Chapter 2

Divide-and-conquer algorithms

The *divide-and-conquer* strategy solves a problem by:

1. Breaking it into *subproblems* that are themselves smaller instances of the same type of problem
2. Recursively solving these subproblems
3. Appropriately combining their answers

The real work is done piecemeal, in three different places: in the partitioning of problems into subproblems; at the very tail end of the recursion, when the subproblems are so small that they are solved outright; and in the gluing together of partial answers. These are held together and coordinated by the algorithm's core recursive structure.

As an introductory example, we'll see how this technique yields a new algorithm for multiplying numbers, one that is much more efficient than the method we all learned in elementary school!

2.1 Multiplication

The mathematician Carl Friedrich Gauss (1777–1855) once noticed that although the product of two complex numbers

$$(a+bi)(c+di) \quad = \quad ac - bd + (bc+ad)i$$

seems to involve *four* real-number multiplications, it can in fact be done with just *three*: ac, bd, and $(a+b)(c+d)$, since

$$bc + ad \quad = \quad (a+b)(c+d) - ac - bd.$$

In our big-O way of thinking, reducing the number of multiplications from four to three seems wasted ingenuity. But this modest improvement becomes very significant *when applied recursively*.

Let's move away from complex numbers and see how this helps with regular multiplication. Suppose x and y are two n-bit integers, and assume for convenience that n is a power of 2 (the more general case is hardly any different). As a first step toward multiplying x and y, split each of them into their left and right halves, which are $n/2$ bits long:

$$x = \boxed{\quad x_L \quad}\,\boxed{\quad x_R \quad} = 2^{n/2}x_L + x_R$$
$$y = \boxed{\quad y_L \quad}\,\boxed{\quad y_R \quad} = 2^{n/2}y_L + y_R.$$

For instance, if $x = 10110110_2$ (the subscript 2 means "binary") then $x_L = 1011_2$, $x_R = 0110_2$, and $x = 1011_2 \times 2^4 + 0110_2$. The product of x and y can then be rewritten as

$$xy \;=\; (2^{n/2}x_L + x_R)(2^{n/2}y_L + y_R) \;=\; 2^n\, x_L y_L \;+\; 2^{n/2}\,(x_L y_R + x_R y_L) \;+\; x_R y_R.$$

We will compute xy via the expression on the right. The additions take linear time, as do the multiplications by powers of 2 (which are merely left-shifts). The significant operations are the four $n/2$-bit multiplications, $x_L y_L,\ x_L y_R,\ x_R y_L,\ x_R y_R$; these we can handle by four recursive calls. Thus our method for multiplying n-bit numbers starts by making recursive calls to multiply these four pairs of $n/2$-bit numbers (four subproblems of half the size), and then evaluates the preceding expression in $O(n)$ time. Writing $T(n)$ for the overall running time on n-bit inputs, we get the *recurrence relation*

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

We will soon see general strategies for solving such equations. In the meantime, this particular one works out to $O(n^2)$, the same running time as the traditional grade-school multiplication technique. So we have a radically new algorithm, but we haven't yet made any progress in efficiency. How can our method be sped up?

This is where Gauss's trick comes to mind. Although the expression for xy seems to demand four $n/2$-bit multiplications, as before just three will do: $x_L y_L, x_R y_R$, and $(x_L + x_R)(y_L + y_R)$, since $x_L y_R + x_R y_L = (x_L + x_R)(y_L + y_R) - x_L y_L - x_R y_R$. The resulting algorithm, shown in Figure 2.1, has an improved running time of[1]

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

The point is that now the constant factor improvement, from 4 to 3, occurs *at every level of the recursion*, and this compounding effect leads to a dramatically lower time bound of $O(n^{1.59})$.

This running time can be derived by looking at the algorithm's pattern of recursive calls, which form a tree structure, as in Figure 2.2. Let's try to understand the shape of this tree. At each successive level of recursion the subproblems get halved in size. At the $(\log_2 n)^{\text{th}}$ level,

[1] Actually, the recurrence should read

$$T(n) \;\le\; 3T(n/2+1) + O(n)$$

since the numbers $(x_L + x_R)$ and $(y_L + y_R)$ could be $n/2 + 1$ bits long. The one we're using is simpler to deal with and can be seen to imply exactly the same big-O running time.

Figure 2.1 A divide-and-conquer algorithm for integer multiplication.

```
function multiply($x, y$)
Input:  Positive integers $x$ and $y$, in binary
Output:  Their product

$n =$ max(size of $x$, size of $y$)
if $n = 1$:  return $xy$

$x_L$, $x_R =$ leftmost $\lceil n/2 \rceil$, rightmost $\lfloor n/2 \rfloor$ bits of $x$
$y_L$, $y_R =$ leftmost $\lceil n/2 \rceil$, rightmost $\lfloor n/2 \rfloor$ bits of $y$

$P_1 =$ multiply($x_L, y_L$)
$P_2 =$ multiply($x_R, y_R$)
$P_3 =$ multiply($x_L + x_R, y_L + y_R$)
return $P_1 \times 2^n + (P_3 - P_1 - P_2) \times 2^{n/2} + P_2$
```

the subproblems get down to size 1, and so the recursion ends. Therefore, the height of the tree is $\log_2 n$. The branching factor is 3—each problem recursively produces three smaller ones—with the result that at depth k in the tree there are 3^k subproblems, each of size $n/2^k$.

For each subproblem, a linear amount of work is done in identifying further subproblems and combining their answers. Therefore the total time spent at depth k in the tree is

$$3^k \times O\left(\frac{n}{2^k}\right) = \left(\frac{3}{2}\right)^k \times O(n).$$

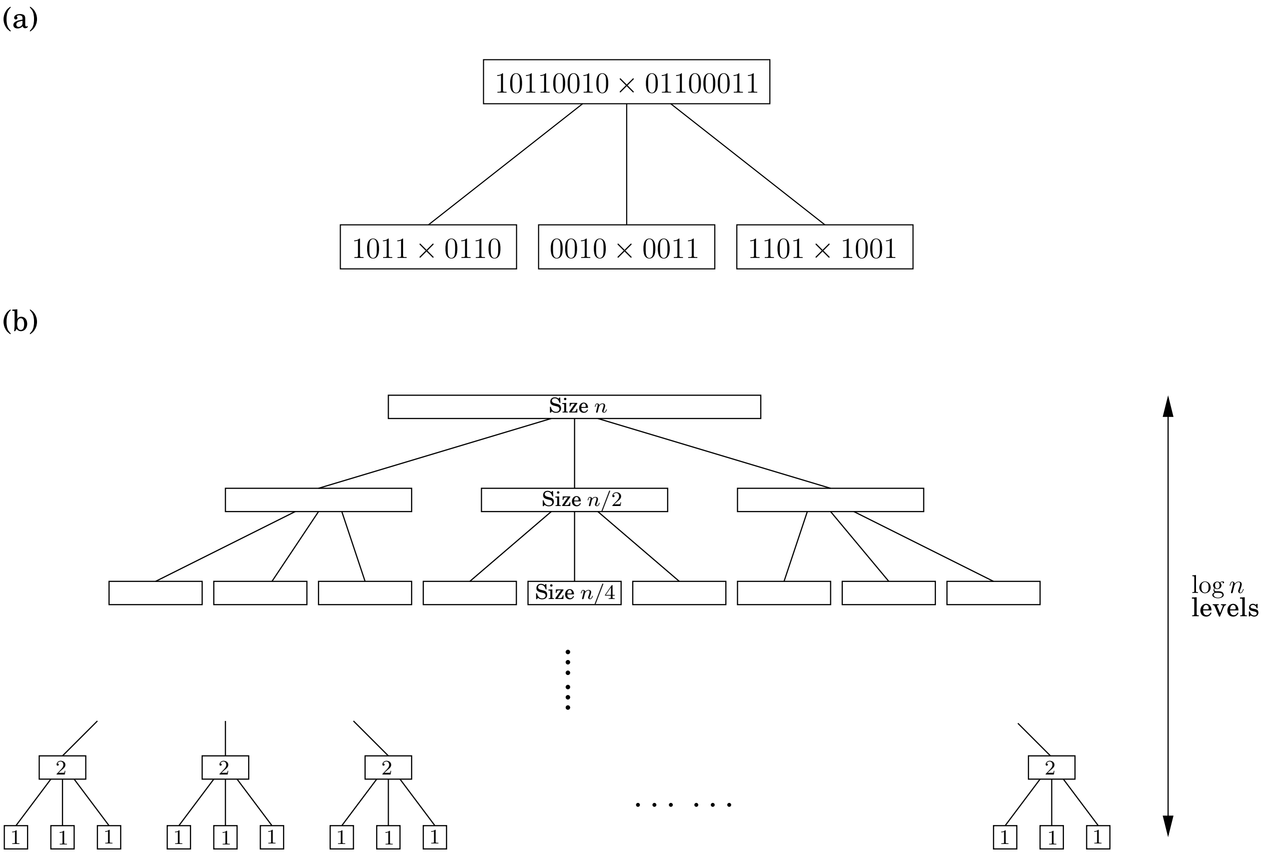
At the very top level, when $k = 0$, this works out to $O(n)$. At the bottom, when $k = \log_2 n$, it is $O(3^{\log_2 n})$, which can be rewritten as $O(n^{\log_2 3})$ (do you see why?). Between these two endpoints, the work done increases *geometrically* from $O(n)$ to $O(n^{\log_2 3})$, by a factor of $3/2$ per level. The sum of any increasing geometric series is, within a constant factor, simply the last term of the series: such is the rapidity of the increase (Exercise 0.2). Therefore the overall running time is $O(n^{\log_2 3})$, which is about $O(n^{1.59})$.

In the absence of Gauss's trick, the recursion tree would have the same height, but the branching factor would be 4. There would be $4^{\log_2 n} = n^2$ leaves, and therefore the running time would be at least this much. In divide-and-conquer algorithms, the number of subproblems translates into the branching factor of the recursion tree; small changes in this coefficient can have a big impact on running time.

A practical note: it generally does not make sense to recurse all the way down to 1 bit. For most processors, 16- or 32-bit multiplication is a single operation, so by the time the numbers get into this range they should be handed over to the built-in procedure.

Finally, the eternal question: *Can we do better?* It turns out that even faster algorithms for multiplying numbers exist, based on another important divide-and-conquer algorithm: the fast Fourier transform, to be explained in Section 2.6.

Figure 2.2 Divide-and-conquer integer multiplication. (a) Each problem is divided into three subproblems. (b) The levels of recursion.



