Vlad Predovic

CS325

Due 4/03/2016

Homework 1

1)

Insertion sort: ~~8n *~~ n steps
Merge sort: ~~64n~~ 8* $\log_2(n)$ steps
Python Solution:

```
>>> num1=2
>>> num2=3
>>> while num2 > num1:
...     num1 += .1
...     num2 = math.log(num1, 2)*8
>>> num2
43.57004983911661
>>> num1
43.60000000000035
```

Therefore n can be a maximum of 43(if Integer). Merge sort becomes more efficient at higher numbers due to having a logarithmic complexity.

2)

Column1	1 sec	1 min	1 hr	1 day	1 mo	1 yr	1 century
lg n	2^1E6	2^60E6	2^3600E6	2^86400E6	2^2592000E6	2^31104000E6	2^3110400000E6
sqrt(n)	1E+12	3.6E+15	1.296E+19	7.46496E+21	6.71846E+24	9.67459E+26	9.67459E+30
n	1000000	60000000	3600000000	86400000000	2.592E+12	3.1104E+13	3.1104E+15
n lg(n)	62746	2801418	133378058.9	2755147513	71870856404	7.8709E+11	6.76995E+13
n^2	1000	7745.966692	60000	293938.7691	1609968.944	5577096.019	55770960.19
n^3	100	391.4867641	1532.618865	4420.837798	13736.57091	31448.89673	145972.8479
2^n	19.93156857	25.83845916	31.74534976	36.33031226	41.23720286	44.82216536	51.46602155
n!	9	11	12	13	15	16	17

3) Use mathematical induction to show that when n is an exact power of 2, the solution of the recurrence

$$T(n) = \begin{cases} 2 & \text{if } n = 2, \\ 2T(n/2) + n & \text{if } n = 2^k, \text{for } k > 1 \end{cases}$$

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

For all n,k in $\mathbb{N}$ that are an exact power of 2, T(n) = nlgn.
Base Case: When n = 2 → 2lg(2) = 2*1 = 2

Assume T(n)= nlogn when $n=2^k$ then for a given number k:
If $n = 2^{k+1}$ then
$T(2^{k+1}) = 2T(2^{k+1}/2)+2^{k+1} = 2T(2^k*2\ /2) + 2^{k+1} = 2T(2^k) + 2^{k+1}$

$T(2^{k+1}) = 2*2^k(\log(2^k)) + 2^{k+1}$
$= 2^{k+1}(\log 2^k + 1)$
$= 2^{k+1}(\log 2^{k+1})$

4)

a. $f(n) = n^{0.75}$; $g(n) = n^{0.5}$ → $f(n) = \Omega(g(n))$ since .75 > .5 and it is known that for example n^3 will always we greater than n^{3-x} for all positive numbers n and x, we can assume that f(n) will at the very least grow as fast as g(n) when n is greater than 1.
b. $f(n) = n$; $g(n) = \log^2 n$ → $f(n) = \Omega(g(n))$ Linear will always be equal to or slower than logarithmic after a set constant c=1
c. $f(n) = \log(n)$; $g(n) = \lg(n)$ → $f(n) = \Theta(g(n))$ When speaking asymptotically, the base of logarithms is irrelevant. This is because of the identity $\log_a b \log_b n = \log_a n$
d. $f(n) = e^n$; $g(n) = 2^n$ → $f(n) = \Omega(g(n))$ The value of e is approximately 2.7. This will always be greater than 2 therefore at best f(n) will increment at a speed equal to g(n)
e. $f(n) = 2^n$; $g(n) = 2^{n-1}$ → $f(n) = \Theta g(n)$ $2^{n-1} = 2^n/2$ $O(2^n/2) = 2^n$ as it only differs by a constant/

f. $f(n) = 2^n$; $g(n) = 2^{2^n}$ → $f(n) = O(g(n))$

For all n greater than 1 $2^n > n$ since $2^n > 2*n > n$. Therefore f(n) at worst will equal g(n).

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

$\lim_{n\to\infty} \frac{2^n}{n!} = 0$, therefore 2^n is bounded above by n!

