Chapter 2

Divide-and-conquer

This chapter revisits the divide-and-conquer paradigms and explains how to solve recurrences, in particular, with the use of the "master theorem." We first illustrate the concept with Strassen's matrix multiplication algorithm (Section 2.1) before explaining the master theorem (Section 2.2), and finally providing techniques to solve recurrences (Section 2.3). These techniques are further illustrated in the exercises of Section 2.4, with solutions found in Section 2.5.

2.1 Strassen's algorithm

The classical matrix multiplication algorithm computes the product of two matrices of size $n \times n$ with $Add(n) = n^2(n-1)$ additions and $Mult(n) = n^3$ multiplications. Indeed, there are n^2 coefficients to compute, each of them corresponding to a scalar product of size n, thus with n multiplications, $n-1$ additions, and one affectation. Can we do better than this?

Note that the question was raised at a time when it was mainly interesting to decrease the number of multiplications, even though this would imply computing more additions. The pipelined architecture of today's processors allows us to perform, in steady-state mode, one addition or one multiplication per cycle time.

Strassen introduced a new method in his seminal paper [102]. Let us compute the product of two 2×2 matrices:

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \times \begin{pmatrix} e & f \\ g & h \end{pmatrix}$$

We first compute seven intermediate products

$$
\begin{aligned}
p_1 &= a(f-h)\\
p_2 &= (a+b)h\\
p_3 &= (c+d)e\\
p_4 &= d(g-e)\\
p_5 &= (a+d)(e+h)\\
p_6 &= (b-d)(g+h)\\
p_7 &= (a-c)(e+f)
\end{aligned}
$$

and then we can write

$$
\begin{aligned}
r &= p_5+p_4-p_2+p_6\\
s &= p_1+p_2\\
t &= p_3+p_4\\
u &= p_5+p_1-p_3-p_7
\end{aligned}
$$

If we count operations for each method, we obtain the following:

Classic	*Strassen*
$Mult(2)=8$	$Mult(2)=7$
$Add(2)=4$	$Add(2)=18$

Strassen's method gains one multiplication, but at the price of 14 extra additions, thus being worse on modern processors than the classical method for 2×2 matrices. However, it is remarkable that the new method does not require the commutativity of multiplication, and, therefore, it can be used, for instance, with matrices instead of numbers. We can readily use it with matrices of even size n, say $n=2m$. We consider that a,b,c,d,e,f,g,h,r,s,t, and u are matrices of size $m\times m$. So, let $n=2m$, and use the previous approach with submatrices of size $m\times m$. To compute each p_i $(1\leqslant i\leqslant 7)$ with the classic matrix multiplication algorithm, we need m^3 multiplications, thus a total $Mult(n)=7m^3=7n^3/8$. For the additions, we need to add the additions performed in the seven matrix multiplications to form the intermediate products p_i, namely $7m^2(m-1)$, with the number of additions required to form the auxiliary matrices, namely $18m^2$. Indeed, there are 10 matrix additions to compute the p_is, and then 8 other matrix additions to obtain r,s,t, and u. Therefore, we have a total of $Add(n)=7m^3+11m^2=7n^3/8+11n^2/4$.

Asymptotically, the dominant term is in $\frac{7}{8}n^3$ for $Mult(n)$ as for $Add(n)$, and the new method is interesting for n large enough. The intuition is the following: Multiplying two matrices of size $n\times n$ requires $O(n^3)$ operations (both for pointwise multiplications and additions), while adding two matrices of size $n\times n$ requires only $O(n^2)$ operations. For n large enough, matrix additions have a negligible cost in comparison to matrix multiplications (and the main source of pointwise additions is within these matrix multiplications). That was not the case for real numbers, hence, the inefficiency of the method for 2×2 matrices.

