CS120: Intro. to Algorithms and their Limitations	Hesterberg & Anshu

Problem Set 1

Harvard SEAS - Fall 2023 — *Due: Wed 2023-09-20 (11:59pm)*

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 with "T" (for True) or "F" (for False) to indicate the relationship between f and g. For example, if f is $O(g)$, the first cell of the row should be "T."

   Recall that, throughout CS120, all logarithms are base 2 unless otherwise specified.

f	g	O	o	Ω	ω	Θ
e^{n^2}	e^{2n^2}					
n^3	$n^{3/n}$					
$n^{2+(-1)^n}$	$\binom{n}{2}$					
$(\log n)^{120}\sqrt{n}$	n					
$\log(e^{n^2})$	$\log(e^{2n^2})$					

   (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 n.

   - For all positive integers a and b, if $f(n) = \Theta(a^n)$ and $g(n) = \Theta(n^b)$, then $f(g(n)) = \Theta(a^{(n^b)})$.
   - For all positive integers a and b, if $f(n) = \Theta(a^n)$ and $g(n) = \Theta(n^b)$, then $g(f(n)) = \Theta((a^n)^b)$.

2. (Understanding computational problems and mathematical notation)

Recall the definition of a *computational problem* from Lecture Notes 1.

Consider the following computational problem $\Pi = (\mathcal{I}, \mathcal{O}, f)$ and algorithm `BC` to solve it, where

- $\mathcal{I} = \mathbb{N} \times \mathbb{N} \times \mathbb{N}$
- $\mathcal{O} = \{(c_0, c_1, \ldots, c_{k-1}) : k, c_0, \ldots, c_{k-1} \in \mathbb{N}\}$
- $f(n, b, k) = \{(c_0, c_1, \ldots, c_{k-1}) : n = c_0 + c_1 b + c_2 b^2 + \cdots + c_{k-1} b^{k-1}, \forall i\ 0 \leq c_i < b\}$.

```
BC(n, b, k)
if b < 2 then return ⊥;
foreach i = 0, ..., k − 1 do
    c_i = n mod b;
    n = (n − c_i)/b;
if n == 0 then return (c_0, c_1, ..., c_{k−1});
else return ⊥;
```

(a) If the input is $(n, b, k) = (11, 10, 4)$, what does the algorithm BC return? (Note that the output is not (1,1).) Is BC's output a valid solution for Π with input $(11, 10, 4)$?

(b) Describe the computational problem Π in words. (You may find it useful to try some more examples with $b = 10$.)

(c) Is there any $x \in \mathcal{I}$ for which $f(x) = \emptyset$? If so, give an example; if not, explain why.

(d) For each possible input $x \in \mathcal{I}$, what is $|f(x)|$? ($|A|$ is the size of a set A.) Justify your answer(s) in one or two sentences.

(e) Let $\Pi' = (\mathcal{I}, \mathcal{O}, f')$ be the problem with the same $\mathcal{I}$ and $\mathcal{O}$ as Π, but $f'(n, b, k) = f(n, b, k) \cup \{(0, 1, \ldots, k-1)\}$. Does every algorithm A that solves Π also solve Π'? (Hint: any differences between inputs that were relevant in the previous subproblem are worth considering here.) Justify your answer with a proof or a counterexample.

3. (Radix Sort) In the Sender–Receiver Exercise associated with lecture 3, you studied the sorting algorithm *Counting Sort*, generalized to arrays of key–value 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, Radix Sort can achieve runtime $O(n+n(\log U)/(\log n))$, so it achieves runtime $O(n)$ whenever $U = n^{O(1)}$. Radix Sort is constructed by using Counting Sort as a subroutine several times, but on a smaller universe size b. 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. Here is pseudocode for Radix Sort, using the algorithm BC above as a subroutine:

1 `RadixSort`(U, b, A)

Input : A universe size $U \in \mathbb{N}$, a base $b \in \mathbb{N}$ with $b \geq 2$, and an array $A = ((K_0, V_0), \ldots, (K_{n-1}, V_{n-1}))$, where each $K_i \in [U]$

Output : A valid sorting of A

2 $k = \lceil (\log U)/(\log b) \rceil$;