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

```
nlgn > n√n . If we take an arbitrary number n=100000
lg(100000) > √100000
lgn' = 1/n while √n'  =  1/(2√n )
1/(2√n ) > 1/n
2√n  < n
n(n - 4) > 0  proving that a change in n causes a bigger change in √n  than in lgn
```

5)

This algorithm needs to determine if there are two numbers in a set S that equal another number X with a time complexity of O(nlogn).

I first use a Mergesort which has a worst case time complexity of O(nlogn) satisfying the requirements.

In order to create this algorithm I nested a search algorithm within an outer loop that would iterate through the set with a complexity of O(n). The algorithm inside this loop would then search for the corresponding number that when summed with the first number. For this I used the binary search.

My codeish pseudocode can be found below..

```
def twoMatch(x, array):
    Sortmerge(set)
    x = 20
    For num in array:              O(n)
        num_srch = x - num         O(1)
        if binSearch(num_srch)          O(logn)
            return true

    return false
```

S = {3, 4, 7, 11, 12, 15} after Mergesort.

Step through each number in the array.

For 3 → binary search for 20-3 (17)

Not found, proceed to next number at next index in array.

For 4 → search for 20-4 (16)

Not found

For 7 → search for 20-7 (13)

Not found

Repeat…….

Loop ends without a match.

Function returns false as no two numbers equaling 20 were found.

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 $f_1(n)+f_2(n) = O(g_1(n)+g_2(n))$.

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

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

a. Applying the rule of sums results in:

Let $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$
There exist constants c1,c2>0 where $0 \leq f_1(n) \leq c1\ g_1(n)$ and $0 \leq f_2(n) \leq c2\ g_2(n)$
There must exist a constant c3 where $(f_1(n) + f_2(n)) \leq c3(g_1(n) + g_2(n))$ for all positive integers greater than a given n_0.

$f_1(n) + f_2(n)) \leq c1g_1(n) + c2g_2(n))$
$f_1(n) + f_2(n)) \leq c3(g_1(n) + g_2(n))$ satisfies the formal definition below

$f(n) = O(g(n))$	$\exists c > 0,\ \exists n_0,\ \forall n > n_0, f(n) \leq c \cdot g(n)$

b. $f_1(n) = n^2$ and $f_2(n) = n$. $g_1(n)=n^3$ and $g_2(n) = n!$

$\lim_{n\to\infty} \frac{n\text{^}2}{n} = \infty$ while $\lim_{n\to\infty} \frac{n\text{^}3}{n!} = 0$

Therefore $\frac{f1(n)}{f2(n)} \neq O(\frac{g1(n)}{g2(n)})$ so this is false

c. To prove this conjecture we first assert the following statements

$f(n) \leq \max(f(n),g(n))$ **$g(n) \leq \max(f(n),g(n))$**

Therefore it can be said that: **$(f(n)+g(n))/2 \leq \max(f(n),g(n))$**

Due to the properties of max it can also be stated that since both f(n) and g(n) are positive functions **$\max(f(n),g(n)) \leq 1(f(n) + g(n))$**

