Richard Moot
CS325
June 26, 2016

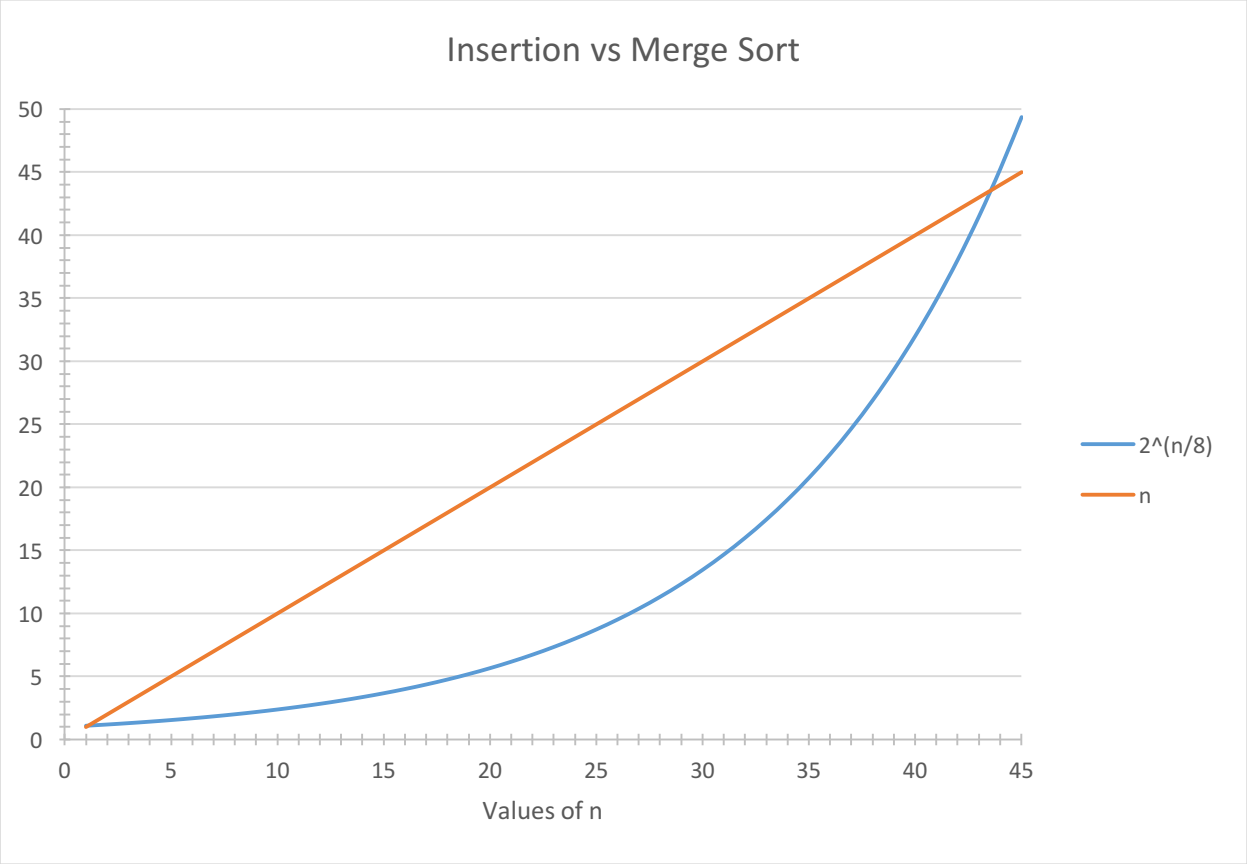
Homework 1

1. (CLRS) 1.2-2. Suppose we are comparing implementations of insertion sort and merge sort on the same machine. For inputs of size n, insertion sort runs in 8n2 steps, while merge sort runs in 64nlgn steps. For which values of n does insertion sort beat merge sort?

When comparing insertion sort and merge sort for find values which insertion sort beats merge sort, we can represent them as an inequality. This can be shown as

$$8n2 < 64n\ lg\ n == \frac{n}{8} < \lg n == 2^{\frac{n}{8}} < n$$

Once we have this, we can then plot values to identify where merge sort crosses over to beat insertion sort.



