CS331: Algorithms and Complexity
Part II: Recursion

Kevin Tian

1 Introduction

In these notes, we cover an algorithm design technique called *recursion*, or informally, "divide-and-conquer." To motivate this technique, consider how we typically analyze the correctness of a complicated algorithm with many subroutines, say with the following pseudocode.

Algorithm 1: ComplexAlgo(inputs)

```
SimpleAlgo$_1$(...)
SimpleAlgo$_2$(...)
...
```

Many algorithms follow the template of Algorithm 1, an example of a *reduction.* The main idea of a reduction is to break a complicated task into simpler tasks which are hopefully easier to solve. We can establish correctness of an algorithm following this template:

1. First, show that the reduction is correct, i.e., that the algorithm really does perform the intended task, assuming that all subroutines are implemented correctly.
2. Next, prove that each subroutine similarly correctly matches its specification.

Recursion is perhaps the simplest variant of this template. In recursion, all subroutines are copies of the algorithm itself, but with smaller inputs. For example, MergeSort (recalled in Algorithm 3, Section 5.1) reduces sorting a list to sorting two smaller lists, along with some additional work. As a strategy, recursion greatly simplifies both correctness proofs and runtime analyses.

A recursive algorithm has the following form, for some constants $0 < c_1, c_2, \ldots < 1$.

Algorithm 2: RecursiveAlgo(input of size n)

```
RecursiveAlgo(input of size $c_1 n$)
RecursiveAlgo(input of size $c_2 n$)
...
```

The analysis of a recursive algorithm is essentially a textbook proof by strong induction (see Section 2, Part I). Let $S(n)$ denote the statement that an algorithm completes its intended task on all inputs of size n. Then, assuming the reduction is correct, to prove $S(n)$, it suffices to establish $S(c_1 n)$, $S(c_2 n)$, and so on. This is exactly what strong induction accomplishes! In other words, assuming base cases are correctly implemented, we get correctness of all subroutines "for free" in a recursive algorithm, letting us focus on the reduction itself. The phenomenon of being able to assume correctness of recursive calls is called the "recursion fairy" by [Eri24].

As we will see, recursion is a surprisingly powerful tool in a variety of important problem settings. At a high level, one should be on the lookout for situations where solving the given problem on a subset of the input makes progress towards the overall goal (e.g., sorting the first half of a list clearly helps in sorting the whole list, in a quantifiable way).

We provide several examples of recursion in algorithm design throughout these notes; our primary goal is to illustrate situations where such self-reductions are helpful.

2 Multiplication

As a first introduction, we consider using recursion for one of the most basic computational tasks: multiplying two n-digit numbers, say in base 10. For analyzing solutions to this task, we treat the runtime of 1-digit addition, subtraction, and multiplication as $O(1)$.[1]

Firstly, why multiplication rather than addition? It turns out that from a computational perspective, multiplication is more challenging. Specifically, let a and b be n-digit numbers, and consider the complexity of addition using the "grade-school algorithm" for computing $a + b$, i.e., repeatedly summing digits (possibly with carry-overs). There are at most $n + 1$ digits of the output we need to compute, each of which involves adding at most three one-digit numbers (one each from a and b, and potentially a carry-over). Therefore, we can add two n-digit numbers in $O(n)$ time. A similar argument shows that subtracting two n-digit numbers also takes $O(n)$ time.

Multiplication is a different story. Consider the "grade-school algorithm" for multiplication, e.g.,

$$\begin{array}{r} 1\,2\,3\,4\,5 \\ \times\ 5\,4\,3\,2\,1 \\ \hline 1\,2\,3\,4\,5 \\ 2\,4\,6\,9\,0 \\ 3\,7\,0\,3\,5 \\ 4\,9\,3\,8\,0 \\ 6\,1\,7\,2\,5 \\ \hline 6\,7\,0\,5\,9\,2\,7\,4\,5 \end{array} \tag{1}$$

when $a = 12345$, $b = 54321$, $n = 5$. As we can see, each of the n intermediate sums in this algorithm requires $O(n)$ time (as each digit involves a multiplication and possibly an addition with a carry-over). This gives an overall $O(n^2)$ runtime, significantly more than addition.

Taking our earlier advice, it seems like multiplication is a good opportunity for recursion, as computing products of parts of the inputs helps achieve our final goal; indeed, the grade-school algorithm computes n partial products. Can we do better using recursion?

A basic idea is to divide our inputs a and b into smaller portions, e.g., the decompositions

$$a = 10^{\frac{n}{2}} a_1 + a_0, \; b = 10^{\frac{n}{2}} b_1 + b_0$$

split each input into its $\frac{n}{2}$ most and least significant digits.[2]

Then, we can compute $a \times b$ recursively (i.e., the "FOIL" method):

$$a \times b = 10^n (a_1 \times b_1) + 10^{\frac{n}{2}} (a_1 \times b_0 + a_0 \times b_1) + a_0 \times b_0. \tag{2}$$

We can then subdivide a_0, a_1, b_0, and b_1 further, until the base case (single-digit multiplication).

Letting $\mathcal{T}(n)$ denote the time to multiply two n-digit numbers, let us understand the complexity of the recursive algorithm which repeatedly uses the formula (2). To compute the product of two n-digit numbers, we first need to multiply two $\frac{n}{2}$-digit numbers four times. We then need to perform a constant number of shifts (appending zeroes to the ends of numbers) and additions, all of which take $O(n)$ time. Hence, we have

$$\mathcal{T}(n) = 4\mathcal{T}\left(\frac{n}{2}\right) + O(n). \tag{3}$$

As we will see in Section 3, solving the recursion (3) yields $\mathcal{T}(n) = O(n^2)$. It seems from this exercise that, while our recursive solution helped formalize the grade-school algorithm's correctness, it did not substantially improve its runtime. In fact, the famous mathematician and computer scientist Andrey Kolmogorov once conjectured that this algorithm was optimal, and organized a seminar to prove that subquadratic runtimes for multiplication are impossible.

[1] That is, in this section specifically, we will explicitly be tracking the number of 1-digit arithmetic operations we perform, rather than working in the word RAM model discussed in Section 7, Part I.

[2] If n is odd, we can treat its first digit as 0 and set $n \leftarrow n+1$, which does not affect any of our asymptotics in n.

Amazingly, within a week, a student named Anatoly Karatsuba disproved this conjecture [KO62]. Karatsuba's approach was also recursive, but more clever than that in (2). Indeed, he noticed that just *three* $\frac{n}{2}$-digit multiplications suffice to multiply n-digit numbers:

$$a \times b = 10^n (a_1 \times b_1) + 10^{\frac{n}{2}} ((a_1 + a_0) \times (b_1 + b_0) - a_1 \times b_1 - a_0 \times b_0) + a_0 \times b_0. \tag{4}$$

Observe that (2) and (4) work out to the same expression, since $(a_1 + a_0) \times (b_1 + b_0) = (a_1 \times b_1 + a_0 \times b_0) + (a_1 \times b_0 + a_0 \times b_1)$. However, in Karatsuba's recursion (4), we are able to reuse the fact that we have already computed $a_1 \times b_1$ and $a_0 \times b_0$ to evaluate this middle expression. Karatsuba's recursion does require more additions, and leads to the overall formula

$$\mathcal{T}(n) = 3\mathcal{T}\left(\frac{n}{2}\right) + O(n). \tag{5}$$

This is because all operations required to carry out the formula (4), beyond performing the three smaller multiplications $a_0 \times b_0$, $a_1 \times b_1$, and $(a_1 + a_0) \times (b_1 + b_0)$, are simple digit shifts, additions, or subtractions, all of which take $O(n)$ time. We show in the following Section 3 that the recursion (5) yields an improved runtime of $\mathcal{T}(n) = O(n^{\log_2(3)}) \approx O(n^{1.58})$.

The lesson from this anecdote is that even for simple strategies such as recursion, there can be significant ingenuity involved in designing algorithms which are as efficient as possible. In fact, we will see an even faster multiplication algorithm in Section 6.2.

More generally, many problems we will encounter have multiple routes towards recursively subdividing inputs into smaller subproblems, and aggregating partial computations into an overall solution. While these more clever recursions may seem like magic at first, the best way to get better at spotting them is hands-on experience designing recursive methods.

3 Recurrences

Expressions such as (3), (5) are very common when analyzing runtimes of recursive algorithms. This should be unsurprising: in general, the cost of a recursive algorithm is the cost of its subroutines (which are calls to the algorithm itself, on smaller inputs), plus additional costs for aggregate the subroutine solutions. In this section, we give tools for analyzing such expressions.

One of the most generic tools we will rely on is known as the *master theorem*.

Theorem 1 (Master theorem)**.** *Let $\mathcal{T}, f : \mathbb{N} \to \mathbb{R}_{>0}$ be increasing functions, satisfying the recursion*

$$\mathcal{T}(n) = a\mathcal{T}\left(\frac{n}{b}\right) + f(n), \tag{6}$$

where $a > 0$ and $b > 1$. Let $\tau := \log_b(a)$.

- *Case 1: Leaves-heavy. If $f(n) = O(n^{\tau-\epsilon})$ for a constant $\epsilon > 0$, then $\mathcal{T}(n) = \Theta(n^\tau)$.*
- *Case 2: Balanced. If $f(n) = \Theta(n^\tau \log^k(n))$ for $k \geq 0$, then $\mathcal{T}(n) = \Theta(n^\tau \log^{k+1}(n))$.*
- *Case 3: Root-heavy. If $f(n) = \Omega(n^{\tau+\epsilon})$ for a constant $\epsilon > 0$, then $\mathcal{T}(n) = \Omega(n^{\tau+\epsilon})$. If further, $a\mathcal{T}\left(\frac{n}{b}\right) \leq cf(n)$ for a constant $c < 1$ and all sufficiently large n, then $\mathcal{T}(n) = \Theta(f(n))$. In particular, this applies if $f(n) = \Theta(n^{\tau+\epsilon} \log^k(n))$ for $\epsilon > 0, k \geq 0$.*

Theorem 1 is useful in a wide range of situations. For example, it implies that the solutions to the recursions (3) and (5) are respectively $\mathcal{T}(n) = \Theta(n^2)$, and $\mathcal{T}(n) = \Theta(n^{\log_2(3)})$, as claimed earlier (both are leaves-heavy). However, not all recurrences you will run into "in the wild" have exactly the form (6), so it is good to know how to prove Theorem 1 so we can generalize it.