If substituted **$h(n) = f(n) + g(n)$**

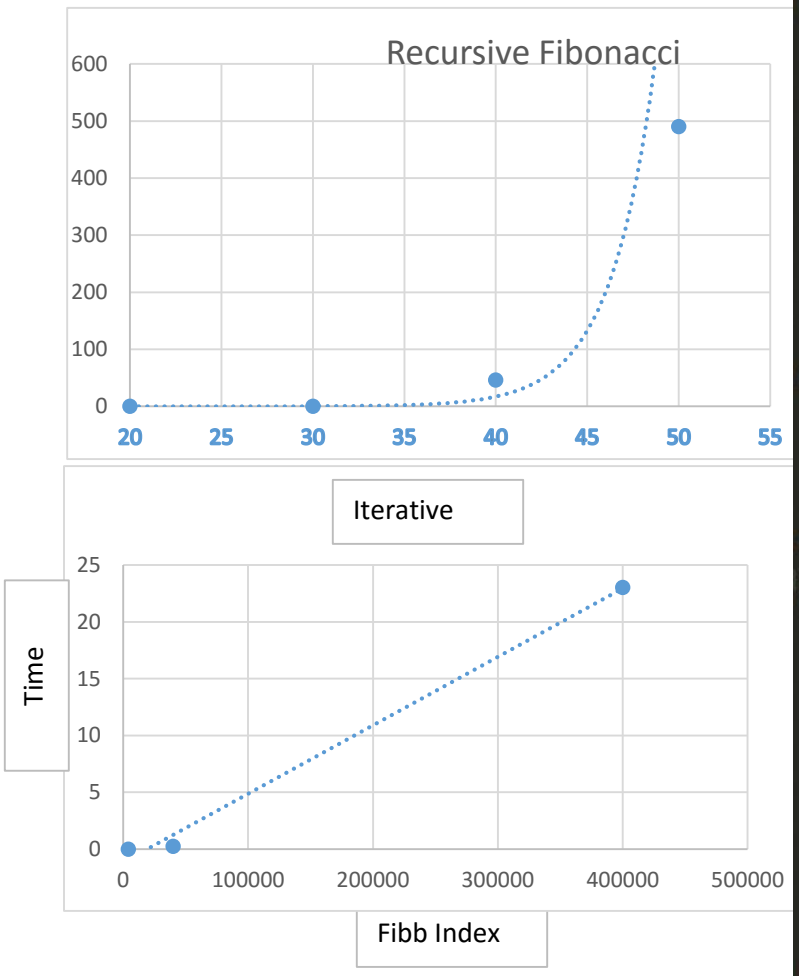
Then it can be concluded that **$c1(h(n)) \leq \max(f(n),g(n)) \leq c2(h(n))$**

Where **$c1 = ½$ and $c2 = 1$**

This satisfies the formal definition given below for the class $f(n) = \Theta(g(n))$

$$\exists c_1, c_2 > 0,\ \exists n_0,\ \forall n > n_0, c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

7)




```
import time

def fiboIterative(x):
    num1 = 0
    num2 = 1
    temp = 0

    if x == 0:
        return 0
    elif x == 1:
        return 1
    else:
        for i in xrange(2, x):
            temp = num2
            num2 += num1
            num1 = temp

    return num2


def fiboRecursive(x):
    if x == 0:
        return 0
    elif x == 1:
        return 1
    else:
        return fiboRecursive(x-1) + fiboRecursive(x-2)

def in_the_forest(duck):
    print(duck.quack())

def main():
    print "insert a number representing the index of the fibb
    num = int(raw_input())

    print "insert i for iterative or r for recursive"
    choice = raw_input()

    if choice == 'i':
        start_time = time.time()
        ans = fiboIterative(num)
    if choice == 'r':
        start_time = time.time()
        ans = fiboRecursive(num)

    print("--- %s seconds ---" % (time.time() - start_time))


if __name__ == "__main__": main()
```

For my data I found very aggressive differences in running times. The recursive function fit best with an exponential graph while the iterative Fibonacci function fit best with a linear graph. The running times are so different due to the algorithmic complexity of each version.

Data Points	Iterative	Recursive
20	0	0.003000021
30	0	0.368999958
40	0	46.28600001
50	0	490.2334
400	0	
4000	0.001999855	
40000	0.242000103	
400000	23.03499985	

In the iterative version there is one loop in which three constant time actions are performed. This ultimately results in a linear time complexity as shown by the data.

In the recursive version, every time a call to the function is made it spawns two additional functions until an end case is met. Since in each step we call the function twice the result is a complexity of $O(t) = 2^n$ which is incredibly slower than the linear counterpart.

```
import time

def fiboIterative(x):
	num1 = 0
	num2 = 1
	temp = 0

	if x == 0:
		return 0
	elif x == 1:
		return 1
	else:
		for i in xrange(2, x):
			temp = num2
			num2 += num1
			num1 = temp

	return num2


def fiboRecursive(x):
	if x == 0:
		return 0
	elif x == 1:
		return 1
	else:
		return fiboRecursive(x-1) + fiboRecursive(x-2)

def in_the_forest(duck):
```