3 **foreach** $i = 0, \ldots, n-1$ **do**

4     $V_i' =$ `BC`(K_i, b, k)

5 **foreach** $j = 0, \ldots, k-1$ **do**

6   **foreach** $i = 0, \ldots, n-1$ **do**

7         $K_i' = V_i'[j]$

8     $((K_0', (V_0, V_0')), \ldots, (K_{n-1}', (V_{n-1}, V_{n-1}'))) =$ `CountingSort`$(b, ((K_0', (V_0, V_0')), \ldots, (K_{n-1}', (V_{n-1}, V_{n-1}')))$;

9 **foreach** $i = 0, \ldots, n-1$ **do**

10     $K_i = V_i'[0] + V_i'[1] \cdot b + V_i'[2] \cdot b^2 + \cdots + V_i'[k-1] \cdot b^{k-1}$

11 **return** $((K_0, V_0), \ldots, (K_{n-1}, V_{n-1}))$

Algorithm 1: Radix Sort

(You can also 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) (proving correctness of algorithms) Prove the correctness of `RadixSort` (i.e. that it correctly solves the Sorting problem).

Hint: You will need to use the stability of `CountingSort` in your argument. Note that if in the 8th line of `RadixSort` algorithm, you replaced `CountingSort` with ExhaustiveSearchSort (or any other sort which isn't stable), the resulting algorithm would not correctly solve sorting.

Here is an example (using ExhaustiveSearchSort instead of stable sort in line 8). Suppose $n = 3, b = 2, U = 4$, $K_0 = 1, K_1 = 3, K_2 = 2$ and V_0, V_1, V_2 are "a", "b", and "c". Then $V_0' = (1, 0), V_1' = (1, 1), V_2' = (0, 1)$. Suppose ExhaustiveSearchSort is such that the permutation $\pi(2) = 0, \pi(1) = 1, \pi(0) = 2$ is tried first. Sorting based on the first bit will lead to the array $(K_2 = 2, (c, (0, 1))), (K_1 = 3, (b, (1, 1))), (K_0 = 1, (a, (1, 0)))$. Next, sorting the second bit using the same ExhaustiveSearchSort will give the array $(K_0 = 1, (a, (1, 0))), (K_1 = 3, (b, (1, 1))), (K_2 = 2, (c, (0, 1)))$. Thus we return the same input array $((1, a), (3, c), (2, b))$!

(b) (analyzing runtime) Show that `RadixSort` has runtime $O((n+b) \cdot \lceil \log_b U \rceil)$. Set $b = \min\{n, U\}$ 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) (implementing algorithms) Implement `RadixSort` using the implementations of `CountingSort` and `BC` that we provide you in the GitHub repository.

(d) (experimentally evaluating algorithms) Run experiments to compare the expected runtime of `CountingSort`, `RadixSort` (with base $b = n$), and `MergeSort` as n and U vary among powers of 2 with $1 \leq n \leq 2^{16}$ and $1 \leq U \leq 2^{20}$. For each pair of (n, U) values you consider, run multiple trials to estimate the expected runtime over random arrays where the keys are chosen uniformly and independently from $[U]$. For each sufficiently large value of n, the asymptotic (albeit worst-case) runtime analyses suggest that `CountingSort` should be the most efficient algorithm for small values of U, `MergeSort` should be the most efficient algorithm for large values of U, and `RadixSort` should be the most efficient somewhere in between. Plot the transition points from `CountingSort` to `RadixSort`, and `RadixSort` to `MergeSort` on a $\log n$ vs. $\log U$ scale (as usual our logarithms are base 2). Do the shapes of the resulting transition curves fit what you'd expect from the asymptotic theory? Explain.

Note: We are expecting to see one (or more, if necessary) graphs that demonstrate, for every value of n, for which value of U `RadixSort` *first outperforms* `CountingSort` *and* `MergeSort` *first outperforms* `RadixSort`. *You should label the graphs appropriately (title, axis labels, etc.) and provide a caption, as well as an answer and explanation to the above question. Please look at the provided starter code for more information on generating random arrays, timing experiments, and graphing. Your implementation of RadixSort, as well as any code you write for experimentation and graphing need not be submitted. Depending on your implementation, running the experiments could take anywhere from 15 minutes to a couple of hours, so don't leave them to the last minute!*