Homework 3

Joni Vrapi

10/22/2022

Statement of Integrity: I, Joni Vrapi, attempted to answer each question honestly and to the best of my abilities. I cited any, and all, help that I received in completing this assignment.

Problem 1. Since this algorithm is recursive, we can prove its time complexity by first writing a recurrence relation that describes it, and then solving that recurrence relation. Recurrence relations take the form

$$T(n) = aT(\frac{n}{b}) + f(n) \tag{1}$$

where a is the number of recursive calls, $\frac{n}{b}$ is the size of each sub problem, and $f(n)$ is the non-recursive work done on each call. In this function, there are 2 recursive calls of size $\frac{n}{2}$ and a constant amount of work done at each call, therefore, we can write the recurrence relation for this problem as

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

Solving this recurrence relation by the Master theorem [1], we get $n^{log_b a} = n$ is asymptotically larger than a constant factor, so case 1 of the Master theorem gives

$$T(n) = \Theta(n^{log_b a}) = \Theta(n) \tag{3}$$

Problem 2a. An AVL tree [2] was the first type of balanced binary search tree invented. Each node has at most 2 children, where the left child is less than the parent, which is less than the right child.

```
AVL-Search(Node N, a, b)
 1  if N == null
 2       return false
 3  if N.x ≥ a and N.x ≤ b
 4       return true
 5  else
 6       if b < N.x
 7            return AVL-Search(N.left, a, b)
 8       if a > N.x
 9            return AVL-Search(N.right, a, b)
10
```

Problem 2b. Since, at every level of the tree that is being searched, we eliminate half of the nodes, we check only $log(n)$ nodes, giving this algorithm a time complexity of $O(log(n))$.

Problem 2c. This algorithm will recurse through the nodes of the tree until it reaches the bottom, where if it does not find a node that meets the condition set on line 3 of the pseudocode above, the condition on line 1 is triggered. It then returns false as requested in part A of this problem.

Problem 3. If we were to re-write COUNTING-SORT as described, it would still work properly (in that we would still get a sorted array), except that like elements taken later would receive a higher index than those taken earlier, making it unstable. Take the example of $A = [1_1, 2, 1_2]$:

- $n = 3$ and $k = 2$ in this example.
- For loop on line 2 initializes C with all zeros.
- For loop on line 4 transforms C to $[0, 2, 1]$.
- For loop on line 7 transforms C to be $[0, 2, 3]$.
- For loop on line 11 transforms B to be $[0, 1_1, 0] \implies [0, 1_1, 2] \implies [1_2, 1_1, 2]$.

As you can see, the modified algorithm is not stable because the 1_1 and 1_2 are out of order relative how they showed up in A originally. Rewriting the modified algorithm like so would maintain the proper index ordering:

```
COUNTING-SORT(A, n, k)
 1  initialize B[1: n] and C[0: k] as new arrays
 2  for i = 0 to k
 3        C[i] = 0
 4  for j = 1 to n
 5        C[A[j]] = C[A[j]] + 1
 6  for i = 1 to k
 7        C[i] = C[i] + C[i - 1]
 8  for j = n to 1
 9        B[C[A[j]]] = A[j]
10        C[A[j]] = C[A[j]] - 1
11  return B
```

Problem 4. For this problem, I was not sure how to begin, so I referenced [3].

Problem 4a. The algorithm will still work in groups of 7 because we know that the median of medians is less than at least 4 elements from half of the $\lceil \frac{n}{7} \rceil$ groups, so it is greater than, as well as less than, $\frac{4n}{14}$ of the elements. Therefore, it is never being called recursively on more than $\frac{10n}{14}$ elements. So we have $T(n) \leq T(\frac{n}{7}) + T(\frac{10n}{14}) + O(n)$. By substitution, we can show linearity. If $T(n) < cn$ for $n < k$, then for $m \geq k$, $T(\frac{m}{7}) + T(\frac{10m}{14}) + O(m) \leq cm(\frac{1}{7} + \frac{10}{14}) + O(m)$. Basically, what this says is that as long as the constant that is inside the Big-Oh notation is less than $\frac{c}{7}$, this will run linearly.

