CS255 (section 2)
Design and Analysis of Algorithms
San Jose State University

Midterm 1, 03/Mar/2014, duration: 1h15m

Read all questions before starting the midterm. When you're asked to specify an algorithm, you can use pseudocode, C, or Java.

(25 points) Question 1.

```
MISTERY($n$)
1  $sum = 0$
2  for $i = 1$ to $n$
3      for $j = i + 1$ to 50000
4          $sum = sum + F(i, j)$
5  return $sum$
```

Consider the algorithm MISTERY described above. The pseudocode for function $F(i, j)$ is not given but you can assume that $F(i, j)$ has running time $\Theta(1)$. What's the running time of MISTERY as a function of n using Θ notation. Justify your answer.

Answer

Lines 1 and 5 take constant time. The inner for loop in lines 3-4 runs in constant time (it doesn't depend on the input size n.) The for loop on line 2 executes n iterations, so the algorithm's running time is $\Theta(n)$

(25 points) Question 2.

Consider the following variation of *MergeSort* for arrays of very large size n. Instead of doing recursive calls until the input size is sufficiently small, we only do recursive calls up to a constant depth d, and then we use *InsertionSort* to sort each of the 2^d resulting subproblems. What's the running time of this variation of MergeSort? Justify your answer. (Note that n can grow arbitrarily, but d is a constant.)

Answer

Expanding the recursion tree yields:

level 0: 1 problem of size n
level 1: 2 subproblems of size $n/2$
level 2: 4 subproblems of size $n/4$
...
level d: 2^d subproblems of size $n/2^d$

The recursion stops at depth d. From here on we have to solve each of the 2^d subproblems with *InsertionSort*. Each subproblem has size $\frac{n}{2^d}$. *InsertionSort* has a quadratic running time. Thus, the running time for the 2^d subproblems is:

$$2^d \cdot \Theta\left(\left(\frac{n}{2^d}\right)^2\right) = \Theta(n^2) \quad \text{, because if } d \text{ is constant, so is } 2^d$$

We still need to add the time to solve the subproblems of levels 0, 1, ..., $d-1$. Each such level contributes with $c \cdot n$, with c being some positive constant. Therefore, the total for levels 0, 1, ..., $d-1$ is $d \cdot c \cdot n$, which is $\Theta(n)$. And the total for level d is $\Theta(n^2)$.

In summary, the total running time is dominated by the execution of *InsertionSort* and is $\Theta(n^2)$.

(25 points) Question 3.

Specify a divide and conquer algorithm that takes as input an array of n integers and determines if all those n integers have the same value. If yes, your algorithm should return TRUE, otherwise it should return FALSE.

Write a recurrence for the worst case running time of your algorithm, and solve it.

Answer

If the array has 1 element return TRUE. If it has more than 1 element, split the array in the middle and recursively solve the problem for the left and right subarrays. If all the elements in the left subarray have the same value, and all the elements in the right subarray have the same value, and if one of the elements from the left subarray has the same value as one of the elements of the right subarray, then the entire input array has all the elements with the same value. Otherwise, not all elements have the same value and the algorithm should return FALSE.

```
AllEqual($A, l, r$)
    if $l == r$        // Base case, 1-element array
        return TRUE
    else $mid = (l + r)/2$
        return AllEqual($A, l, mid$) and AllEqual($A, mid + 1, r$) and $(A[l] == A[r])$
```

Initial call: AllEqual$(A, 1, n)$

Recurrence: $T(n) = 2 \cdot T(n/2) + \Theta(1)$.

Solution: Applying the Master method ($a = 2$, $b = 2$, $f(n) = c$) we can see that we are in case 1 because $n^{\log_b a} = n^{\lg 2} = n$, is polynomially greater than a constant c. $f(n) = c = c \cdot n^0 = O(n^{1-\epsilon})$ for some $0 < \epsilon < 1$. Therefore, $T(n) = \Theta(n)$.

(25 points) Question 4.

You are given an array A of n positive integers. All but 20 elements of array A are within the range $[6, 6n]$ Design an algorithm to sort the array A in O(n) time in the worst case. There's no need to use pseudocode. You can express your solution in plain english referring (if needed) to algorithms we've seen in class.

Answer

If it was not for those 20 elements outside of the range $[6, 6n]$ we could simply apply *Counting Sort* with $k = 6n$. Recall that Counting Sort runs in $\Theta(n + k)$ time. With $k = 6n$ it runs in $\Theta(n)$ time. We just need to find a way to handle those 20 elements outside of the range $[6, 6n]$. Here's a solution.

1. Create three arrays, L and R with maximum length 20, and B with length $n - 20$.
2. Scan the input array A from left to right and copy the elements whose value is less than 6 into array L, copy the elements whose value is greater than $6n$ into array R, and copy the other elements into array B. This operation is clearly $\Theta(n)$.
3. Sort L and R with insertion sort (or with any sorting algorithm.) This takes $\Theta(1)$ time because the size of these arrays is at most 20, a constant.
4. Sort array B using Counting Sort. All elements in B are integers in the range $[6, 6n]$, so Counting Sort takes $\Theta(n)$ time.
5. Concatenate the arrays L, B, R, in this order, to produce the sorted array. This takes $\Theta(n)$ time.

The total running time is the sum of running time of these steps, which is $\Theta(n)$.