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:

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

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 $\boxed{\Theta(n^{\frac{1}{2}} log n)}$

(*c*) We cannot use the master's theorem for this question, but the recurrence is simple enough that we can analyze it. $T(n-4)$ signifies that the function will run $O(\frac{n}{4})$ times since every time it runs, it decreases by 4. Also, $T(n-4)$ will run $O(n^2)$ times, we know this from the $O(n^2)$ in our equations. Our true O notation will be $O(\frac{n}{4} \times n^2)$, which will solve to $O(\frac{n^3}{4})$. Our answer in big O notation will be $\boxed{\mathbf{\Theta(n^3)}}$.

(*d*) Similarly, we cannot use the master's theorem for this question. The patter that we encounter while trying to solve this problem is represented in the following table:

R	T
1	n
2	$n^{\frac{1}{2}}$
3	$n^{\frac{1}{4}}$
4	$n^{\frac{1}{8}}$
k	$n^{\frac{1}{2^k}}$

From this, we can build the following sigma notation to calculate every value of $T(n)$ up to $n = k$:

$$T(n) = \sum_{i=0}^{k} n^{\frac{1}{2^i}}$$

We can formulate a base case like the following $n^{\frac{1}{2^k}} = 2$, since 2 is the smallest number that is the answer of another number squared. From this, we can continue to solve it:

$$\begin{aligned}
\log_2 n^{\frac{1}{2^k}} &= \log_2 2 \\
\frac{1}{2^k} \log_2 n &= \log_2 2 \\
\frac{1}{2^k} &= \frac{\log_2 2}{\log_2 n} \\
2^k &= \frac{\log_2 n}{\log_2 2} \\
\log_2 2^k &= \log_2(\frac{\log_2 n}{\log_2 2}) \\
k &= \log_2(\log_2 n)
\end{aligned}$$

Having solved k, we can then plug it into our previous sum:

$$T(n) = \sum_{i=0}^{\log\log(n)} n^{\frac{1}{2^i}}$$

From this, we can compare it to our previous value:

$$\sum_{i=0}^{\log\log(n)} n^{\frac{1}{2^i}} < \sum_{i=0}^{\log\log(n)} n$$

We can define $T(n) = \sum_{i=0}^{\log\log(n)} n^{\frac{1}{2^i}}$ as smaller than $T(n) = \sum_{i=0}^{\log\log(n)} n$. So it is our lower bound. Our upper bound will be the bigger value. However, we have to make some key assumption here:

i	value
0	n
1	$\sqrt{n}$
2	$\sqrt{n}$
3	$\sqrt{n}$
4	$\sqrt{n}$
k	$\sqrt{n}$

We assume that every value after 0 is going to take $\sqrt{n}$ runtime. With this assumption, we can define our upper bound as our tight bound. So, let's solve the equation:

$$\begin{aligned} T(n) &= n + \sum_{i=1}^{\log_2 \log_2 n} \sqrt{n} \\ &= n + \log_2 \log_2 n \sqrt{n} \end{aligned}$$

Reaching our true estimate. We will then take the upper bound, and simply assume that every element takes $O(n)$ time to reach the answer of $\boxed{\boldsymbol{\Theta}(\log_{\mathbf{2}} \log_{\mathbf{2}}(\mathbf{n})\mathbf{n})}$. Here we eliminated the constant n in front of the equation, and since we can't establish a bound for $n^{\frac{1}{2^i}}$, we simply assume that it takes n time.

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*) With the following logic: A[i] compares with B[i], for A = Array left, and B = Array right:

```
initial array: 7 9 8 6 2 1 4 5
A and B: 7 9 8 6 | 2 1 4 5
A and B: 2 1 | 4 5
A and B: 2 | 1
Answer: 1
```

The number we chose for $n = 8$ is a power of 2. The total number of comparisons done for powers of two will always be $n - 1$, even in the worst case. However, if we have powers that aren't powers of 2, it will be slightly higher, but still proportional to n.
$\therefore$ the worst case should be $\boxed{\mathbf{n-1}}$ comparisons.

(*b*) We can simply modify this method so that, when it finds the smallest element, it simply removes it from the array. Then, you can run the algorithm again to find the 2nd smallest element.

(*c*) Note: The answer to this question can also be the answer to B. When comparing the smallest element, there has to be a time that the 2nd smallest element was compared to the smallest element. We also know that since this halfing of array size follows a tree pattern, that means that the smallest element has to have traveled $\log n$ depth in our binary tree in order to become the smallest element. In our answer to A, we mentioned that there are $n - 1$ comparisons to find the minimum value. We can here omit the value of comparing in the last step, leading to $n - 2$ comparisons instead. Following this logic, we end up with a total of $n + \lceil \log n \rceil - 2$ comparisons.