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

Problem Set 1

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

Samuel Osa-Agbontaen

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	T	T	F	F	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 $f(g(n)) = \Theta(a^{(n^b)})$. This statement does not hold because there is a counterexample for which $f(g(n)) \notin \Theta(a^{(n^b)})$. Specifically, let $a = 2$, $b = 2$, $f(n) = 2^n$, and $g(n) = 7n^2$. In this case, it follows that $f(g(n)) = 2^{(7n^2)}$ and since the $\lim_{n\to\infty} \frac{2^{7n^2}}{2^{n^2}} = \lim_{n\to\infty} 2^{6n^2} = \infty$, we know that $f(g(n))$ is not bounded by $(a^n)^b$, meaning $f(g(n)) \notin O((a^n)^b)$, which means it is also not $\Theta((a^n)^b)$.
   - For all positive integers a and b, if $f(n) = \Theta(a^n)$ and $g(n) = \Theta(n^b)$, then $g(f(n)) = \Theta((a^n)^b)$. This statement holds. First, we define what we know is true. Since $f(n) = \Theta(a^n)$, we know it is also $O(a^n)$,which means $\exists c_0, n_0$ s.t. $f(n) \leq c_0 \cdot a^n$ for all $n > n_0$. If its $\Theta(a^n)$, this means it is also $\Omega(a^n)$, which means that $\exists c_3, n_0$ s.t. $f(n) \geq c_3 \cdot a^n$. Also, since $g = \Theta(n^b)$, we know that $g(n) = O(n^b)$, which means $\exists c_1, n_1$ s.t. $g(n) \leq c_1 \cdot n^b$ for all $n > n_1$. It also means that $g(n) = \Omega(n^b)$ which means $\exists c_4, n_1$ s.t. $g(n) \geq c_4 \cdot n^b$ for all $n > n_1$

First, we can start by showing that $g(f(n)) = O((a^n)^b)$. To start, we can substitute into the formula for $g(n) = O(n^b)$ where we take $f(n)$ as input to g, resulting in $\exists c_1, n_1$ s.t. $g(f(n)) \leq c_1 \cdot (f(n))^b$ for all $n > n_1$.

Since we are reasoning about upper bounds, we can substitute the upper bound for $f(n)$ into this inequality as defined earlier as $c_0 \cdot a^n$, resulting in $g(c_0 \cdot a^n) \leq c_1 \cdot (c_0 \cdot (a^n))^b$. After simplifying, this comes out to $g(c_0 \cdot a^n) \leq c_1 \cdot c_0^b \cdot (a^n)^b$. If we define $c_2 = c_1 \cdot c_0^b$, we can write $g(c_0 \cdot a^n) \leq c_2 \cdot (a^n))^b$, which we can rewrite as $g(f(n)) \leq c_2 \cdot (a^n))^b$ since we are just substituting the upper bound of $f(n)$ out with $f(n)$. This inequality is now the definition of being upper bounded by $(a^n)^b$. In other words, it shows that $g(f(n)) = O((a^n)^b)$.

Next, we need to show that $g(f(n)) = \Omega((a^n)^b)$. To start, we can substitute into the formula for $g(n) = \Omega(n^b)$ where we take $f(n)$ as input to g, resulting in $\exists c_4, n_1$ s.t. $g(f(n)) \geq c_4 \cdot (f(n))^b$ for all $n > n_1$.

Since we are reasoning about lower bounds, we can substitute the lower bound for $f(n)$ into this inequality as defined earlier as $c_3 \cdot a^n$, resulting in $g(c_3 \cdot a^n) \geq c_4 \cdot (c_3 \cdot (a^n))^b$. After simplifying, this comes out to $g(c_3 \cdot a^n) \geq c_4 \cdot c_3^b \cdot (a^n)^b$. If we define $c_5 = c_4 \cdot c_3^b$, we can write $g(c_3 \cdot a^n) \geq c_5 \cdot (a^n)^b$, which we can rewrite as $g(f(n)) \geq c_5 \cdot (a^n)^b$ since we are just substituting the lower bound of $f(n)$ out with $f(n)$. This inequality is now the definition of being lower bounded by $(a^n)^b$. In other words, it shows that $g(f(n)) = \Omega((a^n)^b)$.

By the definition of Θ, since we've shown that $g(f(n)) = O((a^n)^b)$ and $g(f(n)) = \Omega((a^n)^b)$, we can now also say that $g(f(n)) = \Theta((a^n)^b)$. Therefore, the original statement holds.