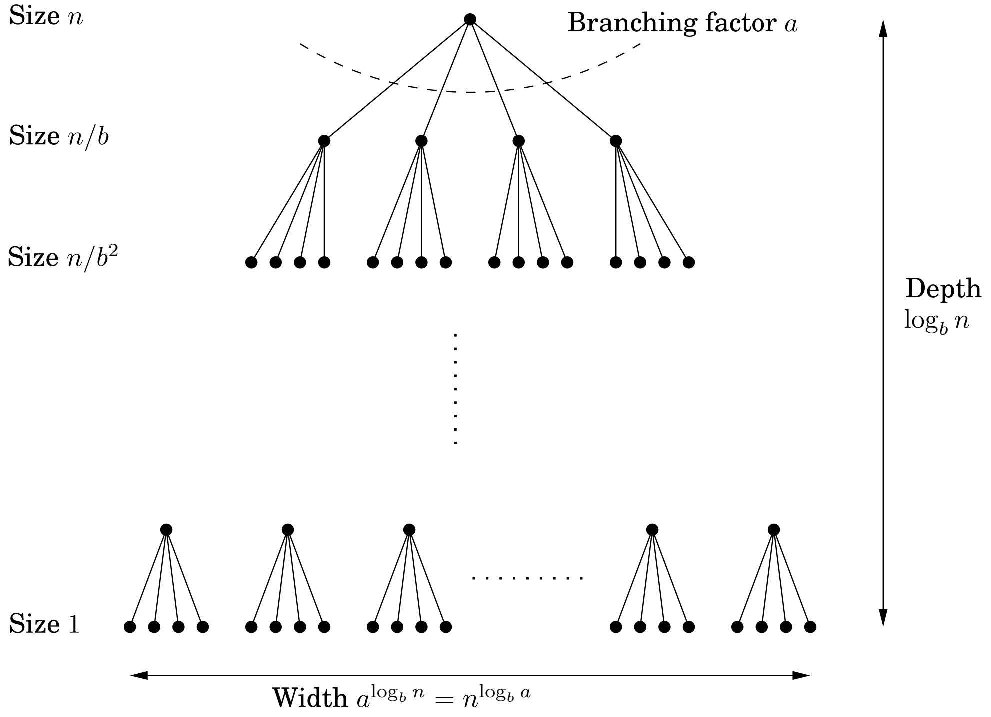
2.2 Recurrence relations

Divide-and-conquer algorithms often follow a generic pattern: they tackle a problem of size n by recursively solving, say, a subproblems of size n/b and then combining these answers in $O(n^d)$ time, for some $a, b, d > 0$ (in the multiplication algorithm, $a = 3$, $b = 2$, and $d = 1$). Their running time can therefore be captured by the equation $T(n) = aT(\lceil n/b \rceil) + O(n^d)$. We next derive a closed-form solution to this general recurrence so that we no longer have to solve it explicitly in each new instance.

Master theorem[2] *If* $T(n) = aT(\lceil n/b \rceil) + O(n^d)$ *for some constants* $a > 0$*,* $b > 1$*, and* $d \geq 0$*,*

[2] There are even more general results of this type, but we will not be needing them.

Figure 2.3 Each problem of size n is divided into a subproblems of size n/b.



then

$$T(n) = \begin{cases} O(n^d) & \textit{if } d > \log_b a \\ O(n^d \log n) & \textit{if } d = \log_b a \\ O(n^{\log_b a}) & \textit{if } d < \log_b a \,. \end{cases}$$

This single theorem tells us the running times of most of the divide-and-conquer procedures we are likely to use.

Proof. To prove the claim, let's start by assuming for the sake of convenience that n is a power of b. This will not influence the final bound in any important way—after all, n is at most a multiplicative factor of b away from some power of b (Exercise 2.2)—and it will allow us to ignore the rounding effect in $\lceil n/b \rceil$.

Next, notice that the size of the subproblems decreases by a factor of b with each level of recursion, and therefore reaches the base case after $\log_b n$ levels. This is the height of the recursion tree. Its branching factor is a, so the kth level of the tree is made up of a^k subproblems, each of size n/b^k (Figure 2.3). The total work done at this level is

$$a^k \times O\left(\frac{n}{b^k}\right)^d = O(n^d) \times \left(\frac{a}{b^d}\right)^k.$$

As k goes from 0 (the root) to $\log_b n$ (the leaves), these numbers form a geometric series with ratio a/b^d. Finding the sum of such a series in big-O notation is easy (Exercise 0.2), and comes down to three cases.

1. *The ratio is less than* 1.

   Then the series is decreasing, and its sum is just given by its first term, $O(n^d)$.

2. *The ratio is greater than* 1.

   The series is increasing and its sum is given by its last term, $O(n^{\log_b a})$:

$$n^d \left(\frac{a}{b^d}\right)^{\log_b n} = n^d \left(\frac{a^{\log_b n}}{(b^{\log_b n})^d}\right) = a^{\log_b n} = a^{(\log_a n)(\log_b a)} = n^{\log_b a}.$$

3. *The ratio is exactly* 1.

   In this case all $O(\log n)$ terms of the series are equal to $O(n^d)$.

These cases translate directly into the three contingencies in the theorem statement. ▮

Binary search

The ultimate divide-and-conquer algorithm is, of course, *binary search:* to find a key k in a large file containing keys $z[0, 1, \ldots, n-1]$ in sorted order, we first compare k with $z[n/2]$, and depending on the result we recurse either on the first half of the file, $z[0, \ldots, n/2 - 1]$, or on the second half, $z[n/2, \ldots, n-1]$. The recurrence now is $T(n) = T(\lceil n/2 \rceil) + O(1)$, which is the case $a = 1, b = 2, d = 0$. Plugging into our master theorem we get the familiar solution: a running time of just $O(\log n)$.

2.3 Mergesort

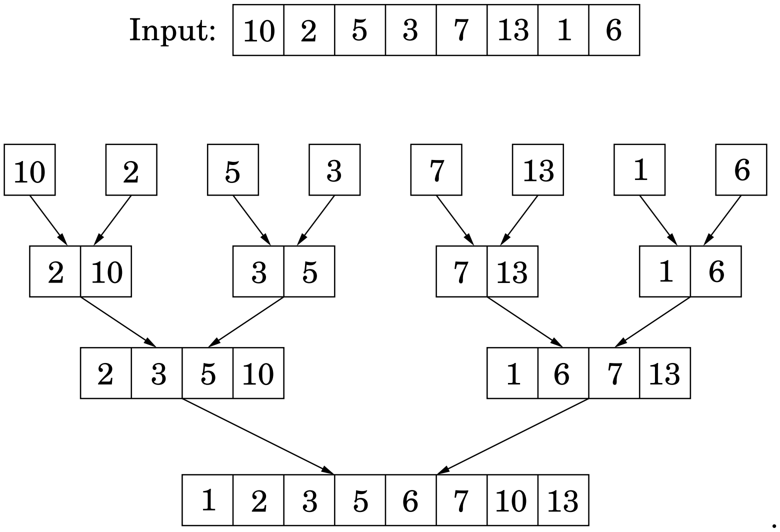
The problem of sorting a list of numbers lends itself immediately to a divide-and-conquer strategy: split the list into two halves, recursively sort each half, and then *merge* the two sorted sublists.

```
function mergesort($a[1 \ldots n]$)
Input:  An array of numbers $a[1 \ldots n]$
Output:  A sorted version of this array

if $n > 1$:
  return merge(mergesort($a[1 \ldots \lfloor n/2 \rfloor]$), mergesort($a[\lfloor n/2 \rfloor + 1 \ldots n]$))
else:
  return $a$
```

The correctness of this algorithm is self-evident, as long as a correct `merge` subroutine is specified. If we are given two sorted arrays $x[1 \ldots k]$ and $y[1 \ldots l]$, how do we efficiently merge them into a single sorted array $z[1 \ldots k + l]$? Well, the very first element of z is either $x[1]$ or $y[1]$, whichever is smaller. The rest of $z[\cdot]$ can then be constructed recursively.

Figure 2.4 The sequence of merge operations in `mergesort`.



```
function merge($x[1\ldots k], y[1\ldots l]$)
if $k = 0$:  return $y[1\ldots l]$
if $l = 0$:  return $x[1\ldots k]$
if $x[1] \leq y[1]$:
  return $x[1] \circ$ merge($x[2\ldots k], y[1\ldots l]$)
else:
  return $y[1] \circ$ merge($x[1\ldots k], y[2\ldots l]$)
```

Here $\circ$ denotes concatenation. This `merge` procedure does a constant amount of work per recursive call (provided the required array space is allocated in advance), for a total running time of $O(k + l)$. Thus `merge`'s are linear, and the overall time taken by `mergesort` is

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

or $O(n \log n)$.

Looking back at the `mergesort` algorithm, we see that all the real work is done in merging, which doesn't start until the recursion gets down to singleton arrays. The singletons are merged in pairs, to yield arrays with two elements. Then pairs of these 2-tuples are merged, producing 4-tuples, and so on. Figure 2.4 shows an example.

This viewpoint also suggests how `mergesort` might be made iterative. At any given moment, there is a set of "active" arrays—initially, the singletons—which are merged in pairs to give the next batch of active arrays. These arrays can be organized in a queue, and processed by repeatedly removing two arrays from the front of the queue, merging them, and putting the result at the end of the queue.

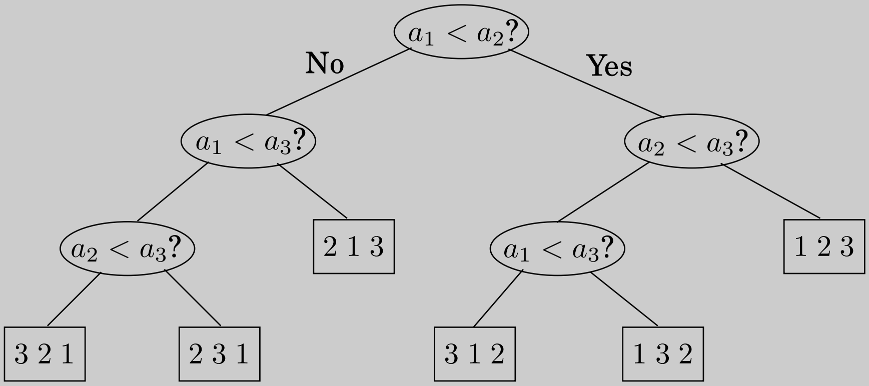
In the following pseudocode, the primitive operation `inject` adds an element to the end of the queue while `eject` removes and returns the element at the front of the queue.

```
function iterative-mergesort($a[1 \ldots n]$)
Input:  elements $a_1, a_2, \ldots, a_n$ to be sorted

$Q = [\ ]$ (empty queue)
for $i = 1$ to $n$:
   inject($Q, [a_i]$)
while $|Q| > 1$:
   inject($Q$,merge(eject($Q$),eject($Q$)))
return eject($Q$)
```

An $n \log n$ lower bound for sorting

Sorting algorithms can be depicted as trees. The one in the following figure sorts an array of three elements, a_1, a_2, a_3. It starts by comparing a_1 to a_2 and, if the first is larger, compares it with a_3; otherwise it compares a_2 and a_3. And so on. Eventually we end up at a leaf, and this leaf is labeled with the true order of the three elements as a permutation of $1, 2, 3$. For example, if $a_2 < a_1 < a_3$, we get the leaf labeled "2 1 3."



