COSC 3020
Algorithms

Name: Jacob Tuttle

Homework 1

October 11, 2019

1. Asymptotic Complexity: Let S be some arbitrary set such that $S \in O(log_2(n))$.

$S(n) \in O(log_2(n)) \implies S(n) \leq c \cdot log_2(n)$ for some arbitrary c.

Now, let c be some value such that both $c \cdot log_2(n)$ and $c \cdot log_{10}(n)$ are greater than or equal to $S(n)$.

$S(n) \leq c \cdot log_{10}(n)$ as a result of our definition of c, and $S(n) \leq c \cdot log_{10}(n) \implies S(n) \in O(log_{10}(n))$.

Now consider $T(n)$ such that $T(n) \in O(log_{10}(n))$.

$T(n) \in O(log_{10}(n)) \implies T(n) \leq c \cdot log10(n)$

Using the same arbitrarily large c as above, we can show that:

$$T(n) \leq c \cdot log_2(n)$$
$$\implies T(n) \in O(log_2(n))$$

Since $T(n) = c \cdot log_{10}(n) = S(n)$

$$T(n) = S(n) \implies O(log_{10}(n)) = O(log_2(n))$$

2. Runtime Analysis: Before being able to perform a runtime analysis evaluation, first the following code must be analyzed for complexity.

```
if (n <= 1) return;                                          (1)
```

```
mystery(n/3)                                                 (2)
```

```
for (var i = 0; i < n*n; i++) { count = count + 1; }         (3)
```

These equations have runtimes as presented in the table below.

Equation	Time Complexity
1	1
2	$T(\frac{n}{3})$
3	n^2

Using these analyzed times, we can then constuct a piecewise function to define the time complexity of the function `mystery` in terms of the input size n.

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 2T(\frac{n}{3}) + n^2 & n > 1 \end{cases}$$

From there, we are able to trace the recurrence relation step by step for some arbitrarily large n.

$$\begin{aligned} T(n) &= T(1) + 2T(\frac{n}{3} + n^2) \\ &= 1 + 2(T(1) + 2T(\frac{n}{9}) + (\frac{n}{3})^2) + n^2 \\ &= 3 + 4T(\frac{n}{9}) + \frac{2n^2}{9} + n^2 \\ &= 3 + 4T(\frac{n}{9}) + \frac{11n^2}{9} \\ &= 3 + 4(T(1) + 2T(\frac{n}{27}) + \frac{n^2}{9}) + \frac{11n^2}{9} \\ &= 7 + 8T(\frac{n}{27}) + \frac{103n^2}{81} \\ &= \vdots \\ &= (2^i - 1) + 2i \cdot T(\frac{n}{3^i}) + \lambda_{i-1} n^2 \end{aligned}$$

This approximation for the runtime of the ith recurence of the loop makes use of an equation[1] (4) for the fraction multiplied with n^2, indicated by λ_{i-1} above. The m present in the equation is the iteration number i minus 1 since the equation uses an input of 0 to produce its first value.

$$\lambda_m = \frac{\frac{-2^{m+1}}{7} + \frac{9^{m+1}}{7}}{9^m} = \frac{-2^{m+1} + 9^{m+1}}{7 \cdot 9^m} \tag{4}$$

Since this function, for some value of n will split it up into thirds, compounding the factor each time, (i.e. the first split is $\frac{1}{3}$, second $\frac{1}{9}$, etc.) the total number of executions will be $i = log_3(n)$. Substituting this value for i provides a recurence relation of:

[1] Part of this formula was retrieved, and then adapted to this context, from the Online Encyclopedia of Integer Sequences, http://oeis.org/A016133. Paolo P. Lava, June 16, 2008

$$
\begin{aligned}
T(n) &= (2^{log_3(n)} - 1) + 2 \cdot log_3(n) \cdot T(\frac{n}{3^{log_3(n)}}) + \lambda_{log_3(n)-1} n^2 \\
&= (2^{log_3(n)} - 1) + 2 \cdot log_3(n) \cdot T(\frac{n}{n}) + \lambda_{log_3(n)-1} n^2 \\
&= (2^{log_3(n)} - 1) + 2 \cdot log_3(n) \cdot 1 + \lambda_{log_3(n)-1} n^2 \\
&= (2^{log_3(n)} - 1) + 2 \cdot log_3(n) + \lambda_{log_3(n)-1} n^2
\end{aligned}
$$

Of the three families of functions represented ($2^{log_3(n)}$, $log_3(n)$, n^2), the function with the most rapid growth is n^2. Disregarding the constants represented by λ, we arrive at the conclusion that $T(n) \in O(n^2)$

3. Sorting — Insertion Sort:

Array at the beginning of Insertion Sort:

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

End of sorting

4. Sorting — Merge Sort:

Code solutions to the iterative, in-place merge sort problem provided in `mergeSort.js`

Test code provided in `mergeSortTest.js`

The first step to providing a Θ bound for this implementation of in-place, iterative merge sort is to analyze each part of the function. For simplicity sake, both for understanding the code as it was written, and to make analysis easier, segments were subdivided into subfunctions. We can analyze each of these individually and then combine the results to provide a total time complexity for the worst case runtime.

