Assignment-02-B403-cmtidmar

cmtidmar

February 2021

1 Introduction

Assignment 02 for CSCI-B403 Spring 2021

2 Master's Theorem for Sections (3) and (4)

The Master's Theorem states:

"Let $a \geq 1$ and $b > 1$ be constants, let f(n) be a function, and let T(n) be defined on the non-negative integers by the recurrence

$$T(n) = aT(n/b) + f(n)$$

where we interpret n/b to mean either $\lfloor n/b \rfloor or \lceil n/b \rceil$.

Then T(n) has to the following asymptotic bounds:

1. If $f(n) = O(n^{log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{log_b a})$.

2. If $f(n) = \Theta(n^{log_b a}), then T(n) = \Theta(n^{log_b a} lgn)$.

3. If $f(n) = \Omega(n^{log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \theta(f(n))$."

(Introduction to Algorithms by Cormen, Leiserson, and Rivest, 3rd ed.)

3 Problem 1

a)

$$f(n) = 32f(n/2) + n^3$$

In this function, let a = 32, b = 2, and $f(n) = n^3$. Subbing in,

$$n^{log_b a} = n^{log_2 32} = n^5$$

we know that $n^5 > f(n) => n^5 > n^3$. Therefore, this f(n) fits case 1 of the Master's Theorem and $\Theta(n^5)$.

b)

$$f(n) = 3f(n/2) + n^2$$

Let a = 3, b = 2, and $f(n) = n^2$. So,

$$n^{log_b a} = n^{log_2 3} \approx 1.59 < f(n) = n^2$$

Therefore, we know that this function belongs to case 3 with the time complexity of $\Theta(n^2)$

4 Problem 2

Using the Master's Theorem, as described in Section 2, and assuming T(1) = constant:

a)

$$T(n) = T(n/2) + n$$

First let's evaluate T(n) with the constant T(1),

$$T(1) = T(1/2) + 1 = 1/2T + 1 => -1 = 1/2T => -2 = T$$

Using the Master's Theorem, let a = 1, b = 2, and f(n) = n.

$$n^{log_b a} = n^{log_2 1} = n^0 = 1$$

Since,

$$\in = T(1) = -2 < 0,$$

$$af(n/b) = 1T(1/2) \geq c(f(n)) = -2(n), \text{ and}$$

$$c = T(1) = -2 < 1$$

we can determine that this function of T(n) fits case 3 of the Master's Theorem and the time complexity of this function is at $\Theta(n)$.

b)

$$T(n) = T(n/5) + n^2$$

$$T(1) = T(1/5) + 1^2 = 1/5T + 1 => -1 = 1/5T => -5 = T$$

Let a = 1, b = 5, and $f(n) = n^2$.

$$n^{log_b a} = n^{log_5 1} = n^0 = 1$$

Since,

$$\in = T(1) = -5 < 0,$$

$$af(n/b) = T(1/5) \geq c(f(n)) = -5(n), \text{ and}$$

$$c = T(1) = -5 < 1$$

we can determine that this function T(n) fits case 3 of the Master's Theorem and the time time complexity of this function is at $\Theta(n^2)$.

c)

$$T(n) = T(n/3) + const$$

Let a = , b = 3, f(n) = const = T(1) = O(1).

$$n^{log_3 1} = n^0 = 1$$

which is the same as $f(n) = const = T(1) = O(1)$. So, we can determine that this function fits into case 2 of the Master's Theorem with the time complexity of $\Theta(logn)$.

d)

$$T(n) = T(n-1) + n^3$$

Using the substitution method,

$$T(n) = T(n-1) + n^3$$

$$T(n-1) = T(n-2) + (n-1)^3$$

$$= T(n-2) + (n-1)^3 + n^3$$

$$= T(n-3) + (n-2)^3 + (n-1)^3 + n^3$$

$$= (n(n-1)/2) * n^3$$

we get the time complexity to be $\Theta(n^3)$.

5 Problem 3

Given two integers x and n, describe and prove the complexity of an algorithm that computes x^n. Is there any way to do multiplications faster than O(n)?

```
// java syntax
int x, n;
if (n == 0){
    return 1;
} else {
    int powerNum = Math.pow(x, n);
    return powerNum;
    }
```

This algorithm is in Java syntax using the power function from the math library. The algorithm takes in an integer as the base, x, and an integer as the exponent, n. If n, the exponent is 0, then return 1 because any number to the power of 0 is 1. However, if the n is not 0, then use Math.pow(x, n) function to compute x to the power of n. Then return the result. This algorithm has the time complexity of $O(1)$ which is better/faster than $O(n)$. So, yes, this is one way that you can perform multiplications faster than $O(n)$ by using $O(1)$ instead.

