CS120: Intro. to Algorithms and their Limitations Prof. Salil Vadhan

Problem Set 1

Harvard SEAS - Fall 2021 *Due: Wed Sep. 15, 2021 (12 noon)*

Please review the Syllabus for information on the collaboration policy, grading scale, revisions, and late days.

1. (Asymptotic Notation)

   (a) (practice using asymptotic notation) Fill in the table below to indicate the relationship between f and g. For example, if f is $O(g)$, the first cell of the row should be "True".

f	g	O	o	Ω	ω	Θ
$n!$	n^n					
$\log(n!)$	$\log(n^n)$					
2.5^n	$n^2 2^n$					
$[\log(n)]^n$	$n^{\log(n)}$					
e^n	2^{2^n}					
$n^{100} + \sin(n)$	$\sqrt[3]{n} + 100n^{99}$					

   (b) (rigorously reasoning about asymptotic notation) For each of the following claims, either justify why the statement holds (for all f, g) or provide a counterexample. In all cases, take the domain of the functions f and g to be the natural numbers (rather than the positive reals), and assume $f(n), g(n) \geq 1$ for all sufficiently large n (so that the logarithms are nonnegative).

   - If $\log(f(n)) = O(\log(g(n)))$, then $f(n) = O(g(n))$.
   - If $g(n) = o(f(n))$, then $f(n) + g(n) = \Theta(f(n))$.
   - If $f(n) \neq O(g(n))$ then $f(n) = \Omega(g(n))$.
   - For all positive constants a, b, c, if $f(n) = \Theta(n^c)$, then $f(an + b) = \Theta(n^c)$.
   - For all positive constants a, b, c, if $f(n) = \Theta(c^n)$, then $f(an + b) = \Theta(c^n)$.

2. (Radix Sort) In the Active Learning Exercise on Thursday September 7, you studied the sorting algorithm *Counting Sort*, generalized to arrays of item–key pairs, and proved that it has running time $O(n + U)$ when the keys are drawn from a universe of size U. In this problem you'll study *Radix Sort*, which improves the dependence on the universe size U from linear to logarithmic. Specifically, when $U > n$, Radix Sort can achieve runtime $O(n + n(\log U)/(\log n))$, so it achieves runtime $O(n)$ whenever $U = n^{O(1)}$. The rough idea of Radix Sort is to write the keys in base b representation for an appropriate choice of b, and then use Counting Sort to sort by the least-significant digit of the keys, then by the next-least significant digit, and so on. Crucially, Radix Sort uses the fact that Counting Sort can be

implemented in a way that is *stable* in the sense that it preserves the order in the input array when the same key appears multiple times. You can read a description of Radix Sort in CLRS Section 8.3 for the case of sorting arrays of keys (without attached items) when U and b are powers of 2, albeit using different notation than us.

(a) (designing and implementing algorithms) Provide a more detailed pseudocode description of Radix Sort (for arrays of item-key pairs) following the above outline, using Counting Sort as a subroutine. The inputs to Counting Sort are a natural number U and an array A of item-key pairs where the items are arbitrary objects and the keys are elements of $[U]$, so for each call your pseudocode makes to Counting Sort, it should explicitly say what value of U and what array A is fed in. Implement your version of Radix Sort in Python, using the implementation of Counting Sort that we will provide you in the github repository. The inputs to your implementation of Radix Sort should consist of an array of item-key pairs, the length n of the array, the universe size U, and the base b.

(b) (rigorously analyzing algorithms) Prove the correctness of your description of Radix Sort (i.e. that it correctly solves the Sorting problem), and show that it has runtime $O((n+b)\cdot\lceil\log_b U\rceil)$. Set $b=n$ to obtain our desired runtime of $O(n+n(\log U)/(\log n))$. (This runtime analysis is outlined in CLRS, but you'd need to adapt it to our notation and slightly more general setting.)

(c) (experimentally evaluating algorithms) Run experiments to compare the expected runtime of Counting Sort, Radix Sort (with base $b=n$), and Merge Sort as n and U vary among all powers of 2 from 2 to 2^{20} on arrays of length n where the keys are drawn uniformly and independently from $[U]$. (Run multiple trials for each value of n and U to estimate the expected runtime over random arrays.) For each sufficiently large value n, the asymptotic (albeit worst-case) runtime analyses suggest that Counting Sort should be the most efficient algorithm for small values of U, Merge Sort should be the most efficient algorithm for large values of U, and Radix Sort should be the most efficient somewhere in between. Plot the transition points from Counting Sort to Radix Sort, and Radix Sort to Merge Sort on a $\log_2 n$ vs. $\log_2 U$ scale. Do the shapes of the resulting transition curves fit what you'd expect from the asymptotic theory? Explain. (Note: depending on your implementation and your computer, the experiments may take hours, so be sure to carry them out well ahead of the deadline.)