Quiz #7: Inheritance, Analysis of Algorithms/Complexity Classes ICS-33 Spring 2014

Name _____

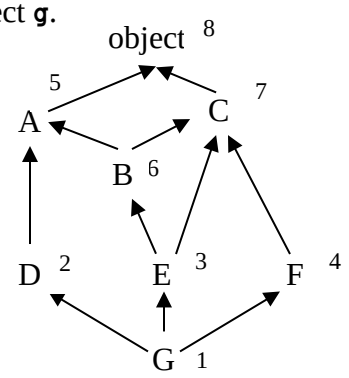
Lab # (1-8) _____

Recall the Academic Integrity statement that you signed. Write all answers clearly on these pages, ensuring your final answers are easily recognizable. The number of points for each problem is clearly marked, for a total of 25 points. I will post my solutions on the web on Monday, off the **Solutions** link, after class.

There is no helper project file for this assignment. Hand in this at the start of class on Wednesday.

1. (4 pts) Given the following class definitions, draw the inheritance network and number each **class** showing in what order the class objects are searched for attributes (1 = first, 2 = second, etc.) in object **g** where **g = G().** Ignore that it first looks in object **g**.

```
class A       : pass
class C       : pass
class B(A,C)  : pass
class D(A)    : pass
class E(B,C)  : pass
class F(C)    : pass
class G(D,E,F): pass
```



2. (2 pts) Assume that a sorting function **s** in in the complexity class O($N\sqrt{N}$). (a) What is its doubling-signature: how much more time (by what factor) does it take to solve a problem twice as large? Show your calculation. (b) This signature is between the signature of the linear and quadratic complexity class. Explain which is it closer to.

(a) Assume **Ts(N) ~ c $N\sqrt{N}$**. Then, T(2N)/T(N) ~ c $2N\sqrt{2N}$ / c $N\sqrt{N}$ ~ $2\sqrt{2}$ ~ 2.83

(b) The linear signature is **2**, the quadratic signature is **4; s**'s signature is **2.83**, a bit closer to linear.

3. (6 pts) Assume that functions **f1** and **f2** compute the same result by processing the same argument. Empirically we find that $\mathbf{T_{f1}(N) = 120\ N}$ and $\mathbf{T_{f2}(N) = 10\ N \log_2 N}$ where the times are in seconds. (a) For what size **N** would these two functions take the same amount of time? Show how you **calculated** your answer (b) for what size arguments is it better to use `f1`? `f2`? (c) Briefly describe how we can write a function **f** that runs as fast as the fastest of **f1** and **f2** for all size inputs.

(a) We solve for when these two functions are equal: i.e., for what **N** does $\mathbf{120\ N = 10\ N \log_2 N}$; dividing each side by **10N** we have $\mathbf{12 = \log_2 N}$ or $\mathbf{2^{12} = N}$ (or **4,096** ~ **N**). If the complexity classes were more complicated, we could have plotted these equations, or plugged-in various values of **N** to find when they cross.

(b) `f2` is faster for all **N <4,096** and `f1` is faster for all **N >4,096**. The lower complexity class is better for all sizes beyond where the curves cross, because it is growing more slowly.

(c) Write a function **f** that tests its argument size; if it is **<= 4096** it calls **f2** to solve the problem; if it is **> 4096 it** call **f1** to solve the problem.

4a. (6 pts) The following functions each determine if any two values in **alist** sum to **asum**. As is shown in the notes, (a) write the complexity class of each statement on its right, where **N** is **len(alist)**, and then (b) **write and simplify** the calculation that shows the complexity class for the entire function. Finally, (c) specify which function you would choose when **alist** was large **and explain why**.

```
def sum_to_a (alist,asum):
    for f in alist:                 O(N)
        for s in alist:             O(N)
            if f+s == asum:         O(1)
                return (f,s)        O(1)
    return None                     O(1)
```

(b) `O(N)*O(N)*O(1) + O(1) is O(N**2)`

```
def sum_to_b (alist,asum):
    aset = set(alist)               O(N)
    for v in alist:                 O(N)
        if asum-v in aset           O(1)
            return(v,asum-v)        O(1)
    return None                     O(1)
```

(b) `O(N) + O(N)*O(1) + O(1) =O(2N) =O(N)`

(c) **sum_to_b**: as N gets bigger this lower-complexity class function is guaranteed to eventually run faster than **sum_to_a** which has a higher complexity class.

4b. (4 pts) Assume that method **m** is in the complexity class **O(N $\log_2$ N)**, and that for **N = 1,000** the program runs in **3 seconds**.

(1) Write a formula, **T(N)** that computes the approximate time that it takes to run **m** for any input of size **N**. Show your work/calculations by hand, approximating logarithms, finish/simplify all the arithmetic.

From the formula, **$T(N) = c\ (N \log_2 N)$** we have **$3 = c\ (1{,}000 \times 10)$**; therefore, **$c = 3\text{x}10^{-4}$**
So, **$T(N) = 3\text{x}10^{-4} * (N \log_2 N)$**

(2) Compute how long it will take to run when **N = 1,000,000 (**which is also written **10^6**). Show your work/calculations by hand, approximating logarithms, finish/simplify all the arithmetic.

$T(10^6) = 3\text{x}10^{-4} * (10^6 \log_2 10^6) = 3\text{x}10^{-4} * (10^6 \times 20) = 3\text{x}10^2 \times 20 = 6000$ seconds (~ >1.5 hour)

Note a quick way to compute (b): N is 1,000 times as large, the $\log_2$ N is twice as large, the result is 2,000 times as large.

4c. (3 pts) Assume that we have recorded the following data when timing three methods (measured in milliseconds). Based on these times (which are approximate, so don't expect perfect results), **fill in an estimate** for the complexity class for each method and **fill in an estimate** for the running time when N = 1,600.

N	Time: Method 1	Time: Method 2	Time: Method 3
100	**300**	**20**	**20**
200	**604**	**76**	**22**
400	**1,196**	**325**	**20**
800	**2,395**	**1,178**	**19**
1,600	**4,800**	**4,800**	**20**
Complexity Class Estimate	**~doubles: O(N)**	**~ quadruples: $O(N^2)$**	**~ no real increase: O(1)**