We first need a basic fact about geometric sequences. Let $a_0, a_1, \ldots, a_k > 0$ be a geometric sequence with common ratio $r > 0$, i.e., $a_i = a_0 \cdot r^i$ for all $i \in [k]$. We proved in Lemma 5, Part I, that for any $r \neq 1$, the sum of this sequence has the formula:

$$a_0 + \ldots + a_k = a_0 \sum_{i=0}^{k} r^i = a_0 \cdot \frac{r^{k+1} - 1}{r - 1}. \tag{7}$$

If the ratio $r < 1$ is a constant, then $\frac{r^{k+1}-1}{r-1} = \frac{1-r^{k+1}}{1-r} \in [1, \frac{1}{1-r}]$ is also a constant, so $a_0 + \ldots + a_k = \Theta(a_0)$, i.e., it is dominated by its first term. This corresponds to the "root-heavy" case.

If $r > 1$ is a constant, then we claim $a_0 + \ldots + a_k = \Theta(a_0 \cdot r^k) = \Theta(a_k)$. A simple way to see this is to reverse the geometric sequence, so it has first term a_k and ratio $\frac{1}{r} < 1$, and we can then apply the previous argument. This corresponds to the "leaves-heavy" case.

If $r = 1$ exactly, then we can no longer use the formula (7), but it is simple to check $a_0 + \ldots + a_k = (k+1)a_0$, as all the terms are the same. This corresponds to the "balanced" case. In this setting, $k \le \lceil \log_b(n) \rceil = O(\log(n))$ is the number of layers before nodes are constant sized.

Consider the recursion (5) as an example, to build intuition for Theorem 1. We make a simplifying assumption for now, to be discussed later: that $\mathcal{T}$ and f extend over the domain $\mathbb{R}_{>0}$ (as defined, they take integers as arguments, but $\frac{n}{b}$ may not be in $\mathbb{N}$). Then, (5) reads:

$$\begin{aligned}
\mathcal{T}(n) &= O(n) + 3\mathcal{T}\left(\frac{n}{2}\right) \\
&= O\left(n + \frac{3n}{2}\right) + 9\mathcal{T}\left(\frac{n}{4}\right) \\
&= O\left(n + \frac{3n}{2} + \frac{9n}{4} + \ldots + \left(\frac{3}{2}\right)^{\log_2(n)} n\right) + O\left(3^{\log_2(n)}\right) \\
&= O\left(3^{\log_2(n)}\right) = O\left(n^{\log_2(3)}\right).
\end{aligned} \tag{8}$$

The third line above can be viewed as an example of a *recursion tree*. Each node of the tree corresponds to a different application of (5). Each size-n node has a base cost of $O(n)$, and spawns three children of half the size. After $\log_2(n)$ such spawns, each node has size 1. At this point, we can use $\mathcal{T}(1) = O(1)$ to bound their costs. There are

$$\left(\frac{3}{2}\right)^{\log_2(n)} n = 3^{\log_2(n)}$$

nodes on this last layer, which dominates the leaves-heavy sum with common ratio $r = \frac{3}{2} > 1$. To see that $3^{\log_2(n)} = n^{\log_2(3)}$, we can take logarithms of each side, both giving $\frac{\log(3)\log(n)}{\log(2)}$.

The basic idea to prove Theorem 1 in general is to create a geometric sequence governing the recurrence. If $f(n) = O(n^c)$, for example, the ratio of the geometric sequence corresponding to (6) is $r = a(\frac{1}{b})^c$. To see why, the base cost of a node with size n in (6) is $f(n) = O(n^c)$. It further spawns a nodes in the next layer with size $\frac{n}{b}$, each incurring an extra cost of $O((\frac{n}{b})^c)$. For example, in (8), the top layer costs $O(n)$, and then the second layer costs $O(3 \cdot (\frac{n}{2})^1)$, and so on. Thus, when $f(n) = O(n^c)$, we expect the cost of the recursion tree to behave like

$$\begin{gathered}
O\left(n^c + a\left(\frac{n}{b}\right)^c + a^2\left(\frac{n}{b^2}\right)^c + \ldots + a^{\lceil \log_b(n) \rceil}\right) = O(n^c) \cdot \left(1 + r + r^2 + \ldots + r^{\lceil \log_b(n) \rceil}\right), \\
\text{where } r := a\left(\frac{1}{b}\right)^c.
\end{gathered} \tag{9}$$

Notice that $a(\frac{1}{b})^c = 1$ precisely when $c = \log_b(a)$, explaining the threshold $\tau = \log_b(a)$ in Theorem 1. Indeed, by plugging (9) into (7) with $a_0 = O(n^c)$, the three claimed cases follow.

We now provide some missing details. A full proof can be found in [CLRS22], Section 4.6.

For the leaves-heavy and balanced cases, we have essentially provided a full proof. To formalize our argument, we simply need to declare a formal inductive hypothesis (the "induction fairy") and add explicit constants everywhere in the proof. The leaves-heavy case is bottlenecked by the fact that there are many leaves ($a^{\log_b(n)} = n^\tau$ of them, up to a constant), all of which cost $\Theta(1)$.

In the balanced case, every recursion level has roughly the same cost, and there are $O(\log(n))$ levels. This argument turns out to generalize just fine to near-polynomials of the form $\Theta(n^\tau \log^k(n))$. Intuitively, each time we halve n, what happens to the logarithmic term is dominated completely by what happens to the polynomial term, due to Lemma 11, Part I.

In the root-heavy case, if we are willing to settle for a crude lower bound of the form $\Omega(n^c)$ for $c = \tau + \epsilon$, then our earlier proof is essentially complete. However, we should not expect the geometric sequence to proceed as in (9) if f is not a polynomial. It turns out that there are pathological f which grow slower than a polynomial but do not satisfy the additional condition in Theorem 1 (see Exercise 4.5-5, [CLRS22]). In such situations, we cannot obtain the tight estimate of $\Theta(f(n))$. Fortunately, all near-polynomials $\Theta(n^c \log^k(n))$, by far the most common runtimes in practice, are not pathological and we can directly apply Theorem 1 in these cases.

Floors and ceilings. Actual algorithms are of course run on inputs with integer-valued sizes, so recurrences in practice do not technically look like, e.g., (3), (5), or (6). For example, consider (5): if n is odd, we cannot split the input exactly in two. A more accurate recursion would be:

$$\mathcal{T}(n) = 3\mathcal{T}\left(\left\lceil \frac{n}{2} \right\rceil\right) + O(n). \tag{10}$$

Here we used the fact that every subproblem involves two numbers of size either $\lceil \frac{n}{2} \rceil$ or $\lfloor \frac{n}{2} \rfloor \le \lceil \frac{n}{2} \rceil$. This seems very unwieldy, but it turns out that these little rounding errors do not matter and Theorem 1 is still true as stated, even if all of the $\frac{n}{b}$-sized subproblems in (6) are replaced with $\lceil \frac{n}{b} \rceil$ or $\lfloor \frac{n}{b} \rfloor$-sized subproblems appropriately, instead. For the rest of course, we will not penalize any such rounding-related issues when dealing with recurrence relations such as (6).

We briefly sketch how to handle this issue, following Section 1.7 of [Eri24]. The basic idea is that while $\mathcal{T}(\lceil \frac{n}{2} \rceil)$ is a strange expression, we have an excellent understanding of $O(\lceil \frac{n}{2} \rceil)$: it is just $O(n)$. So, if we can shift the burden of dealing with floors and ceilings from the $\mathcal{T}(\frac{n}{b})$ component of recurrence relations to the $f(n)$ component, we are in good shape. We assumed that $\mathcal{T}$ and f are monotone, so the following sleight of hand performs this shifting, starting from (10):

$$\begin{aligned}
&\mathcal{T}(n) \le 3\mathcal{T}\left(\frac{n}{2} + 1\right) + O(n) \\
\implies &\mathcal{T}(n+2) \le 3\mathcal{T}\left(\frac{n}{2} + 2\right) + O(n+2) \\
\implies &\mathcal{T}'(n) \le 3\mathcal{T}'\left(\frac{n}{2}\right) + O(n+2) = 2\mathcal{T}'\left(\frac{n}{2}\right) + O(n), \text{ where } \mathcal{T}'(n) := \mathcal{T}(n+2).
\end{aligned}$$

In the first line we used monotonicity of $\mathcal{T}$ and $\lceil \frac{n}{2} \rceil \le \frac{n}{2} + 1$, and in the second line we substituted $n \leftarrow n + 2$ into the first line. The last line defined a new function $\mathcal{T}'$ which gets rid of all rounding errors, and simplifies $O(n+2)$ into $O(n)$. Solving this last recurrence as before using Theorem 1, we obtain $\mathcal{T}'(n) = O(n^{\log_2(3)})$, so $\mathcal{T}(n) = \mathcal{T}'(n-2) = O(n^{\log_2(3)})$ as well.

Akra-Bazzi recurrences. Much heavier hammers than Theorem 1 exist: one example is the "Akra-Bazzi method," which generalizes (6) to handle different-sized subproblems. As an algorithms researcher, I have only ever used this tool a couple of times; it is good to mention that it exists, however. The precise statement is not very instructive for the purpose of this course (you will not need it to solve any problem), but a statement is in Section 4.7, [CLRS22].

4 Orders of magnitude

We take a brief digression to ruminate on runtimes and asymptotics. This discussion guides our later development, when we reflect on and interpret examples in Sections 5 and 6.

Typically, in theoretical computer science, there are four types of runtimes that are most commonly encountered: *exponential*, *polynomial*, *(nearly)-linear*, and *sublinear*.

An exponential runtime for inputs of size n has the form $O(c^n)$ for some $c > 1$, e.g., $O(2^n)$. These are typical of "brute-force" algorithms, and occur in this course when we just want a proof-of-concept that a problem is solvable in finite time. However, they are wholly unwieldy in practice even for moderate n; if $n \approx 250$, 2^n is comparable to the number of particles in the universe. We discuss exponential-time algorithms in greater depth in Part VIII of the notes.

