Chapter 7

Divide and Conquer

By Sariel Har-Peled, September 16, 2021

Divide and conquer is an algorithmic technique for designing recursive algorithms. Canonical example is probably merge sort, which we assume the reader is already familiar with. Here we discuss the technique and show some other examples.

7.1. The basis technique

The basic idea behind divide and conquer is to divide a problem into two or more parts (i.e., divide), solve the problem recursively on each part, and then put the parts together into a solution to the original instance (i.e., conquer).

7.1.1. Merge sort

Let $X[1 \ldots n]$ be array of $n$ distinct numbers. If $n \leq 1$ then there is nothing to do (always a welcome news). Otherwise, MergeSort computes

$$ m = \lfloor n/2 \rfloor, $$

and breaks the array into two parts

$$ X[1 \ldots m] \quad \text{and} \quad X[m + 1 \ldots n]. $$

It sorts each one of them recursively. Next, it takes the two sorted arrays and merge them into a new sorted array $Y$ – this can be done in linear time by, well, merging the two sorted arrays. Then $Y$ can be returned (or copied back to $X$).

Some implementations of MergeSort use a more complicated algorithm that does the merge of the arrays in place, avoiding the use of the helper array $Y$.

The running time follows the standard recurrence

$$ T(n) = O(n) + 2T(\lfloor n/2 \rfloor), $$

and its solution is $O(n \log n)$.

7.2. Multiplying numbers and matrices

7.2.1. Multiplying complex numbers

In the good old days\(^2\) multiplication took significantly more time than other numerical operations like addition (this is of course true if you have to do the computations by hand). Thus, multiplying two

---

\(^1\)This work is licensed under the Creative Commons Attribution-Noncommercial 3.0 License. To view a copy of this license, visit \url{http://creativecommons.org/licenses/by-nc/3.0/} or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

\(^2\)Or more precisely “good” old days. In this specific case, the 1980s.
complex numbers requires four multiplications, since
\[(\alpha + \beta i)(\alpha' + \beta'i) = \alpha\alpha' + \alpha\beta' i + \beta\alpha' i - \beta\beta' = (\alpha\alpha' - \beta\beta') + (\alpha\beta' + \beta\alpha')i\]

Gauss observed that one can reduce the number of multiplications to three by computing first (using three multiplications) the quantities
\[x = \alpha\alpha', \quad y = \beta\beta', \quad \Delta = (\alpha + \beta)(\alpha' + \beta') = \alpha\alpha'\beta\beta' + \beta\alpha' + \beta\beta'.\]

We then have that
\[(\alpha + \beta i)(\alpha' + \beta'i) = \alpha\alpha' - \beta\beta' + (\alpha\beta' + \beta\alpha')i = x - y + (\Delta - x - y)i.\]

Which means that we reduced the number of multiplications to three (from 4).

### 7.2.2. Karatsuba’s algorithm: Multiplying large integer numbers

The above observation can be extended to large integer numbers being multiplied. So consider two input numbers (say represented in base 10) as
\[x = x_{n-1}x_{n-2} \ldots x_0 \quad \text{and} \quad y = y_{n-1}y_{n-2} \ldots y_0.\]

Assume is a power of 2. We break the two numbers as follows
\[x = 10^{n/2}x_L + x_R \quad \text{where} \quad x_L = x_{n-1} \ldots x_{n/2} \quad \text{and} \quad x_R = x_{n/2-1} \ldots x_0.\]

\[y = 10^{n/2}y_L + y_R \quad \text{where} \quad y_L = y_{n-1} \ldots y_{n/2} \quad \text{and} \quad y_R = y_{n/2-1} \ldots y_0.\]

**Example 7.2.1.**

\[1234 \times 5678 = (100 \times 12 + 34) \times (100 \times 56 + 78)\]
\[= 10000 \times 12 \times 56\]
\[+ 100 \times (12 \times 78 + 34 \times 56)\]
\[+ 34 \times 78\]

Therefore, multiplying to \(n\) digits numbers, requires (naively), four multiplications of numbers with \(n/2\) digits. We get
\[xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y_R) = 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R\]

**Running time analysis of the naive algorithm.** The recursive algorithm perform four recursive multiplications of number of size \(n/2\) each plus four additions and left shifts (adding enough 0’s to the right). Thus the running time follows the recursion
\[T(n) = 4T(n/2) + O(n) \quad T(1) = O(1).\]

