Hertie School, Spring 2024
Data Structures & Algorithms (Professor Dimmery)
Problem Set 3, Due Monday, April 22

In this problem set, you will implement an algorithm for matrix multiplication using Divide-and-Conquer and study its performance.

You should submit one pdf on Moodle with your solutions. You should prepare all of your writeup in the document, formatting it nicely: as if you were going to be sharing the results with your colleagues and boss. What does this mean?

- Clearly label your solutions.
- Solutions which require explanation should use complete sentences.
- If you use code or materials from anywhere else, that should be clearly labeled and cited.
- If you are asked to prepare graphs or tables of output, they should be labeled clearly. A good rule of thumb is that charts should be self-explanatory: they should not require reading the text in order to understand.

Note that each question/requirement in this problem set has a number of points associated with it. The maximum number of points on this assignment is 15 which corresponds to the number of points on your final grade.

As a reminder, I don't mind if you work together with classmates when you get stuck, but the work you submit should be purely your own. You can also use AI for help, just let me know what kind of help it was.

For this problem set, we'll be working on how to do division. Everything we'll be doing here will assume simple unsigned integers, represented directly as their binary representation. We won't be worrying about two's-complement notation or anything else.

1. (1 point) Write a function which uses numpy to multiple two matrices together. This isn't a trick question: it should just be a one line function.

2. (1 point) Write a function to do matrix multiplication for a 2×2 matrix. You can implement this to use the "grade school" algorithm. That is, to find $C = AB$, just directly write out the math for $C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j}$. (This is just a direct application of the definition of matrix multiplication.)

3. (5 points) Write a function implementing Strassen's algorithm to multiply two matrices together. This was written in pseudo-code on page 48 of Lecture 8. You shouldn't ever have to calculate a matrix product using numpy (but feel free to add and subtract matrices using standard numpy syntax). The base case is the function you wrote for the previous question. Your implementation should have the following elements:

    - It should throw an exception if any dimension of the matrix is not a power of two, if either matrix is not square, and if they are not compatible sizes.
    - A base case (when $n = 2$) that just performs a simple matrix multiplication using your function from question (2).

- Division of the problem into seven different sub-problems, each of which is a matrix multiplication, defined as we did in class. These will require recursive calls of your function!
- Combination of the sub-problem solutions together into a solution for the full problem.

4. (3 points) Show that your implementation of Strassen's algorithm gives the same result as numpy on four test matrices, of size 2×2, 4×4, 8×8 and 16×16. You don't need to write out these (potentially) large matrices by hand, you can create them randomly through, for example, `np.random.randint`.

5. (5 points) Your final task is to measure the *empirical* performance of this algorithm you've written. For full points, you need the following components:

   - Estimate how long it takes to run your functions from question (1) and (3) across a variety of matrix sizes.
   - Plot the runtime of each algorithm as a function of sample size.
   - Use your chart to evaluate how the runtime of your implementation of Strassen's algorithm (and numpy's) grows as a function of matrix size.

   More detailed instructions follow:

   The way to do this is using the timeit package in Python. In particular, if you have a function and you want to see how long it takes to execute, you can do the following:

```python
import timeit
timeit.timeit(lambda: multiply_matrices(A, B), number=1000)
```

   This will run the code `multiply_matrices(A, B)` 1000 times and report the total time execution took. This is a useful way to understand how your algorithm scales in practice.

   You can use this method to time the functions you created in questions (1) and (3). Record their runtime at each matrix size between 2^1 and 2^9 and plot their runtime on a chart, clearly labeling which is which. If you can't get a version of (3) written successfully, you can still do this analysis for (1)!

   A matrix with dimensions of size 2^10 is large enough (over a million elements!) that it will take quite a while to multiply, so you don't need to go that large. For context, in my tests (on Google Colab) numpy can multiply matrices of integers of size 2^9 1000 times in around 20 seconds. If you're finding yourself waiting minutes for the code to run, you've probably done something wrong.

   One nice trick you can use to visually see how fast runtime is growing is to transform your y-axis. For example, the following matplotlib code would transform the y-axis on a chart so that a function that grows at the speed $O(n^5)$ would appear *linear* in the chart.

```python
deg = 5
ax.set_yscale(
    'function',
    functions=(
        lambda x: np.power(x, 1/deg),
        lambda x: np.power(x, deg)
    )
)
```

A function that grows slower than $O(n^5)$ would appear to have a smaller slope on the right side than on the left side, while a function that grows *faster* than $O(n^5)$ would have a higher slope on the right side than the left side. Try this out with a few different (integer) values to get some intuition for this.

What rate does the runtime appear to be growing at? Recall what our theoretical analysis of Strassen's algorithm using the Master Theorem told us the runtime would be. Does that hold for numpy and for your implementation? I don't expect you to see differences in tenths of a degree, so just compare integer values.