The *depth* of the tree—the number of comparisons on the longest path from root to leaf, in this case 3—is exactly the worst-case time complexity of the algorithm.

This way of looking at sorting algorithms is useful because it allows one to argue that *mergesort is optimal*, in the sense that $\Omega(n \log n)$ comparisons are necessary for sorting n elements.

Here is the argument: Consider any such tree that sorts an array of n elements. Each of its leaves is labeled by a permutation of $\{1, 2, \dots, n\}$. In fact, *every* permutation must appear as the label of a leaf. The reason is simple: if a particular permutation is missing, what happens if we feed the algorithm an input ordered according to this same permutation? And since there are $n!$ permutations of n elements, it follows that the tree has at least $n!$ leaves.

We are almost done: This is a binary tree, and we argued that it has at least $n!$ leaves. Recall now that a binary tree of depth d has at most 2^d leaves (proof: an easy induction on d). So, the depth of our tree—and the complexity of our algorithm—must be at least $\log(n!)$.

And it is well known that $\log(n!) \geq c \cdot n \log n$ for some $c > 0$. There are many ways to see this. The easiest is to notice that $n! \geq (n/2)^{(n/2)}$ because $n! = 1 \cdot 2 \cdot \cdots \cdot n$ contains at least $n/2$ factors larger than $n/2$; and to then take logs of both sides. Another is to recall Stirling's formula

$$n! \approx \sqrt{\pi \left(2n + \frac{1}{3}\right)} \cdot n^n \cdot e^{-n}.$$

Either way, we have established that any comparison tree that sorts n elements must make, in the worst case, $\Omega(n \log n)$ comparisons, and hence mergesort is optimal!

Well, there is some fine print: this neat argument applies only to *algorithms that use comparisons*. Is it conceivable that there are alternative sorting strategies, perhaps using sophisticated numerical manipulations, that work in linear time? The answer is *yes*, under certain exceptional circumstances: the canonical such example is when the elements to be sorted are integers that lie in a small range (Exercise 2.20).

2.4 Medians

The *median* of a list of numbers is its 50th percentile: half the numbers are bigger than it, and half are smaller. For instance, the median of $[45, 1, 10, 30, 25]$ is 25, since this is the middle element when the numbers are arranged in order. If the list has even length, there are two choices for what the middle element could be, in which case we pick the smaller of the two, say.

The purpose of the median is to summarize a set of numbers by a single, typical value. The *mean*, or average, is also very commonly used for this, but the median is in a sense more typical of the data: it is always one of the data values, unlike the mean, and it is less sensitive to outliers. For instance, the median of a list of a hundred 1's is (rightly) 1, as is the mean. However, if just one of these numbers gets accidentally corrupted to 10,000, the mean shoots up above 100, while the median is unaffected.

Computing the median of n numbers is easy: just sort them. The drawback is that this takes $O(n \log n)$ time, whereas we would ideally like something linear. We have reason to be hopeful, because sorting is doing far more work than we really need—we just want the middle element and don't care about the relative ordering of the rest of them.

When looking for a recursive solution, it is paradoxically often easier to work with a *more general* version of the problem—for the simple reason that this gives a more powerful step to recurse upon. In our case, the generalization we will consider is *selection*.

SELECTION

Input: A list of numbers S; an integer k

Output: The kth smallest element of S

For instance, if $k = 1$, the minimum of S is sought, whereas if $k = \lfloor |S|/2 \rfloor$, it is the median.

A randomized divide-and-conquer algorithm for selection

Here's a divide-and-conquer approach to selection. For any number v, imagine splitting list S into three categories: elements smaller than v, those equal to v (there might be duplicates), and those greater than v. Call these S_L, S_v, and S_R respectively. For instance, if the array

$$S : \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|}\hline 2 & 36 & 5 & 21 & 8 & 13 & 11 & 20 & 5 & 4 & 1 \\ \hline \end{array}$$

is split on $v = 5$, the three subarrays generated are

$$S_L : \begin{array}{|c|c|c|}\hline 2 & 4 & 1 \\ \hline \end{array} \qquad S_v : \begin{array}{|c|c|}\hline 5 & 5 \\ \hline \end{array} \qquad S_R : \begin{array}{|c|c|c|c|c|c|}\hline 36 & 21 & 8 & 13 & 11 & 20 \\ \hline \end{array}$$

The search can instantly be narrowed down to one of these sublists. If we want, say, the *eighth*-smallest element of S, we know it must be the *third*-smallest element of S_R since $|S_L| + |S_v| = 5$. That is, $\text{selection}(S, 8) = \text{selection}(S_R, 3)$. More generally, by checking k against the sizes of the subarrays, we can quickly determine which of them holds the desired element:

$$\text{selection}(S, k) = \begin{cases} \text{selection}(S_L, k) & \text{if } k \leq |S_L| \\ v & \text{if } |S_L| < k \leq |S_L| + |S_v| \\ \text{selection}(S_R, k - |S_L| - |S_v|) & \text{if } k > |S_L| + |S_v|. \end{cases}$$

The three sublists S_L, S_v, and S_R can be computed from S in linear time; in fact, this computation can even be done *in place*, that is, without allocating new memory (Exercise 2.15). We then recurse on the appropriate sublist. The effect of the split is thus to shrink the number of elements from $|S|$ to at most $\max\{|S_L|, |S_R|\}$.

Our divide-and-conquer algorithm for selection is now fully specified, except for the crucial detail of how to choose v. It should be picked quickly, and it should shrink the array substantially, the ideal situation being $|S_L|, |S_R| \approx \frac{1}{2}|S|$. If we could always guarantee this situation, we would get a running time of

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

which is linear as desired. But this requires picking v to be the median, which is our ultimate goal! Instead, we follow a much simpler alternative: *we pick v randomly from S.*

Efficiency analysis

Naturally, the running time of our algorithm depends on the random choices of v. It is possible that due to persistent bad luck we keep picking v to be the largest element of the array (or the smallest element), and thereby shrink the array by only one element each time. In the earlier example, we might first pick $v = 36$, then $v = 21$, and so on. This *worst-case* scenario would force our selection algorithm to perform

$$n + (n-1) + (n-2) + \cdots + \frac{n}{2} = \Theta(n^2)$$

operations (when computing the median), but it is extremely unlikely to occur. Equally unlikely is the *best* possible case we discussed before, in which each randomly chosen v just happens to split the array perfectly in half, resulting in a running time of $O(n)$. Where, in this spectrum from $O(n)$ to $\Theta(n^2)$, does the *average* running time lie? Fortunately, it lies very close to the best-case time.

To distinguish between lucky and unlucky choices of v, we will call v *good* if it lies within the 25th to 75th percentile of the array that it is chosen from. We like these choices of v because they ensure that the sublists S_L and S_R have size at most three-fourths that of S (do you see why?), so that the array shrinks substantially. Fortunately, good v's are abundant: half the elements of any list must fall between the 25th to 75th percentile!

Given that a randomly chosen v has a 50% chance of being good, how many v's do we need to pick on average before getting a good one? Here's a more familiar reformulation (see also Exercise 1.34):

Lemma *On average a fair coin needs to be tossed two times before a "heads" is seen.*

Proof. Let E be the expected number of tosses before a heads is seen. We certainly need at least one toss, and if it's heads, we're done. If it's tails (which occurs with probability $1/2$), we need to repeat. Hence $E = 1 + \frac{1}{2}E$, which works out to $E = 2$. ▮

Therefore, after two split operations on average, the array will shrink to at most three-fourths of its size. Letting $T(n)$ be the *expected* running time on an array of size n, we get

$$T(n) \leq T(3n/4) + O(n).$$

This follows by taking expected values of both sides of the following statement:

$$\begin{aligned} &\text{Time taken on an array of size } n \\ &\quad \leq \quad (\text{time taken on an array of size } 3n/4) + (\text{time to reduce array size to} \leq 3n/4), \end{aligned}$$

and, for the right-hand side, using the familiar property that *the expectation of the sum is the sum of the expectations*.

From this recurrence we conclude that $T(n) = O(n)$: on *any* input, our algorithm returns the correct answer after a linear number of steps, on the average.

The Unix `sort` command

Comparing the algorithms for sorting and median-finding we notice that, beyond the common divide-and-conquer philosophy and structure, they are exact opposites. Mergesort splits the array in two in the most convenient way (first half, second half), without any regard to the magnitudes of the elements in each half; but then it works hard to put the sorted subarrays together. In contrast, the median algorithm is careful about its splitting (smaller numbers first, then the larger ones), but its work ends with the recursive call.

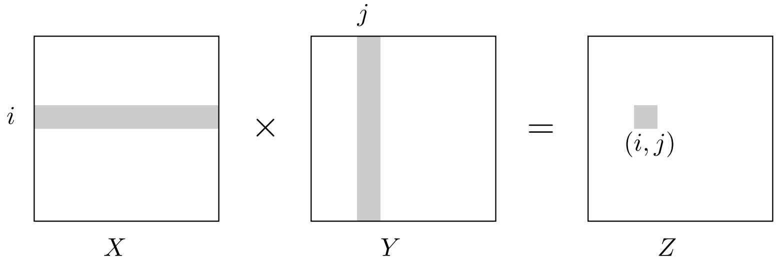
Quicksort is a sorting algorithm that splits the array in exactly the same way as the median algorithm; and once the subarrays are sorted, by two recursive calls, there is nothing more to do. Its worst-case performance is $\Theta(n^2)$, like that of median-finding. But it can be proved (Exercise 2.24) that its average case is $O(n \log n)$; furthermore, empirically it outperforms other sorting algorithms. This has made quicksort a favorite in many applications—for instance, it is the basis of the code by which really enormous files are sorted.

