Algorithms - Assignment 2

Due Thursday, February 22 by 11:59 pm

This assignment will have a programming component in addition to a theory/math component. You will be asked to submit graphs that you create using matplotlib in python. The graphs and short answers should be part of the write-up you submit along with your written homework. The code will be submitted separately on gradescope.

The purpose of the programming section of this assignment (which is the bulk of it) is to have you begin to create a laboratory for implementing and testing the algorithms which we cover theoretically in class. In particular, this assignment will have you implementing and comparing several sorting algorithms. Along the way, you might learn some things about python that you didn't know already. We will be building onto this toolkit in future assignments, so I recommend you create a dedicated folder and/or github repository for our class.

To any student who has never used python before: I don't remember anyone raising their hands when I asked in class if who wasn't familiar with python, and this assignment is made with the assumption that you are. If you weren't there or did not hear the question when I asked, or if you did hear and I just missed you raising your hand, then I would very *strongly* recommend that you come by during my office hours or make an appointment with me as soon as possible. Between my TA's and I, we can easily get you acquainted with the language and transitioned over to using it from whatever you're more familiar with.

I will be referring to python lists and arrays interchangeably throughout this assignment, and you are expected to be working with lists in your own implementations. Typically, I'll use A to denote the list we're sorting, and n to denote the size of a list. **To make it easier for you to quickly find what's being asked of you, I've bolded those instructions on problem parts where there is a lot of extra information (but only those parts!)**

1 Programming Section

A few disclaimers: I will be using python lists and arrays interchangeably throughout this assignment, and you are expected to be working with lists in your own implementations. Typically, I'll use A as the variable to denote the list we're sorting, and n to denote the size of the list.

(1) We talked about a simple quadratic time sorting algorithm in class called bubblesort. Two other simple sorting algorithms which are often compared with bubblesort are insertionsort and selectionsort. You may have covered these in your other classes, but I'll remind you what the idea is

- Selection sort works by moving through A n many times. The first time, it moves through searching for the biggest element. When it finishes scanning the list, it swaps the final element of the list with that biggest element. On the next pass, it does this

looking for the second biggest element, and swaps it with the second to last entry. And so on, until everything is in it's proper place.

- Insertion sort works similarly to bubblesort in that it moves accross the list, looking at pairs of elements as it goes and swapping when it finds a pair i and $i+1$ such that $A[i] > A[i+1]$. The difference is that insertionsort doesn't just perform this single swap. Instead, it continues propagating the element backwards until it's at a place where the element to the left of it is smaller. Doing it this way, the algorithm only needs to do one full scan of the list.

Stepping through vizualizations helps massively with quickly coming to grips with any algorithm. Click here to if you would like to try doing so for any of these three algorithms.

(a) Create three python functions, bubblesort(A), insertionsort(A) and selectionsort(A), which implement these three algorithms. For bubblesort, you may simply copy-paste the code I presented in class. (You can add bogosort too, if you want.)

(b) Write two functions which will assist in testing your code and verifying that your algorithms are correct. First, make a function called is_sorted(A), which checks if A is sorted, returning a boolean indicating the answer. Second, make a function called check_validity(func), which takes a sorting algorithm as input (yes, functions can be inputs to other functions in python!) and tests it on a few hundred randomly generated lists of varying sizes, using the is_sorted function to verify success each time. For this, you will want to import and utilize the random number generation feature of numpy that we discussed in class. You *must* use the numpy rng generator to receive credit for this. Have the function test at least one list of all sizes which are multiples of 10 between 100 and 400, and return a boolean indicating whether your algorithm works or not. Obviously, you should use this algorithm yourself to confirm your algorithms work before continuing to the next problem.

(c) Now that you've got all three algorithms in front of you, what are their theoretical worst-case runtimes in big O notation?