At the other extreme, sublinear algorithms target a runtime budget of $o(n)$; they cannot even read the whole input. For sublinear algorithms to apply, we typically need extra assumptions, as otherwise, important information can be hidden somewhere that we did not read. A common example is assuming that the input has some concise representation, e.g., sparsity. We only briefly touch on sublinear algorithms in this course, primarily in Part VII of the notes.

In most of this course, unless specified otherwise, you are expected to design *polynomial-time algorithms*, i.e., algorithms which run in time $O(n^c)$ for $c \geq 1$. Of course, the exponent matters! The guiding principle behind asymptotics is that a runtime like $O(n \log^2(n))$ is considered "worse" than $O(n \log(n))$ or $O(n)$, but better than a larger polynomial such as $O(n^2)$ (see Lemma 11, Part I). The natural conclusion of this line of thought is that, if we restrict ourselves to algorithms which read the entire input, the gold standard is *nearly-linear* runtimes of the form $O(n \log^k(n))$ for $k \geq 0$, which beat out any larger polynomial $O(n^c)$ for $c > 1$.

However, one must be careful with this logic: asymptotic inequalities such as Lemma 11, Part I only hold in the limit $n \to \infty$. Without thinking too much about the details, it seems like one should clearly prefer an $O(n \log^3(n))$ time algorithm to an $O(n^{1.5})$ time algorithm. Counterintuitively, even if the $O(\cdot)$ hide the same constant, $\log^3(n) > \sqrt{n}$ unless $n \gg 10^7$, so you should actually prefer the latter algorithm unless your input sizes are in the tens of millions![3]

An even more nefarious issue is what constants the $O(\cdot)$ notation is hiding. There are famous examples of recursive algorithms where the "base case" is to solve the problem on an enormous, but constant-sized input; we discuss one such instance in Section 6.1. As a result, the constants in these settings can easily climb to $\gg 2^{250}$, which is the threshold we used to establish that exponential-time algorithms are infeasible. There is a Wikipedia page dedicated to such phenomena [Wik24]; it calls algorithms whose runtimes hide these enormous constants "galactic," as galaxy-sized inputs are the only scale at which it makes sense to run the algorithm over alternatives.

Our goal in this course is to equip you with knowledge to develop computational solutions to real-world problems. Consequently, we aim to be up front when asymptotics are hiding things such as large constants you should be aware about. We will also care about things like improving logarithmic factors in runtimes, as this can quickly make a big difference for small n.

5 Arrays

In this section, we provide a few basic, but sometimes surprisingly creative, applications of recursion. We focus on fundamental algorithms problems on arrays (formally, instances of the data structure Array described in Section 7.1, Part I). This is motivated by the fact that arrays have a naturally recursive structure, e.g., they can be split into halves or even more pieces.

In the remainder of the notes, we use $L[i]$ as shorthand for the i^{th} element in Array L.

5.1 Sorting

To begin, we review the problem of *sorting*. Let L be an Array containing n objects from an ordered universe Ω (such as $\Omega = \mathbb{N}$). Our goal is to return L' containing all the elements of L, but rearranged in sorted (say, nondecreasing) order. There are many brute-force solutions that solve this problem in $O(n^2)$ time, such as computing the rank of each element $x \in L$ in $O(n)$ time, and placing it in the corresponding slot in L'. Here, we briefly reproduce the famous fact that sorting a list can actually be accomplished in time $O(n \log(n))$. This runtime is provably optimal in the computational model where one can only access the list's entries by comparing their relative order, so it is often treated as a natural stopping point for sorting algorithms.[4]

We produce pseudocode for a classic $O(n \log(n))$ time sorting algorithm in Algorithm 3.

We next analyze Algorithm 3's correctness: why does it sort the input list? As discussed in Section 1, we need to show two things. Firstly, we must show that if we assume that all subroutines called by Algorithm 3 do their job successfully (i.e., that Line 6 sorts L_1 and L_2 correctly), then the rest of the algorithm also does its job by sorting L. Secondly, we must show that Line 6

[3] Thanks to Yin Tat Lee for this motivating example.

[4] This is called the *comparison-based lower bound* for sorting, and the argument is roughly as follows. Given h comparisons, the maximum number of distinct outputs we could produce is 2^h, since each comparison has one of two outcomes. We thus need $h \geq \log_2(n!)$ to successfully sort, since there are $n!$ possible permutations (orderings) of the list. We showed in Lemma 14, Part I that this implies $h = \Omega(n \log(n))$. Interestingly, it is possible to improve upon this runtime in more restricted settings, e.g., the RadixSort algorithm for sorting integers.

Algorithm 3: MergeSort(L)

```
1  Input: $L$, an Array instance containing $n := |L|$ objects from the ordered universe $\Omega$
2  if $n == 1$ then
3      return $L$
4  end
5  $L_1, L_2 \leftarrow$ Array instances containing the first $\lceil \frac{n}{2} \rceil$ and last $\lfloor \frac{n}{2} \rfloor$ entries of $L$, respectively
6  $L_1, L_2 \leftarrow$ MergeSort$(L_1)$, MergeSort$(L_2)$
7  $i_1 \leftarrow 1, i_2 \leftarrow 1$
8  for $i \in [n]$ do
9      if $L_1[i_1] \le L_2[i_2]$ then
10         $L$.Insert$(i, L_1[i_1])$
11         $i_1 \leftarrow i_1 + 1$
12     end
13     else
14         $L$.Insert$(i, L_2[i_2])$
15         $i_2 \leftarrow i_2 + 1$
16     end
17 end
18 return $L$
```

actually meets its specifications. Fortunately, because Algorithm 3 is a recursive algorithm (its only subroutines are copies of itself on smaller inputs), the "recursion fairy" (strong induction) completes the second task for free, so we can focus on the first.

To complete our correctness proof, we show that the i^{th} loop of Lines 8 to 17 puts the i^{th} largest entry of the input into $L[i]$, assuming L_1 and L_2 are sorted when the loop begins. We induct on i. Suppose that after the first $i-1$ loops, the $i-1$ smallest entries of the input have been passed over (i.e., they are the entries $L_1[1], \ldots, L_1[i_1 - 1]$, and $L_2[1], \ldots, L_2[i_2 - 1]$). The i^{th} largest input entry must be in either L_1 or L_2 at this point, but because all smaller entries have been passed, and L_1, L_2 are sorted, it is either $L_1[i_1]$ or $L_2[i_2]$. Thus, placing the smaller of $L_1[i_1]$ or $L_2[i_2]$ into $L[i]$, and incrementing the relevant counter, completes the induction. This is exactly what Lines 8 to 17 do, so the output list is sorted after the n loops have completed.

Finally, we analyze the runtime of Algorithm 3. It is clear that all of the work outside the recursive calls in Line 6 take $O(n)$ time, because each loop of Lines 8 to 17 takes $O(1)$ time. Thus, Algorithm 3's runtime satisfies the recurrence

$$\mathcal{T}(n) \le 2\mathcal{T}\left(\frac{n}{2}\right) + O(n). \tag{11}$$

Applying the balanced case of Theorem 1 now shows the claimed $\mathcal{T}(n) = O(n \log(n))$. We mention that (11) is arguably the most famous runtime recurrence in theoretical computer science, and indeed, we run into this same recurrence several more times in these notes.

5.2 Inversions

In this section, we present another common comparison-based problem on arrays. Again, let L be an Array of n objects from an ordered universe Ω. Consider computing the number of inversions in L, i.e., the number of ordered pairs of indices $(i, j) \in [n] \times [n]$, with $L[i] > L[j]$ but $i < j$.

We denote this problem by Inversions(L). Inversions are a natural metric on how far a list is from being sorted. This metric is known as the *Kendall tau distance*, and is often used in practice to compare rankings. Indeed, if L is sorted in nondecreasing order, Inversions$(L) = 0$. Conversely, Inversions(L) is maximized (at $\binom{n}{2}$) when L is in decreasing order and all objects are distinct.

There is a brute-force algorithm that simply checks whether each pair $i < j$ is an inversion, using $O(n^2)$ time. How can we use recursion to speed this up? A natural starting point is recursively splitting L into two contiguous lists L_1 and L_2 of roughly half the size, and computing $c_1 =$ Inversions(L_1) and $c_2 =$ Inversions(L_2). This counts all inversions (i, j) where both i and j are in L_1 or L_2. It remains to compute c_3, the number of pairs $i < j$ where $L[i]$ is in L_1 but $L[j] > L[i]$ is in L_2. In other words, for each $L[i]$ in L_1, we want to know how many entries are smaller in L_2. A priori, this seems to take $O(n)$ time per $L[i]$, or $O(n^2)$ total time again.

The key observation is that this would be very simple if L_2 was already sorted. For example, we could binary search over L_2 in $O(\log(n))$ time, to determine where to place $L[i]$. This also tells us $L[i]$'s contribution to c_3. Let $\mathcal{T}(n)$ denote the time it takes to solve $\mathsf{Inversions}$ on size-n lists, and $\mathcal{S}(n)$ denote the time it takes to sort a size-n list. Then, we have shown the recursion

$$\mathcal{T}(n) = 2\mathcal{T}\left(\frac{n}{2}\right) + \mathcal{S}\left(\frac{n}{2}\right) + O\left(\frac{n}{2}\log\left(\frac{n}{2}\right)\right) = 2\mathcal{T}\left(\frac{n}{2}\right) + O(n\log(n)). \tag{12}$$

The first equality in (12) follows from calling $\mathsf{Inversions}$ twice, sorting L_2, and binary searching to insert each of the entries from L_1 into the newly sorted L_2. We also used $\mathcal{S}(n) = O(n\log(n))$, as proven using $\mathsf{MergeSort}$. Applying Theorem 1 to (12) then yields $\mathcal{T}(n) = O(n\log^2(n))$.

In fact, we can do better. Let us re-examine the loop in Lines 8 to 17. As our inductive hypothesis (the "recursion fairy"), we can assume that L_1 and L_2 are sorted. Now, consider an entry $L_1[i^\star]$. It is inserted into L through the interleaved merging process in Lines 8 to 17. Concretely, it is inserted when all smaller entries in L_1 have been inserted (so the i_1 counter has sequentially reached $i^\star$), and further, $L_2[i_2 - 1] < L_1[i^\star] \le L_2[i_2]$ for the current value of i_2.