6 Problem 4

Algorithms we looked at:

1. Insertion Sort
2. Merge Sort
3. Quick Sort

a)

A[ ] is nearly sorted

If A[ ] is nearly sorted, I would choose insertion sort because it will perform the least swaps. Insertion Sort keeps the first i in order and iterates through each element in the array swapping elements as needed to put them in order. As stated in the lecture, Insertion Sort chooses the "i-th entry finds where it goes among first i-1 already in sorted order then inserts, pushing up all bigger ones". Looking at Insertion Sort's best $O(n)$, average $O(n^2)$, and worst $O(n^2)$ cases, an already partially sorted array would fit in the best case, as it would only need to iterate through the array at $O(n) = O(1)$.

For instance, let A[ ] = [1 3 5 0 2 6]

Insertion Sort starts with the first element. Being the first element, there is nothing for the algorithm to compare it to, so it assumes that the first element is in order.

[**1** 3 5 0 2 6]

Iterating to the second element, the algorithm also assumes the second element is in order as it is greater than 1.

[1 **3** 5 0 2 6]

Moving on to A[2], insertion sort can also assume that it is also in order as A[2] = 5 and $5 > 1$ and 3.

[1 3 **5** 0 2 6]

Now, we can see that 0 is not sorted. Insertion sort compares this to each previous sorted element and determines that 0 is at the beginning as it is less than 1 (the smallest previous already sorted element).

[1 3 5 **0** 2 6]

Iterating to the next element after 0 in the array, we see that 2 is also not sorted as $2 < 5$. Insertion sort performs the same comparisons as in the previous element, 0, and determines that $2 < 3$ but $2 > 1$, so it sorts 2 in between 1 and 3.

[0 1 3 5 **2** 6]

Lastly, we look at the last element, A[5], and notice that it is greater than 5 (our largest already sorted element).

[0 1 2 3 5 **6**]

Therefore, since no swaps were performed for the last element, we can say that this array is now sorted completely, where it only took $O(n)$ to completely sort an already partially sorted array.

[**0 1 2 3 5 6**]

Hence, why Insertion Sort would be the most efficient sorting algorithm to use in a partially sorted array as it uses the best case: $O(n)$.

b)

A[ ] consists of random numbers in random order

I think Merge Sort is the best option for an array that consists of completely random numbers in random order. I was debating between the possibilities of Merge Sort and Quick Sort but when comparing their best, average, and worst cases, Merge sort seems to be the better option, where its worst case is more efficient, $O(nlogn)$, in comparison to Quick Sort's worst case, $O(n^2)$.

Comparing Time Complexities of Merge and Quick Sort				
Sorting Algorithm	Best Case	Average Case	Worst Case	Space
Merge Sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$
Quick Sort	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	$O(logn)$

However, if you prioritize space complexity more-so than time complexity, then Quick Sort is more efficient as it uses less sub-arrays which requires less space needed for the algorithm. Since this question does not specifically specify whether time complexity or space complexity is more important, I believe Merge Sort would be the best option over all for random numbers in random order in terms of the best, average, and worst time complexity.

For instance,

$$\text{A[ ]} = [\ 3\ 6\ 2\ 7\ 9\ 1\ ]$$

First, we can split A [ ] in half and create two sub-arrays, left and right arrays.

$$[\ \mathbf{3}\ \mathbf{6}\ \mathbf{2}\ ] \text{ —— } [\ \mathbf{7}\ \mathbf{9}\ \mathbf{1}\ ]$$

Next, we can split the left and right sub-arrays again to create two more sub-arrays for each side.

$$[\ \mathbf{3}\ ]\ [\ \mathbf{6}\ \mathbf{2}\ ] \text{ —— } [\ \mathbf{7}\ ]\ [\ \mathbf{9}\ \mathbf{1}\ ]$$

Now with the arrays being broken down, we can start sorting the arrays containing two elements.

$$[\ 3\ ]\ [\ \mathbf{6}\ \mathbf{2}\ ] \text{ —— } [\ 7\ ]\ [\ \mathbf{9}\ \mathbf{1}\ ]$$

