Shan Li (08428037) //help from Thomas

COSC 3020

Assignment 01

7 October 2019

For the purpose of making logs easier to type and so the base of the log doesn't appear tiny, the notation for logs in this document is logb(n), where b is the base and n is what you're taking the log of.

Question 1(help from Thomas):

O(log2(n)) and O(log10(n)) are the same if log2(n) ϵ O(log10(n)) and log10(n) ϵ O(log2(n)). This means that for some constant C, (C)log10(n) > log2(n) and for some constant K, (K)log2(n) > log10(n).

(C)log10(n) > log2(n)

(C)$\left(\frac{log2(n)}{log2(10)}\right)$ > log2(n)

(C)log2(n) > (log2(n))(log2(10))

C > log2(10)

(K)log2(n) > log10(n)

(K)$\left(\frac{log10(n)}{log10(2)}\right)$ > log10(n)

(K)log10(n) > (log10(n))(log10(2))

K > log10(2)

So log2(n) ϵ O(log10(n)) because (C)log10(n) > log2(n) if C > log2(10) and log10(n) ϵ O(log2(n)) because (K)log2(n) > log10(n) if K > log10(2).

Another way to think about it is if something is in O(log10(n)), then it must be in O(log2(n)) and vice versa because log10(n) = $\left(\frac{1}{log2(10)}\right)$log2(n) and log2(n) = $\left(\frac{1}{log10(2)}\right)$log10(n). Since $\left(\frac{1}{log2(10)}\right)$ and $\left(\frac{1}{log10(2)}\right)$ are both constants, any function that belongs to O(log10(n)) also belongs to O(log2(n)) because log10(n) can be represented in terms of log2(n) multiplied by a constant and vice versa. And asymptotically, constants don't matter.

Question 2(help from Thomas):

Since when n <= 1, it literally just returns (constant time), the base case would be T(1 or less) = 1. Else, it calls itself recursively with input (n/3), then it does n operations, then calls itself again with input (n/3).

So:

T(n) = $\left\{\begin{matrix} 1 & ,n = 1 \\ 2T\left(\frac{n}{3}\right) + n & ,else \end{matrix}\right\}$

T(n) (If n = 1, stop.)

= $2T\left(\frac{n}{3}\right) + \frac{1n}{1}$ (If n = 3, stop.)

= $3\left(2T\left(\frac{n}{9}\right) + \frac{n}{3}\right) + n$

= $6T\left(\frac{n}{27}\right) + 2n$ (If n = 27, stop.)

= $9\left(2T\left(\frac{n}{81}\right) + \frac{n}{27}\right) + 2n$

= $18T\left(\frac{n}{81}\right) + \frac{n}{3} + 2n$

= $18T\left(\frac{n}{81}\right) + \frac{7n}{3}$ (if n = 81, stop)

= $27\left(2T\left(\frac{n}{243}\right) + \frac{n}{81}\right) + \frac{7n}{3}$

= $54T\left(\frac{n}{243}\right) + \frac{n}{3} + \frac{7n}{3}$

= $54T\left(\frac{n}{243}\right) + \frac{8n}{3}$ (If n = 256, stop)

= $\sqrt{n}T\left(\frac{n}{n}\right) + \frac{\left(\sum_{i=0}^{log9(n)-1} 3^i\right)n}{\left(\frac{1}{3}\right)\sqrt{n}}$

= $\sqrt{n} + 3(\sqrt{n})(2^{log3(n)} - 1)$

= $\sqrt{n} + 3(\sqrt{n})(\sqrt{9^{log3(n)}} - 1)$

= $\sqrt{n} + 3(\sqrt{n})(\sqrt{n} - 1)$

= $\sqrt{n} + 3(n - \sqrt{n})$

= $3n - \sqrt{n}$

= n log3n

Question 2 Proof by Induction:

T(n) = $\begin{cases} 1 & , n = 1 \\ 2T\left(\frac{n}{3}\right) + n & , else \end{cases}$

Hypothesis:

T(n) = $3n - \sqrt{n}$ for all n where n is a power of 3 (3,9,27,81….) and n >= 3.

Base Case: n = 9

T(9)

= $2T\left(\frac{9}{3}\right) + 3$

= 23

= $3(9) - 3$

= $3(9) - \sqrt{9}$

= $3n - \sqrt{n}$

Induction Step: if T(n) = $3n - \sqrt{n}$, then T(9n) = $3(9n) - \sqrt{9n}$

T(9n)

= $2T(n) + 9n$

= $2\left(3n - \sqrt{n}\right) + 9n$ {Induction Hypothesis}

= $6n - \sqrt{n} + 9n$

= $27n - 3\sqrt{n}$

= $3(9n) - 3\sqrt{n}$

= $3(9n) - \sqrt{9n}$

Question 3(help from Thomas):

```
[ 1, 7, 1, 5, 3, -1, 9]
     ^
[ 1, 7, 1, 5, 3, -1, 9]
        ^
[ 1, 1, 7, 5, 3, -1, 9]
           ^
[ 1, 1, 5, 7, 3, -1, 9]
              ^
[ 1, 1, 3, 5, 7, -1, 9]
                 ^
[ -1, 1, 1, 3, 5, 7, 9]
                     ^
[-1, 1, 1, 3, 5, 7, 9]
```

^ points to the next element to be inserted into the sorted part of the array.

