CS170–Fall 2015 — Homework 1 Solutions

Nguyet Minh Duong, SID 24444722

Collaborators: Henry Kwan & James Carlson

1. Getting started

(a) No, it is never appropriate to look at somebody else's solutions, as well as looking at past semester's solutions from other students.

(b) FALSE. It is not appropriate to look up a solution to a problem online even if we cite the source and rewrite it.

2. Compare Growth Rates

(a) $f(n) = log_3(n)$ and $g(n) = log_4(n)$ then $f = \Theta(g)$. This is the answer because if we use logarithmic definition, we can make $f(n) = \frac{log_4(n)}{log_4(3)}$. Now we can remove $log_4(3)$ because it is a constant in the equation, as a result, now $f(n) = log_4(n)$ which is equivalent to $g(n) = log_4(n)$.

(b) $f(n) = nlogn^4$ and $g(n) = n^2logn^3$ then $f = O(g)$. We can see that this is the answer because we can divide the factors in both functions. Once we do that, we get $f(n) = 4$ and $g(n) = 3n$. We can see that $f(n)$ is smaller than $g(n)$ as it is only a constant.

(c) $f(n) = n^{1.04}logn$ and $g(n) = n(logn)^3$ then $f = \Omega(g)$. Function f grows at a slower faster because once we divide out all the common factors, we get: $f(n) = n^{0.04}$ and $g(n) = (logn)^2$. Now by our knowledge, we know that all polynomials grow fasters than any logarithmic – as a result, function f grows faster, hence it takes longer as n gets bigger.

(d) $f(n) = nlogn$ and $g(n) = (log(n))^{logn}$ then $f = \Omega(g)$. Similar to part c, if we simplify the equations, we get: $f(n) = n$ and $g(n) = (logn)^{logn-1}$. Similarly, as stated before, all polynomials grow faster than any logarithmic. Therefore function f grows faster so it takes longer as n gets bigger.

(e) $f(n) = 4^n$ and $g(n) = n!$ then $f = O(g)$. This is by reason: as n gets extremely big, function g multiplies it by that big number, whereas f will multiply 4 by n-times. That big number should already dominate the multiplication of 4 by n-times. Therefore function f is faster.

3. Bit Counter

Answer: If n is the number of bits, then $\Theta(2^n)$ is the growth for how many bit flips is required to go from the value 0 to $2^n - 1$.

After doing some of the lower numbered bits, we found out that there was a pattern to the flips. It is always the number of bits, plus twice the flip of the previous number of bits minus one. So in formula form: $NumFlipBits(n) = n + 2(NumFlipBits(n-1))$ with the base case $n = 1, NumFlipBits(1) = 1$.

We then notice for bigger numbers, 2 multiplies itself n-times. This gives us the exponential value 2^n. In this case, we can ignore the values of when n is being added because 2^n with a large n grows to the point where adding the n's will not affect our asymptotic analysis.

4. Geometric Growth

For the first condition, $\Theta(c^k)$ when $c > 1$ is because for asymptotic analysis, extreme precision is not necessary if one of the values ends up dominating other values because that one specific value is what affects the computational timing the most. It is $\Theta(c^k)$ because it is a summation c raised to the i power, where i starts as 0 and goes up to k. If we make k a very large number, we see that the difference between c^{k-1} and c^k is so large that the value c^k is what is affecting the computation value the most. Hence why we only look at that specific value in our asymptotic analysis.

Similarly, $\Theta(k)$ when $c = 1$ because of the exponential rule – one to any power is always one. Hence the summation ends up adding the value one k-times. This results in the the same value added together – in this case it adds up to k.

Lastly, $\Theta(1)$ when $c < 1$ because of another exponential rule. c can be any number (except 0) but as long as it is raised to the power of 0, the answer will be 1. Again, in asymptotic analysis, we only care of about the dominating value – which is 1 in this case. The reason for this is because any $c < 1$ when raised to a high power, it gets smaller and smaller where $c^0 = 1$ will be its highest value. Hence why it is the dominating value, and therefore $\Theta(1)$ is the answer.