Revisiting our recursive $\mathsf{Inversions}$ solution, we can now answer: what are all the objects in L_2 that are smaller than $L_1[i^\star]$? They are exactly the objects inserted into our list L by $\mathsf{MergeSort}$, at the time when $i_1 = i^\star$. Moreover, we know exactly how many such objects there are: $i_2 - 1$, i.e., all the entries of L_2 that have already been merged. Hence, we can directly piggyback off of $\mathsf{MergeSort}$ to implement $\mathsf{Inversions}$! We provide pseudocode in Algorithm 4.

Algorithm 4: $\mathsf{Inversions}(L)$

```
1  Input: $L$, an Array instance containing $n := |L|$ objects from the ordered universe $\Omega$
2  if $n == 1$ then
3      return $(L, 0)$
4  end
5  $L_1, L_2 \leftarrow$ Array instances containing the first $\lceil \frac{n}{2} \rceil$ and last $\lfloor \frac{n}{2} \rfloor$ entries of $L$, respectively
6  $(L_1, c_1), (L_2, c_2) \leftarrow \mathsf{Inversions}(L_1), \mathsf{Inversions}(L_2)$
7  $i_1 \leftarrow 1, i_2 \leftarrow 1, c_3 \leftarrow 0$
8  for $i \in [n]$ do
9      if $L_1[i_1] \le L_2[i_2]$ then
10         $L.\mathsf{Insert}(i, L_1[i_1])$
11         $c_3 \leftarrow c_3 + (i_2 - 1)$
12         $i_1 \leftarrow i_1 + 1$
13     end
14     else
15         $L.\mathsf{Insert}(i, L_2[i_2])$
16         $i_2 \leftarrow i_2 + 1$
17     end
18 end
19 return $(L, c_1 + c_2 + c_3)$
```

Algorithm 4 is identical to Algorithm 3, outside of Line 11, which takes $O(1)$ time per loop ($O(n)$ in total), and returning an inversion count in addition to a sorted list. Thus, the runtime of Algorithm 4 satisfies (11), so it takes $O(n\log(n))$ time, improving our earlier solution.

The key lesson is: when designing a recursive algorithm, you can enforce any helpful invariant you need from the subproblems, as long as you can efficiently preserve the invariant recursively. For example, in Algorithm 4, we took extra effort to make sure our subproblems sorted their input lists, rather than just returning inversion counts. This effort paid off in the combining step (i.e., Lines 8 to 18, outside of the recursive calls), since it dramatically simplified computation of c_3. Finally, leveraging our $\mathsf{MergeSort}$ know-how, we showed that we can both perform the combining step and recursively maintain our sorted list invariant in $O(n)$ time.

5.3 Selection

Again, let L be a list of n objects from an ordered universe Ω. One of the most basic queries over L is selection. Specifically, let Selection(L, i) denote the problem of returning the i^{th} largest object in L. Clearly, we can implement Selection(L, i) by first sorting L (i.e., using MergeSort), and then returning the i^{th} entry of the sorted list. This requires $O(n \log(n))$ time.

However, for small values of n we can do significantly better. For example, if $i = \sqrt{n}$, here is a faster algorithm. We can first insert all of the objects in L into a Heap data structure (Section 7.2, Part I), which takes $O(n)$ time. We can then call ExtractMin$()$ for $i = \sqrt{n}$ times, which takes $O(\sqrt{n} \log(n))$ time and does not dominate. More generally, if $i = O(\frac{n}{\log(n)})$, this implementation of Selection runs in $O(n)$ time. Can we do better in general, e.g., if $i = \Theta(n)$?

A basic motivation for Selection is QuickSort, a randomized algorithm which first selects a *pivot* entry $L[j]$, and then buckets the remaining entries into two categories: those that are at most $L[j]$, and those that are larger. It then recursively sorts the two halves, using $O(n)$ runtime per recursion level. If the pivot's *rank*, i.e., the value k such that $L[j]$ is the k^{th} largest entry in L, satisfies $\frac{n}{3} \le k \le \frac{2n}{3}$ for instance, then the recursion tree for QuickSort terminates after $O(\log(n))$ levels, because each level decreases node sizes by a constant factor. This would lead to an overall runtime of $O(n \log(n))$ for QuickSort. Unfortunately, the simplest implementation of pivot selection, choosing a random entry, can potentially repeatedly choose poor-quality pivots and require $O(n^2)$ time. If we can implement a median finder, Selection$(L, \lceil \frac{n}{2} \rceil)$, in $O(n)$ time, we can use it to select pivots, and obtain a deterministic QuickSort in $O(n \log(n))$ time.

It turns out that we can implement Selection(L, i) in $O(n)$ time for any $i \in [n]$. In this section, we explain such a *linear-time selection* algorithm, following the presentation of [Eri24]. Algorithm 5 gives a simple skeleton implementation, assuming access to the following subroutines.

- FindPivot(L): on input L, a list of objects from Ω, returns an object $x \in L$.
- Pivot(L, x): on input L, a list of objects from Ω, and $x \in L$, returns (L', k). L' is a permutation of L satisfying: $L'[k] = x$, $L'[i] \le x$ for all $i < k$, and $L'[i] > x$ for all $i \ge k$.

There is a simple implementation of Pivot(L, x) in $O(n)$ time, which iteratively passes through the list to compute the rank k of x, inserts x in the k^{th} entry of the output, and then finally performs another iterative pass to bucket the remaining entries. We postpone Algorithm 6, our implementation of FindPivot, for now (as you may guess, it is recursive). In Lines 11 and 14 of the following Algorithm 5, $L[a : b]$ denotes the sublist formed by all $L[i]$ with $a \le i \le b$.

Algorithm 5: Selection(L, i)

```
1  Input: $L$, an Array instance containing $n := |L|$ objects from the ordered universe $\Omega$, $i \in [n]$
2  if $n == 1$ then
3      return $L[1]$
4  end
5  $x \leftarrow$ FindPivot$(L)$
6  $(k, L) \leftarrow$ Pivot$(L, x)$
7  if $k == i$ then
8      return $L[k]$
9  end
10 else if $i < k$ then
11     return Selection$(L[1 : k - 1], i)$
12 end
13 else
14     return Selection$(L[k + 1 : n], i - k)$
15 end
```

It is straightforward to verify correctness of Selection. If the pivot x we find on Line 5 is the i^{th} largest object, we return it. Otherwise, there are two cases. If the desired output is to the left of $x = L[k]$ after Line 6 has executed, i.e., $i < k$, we recursively select the i^{th} largest object amongst $L[1 : k - 1]$, and this is handled by Line 11. If it is to the right of x (i.e., $i > k$), we similarly must select the $(i-k)^{\text{th}}$ largest object amongst $L[k+1:n]$, handled by Line 14. This is because all of the k elements we removed, $L[1:k]$, are all smaller than the desired output.

How fast is Algorithm 5? We already argued that Line 6 takes $O(n)$ time, and Lines 2 to 4 and 7 to 9 clearly take $O(1)$ time. The only other steps are one call to FindPivot on Line 5, and one recursive call to Selection on either Line 11 or 14. If we can bound the costs of these two steps by a low-order term, we obtain an $O(n)$ runtime via a geometric sequence. Can we implement FindPivot cheaply, so that it recursively reduces sublist sizes by a constant factor?

It is natural to guess that Selection can itself be used to implement FindPivot recursively. It takes quite a bit of ingenuity to do so in a way which decreases the overall cost per recursion level. The first such successful implementation of FindPivot was the *median-of-medians* (henceforth, MoM) method of [BFP+73]. MoM is described in Algorithm 6, and uses the following subroutine.

- ShortMedian(L): on input L, a list of at most 5 objects from Ω, returns the median of L.

Because ShortMedian is only run on inputs of size at most 5, it takes $O(1)$ time.

Algorithm 6: MedianOfMedians(L)

```
1 Input: $L$, an Array instance containing $n := |L|$ objects from the ordered universe $\Omega$
2 $M \leftarrow$ Array.Init($n$)
3 for $j \in [\lceil \frac{n}{5} \rceil]$ do
4     $M$.Insert(ShortMedian($L[5(j-1)+1 : \min(5j, n)]$), $j$)
5 end
6 return Selection($M, \lceil \frac{n}{10} \rceil$)
```

The for loop in Lines 3 to 5 of Algorithm 6 repeatedly peels off a sublist of at most 5 items from L, computes the median of this sublist using ShortMedian, and adds it to M. We then return the median of M using Selection, explaining the name: median-of-medians.

The cost of Algorithm 6 is $O(n)$ time to run Lines 3 to 5, plus a recursive call to Selection. The magical property of MedianOfMedians is that, by using it as our FindPivot algorithm in Line 5 of Algorithm 5, we claim that we can guarantee the recursively generated sublists on Lines 11 and 14 are bounded in size by $\frac{7n}{10}$, up to rounding errors. Moreover, M itself is bounded in size by $\frac{n}{5}$, again up to rounding. Therefore, assuming our claim about the quality of MedianOfMedians as a pivot is true, the runtime of Algorithm 5 on length-n lists L, denoted $\mathcal{T}(n)$, satisfies the recursion

$$\mathcal{T}(n) \le \mathcal{T}\left(\frac{n}{5}\right) + \mathcal{T}\left(\frac{7n}{10}\right) + O(n). \tag{13}$$

The $\mathcal{T}(\frac{n}{5})$ term is due to the recursive call in Line 6 of Algorithm 6, the $\mathcal{T}(\frac{7n}{10})$ term is due to either Line 11 or 14 of Algorithm 5, and the $O(n)$ term covers all other costs of Algorithms 5 and 6.

Because $\frac{1}{5} + \frac{7}{10} = \frac{9}{10} < 1$, each layer of the resulting recursion tree (e.g., a variant of the derivation in (8)) has a total size bounded by $\frac{9}{10}$ of the previous layer. This a geometric sequence with common ratio < 1, so it yields a root-heavy recursion, and (13) resolves to $\mathcal{T}(n) = O(n)$ as claimed.