In the left array, since $6 > 2$, swap 6 and 2. In the right array since $9 > 1$, swap 9 and 1.

$$[\ 3\ ]\ [\ \mathbf{2}\ \mathbf{6}\ ] \text{ —— } [\ 7\ ]\ [\ \mathbf{1}\ \mathbf{9}\ ]$$

Now, you have the furthest right arrays on either side sorted. You will then need to sort both arrays on either side.

$$[\ \mathbf{3}\ ]\ [\ \mathbf{2}\ \mathbf{6}\ ] \text{ —— } [\ \mathbf{7}\ ]\ [\ \mathbf{1}\ \mathbf{9}\ ]$$

On the left side, $3 > 2$, but $3 < 6$, so we place 3 in between 2 and 6. On the right side, we do the same. We see that $7 > 1$, but $7 < 9$, so place 7 in between 1 and 9.

$$[\ \mathbf{2}\ \mathbf{3}\ \mathbf{6}\ ] \text{ —— } [\ \mathbf{1}\ \mathbf{7}\ \mathbf{9}\ ]$$

Now, sorting the left and right arrays together to get the completed sorted array of A[ ],

[**1 2 3 6 7 9**]

From this example, we can see that the time complexity of Merge Sort is $O(nlogn)$ regardless of best, average, and worst cases. However, space complexity is $O(n)$. Let's also look at Quick Sort.

For instance,

A[ ] = [3 6 2 7 9 1]

Let pivot = 2,

A[ ] = [3 6 **2** 7 9 1]

Move the pivot to the end,

A[ ] = [3 6 1 7 9 **2**]

Now, select your left and right elements (left = A[0] and right = A[4]).

A[ ] = [$\underline{3}$ 6 1 7 $\underline{9}$ **2**]

Since $3 < 9$, move the to the next i-1 on the right side until 3 is greater than an element on the right side.

A[ ] = [$\underline{3}$ 6 1 $\underline{7}$ 9 **2**]

A[ ] = [$\underline{3}$ 6 $\underline{1}$ 7 9 **2**]

Now, we can see that $3 > 1$, swap 3 and 1.

A[ ] = [$\underline{1}$ 6 $\underline{3}$ 7 9 **2**]

Now step right,

A[ ] = [1 $\underline{6}$ $\underline{3}$ 7 9 **2**]

Next step left,

A[ ] = [1 $\underline{6}$ 3 7 9 **2**]

Step left again,

A[ ] = [$\underline{1}$ $\underline{6}$ 3 7 9 **2**]

Swap the pivot and 6,

A[ ] = [$\underline{1}$ **2** 3 7 9 $\underline{6}$]

Now, the first two elements are sorted. Select a new pivot,

A[ ] = [1 2 3 **7** 9 6]

Move new pivot to the end,

A[ ] = [1 2 3 6 9 **7**]

Choose the left and right unsorted elements,

$$\text{A[ ]} = [\ 1\ 2\ \underline{3}\ 6\ \underline{9}\ \mathbf{7}\ ]$$

Since $3 < 7$ and $3 < 9$, step right.

$$\text{A[ ]} = [\ 1\ 2\ 3\ \underline{6}\ \underline{9}\ \mathbf{7}\ ]$$

Since $6 < 7$ and $6 < 9$, step right again.

$$\text{A[ ]} = [\ 1\ 2\ 3\ 6\ \underline{9}\ \mathbf{7}\ ]$$

Compare the pivot, 7, and the selected number, 9, since $9 > 7$, swap 7 and 9.

$$\text{A[ ]} = [\ 1\ 2\ 3\ 6\ 7\ 9\ ]$$

Now we have the sorted array.

Comparing the two, we can see that the space that was used is less than the space used in Merge Sort. My point for giving both Merge Sort and Quick Sort examples is to show that either could be best used to sort an array of random numbers in random order. It basically just depends on whether you are considering optimal space complexity. If space is taken largely into consideration, then Quick Sort would be better as it uses less space, however, when using Quick Sort, there is the risk of possibly going though the worst case time complexity if the numbers are entirely in a different order without the slightest possibility of two elements already being sorted at random. But if I had to choose, in a completely randomized order, Merge Sort would be the most optimal if you can afford to use the space required for it as the time complexity for the best, average, and worst cases remain the same $O(nlogn)$.