Problem 4b. Taking the same approach for groups of 3, we have the recurrence $T(n) = T(\lceil \frac{n}{3} \rceil) + T(\frac{4n}{6}) + O(n) \geq T(\frac{n}{3}) + T(\frac{2n}{3}) + O(n)$. To show it is $\geq cnlog(n)$ we have $T(m) \geq c(\frac{m}{3})log(\frac{m}{3}) + c(\frac{2m}{3})log(\frac{2m}{3}) + O(m) \geq cmlog(m) + O(m)$, so it grows more quickly than $O(n)$.

Problem 5. If we define $OPTIMAL(i)$ [4] [5] as the minimum cost of a solution for weeks 1 through i, we can say that, in the ith week, we either use company A or company B. To define this recursively, we know that when using company A, we pay $r * ws_i$ and are optimal through week $i-1$, and when using company B, we pay $4c$ and are optimal through week $i-4$. We can then define the

recursion as $OPTIMAL(i) = min(r * ws_i + OPTIMAL(i-1), 4c + OPTIMAL(i-4))$. Finally, for $i < 4$ we have to choose $OPTIMAL(0) = 0$ and $OPTIMAL(i) = r * ws_i + OPTIMAL(i)$.

The following algorithm will run in $O(n^2)$ time as it takes a constant amount of time to compute $OPTIMAL(i)$, and since we use a 2d array to store the most efficient schedule which requires n^2 iterations, we result in an $O(n^2)$ time.

```
SHIPPING-SCHEDULE(A, r, c)
 1  SCHEDULE[1...A.length][1...A.length]
 2  OPTIMAL[0...A.length]
 3
 4  OPTIMAL[0] = 0
 5  for i from 1 to min(A.length, 3)
 6       OPTIMAL[i] = OPTIMAL[i-1] + r*A[i]
 7
 8       for j from 1 to i - 1
 9            SCHEDULE[i][j] = SCHEDULE[i-1][j]
10       SCHEDULE[i][i] = 'A'
11
12  for i from 4 to A.length
13       OPTIMAL[i] = min(OPTIMAL[i-1] + r*A[i], OPTIMAL[i-4] + 4c)
14
15       if OPTIMAL[i-1] + r*A[i] ¡ OPTIMAL[i-4] + 4c
16            for j from 1 to i - 1
17                 SCHEDULE[i][j] = SCHEDULE[i-1][j]
18
19            SCHEDULE[i][i] = 'A'
20       else
21            for j from 1 to i-4
22                 SCHEDULE[i][j] = SCHEDULE[i-4][j]
23
24            SCHEDULE[i][i-3] = SCHEDULE[i][i-2] = SCHEDULE[i][i-1] = SCHEDULE[i][i] = 'B'
```

Problem 6.

Problem 7a. If we say $W = \{1, 2, 2, 1\}$ and $K = 2$, we can see by this algorithm that there will be 4 trucks needed to haul away these containers. This is due to the fact that the example greedy algorithm simply takes the first available container and puts it on the first available truck, filling them up in-order as much as possible, before moving on to a new truck. However, it is obvious looking at the set W that if the containers were instead ordered like $W = \{1, 1, 2, 2\}$, we would only need 3 trucks to move all the containers.

Problem 7b. If we define the total weights of all containers as $\sum_{i=1}^{n} w_i$, then the number of trucks needed to carry all these containers is

$$\lceil \frac{\sum_{i=1}^{n} w_i}{K} \rceil \tag{4}$$

The example above, $W = \{1, 2, 2, 1\}$ and $K = 2$, requires 4 trucks using the algorithm, and 3 at a minimum. Likewise, we can sort it to $W = \{1, 1, 2, 2\}$ and we require only 3 trucks from the algorithm, and 3 at a minimum. If we had $W = \{1, 2, 1, 2, 1\}$ and $K = 2$, we would need 5 trucks per the algorithm, and 4 trucks at a minimum. From (4) we see that this is always going to be within 1 of the minimum number of trucks, and that falls within a factor of 2, answering this question.

References

[1] "Master theorem." https://brilliant.org/wiki/master-theorem/. Accessed on 2022-10-22.

[2] "Avl tree." https://brilliant.org/wiki/avl-tree/. Accessed on 2022-10-22.

[3] "Rutgers math." https://sites.math.rutgers.edu/~ajl213/CLRS/Ch9.pdf. Accessed on 2022-10-22.

[4] "Dynamic programming." https://web.pdx.edu/~arhodes/alg7.pdf. Accessed on 2022-10-22.

[5] "Cs 4020 2011." https://pdfcoffee.com/csci-4020-computer-algorithms-pdf-free.html. Accessed on 2022-10-22.