Sam Hopkins
Hw 1
CS 312

0.1 a) $f = \Theta(g)$

b) $f = O(g)$

c) $f = \Theta(g)$

d) $f = \Theta(g)$

e) $f = \Theta(g)$

f) $f = \Theta(g)$

m) $f = O(g)$

0.2 a) if $c < 1$

as the exponent of c increases to n, the value of a number c^i approaches 0, so the limit of $g(n)$ only approaches $\frac{1}{1-c}$, which is constant, and so $g(n) = \Theta(1)$ if $c < 1$

b) if $c = 1$, then each c^i will be 1, so the complexity is constant, and depends on n. If $n = 6$ then we get 7, which is $n+1$, so $g(n) = \Theta(n)$ if $c = 1$

c) if $c > 1$

Then each c^i is going to get exponentially bigger, but the highest complexity would be c^n, which is significantly bigger than the rest of the c^i variables, so $g(n) = \Theta(c^n)$ if $c > 1$

```
def fab(n):
   if n <= 2:
      return 1
   else:
      return fab(n-1) + fab(n-2) * fab(n-3)
```

For this exponential function, we see that for each n in fab(n), we get a total big O complexity of around 3^n. Each call to fab() increases the total calls by 3, thus tripling the work done by each function call at each level, making the big O 3^n. We can compare the fab() function to the Fibonacci function analysis on page 14 to show this concept as well.

```
def linFab(n):
  fn = f1 = f2 = f3 = 1
  if n <= 2:
    return 1
  else:
    for x in range(2, n):
      fn = f1 + f2 * f3
      f3 = f2
      f2 = f1
      f1 = fn
  return fn
```

For this linear function of the fabonacci function, we see that for each n > 2, we get a single addition and a single multiplication for n - 2 iterations. For example, if n = 3, we will only ever get n - 2 addition and multiplications, so we only will ever add and multiply once. If n = 10, the same analysis holds because we will have 8 additions and 8 multiplications. If n <= 2, then not a single multiplication or addition will be used.