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

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

```
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 $(11, 10, 4)$? The output is $(1, 1, 0, 0)$. This is valid because it satisfies the condition $n = c_0 + c_1 b + c_2 b^2 + \cdots + c_{k-1} b^{k-1}, \forall i\ 0 \le c_i < b$. Specifically, n is already given to be as 11, and the right hand side comes out to $1 + 1(10) + 0 + 0 = 11$.

(b) Describe the computational problem Π in words. (You may find it useful to try some more examples with $b = 10$.) Π returns the first k digits of n in base b composition from lowest to highest place value.

(c) Is there any $x \in \mathcal{I}$ for which $f(x) = \emptyset$? If so, give an example; if not, explain why. Yes, an example is when $x = (120, 10, 1)$. Since k is less than the highest place value of n in base 10, the algorithm cannot represent each place value. It is only expecting to represent 120 in 1 digit, but in reality it will need at least 3 digits.

(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.

$$|f(x)| = \begin{cases} 1 & \text{if a solution exists} \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

This is true because (given the same k) there is only one way to decompose n in base b or no way at all.

(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. No, there exists an algorithm A that solves Π but does not solve Π'. For instance, the algorithm given in the problem spec. for BC returns $\perp$ in line 7, meaning that it accounts for cases when there is no solution. However, this same algorithm would not solve Π' since Π' always expects a result. Regardless of whether or not there is a valid list to return, Π' will still expect the set of numbers from 0 to $k-1$.

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:

```
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
         $A = ((K_0, V_0), \dots, (K_{n-1}, V_{n-1}))$, where each $K_i \in [U]$
Output : A valid sorting of $A$
$k = \lceil (\log U)/(\log b) \rceil$;
foreach $i = 0, \dots, n-1$ do
    $V'_i = \mathtt{BC}(K_i, b, k)$
foreach $j = 0, \dots, k-1$ do
    foreach $i = 0, \dots, n-1$ do
        $K'_i = V'_i[j]$
    $((K'_0, (V_0, V'_0)), \dots, (K'_{n-1}, (V_{n-1}, V'_{n-1}))) =$
      CountingSort($b, ((K'_0, (V_0, V'_0)), \dots, (K'_{n-1}, (V_{n-1}, V'_{n-1})))$);
foreach $i = 0, \dots, n-1$ do
    $K_i = V'_i[0] + V'_i[1] \cdot b + V'_i[2] \cdot b^2 + \cdots + V'_i[k-1] \cdot b^{k-1}$
return $((K_0, V_0), \dots, (K_{n-1}, V_{n-1}))$
```

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 $((1,a), (3,c), (2,b))$! We will prove the correctness of RadixSort by inducting on the number of digits (place values) we will have to sort.

Let the base case be when we have to sort 0 digits. This case is trivially true because there is nothing that needs to be sorted.

Our inductive hypothesis assumes that on digit k, digit $k-1$ is already sorted.

For our inductive step, we will show that k will always be sorted, regardless of future sorts of subsequent place values/digits. We know this will hold because our implementation of Radix sort relies on Counting Sort, which is a stable sorting algorithm. To elaborate, it ensures that when we sort through digit k, we will not modify the sorting we did of digit $k-1$. Since Radix Sort involves calling counting sort at each place value/digit, we can be sure that each previous digit/place value that has already been sorted will stay sorted, even after we sort the current place value/digit.

Therefore, by mathematical induction, RadixSort is correct.

(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.) The runtime of Radix Sort is dominated by the calls made to Counting Sort, which is called k times in the algorithm. Since Counting Sort is $O(n+U)$ and Radix Sort calls it k times, we can say that Radix Sort is $O((n+U)\cdot k)$. We can substitute $U = b$, because for each place value, the universe size is each possible distinct number at that base (for example, in base 10, the universe ranges from 0 to 9). Also, we can substitute $k = \lceil logU/logb\rceil$, since this is defined in the algorithm in line 2. Applying the change of base formula, we can say that $\lceil logU/logb\rceil = log_b U$. This gives us a runtime of $O((n+b)\cdot\lceil log_b U\rceil)$.

Next, we show that regardless of whether or not $b = n$ or $b = U$, the runtime will still be $O(n+n(\log U)/(\log n))$.

<u>For case $b = U$:</u>

$$\begin{aligned}
O((n+b)\cdot log_b U) &= O((n+U)\cdot log_U U) && \text{Initial equation} && (2)\\
&= O((n+U)\cdot logU/logU) && \text{Change of base formula} && (3)\\
&= O(n+U) && \text{Simplify} && (4)\\
&= O(n+n) && \text{since in this case, } b = min\{n,u\}\text{, we know } U < n && (5)\\
&= O(2n) && \text{Simplify} && (6)\\
&= O(n) && \text{Coefficients don't matter in runtime} && (7)
\end{aligned}$$

For case $b = n$:

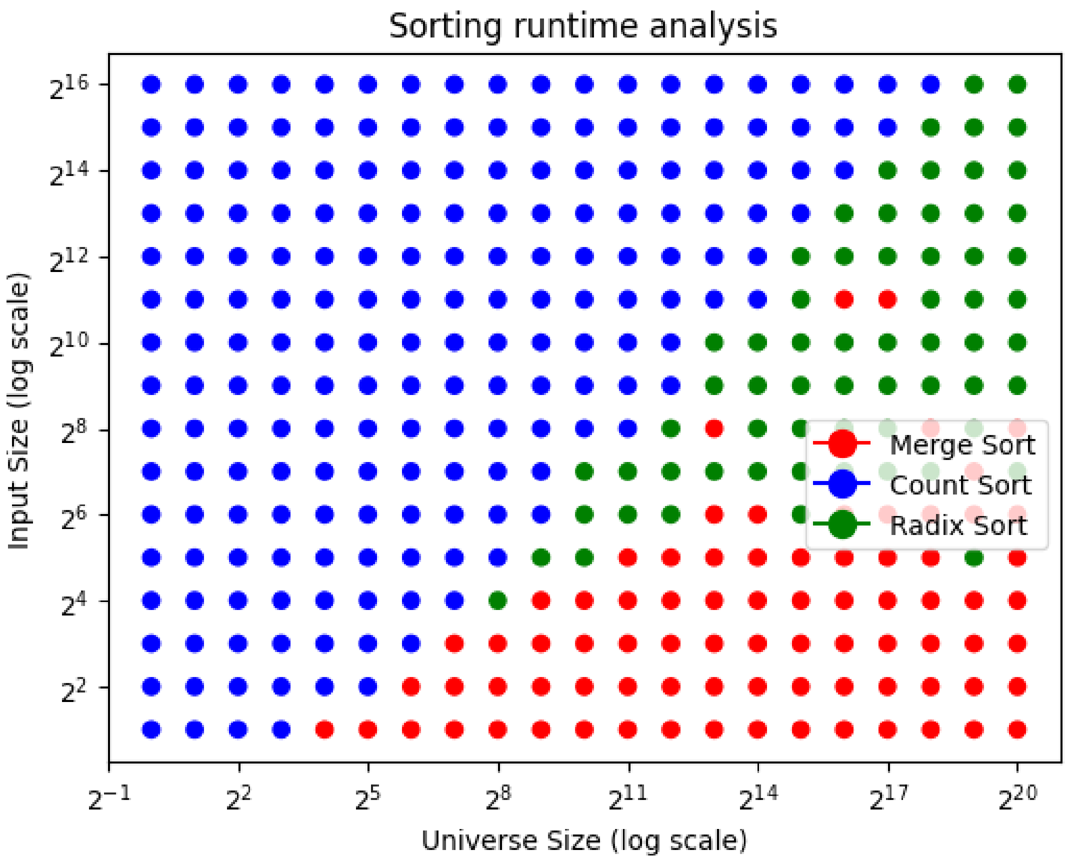
$$\begin{aligned}
O((n+b)\cdot log_b U) &= O((n+n)\cdot log_n U) && \text{Initial equation} && (8)\\
&= O(2n \cdot log_n U) && \text{Simplify} && (9)\\
&= O(2n \cdot \frac{logU}{logn}) && \text{Change of Base Formula} && (10)\\
&= O(n \cdot \frac{logU}{logn}) && \text{Coefficients don't matter in runtime} && (11)
\end{aligned}$$

We know that the runtime of each case will be bounded by the runtime of both cases combined, so we can say that both cases are O(runtime of case 1 + runtime of case 2). In other words, $O(n + \frac{nlogU}{logn})$, which is our desired runtime.

(c) (implementing algorithms) Implement `RadixSort` using the implementations of `CountingSort` and `BC` that we provide you in the GitHub repository. Implemented in code. Reflected in graph in part d.

(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!* The graph below shows that for small values of U, Counting Sort is the most efficient algorithm. However, as U increases, Merge sort becomes the most efficient, but when the input size increases as U also increases, Radix Sort becomes the most efficient. This graph fits what is expected from asymptotic theory because as the input size increases, Counting Sort's linear runtime proves to be reliable, while Radix Sort's near linear runtime also proves reliable, but Merge Sort's nlogn runtime makes the algorithm less reliable.


Sorting runtime analysis
Input Size (log scale)
Universe Size (log scale)
2^{16}
2^{14}
2^{12}
2^{10}
2^{8}
2^{6}
2^{4}
2^{2}
2^{-1}
2^{2}
2^{5}
2^{8}
2^{11}
2^{14}
2^{17}
2^{20}
Merge Sort
Count Sort
Radix Sort