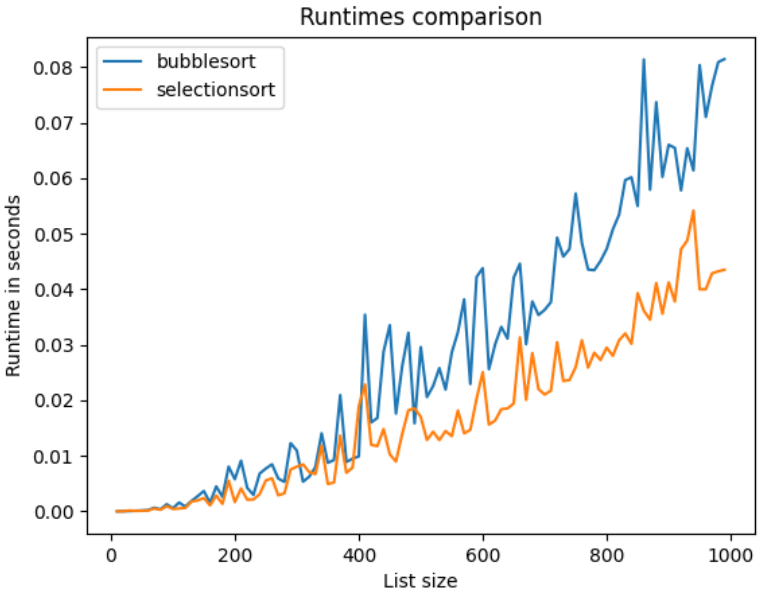
(d) A sorting algorithm is called **in-place** if it uses space $O(1)$. What is the worst-case space complexity of these algorithms. Are any of them in-place?

(2) (a) **Create a function called compare_times(algo_1, algo_2) which does the following.** First, create random lists of integers between 0 and 1000 (or floats, doesn't matter). Lists created should be sizes between 10 and 1000 which are multiples of 10. For each list, use the perf_counter() function of the time module in order to obtain the runtime of each algorithm used on the generated list. These runtimes should be saved in lists for each algorithm (e.g. times_1 and times_2 should be lists containing the runtimes of each algorithm that are populated during the loop). **Second, your function needs to plot these lists using matplotlib.** I'll give you the code for this first plot, for you to build off of:

```
fig, ax = plt.subplots()
ax.plot(x_axis, times_1, label = f"{algo_1.__name__}")
ax.plot(x_axis, times_2, label = f"{algo_2.__name__}")
ax.set_title("Runtimes comparison")
ax.set_xlabel("List size")
ax.set_ylabel("Runtime in seconds")
```

```
ax.legend()
plt.show()
```

If the rest of your code is correct, this should produce a graph which looks like this:



Code explanation: Here, the first command constructs instances of the two objects which are central to matplotlib: a Figure and and Axis. The Figure object is the window on your computer screen that everything goes inside of, while an Axis is any box inside of the figure that can contain all sorts of things - animation, pictures, plots, and so on. One single figure can have multiple axes inside of it - we'll do some of that later. The next two lines plot the times. The first argument of both instructions, x_axis (which you'll want to create), is a list containing every array size that the algorithms were tested on. The second arguments is obviously the arrays created during testing, and represent the y-axis. The third argument, label, is the text that displays next to the line colors on the legend, which is created using the legend() method in the second to last line. If you've never done any object oriented programming, then __name__ will look odd to you. This is a *dunder* property - a hidden special attribute which all python objects posess. In this case, __name__ is exactly what it sounds like. set_title, set_xlabel and set_ylabel should be self-explanatory. Finally, plt.show() is a method which displays the figure that you've created and populated on the screen.

You can save a png of your figure once creating it by clicking the floppy-disk icon at the top. As a solution to this problem (and the others of this assignment), **you must both submit code of your functions to gradescope as well as a pdf write-up file which contains images of your plots along with the rest of your written or typed solutions.** You can use the .tex file of this assignment to see how to embed images in latex. Additionally, you can check out this tutorial for more formatting details: https://www.overleaf.com/learn/latex/Inserting_Images.

A couple of warnings: Firstly, remember that the 'lists' created by numpy aren't really lists, but rather numpy arrays. For the most part, the difference between these objects is negligable; access syntax is the same as lists (using square brackets) and matplotlib is programmed to accept both lists and numpy arrays interchangeably. However, they can be casted as lists very simply - just use the command A = list(A). You may wish to do so here just to be safe and make sure nothing is happening that you aren't aware of. Second, it is *crucial* to understand that lists and (numpy arrays) are *mutable*, and are thus passed by reference into functions (in python, all mutable objects are passed by reference, while immutable objects are passed by value). If you pass a list to a sorting algorithm, and that sorting algorithm makes changes to it's input, then those changes will effect the list whether or not a list is returned. Thus, if you pass a list into a sorting algorithm, and then pass the same list into a second sorting algorithm, *then that algorithm will be given an already sorted list.* You can avoid this by instead passing a casted copy of the list:

```
# instead of this:
bubblesort(A)
# do this instead:
bubblesort(list(A))
```

In the second case, you are passing a copy of A instead of A itself, and thus the second time you pass A this way, it won't be pre-sorted. If your results in matplotlib aren't what you expect, this is probably why!

