HW 1 - CS 120

Evan Jolley

September 2022

1

1.1

	O	o	Ω	ω	Θ
1	x	x	✓	✓	x
2	x	x	✓	✓	x
3	✓	✓	x	x	x
4	✓	x	✓	x	✓
5	x	x	✓	✓	x
6	x	x	x	x	x

1.2

1.2.1

Let c equal a constant, not all c's in the following statements are equal. We can use formulas from lecture 2 part 3.2 to help with our proof. This is what we know:

$$f(n) = \Theta(a^n)$$
$$f(n) \geq c * a^n$$
$$f(n) \leq c * a^n$$

$$g(n) = \Theta(n^b)$$
$$g(n) \geq c * n^b$$
$$g(n) \leq c * n^b$$

This is what is possibly true:

$$f(g(n)) = \Theta(a^{(n^b)})?$$
$$f(g(n)) \geq c * a^{(n^b)}$$

$$f(g(n)) \leq c * a^{(n^b)}$$

Plugging $g(n)$ into $f(n)$:

$$f(g(n)) \geq c * a^{(c*n^b)}$$

$$f(g(n)) \leq c * a^{(c*n^b)}$$

c can be any constant, so $f(g(n)) = \Theta(a^{(n^b)})$ is only true when the constant in the exponent is 1 and false at all other times. Thus, we find this statement to be false.

1.2.2

Let c equal a constant, not all c's in the following statements are equal. We can use formulas from lecture 2 part 3.2 to help with our proof. This is what we know:

$$f(n) = \Theta(a^n)$$

$$f(n) \geq c * a^n$$

$$f(n) \leq c * a^n$$

$$g(n) = \Theta(n^b)$$

$$g(n) \geq c * n^b$$

$$g(n) \leq c * n^b$$

This is what is possibly true:

$$g(f(n)) = \Theta((a^n)^b)?$$

$$g(f(n)) \geq c * (a^n)^b$$

$$g(f(n)) \leq c * (a^n)^b$$

Plugging what $f(n)$ into $g(n)$:

$$g(f(n)) \geq c * (c * a^n)^b$$

$$g(f(n)) \geq c * c^b * (a^n)^b$$

$$g(f(n)) \geq c * (a^n)^b$$

Note that $c * c^b = c$ because c simply represents a constant. This matches the "This is what is possibly true" section for all values of c. Thus, we find this statement to be true.

2

2.1

This computational problem calculates the coefficients, c_i with $0 \leq i \leq k-1$, for the polynomial

$$n = c_o + c_1 b + c_2 b^2 + ... + c_{k-1} b^{k-1}$$

for given n, b, and k. This is calculating n in base b by using the remainder of n when divided by the base for our coefficients and repeating the process until all of n has been converted.

2.2

The cardinality of the function can be 1 or 0. It is one if there is a solution, as one list is outputted. It is zero if there is no solution and $\perp$ is returned, as the cardinality of $\emptyset$ is zero.

2.3

Take the example where $f(x) = \emptyset$ and $f'(x)$ is equal to 1. $\emptyset$ is within the scope of any set by definition, so $f(x)$ is in the scope of $f'(x)$. Any algorithm that solves Π by sending I to $\emptyset$ will not succeed in solving Π', so false.

If $f(x)$ can not be $\emptyset$, this answer changes to true. This is because if $f(x)$ actually has a solution, then it must be a solution to $f'(x)$ as well. It is only the idea that $\emptyset$ is within the scope of all other sets that let's us skirt this in the first part of the problem.

3

3.1

Let's prove the correctness of this algorithm using induction.

Our base case can be when $k = 1$. This essentially reduces this algorithm down to *CountSort*, as the *BC* function would return a list of length 1, and the loop starting on line 5 will only run once. We know that *CountSort* is a valid algorithm, and thus we know that the base case will work.

Induction hypothesis: We assume that the algorithm for some $k = m$. Now we must prove that it works for $k = m + 1$.