Here we can see that at 43 that there is a crossover and merge sort starts to beat out insertion sort.

2. 2) (CLRS) Problem 1-1 on pages 14-15.

From the very beginning here, its clear that that numbers generated are too large to write, since they are 300k+ digits in length.

	1 Second	1 Minute	1 hour	1 day	1 month	1 year	1 century
lg n	$2^{10\text{^}6}$	$2^{10\text{^}6}$*60	$2^{10\text{^}6}$*60^2	$2^{10\text{^}6}$*86,400	$2^{10\text{^}6}$*2.592e6	$2^{10\text{^}6}$*(60^2)*(24)*365	$2^{10\text{^}6}$*3.1536e9
$\sqrt{n}$	10^{12}	10^{12}*60	10^{12}*60^2	10^{12}*86,400	10^{12}*2.592e6	10^{12}*(60^2)*(24)*365	10^{12}*3.1536e9
n	10^6	10^6*60	10^6*60^2	10^6*86,400	10^6*2.592e6	10^6*(60^2)*(24)*365	10^6*3.1536e9
n lg n	62746	3764760	2.26E+08	5.42E+09	1.62638E+11	1.97876E+12	1.97876E+14
n^2	1000	60000	3600000	86400000	2.59E+09	3.15E+10	3.15E+12
n^3	100	6000	360000	8640000	2.59E+08	3.15E+09	3.1536E+11
2^n	19	1140	68400	1641600	49248000	5.99E+08	5.99E+10
n!	9	540	32400	777600	23328000	2.84E+08	2.84E+10

3. (CLRS) 2.3-3 on page 39. Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2, \\ 2T\left(\frac{n}{2}\right) + n & \text{if } n = 2^k, \text{for } k < 1 \end{cases}$$

is $T(n) = n \lg(n)$

Base Case: When $n = 2, T(2) = 2 * \log(2) = 2$, so we can see that the base case holds for our initial step.

Inductive Step: Let us assume there exists a value k > 1, such that $T(2^k) = 2^k \log 2^k$. It is necessary that we prove that the case holds for k + 1 as well.

$$\begin{aligned} T(2^{k+1} &= 2T\left(\frac{2^{k+1}}{2}\right) + 2^{k+1} \\ &= 2T(2^k) + 2 * 2^k \\ &= 2 * 2^k \lg 2^k + 2 * 2^k \\ &= 2 * 2^k (\lg 2^k + 1) \\ &= 2^{k+1}(\lg 2^k + \lg 2) \\ &= 2^{k+1} \lg 2^{k+1} \end{aligned}$$

We have successfully proven k+1, so the inductive step is complete. Since both the base step and the inductive step have been completed by mathematical induction, the statement holds that "$T(2^k) = 2^k \log 2^k$ holds for all n that are exact power of 2".

4. For each of the following pairs of functions, either f(n) is O(g(n)), f(n) is Ω(g(n)), or f(n) = Θ(g(n)). Determine which relationship is correct and explain.

a. $f(n) = n^{0.75}$; $g(n) = n^{0.5}$

   f(n) is O(g(n)) since the exponent digresses by a constant factor

b. $f(n) = n$; $g(n) = \log^2 n$

   f(n) is Ω(g(n)) since f(n) grows significantly faster than g(n)

c. $f(n) = \log n$; $g(n) = \lg n$

   f(n) is Θ(g(n)) since they differ on a constant factor of $\log_{10}2$ and n doesn't influence the change in growth between these

d. $f(n) = e^n$; $g(n) = 2^n$

   f(n) is Ω(g(n)) because the difference between their values grows wider as *n* increases

e. $f(n) = 2^n$; $g(n) = 2^{n-1}$

   f(n) is Θ(g(n)) since we can ignore the constant -1 on g(n) since it would influence very little as n grows.

f. $f(n) = 2^n$; $g(n) = 2^{2\text{^}n}$

   f(n) is O(g(n)) since g(n) grows significantly faster than f(n)

g. $f(n) = 2^n$; $g(n) = n!$

   f(n) is O(g(n)) for the same reason as the previous problem. As *n* grows larger, n! will grow faster than 2^n

h. $f(n) = n \lg n$; $g(n) = n\sqrt{n}$

   f(n) is O(g(n)) since we can disregard the additional n's factors as they grow at the same time, but lg(n) grows more slowly than n squared.

