CSCI 532 Homework 2

Brittany Boles, Redempta Manzi, Madie Munro

September 17th, 2024

Question 1 (2 marks): Suppose you are choosing between the following three algorithms:

1. Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.
2. Algorithm B solves problems of size n by recursively solving two subproblems of size $n-1$ and then combining the solutions in constant time (Master theorem does not apply here, so solve by hand)
3. Algorithm C solves problems of size n by dividing them into nine subproblems of size $\frac{n}{3}$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

What are the running times of each of these algorithms (in big-O notation), and which would you choose?

Answer:

1. Recurrence Relation: $T(n) = 5T(\frac{n}{2}) + O(n)$
   We apply the Master Theorem where $a = 5$, $b = 2$, and $f(n) = O(n)$.
   We can apply case 1 of the Master Theorem since $f(n) = n^{log_2(5)} = n^{2.32-\epsilon}$ where $\epsilon = 1.32$.
   $\therefore$ Algorithm A RT: $\Theta(n^{\lg 5})$
2. Recurrence Relation: $T(n) = 2T(n-1) + O(1)$
   We expand the recurrence relation:

$$
\begin{aligned}
T(n) &= 2T(n-1) + O(1) \\
T(n-1) &= 2T(n-2) + O(1) = 2(2T(n-1) + O(1)) + O(1) \\
T(n-2) &= 2T(n-3) + O(1) = 2(2(2T(n-1) + O(1)) + O(1)) + O(1) \\
&\dots \\
T(n) &= 2^k T(n-k) + \sum_{i=1}^{k-1} 2^i O(1)
\end{aligned}
$$

$\therefore$ Algorithm B RT: $O(2^n)$

3. Recurrence Relation: $T(n) = 9T(\frac{n}{3}) + O(n^2)$
   We apply the Master Theorem where $a = 9$, $b = 3$, and $f(n) = O(n^2)$.
   We can apply case 2 of the Master Theorem since $f(n) = n^{log_3(9)} = n^2$.
   $\therefore$ Algorithm C RT: $\Theta(n^2 \lg n)$

Based on these results, we would pick Algorithm A.

Question 2 (2 marks): Show that an array of integers $A[1...n]$ can be sorted in $O(n+D)$ time, where $D = \max_i(A[i]) - \min_i(A[i])$ (you may use some extra storage).

Answer: We implement an algorithm that similar to bucket sort:

Algorithm 1 MODIFIED BUCKET SORT(A)

Require: A, an array of positive integers of size n
Ensure: A_{SORTED}, array A sorted
  $n \leftarrow A.size$
  CALCULATE THE MIN$A[i]$ VALUE $\min_i(A[i])$
  CALCULATE THE MAX$A[i]$ VALUE $\max_i(A[i])$
  $D \leftarrow \max_i(A[i]) - \min_i(A[i])$
  INITIALIZE A BUCKET ARRAY A_B OF SIZE D
  INITIALIZE AN ARRAY A_{SORTED} OF SIZE n
  for $i = 1$ TO n **do**
    $A[i] \leftarrow A[i] - \min_i(A[i])$ {Since D was shifted, we need to shift each element in A}
  end for
  for a IN A **do**
    MAP UNIQUE ELEMENTS $a \in A$ AS INDICES IN A_B
    INCREMENT BY ONE THE OCCURRENCE OF EACH a AT THEIR RESPECTIVE INDICES IN A_B
  end for
  for $i = 1$ TO D **do**
    CALCULATE THE PREFIX SUM OF EACH BUCKET INDEX i IN A_B
  end for
  for $i = n$ TO $1, i--$ **do**
    $A[i] \leftarrow A[i] + \min_i(A[i])$
    REBUILD A_{SORTED} AT THE INDEX CORRESPONDING TO $A_B[i]$ WHERE $i = A[i]$
    DECREMENT BY ONE EACH ENTRY IN A_B AT $i = A[i]$
  end for
  return A_{SORTED}

Calculating the max and min take $O(n)$ time, and each for loop in the algorithm executes in either $O(n)$ time (for array A) or $O(D)$ time (for array A_B) for a total runtime of $O(n + D)$. The remaining calculations can be executed at constant time. Space complexity here doesn't matter since we are allowed to use extra storage.