2.5 Matrix multiplication

The product of two $n \times n$ matrices X and Y is a third $n \times n$ matrix $Z = XY$, with (i, j)th entry

$$Z_{ij} = \sum_{k=1}^{n} X_{ik} Y_{kj}.$$

To make it more visual, Z_{ij} is the dot product of the ith row of X with the jth column of Y:



In general, XY is not the same as YX; matrix multiplication is not commutative.

The preceding formula implies an $O(n^3)$ algorithm for matrix multiplication: there are n^2 entries to be computed, and each takes $O(n)$ time. For quite a while, this was widely believed to be the best running time possible, and it was even proved that in certain models of computation no algorithm could do better. It was therefore a source of great excitement when in 1969, the German mathematician Volker Strassen announced a significantly more efficient algorithm, based upon divide-and-conquer.

Matrix multiplication is particularly easy to break into subproblems, because it can be performed *blockwise*. To see what this means, carve X into four $n/2 \times n/2$ blocks, and also Y:

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}.$$

Then their product can be expressed in terms of these blocks and is exactly as if the blocks were single elements (Exercise 2.11).

$$XY = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} = \begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix}$$

We now have a divide-and-conquer strategy: to compute the size-n product XY, recursively compute eight size-$n/2$ products $AE, BG, AF, BH, CE, DG, CF, DH$, and then do a few $O(n^2)$-time additions. The total running time is described by the recurrence relation

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

This comes out to an unimpressive $O(n^3)$, the same as for the default algorithm. But the efficiency *can* be further improved, and as with integer multiplication, the key is some clever algebra. It turns out XY can be computed from just *seven* $n/2 \times n/2$ subproblems, via a decomposition so tricky and intricate that one wonders how Strassen was ever able to discover it!

$$XY = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{bmatrix}$$

where

$$\begin{aligned} P_1 &= A(F-H) & \qquad P_5 &= (A+D)(E+H) \\ P_2 &= (A+B)H & P_6 &= (B-D)(G+H) \\ P_3 &= (C+D)E & P_7 &= (A-C)(E+F) \\ P_4 &= D(G-E) & & \end{aligned}$$

The new running time is

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

which by the master theorem works out to $O(n^{\log_2 7}) \approx O(n^{2.81})$.

2.6 The fast Fourier transform

We have so far seen how divide-and-conquer gives fast algorithms for multiplying integers and matrices; our next target is *polynomials*. The product of two degree-d polynomials is a polynomial of degree $2d$, for example:

$$(1 + 2x + 3x^2) \cdot (2 + x + 4x^2) = 2 + 5x + 12x^2 + 11x^3 + 12x^4.$$

More generally, if $A(x) = a_0 + a_1x + \cdots + a_dx^d$ and $B(x) = b_0 + b_1x + \cdots + b_dx^d$, their product $C(x) = A(x) \cdot B(x) = c_0 + c_1x + \cdots + c_{2d}x^{2d}$ has coefficients

$$c_k \;=\; a_0b_k + a_1b_{k-1} + \cdots + a_kb_0 \;=\; \sum_{i=0}^{k} a_ib_{k-i}$$

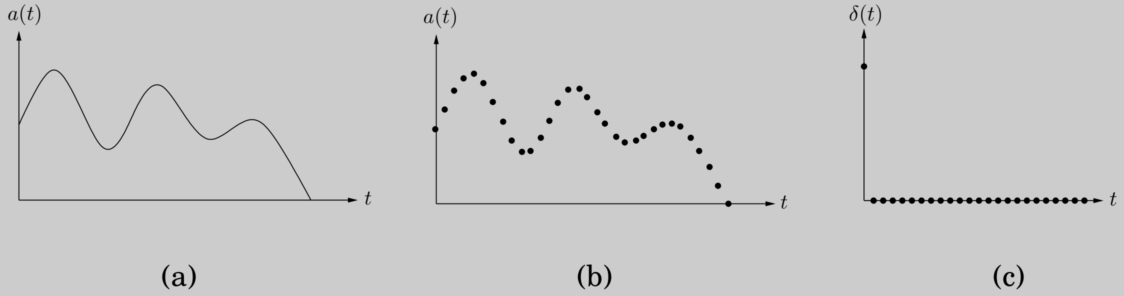
(for $i > d$, take a_i and b_i to be zero). Computing c_k from this formula takes $O(k)$ steps, and finding all $2d + 1$ coefficients would therefore seem to require $\Theta(d^2)$ time. *Can we possibly multiply polynomials faster than this?*

The solution we will develop, the fast Fourier transform, has revolutionized—indeed, defined—the field of signal processing (see the following box). Because of its huge importance, and its wealth of insights from different fields of study, we will approach it a little more leisurely than usual. The reader who wants just the core algorithm can skip directly to Section 2.6.4.

Why multiply polynomials?

For one thing, it turns out that the fastest algorithms we have for multiplying integers rely heavily on polynomial multiplication; after all, polynomials and binary integers are quite similar—just replace the variable x by the base 2, and watch out for carries. But perhaps more importantly, multiplying polynomials is crucial for *signal processing*.

A *signal* is any quantity that is a function of time (as in Figure (a)) or of position. It might, for instance, capture a human voice by measuring fluctuations in air pressure close to the speaker's mouth, or alternatively, the pattern of stars in the night sky, by measuring brightness as a function of angle.



(a) (b) (c)

In order to extract information from a signal, we need to first *digitize* it by sampling (Figure (b))—and, then, to put it through a *system* that will transform it in some way. The output is called the *response* of the system:

$$\text{signal} \longrightarrow \boxed{\text{SYSTEM}} \longrightarrow \text{response}$$

An important class of systems are those that are *linear*—the response to the sum of two signals is just the sum of their individual responses—and *time invariant*—shifting the input signal by time t produces the same output, also shifted by t. Any system with these properties is completely characterized by its response to the simplest possible input signal: the *unit impulse* $\delta(t)$, consisting solely of a "jerk" at $t = 0$ (Figure (c)). To see this, first consider the close relative $\delta(t - i)$, a shifted impulse in which the jerk occurs at time i. Any signal $a(t)$ can be expressed as a linear combination of these, letting $\delta(t - i)$ pick out its behavior at time i,

$$a(t) = \sum_{i=0}^{T-1} a(i)\delta(t - i)$$

(if the signal consists of T samples). By linearity, the system response to input $a(t)$ is determined by the responses to the various $\delta(t-i)$. And by time invariance, these are in turn just shifted copies of the *impulse response* $b(t)$, the response to $\delta(t)$.

In other words, the output of the system at time k is

$$c(k) \;=\; \sum_{i=0}^{k} a(i)b(k - i),$$

exactly the formula for polynomial multiplication!

2.6.1 An alternative representation of polynomials

To arrive at a fast algorithm for polynomial multiplication we take inspiration from an important property of polynomials.

Fact *A degree-d polynomial is uniquely characterized by its values at any $d+1$ distinct points.*

A familiar instance of this is that "any two points determine a line." We will later see why the more general statement is true (page 76), but for the time being it gives us an *alternative representation* of polynomials. Fix any distinct points $x_0, \ldots, x_d$. We can specify a degree-d polynomial $A(x) = a_0 + a_1 x + \cdots + a_d x^d$ by either one of the following:

1. Its coefficients $a_0, a_1, \ldots, a_d$

2. The values $A(x_0), A(x_1), \ldots, A(x_d)$

Of these two representations, the second is the more attractive for polynomial multiplication. Since the product $C(x)$ has degree $2d$, it is completely determined by its value at any $2d+1$ points. And its value at any given point z is easy enough to figure out, just $A(z)$ times $B(z)$. Thus *polynomial multiplication takes linear time in the value representation.*

The problem is that we expect the input polynomials, and also their product, to be specified by coefficients. So we need to first translate from coefficients to values—which is just a matter of *evaluating* the polynomial at the chosen points—then multiply in the value representation, and finally translate back to coefficients, a process called *interpolation*.

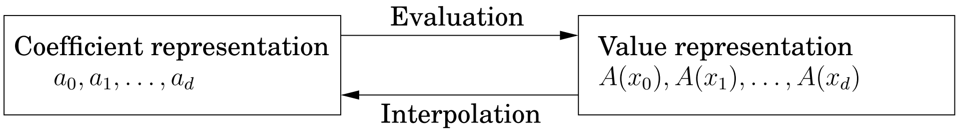


Figure 2.5 presents the resulting algorithm.

The equivalence of the two polynomial representations makes it clear that this high-level approach is correct, but how efficient is it? Certainly the selection step and the n multiplications are no trouble at all, just linear time.[3] But (leaving aside interpolation, about which we know even less) how about evaluation? Evaluating a polynomial of degree $d \le n$ at a single point takes $O(n)$ steps (Exercise 2.29), and so the baseline for n points is $\Theta(n^2)$. We'll now see that the fast Fourier transform (FFT) does it in just $O(n \log n)$ time, for a particularly clever choice of $x_0, \ldots, x_{n-1}$ in which the computations required by the individual points overlap with one another and can be shared.

[3] In a typical setting for polynomial multiplication, the coefficients of the polynomials are real numbers and, moreover, are small enough that basic arithmetic operations (adding and multiplying) take unit time. We will assume this to be the case without any great loss of generality; in particular, the time bounds we obtain are easily adjustable to situations with larger numbers.

Figure 2.5 Polynomial multiplication

```
Input:  Coefficients of two polynomials, $A(x)$ and $B(x)$, of degree $d$
Output:  Their product $C = A \cdot B$

Selection
    Pick some points $x_0, x_1, \ldots, x_{n-1}$, where $n \geq 2d + 1$
Evaluation
    Compute $A(x_0), A(x_1), \ldots, A(x_{n-1})$ and $B(x_0), B(x_1), \ldots, B(x_{n-1})$
Multiplication
    Compute $C(x_k) = A(x_k)B(x_k)$ for all $k = 0, \ldots, n-1$
Interpolation
    Recover $C(x) = c_0 + c_1 x + \cdots + c_{2d} x^{2d}$
```

2.6.2 Evaluation by divide-and-conquer

Here's an idea for how to pick the n points at which to evaluate a polynomial $A(x)$ of degree $\leq n - 1$. If we choose them to be positive-negative pairs, that is,

$$\pm x_0, \pm x_1, \ldots, \pm x_{n/2-1},$$

then the computations required for each $A(x_i)$ and $A(-x_i)$ overlap a lot, because the even powers of x_i coincide with those of $-x_i$.

