Michael Nutt
CS 325.400
HW 2

1.

a. $T(n) = b * T(n - 1) + 1$
Using substitution:
$T(n - 1) = b * T(n - 2) + 1$
$T(n - 2) = b * T(n - 3) + 1$
…
$T(2) = b * T(1) + 1$
$T(1) = b * T(0) + 1$
Substituting T(1) in T(2):
$T(2) = b * (b * T(0) + 1) + 1$
$= b^2 * T(0) + b + 1$
Substituting T(2) in T(3):
$T(3) = b * (b^2 * T(0) + b + 1) + 1$
$= b^3 * T(0) + b^2 + b + 1$
We've found a recognizable pattern:
$T(n) = b^n * T(0) + b^{n-1} + b^{n-2} \ldots b^1 + b^0$
$= b^n * T(0) + (b^n - 1)/(b - 1)$
$= b^n * T(0) + (b^n - 1) * (1/(b - 1)$
Let $1/(b - 1) = c$
**$= b^n * T(0) * (b^n - 1) * c$**

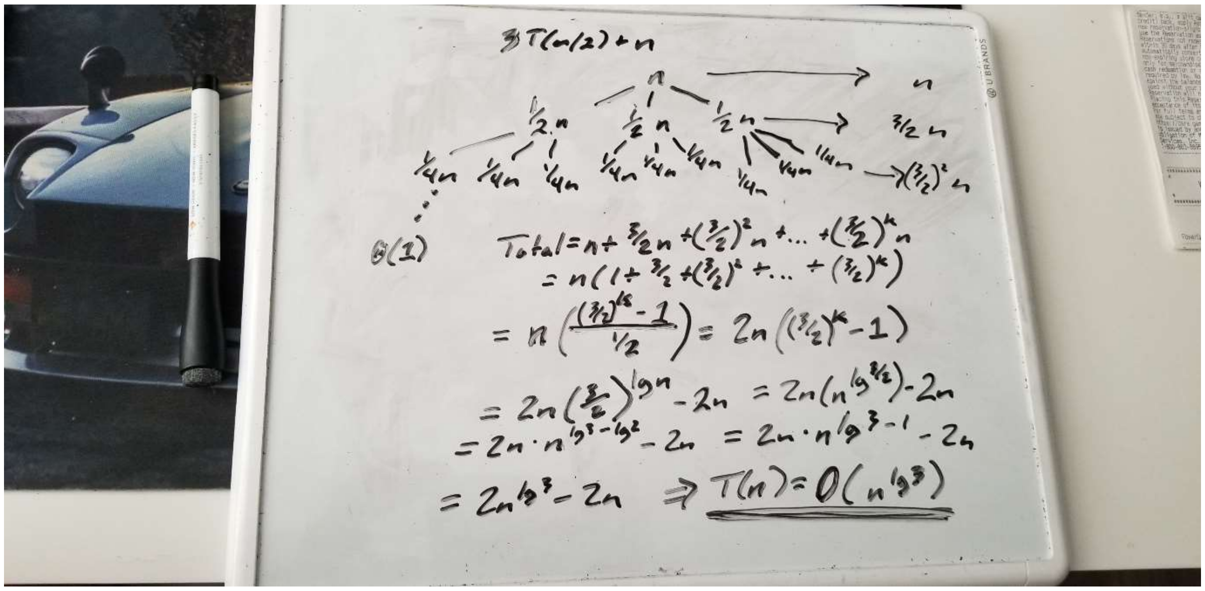
b. $T(n) = 3 * T(n/9) + n\log n$
This recurrence is of the form $T(n) = a * T(n / b) + f(n)$, where $a >= 1$, $b > 1$, and f is positive. Thus, we can use the Master Method; $a = 3$, $b = 9 => n^{\log_3 9} = n^2$, and $f(n) = n\log n$.
$f(n) = O(n^{2 - \varepsilon})$
Thus, $T(n) = \Theta(n\log n)$

Michael Nutt
CS 325.400
HW 2

2.
```
length = len(array)
max_sum = 0
current_sum = 0
for i = 1 to length:
    current_highest = i
    if (current_sum > 0):
        current_sum = sum + array[i]
    else:
        current_lowest = i
        current_sum = array[i]
    if (current_sum > max_sum):
        max_sum = current_sum
        lowest = current_lowest
        highest = current_highest
return (lowest, highest, max_sum)
```

3.

a.


