Divide and Conquer

COMP 251, Fall 2024
Will Zahary Henderson and Giulia Alberini

This set of notes provides an overview of the divide-and-conquer paradigm, motivated by examples. These notes go into slightly more depth than the lectures on November 5 and 7.

Contents

1 Definition **2**

2 Master Theorem **2**
2.1 Context 2
2.2 The theorem 4
2.3 Examples 4

3 Algorithm Examples **5**
3.1 Merge sort 5
3.2 Chip testing 5
3.3 Karatsuba multiplication 7
3.4 Strassen matrix multiplication 8
3.5 Fast Fourier transform 10

4 Exercises **11**

5 Exercise Solutions **12**

1 Definition

The **divide-and-conquer** paradigm involves breaking down (“dividing”) a problem into smaller sub-problems recursively, solving the small subproblems (“conquering”), and then merging their results to result in a solution to the original problem.

Applying divide-and-conquer to a problem P admits a nice inductive interpretation:

- **Base case:** if an instance A of P is small enough to solve in a simple manner, return the solution $P(A)$.
- **Inductive step:**
  1. Divide the instance A into several smaller instances of P,
  2. Recurse on each of the smaller instances, and
  3. Merge the results of the recursive solutions to solve A.

You may realize that this is not a brand new concept for us. We have seen recursive algorithms that build up to a larger solution in the past.

One of the nice properties of divide-and-conquer algorithms is that they tend to admit recurrences that are easy to solve using the master theorem, introduced in the next section. We will see many of these in the examples.

2 Master Theorem

2.1 Context

We already know how to solve recurrences using the substitution method and recurrence trees, but there is a nice technique called the **master theorem** that can give us faster solutions to certain types of recurrences that often apply to divide-and-conquer algorithms.

In a recurrence of the form

$$T(n) = aT(n/b) + f(n),$$

- $a \geq 1$ is the number of subproblems to recurse on,
- $b > 0$ is the factor by which the subproblem size decreases, and
- $f(n)$ is the amount of work to “conquer” or merge the subproblems.

In the tree for this recurrence, we have

- $k = \log_b n$ levels,
- a^i subproblems at level i, and
- subproblems of size $f(n)/b^i$ at level i.

There are three main types of tree we can obtain when analyzing these recurrences.

Case 1: the total cost is dominated by the cost of the leaves

The level in the tree that requires the most time is the bottom level: the leaves. For example, consider the recurrence

$$T(n) = 3T(n/2) + n.$$

At level i, there are 3^i subproblems, and each subproblem takes time $n/2^i$. As i increases from 1 to $k = \log_2 3$, the amount of work at level i increases: the root $i = 0$ takes time n, and the leaves take time $3^{\log_2 n} \cdot \frac{n}{2^{\log_2 n}} = n^{\log_2 3}$. The total cost is $\Theta(n^{\log_2 3})$.

Case 2: the total cost is dominated by the cost of the root

The level in the tree that requires the most time is the root, and all levels below it are faster to solve. For example, consider

$$T(n) = 3T(n/4) + n^5.$$

At level i, there are 3^i subproblems, and each subproblem takes time $(n/4^i)^5$. As i increases from 1 to $k = \log_4 3$, the amount of work at level i decreases: the root $i = 0$ takes time n^5, and the leaves take time $3^{\log_4 n} \cdot \left(\frac{n}{4^{\log_4 n}}\right)^5 = n^{\log_4 3}$. The total cost is $\Theta(n^5)$.

Case 3: the total cost is evenly distributed among the levels

Each level in the tree requires the same amount of time. For example, consider

$$T(n) = 2T(n/2) + n.$$

At level i, there are 2^i subproblems, and each subproblem takes time $n/2^i$. Note that at all levels i, the entire level takes time $2^i \cdot \frac{n}{2^i} = n$. Thus, the cost of the entire recurrence is not asymptotically dominated by any particular level, and we take the sum of the $\log_2 n + 1$ levels to obtain a total cost of $n(\log_2 n + 1)$, so the total cost is $\Theta(n \log n)$.

