CS1200: Intro. to Algorithms and their Limitations Prof. Anurag Anshu

Problem Set 1

Harvard SEAS - Spring 2026 *Due: Sat 2026-02-07 (11:59pm)*

Please see the syllabus for the full collaboration and generative AI policy, as well as information on grading, late days, and revisions.

All sources of ideas, including (but not restricted to) any collaborators, AI tools, people outside of the course, websites, ARC tutors, and textbooks other than Hesterberg–Vadhan must be listed on your submitted homework along with a brief description of how they influenced your work. You need not cite core course resources, which are lectures, the Hesterberg–Vadhan textbook, sections, SREs, earlier problem sets and earlier solutions sets in this semester. If you use any concepts, terminology, or problem-solving approaches not covered in the course material by that point in the semester, you must describe the source of that idea. If you credit an AI tool for a particular idea, then you should also provide a primary source that corroborates it. Github Copilot and similar tools should be turned off when working on programming assignments.

If you did not have any collaborators or external resources, please write 'none.' Please remember to select pages when you submit on Gradescope. A problem set on the border between two letter grades cannot be rounded up if pages are not selected.

Your name: Joaquin de Castro

Collaborators and External Resources: Please see link to thread for last part of Problem 2

No. of late days used on previous psets: 0

No. of late days used after including this pset: 0

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
$3(\log_2 n)^{3/2}$	$2(\ln n) \cdot (\ln n + 3)$	F	F	F
$3n^3$	$\lvert\{S \subseteq [n] : \lvert S\rvert \leq 4\}\rvert$	F	F	F
$(n+1)^{n+1}$	$(n+1) \times n!$	T	F	T
4^n	$(4 + (-1)^n)^n$	F	F	F

   (b) (runtimes: $T^=$ vs. T) Let $g : \mathbb{R}^{\geq 0} \to \mathbb{R}^{\geq 0}$ be a nondecreasing function, i.e. if $x_0 \geq x_1$, then $g(x_0) \geq g(x_1)$. (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. Consider the two statements:

$$\text{“}T^= = \Omega(g) \implies T = \Omega(g)''$$

      and

$$\text{“}T = \Omega(g) \implies T^= = \Omega(g)''.$$

      One of these statements is true and one is false. Prove the correct statement. For the false statement, 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.)
      Answer:
      From the definition of runtime, we know that $T(n) \geq T^=(n)$, because $T(n)$ is the maximum value of $T^=(n')$ for $n' \leq n$. Thus, we can prove the first statement using the definition of big Omega notation:
      $T^= = \Omega(g) \implies \exists c$ s.t. $\forall$ sufficiently large $n, T^=(n) \geq c * g(n)$
      Then $T(n) \geq T^=(n) \geq c * g(n)$
      Hence $T = \Omega(g)$
      As a counter example, consider if
      $T^=(n) = (4 + (-1)^n)^n$
      then
      $T(n) > (4 + (-1)^{n-1})^{n-1}$
      since the maximum of n is the greatest even $n' \leq n$, which is either itself or $n - 1$.
      Let's choose the worst-case scenario where $n - 1$ is even, so we can say that
      $T(n) > (5)^{n-1}$
      clearly this grows at least as fast as $g(n) = 4^n$
      In fact T/g approaches ∞ as n increases, thus
      $T(n) = \Omega(g)$

However, note that for for any n, we can choose $T^{=}(n)$ such that $T^{=}(n) < 4^n$.
In particular, if any sufficiently large n_0 is proposed so that
$T^{=}(n_0) = 5^{n_0} \geq g(n_0)$, we can choose $n = n_0 + 1$ so that
$T^{=}(n_0 + 1) = 3^{n_0+1} \leq g(n_0 + 1)$
This shows that the second statement must be false.

ii. (Optional question[1]) Prove that $T^{=} = O(g)$ iff $T = O(g)$. Thus the distinction between T and $T^{=}$ does not matter when we are interested in "big-oh" runtime bounds, which are nondecreasing in most "natural" cases.
Answer:
Right to left:
If $T = O(g)$
then $\exists c$ s.t. $T(n) \leq g(n)$ for large enough n
note that $T^{=}(n) \leq T(n)$
then we can use the stated definition of Big O notation and transitivity to shows
$T^{=}(n) \leq T(n) \leq c * g(n)$
$T^{=}(n) \leq c * g(n)$
for the same sufficiently large n, thus $T^{=}(n) = O(g)$ QED
Left to right:
If $T^{=}(n) = O(g)$
then $\exists c$ s.t. $T^{=}(n) \leq c * g(n)$ for large enough n
Now since we just have to prove Big O for sufficiently large n, let us choose
$g(n) \geq g(n_0) \geq 1$
This is because we want to prove
$\exists C$ s.t. $T(n) \leq C * g(n)$ for large enough n
Dividing both sides by $g(n)$ we get
$T(n)/g(n) \leq C$
So the problem reduces to finding C such that this inequality is true.
However note now that the LHS is bounded from above by $T(n)$
Now we are almost finished, but because C must be a constant, and this inequality must be true for all sufficiently large n,
we should choose the maximum value of $T^{=}(n)$ where n is from 0 to n_0, and the inequality is true for $n > n_0$
Thus $T(n) = O(g)$

[1]This question will be graded only if you have seriously attempted all the other questions. It can only be used to increase your grade from an R to an R^{+}.