$3T(n/2)+n$
$n$
$\frac{1}{2}n$ $\frac{1}{2}n$ $\frac{1}{2}n$
$\frac{1}{4}n$ $\frac{1}{4}n$ $\frac{1}{4}n$ $\frac{1}{4}n$ $\frac{1}{4}n$ $\frac{1}{4}n$ $\frac{1}{4}n$ $\frac{1}{4}n$ $\frac{1}{4}n$
$\Theta(1)$
$n$
$\frac{3}{2}n$
$(\frac{3}{2})^2 n$

$$\text{Total} = n + \tfrac{3}{2}n + (\tfrac{3}{2})^2 n + \dots + (\tfrac{3}{2})^k n$$
$$= n(1 + \tfrac{3}{2} + (\tfrac{3}{2})^2 + \dots + (\tfrac{3}{2})^k)$$
$$= n\left(\frac{(\frac{3}{2})^k - 1}{\frac{1}{2}}\right) = 2n((\tfrac{3}{2})^k - 1)$$
$$= 2n(\tfrac{3}{2})^{\lg n} - 2n = 2n(n^{\lg \frac{3}{2}}) - 2n$$
$$= 2n \cdot n^{\lg 3 - \lg 2} - 2n = 2n \cdot n^{\lg 3 - 1} - 2n$$
$$= 2n^{\lg 3} - 2n \Rightarrow \underline{T(n) = O(n^{\lg 3})}$$

Michael Nutt
CS 325.400
HW 2

b. $T(n) <= c * n^{lg3}$ for some constant c
   Base case: Assume T(1) = 1; $T(1) <= c * 1^{lg3}$; 1 <= c * 1 where c = 1 Thus, our basis holds.
   Induction: T(n) = 3 * T(n/2) + n
   Suppose $T(n/2) <= c * (n/2)^{lg3}$
   $T(n) <= 3 * (c * (n/2)^{lg3}) + n$
   $<= 3 * (c * (n^{lg3}/2^{lg3})) + n \Rightarrow 2^{lg3} = 3$
   $<= 3 * ((c * n^{lg3})/3) + n$
   $<= c * n^{lg3} + n \Rightarrow c * n^{lg3}$
   Thus, $T(n) <= c*n^{lg3} \Rightarrow T(n) = O(n^{lg3})$

4.

a. Assume $\alpha$ = ½ and n is a power of 2. When n = 2, badSort compares and sorts A[0] and A[1]. While n > 2, badSort sorts the first half of the array (A[0 … n/2-1]) first, then the second half of the array (A[n/2 … n-1]), then the already-sorted first half of the array again (A[0 .. n/2-1]). Now, suppose A is an unsorted array and in the first half of A there exists an element larger than all other values in A, A[i]. badSort will sort the first half of the array, and A[i] will be moved to A[n/2-1]. The second half of the array will then be sorted. Then, the first half of A will be sorted again, leaving A[i] at A[n/2-1]. As A[i] is the largest element in A, it should be located at A[n-1] for A to be sorted. However, since it is located at A[n/2-1], badSort fails to sort A when $\alpha$= ½. For any $\alpha$ <= ½, badSort will sort the two halves individually, and as long as there exists an element in the first sub array that is larger than any element in the second sub array, badSort will not properly sort A.
b. Assume $\alpha$ = ¾. For each recursive call of badSort, a sub array of size m is passed in. For the base case to be satisfied and to end the recursion, the array's size must be reduced to 2. However, if an array of size 3 is passed to badSort, then m = 3, and the size will never be reduced to 2. Thus, an infinite loop is created. This can be corrected by adding an if statement to check if m = n. If so, reduce m by 1, thus triggering the base case.
c. After performing constant time operations, badSort makes three recursive calls. Each recursive call sorts an array of size m, which equates to $\alpha$ * n. Thus we have:
   $T(n) = 3*T(\alpha * n) + O(1)$
d. Assume $\alpha$ = 2/3. We then have the recurrence T(n) = 3 * T(2/3n) + O(1). We can apply the master theorem as follows:
   a = 3, b = 3/2 => $n^{\log_{3/2}3}$, $f(n) = n^0$
   Since $0 < \log_{3/2}3$, $T(n) = \Theta(n^{\log_{3/2}3})$

Michael Nutt
CS 325.400
HW 2

5.