2.2 The theorem

Theorem 1 (Master theorem)**.** Suppose $T(n)$ is a non-negative sequence (or function) that satisfies the recurrence

$$T(n) = aT(n/b) + f(n),$$

where a and b are both constants. We have the following cases corresponding to cases 1, 2, and 3 from before.

(a) If $f(n) = O(n^{\log_b a-\varepsilon})$ for some $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

(b) If $f(n) = \Omega(n^{\log_b a-\varepsilon})$ for some $\varepsilon > 0$ and $\liminf_{n\to\infty}\left(\frac{f(n)}{a\cdot f(n/b)}\right) > 1$, then $T(n) = \Theta(f(n))$.[1]

(c) If $f(n) = \Theta(n^{\log_b a} \cdot \log^p n)$, then $T(n) = \Theta(n^{\log_b a} \cdot \log^{p+1} n)$.

■

Note that not all recurrences can be solved with the master theorem. The number of subproblems should be fixed, meaning a should be a constant. Likewise, we must be careful with case (b) as the difference between $f(n)$ and $n^{\log_b a}$ must be polynomial, made clear by the necessary lower bound on the lim inf of $\frac{f(n)}{af(n/b)}$.

2.3 Examples

Example 1.

$$T(n) = 3T(n/2) + n.$$

We have a recurrence with $a = 3$, $b = 2$, and $f(n) = n$. Since $n = O(n^{\log_2 3-\varepsilon})$ for any $0 < \varepsilon < \log_2 3 - 1 \approx 0.58$, case (a) applies and $T(n) = \Theta(n^{\log_2 3})$.

Example 2.

$$T(n) = 2T(n/2) + n^5.$$

We have a recurrence with $a = 3$, $b = 4$, and $f(n) = n^5$. Since $1 - \log_4 3 > 0$, we can let $\varepsilon = 1 - \log_4 3$ and see that $n^5 = \Omega(n^{\log_4 3+(1-\log_4 3)}) = \Omega(n)$, so the first property of case (b) is satisfied. Since $3(n/4)^5 \leq cn^5$ clearly holds for $c = 1$ and large n, the second property of case (b) is also satisfied. Thus, case (b) applies, and we can conclude that $T(n) = \Theta(n^5)$.

Example 3.

$$T(n) = 2T(n/2) + n\log n.$$

We have a recurrence with $a = 2$, $b = 2$, and $f(n) = n\log n$. Since $n\log n = \Theta(n^{\log_2 2}\log n)$, case (c) applies, and we can conclude that $T(n) = \Theta(n\log^2 n)$.

[1]If you are not familiar with lim inf, this is equivalent to saying that $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n.

3 Algorithm Examples

3.1 Merge sort

Merge sort is one of the classic examples of the divide-and-conquer paradigm. In fact, many divide-and-conquer algorithms can be seen as modified versions of merge sort.

Context

Given a list A of size n, we wish to return a sorted version of A.

Solution

If we have two sorted lists B and C, each of size $n/2$, we can merge them to form a sorted list A of size n by moving the smaller element between the smallest element in B and the smallest element in C and adding it to A. This takes exactly $n-1$ comparisons and is thus done in time $\Theta(n)$.

With this merge step in mind, we can proceed with the divide-and-conquer algorithm on $A = [a_1, \dots, a_n]$ as follows:

1. Divide A in half: $A_1 = [a_1, \dots, a_{n/2}]$ and $A_2 = [a_{n/2+1}, \dots, n]$.
2. Sort on A_1 and A_2 recursively until there is only one element in each.
3. Merge the two sorted lists.

