Chapter 7

Divide and Conquer

By Sariel Har-Peled, December 2, 2021[1] Version: 0.3

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. Maximum subarray: A somewhat silly example

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

$$\Bigl(\max_{i \leq n/2} \ell(i)\Bigr) + \Bigl(\max_{j > n/2} r(j)\Bigr), \qquad \text{where } \ell(i) = \sum_{t=i}^{n/2} X[t], \quad \text{and} \quad r(i) = \sum_{t=n/2+1}^{j} X[t].$$

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

Remark 7.1.2. This problem can be solved in linear time using dynamic programming.

7.1.2. Merge sort

Let $X[1 \,..\, n]$ 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

$$X[1 \,..\, m] \qquad \text{and} \qquad X[m+1 \,..\, n].$$

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

$$T(n) = O(n) + 2T(\lceil n/2 \rceil),$$

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

7.1.3. Counting inversions

Given an array $B[1 \,..\, n]$ consider the problem of computing the number of swaps bubble sort would perform sorting B. As a reminder, bubble sort swap only two adjacent numbers if they are in the wrong order. Formally, the *number of inversions* of B is the number of pairs $i < j$ such that $B[i] > B[j]$. Let $\sigma(i)$ be the location of $B[i]$ in the sorted array. Consider the segment Consider connecting the point $(0, i)$ to the point $(1, \sigma(i))$ by an x-monotone curve, such that no three curves intersect in a point. The number of inversions is ***exactly*** the number of intersection points of curves (known, somewhat confusingly, as vertices of the arrangement of curves).

A natural approach to counting the inversions, is to use the rupture and triumph[2] technique. Here, one count the number of inversions in $B[1 \,..\, n/2]$ and $B[n/2+1 \,..\, n]$ recursively – doing this, one might as well sort these two subarrays, and let $C[1 \,..\, n/2]$ and $C[n/2+1 \,..\, n]$ denote the two subarrays after these two parts were sorted/counted. Now, one counts the inversions in C – this can be done using similar ideas to merge in merge sort.

Exercise 7.1.3. *Given $C[1 \,..\, n]$ after its two halves were sorted, describe how to count the number of inversions in C in $O(n)$ time.*

Adding these three numbers of inversions, gives the total number of inversions in the original array B. We thus get the following.

Lemma 7.1.4. *Given an unsorted array $B[1 \,..\, n]$, one can count the number of inversions in B in $O(n \log n)$ time.*

Exercise 7.1.5. *Show how to solve this problem in $O(n)$ time, after a single invocation of sorting.*

[2] Ooops. I mean divide and conquer.

7.2. Closest pair

Problem 7.2.1. Given a set P of n points in the plane, find the pair of points closest to each other. Formally, return the pair of points realizing $\mathcal{CP}(\mathsf{P}) = \min_{\mathsf{p} \neq \mathsf{q},\, \mathsf{p},\mathsf{q} \in \mathsf{P}} \|\mathsf{p} - \mathsf{q}\|$.

There is a beautiful randomized linear time algorithm to solve this problem. Here we present a somewhat slower algorithm that is nevertheless a nice example of the split and crush technique.

Algorithm. The algorithm pre-sort the points of P by their x coordinate, and also by their y-coordinate. This takes $O(n \log n)$ time.

Splitting. An important property the algorithm uses, is that when we split P into two sets, we can also split the two sorted lists storing the points of P (i.e., the points of P in their x and y order), so that each of the two parts has the (two) sorted lists representing its points, and this can be done in linear time.

In this specific case, the algorithm splits the points of P in their median in x (which is readily available because we have the points available sorted in their x-coordinate order) – let x_m be this value. Let $PS_{\leftarrow} = \{(x, y) \in \mathsf{P} \mid x \leq x_m\}$ and $\mathsf{P}_{\rightarrow} == \{(x, y) \in \mathsf{P} \mid x > x_m\}$ be the two parts of P.

The algorithm recursively computes the closet pairs in the two sets. Formally, we have

$$\alpha_{\leftarrow} = \mathcal{CP}(\mathsf{P}_{\leftarrow}) \qquad \text{and} \qquad \alpha_{\rightarrow} = \mathcal{CP}(\mathsf{P}_{\leftarrow}).$$

The crushing. Clearly, $\beta = \min(\alpha_{\leftarrow}, \alpha_{\rightarrow})$ is a decent candidate to be the closet pair distance. Unfortunately, we might still be missing points that are close to each other across the "border" line $x = x_m$. One can use an "elevator" like argument on the strip $\mathcal{S} = [x_m - \beta, x_m + \beta] \times [-infty, +\infty]$ to compute such closest pairs that might be closer than β and they are across the border.