2. (Understanding computational problems and mathematical notation)

Recall the definition of a *computational problem* from Lecture Notes 2.

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:

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

(a) If the input is $(n, b, k) = (57, 12, 3)$, what does the algorithm `BC` return? Is `BC`'s output a valid answer for Π with input $(57, 12, 3)$?

For i = 0:

$c_0 = 57 \mod 12 = 9$ $n = (57 - 9)/12$ $n = 4$

For i=1

$c_1 = 4 \mod 12 = 4$ $n = (4 - 4)/12$ $n = 0$

For i=2

$c_2 = 0 \mod 12 = 0$ $n = (0 - 0)/12$ $n = 0$

BC thus outputs $(9, 4, 0)$

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

The possible inputs are all sets of 3 integers (n,b,k) where each is greater than 1 The set of possible outputs are all subsets of the set of natural numbers (n,b,k) is mapped to a subset of the natural number $(c_0, ..., c_{k-1})$ if they satisfy an equation for n in terms of b.

BC essentially solves for the coefficients of powers of b in increasing order.

At any iteration i, the algorithm saves the best approximation it can give of n. The algorithm starts by isolating c0 through getting the remainder of n divided by b. All terms that has a factor of b do not contribute to this, leaving only c0. We save this and subtract it from the old n to get the number that we now have to equate to a linear combination of powers of b. Then dividing this new number by b makes the new constant term c1. Repeating this process allows us to solve for coefficients of increasing powers of b by making it the constant by subtracting smaller terms and dividing by b. This algorithm assumes there is a solution and if not returns $\perp$

We can also think of the computational problem as decomposing a number n into base-b. For example $b = 10$ then c_i as the remainder at that point just gets the i(+1)th digit in base 10 representation.

Note that the output is $\perp$ if k does not give us enough "places"/large enough powers of b to express the number. In decimal representation, we cannot express 1,000 for example as just 2 place values in base-10, as the largest two digit number is 99.

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

Given that the output of the problem is not $\perp$, the output of f(n,b,k) must be a set/tuple of size k. The empty set however has 0 elements in it, so this could only possibly be an output if k=0. However, we know that k must be at least 2 given the problem constraints. This contradiction implies that the empty set cannot be an output of f(x).

(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.

Some inputs might have output $\perp$, so in this case there is not a well defined way to get the number of elements of this null value.

Excluding this case, for each possible input $x = (n, b, k)$, the size of the output f(x) is k. This is because the output must correspond to the coefficients of b^0 to b^{k-1}, which are k terms. Even if the coefficient of one term in the equation is 0, this still corresponds to a value of 0 for that corresponding element in the 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.

ChatGPT thread where I asked two questions to clarify, both of which were just a clarification of notation: https://chatgpt.com/share/e/6987c755-7a0c-8000-ba33-5a0171378b0d

Prompt 1:

I have attached a problem I am seeking clarification on, but I have also pasted the course's generative ai policy below: ... My understanding of the question is that the output is the same except it takes the union of the original output with the set of numbers 0 to k-1. So my thinking is that the algorithm would only solve it (in the case k=3 for ex) if $n = 0 + 1b + 2b^2$ or some reordering of 0,1,2. I just want to know if im understanding the question right. If any part of my prompt is not in line with the generative ai policy please do not answer it

Prompt 2:

Is it right that given an input, $\perp$ is considering solving a computational problem by an algorithm if and only if there is no acceptable output for the computational problem. So if an algorithm outputs $\perp$ but for example 72 is an acceptable output for all inputs, that algorithm would not solve the problem? If any part of my prompt is not in line with the generative ai policy please do not answer it

(I think both questions would have also been completely acceptable questions to ask in office hours and would have received just the same extent of a response)

Any algorithm that solves Π does not necessarily solve Π'

Consider BC, which solves Π.