To investigate this, we need to split $A(x)$ into its odd and even powers, for instance

$$3 + 4x + 6x^2 + 2x^3 + x^4 + 10x^5 \ = \ (3 + 6x^2 + x^4) + x(4 + 2x^2 + 10x^4).$$

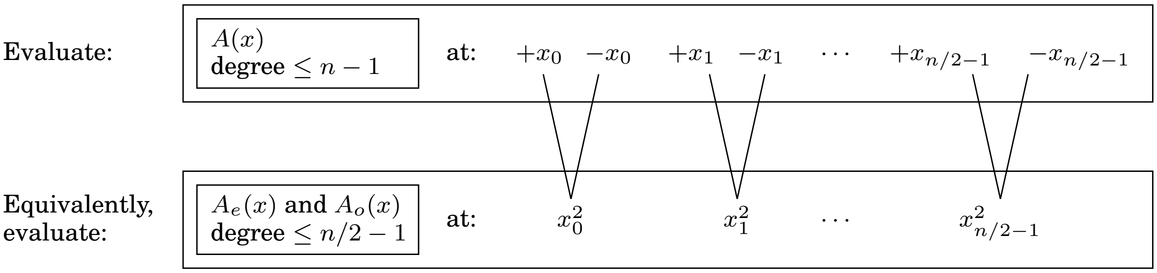
Notice that the terms in parentheses are polynomials in x^2. More generally,

$$A(x) \ = \ A_e(x^2) + xA_o(x^2),$$

where $A_e(\cdot)$, with the even-numbered coefficients, and $A_o(\cdot)$, with the odd-numbered coefficients, are polynomials of degree $\leq n/2 - 1$ (assume for convenience that n is even). Given *paired* points $\pm x_i$, the calculations needed for $A(x_i)$ can be recycled toward computing $A(-x_i)$:

$$\begin{aligned} A(x_i) &= A_e(x_i^2) + x_i A_o(x_i^2) \\ A(-x_i) &= A_e(x_i^2) - x_i A_o(x_i^2). \end{aligned}$$

In other words, evaluating $A(x)$ at n paired points $\pm x_0, \ldots, \pm x_{n/2-1}$ reduces to evaluating $A_e(x)$ and $A_o(x)$ (which each have half the degree of $A(x)$) at just $n/2$ points, $x_0^2, \ldots, x_{n/2-1}^2$.



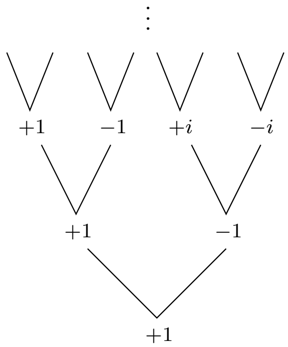
The original problem of size n is in this way recast as two subproblems of size $n/2$, followed by some linear-time arithmetic. If we could recurse, we would get a divide-and-conquer procedure with running time

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

which is $O(n \log n)$, exactly what we want.

But we have a problem: The plus-minus trick only works at the top level of the recursion. To recurse at the next level, we need the $n/2$ evaluation points $x_0^2, x_1^2, \ldots, x_{n/2-1}^2$ to be *themselves* plus-minus pairs. But how can a square be negative? The task seems impossible! *Unless, of course, we use complex numbers*.

Fine, but which complex numbers? To figure this out, let us "reverse engineer" the process. At the very bottom of the recursion, we have a single point. This point might as well be 1, in which case the level above it must consist of its square roots, $\pm\sqrt{1} = \pm 1$.



The next level up then has $\pm\sqrt{+1} = \pm 1$ as well as the *complex* numbers $\pm\sqrt{-1} = \pm i$, where i is the imaginary unit. By continuing in this manner, we eventually reach the initial set of n points. Perhaps you have already guessed what they are: the *complex nth roots of unity*, that is, the n complex solutions to the equation $z^n = 1$.

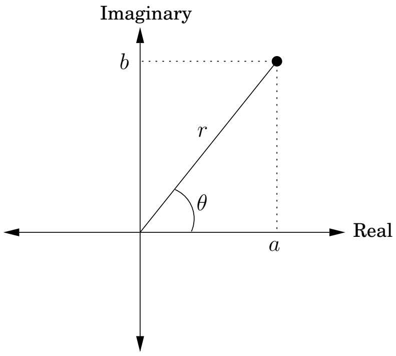
Figure 2.6 is a pictorial review of some basic facts about complex numbers. The third panel of this figure introduces the nth roots of unity: the complex numbers $1, \omega, \omega^2, \ldots, \omega^{n-1}$, where $\omega = e^{2\pi i/n}$. If n is even,

1. The nth roots are plus-minus paired, $\omega^{n/2+j} = -\omega^j$.

2. Squaring them produces the $(n/2)$nd roots of unity.

Therefore, if we start with these numbers for some n that is a power of 2, then at successive levels of recursion we will have the $(n/2^k)$th roots of unity, for $k = 0, 1, 2, 3, \ldots$. All these sets of numbers are plus-minus paired, and so our divide-and-conquer, as shown in the last panel, works perfectly. The resulting algorithm is the fast Fourier transform (Figure 2.7).

Figure 2.6 The complex roots of unity are ideal for our divide-and-conquer scheme.



The complex plane

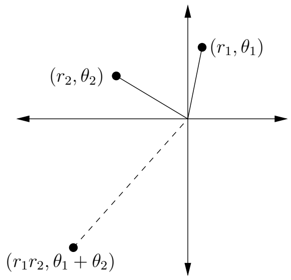
$z = a + bi$ is plotted at position (a, b).

Polar coordinates: rewrite as $z = r(\cos\theta + i\sin\theta) = re^{i\theta}$, denoted (r, θ).

- *length* $r = \sqrt{a^2 + b^2}$.
- *angle* $\theta \in [0, 2\pi)$: $\cos\theta = a/r, \sin\theta = b/r$.
- θ can always be reduced modulo 2π.

Examples:

Number	-1	i	$5 + 5i$
Polar coords	$(1, \pi)$	$(1, \pi/2)$	$(5\sqrt{2}, \pi/4)$

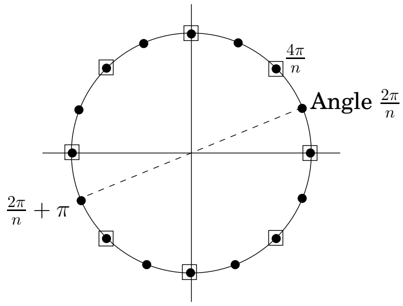


Multiplying is easy in polar coordinates

Multiply the lengths and add the angles:
$(r_1, \theta_1) \times (r_2, \theta_2) = (r_1 r_2, \theta_1 + \theta_2)$.

For any $z = (r, \theta)$,

- $-z = (r, \theta + \pi)$ since $-1 = (1, \pi)$.
- If z is on the *unit circle* (i.e., $r = 1$), then $z^n = (1, n\theta)$.



The nth complex roots of unity

Solutions to the equation $z^n = 1$.

By the multiplication rule: solutions are $z = (1, \theta)$, for θ a multiple of $2\pi/n$ (shown here for $n = 16$).

For even n:

- These numbers are *plus-minus paired*: $-(1, \theta) = (1, \theta + \pi)$.
- Their squares are the $(n/2)$nd roots of unity, shown here with boxes around them.

Divide-and-conquer step

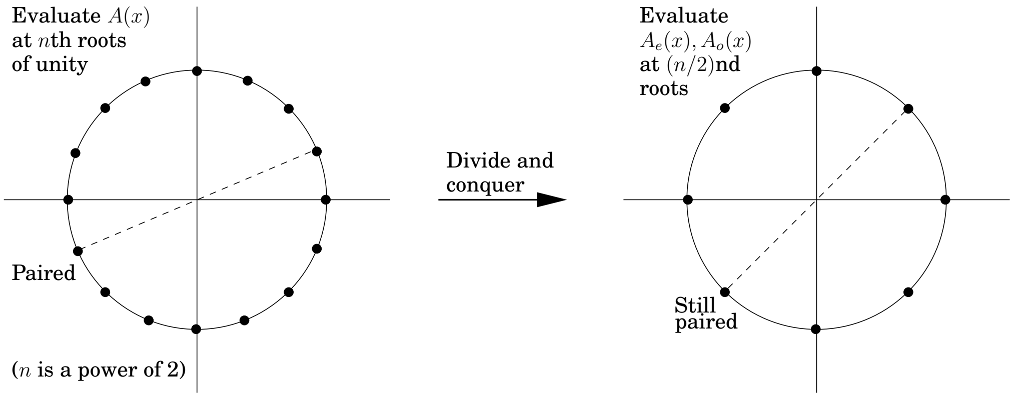

Evaluate $A(x)$ at nth roots of unity
Paired
(n is a power of 2)
Divide and conquer
Evaluate $A_e(x), A_o(x)$ at $(n/2)$nd roots
Still paired

Figure 2.7 The fast Fourier transform (polynomial formulation)

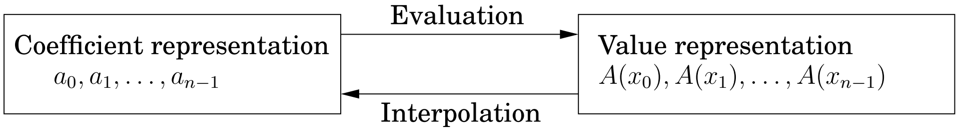
```
function FFT($A, \omega$)
Input:  Coefficient representation of a polynomial $A(x)$
        of degree $\leq n-1$, where $n$ is a power of $2$
        $\omega$, an $n$th root of unity
Output:  Value representation $A(\omega^0), \ldots, A(\omega^{n-1})$

if $\omega = 1$:  return $A(1)$
express $A(x)$ in the form $A_e(x^2) + xA_o(x^2)$
call FFT($A_e, \omega^2$) to evaluate $A_e$ at even powers of $\omega$
call FFT($A_o, \omega^2$) to evaluate $A_o$ at even powers of $\omega$
for $j = 0$ to $n-1$:
   compute $A(\omega^j) = A_e(\omega^{2j}) + \omega^j A_o(\omega^{2j})$

return $A(\omega^0), \ldots, A(\omega^{n-1})$
```

2.6.3 Interpolation

Let's take stock of where we are. We first developed a high-level scheme for multiplying polynomials (Figure 2.5), based on the observation that polynomials can be represented in two ways, in terms of their *coefficients* or in terms of their *values* at a selected set of points.


Evaluation
Coefficient representation
$a_0, a_1, \ldots, a_{n-1}$
Value representation
$A(x_0), A(x_1), \ldots, A(x_{n-1})$
Interpolation

The value representation makes it trivial to multiply polynomials, but we cannot ignore the coefficient representation since it is the form in which the input and output of our overall algorithm are specified.