Exercise 7.2.2. *Show how to compute the closest pair in* $\mathsf{P} \cap S$ *in* $O(n)$ *time.*

Taking the closet pair distance computed of these three options, is the true closest pair. We thus get the following.

Theorem 7.2.3. *Let* P *be a set of* n *points in the plane. One can compute the closest pair of points in* P *in* $O(n \log n)$ *time.*

7.3. Multiplying numbers and matrices

7.3.1. Multiplying complex numbers

In the good old days[3] 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 complex numbers requires four multiplications, since

$$(\alpha + \beta i)(\alpha' + \beta' i) = \alpha\alpha' + \alpha\beta' i + \beta\alpha' i - \beta\beta' = (\alpha\alpha' - \beta\beta') + (\alpha\beta' + \beta\alpha')i$$

[3]Or more precisely "good" old days. In this specific case, the 1980s.

Gauss observed that one can reduce the number of multiplications to three by computing first (using three multiplications) the quantities

$$x = \alpha\alpha', \qquad y = \beta\beta, \qquad \Delta = (\alpha+\beta)(\alpha'+\beta') = \alpha\alpha'\alpha\beta' + \beta\alpha' + \beta\beta'.$$

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).

7.3.2. 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

$$x = x_{n-1}x_{n-2}\ldots x_0 \qquad \text{and } y = y_{n-1}y_{n-2}\ldots y_0.$$

Assume is a power of 2. We break the two numbers as follows

$$x = 10^{n/2}x_L + x_R \qquad \text{where} x_L = x_{n-1}\ldots x_{n/2} \text{ and } x_R = x_{n/2-1}\ldots x_0.$$

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

Example 7.3.1.

$$\begin{aligned} 1234 \times 5678 &= (100\times 12 + 34)\times(100\times 56 + 78) \\ &= 10000\times 12\times 56 \\ &\quad +100\times(12\times 78 + 34\times 56) \\ &\quad +34\times 78 \end{aligned}$$

Therefore, multiplying to n digits numbers, requires (naively), four multiplications of numbers with $n/2$ digits. We get

$$xy = (10^{n/2}x_L + x_R)(10^{n/2}y_L + y = 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R$$

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

$$T(n) = 4T(n/2) + O(n) \qquad T(1) = O(1).$$

And the solution to this recurrence is $T(n) = \Theta(n^2)$. Disappointing.

Karatsuba's algorithm. Using Gauss observation, we have

$$x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R.$$

So, the algorithm computes recursively the following three quantities:

$$\alpha = x_L y_L, \qquad \beta = x_R y_R, \qquad \text{and} \qquad \Delta = (x_L + x_R)(y_L + y_R).$$

The desired answer is then

$$xy = 10^n x_L y_L + 10^{n/2}(x_L y_R + x_R y_L) + x_R y_R = 10^n\alpha + 10^{n/2}(\Delta - \alpha - \beta) + \beta.$$

This now requires only three recursive calls.

Running time. The running time is given by the recurrence

$$T(n) = 3T(n/2) + O(n) \qquad\qquad T(1) = O(1).$$

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

Remark 7.3.2. 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 $O(n \log n 2^{O(\log^* n)})$.

7.3.3. 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 $n/2 \times n/2$:

$$\begin{aligned}
M_1 &= (A_{1,1} + A_{2,2})(B_{1,1} + B_{2,2}) \\
M_2 &= (A_{2,1} + A_{2,2})B_{1,1} \\
M_3 &= A_{1,1}(B_{1,2} - B_{2,2}) \\
M_4 &= A_{2,2}(B_{2,1} - B_{1,1}) \\
M_5 &= (A_{1,1} + A_{1,2})B_{2,2} \\
M_6 &= (A_{2,1} - A_{1,1})(B_{1,1} + B_{1,2}) \\
M_7 &= (A_{1,2} - A_{2,2})(B_{2,1} + B_{2,2})
\end{aligned}$$

The algorithm now uses the following formulas:

$$\begin{aligned}
C_{1,1} &= M_1 + M_4 - M_5 + M_7 \\
C_{1,2} &= M_3 + M_5 \\
C_{2,1} &= M_2 + M_4 \\
C_{2,2} &= M_1 - M_2 + M_3 + M_5.
\end{aligned}$$

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

$$T(n) = O\Bigl(\sum_{i=0}^{h} 7^i (n/2^i)^2\Bigr) = O\Bigl(n^2 \sum_{i=0}^{h} (7/4)^i\Bigr) = O\Bigl(n^2 (7/4)^h\Bigr) = O\Bigl(n^{2+\log_2(7/4)}\Bigr) = O(n^{2.807355}).$$

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 2×2). 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.

Bibliography

[AW21] Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In Dániel Marx, editor, *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms, SODA 2021, Virtual Conference, January 10 - 13, 2021*, pages 522–539. SIAM, 2021.