And the solution to this recurrence is \(T(n) = \Theta(n^2)\). Disappointing.
Karatsuba’s algorithm. Using Gauss observation, we have

\[ x_{LYR} + x_{RYL} = (x_L + x_R)(y_L + y_R) - x_{LYL} - x_{RYR}. \]

So, the algorithm computes recursively the following three quantities:

\[ \alpha = x_{LYL}, \quad \beta = x_{RYR}, \quad \text{and} \quad \Delta = (x_L + x_R)(y_L + y_R). \]

The desired answer is then

\[ xy = 10^n x_{LYL} + 10^{n/2}(x_{LYR} + x_{RYL}) + x_{RYR} = 10^n \alpha + 10^{n/2}(\Delta - \alpha - \beta) + \beta. \]

This now requires only three recursive calls.

Running time. The running time is given by the recurrence

\[ T(n) = 3T(n/2) + O(n) \quad \text{and} \quad T(1) = O(1). \]

The solution of this recurrence is

\[ T(n) = O(n^{\log_2 3}) = O(n^{1.585}) \]

To see that, use recursion tree method. The depth of recursion is \( L = \log n \). The total work at depth \( i \) is \( O(3^i n/2^i) \) respectively: number of children at depth \( i \) times the work at each child. As such, the total work is therefore

\[ O\left(n \sum_{i=0}^{L} (3/2)^i\right) = (n^{\log_2 3}). \]

Remark 7.2.2. There are better algorithms known by now. Schönhage-Strassen showed in 1971 an algorithm with running time \( O(n \log n \log \log n) \) time using Fast-Fourier-Transform (FFT).

More recently, Martin F¨urer, in 2007, improved the running time to \( O(n \log n 2^{O(\log^* n)}) \).

7.2.3. Strassen algorithm for matrix multiplication

Consider multiplying two matrices of size \( n \times n \), and \( n \) is divisible by two. We then have the following:

\[
\begin{pmatrix}
C_{1,1} & C_{1,2} \\
C_{2,1} & C_{2,2}
\end{pmatrix} =
\begin{pmatrix}
A_{1,1} & A_{1,2} \\
A_{2,1} & A_{2,2}
\end{pmatrix}
\begin{pmatrix}
B_{1,1} & B_{1,2} \\
B_{2,1} & B_{2,2}
\end{pmatrix} =
\begin{pmatrix}
A_{1,1}B_{1,1} + A_{1,2}B_{2,1} & A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\
A_{2,1}B_{1,1} + A_{2,2}B_{2,1} & A_{2,1}B_{1,2} + A_{2,2}B_{2,2}
\end{pmatrix}.
\]

Namely, one can compute the product of two such matrices by computing the product of 8 matrices of size \( n/2 \times n/2 \). It turns out that one can do better, in a similar spirit to the above two algorithms. In particular, the algorithm performs the following seven multiplications of matrices of size \( n/2 \times n/2 \):

\[
\begin{align*}
M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\
M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\
M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\
M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\
M_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\
M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\
M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})
\end{align*}
\]
The algorithm now uses the following formulas:
\[
C_{1,1} = M_1 + M_4 - M_5 + M_7 \\
C_{1,2} = M_3 + M_5 \\
C_{2,1} = M_2 + M_4 \\
C_{2,2} = M_1 - M_2 + M_3 + M_5.
\]
We verify one of these formulas as an example:
\[
C_{1,2} = M_3 + M_5 = A_{1,1}(B_{1,2} - B_{2,2}) + (A_{1,1} + A_{1,2})B_{2,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}.
\]
Namely, one can compute the product of \( n \times n \) matrices by performing seven products of \( n/2 \times n/2 \) submatrices, and then doing additional \( O(n^2) \) work. We get the following recurrence:
\[
T(n) = O(n^2) + 7T(n/2).
\]
Setting \( h = \log_2 n \) (which we assume is an integer), the solution to this recurrence is
\[
T(n) = O\left(\sum_{i=0}^{h} 7^i (n/2^i)^2\right) = O\left(n^2 \sum_{i=0}^{h} (7/4)^i\right) = O\left(n^2 (7/4)^h\right) = O\left(n^{2+\log_2(7/4)}\right) = O(n^{2.807355}).
\]