(b) Now we will modify our function and make it more robust. If you complete this part, please submit it in place of part (a) for full credit on both parts. (In other words, part (a) was only there to act as a stepping stone.) First, python functions can be made to take an arbitrary number of arguments using *args and **kwargs. Let's change our

function from part (a) to look like this:

```
# By putting a * in front of the argument:
def compare_times(*algos):
# You can call the function with any number of inputs:
compare_times(bubblesort, insertionsort, selectionsort)
```

What is passed here when this function is called is a list called algos (*not* *algos, just algos) containing all of the arguments given. The next thing we can do is add an *optional argument* to our function:

```
def compare_times(*algos, max_size = 400)
```

By setting an argument equal to something in a function definition, you make that input optional. If the user doesn't input anything as the max_size, it gets set to 400. However, because there could be any number of arguments prior to this one, when you call it, you have to input the max_size as a *keyword argument*:

```
# you have to tell python when the function args end!
# and where the additional arguments begin!
compare_times(bubblesort, insertionsort, selectionsort, max_size = 700)
```

Unlike the more standard positional arguments, keyword arguments can be entered in any order. However, they must follow *after* any positional arguments.

Modify your function from part (a) to create a plot containing all three sorting algorithms instead of just a pair of them. Make sure to write code that would work for any number of function arguments, by looping over the algos list. **Then modify your function again** to allow for a max_size optional keyword argument, which controls the number of different sized arrays your test goes generates. **Finally, use this function to plot all three sorting algorithms against each other**, with a max list size of 500. Save a picture of this plot and include it in your write-up. **Then answer the question: which algorithm seems the fastest?**

Now that we've got the basics down, let's make our analysis more rigorous by making three additional functions

(c) Make a function called compare_worst(*algos, max_size = 400, trials = 100) which, instead of just generating and testing one array of each size, generates [trials] many different lists of each size and tests each algorithm on each of them. After testing on a hundred or so different lists of a particular size, save only the *worst* of those times to the algorithm's list of times. Then plots those times against each other, and add a picture of it to your write-up. Your plot should be properly labelled with a legend, a title, and text describing the axes. (I recommend starting out by making a function which only acts on two different algorithms to start, and then modifying it once you have that working. It will be less of a headache that way.) **Which algorithm performs the best here? Which performs worst?**

(d) Make a function called compare_average(*algos, max_size = 400, trials = 100) which, instead of just generating and testing one array of each size, generates [trials] many different lists of that size and tests each algorithm on each of them, and then saves the *average* performance times for each each list size in separate lists (one for each algorithm), then plots those times against each other. Again, save a picture and add to your write-up. How does this compare with the worst-case test? Is anything different?

(e) Finally, make a function called compare_best(*algos, max_size = 400, trials = 100) which does exactly what part (c) did, except choosing the best case instead of the worst. Again,

save a picture to your write-up and answer the following. Is anything different here from the other two cases? Copying and pasting code is your friend for these three problems.

(3) The compare_best test you made in the previous problem isn't actually very good at evaluating best-case performance. The reason for this is that best-case performance doesn't just happen randomly. Rather, it happens when the lists being sorted have a particular structure which plays nicely with the algorithm.

(a) In the case of bubblesort and insertionsort, the best-case performance is clear: it happens when the list is already fully sorted! To demonstrate this, modify your best-case performance function to *pre-sort* the list before testing your algorithms on it. This can be done using the built-in sorting method in python:

```
# this will result in A being sorted
sorted(A)
```

Just as an FYI, python's built-in sorting algorithm is called timsort. We'll look a little more closely at it when we get to the more advanced sorting algorithms. **Anyway, modify your best-case algorithm to pre-sort your lists before testing the algorithms on them. Call it, save a png and add the picture to your write-up.** You should see bubblesort and insertionsort massively outperforming selectionsort.

(b) Answer the following questions: Are these better runtimes for bubblesort and insertionsort still quadratic? If not, what do they look like? (Consider plotting just these two against each other to better make out the curve.) Can you justify what you're seeing by looking at the algorithms? Try to do so. Is there any way to modify and optimize the performance of selectionsort to achieve comparable best-case time complexity to the other two? If not, why? And furthermore, is there any special list which would be faster for selectionsort? Finally, what can you conclude is the best case big-Ω performance of these three algorithms?