5) Describe in words and give pseudocode for a $\Theta(n \lg n)$ time algorithm that, given a set S of n integers and another integer x, determines whether or not there exist two elements in S whose sum is exactly x. Demonstrate your algorithm on the set S = { 12, 3, 4, 15, 11, 7 } and x = 20.

First, you'll want to run merge sort on S to get a sorted array and then scan through that array to compare each element the one next to it to determine if there is a match to x.

```
    SUM(S,x)
1.  MERGE – SORT(S)
2.  i ⇐ 1
3.  j ⇐ n
4.  while I < j
5.      do if S[i] + S[j] < x
6.          then i ⇐ I + 1
7.          else if S[i] + S[j] > x
8.              then j ⇐ j – 1
9.              else return S[i], S[j]
10. if I = j
11.     then return  NIL
```

Example:

1. S = {12, 3, 4, 15, 11, 7}; x = 20
2. S = {3, 4, 7, 11, 12, 15}; x = 20; i = 1; j = 6;
3. S[i] + S[j] = 18; x = 20 i = 1; j = 6;
4. S[i] + S[j] = 19; x = 20 i = 2; j = 6;
5. S[i] + S[j] = 22; x = 20 i = 3; j = 6;
6. S[i] + S[j] = 19; x = 20 i = 3; j = 5;
7. S[i] + S[j] = 23; x = 20 j = 4; j = 5;
8. Return NIL

6) Let f_1 and f_2 be asymptotically positive functions. Prove or disprove each of the following conjectures. To disprove give a counter example.

a. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ then $f_1(n)+f_2(n) = O(g_1(n)+g_2(n))$

$$\begin{aligned} f_1(n)+f_2(n) &\leq c_1 g_1(n) + c_2 g_2(n) \\ &\leq c_1 \max(g_1(n),g_2(n)) + c_2 \max(g_1(n),g_2(n)) \\ &\leq (c_1+c_2)\ \max(g_1(n), g_2(n)) \end{aligned}$$

We can see that since O() is only concerned with the maximal value existing inside, it would have to return a boundary that would ultimately bound f_1 and f_2

b. If $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$, then $\frac{f_1(n)}{f_2(n)} = O(\frac{g_1(n)}{g_2(n)})$

$$\begin{aligned} \frac{f_1(n)}{f_2(n)} &\leq (\frac{c_1 g_1(n)}{c_2 g_2(n)}) \\ &\leq \frac{c_1}{c_2} * (\frac{g_1(n)}{g_2(n)}) \\ &\leq \frac{c_1}{c_2} * (\frac{g_1(n)}{g_2(n)}) \end{aligned}$$

We can see here that it holds true since it reduces out to a constant factor that is just a ration of the constant for f_1 over f_2.

c. $\max(f_1(n), f_2(n)) = \Theta(f_1(n) + f_2(n))$

$$\max(f_1(n), f_2(n)) = c_1 f_1(n) + c_2 f_2(n)$$

We can already start to see that it is not possible for these two to be equivalent since on the LHS we are only taking the maximum, while on the RHS, we are looking for a tight boundary on both functions.