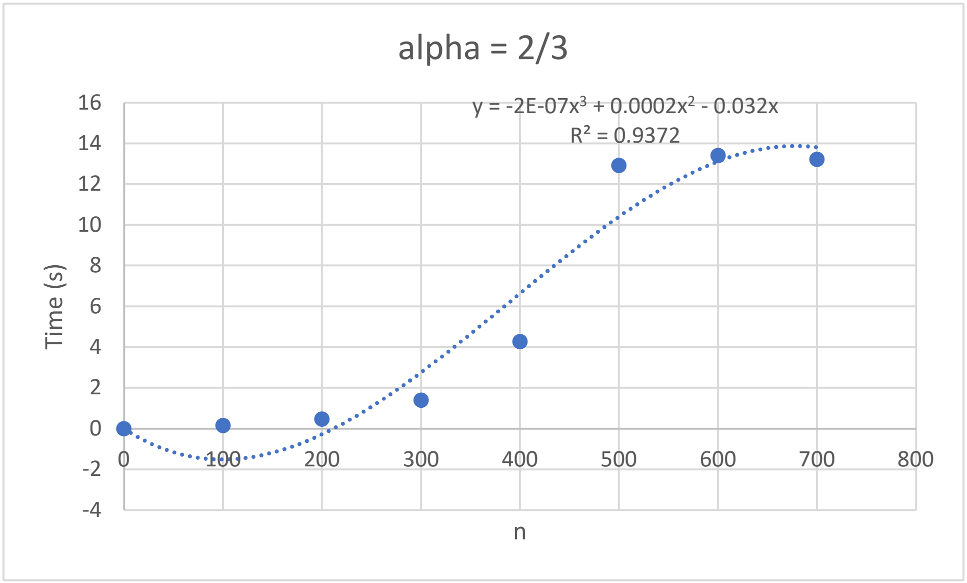
b.

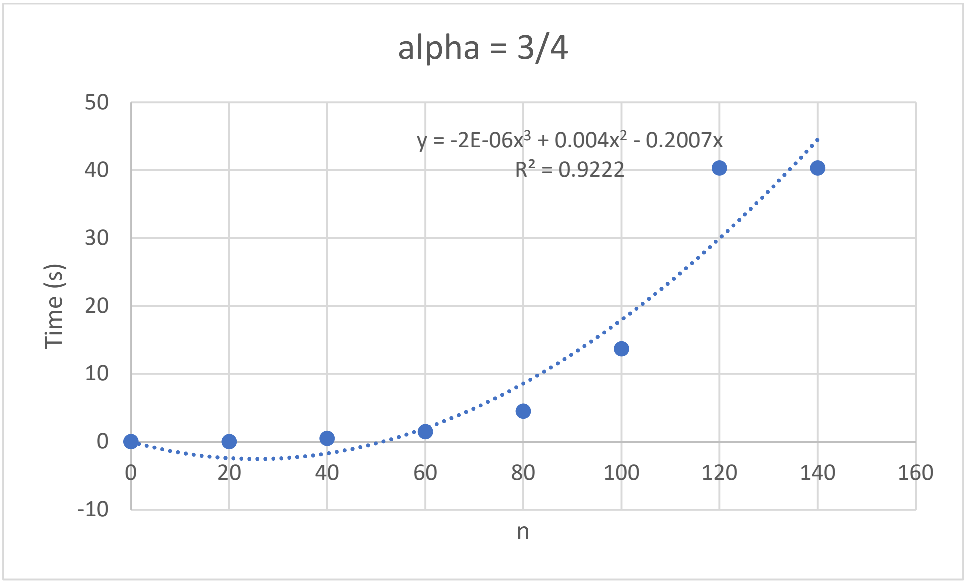
| $\alpha$ = 3/4 | |
|---|---|
| n | Time (s) |
| 0 | 0 |
| 20 | 0.01394 |
| 40 | 0.49564 |
| 60 | 1.47398 |
| 80 | 4.47301 |
| 100 | 13.7126 |
| 120 | 40.31391 |
| 140 | 40.34117 |

| $\alpha$ = 2/3 | |
|---|---|
| n | Time (s) |
| 0 | 0 |
| 100 | 0.15597 |
| 200 | 0.48289 |
| 300 | 1.41165 |
| 400 | 4.28028 |
| 500 | 12.93136 |
| 600 | 13.40873 |
| 700 | 13.22656 |

Michael Nutt
CS 325.400
HW 2

c.





The curve for the experimental running times are somewhat off from the theoretical running time, but I think that is more due to limitations in the functionality of excel. For example, when alpha is ¾, the theoretical runtime is $T(n) = \Theta(n^{\log_{3/2} 3})$, with $n^{\log_{3/2} 3} \approx n^{2.71}$. Thus, I picked the polynomial curve with a factor of 3 in order to attempt to get a closer fit.

d. An alpha of 2/3 provided a much better performance than and alpha of ¾.