Strassen's algorithm is the recursive use of the decomposition explained above. We consider the case in which n is a power of 2, i.e., $n = 2^s$. Otherwise, we can extend all matrices with zeroes so that they have a size that is the first power of 2 greater than n, and replace in the following $\log(n)$ by $\lceil \log(n) \rceil$:

$$(X) \longrightarrow \begin{pmatrix} X & 0 \\ 0 & 0 \end{pmatrix} .$$

Let us consider matrices of size $n \times n$, where $n = 2^s$. We proceed by induction. We use the method recursively to compute each of the matrix products p_i, for $1 \leqslant i \leqslant 7$. We stop when matrices are of size 1 or, better, when Strassen's method is more costly than the classical method, for matrix sizes below a "crossover point." In practice, this crossover point is highly system dependent. By ignoring cache effects, we can obtain crossover points as low as $n = 8$ [50], while [30] determines the crossover points by benchmarking on various systems, and it ranges from $n = 400$ to $n = 2150$.

In the following, we stop the recursion when $n = 1$, and:

- $M(n)$ is the number of multiplications done by Strassen's algorithm to multiply two matrices of size $n \times n$;
- $A(n)$ is the number of additions done by Strassen's algorithm to multiply two matrices of size $n \times n$.

For the multiplications, we have:

$$\begin{cases} M(1) = 1 \\ M(n) = 7 \times M(n/2) \end{cases} \Longrightarrow M(n) = 7^s = 7^{\log(n)} = n^{\log(7)}.$$

As before, additions come from two different sources: the additions that are done in the 7 matrix multiplications (recursive call), and the 18 matrix additions (construction of the p_i's and of r, s, t, and u). We finally have:

$$\begin{cases} A(1) = 0 \\ A(n) = 7 \times A(n/2) + 18 \times (n/2)^2 \end{cases} \Longrightarrow A(n) = 6 \times (n^{\log(7)} - n^2) \qquad (2.1)$$

We explain in Section 2.3 how this recurrence can be solved. Note that the recursive approach has improved the order of magnitude of the total computation cost, not just only the constant (previously, we only had $\frac{7}{8}n^3$ instead of n^3). The new order of magnitude is $O(n^{\log(7)})$ and $\log(7) \approx 2.81$.

Finally, we conclude by saying that Strassen's algorithm is not widely used, because it introduces some numerical instability. Also, there are some algorithms with a better complexity. At the time of this writing, the best algorithm is the Coppersmith–Winograd algorithm, in $O(n^{2.376})$ [27]. The problem of establishing the complexity of matrix product is still open. The only known lower bound is a disappointing $O(n^2)$; we need to touch each coefficient at least once.

Strassen's algorithm provides, however, an excellent illustration of the divide-and-conquer paradigm, that we formalize in the next section through the master theorem.

2.2 Master theorem

Before formulating the master theorem, we need to formalize the divide-and-conquer paradigm that was illustrated in the previous section through the Strassen's algorithm.

DEFINITION 2.1 (Divide-and-conquer)**.** Consider a problem of size n. In order to solve the problem, divide it into a subproblems of size n/b that will allow us to find the solution. The cost of this divide-and-conquer algorithm is then

$$S(n) = a \times S\left(\frac{n}{b}\right) + R(n) \tag{2.2}$$

where $R(n)$ is the cost to reconstruct the solution of the problem of size n from the solutions of the subproblems; it is often equal to $R(n) = c \times n^\alpha$, for some constants c and α. Initially, we often have $S(1) = 1$ (or equal to another constant value).

For instance, with Strassen's algorithm, if we consider the number of additions to be executed in a matrix product, we have $a = 7$, $b = 2$, $\alpha = 2$, and $c = \frac{18}{4}$. Indeed, the product of two matrices of size $n \times n$ is performed by first computing 7 products of matrices of size $n/2 \times n/2$, and reconstructing the solution through 18 additions of matrices of size $n/2 \times n/2$, therefore, $R(n) = 18(n/2)^2 = \frac{18}{4}n^2$. In this case, the initial cost is $S(1) = 0$.