So we designed the FFT, a way to move from coefficients to values in time just $O(n \log n)$, when the points $\{x_i\}$ are complex nth roots of unity ($1, \omega, \omega^2, \ldots, \omega^{n-1}$).

$$\langle\text{values}\rangle \;=\; \text{FFT}(\langle\text{coefficients}\rangle, \omega).$$

The last remaining piece of the puzzle is the inverse operation, interpolation. It will turn out, amazingly, that

$$\langle\text{coefficients}\rangle \;=\; \frac{1}{n}\,\text{FFT}(\langle\text{values}\rangle, \omega^{-1}).$$

Interpolation is thus solved in the most simple and elegant way we could possibly have hoped for—using the same FFT algorithm, but called with ω^{-1} in place of ω! This might seem like a miraculous coincidence, but it will make a lot more sense when we recast our polynomial operations in the language of linear algebra. Meanwhile, our $O(n \log n)$ polynomial multiplication algorithm (Figure 2.5) is now fully specified.

A matrix reformulation

To get a clearer view of interpolation, let's explicitly set down the relationship between our two representations for a polynomial $A(x)$ of degree $\leq n-1$. They are both vectors of n numbers, and one is a linear transformation of the other:

$$\begin{bmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ & & \vdots & & \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix}.$$

Call the matrix in the middle M. Its specialized format—a *Vandermonde* matrix—gives it many remarkable properties, of which the following is particularly relevant to us.

> *If $x_0, \ldots, x_{n-1}$ are distinct numbers, then M is invertible.*

The existence of M^{-1} allows us to invert the preceding matrix equation so as to express coefficients in terms of values. In brief,

> *Evaluation is multiplication by M, while interpolation is multiplication by M^{-1}.*

This reformulation of our polynomial operations reveals their essential nature more clearly. Among other things, it finally justifies an assumption we have been making throughout, that $A(x)$ is uniquely characterized by its values at any n points—in fact, we now have an explicit formula that will give us the coefficients of $A(x)$ in this situation. Vandermonde matrices also have the distinction of being quicker to invert than more general matrices, in $O(n^2)$ time instead of $O(n^3)$. However, using this for interpolation would still not be fast enough for us, so once again we turn to our special choice of points—the complex roots of unity.

Interpolation resolved

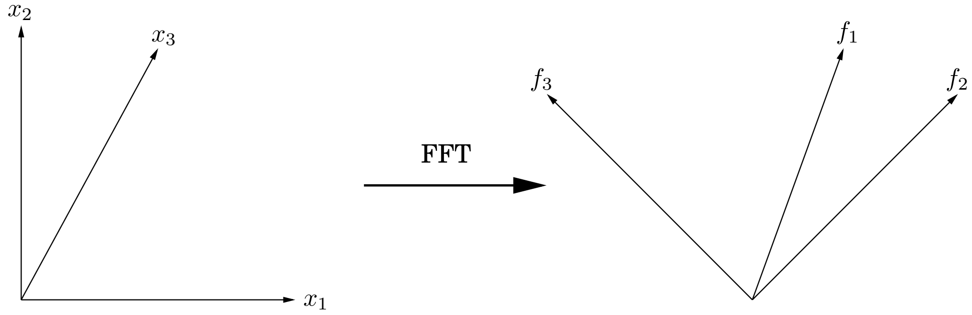
In linear algebra terms, the FFT multiplies an arbitrary n-dimensional vector—which we have been calling the *coefficient representation*—by the $n \times n$ matrix

$$M_n(\omega) = \left[\begin{array}{ccccc} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ & & \vdots & & \\ 1 & \omega^j & \omega^{2j} & \cdots & \omega^{(n-1)j} \\ & & \vdots & & \\ 1 & \omega^{(n-1)} & \omega^{2(n-1)} & \cdots & \omega^{(n-1)(n-1)} \end{array}\right] \begin{array}{ll} \longleftarrow & \text{row for } \omega^0 = 1 \\ \longleftarrow & \omega \\ \longleftarrow & \omega^2 \\ & \vdots \\ \longleftarrow & \omega^j \\ & \vdots \\ \longleftarrow & \omega^{n-1} \end{array}$$

where ω is a complex nth root of unity, and n is a power of 2. Notice how simple this matrix is to describe: its (j,k)th entry (starting row- and column-count at zero) is ω^{jk}.

Multiplication by $M = M_n(\omega)$ maps the kth coordinate axis (the vector with all zeros except for a 1 at position k) onto the kth column of M. Now here's the crucial observation, which we'll

Figure 2.8 The FFT takes points in the standard coordinate system, whose axes are shown here as x_1, x_2, x_3, and rotates them into the Fourier basis, whose axes are the columns of $M_n(\omega)$, shown here as f_1, f_2, f_3. For instance, points in direction x_1 get mapped into direction f_1.



prove shortly: *the columns of M are orthogonal (at right angles) to each other*. Therefore they can be thought of as the axes of an alternative coordinate system, which is often called the *Fourier basis*. The effect of multiplying a vector by M is to rotate it from the standard basis, with the usual set of axes, into the Fourier basis, which is defined by the columns of M (Figure 2.8). The FFT is thus a change of basis, a *rigid rotation*. The inverse of M is the opposite rotation, from the Fourier basis back into the standard basis. When we write out the orthogonality condition precisely, we will be able to read off this inverse transformation with ease:

Inversion formula $M_n(\omega)^{-1} \;=\; \frac{1}{n} M_n(\omega^{-1})$.

But ω^{-1} is also an nth root of unity, and so interpolation—or equivalently, multiplication by $M_n(\omega)^{-1}$—is itself just an FFT operation, but with ω replaced by ω^{-1}.

Now let's get into the details. Take ω to be $e^{2\pi i/n}$ for convenience, and think of the columns of M as vectors in $\mathbb{C}^n$. Recall that the *angle* between two vectors $u = (u_0, \ldots, u_{n-1})$ and $v = (v_0, \ldots, v_{n-1})$ in $\mathbb{C}^n$ is just a scaling factor times their *inner product*

$$u \cdot v^* \;=\; u_0 v_0^* + u_1 v_1^* + \cdots + u_{n-1} v_{n-1}^*,$$

where z^* denotes the complex conjugate[4] of z. This quantity is maximized when the vectors lie in the same direction and is zero when the vectors are orthogonal to each other.

The fundamental observation we need is the following.

Lemma *The columns of matrix M are orthogonal to each other.*

Proof. Take the inner product of any columns j and k of matrix M,

$$1 + \omega^{j-k} + \omega^{2(j-k)} + \cdots + \omega^{(n-1)(j-k)}.$$

[4] The *complex conjugate* of a complex number $z = re^{i\theta}$ is $z^* = re^{-i\theta}$. The complex conjugate of a vector (or matrix) is obtained by taking the complex conjugates of all its entries.

This is a geometric series with first term 1, last term $\omega^{(n-1)(j-k)}$, and ratio $\omega^{(j-k)}$. Therefore it evaluates to $(1-\omega^{n(j-k)})/(1-\omega^{(j-k)})$, which is 0—except when $j=k$, in which case all terms are 1 and the sum is n. ▮

The orthogonality property can be summarized in the single equation

$$MM^* = nI,$$

since $(MM^*)_{ij}$ is the inner product of the ith and jth columns of M (do you see why?). This immediately implies $M^{-1} = (1/n)M^*$: we have an inversion formula! But is it the same formula we earlier claimed? Let's see—the (j,k)th entry of M^* is the complex conjugate of the corresponding entry of M, in other words ω^{-jk}. Whereupon $M^* = M_n(\omega^{-1})$, and we're done.

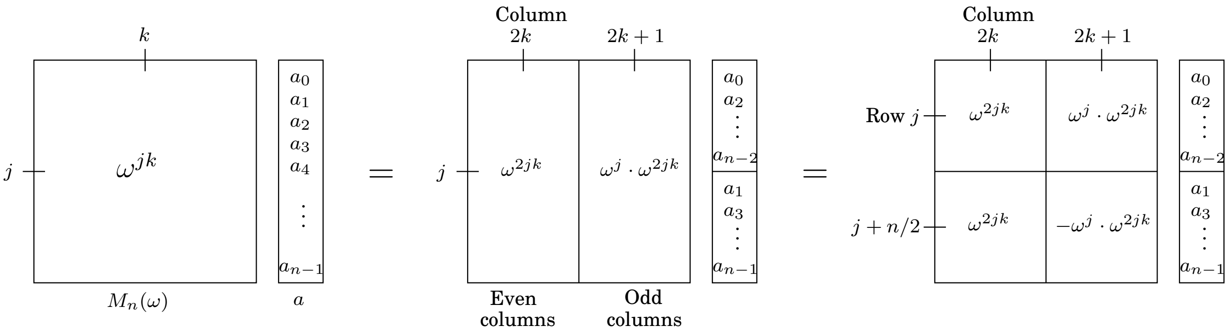
And now we can finally step back and view the whole affair geometrically. The task we need to perform, polynomial multiplication, is a lot easier in the Fourier basis than in the standard basis. Therefore, we first rotate vectors into the Fourier basis (*evaluation*), then perform the task (*multiplication*), and finally rotate back (*interpolation*). The initial vectors are *coefficient representations*, while their rotated counterparts are *value representations*. To efficiently switch between these, back and forth, is the province of the FFT.

2.6.4 A closer look at the fast Fourier transform

Now that our efficient scheme for polynomial multiplication is fully realized, let's hone in more closely on the core subroutine that makes it all possible, the fast Fourier transform.

