Lecture Notes for CSCI 311: Algorithms

Set 6-Divide & Conquer: Square Matrix Multiplication

Professor Talmage

February 17, 2025

---

1 Algorithm Paradigms

For the rest of the course, we will be learning algorithms for solving various problems of interest. More importantly, we will be learning *algorithm design paradigms*. That is, we will be learning approaches to algorithm design that are useful for solving many different problems. These techniques are the true takeaway from this course. While the algorithms we cover are fundamental knowledge for a computer scientist (watch for them to appear on interviews!), they will only get you so far. You need to be able to apply the ideas and techniques from these specific algorithms to solve new and interesting problems, either variations on those you have seen or which you can decompose to have logically-similar components. To this end, from this point on, you need to constantly ask yourself whether you understand both the algorithm at hand and the idea behind it separate from the particular problem where we are currently applying it.

The first paradigm we will consider, though only briefly by itself, is *Divide & Conquer*. This is the fundamental idea behind recursive solutions: If a problem is intractable for direct solution, but is related to a smaller version of itself, then we can divide it into smaller pieces, solve each of those, and combine the results to get an overall solution. There are many variations of this technique, such as *Decrease & Conquer*, which instead of dividing the problem into fractional parts, shaves a little off the input each time, working much like a loop. In any case, there are three fundamental components of a D&C solution:

1 Split ("Divide")

2 Recur

3 Combine ("Conquer")

The most important constraint is that the input to your recursive calls must be smaller than the original input. (Technically, not always numerically smaller, but must be closer to a base case.)

2 Square Matrix Multiplication

Recall the definition of matrix multiplication:

- The dot product of two vectors is the sum of the products of corresponding elements:

$$\langle 1, 2, 3\rangle \cdot \langle 9, 8, 7\rangle = (1)(9) + (2)(8) + (3)(7) = 46$$

Exercise: What are upper and lower bounds on the runtime of computing a dot product?

- To multiply two $n \times n$ matrices A and B, the first row of A controls the first row of the output, with the jth element of that row the dot product of $A[1][]$ with $B[][j]$ (column j of B). This continues, with the second row of the product formed of dot products of $A[2][]$ with $B[][j]$, and so on. In general, $AB[i, j] = A[i] \cdot B[][j]$.

Exercise: Multiply the following matrices:

$$\begin{pmatrix} 3 & 8 \\ 4 & 10 \end{pmatrix} \begin{pmatrix} 5 & 2 \\ 1 & 1 \end{pmatrix}$$

Exercise: What is the runtime of square matrix multiplication, computed directly?

We can reorganize our solution to explicitly divide and conquer. This depends on the fact that matrix products can be represented by products of submatrices, which allows us to recursively multiply smaller components and combine the results.

- Let A be an $n \times n$ matrix and let $A_{11}, A_{12}, A_{21}, A_{22}$ be the four quadrants of A, each an $n/2 \times n/2$ matrix. Divide another $n \times n$ matrix B similarly.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

- Now we can multiply $A \times B$ as if they were 2×2 matrices, with each element being a matrix product instead of scalar.

$$AB = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

- This requires 4 additions of $n/2 \times n/2$ matrices, plus 8 recursive multiplications of $n/2 \times n/2$ matrices.
- Adding $n/2 \times n/2$ matrices takes $(n/2)^2$ additions and stores, so our non-recursive cost is $O(n^2)$ (Note that considering splitting will not increase them, if we copy the elements to separate sub-matrices instead of just delimiting them.)
- We get a total runtime function of $T(n) = 8T(n/2) + O(n^2)$.

Exercise: Solve this recurrence.

- The Master Theorem gives $T(n) = \Theta(n^3)$, as the recursive cost dominates.

In general, to reduce the cost of a recursive function, there are three different components we might consider trying to reduce. The number of subproblems, the size of subproblems, and the non-recursive (split and combine) costs. These correspond to a, b, and $f(n)$ in the Master Theorem. For SMM, we would particularly like to reduce the number of recursive calls, as that could reduce our runtime.

Exercise:

- Why would we not want to reduce the non-recursive cost?
- Why can no algorithm multiply $n \times n$ matrices in less than $\Omega(n^2)$ steps?
- How might we try to reduce the number of recursive calls?

- Since the recursive cost dominates, we need to find a way to reduce it. Though generally worth exploring, reducing the size of the subproblems (increasing b in the Master Theorem nomenclature) often increases the number of subproblems (a) proportionally, so that we have the same result.
- The product AB has n^2 elements and we must store each of them, which will take $\Omega(n^2)$ time.
- Note that each of the quarters of the input matrices appears more than once in the output. If we can combine them in more interesting ways, perhaps we will need fewer recursive multiplications. We will increase the number of additions and subtractions, but a constant number more of those will be absorbed by the $O(n^2)$ term without affecting the asymptotic behavior of the recurrence.

3 Strassen's Algorithm

As we have suggested, Strassen's algorithm reduces the number of recursive calls, increasing the algorithm's runtime. The idea is to better balance recursive and non-recursive portions of the runtime by finding clever ways to use extra matrix additions and subtractions to eliminate one of the recursive calls. We will not be exploring the algorithm in full detail, as it is quite complex, but the details are in the textbook if you are interested.

The outline of Strassen's algorithm is:

1. Quarter input matrices A and B, as above.
2. By adding and subtracting different pieces of A and B, create 10 intermediate matrices $S_1, \ldots, s_{10}$
   - E.g. $S_3 = A_{21} + A_{22}, S_4 = B_{21} - B_{11}$
3. Do 7 recursive multiplications of $n/2 \times n/2$ matrices to get intermediate matrices $P_1, \ldots, P_7$.
   - E.g. $P_3 = S_3 \times B_{11}, P_4 = A_{22} \times S_4$
4. Add and subtract combinations of P_i's to form the product $C = AB$.
   - E.g. $C_{21} = P_3 + P_4$

The runtime recurrence is $T(n) = 7T(n/2) + \Theta(n^2) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.807}) = o(n^3)$. Thus, Strassen's algorithm is strictly faster than the basic SMM algorithm.

- While better, the algorithm is much harder to generate, prove correct, and remember.
- The constants hidden in the $\Theta(n^2)$ non-recursive portion of $T(n)$ are scarily large, meaning that for small n, Strassen's algorithm is actually *less* efficient than the $\Theta(n^3)$ algorithm.
- Practical system use the simple algorithm when n is small and Strassen's (or similar) when n is very large. They actually switch which they use in the middle of the recursion.

A few other fun facts from the CLRS Chapter 4 notes and Wikipedia:

- CLRS 3rd edition (published 2009) had $\Theta(n^{2.376})$ as the best known bound. That has since been improved, with Wikipedia in 2022 citing a 2021 paper reducing it to $\Theta(n^{2.3728596})$
- The method used for these incremental improvements has been shown to be unable to prove an upper bound less than $n^{2.3725}$.
- Wikipedia now (2025) cites a 2022 paper which modified the previous technique and achieves $O(n^{2.37188}$, as well as 2024 papers achieving $O(n^{2.371552})$ and $O(n^{2.371339})$. But we also have a proof that this new technique will not be able to go below $n^{2.3078}$. Note that the 2.371339 paper is a preprint and not yet peer-reviewed.

- These smaller asymptotic upper bounds are known as "galactic" algorithms, meaning they have astronomically large hidden constants, and are generally not practical.
- No general lower bound larger than $\Omega(n^2)$ is known, though in some limited models we have $\Omega(n^2 \log n)$.