It remains to prove that MedianOfMedians gives a good pivot. We show that, up to rounding errors, the rank of the output x is in $[\frac{3n}{10}, \frac{7n}{10}]$. If this is true, then both buckets in the list pivoted around x have size at most $\frac{7n}{10}$. Now, observe that x is at least as large as half the objects in M ($\frac{n}{10}$ total), as it is the median. Moreover, these $\frac{n}{10}$ objects are themselves medians of length 5 sublists, so they are each at least as large as 2 other objects in their own sublists. This leads to a total of $\frac{n}{10} + 2 \cdot \frac{n}{10} = \frac{3n}{10}$ distinct objects in L guaranteed to be at most x. Symmetrically, x is guaranteed to be smaller than $\frac{3n}{10}$ distinct objects, concluding the proof.

Unfortunately, the constants hidden by our analysis are somewhat unwieldy. For instance, the recursion tree resulting from (13) has a geometric ratio of $\frac{9}{10}$, which translates to a 10× overhead over the root cost. Further, the constant factors in MedianOfMedians are themselves large, e.g., the runtime of ShortMedian, which we budgeted as $O(1)$. To our knowledge, randomized or hybrid pivot methods are more commonly preferred in practice for implementing selection.

6 Matrices

In this section, we cover several algorithms for *matrix multiplication* (see Section 5.1, Part I for relevant definitions). Matrix multiplication is one of the most fundamental problems in algorithms, and we will see applications of matrix multiplication methods all over the course. For example, in Section 6.2 we show how to use a fast structured matrix multiplication algorithm to improve our integer multiplication algorithms in Section 2.

In matrix multiplication, the input is two matrices: $\mathbf{A} \in \mathbb{R}^{n \times d}$, and $\mathbf{B} \in \mathbb{R}^{d \times k}$, for dimensions $n, d, k \in \mathbb{N}$. We assume that all entries of $\mathbf{A}$ and $\mathbf{B}$ fit in one word of memory (see Section 7, Part I); usually this just means they are bounded by $\text{poly}(n, d, k)$ in size. The goal is to compute the matrix product $\mathbf{AB} \in \mathbb{R}^{n \times k}$, defined as:

$$[\mathbf{AB}]_{ij} = \mathbf{A}_{i:}^\top \mathbf{B}_{:j} = \sum_{\ell \in [d]} \mathbf{A}_{i\ell} \mathbf{B}_{j\ell}, \text{ for all } i \in [n],\ j \in [k]. \tag{14}$$

In other words, the $(i, j)^{\text{th}}$ element of $\mathbf{AB}$ is just the inner product between the i^{th} row of $\mathbf{A}$ and the j^{th} column of $\mathbf{B}$, viewed as vectors. How expensive is this computation?

The starting point is the case when $\mathbf{A} = \mathbf{a}^\top \in \mathbb{R}^{1 \times d}$ and $\mathbf{B} = \mathbf{b} \in \mathbb{R}^{d \times 1}$ are themselves vectors (one-dimensional matrices), so $n = k = 1$, and the goal is to compute $\mathbf{a}^\top \mathbf{b} = \langle \mathbf{a}, \mathbf{b} \rangle = \sum_{i \in [d]} \mathbf{a}_i \mathbf{b}_i$. Under pretty much any reasonable model of representing $\mathbf{a}, \mathbf{b}$, e.g., as Array instances, we can compute this inner product in $O(d)$ time using additions and multiplications.

Generalizing this argument, if $\mathbf{A} \in \mathbb{R}^{n \times d}$ and $\mathbf{B} \in \mathbb{R}^{d \times k}$ are represented as two-dimensional arrays, i.e., with each row stored as an Array inside a larger Array, we can compute their matrix product (14) using nk different inner products. Thus, the overall cost of matrix multiplication using this straightforward method is $O(ndk)$. Unfortunately, this is not a linear runtime: the input size is only $O(nd + dk)$, the number of entries in $\mathbf{A}$ and $\mathbf{B}$. For example, when $n = d = k$ (i.e., both matrices are square), straightforward matrix multiplication takes time $O(n^3)$ and the input size is $O(n^2)$. We describe algorithms which partially close this gap in Section 6.1.

Similarly, the cost of straightforward matrix-vector multiplication (between $\mathbf{A} \in \mathbb{R}^{n \times d}$ and $\mathbf{v} \in \mathbb{R}^{d \times 1}$) is $O(nd)$, since it reduces to n inner products. This actually is a linear time algorithm for dense inputs, because $\mathbf{A}$ has size $O(nd)$. However, if the matrix $\mathbf{A}$ is more structured, it can take much less than $O(nd)$ time to describe. We give an improved matrix-vector multiplication algorithm for a specific $\mathbf{A}$, the *discrete Fourier transform* (DFT), in Section 6.2. Matrix-vector products with the DFT matrix are instrumental in many signal processing applications; we show how faster DFT products speed up scalar multiplication, as well.

Finally, in Section 6.3, we use matrices to speed up recurrences, e.g., Fibonacci numbers.

6.1 Strassen's algorithm

We first consider the simplest case of matrix multiplication, when the inputs $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{n \times n}$ are square, and we want to compute their product $\mathbf{AB} \in \mathbb{R}^{n \times n}$. As discussed earlier, the straightforward method for computing $\mathbf{AB}$ takes $O(n^3)$ time. Can we do better in general?

In 1969, Volker Strassen discovered a faster matrix multiplication algorithm [Str69], very similar in spirit to Karatsuba's algorithm (Section 2). To motivate Strassen's algorithm, let us first consider a naïve application of recursion to matrix multiplication. For simplicity, assume n is even (handling rounding errors as discussed in Section 3). Then, we partition $\mathbf{A}$ and $\mathbf{B}$, defining:

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix}, \ \mathbf{B} = \begin{pmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{pmatrix}, \\ \text{for } \mathbf{A}_{11},\ \mathbf{A}_{12},\ \mathbf{A}_{21},\ \mathbf{A}_{22},\ \mathbf{A}_{11},\ \mathbf{A}_{12},\ \mathbf{A}_{21},\ \mathbf{A}_{22} \in \mathbb{R}^{\frac{n}{2} \times \frac{n}{2}}. \tag{15}$$

Because the inner product of n-dimensional vectors is really just the sum of two $\frac{n}{2}$-dimensional inner products, the following recursion holds, for the partitions in (15):

$$\mathbf{AB} = \begin{pmatrix} \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} & \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} & \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} \end{pmatrix}. \tag{16}$$

The recursion (16) reduces the problem of multiplying $n \times n$ matrices to eight $\frac{n}{2} \times \frac{n}{2}$ matrix multiplications, and four matrix additions. Adding $O(n) \times O(n)$ matrices has the same complexity as adding $O(n^2)$-sized vectors, so all additions take time $O(n^2)$.

In conclusion, letting $\mathcal{T}(n)$ denote the time it takes to multiply two $n \times n$ matrices, (16) shows

$$\mathcal{T}(n) \le 8\mathcal{T}\left(\frac{n}{2}\right) + O(n^2).$$

Noting that this is a leaves-heavy recurrence, Theorem 1 gives $\mathcal{T}(n) = O(n^{\log_2(8)}) = O(n^3)$. So, this recursive viewpoint did not save us any computation over the straightforward algorithm. This should be unsurprising; the recursion is essentially evaluating the same sums (14) as before, but just split up across different subproblems. This situation is reminiscent of how the straightforward recursion for scalar multiplication does not yield speedups in Section 2.

Strassen's main observation was that we can write a recursive formula for $\mathbf{AB}$ as in (15), but which uses merely seven $\frac{n}{2} \times \frac{n}{2}$ multiplications, and a constant number of additions. Let us note that this already yields an improved runtime. Indeed, the resulting recurrence relationship is

$$\mathcal{T}(n) \le 7\mathcal{T}\left(\frac{n}{2}\right) + O(n^2),$$

and Theorem 1 shows that this solves to $\mathcal{T}(n) = O(n^{\log_2(7)}) \approx O(n^{2.807})$.

Strassen's actual recursive formula is very complicated, so for brevity, we do not reproduce it here: a full description is in Section 4.2 of [CLRS22]. It involves adding $\frac{n}{2} \times \frac{n}{2}$ matrices ten times, multiplying them seven times, and then finally adding them twelve more times. We do not claim to have much intuition on how Strassen came up with this formula.

A long line of work built upon Strassen's algorithm to give faster matrix multiplication algorithms: the current (peer-reviewed) world record as of the writing these notes is $\mathcal{T}(n) = O(n^{2.3714})$ [ADV+25]. These algorithms are commonly cited as examples of "galactic algorithms" (see Section 4), and are quite sophisticated. They are based on bounds on another complicated algebraic problem (the "border rank of the Coppersmith-Winograd tensor") containing matrix multiplication as a subproblem. These methods, as currently developed, are not very practical. There is also evidence that all known techniques cannot obtain runtimes beyond $\approx n^{2.168}$ [Alm19].

However, there are also no known lower bounds other than the $\Omega(n^2)$ time needed to write down the $n \times n$ output, even if computing it was free. Characterizing the complexity of matrix multiplication is one of the most interesting and potentially impactful open problems in algorithm design today. As we will see in the following Section 6.2, improvements for fundamental problems can often come from unexpected places: when trying to design faster algorithms, you can try anything!

6.2 Fast Fourier transform

In this section, we explain the *fast Fourier transform* (FFT).[5] The FFT is simply a fast algorithm for solving the discrete Fourier transform (DFT) problem, which asks to multiply a given complex vector $\mathbf{v} \in \mathbb{C}^n$ by a certain recursively-defined "DFT matrix" $\mathbf{F}_n \in \mathbb{C}^{n \times n}$. We denote this problem by $\mathsf{DFT}_n(\mathbf{v})$: in other words, DFT_n is an algorithm which on input $\mathbf{v} \in \mathbb{C}^n$, returns the matrix-vector product $\mathbf{F}_n\mathbf{v}$. For simplicity, we only explain the case when n is a power of 2.