The definitive FFT algorithm

The FFT takes as input a vector $a = (a_0, \ldots, a_{n-1})$ and a complex number ω whose powers $1, \omega, \omega^2, \ldots, \omega^{n-1}$ are the complex nth roots of unity. It multiplies vector a by the $n \times n$ matrix $M_n(\omega)$, which has (j,k)th entry (starting row- and column-count at zero) ω^{jk}. The potential for using divide-and-conquer in this matrix-vector multiplication becomes apparent when M's columns are segregated into evens and odds:



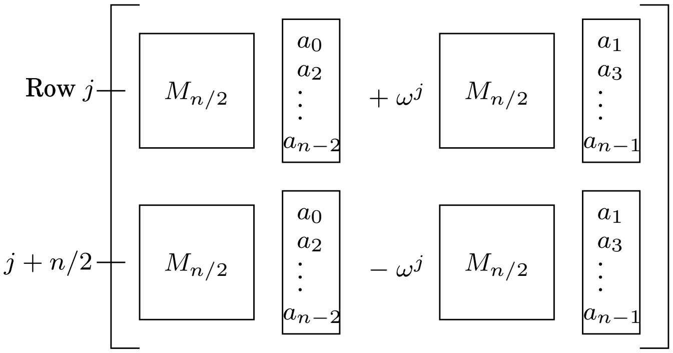
In the second step, we have simplified entries in the bottom half of the matrix using $\omega^{n/2} = -1$ and $\omega^n = 1$. Notice that the top left $n/2 \times n/2$ submatrix is $M_{n/2}(\omega^2)$, as is the one on the

Figure 2.9 The fast Fourier transform

```
function FFT($a, \omega$)
Input:  An array $a = (a_0, a_1, \ldots, a_{n-1})$, for $n$ a power of $2$
        A primitive $n$th root of unity, $\omega$
Output: $M_n(\omega)\, a$

if $\omega = 1$:  return $a$
$(s_0, s_1, \ldots, s_{n/2-1}) = \texttt{FFT}((a_0, a_2, \ldots, a_{n-2}), \omega^2)$
$(s'_0, s'_1, \ldots, s'_{n/2-1}) = \texttt{FFT}((a_1, a_3, \ldots, a_{n-1}), \omega^2)$
for $j = 0$ to $n/2 - 1$:
    $r_j = s_j + \omega^j s'_j$
    $r_{j+n/2} = s_j - \omega^j s'_j$
return $(r_0, r_1, \ldots, r_{n-1})$
```

bottom left. And the top and bottom right submatrices are almost the same as $M_{n/2}(\omega^2)$, but with their jth rows multiplied through by ω^j and $-\omega^j$, respectively. Therefore the final product is the vector


Row j
$j + n/2$
$M_{n/2}$
a_0 a_2 $\vdots$ a_{n-2}
$+\,\omega^j$
$M_{n/2}$
a_1 a_3 $\vdots$ a_{n-1}
$M_{n/2}$
a_0 a_2 $\vdots$ a_{n-2}
$-\,\omega^j$
$M_{n/2}$
a_1 a_3 $\vdots$ a_{n-1}

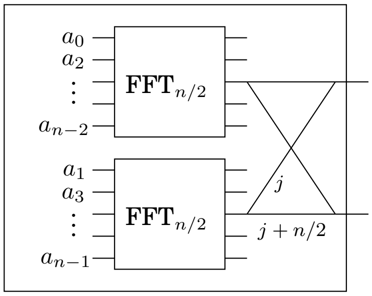
In short, the product of $M_n(\omega)$ with vector $(a_0, \ldots, a_{n-1})$, a size-n problem, can be expressed in terms of two size-$n/2$ problems: the product of $M_{n/2}(\omega^2)$ with $(a_0, a_2, \ldots, a_{n-2})$ and with $(a_1, a_3, \ldots, a_{n-1})$. This divide-and-conquer strategy leads to the definitive FFT algorithm of Figure 2.9, whose running time is $T(n) = 2T(n/2) + O(n) = O(n \log n)$.

The fast Fourier transform unraveled

Throughout all our discussions so far, the fast Fourier transform has remained tightly cocooned within a divide-and-conquer formalism. To fully expose its structure, we now unravel the recursion.

The divide-and-conquer step of the FFT can be drawn as a very simple circuit. Here is how a problem of size n is reduced to two subproblems of size $n/2$ (for clarity, one pair of outputs $(j, j + n/2)$ is singled out):

FFT$_n$ (input: $a_0, \ldots, a_{n-1}$, output: $r_0, \ldots, r_{n-1}$)



We're using a particular shorthand: the edges are wires carrying complex numbers from left to right. A weight of j means "multiply the number on this wire by ω^j." And when two wires come into a junction from the left, the numbers they are carrying get added up. So the two outputs depicted are executing the commands

$$\begin{aligned} r_j &= s_j + \omega^j s'_j \\ r_{j+n/2} &= s_j - \omega^j s'_j \end{aligned}$$

from the FFT algorithm (Figure 2.9), via a pattern of wires known as a *butterfly*: ⧖.

Unraveling the FFT circuit completely for $n = 8$ elements, we get Figure 10.4. Notice the following.

1. For n inputs there are $\log_2 n$ levels, each with n nodes, for a total of $n \log n$ operations.

2. The inputs are arranged in a peculiar order: $0, 4, 2, 6, 1, 5, 3, 7$.

Why? Recall that at the top level of recursion, we first bring up the even coefficients of the input and then move on to the odd ones. Then at the next level, the even coefficients of this first group (which therefore are multiples of 4, or equivalently, have zero as their two least significant bits) are brought up, and so on. To put it otherwise, the inputs are arranged by increasing *last* bit of the binary representation of their index, resolving ties by looking at the next more significant bit(s). The resulting order in binary, $000, 100, 010, 110, 001, 101, 011, 111$, is the same as the natural one, $000, 001, 010, 011, 100, 101, 110, 111$ *except the bits are mirrored!*

3. There is a unique path between each input a_j and each output $A(\omega^k)$.

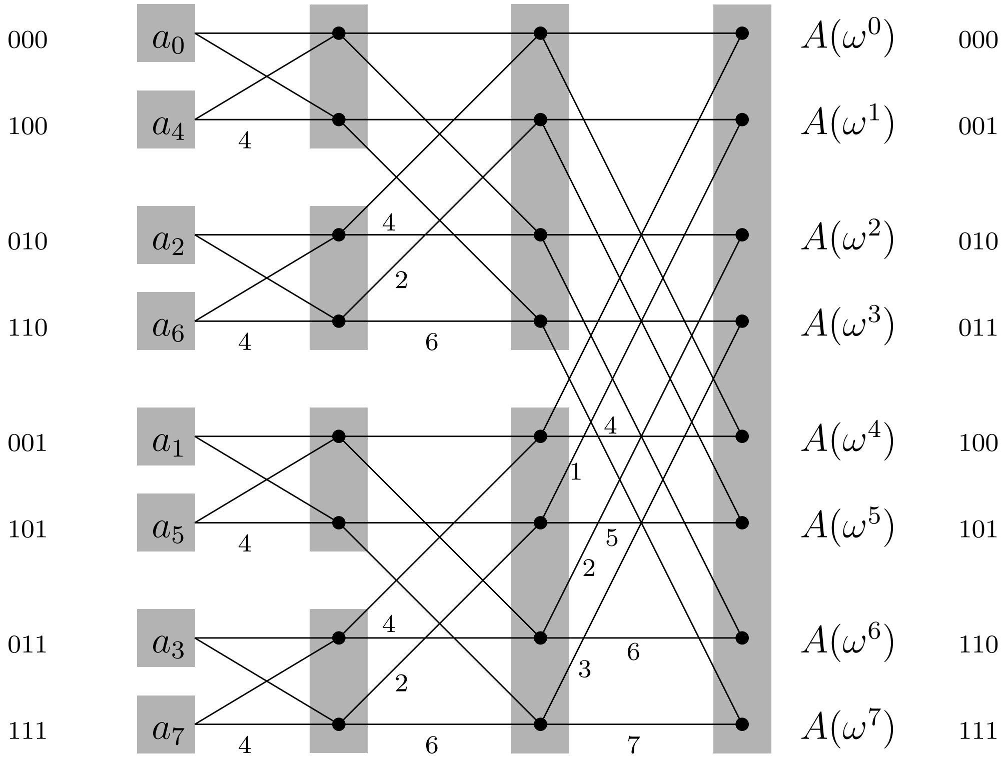
This path is most easily described using the binary representations of j and k (shown in Figure 10.4 for convenience). There are two edges out of each node, one going up (the 0-edge) and one going down (the 1-edge). To get to $A(\omega^k)$ from any input node, simply follow the edges specified in the bit representation of k, starting from the rightmost bit. (Can you similarly specify the path in the reverse direction?)

4. On the path between a_j and $A(\omega^k)$, the labels add up to $jk \bmod 8$.

Since $\omega^8 = 1$, this means that the contribution of input a_j to output $A(\omega^k)$ is $a_j\omega^{jk}$, and therefore the circuit computes correctly the values of polynomial $A(x)$.

5. And finally, notice that the FFT circuit is a natural for parallel computation and direct implementation in hardware.

Figure 2.10 The fast Fourier transform circuit.



The slow spread of a fast algorithm

In 1963, during a meeting of President Kennedy's scientific advisors, John Tukey, a mathematician from Princeton, explained to IBM's Dick Garwin a fast method for computing Fourier transforms. Garwin listened carefully, because he was at the time working on ways to detect nuclear explosions from seismographic data, and Fourier transforms were the bottleneck of his method. When he went back to IBM, he asked John Cooley to implement Tukey's algorithm; they decided that a paper should be published so that the idea could not be patented.

Tukey was not very keen to write a paper on the subject, so Cooley took the initiative. And this is how one of the most famous and most cited scientific papers was published in 1965, co-authored by Cooley and Tukey. The reason Tukey was reluctant to publish the FFT was not secretiveness or pursuit of profit via patents. He just felt that this was a simple observation that was probably already known. This was typical of the period: back then (and for some time later) algorithms were considered second-class mathematical objects, devoid of depth and elegance, and unworthy of serious attention.

But Tukey was right about one thing: it was later discovered that British engineers had used the FFT for hand calculations during the late 1930s. And—to end this chapter with the same great mathematician who started it—a paper by Gauss in the early 1800s on (what else?) interpolation contained essentially the same idea in it! Gauss's paper had remained a secret for so long because it was protected by an old-fashioned cryptographic technique: like most scientific papers of its era, it was written in Latin.

Exercises

2.1. Use the divide-and-conquer integer multiplication algorithm to multiply the two binary integers 10011011 and 10111010.

2.2. Show that for any positive integers n and any base b, there must some power of b lying in the range $[n, bn]$.

2.3. Section 2.2 describes a method for solving recurrence relations which is based on analyzing the recursion tree and deriving a formula for the work done at each level. Another (closely related) method is to expand out the recurrence a few times, until a pattern emerges. For instance, let's start with the familiar $T(n) = 2T(n/2) + O(n)$. Think of $O(n)$ as being $\leq cn$ for some constant c, so: $T(n) \leq 2T(n/2) + cn$. By repeatedly applying this rule, we can bound $T(n)$ in terms of $T(n/2)$, then $T(n/4)$, then $T(n/8)$, and so on, at each step getting closer to the value of $T(\cdot)$ we do know, namely $T(1) = O(1)$.

$$\begin{array}{rcl} T(n) & \leq & 2T(n/2) + cn \\ & \leq & 2[2T(n/4) + cn/2] + cn \quad = \quad 4T(n/4) + 2cn \\ & \leq & 4[2T(n/8) + cn/4] + 2cn \quad = \quad 8T(n/8) + 3cn \\ & \leq & 8[2T(n/16) + cn/8] + 3cn \quad = \quad 16T(n/16) + 4cn \\ & \vdots & \end{array}$$

A pattern is emerging... the general term is

$$T(n) \leq 2^k T(n/2^k) + kcn.$$

Plugging in $k = \log_2 n$, we get $T(n) \leq nT(1) + cn \log_2 n = O(n \log n)$.

