CS1200: Intro. to Algorithms and their Limitations Anshu & Vadhan

Problem Set 1

Harvard SEAS - Fall 2024 *Due: Wed 2024-09-18 (11:59pm)*

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

Unfortunately, we made a few mistakes in our previous iteration of trying to use GitHub Classroom–this time, we have (hopefully) figured out those wrinkles and have a new GitHub Classroom assignment for you all to accept. If you were initially using a fork of the public repo, we recommend using Classroom, since we'll be able to post things like practice materials, section notes, and problem set solutions which we usually don't want to display on the public repository. This time you should be able to easily access updates by viewing your repo on the website, clicking on the pull request and accepting it (more details on Ed to come).

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 $\Omega(g)$, the first cell of the row should be "T." No justification necessary. Notice that some of the functions are the same as in Problem Set 0.

   Recall that, throughout CS1200, all logarithms are base 2 unless otherwise specified.

f	g	Ω	ω	Θ
$3\log^3 n$	n^2+1	F	F	F
$4n^3$	$\lvert\{S \subseteq [n] : \lvert S\rvert \leq 3\}\rvert$	T	F	T
$n!$	5^n	T	T	F
3^n	$(2+(-1)^n)^n$	T	T	F

   (b) (runtimes: $T^=$ vs. T) Let $g : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$ be a nondecreasing function, i.e. if $x \geq y$, then $g(x) \geq g(y)$. (For example $g(n) = n^2$, $g(n) = n \log n$, or $g(n) = 2^n$.) Let $T^= : \mathbb{N} \to \mathbb{N}$ and $T : \mathbb{R}^{\geq 0} \to \mathbb{N}$ be the runtimes of an algorithm A, as defined in Lecture 2.

      i. Prove that $T^= = O(g)$ iff $T = O(g)$. Thus the distinction between T and $T^=$ does not matter when we are interested in natural runtime bounds, which are nondecreasing.

      To prove that $T^= = O(g)$ iff $T = O(g)$ we have to go both directions.
      Forward: $T^= = O(g) \to T = O(g)$
      We want to prove that $T = O(g)$ given $T^= O(g)$ which means we know that $T^=(x) \leq c \cdot g(x)$ for some c. Since $T(x)$ is on any non-negative real number, we can find a $n = \lceil x \rceil$ which implies (since g is non-decreasing) that $g(n) \geq g(x)$. Thus we know that $T^=(x) \leq T(n)$ since $n = \lceil x \rceil$ (i.e. you round up x to the nearest natural number to get n). From there we show that since we know $T^=(n) \leq c \cdot g(n)$ and

$T(x) \leq T^{=}(n)$ then $T(x) \leq T^{=}(n) \leq c \cdot g(n) \to T(x) \leq c \cdot g(n) \to T = O(g)$.
***Backward** $T = O(g) \to T^{=}O(g)$*
This one is easier, because since $T^{=}(n) = T(n)$ for natural numbers n and we are given $T = O(g)$ or $T(n) \leq c \cdot g(n)$ then $T^{=}(n) = T(n) \leq g(n) \to T^{=}(n) \leq c \cdot g(n)$ or $T^{=} = O(g)$.

ii. The same equivalence does not hold in general if we replace $O(\cdot)$ with $\Omega(\cdot)$. Which direction ($T^{=} = \Omega(g) \Rightarrow T = \Omega(g)$ or $T = \Omega(g) \Rightarrow T^{=} = \Omega(g)$) fails? Give an example of a potential runtime $T^{=}$ and a function g to demonstrate. (Hint: one of the pairs of functions in the table above may be helpful.)

In this case $\Omega(\cdot)$ implies $T(n) \geq c \cdot g(n)$ for some constant $c > 0$. Importantly $T(x)$ implies the worst case scenario for all values up to x whereas $T^{=}(n)$ only considers inputs of exactly size n. This means in situations where odd versus even values differ in worst case scenarios, $T^{=}$ suffers. Take $g(n) = (2 + (-1)^n)^n$ where when n is odd we get $g(n) = 1$. If it is even we get $g(n) = 3^n$. So even though if we know $T = \Omega(g)$ which takes into account all values up to x, (from our example worst case would be when x is even and large $\to 3^n$), it does not imply $T^{=} = \Omega(g)$ since it can only recognize inputs of exactly size n, which in cases where n is odd, means $g(n) = 1$ which if n is still large means $T^{=} = 3^n$ will be at least larger than $g(n)$ for sufficiently large n (i.e. $T^{=} \geq c \cdot g(n)$ fails in those cases and is not implied by $T = \Omega(g)$).