Why is DFT_n interesting? The problem involves multiplication by an $n \times n$ matrix with $O(n^2)$ numbers in its representation, so it is natural to conjecture $O(n^2)$ is the fastest possible runtime. Indeed, the straightforward algorithm which writes down $\mathbf{F}_n$ achieves this runtime. However, it turns out that we can use the fast Fourier transform to solve this problem in time $O(n\log(n))$. That is, we can essentially directly write down the output (with a logarithmic factor overhead in the runtime) without actually bothering to write down the whole matrix $\mathbf{F}_n$.

This fact alone has a number of very exciting implications. For now, let us first define the problem and explaining its efficient solution. Let $\iota := \sqrt{-1}$ denote the imaginary unit, and let n be a power of two. Further, let $\omega_n := \exp(\frac{2\pi\iota}{n}) = \cos(\frac{2\pi}{n}) + \iota\sin(\frac{2\pi}{n})$ denote the n^{th} *root of unity*. Recall that $\omega_n^n = 1 = \cos(0) + \iota\sin(0)$, and that ω_n can intuitively be seen as a rotation of angle $\frac{2\pi}{n}$ from 1 along the unit circle in $\mathbb{C}$. In fact, the powers $\{\omega_n^j\}_{j=0}^{n-1}$ are the various n^{th} roots of 1.

[5] We follow the excellent presentation of [O'D20]. This whole lecture series is great.

The DFT matrix $\mathbf{F}_n$ has $(i,j)^{\text{th}}$ entry $\omega_n^{(i-1)(j-1)}$, for all $(i,j) \in [n] \times [n]$. We give some examples, remembering that $\omega_n^n = 1$ for all n, so we simplify exponents accordingly:

$$\mathbf{F}_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix},\ \mathbf{F}_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \iota & -1 & -\iota \\ 1 & -1 & 1 & -1 \\ 1 & -\iota & -1 & \iota \end{pmatrix},\ \mathbf{F}_8 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega & \omega^6 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega \end{pmatrix},$$

where we let $\omega := \omega_8$ for readability in the last example. This might be a lot to take in at first, but if you stare at $\mathbf{F}_8$, all sorts of patterns start to appear. Take every odd column of $\mathbf{F}_8$, i.e., every other starting from the first. Somewhat magically, we get two copies of $\mathbf{F}_4$:

$$\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^6 & \omega^4 & \omega^2 \end{pmatrix} = \begin{pmatrix} \mathbf{F}_4 \\ \mathbf{F}_4 \end{pmatrix}. \tag{17}$$

Similarly, taking every even column, i.e., every other starting from the second, gives:

$$\begin{aligned} \begin{pmatrix} 1 & 1 & 1 & 1 \\ \omega & \omega^3 & \omega^5 & \omega^7 \\ \omega^2 & \omega^6 & \omega^2 & \omega^6 \\ \omega^3 & \omega & \omega^7 & \omega^5 \\ \omega^4 & \omega^4 & \omega^4 & \omega^4 \\ \omega^5 & \omega^7 & \omega & \omega^3 \\ \omega^6 & \omega^2 & \omega^6 & \omega^2 \\ \omega^7 & \omega^5 & \omega^3 & \omega \end{pmatrix} &= \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \omega & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \omega^2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \omega^3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \omega^4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \omega^5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \omega^6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \omega^7 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & 1 & 1 & 1 \\ 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^6 & \omega^4 & \omega^2 \end{pmatrix} \\ &= \mathbf{diag}\left(\left\{\omega^i\right\}_{i=0}^{7}\right) \begin{pmatrix} \mathbf{F}_4 \\ \mathbf{F}_4 \end{pmatrix}. \end{aligned} \tag{18}$$

Here we noticed that left-multiplication by a diagonal matrix is the same as multiplying each row by the diagonal elements, so the even columns are just a "twist" of the odd columns. This pattern straightforwardly generalizes to any n a power of two: $\mathbf{F}_n$'s odd and even columns are

$$\begin{pmatrix} \mathbf{F}_{\frac{n}{2}} \\ \mathbf{F}_{\frac{n}{2}} \end{pmatrix},\ \mathbf{diag}\left(\left\{\omega_n^i\right\}_{i=0}^{n-1}\right) \begin{pmatrix} \mathbf{F}_{\frac{n}{2}} \\ \mathbf{F}_{\frac{n}{2}} \end{pmatrix}. \tag{19}$$

Next, consider the following interpretation of $\mathbf{A}\mathbf{v}$, for $\mathbf{A} \in \mathbb{C}^{n \times n}$ and $\mathbf{v} \in \mathbb{C}^n$. The matrix-vector product is a linear combination of $\{\mathbf{A}_{:j}\}_{j \in [n]}$, the columns of $\mathbf{A}$, with coefficients given by $\mathbf{v}$:

$$\mathbf{A}\mathbf{v} = \mathbf{v}_1 \mathbf{A}_{:1} + \mathbf{v}_2 \mathbf{A}_{:2} + \ldots + \mathbf{v}_n \mathbf{A}_{:n}.$$

Hence, a matrix-vector multiplication of $\mathbf{v}$ with $\mathbf{A}$ is just a multiplication of $\mathbf{v}$'s odd entries with $\mathbf{A}$'s odd columns, summed with a multiplication of $\mathbf{v}$'s even entries with $\mathbf{A}$'s even columns.

Finally, let us return to implementing DFT. For an input $\mathbf{v} \in \mathbb{C}^n$, let $\mathbf{v}_{\text{odd}} \in \mathbb{C}^{\frac{n}{2}}$ and $\mathbf{v}_{\text{even}} \in \mathbb{C}^{\frac{n}{2}}$ be the odd and even coordinates of $\mathbf{v}$, concatenated in order. We have shown from (19), our characterizations of the odd and even columns of $\mathbf{F}_n$, that the following recurrence holds:

$$\mathbf{F}_n \mathbf{v} = \underbrace{\begin{pmatrix} \mathbf{F}_{\frac{n}{2}} \\ \mathbf{F}_{\frac{n}{2}} \end{pmatrix} \mathbf{v}_{\text{odd}}}_{:=\mathbf{s}_{\text{odd}}} + \mathbf{diag}\left(\left\{\omega_n^i\right\}_{i=0}^{n-1}\right) \underbrace{\begin{pmatrix} \mathbf{F}_{\frac{n}{2}} \\ \mathbf{F}_{\frac{n}{2}} \end{pmatrix} \mathbf{v}_{\text{even}}}_{:=\mathbf{s}_{\text{even}}}. \tag{20}$$

Algorithm 7: $\mathsf{DFT}(n, \mathbf{v})$

```
1 Input: $n \in \mathbb{N}$ a power of two, $\mathbf{v} \in \mathbb{C}^n$
2 if $n == 1$ then
3     return $\mathbf{v}$
4 end
5 $\mathbf{v}_{\text{odd}}, \mathbf{v}_{\text{even}} \leftarrow$ vectors in $\mathbb{C}^{\frac{n}{2}}$ concatenating the odd and even coordinates of $\mathbf{v}$ in order,
  respectively
6 $\mathbf{a}_{\text{odd}}, \mathbf{a}_{\text{even}} \leftarrow \mathsf{DFT}(\frac{n}{2}, \mathbf{v}_{\text{odd}}), \mathsf{DFT}(\frac{n}{2}, \mathbf{v}_{\text{even}})$
7 $\mathbf{s}_{\text{odd}}, \mathbf{s}_{\text{even}} \leftarrow \begin{pmatrix} \mathbf{a}_{\text{odd}} \\ \mathbf{a}_{\text{odd}} \end{pmatrix}, \begin{pmatrix} \mathbf{a}_{\text{even}} \\ \mathbf{a}_{\text{even}} \end{pmatrix}$
8 $\widetilde{\mathbf{s}}_{\text{even}} \leftarrow$ vector in $\mathbb{C}^n$ with $[\widetilde{\mathbf{s}}_{\text{even}}]_{j+1} = [\mathbf{s}_{\text{even}}]_{j+1} \cdot \omega_n^j$ for all $0 \le j \le n-1$
9 return $\mathbf{s}_{\text{odd}} + \widetilde{\mathbf{s}}_{\text{even}}$
```

In other words, we can compute $\mathbf{F}_n\mathbf{v} = \mathsf{DFT}_n(\mathbf{v})$ in the following recursive way. To make the pseudocode cleaner, we let n be formally passed in as an argument to the algorithm.

It is straightforward to check that all steps of Algorithm 7 other than Line 6 take $O(n)$ arithmetic operations. Moreover, Line 6 requires two calls to $\mathsf{DFT}_{\frac{n}{2}}$. Therefore, letting $\mathcal{T}(n)$ denote the number of operations required by DFT_n, our implementation satisfies

$$\mathcal{T}(n) \le 2\mathcal{T}\left(\frac{n}{2}\right) + O(n).$$

Now, we can apply Theorem 1 to conclude we are in the balanced case, and the recurrence solves to $\mathcal{T}(n) = O(n \log(n))$, the claimed runtime of DFT_n.

Why is it important that we can solve DFT_n quickly? To explain this, we need to first explain two properties of the DFT matrix $\mathbf{F}_n$: that it is a *unitary matrix*, as well as a *Vandermonde matrix*.