(c) A sorting algorithm is called **adaptive** if it's faster on lists that are already mostly sorted. Clearly, both bubblesort and insertionsort are adaptive. Even more clearly, insertionsort is not. However, of the two which are adaptive, is one more adaptive than the other? I'm feeling nice, so I'll just give you the function you'll need to test this:

```
def make_sorta_sorted(n, mess_factor):
    rng = r.default_rng()
    A = sorted(rng.integers(1,1000,n))
    percent = int((mess_factor * n) // 2)
    for i in range(percent):
            j_1, j_2 = rng.integers(0,n,2)
            A[j_1], A[j_2] = A[j_2], A[j_1]
    return A
```

This function generates a random list of n many numbers, sorts them, then messes it up to a degree based on the mess_factor, which is expected to be between 0 and 1 (inclusive). The idea is that a mess factor of 0.2 will mess up 20% of the list, resulting in a list which is 80% sorted. **Modify your compare_best function once more to generate lists which are 80% sorted using my function. Save a plot, add it to the write-up, and then answer the question: what do you see? Is one algorithm indeed more adaptive than another?**

(d) From all of the tests we've done and analysis we've conducted, can we conclude that one of these three algorithms is better than the others? Which is it? Make sure to summarize all reasons why in your answer.

There are other properties of sorting algorithms which are worth considering besides the ones discussed here, such as being online, stable, or parallelizable. We may consider these properties on future homework assignments. That will do it for what you need to submit for the coding portion of this assignment. However, I have one more thing I'd ask you to do and understand, regarding organization.

(4) The last thing we will do is write a function which allows us to compare a sorting algorithm with it's theoretical big-O runtime. Our function, called compare_with_theory(algo, theo, max_size=500 resolution=1), will allows us to view the runtime (or average runtime if resolution > 1) of an algorithm side by side with the graph of a regular math function which we think is supposed to be modelling the runtime in the big-O sense. Getting the average runtimes is going to be identical to before. (Resolution is the same variable as trials in the compare_average function, we're just calling it something different because it's going to serve a different purpose.) This part you should be able to more or less copy and paste.

As for the theo argument, we want to be able to input the very simple function $f(x) = x^2$. Now, we could do this identically to how we did it before: define a function called squared(n) which squares n and returns it, and feed that into our compare_with_theory function as input, just like whatever sorting function we're choosing. However, that's a little silly, isn't it? Shouldn't we be able to feed this in as just a math expression? This can be done with the help of *lambda functions*. The best way to understand these is probably to just see it in use:

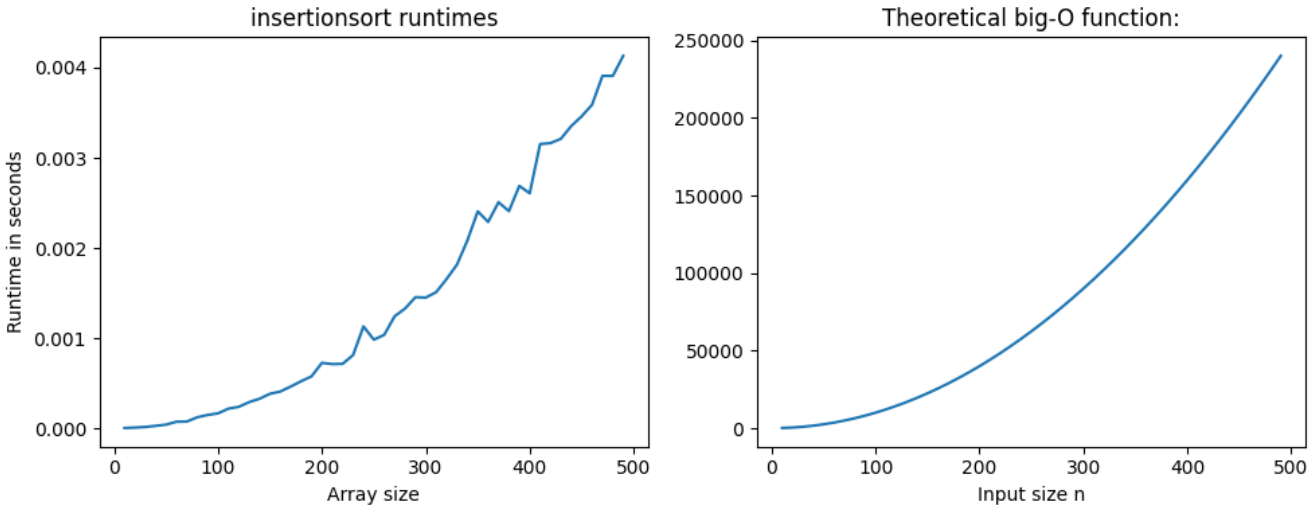
```
compare_with_theory(insertionsort, lambda n: n*n)
```

What's happening here is this: as the second theo argument, we're creating an anonymous one-line function on the spot. One which takes a number n as an input, and returns that number multiplied by itself. That function gets created and then immediately passed as an argument into the compare_with_theory. And that's all you need to do. Within the function definition, you can treat theo as exactly the same kind of thing as other function arguments. In the function definition, you'll want to create a list of theo_vals by plugging all of the n's in that you used as list sizes and appending the outputs.