Complexity

We measure the time $T(n)$ to perform merge sort in terms of the number of times we have to compare elements at the merge step. As mentioned before, each merge on two sets of $n/2$ elements takes time $\Theta(n)$. Thus, $T(n)$ satisfies the recurrence

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + n - 1 \approx 2T(n/2) + n,$$

and by the master theorem, $T(n) = \Theta(n \log n)$.

3.2 Chip testing

Context

A factory produces chips that are either *good* (G) or *bad* (B).

We have a testing device where we can insert a pair of chips and each of the two chips will evaluate whether the other is good or bad. Good chips tell the truth; a good chip tells us whether the other chip in the tester is good or bad. On the other hand, bad chips are unreliable and *might* return an incorrect evaluation (that is, they can lie).

The tester has three possible outputs according to the evaluations given by the chips: GG, BB, or BG. GG implies that both chips are good (and both are telling the truth) *or* both chips are bad (and both are lying), while BB and BG both imply that the pair contains at least one bad chip.

Problem: given n chips, we wish to determine the state (G or B) of every chip. The only constraint is that the set of good chips $\mathcal{G}$ is larger than the set of bad chips $\mathcal{B}$, that is, $|\mathcal{G}| > |\mathcal{B}|$. Complexity is measured in terms of the number of times we use the testing device (each use is constant-time).

Solution

Immediately, we may note that if we can confidently identify a single good chip, we can simply test the $n-1$ other chips against it to learn their states.

The problem then becomes finding the single good chip. We can do so with a divide-and-conquer approach.

Let us first assume that n is even. We can proceed as follows:

1. Form $n/2$ pairs of chips and test each pair.
2. Eliminate all pairs that do not return GG.
3. For each remaining pair, eliminate the second chip.
4. Apply steps 1-3 to the remaining chips until we are left with a single chip.

In step 2, since $|\mathcal{G}| > |\mathcal{B}|$ and we only eliminate BB and GB pairs, we are eliminating at least as many bad chips as good chips, and the property $|\mathcal{G}| > |\mathcal{B}|$ is preserved throughout the recursive applications. Therefore, the last chip must be G.

The crucial divide step occurs at step 3. If we did not eliminate half of the chips at this point, we could have an infinite loop.

In the case where n is odd, we first take the n-th chip and perform $n-1$ tests against all the other chips and take the majority vote of the $n-1$ chips, knowing that at least half of the $n-1$ chips are good and will tell the truth. If this n-th chip is deemed good, we have found a good chip; if it is deemed bad, we can discard it and apply the even-case algorithm to the remaining chips $n-1$.

Complexity

Once we have found a good chip, testing against the other $n-1$ chips takes time $\Theta(n)$.

The time to find the good chip $T(n)$ satisfies the recurrence

$$T(n) \leq T\left(\frac{n}{2}\right) + \frac{n}{2}$$

since we test $n/2$ pairs at step 1 and recurse on at most $n/2$ chips (because of step 3). By the master theorem, this takes time $O(n)$.

Thus, the overall algorithm takes time $\Theta(n)$.

3.3 Karatsuba multiplication

Context

Recall that multiplying values in constant time is only viable when we operate on primitive (bounded-sized) values in the RAM model. Multiplying two n-bit numbers, for unknown n, is *not* a constant time operation; the naive multiplication algorithm, similar to “long” or “grade-school” multiplication, takes time $O(n^2)$, but we wish to do better than this.

Consider the situation where we we wish to multiply a_n and b_n, both n-bits[2].

A naïve divide-and-conquer approach

Notice that we can divide the n-bit a_n into two $\frac{n}{2}$-bit parts, α_1 and α_2. For example, if $a_n = 10010101$, we have

$$\alpha_1 = 0101 \quad \text{and} \quad \alpha_2 = 1001$$

and $a_n = \alpha_1 + \alpha_2 \times 2^{n/2}$, where the multiplication $\alpha_2 \times 2^{n/2}$ is performed using bit shifts. We can divide b_n similarly, letting $b_n = \beta_1 + \beta_2 \times 2^{n/2}$.