2. (Understanding computational problems and mathematical notation)

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

Consider the following computational problem $\Pi = (\mathcal{I}, \mathcal{O}, f)$:

- $\mathcal{I} = \mathbb{N} \times \mathbb{N}^{\geq 2} \times \mathbb{N}$, where $\mathbb{N}^{\geq 2} = \{2, 3, 4, \ldots\}$.
- $\mathcal{O} = \{(c_0, c_1, \ldots, c_{k-1}) : k, c_0, \ldots, c_{k-1} \in \mathbb{N}\}$
- $f(n, b, k) = \{(c_0, c_1, \ldots, c_{k-1}) : n = c_0 + c_1 b + c_2 b^2 + \cdots + c_{k-1} b^{k-1}, \forall i \; 0 \leq c_i < b\}$.

Here is an algorithm BC to solve Π:

```
BC($n, b, k$)
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 $(n, b, k) = (35, 10, 4)$, what does the algorithm BC return? Is BC's output a valid answer for Π with input $(35, 10, 4)$?

First for $i = 0$, we know $c_0 = 35 \mod 10 = 5$ so then n gets updated to $n = (35-5)/10 = 3$ which for next loop $k = 1$ we get $c_1 = 3 \mod 10 = 3$ so again n gets updated to $n = 0$ and for next two loops $c_i = 0$, completing output. Thus algorithm BC *returns* $(5, 3, 0, 0)$ *which is a valid representation of 35 in base-10 across 4 digits. It is a valid representation because $f(35, 10, 4)$ produces a set $(5, 3, 0, 0)$ which is a valid sorting of O(i.e. is included in Oand $n = c_0 + c_1 b + c_2 b^2 + \cdots + c_{k-1} b^{k-1}$.*

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

The computational problem essentially asks for a number n and wants it broken down into k base-b elements. In other words, it seeks to break down n according to a base-b, in the case that $b = 10$, it is a base-10 system like our number system, breaking it down into its ones, then tens, then hundredths... places. However, it only does the first k base-b elements, so for $k = 10$ and $b = 2$ it only includes the ones and tens place.

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

Yes there is an $x \in \mathcal{I}$ for which $f(x) = \emptyset$. It occurs when k is not large enough sufficiently satisfy $n = c_0 + c_1 b + c_2 b^2 + \cdots + c_{k-1} b^{k-1}$. An example is $f(35, 10, 1)$ where n cannot be represented by a base 10 system in 1 term.

(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 $x \in \mathcal{I}$ $|f(x)|$ is $\in \{0,1\}$, which means the size of the outputs of f are either 0, or 1. This is because there are either no outputs to that input (e.g. what we saw in 2c.) which means $|f(x)| = 0$ or there is 1 possible output. Since there is only 1 way to represent an integer as a sum of powers b with constraint on the digits, $|f(x)|$, should there be a valid solution is 1 (i.e. there is only one solution for every input that has a valid output).

(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' because there are algorithms like `BC` which produce $\emptyset$ for Π but do not produce $\emptyset$ for Π'. This is because even though the 'space' of Π's $\mathcal{I}$ and $\mathcal{O}$ are the same as Π' the solutions $f'(n,b,k)$ are not the same. Take $(35,10,1)$ which produced $f(x) = \emptyset$, however, $f'(x) = f(x) \cup (0) = (0)$ which is not the empty set. Thus since `BC` solves Π and (35,10,1) produce different results for Π and Π' then we prove by counterexample that very algorithm A that solves Π does not also solve Π'.

3. (Radix Sort) In the Sender–Receiver Exercise associated with lecture 3, you studied the sorting algorithm `SingletonBucketSort`, 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 `RadixSort`, which improves the dependence on the universe size U from linear to logarithmic. Specifically, `RadixSort` can achieve runtime $O(n + n \cdot (\log U)/(\log n))$, so it achieves runtime $O(n)$ whenever $U = n^{O(1)}$.

   `RadixSort` is constructed by using `SingletonBucketSort` as a subroutine several times, but on a smaller universe size b. Specifically, it turns each key from $[U]$ into an array of k subkeys from $[b]$ using the algorithm `BC` from Problem 2 above as a subroutine, and then iteratively sorts on each of the k subkeys, Crucially, `RadixSort` uses the fact that `SingletonBucketSort` 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. (See the "Food for Thought" section in the SRE notes.) Here is pseudocode for `RadixSort`:

```
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 $A$
$k = \lceil \log_b U \rceil$;
foreach $i = 0, \ldots, n-1$ do
    $V'_i = \texttt{BC}(K_i, b, k)$ ;                     /* $V'_i$ is an array of length $k$ */
foreach $j = 0, \ldots, k-1$ do
    foreach $i = 0, \ldots, n-1$ do
        $K'_i = V'_i[j]$
    $((K'_0, (V_0, V'_0)), \ldots, (K'_{n-1}, (V_{n-1}, V'_{n-1}))) =$
      SingletonBucketSort($b, ((K'_0, (V_0, V'_0)), \ldots, (K'_{n-1}, (V_{n-1}, V'_{n-1})))$);
foreach $i = 0, \ldots, n-1$ 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 $((K_0, V_0), \ldots, (K_{n-1}, V_{n-1}))$
```

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 SortingOnFiniteUniverse problem defined in SRE 1).

      Hint: You will need to use the stability of `SingletonBucketSort` in your argument. If it were replaced with an instable implementation (or any other unstable sorting algorithm, such as `ExhaustiveSearchSort` with an unfortunate ordering on permutations), then the resulting algorithm would not be a correct sorting algorithm. For intuition, you may want to think about what happens when you sort a spreadsheet by one column at a time.

      Proof by induction:
      Base Case: *The least significant subkey.*
      Since we already can assume the correctness of `SingletonBucketSort` *based on the*

notion of reductions we know that it will sort the least significant subkey validly. Furthermore `SingletonBucketSort` *is stable which means in cases of ties, it preserves the previous ordering. This is important later, but is necessary to be recognized in the base case.*
Inductive Step:
Hypothesis: Assume that after sorting based on subkeys $(0, 1, \ldots, j)$, the array is correctly sorted around these subkeys.
Induction: Since `SingletonBucketSort` *is stable it preserves the valid ordering from the sorting of subkeys up until $(0, 1, \ldots, j)$ thus sorting the $j + 1$ subkey maintains that order and produces valid sorting.*
After sorting for $j = 0$ to $j = k - 1$ subkeys in order of least significants, the keys which are validly represented by their subkeys due to `BC` *(reduction algorithm), will be in a valid order because the dominant subkeys will be prioritized in later subkey orderings and tie-breaks in less significant values will be preserved by the stability of* `SingletonBucketSort` *in less significant subkey sortings.*
Conclusion:
By the stability of `SingletonBucketSort` *and the validity of that algorithm along with* `BC` *as reductions, we know the keys will be broken down into representative subkeys that are sorted, thus producing a sorted array A.*

(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 $O(n + n \cdot (\log U)/(\log n))$. (This runtime analysis is outlined in CLRS, but you'd need to adapt it to our notation and slightly more general setting.)

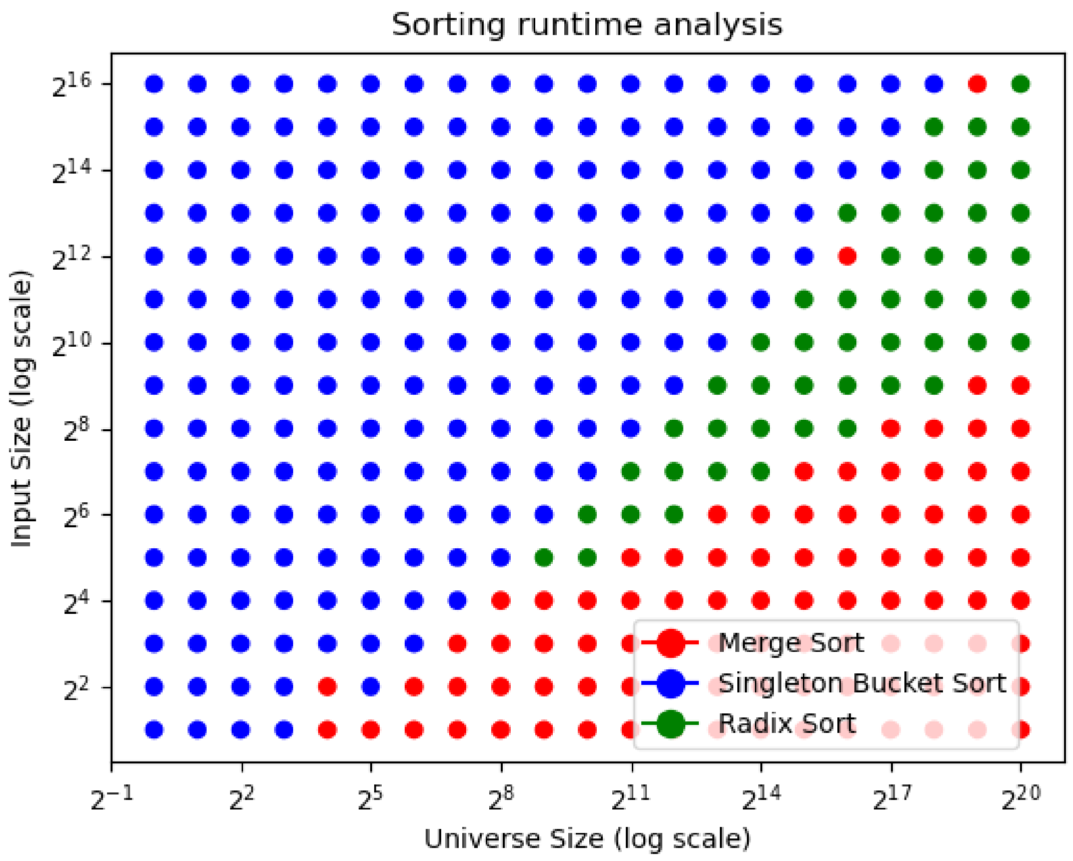
RadixSort has runtime $O((n + b) \cdot \lceil \log_b U \rceil)$ because it is broken into a few steps:
Splitting subkeys: This only happens once and happens across n elements and has $\lceil \log_b U \rceil = k$ subkeys at each element. Runtime $= (k \cdot n)$
Sorting: Sorting each subkey takes $O(n+b)$ as we know that's is the runtime of `SingletonBucketSort`*, and occurs for $\lceil \log_b U \rceil = k$ subkeys. Runtime $= O(k \cdot (n + b))$*
Thus overall the runtime is $O(k \cdot (n+b))$ where $k = \lceil \log_b U \rceil$ hence runtime is $O(\lceil \log_b U \rceil) \cdot (n + b))$.

When you set $b = \min\{n, U\}$ you have 2 paths.
$U \leq n$, which means $b = U$ so $\lceil \log_b U \rceil = \lceil \log_U U \rceil = 1$. Thus runtime is $O(n + U)$.
$n < U$, which means $b = n$ so $\lceil \log_b U \rceil = \lceil \log_n U \rceil$ and runtime of $O(\lceil \log_b U \rceil) \cdot (n+b)) = O(\lceil \log_n U \rceil \cdot (n + n) = O((n + n) \cdot (\frac{logU}{logb}$. Thus RadixSort achieves linear time when the universe size U is polynomially bounded by n (i.e., $U = n^{O(1)}$).

(c) (implementing algorithms) Implement `RadixSort` using the implementations of `SingletonBucketSort` and `BC` that we provide you in the GitHub repository.

(d) (experimentally evaluating algorithms) In `ps1_experiments.py`, we've provided code for running experiments to evaluate the runtime of sorting algorithms on random arrays

(with $b = \min\{n, U\}$ in the case of `RadixSort`) and for graphing the results. Run this code and attach the resulting graph (you should see that each sorting algorithm dominates in some region of the graph – if you want better results you can try increasing the number of trials in the experiments file).
Note: 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!



(e) Do the shapes of the transition curves found in Part 3d match what we'd expect from the asymptotic runtime formulas we have for the algorithms? Explain. For a most thorough answer, try setting the asymptotic runtimes of `SingletonBucketSort` and `RadixSort` to be equal to each other (ignoring the hidden constant in $O(\cdot)$) and see what $\log U$ vs. $\log n$ relationship follows, and similarly for comparing `RadixSort` and `MergeSort`.

Generally, I believe the shapes of the transition curves match what we'd expect. The granularity of the graph is not enough to see too much detail in the transition curves, but the positioning of the transitions make sense. To compare when we would expect the different types of sorts to be better lets compare the runtimes of `MergeSort` *with* `RadixSort` *then* `SingletonBucketSort` *with* `RadixSort`*.*

`RadixSort` *x* `MergeSort`*:*
$O_{Radix}(n + n \cdot \frac{logU}{logn}) = O_{Merge}(n \cdot logn) \rightarrow O_R(1 + \frac{logU}{logn}) = O_M(logn)$. *This shows us the relationship between the two sorts. If logn is growing faster than logU, the runtime of* `RadixSort` *will increase slower than* `MergeSort` *making it faster. We see this in the graph when if Input Size (n and also y-axis) is growing faster than Universe Size (U and also x-axis),* `RadixSort` *is faster.*

`RadixSort` *x* `SingletonBucketSort`
