Homework 1

Kevin Martin
CIS675 - Syracuse University

January 23, 2020

1. Question 1 - Show the asymptotic relation between each function pair.
   (a) $f(x) = 7x^4 + 5x^3 - 2x^2 + 100000$ and $g(x) = 2x^5 + x - 2$

   By using the Limit Theoroem and applying L'Hopital's rule, we can show the following:
   $\lim_{x\to\infty} \frac{f(x)}{g(x)} = \frac{7x^4+5x^3-2x^2+10000}{2x^5+x-2}$
   $\prime = \frac{28x^3+15x^2-4x}{10x^4+1}$
   $\prime = \frac{84x^2+30x-4}{40x^3}$
   $\prime = \frac{168x+30}{120x^2}$
   $\prime = \frac{168}{240x}$
   $\prime = 0$

   Because L=0, f(x)=O(g(x)).

   (b) $f(x) = x^{10}2^{2x}$ and $g(x) = 2048 * 5^x$

   Once again, we can use the Limit Theorem and L'Hopital's rule to prove the asymptotic relation. However this time, we will also use the Product Rule in finding the derivatives for each function:
   $\lim_{x\to\infty} \frac{f(x)}{g(x)} = \frac{x^{10}2^{2x}}{2048*5^x}$
   $\prime = \frac{10x^9 2^{2x}+x^1 0x}{2048x+5}$
   At this point, we can already see that f(x) grows substantially faster than g(x). Thus L=∞, and g(x)=o(f(x)) and f(x)=Ω(g(x)), which means g(x)=O(f(x)).

2. Question 2 - Find the running time of the following loops in terms of Big-O of N (size of input). Demonstrate how you determined the running time of the given code.

   (a) def test1 (input): (constant time)
   I = len(input) (constant time)

while I>1: (first loop)
K=I (constant time)
while K>1: (second loop)
K = K/4 (constant time)
I = I/2 (constant time)

For this function, there are two nested while loops, each running O(n) times (less the constants of 2 and 4, respectively). The work done defining the variables I and K is constant time, and the work dividing each is also constant time. Because the second while loop is dependendt on the first, we must multiply the two together, which gives us the following:
$c_1 + c_2 + c_3 + c_4 + (\frac{n}{2} * \frac{n}{4})$
This simplifies to O(n^2) running time.

(b) def test2(input):
I = len(input)
while I > 1:
I = I - 3

This function simply iterates through a list in linear time O(n). As long as $n > 1$, then the only work being done is constant time to declare the variable and subtract 3 from I.

(c) def powerOf3(n):
if n = 0:
return 1
elif n mod 2 = 1: (i.e. n is odd)
x = powerOf3((n - 1) /2)
return 3 * x * x
else: (if n is even)
x = powerOf3(n / 2)
return x * x

This function is recursive, and it also divides the input n in half each time it is called. Thus, it is a good candidate for the Master Theorem. Even though there are two recursive calls, they are separated by an if/else statement. Thus they are not dependent on each other and their running times do not need to be multiplied together. To calculate the running time, we must determine the inputs to the Master Theorem:
The work done in setting up the variables, checking if n is odd or even, and even the multiplication needed for the return statement are all done in O(1) time. This means n raised to the power of 0, so d = 0. There is only one recursive call per loop, again because of the

if/else statement, so a = 1. Finally, each recursive call divides n by 2, so b = 2. Putting this together, we get Case 2, where d = $log_b a$. Thus the run time is O(log n).

(d) def test4(input):
for i in range(len(input)):
mdx = i
for j in range (i+1, len(input)+1):
If input[j] < input[mdx]:
mdx = j
if mdx != i:
tmp = input[i]
input[i] = input[mdx]
input[mdx] = tmp

The final function here does not have a recursive call, thus we will not use the Master Theorem. Instead, we can look once again at the individual lines to determine the running time. Again, we see a nested loop (this time a for loop), where the inner loop is dependent on the outer one. The rest of the steps for the function are constant time (setting variables, comparing two numbers, and reassigning various index positions new values): O(1). The two for loops iterate through each item of the input, thus the running times of each are O(n), and must be multiplied together. The total running time for this algorithm is O(n^2).