We can then use divide-and-conquer approach to compute

$$\begin{aligned} a_n \times b_n &= (\alpha_1 + \alpha_2 \times 2^{n/2}) \times (\beta_1 + \beta_2 \times 2^{n/2}) \\ &= \alpha_1\beta_1 + \alpha_2\beta_2 \times 2^n + (\alpha_1\beta_2 + \alpha_2\beta_1) \times 2^{n/2} \end{aligned}$$

by performing each multiplication recursively.

However, the time $T(n)$ to multiply two n-bit numbers in this way satisfies the recurrence

$$T(n) = 4T(n/2) + 2n + 6n$$

where $2n$ comes from the bit shifts and $6n$ comes from the three additions. By the master theorem, $T(n) = \Theta(n^2)$, which is not an improvement over the original multiplication algorithm.

Karatsuba’s innovation

In 1960, Russian mathematician Anatoly Karatsuba realized that

$$\alpha_1\beta_2 + \alpha_2\beta_1 = (\alpha_1 - \alpha_2)(\beta_2 - \beta_1) + \alpha_1\beta_1 + \alpha_2\beta_2.$$

With this formulation, the algorithm now only needs to perform three $\frac{n}{2}$-bit multiplications rather than four.

[2]If one number has fewer bits than the other, we can pad it with zeroes in the start to make them have an equal number of bits. Additionally, we can assume n is even because if not, we can pad it with an extra bit to make it even

The time $T(n)$ to multiply two n-bit numbers with this trick then satisfies the recurrence

$$T(n) = 3T(n/2) + 2n + 10n,$$

where again the $2n$ comes from shifting and the $10n$ comes from additions and subtractions. By the master theorem, this yields $T(n) = \Theta(n^{\log_2 3}) \approx \Theta(n^{1.58})$, a huge improvement over $\Theta(n^2)$.

Further developments

You do not need to memorize these facts; I included them because they are interesting.

Similar strategies have been used to obtain even better time complexities for multiplication. Namely, the **Toom-Cook** algorithm divides a_n and b_n into three parts each instead of two, yielding the recurrence

$$T(n) = 5T(n/3) + O(n)$$

which gives $T(n) = \Theta(n^{\log_3 5}) \approx \Theta(n^{1.46})$ by the master theorem.

From 1971 to 2007, the fastest known multiplication method was the **Schönage-Strassen** algorithm which used recursive applications of the fast Fourier transform (FFT) to multiply two n-bit numbers in time $O(n \log n \log \log n)$.

In 2007, Martin Fürer published a paper[3] that describes an algorithm, also based on FFT, that performs multiplications in time $O\left(n \log n 2^{O(\log^* n)}\right)$, extremely close to $O(n \log n)$.[4]

In 2019, David Harvey and Joris van der Hoeven introduced a galactic algorithm[5] for integer multiplication in time $O(n \log n)$, based on a 1729-dimensional Fourier transform.

3.4 Strassen matrix multiplication

Context

Consider the problem of multiplying two $n \times n$ matrices A and B. The familiar algorithm for matrix multiplication $A \times B = C$ is

$$c_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

which takes $\Theta(n^3)$ time, since we compute a sum of n values for each of the n^2 entries.

[3] https://ivv5hpp.uni-muenster.de/u/cl/WS2007-8/mult.pdf

[4] $\log^* n$, the **iterated logarithm** of n, is the number of times the logarithm must be iteratively applied before the result is less than or equal to 1. It grows very slowly; $\log_2^* n = 5$ for $n \leq 2^{65536}$.