Question 3 (2 marks): You are interested in analyzing some remote data from two separate databases. Each database contains n numerical values-—so there are $2n$ values total-—and you may assume that no two values are the same. You'd like to determine the median of this set of $2n$ values, which we will define here to be the nth smallest value. However, the only way you can access these values is through queries to the databases. In a single query, you can specify a value k to one of the two databases, and the chosen database will return the kth smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

1. Devise an algorithm that finds the median value using at most $O(\log n)$ queries. Explain why your algorithm achieves this query bound.

   Answer: We can implement an algorithm similar to a binary search in order to find the nth smallest median value between the two databases. Basically, we divide each list of values for each database around a median index, k, and recursively find the nth smallest value by comparing each partitioned database A and B at k, which represents a temporary median for the two databases. If the current median of A is less than the current median of B, then we only need to search the next $k+1$ values of A for the smallest median, and we can ignore the $k+1$th values in B since they are also greater than the current median value, but we search the first k values in B. The same logic is applied if B's current median is less than A's current median. This is how we make $O(\log n)$ queries since we reduce the size of each database in half as we recursively find the median of the two databases. Here is the pseudocode for this algorithm:

Algorithm 2 SMALLEST MEDIAN VALUE(A, B)

Require: A, B, TWO DATABASES EACH CONTAINING n VALUES (SORTED)
Ensure: THE SMALLEST MEDIAN VALUE BETWEEN THE TWO DATABASES

```
n ← A.size
if n == 1 then
    return min(A[n], B[n])
end if
k ← ⌊n/2⌋
if A[k] < B[k] then
    SMALLEST MEDIAN VALUE(A[k + 1..n], B[1..n − k])
else
    SMALLEST MEDIAN VALUE(A[1..n − k], B[k + 1..n])
end if
```

2. Write a small program to test the method that you devised. You may assume that the databases are just two sorted arrays and you want to make $O(\log n)$ accesses to those arrays. Demonstrate it running on an example.

```r
# calculate the nth smallest median between two arrays of the same size

smallest_median_value <- function(A, B) {
  n <- length(A)
  if (n == 1) {
        return(min(A[n], B[n]))  #base case
    }

    k <- floor(n/2) #starting median

    if (A[k] < B[k]) {
        return(smallest_median_value(A[(k+1):n], B[1:(n-k)]))
    } else {
        return(smallest_median_value(A[1:(n-k)], B[(k+1):n]))
    }
}
```

Figure 1: R code to find smallest median value between two arrays

```r
A <- c(3, 5, 7, 9)
B <- c(2, 4, 6, 8)

smallest_median_value(A, B)

A <- c(6, 8, 13, 15, 19)
B <- c(3, 7, 10, 11, 16)

smallest_median_value(A, B)

A <- c(1,4,8,12,20,25,50,65,67,79)
B <- c(2,3,5,19,21,28,40,59,70,80)

smallest_median_value(A, B)
```

Figure 2: Example arrays to test the algorithm

```
>
> A <- c(3, 5, 7, 9)
> B <- c(2, 4, 6, 8)
>
> smallest_median_value(A, B)
[1] 5
>
> A <- c(6, 8, 13, 15, 19)
> B <- c(3, 7, 10, 11, 16)
>
> smallest_median_value(A, B)
[1] 10
>
> A <- c(1,4,8,12,20,25,50,65,67,79)
> B <- c(2,3,5,19,21,28,40,59,70,80)
>
> smallest_median_value(A, B)
[1] 21
>
```

Figure 3: Smallest median calculated for the three examples