At k=1 we sort based on the least significant digit. If the newly added, more significant digits are in the correct order based on the previous sorting, the sorting will stay the same due to the stability of *CountSort*. This will, of course

result in a correct sorting. But what if the newly added digits are not in order? Then, *CountSort* will run again, maintaining the order of the least significant digits within each new, more significant denomination.

In either of these cases, *RadixSort* results in a correct sorting, and we have proved the correctness of this algorithm.

3.2

RadixSort uses *CountingSort* multiple times, which we know has runtime $O(n + k)$ where k is the range of possible keys. We know that by checking each digit individually, there will be the same number of possible keys as b (e.g. $0 - 9$ for $b = 10$). Thus we can write that each step of *RadixSort* runs in time $O(n + b)$.

As for how many times *CountingSort* is run, it is clear that it runs the same amount as the number of digits it is checking. We can use $\lceil log_b U \rceil$ to calculate the number of digits easily (e.g. $log_{10}1000 = 3$ digits, $000 - 999$). Note the usage of the ceiling function to ensure a whole number of digits. Thus, we find that *RadixSort* runs in time $O((n + b) * \lceil log_b U \rceil)$

$$b = min(U, n)$$

$$O((n + min(U, n))log_{min(U,n)}U)$$

Case where $min(U, n) = U$:

$$O((n + U)log_U U)$$

$$\underline{O(n + U)}$$

Case where $min(U, n) = n$:

$$O((n + n)log_n U)$$

To account for the omission of the ceiling function, we must add n, as there will be n extra iterations if the *log* evaluates to its highest value in the worst case.

$$O((n + n)\frac{log_U}{logn} + n)$$

$$O(2n\frac{log_U}{logn} + n)$$

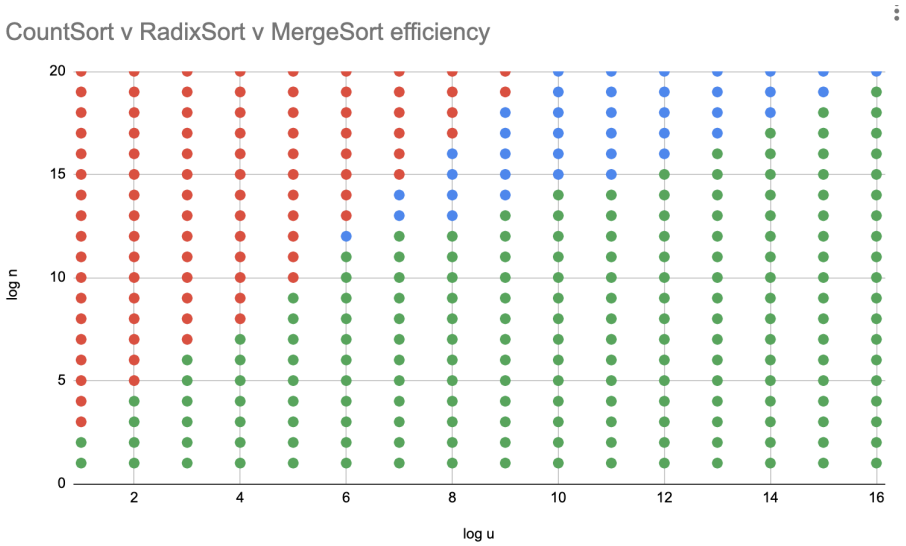
Getting rid of constants:

$$\underline{O(n\frac{log_U}{logn} + n)}$$

We now have to possible runtimes, but we know that "big O" is the worst case. As our value when $min(U, n) = n$ is slower, this is our correct runtime.

3.3

3.4



Red means $MergeSort$ was fastest, green is $CountSort$, and blue is $RadixSort$.

The movement of these curves does make sense. $CountSort$ is best with large n, $MergeSort$ is best with large U, and $Radixsort$ is in the middle. This was described in the asymptotic theory. I was worried that my poor $Radix$ code might affect results, but it is good to see that theory held.