[5] https://en.wikipedia.org/wiki/Galactic_algorithm

A naïve divide-and-conquer approach

Operating under the assumption that $n = 2^k$ for some k (if not, we can pad the matrices with extra zeroes), we can divide each of A and B into four $\frac{n}{2} \times \frac{n}{2}$ submatrices and multiply them to obtain $A \times B$ as follows:

$$\begin{aligned} A \times B &= \left[\begin{array}{c|c} A_1 & A_2 \\ \hline A_3 & A_4 \end{array}\right] \times \left[\begin{array}{c|c} B_1 & B_2 \\ \hline B_3 & B_4 \end{array}\right] \\ &= \left[\begin{array}{c|c} A_1B_1 + A_2B_2 & A_1B_2 + A_2B_4 \\ \hline A_3B_1 + A_4B_3 & A_3B_2 + A_4B_4 \end{array}\right]. \end{aligned}$$

The time to compute this satisfies the recurrence

$$T(n) = 8T(n/2) + O(n^2)$$

since we perform eight multiplications of $\frac{n}{2} \times \frac{n}{2}$ matrices and partitioning the matrices takes time $O(n^2)$. However, by the master theorem, this is $T(n) = \Theta(n^3)$, which does not improve upon the standard algorithm.

Strassen's innovation

In 1969, Volker Strassen published an algorithm that improves upon the $\Theta(n^3)$ bound. Similarly to Karatsuba, Strassen noticed that we can get away with doing one fewer multiplication. The factoring scheme is as follows:[6]

$$\begin{aligned} M_1 &= (A_1 + A_4)(B_1 + B_4), \\ M_2 &= (A_3 + A_4)B_1, \\ M_3 &= A_1(B_2 - B_4), \\ M_4 &= A_4(B_3 - B_1), \\ M_5 &= (A_1 + A_2)B_4, \\ M_6 &= (A_3 - A_1)(B_1 + B_2), \text{ and} \\ M_7 &= (A_2 - A_4)(B_3 + B_4). \end{aligned}$$

With these seven terms in mind, we can express the product $A \times B = C$ as

$$C = \left[\begin{array}{c|c} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ \hline M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{array}\right].$$

The time $T(n)$ then satisfies the recurrence

$$T(n) = 7T(n/2) + O(n^2),$$

and by the master theorem $T(n) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$.

[6]I do not expect you to memorize these; I have not memorized them myself!

Further developments

In 1990, Dan Coppersmith and Shmuel Winograd found an algorithm that performs matrix multiplication in time $O(n^{2.376})$. In 2011, Virginia Vassilevska Williams improved this time to $O(n^{2.373})$.

In 2024, just a few months ago, Williams (and others) discovered a galactic algorithm that runs in time $O(n^{2.371552})$, the best currently known.

The best known theoretical lower bound for matrix multiplication is $\Omega(n^2 \log n)$, which is considerably smaller than the best known algorithm. That is to say, there may be more work for theoretical computer scientists to do!

3.5 Fast Fourier transform

The **discrete Fourier transform** (DFT) is a mathematical transform with numerous applications in signal processing, data encoding, and algorithms. The fast Fourier transform (FFT) is an algorithm that computes the DFT.

One of the classic and most easily-understood objectives of the FFT is multiplication of two n-degree polynomials

$$P(x) = p_0 + p_1 x + \cdots + p_n x^n$$

and

$$Q(x) = q_0 + q_1 x + \cdots + q_n x^n.$$

In 1965, J. W. Cooley and John Tukey introduced an algorithm to perform the FFT, and therefore calculate the product $P(x)Q(x)$, in time $O(n \log n)$. Although we do not have time to cover the algorithm in class, it is an interesting example of divide-and-conquer algorithms and is quite useful to know, especially if you are interested in competitive programming.

If you have the time and the curiosity, I recommend reading the Cooley-Tukey paper.[7]

[7] https://www.ams.org/journals/mcom/1965-19-090/S0025-5718-1965-0178586-1/S0025-5718-1965-0178586-1.pdf

4 Exercises

Exercise 1. Solve the following recurrences with the master theorem, or explain why it does not apply.