3. Question 3 - Solve the following recurrences with substitution method (include proof by induction):

(a) T(1)=1, T(n)=T(n/3)+T(n/9)+T(n/27)+n (Assume n is a power of 3)
Claim: The recurrence relation is given by:
$T(1) = 1$
$T(n) = T(n/3) + T(n/9) + T(n/27) + n$
has the solution of $3^{log_3 n}$

Proof. Proof by induction
Base case: let n=1, $T(1) = (0/3) + (0/9) + (0/27) + 1 = 1$
IH: Assume that the claim holds for some $3^u, T(3^u) = 3^u$.
We will show that the claim holds for 3^{u+1}:
$T(3^{u+1}) = T(3^{u+1}/3) + T(3^{u+1}/9) + T(3^{u+1}/27) + 3^{u+1}$
$= (3^u) + (3^u/3) + (3^u/9) + 3^{u+1}$
$T(3^{u+1}) = 3^u + 1$

■

(b) T(1)=1, T(n)=4T(n/2)+n^2 (I struggled with this one as well)
Claim: The recurrence relation given by:
$T(1) = 1$
$T(n) = 4T(n/2) + n^2$
has the solution T(n) = $T(n) = 2^{logn}$

Proof. Proof by induction
Base case: let $n = 1, T(1) = 2^{log1} = 2^0 = 1$
IH: assume claim holds true for some 2^u, where $T(2^u) = 2^u$
Now show that claim holds true for 2^{u+1}:
$T(2^{u+1} = 4T(n/2) + n^2$
$= 4T(2^{u+1}/2 + 2^{u+1^2} = 4T(2^u) + 2^{2u+2} = 4 * 2^u + 2^u * 4 * 4\ T(2^{u+1} = 132 * 2^{u+1}$

■

4. Question 4 - For Each of the following recurrences, give asymptotic bound by using master method (if applicable). Demonstrate your conclusion.

(a) T(n)=7T(n/2)+$\sqrt{n}$
a = 7, b = 2, d = $\frac{1}{2}$. This gives us Case 3, as $\frac{1}{2} < log_2 7$
The asymptotic bound is $O(n^{log_2 7}) \approx O(n^{2.81})$

(b) T(n)=4T(n/2)+3log n
a = 4, b = 2, d = 1. This gives us Case 3, as $1 < log_2 4$
The asymptotic bound is $O(n^{log_2 4}) = O(n^2)$

(c) T(n)=15T(n/4)+$3n^2$+10000000
a = 15, b = 4, d = 2. This gives us Case 1, as $2 > log_4 15$
The asymptotic bound is $O(n^2)$

(d) T(n)=8T(n-2)+$n^3 - 2n^2$
The Master Theorem is not applicable in this case sa the recurrence does not take the proper form. The function must reduce n by some factor b, not simply subtract a constant from it.

5. Question 5a - Write a recurrence for the running time T(n) of f(), and solve that recurrence.

For the given function, there will be at most two recurrence calls. In the else block of the function, both the variable lft_found and rht_found require a call back to f. Thus the constant in front of T will be 2 (and a = 2). Next, the function will, at best, reduce the input in half. Thus n will be divided by 2 (and b=2). Finally, there are multiple operations that

need to be performed. The assignment of variables (other than those that require a recursive call) is done in constant (O(1)) time. The function scan() has been stated to run in O(n) time. So the extra "work" that the function must do is simply the O(n) runtime (and d=1), since we can ignore the constant time operations.

Putting it all together, we can come up with the following function: $T(n) = 2T(\frac{n}{2}) + O(n)$.
This gives us Case 2, as $1 = log_2 2$, for a running time of $O(nlogn)$.

6. Question 5b - U.S. Mint is collecting ideas to improve their quality check program. One of proposals suggested fast solution to detect any out of specification coin (either heavier or lighter than a genuine coin). This proposal also gave an example to demonstrate how to find such a coin out of 26 coins by using three rounds of weighings. Assume that we represent n coins in an array c[].

   To solve this problem, cosider the following algorithm:

```
def coinSort(c [])
n = lenth(c)
if n = 1 then
  return 1 // this indicates the coin is fake
else
  set1 = roundup(n/3) // equal set 1
  set2 = roundup(n/3) // equal set 2
  set3 = n - (2 * roundup(n/3)) // remainder, equal if n mod 3 = 0
  if (weight(set1) == weight(set2)) then // no fake in either set1 or set 2
    return coinSort(set3) //possible fake coin in this set
  else
    return coinSort min(weight(set1), weight(set2)) // check lighter set
```

   Proof. To prove, we can assume the Master Theorem because the algorithm divides n by some factor.
   In this case, it is dvided by 2, so b = 2. The algorithm requires two recursive calls, however they are separated by an if/else statement. As such, each pass is only one recursive call, so a = 1. Finally, the extra work done here is not dependent on the length of n. As such, that work can be done in constant time, O(1), where d = 0. The algorithm can be summarized by

   $T(n) = T(\frac{n}{2}) + O(1)$

This puts us in Case 2, where $0 = log_2 1$. The final run time is $O(n^0 logn) = O(logn)$, which is better than $O(n)$, our original goal.

■