The DFT matrix is unitary. A complex matrix $\mathbf{U} \in \mathbb{C}^{n\times n}$ is called *unitary* if its columns form an orthonormal basis of $\mathbb{C}^n$. For our purposes, this just means $\mathbf{U}$'s inverse $\mathbf{U}^{-1}$ has a simple expression: $\mathbf{U}^{-1} = \mathbf{U}^\dagger$, where † first transposes the matrix, and then takes the elementwise complex conjugate. We interpret this in the real setting as the columns $\{\mathbf{U}_{:i}\}_{i\in[n]}$ acting like orthogonal unit vectors in Section 5.2, Part I, i.e., $\langle \mathbf{U}_{:i}, \mathbf{U}_{:j} \rangle = 1$ iff $i = j$, and $= 0$ otherwise.[6]

The punchline is that $\frac{1}{\sqrt{n}}\mathbf{F}_n$ is unitary. This means the following equation holds:

$$\left(\frac{1}{\sqrt{n}}\mathbf{F}_n\right)^\dagger \left(\frac{1}{\sqrt{n}}\mathbf{F}_n\right) = \frac{1}{n}\mathbf{F}_n^\dagger \mathbf{F}_n = \mathbf{I}_n \implies \mathbf{F}_n^{-1} = \frac{1}{n}\mathbf{F}_n^\dagger. \tag{21}$$

It is a nice geometric exercise to visualize why this is true. The proof follows from the equation

$$\frac{1}{n}\sum_{k=0}^{n-1} \omega^{ik}\overline{\omega^{jk}} = \frac{1}{n}\sum_{k=0}^{n-1} \omega^{(i-j)k} = \begin{cases} 1 & i = j \\ 0 & i \ne j \end{cases} \text{ for any } 0 \le i \le n-1,\ 0 \le j \le n-1. \tag{22}$$

The left-hand side above is the inner product of the i^{th} and j^{th} columns of $\frac{1}{\sqrt{n}}\mathbf{F}_n$, where we abbreviated $\omega := \omega_n$, and recalled $\overline{\omega} = \omega^{-1}$. Intuitively, if $i = j$, then the expression (22) averages n copies of 1, and if not, it averages n evenly spaced points around the unit circle.

Now, observe that because $\mathbf{F}_n$ is symmetric, $\mathbf{F}_n^\dagger$ is just the elementwise conjugate of $\mathbf{F}_n$, and its rows are a permutation of $\mathbf{F}_n$'s rows. In fact, the rows of $\mathbf{F}_n^\dagger$ are just the rows of $\mathbf{F}_n$ in reverse

[6] For reasons involving the identity $(a + b\imath)\overline{(a + b\imath)} = a^2 + b^2$, we recall that we have to conjugate one vector when computing inner products in $\mathbb{C}^n$. So when we write $\langle \mathbf{u}, \mathbf{v} \rangle$ for $\mathbf{u}, \mathbf{v} \in \mathbb{C}^n$, we really mean $\sum_{i\in[n]} \mathbf{u}_i \overline{\mathbf{v}_i}$.

order, except for the all-ones row. To see this, it is easiest to use $\mathbf{F}_8$ as an example:

$$\mathbf{F}_8 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega^3 & \omega^6 & \omega & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega & \omega^6 & \omega^3 \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega \end{pmatrix}, \; \mathbf{F}_8^\dagger = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega^7 & \omega^6 & \omega^5 & \omega^4 & \omega^3 & \omega^2 & \omega \\ 1 & \omega^6 & \omega^4 & \omega^2 & 1 & \omega^6 & \omega^4 & \omega^2 \\ 1 & \omega^5 & \omega^2 & \omega^7 & \omega^4 & \omega & \omega^6 & \omega^3 \\ 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 & 1 & \omega^4 \\ 1 & \omega^3 & \omega^6 & \omega & \omega^4 & \omega^7 & \omega^2 & \omega^5 \\ 1 & \omega^2 & \omega^4 & \omega^6 & 1 & \omega^2 & \omega^4 & \omega^6 \\ 1 & \omega & \omega^2 & \omega^3 & \omega^4 & \omega^5 & \omega^6 & \omega^7 \end{pmatrix}.$$

Therefore, multiplication with $\mathbf{F}_n^\dagger$ takes the same asymptotic time as multiplication with $\mathbf{F}_n$, i.e., $O(n\log(n))$ via DFT_n, because to evaluate $\mathbf{F}_n^\dagger \mathbf{v}$ we can just permute the entries of $\mathbf{F}_n \mathbf{v}$.

Let DFTInv_n denote an algorithm which on input $\mathbf{v} \in \mathbb{C}^n$ returns the matrix-vector product $\mathbf{F}_n^{-1}\mathbf{v}$. By using the identity (21) and the special structure of $\mathbf{F}_n^\dagger$, we have shown that we can implement DFTInv_n in $O(n\log(n))$ time. We provide pseudocode below in Algorithm 8.

Algorithm 8: $\mathsf{DFTInv}(n, \mathbf{v})$

```
Input: $n \in \mathbb{N}$ a power of two, $\mathbf{v} \in \mathbb{C}^n$
$\mathbf{u} \leftarrow \mathsf{DFT}(n, \mathbf{v})$
$\widetilde{\mathbf{u}} \leftarrow$ vector in $\mathbb{C}^n$ with $\widetilde{\mathbf{u}}_1 = \mathbf{u}_1$ and $\widetilde{\mathbf{u}}_j = \mathbf{u}_{n+2-j}$ for all $2 \le j \le n$
return $\frac{1}{n}\widetilde{\mathbf{u}}$
```

The DFT matrix is Vandermonde. The other important property of $\mathbf{F}_n$ is that it can be viewed as evaluating degree-n polynomials. Let $\{a_i\}_{i=0}^{n-1}$ be the coefficients of a degree-n polynomial $p_{\mathbf{a}}$, where $\mathbf{a} \in \mathbb{C}^n$ concatenates the coefficients as $\mathbf{a}_i = a_{i-1}$ for all $i \in [n]$:

$$p_{\mathbf{a}}(x) = a_0 + a_1 x + a_2 x^2 + a_3 x^3 + \ldots + a_{n-1}x^{n-1} = \sum_{i=0}^{n-1} a_i x^i. \tag{23}$$

Notice that for all $0 \le j \le n-1$, the $(j+1)^{\text{th}}$ element of $\mathbf{F}_n\mathbf{a}$ evaluates $p_{\mathbf{a}}$ at $x = \omega^j$:

$$[\mathbf{F}_n]_{(j+1):}^\top \mathbf{a} = \sum_{i=0}^{n-1} a_i \left(\omega^j\right)^i = p_{\mathbf{a}}\left(\omega^j\right).$$

Therefore, elements of $\mathbf{F}_n\mathbf{a}$ just correspond to different polynomial evaluations. Succinctly,

$$\mathbf{F}_n\mathbf{a} = \left\{p_{\mathbf{a}}\left(\omega^j\right)\right\}_{j=0}^{n-1}. \tag{24}$$

Matrices for which this expression is true, for some set of evaluation points, are called *Vandermonde*.

Just as applying $\mathbf{F}_n$ to a coefficient vector $\mathbf{a}$ evaluates the corresponding polynomial $p_{\mathbf{a}}$ at the points $\{\omega^j\}_{j=0}^{n-1}$, we can take values $\mathbf{v} = \{p_{\mathbf{a}}(\omega^j)\}_{j=0}^{n-1}$ and recover the coefficients $\mathbf{a}$ using $\mathbf{F}_n^{-1}$:

$$\mathbf{F}_n^{-1}\mathbf{v} = \mathbf{a}, \text{ where } p_{\mathbf{a}}\left(\omega^j\right) = \mathbf{v}_{j+1} \text{ for all } 0 \le j \le n-1. \tag{25}$$

This is a basic example of *polynomial interpolation*, where we reconstruct the coefficients of a polynomial, given its values at a fixed set of points. Just as two points determine a line, three determine a quadratic, etc., the n evaluations of a degree-$(n-1)$ polynomial at $\{\omega^j\}_{j=0}^{n-1}$ uniquely determine the polynomial, whose coefficients are recoverable via (25).

Faster integer multiplication. The reason why this is exciting is because integer multiplication in the sense of Section 2 is just polynomial multiplication. We can represent two m-digit numbers $a = \{a_i\}_{i=0}^{m-1}$ and $b = \{b_i\}_{i=0}^{m-1}$ in base 10 as polynomial evaluations at $x = 10$:

$$a = \sum_{i=0}^{m-1} a_i 10^i, \; b = \sum_{i=0}^{m-1} b_i 10^i.$$

So, if we equate polynomials $p_{\mathbf{a}} \equiv a$ and $p_{\mathbf{b}} \equiv b$ in the canonical way, we can compute ab via

$$ab = \left(\sum_{i=0}^{m-1} a_i 10^i\right)\left(\sum_{i=0}^{m-1} b_i 10^i\right) = p_{\mathbf{a}}(10)p_{\mathbf{b}}(10) = (p_{\mathbf{a}}p_{\mathbf{b}})(10).$$

Here, $p_{\mathbf{a}}p_{\mathbf{b}}$ is the degree-$2(m-1)$ polynomial which satisfies $(p_{\mathbf{a}}p_{\mathbf{b}})(x) = p_{\mathbf{a}}(x)p_{\mathbf{b}}(x)$ for all $x \in \mathbb{C}$. We can directly compute the coefficients of $p_{\mathbf{a}}p_{\mathbf{b}}$ in $O(m^2)$ time, using an expansion similar to (1). However, we can actually compute them in only $O(m\log(m))$ time, via the fast Fourier transform. Notice that if we had the coefficients of $p_{\mathbf{a}}p_{\mathbf{b}}$, we could just directly evaluate $(p_{\mathbf{a}}p_{\mathbf{b}})(10)$ to obtain the product in $O(m)$ time, so computing coefficients in $O(m\log(m))$ time is the bottleneck step.

To see how to do this, let n be a power of two with $2m \le n < 4m$. This way, ab has at most n digits. By appropriately padding with zeroes, a and b can be viewed as coefficient vectors $\mathbf{a}, \mathbf{b} \in \mathbb{C}^n$ for degree-$(m-1)$ polynomials, so $a = p_{\mathbf{a}}(10)$, $b = p_{\mathbf{b}}(10)$, and we wish to compute ab.

We can evaluate the coefficients of the polynomial $p_{\mathbf{a}}p_{\mathbf{b}}$ in $O(m\log(m))$ time as follows.

1. Recalling the expression (24), we can compute all of the values $\mathbf{F}_n\mathbf{a} = \{p_{\mathbf{a}}(\omega^j)\}_{j=0}^{n-1}$ and $\mathbf{F}_n\mathbf{b} = \{p_{\mathbf{b}}(\omega^j)\}_{j=0}^{n-1}$ in time $O(n\log(n))$, using DFT_n twice.
2. By taking the products of these lists, this gives $\{(p_{\mathbf{a}}p_{\mathbf{b}})(\omega^j)\}_{j=0}^{n-1}$ in $O(n)$ additional time.
3. Since $p_{\mathbf{a}}p_{\mathbf{b}}$ is a degree-$(n-1)$ polynomial (padding with zeroes as necessary), and we have n evaluations of it, (25) recovers its coefficients $\mathbf{c} \in \mathbb{C}^n$, in time $O(n\log(n))$ using DFTInv_n.