1. $T(n) = 3T(n/2) + n^2$
2. $T(n) = T(n/2) + 2^n$
3. $T(n) = 16T(n/4) + n$
4. $T(n) = 2T(n/2) + n \log n$
5. $T(n) = 2^n T(n/2) + n^n$

Exercise 2. In an array of n integers $A = [a_1, \ldots, a_n]$, the number of **inversions** is equal to the number of pairs i, j with $1 \le i < j \le n$ such that $a_i > a_j$. It can be thought of as a metric for "how unsorted" an array is. Write a divide-and-conquer algorithm to calculate the number of inversions in A in time $\Theta(n \log n)$.

Exercise 3. Given an array $A = [a_1, \ldots, a_n]$ containing integers (positive and negative), the **maximum subarray problem** is the task of finding a contiguous subarray $A[i, j] = [a_i, a_{i+1}, \ldots, a_{j-1}, a_j]$ where $\sum_{k=i}^{j} a_k$ is maximized over all subarrays. For example, the maximum subarray in $A = [-3, 6, -5, 8, 3, -4]$ is $A[2, 5] = [6, -5, 8, 3]$ which sums to 12.

(a) Devise a divide-and-conquer algorithm that solves this problem in time $\Theta(n \log n)$.

(b) Try to adjust your algorithm to make it perform in time $\Theta(n)$, still using divide-and-conquer.

Note that there is a solution to this problem that runs in $\Theta(n)$ without using divide-and-conquer, and instead using a trivial form of dynamic programming. Although we have not learned this paradigm yet, the algorithm is very simple, and you may find it easier to come up with than the divide-and-conquer algorithm.

Exercise 4. In a set of points $A \subseteq \mathbb{R}^2$, a point $x \in A$ is on the **convex hull** of A if there exists a line through x that has all other points in A on one side of the line. Write a divide-and-conquer algorithm that finds the convex hull of A in time $\Theta(n \log n)$. (*Hint: it uses a similar framework to merge sort, but the heuristic for the "merge" step is tricky to come up with.*)

5 Exercise Solutions

Solution 1.

1. $\Theta(n^2)$, use case (b)
2. $\Theta(2^n)$, use case (b)
3. $\Theta(n^2)$, use case (a)
4. $\Theta(n \log^2 n)$, use case (c)
5. Master theorem does not apply because $a = 2^n$ is not a constant

Solution 2. Augment the code for merge sort by initializing a variable `numInversions = 0`. At each step where you merge two arrays A and B both of size $n/2$ (where A contains values originally occurring before the values of B in the original array), each time you append a value from B of the array onto the sorted array, increment `numInversions` by $n/2 - i$ where i is the number of values from A that were already added to the sorted array.

Solution 3. See this short paper for a discussion on both versions of the algorithm. The simple dynamic programming solution is called **Kadane's algorithm** and proceeds as follows:

```
max_subarray(A):
    max_sum ← −∞
    curr_sum ← 0
    for each x in A
        curr_sum ← max(x, curr_sum + x)
        max_sum ← max(max_sum, curr_sum)
    return max_sum
```

Solution 4. See Example 4 on page 5 of Luc's notes.

Acknowledgements

These notes are largely based on scribed notes written for Luc Devroye's COMP 252 class. In addition, some definitions and examples are taken from CLRS. I recommend using both of these resources for additional reference.

References

[1] Anna Brandenberger and Anton Malakhveitchouk. Divide and conquer. `https://luc.devroye.org/Brandenberger-Malakhveitchouk--Divide+ConquerI-McGillUniversity-2018.pdf`, 2018.

[2] James W. Cooley and John W. Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of Computation*, 19(90):297–301, 1965.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[4] Martin Fürer. Faster integer multiplication. *SIAM Journal on Computing*, 39(3):979–1005, 2009.

[5] Jason Pizzuco and Adam Wertheimer. A lecture on divide-and-conquer algorithms and the master theorem. `https://luc.devroye.org/Wertheimer-Pizzuco--Divide+Conquer-McGillUniversity-2018.pdf`, 2018.