However, consider input (100,10,2)

For i=0 $c_0 = 100 \mod 10 = 0$ $n = (100)/10 = 10$

For i = 1 $c_1 = 10 mod 10 = 0$ $n = 10/10 = 1$

Note that n is not 1 so the algorithm outputs $\perp$
However $(0, 1)$ is a possible output for this input in Π'
Thus this does not 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`:

```
1  RadixSort(U, b, A)
   Input  : A universe size U ∈ ℕ, a base b ∈ ℕ with b ≥ 2, and an array
            A = ((K_0, V_0), ..., (K_{n-1}, V_{n-1})), where each K_i ∈ [U]
   Output : A valid sorting of A
2  k = ⌈log_b U⌉;
3  foreach i = 0, ..., n − 1 do
4  |  V'_i = BC(K_i, b, k) ;                          /* V'_i is an array of length k */
5  foreach j = 0, ..., k − 1 do
6  |  foreach i = 0, ..., n − 1 do
7  |  |  K'_i = V'_i[j]
8  |  ((K'_0, (V_0, V'_0)), ..., (K'_{n-1}, (V_{n-1}, V'_{n-1}))) =
   |   SingletonBucketSort(b, ((K'_0, (V_0, V'_0)), ..., (K'_{n-1}, (V_{n-1}, V'_{n-1}))));
9  foreach i = 0, ..., n − 1 do
10 |  K_i = V'_i[0] + V'_i[1] · b + V'_i[2] · b^2 + ··· + V'_i[k − 1] · b^{k−1}
11 return ((K_0, V_0), ..., (K_{n-1}, V_{n-1}))
```

Algorithm 0.0.1: 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.

Lemma:

A number x is bigger than y if given some base b and some universe size $U \geq \max(x, y)$

if for $C^x = BC(x, b, k)$ $C^y = BC(y, b, k)$, $C^x[i_{max}] > C^y[i_{max}]$ where i_{max} is the largest index i for which $C[i] \neq C[i]$.

We'll call i_{max} here our most significant digit.

Where $k = \lceil \log_b U \rceil$

I'm honestly not sure if this lemma is necessary to prove, but to convince ourselves we can consider the worst case scenario, where we choose the smallest possible x and the largest possible y satisfying this condition.

Then $x = C^x[i] * b^i$, $y = C^y[i] * b^i + \sum_{m=0}^{i-1}(b-1) * b^m$

In the worst possible case $C^x[i] - C^y[i] = 1$

So $x - y = b^i - (b-1)\sum_{m=0}^{i-1} b^m$

Using the formula for the sum of a geometric sequence

$x - y = b^i - \frac{b^i - 1}{b-1}$

$x - y = \frac{b^{i+1} - 2b^i + 1}{b-1}$

This is a strictly increasing function for $b > 1$, and in the worst-case that $b = 2$ this is still 1 which is positive.

Thus the lemma is true.

Let us induct on $j = 0, 1, 2, ..., k-1$

Note that there is a direct correspondence between K_i and V_i, so for notational simplicity we can treat this as a sorting of a set of arrays V_i

Our loop invariant at j will be that the arrays $\{V_i'\}$ correspond to a valid sort of $\{(K_i'(j), V_i')\}$

where $K_i'(j) = K_i = V_i'[0] + V_i'[1] \cdot b + V_i'[2] \cdot b^2 + \cdots + V_i'[j] \cdot b^j$

Now consider the base case $j = 0$

RadixSort calls *SingletonBucketSort* on the constant terms and thus produces a valid sort of the array for the lowest place value. This satifies our loop invariant since we only care about the lowest place value so far.

Now the inductive hypothesis, assume $\{(K_i', V_i')\}$ is a valid sort up to $j = q$, or that $\{(K_i'(q), V_i')\}$ is a valid sort of the array.

Then calling Singleton Bucket Sort on $j = q + 1$ creates a valid sort of $\{(V_i[q+1], V_i')\}$.

Now note that the stability of Singleton Bucket Sort guarantees that for ties on $V_i[q+1]$ the ordering of the original array a step before is preserved.

But we just said that at $j = q$ the array was a valid sort on key $K_i(q)'$. Thus by our lemma, $\{(K_i'(q+1), V_i')\}$ is now a valid sort.

This is becaues when comparing any two numbers here, if the most significant digit is $q + 1$, Singleton Bucket Sort guarantees this is sorted correctly.

Otherwise then we can use that the prior step was a valid ordering up to $j = q$ or our loop invariant to be confident that ties are sorted correctly.

Hence by the principle of mathematical induction, the output array after reaching $j = k - 1$ is a valid 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 $O(n + n \cdot (\log U)/(\log n))$. (This runtime

analysis is outlined in CLRS, but you'd need to adapt it to our notation and slightly more general setting.)

First we check the runtime of calling BC n times.

Looking at the BC algorithm, we see that the for loop runs until n = 0.

When does this happen? Note that n decreases by a factor of b each run, and when n becomes 1, in the next run $1 mod b$ will also give 1, and thus n becomes 0.

Hence the algorithm will run in $O(\log_b n)$ time.