Example:

```
[ 1, 2, 3, 4, 5, 7, 6, 8, 10, 9 ]
                    ^
```

6 must be inserted into the subarray [1, 2, 3, 4, 5, 7]

Question 4(help from Thomas):
The code is in Question04.js if you want to run it, but here's the code:

```
// Thomas Wise(09086674) & Shan Li(08428037)
// COSC 3020
// Assignment 1, Question 4
// 25 Feb 2019


// Generates a random array 10 - 19 elements long with
// values from 0 - 99.
var a = Array(Math.floor(Math.random() * 10) + 10)
for (var i = 0; i < a.length; i++) {
    a[i] = Math.floor(Math.random() * 100);
}

// Displays the array, sorts it,
// then displays it again.
console.log("Unsorted List:");
console.log(a);
mergeSort(a);
console.log("Sorted List:");
console.log(a);

// Checks to see if the list is really sorted.
// Makes testing easier.
var sorted = true;
for (var i = 0; i < a.length - 2; i++) {
    if (a[i] > a[i + 1]) {
        sorted = false;
    }
}
console.log("Sorted Correctly: " + sorted);
```

```
function mergeSort(a) {
    var lo;
    var hi;

    // Outer loop determines size of the parts
    // being merged.
    for (var i = 2; i < 2 * a.length; i *= 2) {
        // Inner loop determines where the two parts
        // being merged begin in the array.
        for (lo = 0; lo < a.length; lo += i) {
            hi = lo + (i - 1);
            merge(a, lo, hi);
        }

    }
}
```

```
function merge(a, lo, hi) {
    // Splits the section given into two
    // smaller sections to be merged.
    var mid = Math.floor((lo + hi) / 2);
    // Merges the two sections in place.
    // Kind of similar to insertion sort.
    var iterator1 = lo;
    var iterator2 = mid + 1;

    while (iterator1 <= mid
        && iterator1 < a.length
        && iterator2 <= hi
        && iterator2 < a.length) {
        if (a[iterator2] < a[iterator1]) {
            var val = a[iterator2];
            for (var i = iterator2; i > iterator1; i--) {
                a[i] = a[i - 1];
            }
            a[iterator1] = val;
            iterator1++;
            iterator2++;
            mid++;
        }
        else {
            iterator1++;
        }
    }
}
```

```
Question 4 runtime analysis:
```

The outer loop will run log2(n) times. The inner loop will run n/2 times the first time through and the second time it will run n/4 times and then n/8 and so on and so forth until it's running n/n times. So it runs $\sum_{i=1}^{log2(n)} \frac{n}{2^i}$ times.

If the inner loop runs $\frac{n}{2^i}$ times in one iteration of the outer loop, then it is taking $\frac{n}{2^i}$ subarrays of size 2^i and merging the left and right half of the subarrays. So the final runtime of the program will be $\sum_{i=1}^{log2(n)} \frac{n}{2^i} T(2^i)$ where T(x) is the runtime of merging the left and right half of a subarray of size x.

The worst case for the actual merging function would be if you had to move everything from the right side in front of everything on the left side. The can only occur if every element in the initial array is less than every element to the left of it. This can only occur if the initial array is in reverse order. If this is true, then even though at the beginning of each call to the merger function the left and right sides are sorted, every element from the right side must be shifted in front of every element on the left side. This means that n/2 times, n/2 items must be shifted to the right. This gives the actual merging part a worst-case runtime of: T(n) = $\frac{n^2}{4}$

So with the runtime of the whole program being $\sum_{i=1}^{log2(n)} \frac{n}{2^i} T(2^i)$ and the fact that the worst-case runtime of T(x) is $\frac{x^2}{4}$, the worst case runtime of the whole program is

$$\sum_{i=1}^{log2(n)} \frac{n}{2^i} * \frac{(2^i)^2}{4}$$

$$= \sum_{i=1}^{log2(n)} \frac{n*2^i}{4}$$

$$= \frac{n}{4} \sum_{i=1}^{log2(n)} 2^i$$

$$= \left(\frac{n}{4}\right)\left(2^{log2(n)+1} - 1\right)$$

$$= \left(\frac{n}{4}\right)\left(2(2^{log2(n)}) - 1\right)$$

$$= \left(\frac{n}{4}\right)(2n - 1)$$

$$= \frac{1}{2}n^2 - \frac{1}{4}n$$

This gives the overall program a worst-case time complexity of Big-Theta(n^2).

Question 5(help from Thomas):

So the question is what is the probability of the pivot being in the middle half of the elements to be sorted. So one quarter will be to the left of this middle “good” half and one quarter will be to the right of this “good” half. These quarters on the side are the “bad sections.”

Choosing the first element gives you a 50% chance of picking a good pivot. The median of 3 will be in the middle “good section” if at least one selected is in the “good section” and the remaining two are not both in the same “bad section” (i.e. both too low or both too high).

The probability of at least one being in the middle is $1-(\frac{1}{2})^3$ and the chance of the other not two being in the same ‘bad section” is $1-(\frac{1}{2}*\frac{1}{4})$. So the odds of both of these things happening is

$$\left(1-\left(\frac{1}{2}\right)^3\right)\left(1-\left(\frac{1}{2}\right)\left(\frac{1}{4}\right)\right)$$

$$=\left(\frac{7}{8}\right)\left(\frac{7}{8}\right)$$

$$=\frac{49}{64}$$

This is approximately 76.65%. So you have a better probability of picking a good pivot with the median of 3 method.