the search space by half, leading to a logarithmic time complexity.

5. **[25 points]** Stock Market Problem

You're given an array of numbers representing a stock's prices over n days. You goal is to identify the longest consecutive number of days during which the stock's value does not decrease. For example, consider the stock values below:

```
Day:      1   2   3   4   5   6   7   8
Value:    42  40  45  45  44  43  50  49
```

In this example, the length of the longest consecutive non-decreasing run is 3. This run goes from day 2 to day 4.

(a) Briefly describe a very simple "naive" algorithm for this problem and explain why the worst-case running time is $\Theta(n^2)$.

One might initiate a variable named "result" to hold the integer outcome. Next, two pointers are set up: a left pointer at the array's beginning and a right pointer at the array's second element. The right pointer is advanced as long as its corresponding element is greater than or equal to the element at the left pointer. Upon finding an element at the right pointer smaller than at the left, the difference in their positions (r-l) is calculated, updating 'result' to the greater of itself or this new difference.

For subsequent comparisons, the left pointer is moved one step to the right, with the right pointer positioned immediately after the left. This process is repeated, mimicking the initial steps, until reaching the array's penultimate element. This methodical examination across the array for each element explains the $\Theta(n^2)$ time complexity. We finally return "result".

(b) Describe a divide-and-conquer algorithm whose running time is asymptotically better than $\Theta(n^2)$. Provide pseudocode and/or a clear English description of your algorithm. (Note that your algorithm must be a divide-and-conquer algorithm. And yes there are non-divide-and-conquer algorithms that are very, very good!)

Hint: Like writing recursive functions, when trying to come up with a divide-and-conquer solution, assume that your algorithm works correctly on the divided parts. Then, how do you construct your answer to the overall solution?

The algorithm splits the array, recursively identifies the longest non-decreasing sequences in both halves, and merges them, including any sequence spanning across the divide, to find the overall longest sequence.

```
stockMarket($A$, $start$, $end$)
    **if** $start == end$
        **return** 1
    $mid = (start + end)/2$
    $leftSide$ = stockMarket($A$, $start$, $mid$)
    $rightSide$ = stockMarket($A$, $mid + 1$, $end$)
    $middle$ = findMid($A$, $start$, $mid$, $end$)
    **return** max($leftSide, rightSide, middle$)


findMid($A$, $start$, $mid$, $end$)
$leftLength = 1$
    **for** $i = mid$ **to** $start$
                **if** $i > start$ **and** $A[i] \geq A[i-1]$
                    $leftLength += 1$
                **else**
                    **break**
$rightLength = 1$
    **for** $i = mid + 1$ **to** $end$
                    **if** $i < end$ **and** $A[i] \geq A[i-1]$
                        $rightLength += 1$
                    **else**
                        **break**
**if** $A[mid] \leq A[mid + 1]$
    **return** $leftLength + rightLength - 1$ // **Subtract 1 because mid is counted twice**
**else**
    **return** max($leftLength, rightLength$)
```

(c) Analyze the running time of your algorithm: what are tight bounds on the best and worst case behavior?

The algorithm divides the input array into two halves, recursively finds the longest non-decreasing run in each half, and then combines these results. The combination step also involves checking the longest non-decreasing sequence that could span the middle of the array. Thus, the recurrence relation is: $T(n) = 2T\left(\frac{n}{2}\right) + O(n)$

By the master method, the running time is $\theta(n \log n)$.

(Note that next week's assignment will ask you to implement your divide-and-conquer algorithm.)