5. Universal Hashing

$\mathcal{H}$ is a universal hash family if $x \neq y, Pr_{h \leftarrow \mathcal{H}}[h(x) = h(y)] = \frac{1}{|B|}$, where in this case, $|B| = 2$ because $A = \{0,1\}^n$ and $B = \{0,1\}^m$.

Starting out with the most basic case, let $m = 1$. Then our matrix h is $m \times n$. And since we know that $x, y \in A$ and $x \neq y$, at least one of the values in vector x and y must be different. Our vectors has values $x_1, ..., x_n$ and $y_1, ..., y_n$. Suppose all values are equal except for x_n and y_n. Note that $y_n - x_n \equiv 1 \pmod 2$ because of our original statement that $x \neq y$ and that all values in A are either 1 or 0.

This means that now we need to find $Pr[\sum_{i=0}^{n-1} a_i(x_i - y_i) \equiv a_n(y_n - x_n) \pmod 2]$. We can substitute in our $y_n - x_n \equiv 1 \pmod 2$ for the correct space.

Now we have $Pr[\sum_{i=0}^{n-1} a_i(x_i - y_i) \equiv a_n(1) \pmod 2] = Pr[\sum_{i=0}^{n-1} a_i(x_i - y_i) \equiv a_n \pmod 2]$. We also know that $\sum_{i=0}^{n-1} a_i(x_i - y_i)$ is a variable, let's call C such that $C \in 0, 1$ because of the given information that it is using matrix multiplication modulo 2. So our new probability is seen as $Pr[C \in \{0,1\} \equiv a_n \pmod 2]$.

There are two values a_n can be, one being that it will return the same value when calling on the hash function. This gives it the appropriate $\frac{1}{2}$ as stated by what Universal Hashing is supposed to be. Hence the $\mathcal{H}$ given in this instance is a universal hash family.

6. Mario's Workout

Main Idea

We split it down the middle continuously until we're only given a set of array of sizes 1 or 2. If the size is 1, return zero for the difference, and the platform height as both left and right. If the size is two, subtract the left platform from the right platform and return it as the difference, and return the left platform and right platform as left and right respectively. Then we compare the two left values; depending on its values, we decide whether or not to compare the difference or not. After we're done comparing, we return the best left and right values such that it gives the highest difference.

Pseudocode

```
Function BiggestJump(A):
  if array A.size is equal to 1:
    return difference: 0, left: A, right: A
  if A.size is even, split evenly: A.left and A.right
  if A.size is odd, A.left has one less value than A.right
  A.leftValues = BiggestJump(A.left)
  A.rightValues = BiggestJump(A.right)
  if A.leftValues-left < A.rightValues-left:
    return difference: (A.rightValues-right - A.leftValues-left)
    return left: A.leftValues-left
    return right: A.rightValues-right
  else if A.leftValues-difference > A.rightValues-difference:
    return difference: A.leftValues-difference
    return left: A.leftValues-left
    return right: A.leftValues-right
  else if A.leftValues-difference < A.rightValues-difference:
    return difference: A.rightValues-difference
    return left: A.rightValues-left
    return right: A.rightValues-right
```

Proof of Correctness

Case 1:
There's only one platform, then Mario can't jump. As a result, the maximum elevation difference is 0.

Case 2:
There's two platforms, Mario can only jump to his right. Therefore his maximum elevation difference is the right platform height subtract the left platform height.

Case 3:
There's more than two platforms, then he should divide the list out evenly (as much as possible) into two. He then needs to figure out the highest jump possible on the left set of platforms and the right set of platforms. He can do so by repeating the three cases. He can get the highest difference from both sides, and its left-most platform to create the highest jump, as well as the right-most platform to create the highest jump. If the left-most platform in the left set is smaller than the corresponding right set, no matter what, Mario should be able to jump more since it's shorter than the previous. Otherwise, he should

stick with whichever set has the higher difference because that's the only way to get the most optimal jump.

Running Time Analysis
$O(nlogn)$

Justification
Because the array is cut down the middle until it reaches to sizes of either 1 or 2, it will create a balanced tree. We know that the depth of a tree of this sort is $logn$. And similarly on every level we must do a comparison between at most n-numbers. So if we consider the worst case scenerio, it will do a comparison n times on $logn$ levels. This will give us a run time of $O(nlogn)$.