Henry Daniels-Koch
September 16 2016

Algorithms: Lab 1

Homework Problems

1). This algorithm recursively divides the list into two lists until the list contains fewer than 3 numbers. The algorithm then checks to see which number in the 1-2 size list is the higher number, and saves the resulting minimum and maximum. After the algorithm has recursively determined the minimum and maximum of the next 1-2 numbers, it merges the lists. In the merge process, the algorithm both compares the two maximums and the two minimums. This recursion process ends when the end of the list is reached and fully merged.

```
(max,min)=findMaxMin(array[n])
min,max = array[0]
initialize arrayMax and arrayMin
while i < (n - 2):      if (array[i] < array[i + 1]):
        add A[i+1] to arrayMax
        add A[i] to arrayMin
    else:           add A[i] to arrayMax
        add A[i+1] to arrayMin
    i+=2
for i=0 to n/2:     if arrayMax[i] > max:
        max = arrayMax[i]
    if arrayMin[i] < min:
        min = arrayMin[i]
return (min,max)
```

Runtime: This algorithm runs with fewer than $3n/2$ comparisons. The "while" loop makes $\frac{n}{2}$ comparison while iterating through each array of maxes or mins makes $\frac{n}{2}$ comparisons. Thus, in total the algorithm has $3n/2$ comparisons and is $O(n)$.

2). The following function is an example of a function that is neither $O(n)$ nor $\Omega(n)$:

$$f(n) = n\cos^2(n) + \frac{n}{2} \tag{1}$$

As n$\to \infty$, $f(n)$ oscillates about the function $g(n) = n$. We know this to be true because the average value of $\cos^2(x) = \frac{1}{2}$, which implies

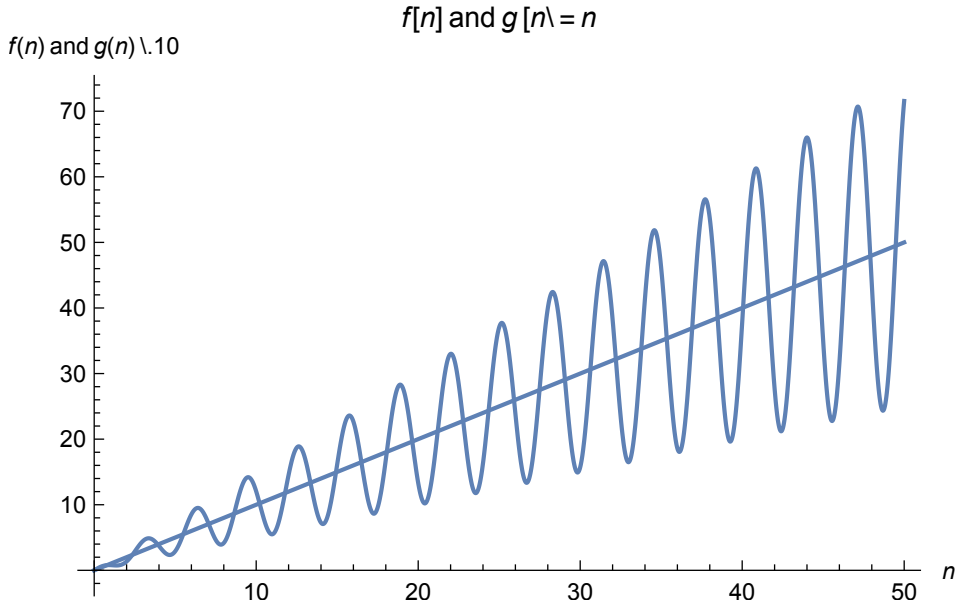
$$\implies \langle f(n)\rangle = n\left\langle\cos^2(n)\right\rangle \frac{n}{2} \tag{2}$$

$$\implies \langle f(n)\rangle = \frac{n}{2} + \frac{n}{2} \tag{3}$$

$$\implies \langle f(n)\rangle = n \tag{4}$$

$$\tag{5}$$

However, due the oscillation from the $\cos^2(x)$ term, $f(n)$ oscillates about $g(n) = n$. Since $f(n)$ oscillates about $g(n) = n$, it is neither bounded above or below by $g(n) = n$ and thus is not $O(n)$ nor $\Omega(n)$.



3). See attached

4). The algorithm below traverses across columns in an array A, until it arrives at an index with a 0 in the array. Prior to hitting a 0, if the column contains a 1, the row is marked as the row with the most 1's (maxRow). If a 0 is reached, the algorithm moves down a row to check for a 1 by increasing the row number. NOTE: This algorithm does not include a way to manage edge cases. One could manage this by either checking if the algorithm has reached the last row and encountered a 0 or using a for loop over j and maintaining the same j when a 0 is reached. The pseudocode is as follows:

```
maxRow = 0
i,j=0      //(i is row number, j is column number)
while(j < (n − 1))
    if A[i,j] == 1:
        maxRow = i
        j++
    else:
        i++
return maxRow
```

The runtime of the function $f(n)$ is $O(n)$. We can show this by thinking of a pointer moving across the array and calculate the runtime based on the number of traversed places in the array. We know the pointer must move to the right, down, or stop in each "move". When the pointer stops, it has made $n-1$ comparisons that resulted in a move to the right. To maximize the number of moves in the worst case scenario, the pointer will move all the way down to the last row before reaching the final column. In such a case, it will have made $n-1$ comparisons. Thus the total number of comparisons is:

$$\begin{aligned} N &= (n-1) + (n-1+) \\ &= 2n-2 \end{aligned}$$

Thus, the algorithm is $O(n)$.

5). When you double the input size, the runtime becomes:

$$a).(2n)^2 = 4n^2$$
$$b).100(2n)^2 = 400n^2$$
$$c).(2n)^3 = 8n^3$$
$$d).(2n)lg(2n)$$
$$e).2^{2n} = 4^n$$

I plugged in $2n$ where the initial algorithm had size n.
i.e. $n^2 \rightarrow (2n^2) = 4n^2$