The lowest level function (i.e. the one that will only ever be at the top of the call stack) is `insert(arr, ins, tar)`[2].

```
function insert(arr, ins, tar) {
    arr.splice(ins, 1);
```

[2]Variable declarations, comments, and other code that contributes very little to our discussion on time complexity excluded in this document but present within the source code for the sake of brevity.

```
    arr.splice(tar, 0, tmp);
}
```

Within this function, the only lines doing work are those containing the list function `splice`. Each of these functions must traverse the array up to their goal index. This function takes a value from the second partition being merged and places it into the first partion. The worst case scenario is that the last element from partion two is being inserted as the last element in partition one. Since we are analyzing the worst case for this algorithm, the execution time for each of these traversals and operations would be n and $\frac{n}{2}$ respectively where n is the total size of the segments to be merged. Overall, this brings the worst case time complexity for this code segment to $\Theta(1.5n)$.

The next lowest level function, being the function where most of the work takes place, is `merge(array, a, b, c)`.

```
function merge(array, bottom, half, top) {
    while (bottom <= half && half <= top) {
        if (array[bottom] <= array[half]) {
            bottom++;
        } else {
            array = insert(array, half, bottom);
            half++;
        }
    }
}
```

Within this function, the while loop provides the majority of the time sink, as well as being the function where `insert` is called. This while loop, if it were to execute the maximum number of times, would execute n times where n is the size of the partition represented by the bounds `bottom` and `top`. The insert operation, one that takes elements from one partition and shifts them to another would happen $top - half$ times in the worst case scenario, or $\frac{n}{2}$ times. The other branch of the if statement simply increments a counter, an operation that takes constant time.

Therefore, we can conclude that the time complexity of the if-else block is 1 for the if and $1.5n$ respectively. When we place each of these within the branches and consider how often they will execute in the worst case, we can see that a worse case scenario is where every single element of the second partition must be inserted into the first partition (i.e. choosing the else branch every possible time). When this happens, the loop will have executed $\frac{n}{2}$ times giving a total complexity for the `merge` function of $0.75n^2$

Finally, we arrive at the analysis of the `mergeSort(arr)` function.

```
function mergeSort(arr) {
    for (var partSize = 2; partSize < 2 * arr.length;
        partSize *= 2) {

        for (var partBottom = 0; partBottom < arr.length;
            partBottom += partSize) {
```

```
                if (partHalf - 1 < partTop) {
                    arr = merge(arr, partBottom, partHalf, partTop);
                }
            }
        }
    }
```

First, while n was used to represent the partition size while examining the previous two functions, we will now use it to represent the total array size so that we may achieve an accurate Θ bound. Our previous analysis of `merge` is re-written as $\frac{3n^2}{4s}$ where s is the partition size (`partSize` within the code).

With that said, we shall continue the analysis by analyzing the outer for loop to determine how many times the inner loops will be run. Increasing by a factor of 2 each time, and by tracing out a few example inputs, the outer for loop will execute $log_2(n)$ times. The inner loop executes a total of $n-1$ times; however, this number isn't as useful as determining the number of times that `merge` will be called. On each iteration of the outer for loop, `merge` will be called. In other words, for each iteration of the outer loop, `merge` will be called s times. With this knowledge, it can be seen that the time complexity of the inner for loop will be $\frac{3n^2}{4}$. This loop, executing $log_2(n)$ times causes the inner loop to execute a total of $\frac{3n^2}{4} \cdot log_2(n)$ times giving a final worst case time complexity for inplace, iterative merge sort of $\Theta(n^2 \cdot log_2 n)$.

5. Sorting — Quicksort:

Consider an input array T and its sorted counterpart S. Choosing an element $t \in T$ at position i, all elements within T are equally likely to be at position i within S. Choosing any 3 elements from T extends this concept as all three have the same probability of being in their respecitve correct position. Now consider what makes a good pivot. A good pivot is one that partitions the input array into two approximately equally sized segments. Let us define a good pivot as one within the center third of S. Choosing three elements from T, they all have the same probability of being within this good region; furthermore, they have an equally likely probability to be within the lower third or upper third of S.

Let us represent the position of these pivots within S using a three digit, base three number. The digit represents the number of pivots in that position and the position represents whether those pivots are in the left, center, or right partition. For instance, the reprsentation 012 indicates that no pivots selected were in the first third of S, one was in the center third, and two were in the final third. All valid selections are then represented by the list {012, 021, 102, 111, 120, 201, 210}.

The mechanism used to pick the final pivot is to take the median of the three sampled pivots. Using this method, we can separate the representations into good, $G = \{021, 111, 120\}$, and bad, $B = \{012, 102, 201, 210\}$, selections by determining whether the median selected pivot would be within the center partition of S. Therefore, the chance of selecting a good final pivot would be represented by $\frac{|G|}{|G \cup B|} = \frac{3}{7} \approx 0.429$.

Similarly we can encode choosing a single pivot in a similar way. All possible pivots being {001, 010, 100}. Of these, only 1 is a good pivot, making the probability of picking a good

pivot with only 1 sample $33.\overline{3}$. Therefore, we can conclude that selecting a pivot by using the median-of-three method is certainly better than using only a single pivot as the probability of selecting a good pivot is nearly 30% greater.