Since we call it n times, the run time of this first loop is $O(n \log_b n)$

Now the second loop calls singleton bucket sort on a universe size b with n elements, so this takes $O(b + n)$

There are $k = \lceil \log_b(U) \rceil$ passes of this, so this next part will run in $k = (b+n)\lceil \log_b(U)$

Now here the stronger linear growth in the second term dominates, so the second term dominates the run time.

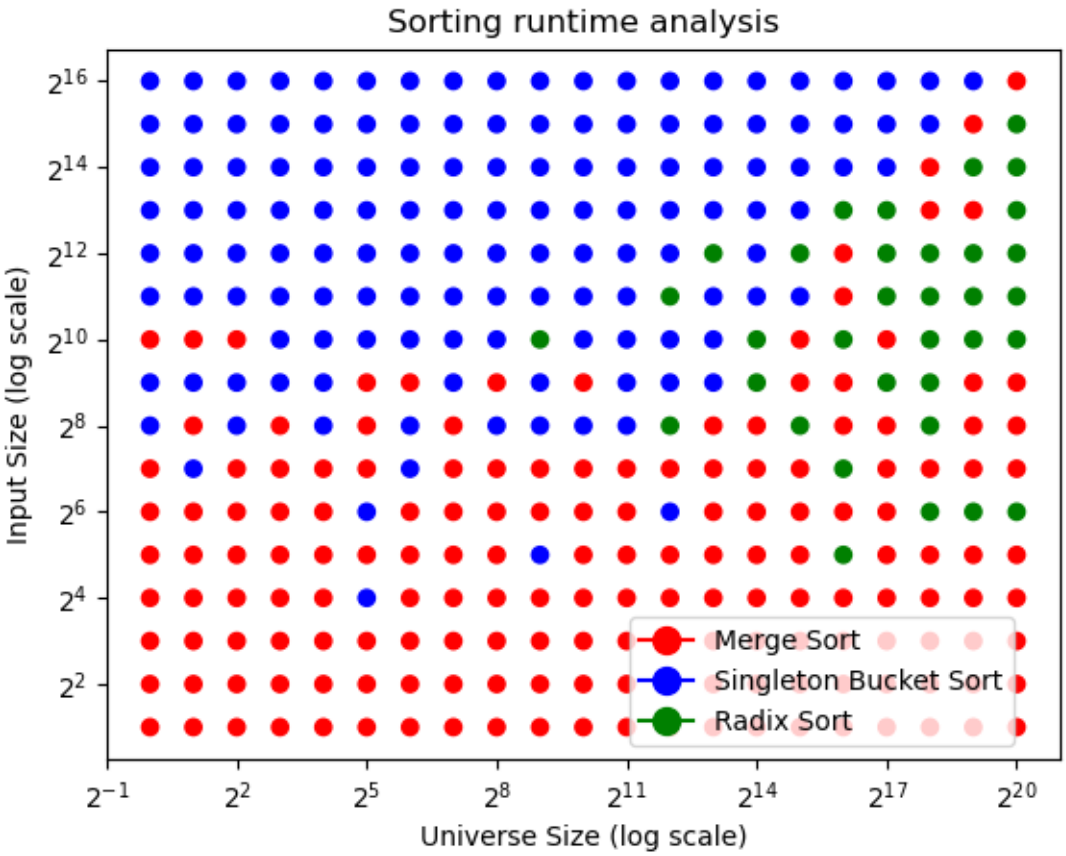
Then setting $b = min(n, U)$, we obtain

$O(n + n \cdot (\log U)/(\log n))$

(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`.

Setting the asymptotic runtimes equal to each other we have that

SBS and RadixSort:

$n + U = n + n * \log U / \log n$

$U = n * \log U / \log n$

$\log n = (n/U) * logU$

Merge Sort and RadixSort:

$n \log n = n + n * \log U / \log n$

$n \log n = n + n * \log U / \log n \ \log^2 n = \log n + \log U$

I plotted these curves on Desmos setting x and y to be log(U) and log(n) respectively, and found that the top part of the right-opening parabola indeed corresponds to the boundary between SBS and RadixSort and the lower part of the first quadrant of the blue graph indeed loosely approximates the boundary between merge sort and radix sort

https://www.desmos.com/calculator/sd7xnhqpem

Furthermore, it makes sense that at higher universe size Radix Sort performs better, as its purpose was precisely to decrease the dependence on U from linear to logarithmic, which is why we only see Radix Sort performing better at higher universe size values.

4. (Reflection Question) There are a number of resources to support your learning in CS1200, such as Lecture, Ed, Office Hours, Sections, Hesterberg-Vadhan book, Recommended Readings, Collaboration with Classmates, 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?

   Quick note on grading : Good responses are usually about a paragraph, with something like 7 or 8 sentences. Most importantly, please make sure your answer is specific to this class and your experiences in it. If your answer could have been edited lightly to apply to another class at Harvard, points will be taken off.

   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.

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!