$O_{Radix}(n + n \cdot (\frac{logU}{logn})) = O_{Single}(n + U) \rightarrow O_R(n \cdot \frac{logU}{logn}) = O_S(U) \rightarrow O_R(\frac{logU}{logn}) = O_S(\frac{U}{n}$. *Which shows the relationship between the two sorting methods. Since we know increases in U will always affect* `SingletonBucketSort` *more (in terms of runtime cost) since $U < logU$ for all positive U, but increases in Input Size (n or x-axis) will improve runtime more for* `SingletonBucketSort`*, then it makes sense that* `RadixSort` *is more efficient at higher values of U (Universe Size or x-axis) than* `SingletonBucketSort` *(in situations where n is not too large). And on the other hand, when n is large and U is small,* `SingletonBucketSort` *will be more efficient.*

4. (Reflection Question) There are a number of resources to support your learning in CS1200, such as Lecture, Ed, Office Hours, Section, Detailed Lecture Notes (posted after class), Recommended Readings, Collaboration with Classmates, the Patel Fellow, the Academic Resource Center (ARC), Sender-Receiver Exercises. Which of these (or any others that come to mind) have you found most helpful so far and why? Are there ones that you should take more advantage of going forward? Do you have suggestions for how the course can make these more helpful to you?

   Note: As with the previous pset, you may include your answer in your PDF submission, but the answer should ultimately go into a separate Gradescope submission form.

   I found OH very helpful. I was stuck trying to understand how `RadixSort` *worked, but went to Professor Anshu's office hours and we just chatted about the algorithm. I went in with a few questions, and felt rewarded that as I answered these, and understood the algorithm, I was able to do the problem on my own. I enjoyed that because often OH in other classes do not teach you the material and can provide answers, but in this case I improved my understanding. I also have found some benefit in ED. Many of my classmates have had similar basic questions as me, and I am able to both feel validated in my confusion on that topic :) and get some answers quickly. On the other hand, I definitely should take more advantage of section. I have struggled to fit it into my schedule, but feel like it would help a lot with the fundamentals that are important for this class. I plan to go to one tomorrow in fact, so hopefully it is helpful. I am not sure about how to make these resources more helpful, but some feedback about lecture is I feel that we often go slowly over easier things (like left-childs of binary trees contain lesser values than the vertex), but go fast over things that warrant more time. This causes some confusion for me when reviewing my notes which are much more sparse on these complex topics.*

5. Once you're done with this problem set, please fill out this survey so that we can gather students' thoughts on the problem set, and the class in general. It's not required, but we really appreciate all responses!