Let us assume that there exists $k \in \mathbb{N}$ such that $n = b^k$, thus $k = \log_b(n)$ and $a^k = n^{\log_b(a)}$. If we develop the formula in equation (2.2), we obtain the following:

$$\begin{aligned} S(n) &= a \times S(\tfrac{n}{b}) \quad + R(n) \\ &= a^2 \times S(\tfrac{n}{b^2}) + a \times R(\tfrac{n}{b}) + R(n) \\ &= \cdots \\ &= a^k \times S(1) \quad + \textstyle\sum_{i=0}^{k-1} a^i \times R(\tfrac{n}{b^i}). \end{aligned}$$

We consider the most usual case in which $R(n) = c \times n^\alpha$, and, therefore, we have $\sigma = \sum_{i=0}^{k-1} a^i \times R(\frac{n}{b^i}) = c \times n^\alpha \sum_{i=0}^{k-1} (a/b^\alpha)^i$.

We then distinguish several cases:

1. $(a > b^\alpha)$: $\sigma = \Theta(n^\alpha \times (\frac{a}{b^\alpha})^k) = \Theta(a^k) \Longrightarrow S(n) = \Theta(n^{\log_b(a)})$;
2. $(a = b^\alpha)$: $\sigma = \Theta(k \times n^\alpha) \Longrightarrow S(n) = \Theta(n^\alpha \times \log(n))$;
3. $(a < b^\alpha)$: $\sigma = \Theta\left(n^\alpha \times \frac{1}{1 - \frac{a}{b^\alpha}}\right) \Longrightarrow S(n) = \Theta(n^\alpha)$.

We have proved the following theorem:

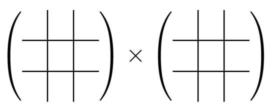
THEOREM 2.1 (Master theorem)**.** *The cost of a divide-and-conquer algorithm such that $S(n) = a \times S(\frac{n}{b}) + c \times n^\alpha$ is the following:*

(i) *if* $a > b^\alpha$, *then* $S(n) = \Theta(n^{\log_b(a)})$;

(ii) *if* $a = b^\alpha$, *then* $S(n) = \Theta(n^\alpha \times \log(n))$;

(iii) *if* $a < b^\alpha$, *then* $S(n) = \Theta(n^\alpha)$.

A fully detailed proof of Theorem 2.1 is given in [28]. Let us come back to Strassen's algorithm. We divided the matrices into four blocs of size $n/2$, and we would like to investigate a solution in which we would rather divide matrices into nine blocs of size $n/3$:



We would then have $b = 3$ and $\alpha = 2$ (the reconstruction cost is still in n^2). Let us assume that we are in case (i) of the master theorem. Then, this new algorithm would become better than Strassen's if and only if:

$$\begin{aligned} & \log_3(a) < \log(7) \\ \iff \quad & \log(a) < \log(7) \times \log(3) \\ \iff \quad & a < 7^{\log(3)} \approx 21.8. \end{aligned}$$

This is an open problem; one knows a method with $a = 23$ subproblems [71], but not with $a = 21$!

2.3 Solving recurrences

In this section, we detail how to solve recurrences that occur in the cost analysis of divide-and-conquer algorithms, but that are slightly more complex than in the application case of the master theorem. We start with homogeneous recurrences, and then consider the most general case of recurrences with a second member.

2.3.1 Solving homogeneous recurrences

A homogeneous linear recurrence with constant coefficients has the form $p_0 \times s_n + p_1 \times s_{n-1} + \cdots + p_k \times s_{n-k} = 0$, where each p_i is a constant and $(s_i)_{i \geqslant 0}$ is an unknown sequence. It is said to be homogeneous because the second member is null, i.e., the linear combination is set equal to zero. Solving such recurrences requires finding all the roots of the polynomial $P = \sum_{i=0}^{k} p_i \times X^{k-i}$, together with their multiplicity order. However, we see that P is a polynomial of degree k, and no algebraic method can find the roots of arbitrary polynomials