Once you have the two lists created, you want to plot them, but we don't want them to be on the same plot. Instead, we want two different graphs side by side, i.e. two Axes. Here is the code for you to copy and paste:

```
fix, ax = plt.subplots(1,2,figsize=(10,4)) # this creates a 1x2 grid of Axes
ax[0].set_title(f"{str(algo.__name__)} runtimes")
ax[0].set_xlabel("Array size")
ax[0].set_ylabel("Runtime in seconds")
ax[0].plot(x_axis, algo_times)
ax[1].set_title("Theoretical big-O function:")
ax[1].set_xlabel(f"Input size n")
ax[1].plot(x_axis, theo_vals)
plt.tight_layout()
plt.show()
```

The first line creates two axes which will be displayed next to each other. The variable ax is now a list containing two Axis objects. ax[0] is the plot which will appear on the left, and ax[1] will appear on the right. Once you've coded your function, it should produce something which looks similar to this when called as above: As you can see, the scales are massively



different, but the shape is uncanny. By increasing the resolution from 1, you should get something smoother looking, which I think should converge to the theoretical function as resolution goes to infinity by the central limit theorem. Run your function with a resolution of 10 and save a picture to your write-up. Try something other than n^2 for the lambda function input. Unless it's $O(n^2)$ it won't look quite right. We'll have more to say about this comparison later in a few weeks.

There are other properties of sorting algorithms which are worth considering besides the ones discussed here, such as being online, stable, or parallelizable. We may consider these properties on future homework assignments. However, that's going to do it for the coding portion of this assignment. However, I do have one more thing I'd ask you to do and understand, regarding organization and how I'd like the code organized and submitted.

Code Submission: As you can see, we have a lot of functions, and leaving them all in a single .py file is going to quickly become cumbersome. In particular, we have the actual algorithms we are analyzing, and the test functions we are writing to perform on those algorithms. At the very least, these should be kept separate. There is a fairly crpytic if-statement which will help in keeping things organized. From now on in this class, your .py files should be structured as follows:

```
# first, import whatever you want
import stuff
# then, define your functions and/or classes
def function_1(args)
...
def function_n(args)
# then, have this:
if __name__ == "__main__":
    # put the main body of your code inside of this if statement!
```

We saw the __name__ dunder attribute already in this assignment. When you run a .py file, the running program gets a name attribute, which is "__main__". That is *as opposed to* whatever .py files you're importing, which get the names of the file as their __name__. In other words, what is inside of this if statement will only execute if the .py file is being run as the 'main' file.

Let's say you've saved your sorting algorithms in a .py file named "sorting_algos.py", and that inside of that file you've written some "scratch-work" code inside of the weird __name__ == "__main__": block to test random things out while you were making them. Then let's say that separately you've saved the tests we created for this assignment as a new file called "testing_utilities.py". To call your sorting algorithms for testing in this second file, you can import the file, bringing it along for the ride at runtime:

```
import sorting_algos as sorts
```

(Make sure that the file you want to import is in the same directory as the file you're writing code in.) You can then use any of those functions like you're used to by calling it by it's nickname (giving it a nickname is optional):

```
sorts.bubblesort(A)
```

Since all of the code which is supposed to be executed in the sorting_algos.py file is contained inside of the if statement, it is safely ignored upon being imported. *This way, you can write multi-functional code which does something upon being ran on it's own, but which can also be imorted by other .py files in order to re-use of the functions and classes defined within.* If you don't like typing sorts every time to call your functions, you can also do the following

```
# you can grab specific things,
from sorting_algos import bubblesort, selectionsort, insertionsort
# or just grab everything
from sorting_algos import *
```

This will allow you to call your functions as if you defined them in the python file you're working in. To get off to a good start organizationally, please submit your code for this assignment as two files, one containing the sorting algorithms, and the other containing all of the testing and utility functions we used to test them. What you have inside of the if __name__ == "__main__" blocks of either file is unimportant. This will allow us to stay organized as we continue to build machinery and implement algorithms throughout the class.