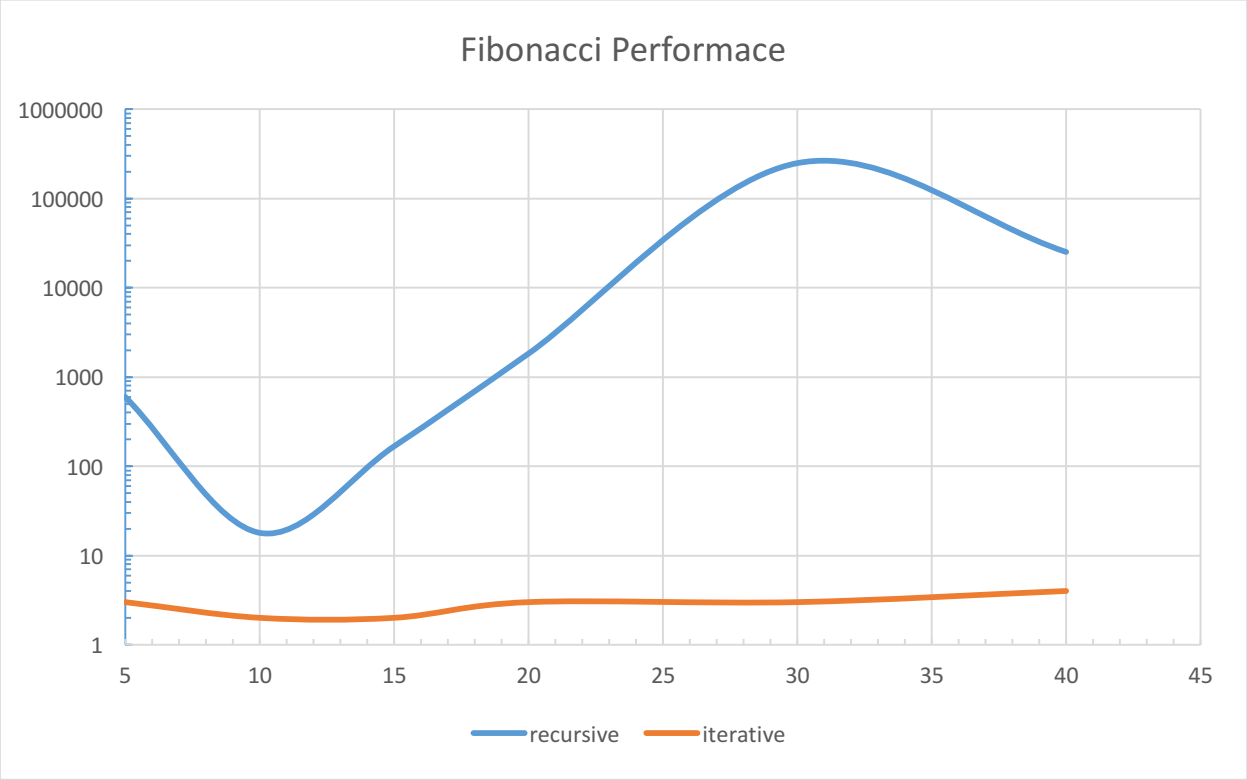
7a.

```
from datetime import datetime

def recursiveFib(n):
    if n == 1 or n == 2:
        return 1
    else:
        return recursiveFib(n - 1) + recursiveFib(n - 2)

def iterativeFib(n):
    a,b = 1,1
    for i in range(n-1):
        a,b = b,a+b
    return a

testdata = [5, 10, 15, 20, 30, 40]
recursiveResults = []
iterativeResults = []

recursiveFib(5)
for i in testdata:
    start = datetime.now()
    recursiveFib(i)
    end = datetime.now()
    recursiveResults.append((end - start).microseconds)

iterativeFib(5)
for i in testdata:
    start = datetime.now()
    iterativeFib(i)
    end = datetime.now()
    iterativeResults.append((end - start).microseconds)

print recursiveResults
print iterativeResults
```

7b.

n	5	10	15	20	30	40
recursive	600ms	18ms	168ms	1838ms	250808ms	25344ms
iterative	3ms	2ms	2ms	3ms	3ms	4ms

7c.



7d.

The iterative Fibonacci method has a linear curve to it and grows at a constant rate (would be more apparent at higher values). The recursive implementation has a polynomial curve, increasing rapidly for even very small values of n. In testing, it was clear going over 40 took several minutes and it was of concern it would not finish computing in a reasonable amount of time.

The iterative approach is much faster since it can simply go through the calculations, while the recursive approach doesn't have its final result until it finishes computing each function call. This creates a new stack frame for calculating the next iteration, but it won't complete until it reaches the last number and starts to return back out of all of the created stack frames.