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}	T	T	F	F	F
n^3	$n^{3/n}$	F	F	T	T	F
$n^{2+(-1)^n}$	$\binom{n}{2}$	F	F	F	F	F
$(\log n)^{120}\sqrt{n}$	n	F	F	T	T	F
$\log(e^{n^2})$	$\log(e^{2n^2})$	T	F	T	F	T

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

Proof. For both statements, we will apply the limit definition of $f = \Theta(g)$, $0 < \lim_{x\to\infty} \frac{f(x)}{g(x)} < \infty$. This tells us that the limit must equal some positive constant c, which means that f and g must grow at the same rate.

If $f(n) = \Theta(a^n)$ and $g(n) = \Theta(n^b)$, then it follows that $f(n) = c_1 \cdot a^n$ and $g(n) = c_2 \cdot n^b$ for some constants $c_1, c_2 > 0$ since f and g in both cases must grow at the same rate (this can also be verified by plugging in the respective values into the limit definition described above, which would yield $\lim_{x\to\infty} \frac{c_1 \cdot a^n}{a^n} = c_1$ and $\lim_{x\to\infty} \frac{c_2 \cdot n^b}{n^b} = c_2$, both of which satisfy the definition of Θ). Now we plug in these equations for $f(n)$ and $g(n)$ into $f(g(n)) = \Theta(a^{(n^b)})$ and apply the limit definition, which gives us

$$0 < \lim_{x\to\infty} \frac{c_1 \cdot a^{c_2 \cdot n^b}}{a^{(n^b)}} < \infty.$$

We can see that the equation does not hold for any value of $c_2 \neq 1$ because the limit will equal infinity, instead of approaching the constant c_1. For example, plugging in the values $a = 2, c_2 = 3, b = 4$, and $c_1 = 5$ would result in the simplified limit $5 \cdot \lim_{x\to\infty} 4^{x^4}$ which equals ∞. Hence, we have shown that the first claim does not hold via counterexample.

Applying the same logic from the first claim to the second, it still holds that $f(n) = c_1 \cdot a^n$ and $g(n) = c_2 \cdot n^b$ for some constants $c_1, c_2 > 0$. Plugging these equations into $g(f(n)) = \Theta((a^n)^b)$ and applying the limit definition, we get

$$0 < \lim_{x\to\infty} \frac{c_2(c_1 \cdot a^n)^b}{(a^n)^b} < \infty.$$

Since the exponential values, n and b, are held constant in both the numerator and the denominator, we know that the numerator and denominator must be of the same degree meaning that they grow at the same rate and thus, the limit cannot equal ∞ nor 0 for all positive integers a and b. Since c_1 and c_2 are positive constants, we know that the limit must converge to some positive constant c, which satisfies the definition of Θ. Hence, the second claim holds for all f and g. $\square$