### 7.3. Maximum subarray

Let \( X[1..n] \) be an array of \( n \) numbers. We would like to compute the indices \( i \leq j \), that maximizes the value of the subarray:
\[
v(i, j) = \sum_{t=i}^{j} X[t].
\]
This problem can be solved in \( O(n \log n) \) time using some data-structure magic. However, there is a direct simple solution using divide and conquer. Consider the middle location \( m = \lfloor n/2 \rfloor \). If \( M[n/2] \) is included in the solution, then we need to find the indices \( i \) and \( j \), that realizes
\[
\left(\max_{i \leq n/2} \ell(i)\right) + \left(\max_{j > n/2} r(j)\right),
\]
where \( \ell(i) = \sum_{t=i}^{n/2} X[t] \), and \( r(i) = \sum_{t=n/2}^{j} X[t] \). These two quantities can be computed directly in \( O(n) \) time using prefix sums. As such, we need to worry only about the possibility that the optimal solution is in the first half or second half of the array, but these subproblems can be solved directly with recursion. We get the running time
\[
T(n) = O(n) + 2T(n/2) = O(n \log n).
\]

**Lemma 7.3.1.** Give an array \( X[1..n] \), computing the maximum subarray can be done in \( O(n \log n) \) time.

### 7.4. Bibliographical notes

**Faster matrix multiplication algorithms.** One can extend Strassen algorithm by using bigger matrices to do the partition (i.e., \( k \times k \) instead of \( 2 \times 2 \)). Let \( O(n^\omega) \) be the fastest possible algorithm for matrix multiplication. It is currently known that \( \omega < 2.37287 \) [AW21]. These algorithms do not seem to be used in practice since the overhead is too large to be useful, and more importantly they are not numerically as stable as the naive algorithm.

---

The algorithm now uses the following formulas:
\[
C_{1,1} = M_1 + M_4 - M_5 + M_7 \\
C_{1,2} = M_3 + M_5 \\
C_{2,1} = M_2 + M_4 \\
C_{2,2} = M_1 - M_2 + M_3 + M_5.
\]
We verify one of these formulas as an example:
\[
C_{1,2} = M_3 + M_5 = A_{1,1}(B_{1,2} - B_{2,2}) + (A_{1,1} + A_{1,2})B_{2,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2}.
\]
Namely, one can compute the product of \( n \times n \) matrices by performing seven products of \( n/2 \times n/2 \) submatrices, and then doing additional \( O(n^2) \) work. We get the following recurrence:
\[
T(n) = O(n^2) + 7T(n/2).
\]
Setting \( h = \log_2 n \) (which we assume is an integer), the solution to this recurrence is
\[
T(n) = O\left(\sum_{i=0}^{h} 7^i (n/2^i)^2\right) = O\left(n^2 \sum_{i=0}^{h} (7/4)^i\right) = O\left(n^2 (7/4)^h\right) = O\left(n^{2+\log_2(7/4)}\right) = O(n^{2.807355}).
\]

### 7.3. Maximum subarray

Let \( X[1..n] \) be an array of \( n \) numbers. We would like to compute the indices \( i \leq j \), that maximizes the value of the subarray:
\[
v(i, j) = \sum_{t=i}^{j} X[t].
\]
This problem can be solved in \( O(n \log n) \) time using some data-structure magic. However, there is a direct simple solution using divide and conquer. Consider the middle location \( m = \lfloor n/2 \rfloor \). If \( M[n/2] \) is included in the solution, then we need to find the indices \( i \) and \( j \), that realizes
\[
\left(\max_{i \leq n/2} \ell(i)\right) + \left(\max_{j > n/2} r(j)\right),
\]
where \( \ell(i) = \sum_{t=i}^{n/2} X[t] \), and \( r(i) = \sum_{t=n/2}^{j} X[t] \). These two quantities can be computed directly in \( O(n) \) time using prefix sums. As such, we need to worry only about the possibility that the optimal solution is in the first half or second half of the array, but these subproblems can be solved directly with recursion. We get the running time
\[
T(n) = O(n) + 2T(n/2) = O(n \log n).
\]

**Lemma 7.3.1.** Give an array \( X[1..n] \), computing the maximum subarray can be done in \( O(n \log n) \) time.

### 7.4. Bibliographical notes

**Faster matrix multiplication algorithms.** One can extend Strassen algorithm by using bigger matrices to do the partition (i.e., \( k \times k \) instead of \( 2 \times 2 \)). Let \( O(n^\omega) \) be the fastest possible algorithm for matrix multiplication. It is currently known that \( \omega < 2.37287 \) [AW21]. These algorithms do not seem to be used in practice since the overhead is too large to be useful, and more importantly they are not numerically as stable as the naive algorithm.