Now, given the coefficients $\mathbf{c}$, we can evaluate $p_{\mathbf{c}}(10) = p_{\mathbf{a}}(10)p_{\mathbf{b}}(10)$ in time $O(n)$. The dominant steps are the applications of DFT_n to evaluate polynomials, and DFTInv_n to recover the coefficients of the product polynomial, given n evaluations. The overall runtime is $O(n\log(n)) = O(m\log(m))$, as claimed. This is a faster runtime for integer multiplication compared to Section 2.

We swept an important detail under the rug: justifying that adding and multiplying two numbers in $\mathbb{C}$ can be done in $O(1)$ time. Even in the word RAM model, we can only hope to do this to finite precision (i.e., a few bits). Fortunately, we can adapt the ideas above to integer arithmetic, by treating ω_n as an integer "n^{th} root of unity" satisfying $\omega_n^n \equiv 1$ in appropriate modular arithmetic. This strategy was formalized by Schönhage and Strassen [SS71], who gave an integer multiplication algorithm using $O(n\log(n)\log\log(n))$ bit operations; this was improved to $O(n\log(n))$ by recent work of [HvdH21], but the latter algorithm has a much larger leading constant.

In practice, the fast Fourier transform is used for all sorts of applications in signal processing, optics, compression, and more. Regarding integer multiplication specifically, Python uses a mix of the grade-school algorithm (1) and Karatsuba's algorithm, depending on the size of the input. However, this is likely because most common uses of multiplication are on relatively small integers. The Schönhage-Strassen FFT has highly-optimized implementations in free GNU libraries (GMP), for instance, and is preferred in practice for large integers. For applications in cryptography which repeatedly multiply large numbers, FFT-based algorithms are your best bet.

6.3 Fibonacci numbers

We give another application of matrix multiplication, to solving *recurrence relations*. Consider $\mathsf{Fib}(n)$, the problem of outputting the n^{th} Fibonacci number F_n, defined via

$$F_0 = 1,\ F_1 = 2,\ F_n = F_{n-1} + F_{n-2},\ \text{for all } n \ge 2. \tag{26}$$

The straightforward implementation of $\mathsf{Fib}(n)$ requires n integer additions: we can always keep variables storing the values F_{i-2} and F_{i-1}, and add these to obtain F_i, in a for loop over $2 \le i \le n$.[7]

We can obtain signifacnt speedups using matrix multiplication. Note that

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x+y \\ x \end{pmatrix} \implies \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}\begin{pmatrix} F_{n-1} \\ F_{n-2} \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix},\ \text{for all } n \ge 2. \tag{27}$$

[7] There is actually an even more basic way to implement $\mathsf{Fib}(n)$, which directly calls $\mathsf{Fib}(n-1)$ and $\mathsf{Fib}(n-2)$ as subroutines. However, this leads to an exponential blowup in the number of recursive calls to Fib, because e.g., $\mathsf{Fib}(n-1)$ leads to another call to $\mathsf{Fib}(n-2)$, and both call $\mathsf{Fib}(n-3)$, etc. The implementation we describe makes sure that as soon as we figure out $\mathsf{Fib}(i)$ for $i \le n$, we never compute it again: it is a "bottom-up" recursion rather than a "top-down" one. This is our first example of *dynamic programming*, which we revisit in Part III of the notes.

Therefore, it inductively holds that for all $n \geq 1$,

$$\mathbf{A}^n \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} F_n \\ F_{n-1} \end{pmatrix}, \text{ for } \mathbf{A} := \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, \tag{28}$$

since we can verify the base case $n = 1$ via (26), and the inductive step follows using (27). So, if we want to compute F_n, it is enough to compute $\mathbf{A}^n$ and take its top-left element.

We now give a useful trick for computing $\mathbf{A}^n$ using only $O(\log(n))$ multiplications of 2×2 matrices. Let $p = 2^k$ be the largest power of two which is $\leq n$, i.e., $p \leq n < 2p$. We can compute

$$\mathbf{A},\ \mathbf{A}^2,\ \mathbf{A}^4,\ \ldots,\ \mathbf{A}^{2^{k-1}},\ \mathbf{A}^{2^k}, \tag{29}$$

using only $O(k) = O(\log(n))$ matrix multiplications, since each $\mathbf{A}^{2^i}$ is just the square of $\mathbf{A}^{2^{i-1}}$. Next, we can write $\mathbf{A}^n$ as a product of at most $O(\log(n))$ of the matrices in (29), using the binary representation of n. For example, if $n = 23$,

$$\mathbf{A}^{23} = \mathbf{A}^{16} \cdot \mathbf{A}^4 \cdot \mathbf{A}^2 \cdot \mathbf{A}^1 = \mathbf{A}^{2^4} \cdot \mathbf{A}^{2^2} \cdot \mathbf{A}^{2^1} \cdot \mathbf{A}^{2^0}.$$

We have shown that F_n is computable using $O(\log(n))$ matrix multiplications, each involving $O(1)$ integer operations (addition and multiplication), because the matrices involved are 2×2 (they are all powers of $\mathbf{A} \in \mathbb{R}^{2\times 2}$). Hence, it seems that we have obtained an *exponential improvement* over the straightforward algorithm, which used $O(n)$ integer additions. Indeed, repeated squaring implements (28) in just $O(\log(n))$ arithmetic operations.

Unfortunately, the devil is in the details: while (28) does imply a faster implementation of $\mathsf{Fib}(n)$ than the straightforward method, it is actually only a *polynomial improvement* once we account for the size of the integers we need to manipulate. Recall from Lemma 14, Part I, that F_n grows exponentially in n. Therefore, we require $\Theta(n)$ digits to represent the integer F_n. It is thus unreasonable to expect that we can add e.g., F_{n-1} and F_{n-2} in $O(\log(n))$ time.

With this caveat in mind, let us re-examine the complexity of both the straightforward implementation of $\mathsf{Fib}(n)$, and the implementation via (28) and repeated squaring. Specifically, we bound the number of bit operations used by both methods. The straightforward method requires $O(n^2)$ bit operations, as it uses $O(n)$ integer additions, and each integer is expressible using $O(n)$ bits.

On the other hand, the bottleneck operation for the implementation via (28) is computing all of the powers (29). Indeed, all further matrix multiplications are no more expensive than evaluating the largest matrix power $\mathbf{A}^{2^k} = \mathbf{A}^{2^{k-1}} \cdot \mathbf{A}^{2^{k-1}}$. Let $\mathcal{T}(2^i)$ denote the cost of computing the matrix power $\mathbf{A}^{2^i}$, in bit operations. We have the recurrence

$$\mathcal{T}(2^i) = \mathcal{T}(2^{i-1}) + O(2^i \log(2^i)), \tag{30}$$

since we need to compute $\mathbf{A}^{2^{i-1}}$, and then multiply or add $O(2^i)$-digit integers a few times to square $\mathbf{A}^{2^{i-1}}$. As discussed in Section 6.2, this costs $O(2^i \log(2^i))$ bit operations [HvdH21].

Reparameterizing the recurrence (30) as $\mathcal{T}(n) = \mathcal{T}(\frac{n}{2}) + O(n \log(n))$, Theorem 1 shows that $\mathcal{T}(n) = O(n \log(n))$, which is also the asymptotic cost of the overall method. Thus, while not as dramatic as we originally hoped, the matrix multiplication-based approach to implementing $\mathsf{Fib}(n)$ still represents a nearly-quadratic improvement over the straightforward approach.

Further reading

For more on Section 2, see Chapter 1.9, [Eri24] or Chapter 5.5, [KT05] or Chapter 1, [Rou22].

For more on Section 3, see Chapters 4.3 to 4.7 of [CLRS22] or Chapter 1.7, [Eri24] or Chapter 5.2, [KT05] or Chapter 4, [Rou22].

For more on Section 5.2, see Chapter 5.3, [KT05] or Chapter 2, [Rou22].

For more on Section 5.3, see Chapter 9, [CLRS22] or Chapter 1.8, [Eri24] or Chapter 6, [Rou22].

For more on Section 6.1, see Chapter 4.2, [CLRS22] or Chapter 3.3, [Rou22].

For more on Section 6.2, see Chapter A, [Eri24] or Chapter 5.6, [KT05].

For more on Section 6.3, see Chapter 3.2, [Eri24].

References

[ADV+25] Josh Alman, Ran Duan, Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. More asymmetry yields faster matrix multiplication. In *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2025*, pages 2005–2039. SIAM, 2025.

[Alm19] Josh Alman. Limits on the universal method for matrix multiplication. In *34th Computational Complexity Conference, CCC 2019*, volume 137 of *LIPIcs*, pages 12:1–12:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[BFP+73] Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time bounds for selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973.

[CLRS22] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Fourth Edition.* The MIT Press, 2022.

[Eri24] Jeff Erickson. *Algorithms.* 2024.

[HvdH21] David Harvey and Joris van der Hoeven. Integer multiplication in time $o(n \log n)$. *Annals of Mathematics*, 193(2):563–617, 2021.

[KO62] A. Karatsuba and Yu. Ofman. Multiplication of many-digital numbers by automatic computers. *Proceedings of the USSR Academy of Sciences*, 145:293–294, 1962.

[KT05] Jon Kleinberg and Éva Tardos. *Algorithm Design.* 2005.

[O'D20] Ryan O'Donnell. Lecture 7c: Fast fourier transform (fft). cmu 15-751: Cs theory toolkit. https://www.youtube.com/watch?v=j77uFXx8tAQ, 2020. Accessed: 2024-08-17.

[Rou22] Tim Roughgarden. *Algorithms Illuminated.* Soundlikeyourself Publishing, 2022.

[SS71] Arnold Schönhage and Volker Strassen. Schnelle multiplikation großer zahlen. *Computing*, 7(3–4):281–292, 1971.

[Str69] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, 1969.

[Wik24] Wikipedia. Galactic algorithm. https://en.wikipedia.org/wiki/Galactic_algorithm, 2024. Accessed: 2024-08-17.