2 Written Section

(4) This question is intented to give you some practice with amortized analysis using the aggregate method. (The other two kinds are the accounting method and the potential method, but this one works just fine and seems the most straightforward.)

You're hopefully already familiar with the stack data structure. A stack is a list in which items are pushed (added to the 'top' of the stack) or popped (removed from top of the stack and returned). Stacks are 'last in, first out', i.e. the most recent thing added to a stack is the first thing which will be popped off of it, hence the name.

The two main operations for interacting with a stack S are therefore S.push(k), and S.pop(), which do the pushing and popping respectively. However, we can add other operations, and for this problem we will add a third stack operation: multipop(s). Here is the pseudocode for multipop:

```
function multipop(s)
    while S.size > 0 and s > 0:
        S.pop()
        s = s−1
```

Thus multipop pops s many elements from the stack at once, rather than just 1. Like we did for dynamic arrays, an amortized analysis of this data structure involves considering a sequence of n many operations on an initially empty stack S. These operations could be any of the three functions we've mentioned: push, pop, and multipop.

(a) Clearly, the maximum size of the stack after n many operations is n. What, then is the worst case run-time of push(k), pop(), and multipop(s), in terms of n?

(b) Being as pessimistic and mechanistic as possible in your thinking, what is the worst-case runtime for the whole sequence of n operations, in terms of n?

(c) In what way is the runtime from part (b) overly pessimistic? Can you find a more realistic worst-case runtime, $T(n)$? (Hint: is the complexity of multipop limited at all by the stack's history up to that point?)

(d) The aggregate analysis is completed by finding the average runtime per operation, $\frac{T(n)}{n}$. What is this in big-O notation? What can we conclude about the multipop operation from this analysis?

(e) Suppose we added a fourth operation, multipush(k), which pushed k many operations onto the stack. Would that change the average runtime per operation which you found in part (d)?

(5) Finally, some more big-O practice. This time, the practice is meant to get you more accustomed to working with the whole gang.

(a) Prove that if $f(n) = O(g(n))$ then $g(n) = \Omega(f(n))$. (Intended difficulty: Easy)

(b) Prove that if $f(n) = \omega(g(n))$ then $f(n) \neq O(g(n))$ (Intended difficulty: Easy)

(c) Prove that if $f(n) = O(1)$, then $f(n) = \Theta(1)$. (Hint: remember that all functions in our class are assumed to be strictly positive.) (Intended difficulty: Easy)

(d) Prove that $\log(n!) = \Theta(n \log(n))$. I.e. prove that it's both $O(n \log(n))$ and $\Omega(n \log(n))$. To do this, use Stirling's Approximation:

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right) \tag{1}$$

(Hint: The above identity is a product of three terms. Use the fact that $\log(ab) = \log(a) + \log(b)$ to break the function into pieces.) (Intended difficulty: Medium, easier than it looks)

The next three parts (one of which is optional) are all concerning an interesting and moderately annoying question I've run accross while perusing this material myself: For all fixed

$\epsilon > 0$ and all integers k, is it the case that

$$O(n^{k-\epsilon}) \stackrel{?}{=} o(n^k) \tag{2}$$

It seems true enough at a glance (stop a moment and think about it), but this 'identity' is one of the places where our abuse of the equal sign can get us in trouble. First, let's observe that it's at least half true:

(e) Prove that for any $\epsilon > 0$, if $f(n) = O(n^{k-\epsilon})$, then $f(n) = o(n^k)$. (Equivalently, what I'm really asking you to prove is that $O(n^{k-\epsilon}) \subseteq o(n^k)$.)

(f) The $\supseteq$ direction is where our equation fails to hold. A function I found which witnesses this fact is $f(n) = n^{k-\frac{\ln(n)}{n}}$. Prove that $f(n)$ is $o(n^k)$ for any k. (Hint: Remember from calculus that for continuous functions $f(n)$,

$$\lim_{n\to\infty} f(n) = e^{\lim_{n\to\infty} \ln(f(n))} \tag{3}$$

(Intended difficulty: Medium with the hint.)

(g) (Optional) Show that for any $\epsilon > 0$, $f(n) = \omega(n^{k-\epsilon})$ (And so, by part (b), it follows that $f(n) \neq O(n^{k-\epsilon})$). Try Explain in words what is being overlooked in equation (3). (Intended difficulty: Medium)