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 $\perp$;
foreach $i = 0, \ldots, k-1$ do
    $c_i = n \bmod b$;
    $n = (n - c_i)/b$;
if $n == 0$ then return $(c_0, c_1, \ldots, c_{k-1})$;
else return $\perp$;
```

(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 $(11, 10, 4)$?

For the given input, $(n, b, k) = (11, 10, 4)$, the algorithm BC returns (1, 1, 0, 0). This output is a valid solution for Π with input (11, 10, 4) because when plugging in the corresponding input and output values into $n = c_0 + c_1 b + c_2 b^2 + \cdots + c_{k-1} b^{k-1}$, we get

$$11 = 1 + 1(10) + 0(10^2) + 0(10^3).$$

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

The computational problem Π takes in an array consisting of three natural numbers as its input, $\mathcal{I}$, which we will denote as (n, b, k). The set of outputs, $\mathcal{O}$, is an array consisting of k elements which are all natural numbers. The set of solutions, $f(n, b, k)$, is an array of k elements that represent n in terms of base-b numerals. For instance, if we were to consider more examples with $b = 10$, then (2, 1, 0, 0), (6, 5, 0, 0), and (3, 2, 0, 2) would all be valid solutions to inputs corresponding to (12, 10, 4), (56, 10, 4), and (2023, 10, 4) respectively. In the case of b=10 with the solution set array, $(c_0, c_1, c_2, ..., c_{k-1})$, c_0 represents the ones place, c_1 represents the tens place, c_2 represents the hundreds place and so on for the n value in the input array.

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

If we are again considering the case where $b = 10$, then an example of an input x for which $f(x) = \emptyset$ are inputs where the k-value is less than the number of digits in n.

For instance, if $k = 4$ but n is a five-digit number, it would be impossible to represent n in base-10 with an array consisting of only 4 elements (since we can't represent the ten-thousands place in (c_0, c_1, c_2, c_3) without the c_4 element). Hence, the solutions set for the example described above would be the empty set, or equivalently, $f(x) = \emptyset$.

(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.

For each possible input $x \in \mathcal{I}$, $|f(x)|$, or the size of the solution set, is either 1 or 0. Every $x \in \mathcal{I}$ where $f(x) \neq \emptyset$ has a solution set that consists of a single array so in that case, $|f(x)| = 1$, whereas for inputs for which $f(x) = \emptyset$ (such as the one described in (c)), $|f(x)| = 0$ since the size of the emptyset is 0.

(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.

Every algorithm A that solves Π does not also solve Π'. Consider inputs $x \in \mathcal{I}$ for which $f(x) = \emptyset$ such as the one described in (c) or inputs with $b < 2$. In this case, an algorithm A that solves Π, such as BC, would return $\perp$, or equivalently $A(x) = \perp$ since the solution set (f) is empty. However, for the same such inputs, the solution set of Π' would consist of $\emptyset \cup \{(0, 1, \ldots, k-1)\}$. Since the union of any set with the emptyset is the set itself, it follows that $f'(x) = \{(0, 1, \ldots, k-1)\}$. Recall that an algorithm A that solves Π in this case would result in $A(x) = \perp$, which can only happen when the set of solutions is the empty set. $f'(x) \neq \emptyset$ and $A(x) \notin f'(x)$, hence the claim does not hold.

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      $V'_i = \mathtt{BC}(K_i, b, k)$
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 $((1,a), (3,c), (2,b))$!

Proof. We will prove the correctness of Radix Sort by utilizing induction.

Base Case: For our base case, we have $k = 0$. Since k corresponds to the place value that we sort up to for base b (the number of digits used to sort the keys in A in base b), this means that we are calling Counting Sort on no digits. Another way to look at it is by recalling line 2 from the pseudocode above. We can see that k can only equal 0 if equivalently, $\log U = 0$. This can only occur for a universe size $U = 1$, which means that there is only one key-value pair within the input array. Since an array consisting of one element is already sorted by default, Counting Sort returns a valid sorting for $k = 0$. Hence, the base case holds.
