RENSSELAER POLYTECHNIC INSTITUTE

Homework 4

Student name: *Jiawei Wu*

Course: *Introduction to Algorithms (CSCI 2300)* – Professor: *Dr. Bulent Yener*
Due date: *Feb 10, 2022*

Question 1

Computer tight big-Oh bounds for the following recurrences:

(a) $T(n) = 8T(\frac{n}{4}) + O(n)$

(b) $T(n) = 2T(\frac{n}{4}) + O(\sqrt{n})$

(c) $T(n) = T(n-4) + O(n^2)$

(d) $T(n) = T(\sqrt{n}) + O(n)$

For these questions I used the formulas given to us in lecture with the following cases:

$$T(n) = \Theta(n^d) \text{ if } d > \log_b a$$
$$T(n) = \Theta(n^d logn) \text{ if } d = \log_b a$$
$$T(n) = \Theta(n^{\log_b a}) \text{ if } d < \log_b a$$

Answer.

(a) $a = 8, b = 4, d = 1$
Using theorem, we get $\log_b a = \log_4 8$. We can solve the algorithm by

$$\begin{aligned}\log_4 8 &= log_{2^2} 2^3 \\ &= \frac{1}{2}\log_2 2^3 = \frac{3}{2}\end{aligned}$$

From the theorem, we know that since $d < \log_b a$, it follows that the tight bound is $\Theta(n^{\log_4 8})$ or $\boxed{\Theta(n^{\frac{3}{2}})}$

(b) $a = 2, b = 4, d = \frac{1}{2}$
Solve $\log_b a = \log_4 2$. This solves to $\frac{1}{2}$.
By the theorem, $d = \log_b a$. Therefore, the tight bound is $\boxed{\Theta(n^{\frac{1}{2}} logn)}$

(c) $a = 1, b = 1, d = 2$
Solve $\log_b a = \log_1 1$. This solves to 1.
By the theorem, $d > log_b a$. The tight bound is $\boxed{\Theta(n^2)}$

(d) $a = 1, b = 1, d = 1$ Solve $\log_b a = \log_1 1$. This solves to 1.
By the theorem, $d = \log_b a$. The tight bound is $\boxed{\Theta(nlogn)}$

Question 2

Let A be an array of n integers, and let R be the range of values in A, i.e., $R = \max(A) - \min(A)$. Give an $O(n + R)$ time algorithm to sort all the values in A.

Answer.

```
def sortArr(A):
    R = max(A) - min(A)
    count = [0 for i in range(R + 1)]

    for num in A:
        index = num - min(A)
        count[index] += 1

    sortedA = []
    for i in range(len(count)):
        value = i + min(A)

        for dup in range(count[i]):
            sortedA.append(value)

    return sortedA
```

In this code, I implemented a type of function that utilizes a type of counting hash function. I begin the program by calculating the value of R and initializing an array that holds $R + 1$ values. I then loop through every value in A and determine their index based on the difference between their value and the minimum value. A for loop $O(n)$ complexity is then implemented to add the values into the "hash list". Then, I initialize an array called sortedA that holds the return array for the function.

My next for loop goes over $O(R)$ values, this is because the length of the count array is $R + 1$. For each value, I then add the *min* of the array back to the index to get the original value. Since count[i] stores how many times that value appears. Since it is implemented as a hash function, we can assume the inner for loop is indeed $O(1)$ to go over all the duplicates.

This implementation is based upon the assumption that we don't have an array such as [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1], where in this case, the algorithm wouldn't be very useful in this case because it

Question 3

Let A be an array of n distinct integers. Consider an algorithm to find the minimum value, where we pair up the elements, and retain the smaller of the values from each pair. This will result in an array of half the size (actually the resulting size will be $\lceil n/2 \rceil$. We can then recursively apply the same approach, until we get a final array with just two elements. We compare these two values and return the minimum of those values as the answer. Answer the following questions:

(*a*) How many comparisons are done in the above algorithm in the worst case.

(*b*) Show how to modify/extend this method to find the second smallest element.

(*c*) Prove that we can find the second smallest element in $n + \lceil \log n \rceil - 2$ comparisons in the worst case.

Note that the above questions are not asking for the big-Oh complexity, but rather the (exact) number of comparisons in the worst case. For example, it is easy to find the 2nd smallest element in at most $2n$ comparisons, but that is $n + n$ which is larger than $n + \lceil \log n \rceil - 2$ comparisons.

Answer.

(*a*) The structure of this algorithm is one of a binary tree. Every time we go lower in depth, the array is being split twice. The height of a binary tree increases at $\log n$ compared to the number of nodes. In the worst case, the algorithm must traverse the tree from the root all the way down to the leaf node that contains the smallest element in the array, comparing $\boxed{\log n}$ times in total.

(*b*) We can use $\log n$ to find the smallest element as mentioned above, and then we would have to re-establish the tree form. At most, each node can have a maximum of 3 other connected nodes (parent, 2 child). After finding the smallest one, we would search again using our $\log n$ function for the 2nd smallest element, totaling an algorithm complexity of $\boxed{2\log n + 3}$.