Divide & Conquer #2 - Matrix Multiplication

▼ Matrix multiplication problem overview

Let's say we have two matrices that are n x n matrices. The product of these two matrices ($C = A \cdot B$) is also an n-by-n matrix.

Here's an example with $n = 3$:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \cdot \begin{bmatrix} 10 & 11 & 12 \\ 13 & 14 & 15 \\ 16 & 17 & 18 \end{bmatrix} =$$

$$\begin{bmatrix} 1 \cdot 10 + 2 \cdot 13 + 3 \cdot 16 & ... & ... \\ ... & ... & ... \\ ... & ... & 7 \cdot 12 + 8 \cdot 15 + 9 \cdot 18 \end{bmatrix}$$

Each element $C[i][j] = \sum_{1 \le k \le n} A[i][k] \cdot B[k][j]$.

> You take the row (i) of the first matrix, multiply each item by each item in the column (j) of the second matrix, and add the stuff together.

This leads to this naive algorithm:

```
for i in range(1, n):
  for j in range(1, n):
    c[i][j] = 0;
    for k in range(1, n):
      c[i][j] += A[i][k] * B[k][j];
```

This is $\theta(n^3)$.

▼ Divide-and-conquer algorithm for matrix multiplication

We need to split these up into subproblems. Let's rewrite the matrices like this:

$$A = \begin{bmatrix} S & T \\ U & V \end{bmatrix}$$
$$B = \begin{bmatrix} W & X \\ Y & Z \end{bmatrix}$$

S, T, U, V, W, X, Y, Z are a *square piece of the matrix*, one of the four corners of the matrix. They're each of size $\frac{n}{2}$. All of these still comprise all of the matrices A and B.

Now that we have this broken down, we can derive C, our multiplied matrix, like so:

$$C = A \cdot B = \begin{bmatrix} S & T \\ U & V \end{bmatrix} \cdot \begin{bmatrix} W & X \\ Y & Z \end{bmatrix} = \begin{bmatrix} S \cdot W + T \cdot Y & S \cdot X + T \cdot Z \\ U \cdot W + V \cdot Y & U \cdot X + V \cdot Z \end{bmatrix}$$

This gives us eight recursive calls that we need to solve:

$$\begin{matrix} S \cdot W & T \cdot Y \\ S \cdot X & T \cdot Z \\ U \cdot W & V \cdot Y \\ U \cdot X & V \cdot Z \end{matrix}$$

These are not *scalar* operations we're doing (operations with values), these are operations with **matrices**. Each of the results is a $\frac{n}{2}$ matrix.

▼ Runtime analysis (master theorem)

Here's the recurrence relation for this problem:

$$T(n) = 8T(\frac{n}{2}) + \theta(n^2)$$
$$a = 8, b = 2, k = 2$$
$$log_2(8) = 3, 3 > 2$$
$$T(n) = \theta(n^3)$$

We have 8 recursive calls, each of those problem sizes is $\frac{n}{2}$.

Where did that $\theta(n^2)$ come from though? Well, we have to do matrix addition on each matrix (ex. $S \cdot W$). We can do matrix addition in time of

> the size of the matrix. The matrix is n^2, so we do matrix multiplication in $\theta(n^2)$ time.

So our runtime is $\theta(n^3)$, which is really not ideal. *This has the same runtime as our naive algorithm from earlier*. It's also much more complex.

▼ **Better divide-and-conquer matrix multiplication (Strassen's algorithm)**

Let's use an algorithm called **Strassen's algorithm**. Start by writing the matrices the same way:

$$A = \begin{bmatrix} S & T \\ U & V \end{bmatrix}$$
$$B = \begin{bmatrix} W & X \\ Y & Z \end{bmatrix}$$

Let's translate these into only *7 recursive calls* (reducing the number of calls that we need to do):

$$P1 = S \cdot (X - Z)$$
$$P2 = (S + T) \cdot Z$$
$$P3 = (U + V) \cdot W$$
$$P4 = V \cdot (Y - W)$$
$$P5 = (S + V) \cdot (W + Z)$$
$$P6 = (T - V) \cdot (Y + Z)$$
$$P7 = (U - S) \cdot (W + X)$$

> Each of these requires some sort of matrix addition and some sort of matrix multiplication. It's not immediately obvious how we're supposed to drive our result.

Here's the way to get the result:

$$C = \begin{bmatrix} P5 + P4 - P2 + P6 & P1 + P2 \\ P3 + P4 & P5 + P1 - P3 + P7 \end{bmatrix}$$

This is extremely difficult to derive and reduces the number of operations needed by one, but we can verify that it is correct with some algebra:

$$P1 + P2 = (SX - SZ) + (SZ + TZ)$$
$$P1 + P2 = SX + TZ$$

So the cleverness of this is only needing to use *7 recursive problems* instead of 8.

▼ Runtime Analysis (Master Theorem)

$$T(n) = 7T(\frac{n}{2}) + \theta(n^2)$$
$$a = 7, b = 2, k = 2$$
$$log_2(7) > 2$$
$$T(n) = \theta(n^{lg(7)}) \approx \theta(n^{2.81})$$

So because of this (honestly complex) refactoring, we've brought the runtime down to approximately $\theta(n^{2.81})$.