Inductive Hypothesis: Suppose that for some arbitrary natural number p, Radix Sort will return a valid sorting of an input array A for all $k \leq p$.
Inductive Step: Consider the $p + 1$th place of the keys within A. By the inductive hypothesis, we know that the first p digits of the keys within A are properly sorted while maintaining stability. When applying Counting Sort to the $p + 1$th digit of the keys, we must consider the following cases:
Case 1: If two keys have different $p + 1$th digit values, then Counting Sort will sort the keys properly (in increasing order by the $p + 1$th digit value), which will allow for a correct sorting via Radix Sort.
Case 2: If two or more keys have the same $p + 1$th digit value, then by the stability of Counting Sort, the keys will be listed in the order that they were called into the Counting Sort (preserving the order in the input array).

Recall that the previous p digits of they keys are also properly sorted while continuously maintaining the stability of the input array (inductive hypothesis). Hence, this valid and stable sorting of the $p + 1$th digits will result in Radix Sort returning a correct sorting of the entire array A.

Since the base case and inductive step both hold, by induction we have proved the correctness of Radix Sort.

□

(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.)

By looking at the pseudocode provided above, we can see that the runtime of Radix Sort is $O(n) + O(k)O(n) + O(k)O(n + b) + O(n)$. The first $O(n)$ corresponds to the for loop on line 3, the $O(k)O(n)$ corresponds to the nested for loops on line 5 and 6, the $O(k)O(n + b)$ corresponds to the Counting Sort call on line 8 (since Counting Sort has running time $O(n + U)$ but within the context of Radix Sort it is applied to a smaller universe size b), and the last $O(n)$ is the for loop on line 9.

To determine the asymptotic growth of Radix Sort, we only need to consider the fastest growing part of the equation which is $O(k)O(n+b)$. From line 2, we know that $k = \lceil \log U / \log b \rceil$, or equivalently, $k = \lceil \log_b U \rceil$. Plugging in k, we get $O((n+b) \cdot \lceil \log_b U \rceil)$.

We will now show that when $b = \min\{n, U\}$, we will obtain our desired runtime of $O(n + n(\log U)/(\log n))$. We must consider the following cases:
Case 1: Let $n < U$, then $b = n$. Since big-O describes the worst case or upper bound running time complexity, we can substitute $\log_b U + 1$ for $\lceil \log_b U \rceil$ (we know that $\lceil \log_b U \rceil < \log_b U + 1$ by definition of ceiling). This gives us

$$\begin{aligned}
O((n+b) \cdot (\log_b U + 1)) &= O((n+n) \cdot (\log_n U + 1)) \\
&= O((n) \cdot (\log_n U + 1)) \\
&= O((n)(\log_n U) + n) \\
&= O(n + n(\frac{\log U}{\log n}))
\end{aligned}$$

Case 2: Let $n > U$, then $b = U$. Applying the same logic from case one, we again substitute $\log_b U + 1$ for $\lceil \log_b U \rceil$ as well as $b = U$ to get

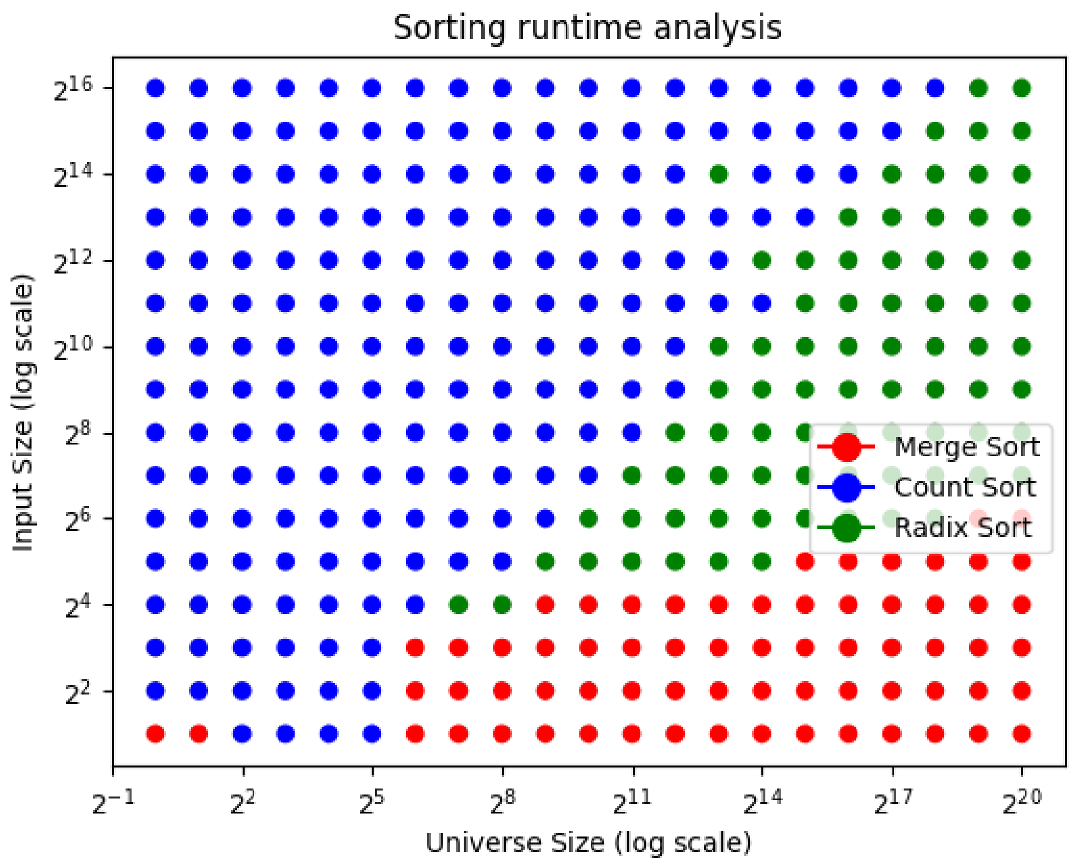
$$\begin{aligned}
O((n+b) \cdot (\log_b U + 1)) &= O((n+U) \cdot (\log_U U + 1)) \\
&= O((n+U) \cdot (1+1)) \\
&= O(n)
\end{aligned}$$

We know that $O(n)$ grows at a slower rate than $O(n + n(\log U)/(\log n))$, or equivalently, $O(n) < O(n + n(\log U)/(\log n))$. Hence, we have shown that Radix Sort has runtime $O(n + n(\log U)/(\log n))$ when $n < U$.

(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!*



This graph represents the results from the experiments when I set N or the number of trials to run for each algorithm to 100. For the most part, the shapes of the transition curves fit what we would expect from asymptotic theory. Although there were some outliers, Counting Sort was the most efficient algorithm for small values of U, Merge Sort was fastest for large values of U (values above 2^5 on the log scale), and Radix Sort was the most efficient somewhere in between, which are the exact results that the asymptotic runtime analyses suggests.