(a) Do the same thing for the recurrence $T(n) = 3T(n/2) + O(n)$. What is the general kth term in this case? And what value of k should be plugged in to get the answer?

(b) Now try the recurrence $T(n) = T(n-1) + O(1)$, a case which is not covered by the master theorem. Can you solve this too?

2.4. Suppose you are choosing between the following three algorithms:

- Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.
- Algorithm B solves problems of size n by recursively solving two subproblems of size $n-1$ and then combining the solutions in constant time.
- Algorithm C solves problems of size n by dividing them into nine subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.

What are the running times of each of these algorithms (in big-O notation), and which would you choose?

2.5. Solve the following recurrence relations and give a Θ bound for each of them.

(a) $T(n) = 2T(n/3) + 1$

(b) $T(n) = 5T(n/4) + n$

(c) $T(n) = 7T(n/7) + n$

(d) $T(n) = 9T(n/3) + n^2$

(e) $T(n) = 8T(n/2) + n^3$

(f) $T(n) = 49T(n/25) + n^{3/2} \log n$

(g) $T(n) = T(n-1) + 2$

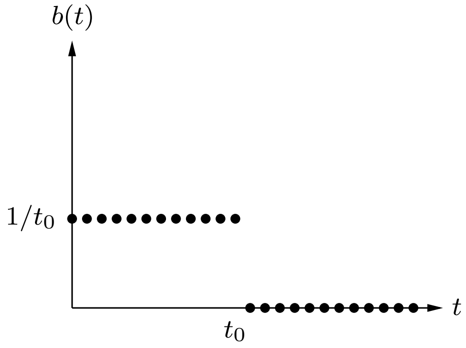
(h) $T(n) = T(n-1) + n^c$, where $c \geq 1$ is a constant

(i) $T(n) = T(n-1) + c^n$, where $c > 1$ is some constant

(j) $T(n) = 2T(n-1) + 1$

(k) $T(n) = T(\sqrt{n}) + 1$

2.6. A linear, time-invariant system has the following impulse response:



(a) Describe in words the effect of this system.

(b) What is the corresponding polynomial?

2.7. What is the sum of the nth roots of unity? What is their product if n is odd? If n is even?

2.8. Practice with the fast Fourier transform.

(a) What is the FFT of $(1, 0, 0, 0)$? What is the appropriate value of ω in this case? And of which sequence is $(1, 0, 0, 0)$ the FFT?

(b) Repeat for $(1, 0, 1, -1)$.

2.9. Practice with polynomial multiplication by FFT.

(a) Suppose that you want to multiply the two polynomials $x + 1$ and $x^2 + 1$ using the FFT. Choose an appropriate power of two, find the FFT of the two sequences, multiply the results componentwise, and compute the inverse FFT to get the final result.

(b) Repeat for the pair of polynomials $1 + x + 2x^2$ and $2 + 3x$.

2.10. Find the unique polynomial of degree 4 that takes on values $p(1) = 2$, $p(2) = 1$, $p(3) = 0$, $p(4) = 4$, and $p(5) = 0$. Write your answer in the coefficient representation.

2.11. In justifying our matrix multiplication algorithm (Section 2.5), we claimed the following blockwise property: if X and Y are $n \times n$ matrices, and

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \quad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}.$$

where A, B, C, D, E, F, G, and H are $n/2 \times n/2$ submatrices, then the product XY can be expressed in terms of these blocks:

$$XY \;=\; \begin{bmatrix} A & B \\ C & D \end{bmatrix} \begin{bmatrix} E & F \\ G & H \end{bmatrix} \;=\; \begin{bmatrix} AE+BG & AF+BH \\ CE+DG & CF+DH \end{bmatrix}$$

Prove this property.

2.12. How many lines, as a function of n (in $\Theta(\cdot)$ form), does the following program print? Write a recurrence and solve it. You may assume n is a power of 2.

```
function f(n)
  if n > 1:
    print_line(‘‘still going’’)
    f(n/2)
    f(n/2)
```

2.13. A binary tree is *full* if all of its vertices have either zero or two children. Let B_n denote the number of full binary trees with n vertices.

(a) By drawing out all full binary trees with 3, 5, or 7 vertices, determine the exact values of B_3, B_5, and B_7. Why have we left out even numbers of vertices, like B_4?

(b) For general n, derive a recurrence relation for B_n.

(c) Show by induction that B_n is $\Omega(2^n)$.

2.14. You are given an array of n elements, and you notice that some of the elements are duplicates; that is, they appear more than once in the array. Show how to remove all duplicates from the array in time $O(n \log n)$.

2.15. In our median-finding algorithm (Section 2.4), a basic primitive is the `split` operation, which takes as input an array S and a value v and then divides S into three sets: the elements less than v, the elements equal to v, and the elements greater than v. Show how to implement this `split` operation *in place*, that is, without allocating new memory.

2.16. You are given an infinite array $A[\cdot]$ in which the first n cells contain integers in sorted order and the rest of the cells are filled with ∞. You are *not* given the value of n. Describe an algorithm that takes an integer x as input and finds a position in the array containing x, if such a position exists, in $O(\log n)$ time. (If you are disturbed by the fact that the array A has infinite length, assume instead that it is of length n, but that you don't know this length, and that the implementation of the array data type in your programming language returns the error message ∞ whenever elements $A[i]$ with $i > n$ are accessed.)

2.17. Given a sorted array of distinct integers $A[1, \ldots, n]$, you want to find out whether there is an index i for which $A[i] = i$. Give a divide-and-conquer algorithm that runs in time $O(\log n)$.

2.18. Consider the task of searching a sorted array $A[1 \ldots n]$ for a given element x: a task we usually perform by binary search in time $O(\log n)$. Show that any algorithm that accesses the array only via comparisons (that is, by asking questions of the form "is $A[i] \leq z$?"), must take $\Omega(\log n)$ steps.

2.19. *A k-way merge operation.* Suppose you have k sorted arrays, each with n elements, and you want to combine them into a single sorted array of kn elements.

(a) Here's one strategy: Using the `merge` procedure from Section 2.3, merge the first two arrays, then merge in the third, then merge in the fourth, and so on. What is the time complexity of this algorithm, in terms of k and n?

(b) Give a more efficient solution to this problem, using divide-and-conquer.

2.20. Show that any array of integers $x[1 \dots n]$ can be sorted in $O(n + M)$ time, where

$$M \;=\; \max_i x_i - \min_i x_i.$$

For small M, this is linear time: why doesn't the $\Omega(n \log n)$ lower bound apply in this case?

2.21. *Mean and median.* One of the most basic tasks in statistics is to summarize a set of observations $\{x_1, x_2, \dots, x_n\} \subseteq \mathbb{R}$ by a single number. Two popular choices for this summary statistic are:

- The median, which we'll call μ_1
- The mean, which we'll call μ_2

(a) Show that the median is the value of μ that minimizes the function

$$\sum_i |x_i - \mu|.$$

You can assume for simplicity that n is odd. (*Hint:* Show that for any $\mu \neq \mu_1$, the function decreases if you move μ either slightly to the left or slightly to the right.)

(b) Show that the mean is the value of μ that minimizes the function

$$\sum_i (x_i - \mu)^2.$$

One way to do this is by calculus. Another method is to prove that for any $\mu \in \mathbb{R}$,

$$\sum_i (x_i - \mu)^2 \;=\; \sum_i (x_i - \mu_2)^2 + n(\mu - \mu_2)^2.$$

Notice how the function for μ_2 penalizes points that are far from μ much more heavily than the function for μ_1. Thus μ_2 tries much harder to be close to *all* the observations. This might sound like a good thing at some level, but it is statistically undesirable because just a few outliers can severely throw off the estimate of μ_2. It is therefore sometimes said that μ_1 is a more robust estimator than μ_2. Worse than either of them, however, is μ_∞, the value of μ that minimizes the function

$$\max_i |x_i - \mu|.$$

(c) Show that μ_∞ can be computed in $O(n)$ time (assuming the numbers x_i are small enough that basic arithmetic operations on them take unit time).

2.22. You are given two sorted lists of size m and n. Give an $O(\log m + \log n)$ time algorithm for computing the kth smallest element in the union of the two lists.

2.23. An array $A[1 \dots n]$ is said to have a *majority element* if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is $A[i] > A[j]$?". (Think of the array elements as GIF files, say.) However you *can* answer questions of the form: "is $A[i] = A[j]$?" in constant time.

(a) Show how to solve this problem in $O(n \log n)$ time. (*Hint:* Split the array A into two arrays A_1 and A_2 of half the size. Does knowing the majority elements of A_1 and A_2 help you figure out the majority element of A? If so, you can use a divide-and-conquer approach.)

(b) Can you give a linear-time algorithm? (*Hint:* Here's another divide-and-conquer approach:

- Pair up the elements of A arbitrarily, to get $n/2$ pairs
- Look at each pair: if the two elements are different, discard both of them; if they are the same, keep just one of them

Show that after this procedure there are at most $n/2$ elements left, and that they have a majority element if and only if A does.)

2.24. On page 66 there is a high-level description of the quicksort algorithm.

(a) Write down the pseudocode for quicksort.

(b) Show that its *worst-case* running time on an array of size n is $\Theta(n^2)$.

(c) Show that its *expected* running time satisfies the recurrence relation

$$T(n) \leq O(n) + \frac{1}{n}\sum_{i=1}^{n-1}(T(i) + T(n-i)).$$

Then, show that the solution to this recurrence is $O(n \log n)$.

2.25. In Section 2.1 we described an algorithm that multiplies two n-bit binary integers x and y in time n^a, where $a = \log_2 3$. Call this procedure fastmultiply(x, y).

(a) We want to convert the decimal integer 10^n (a 1 followed by n zeros) into binary. Here is the algorithm (assume n is a power of 2):

```
function pwr2bin($n$)
  if $n = 1$:  return $1010_2$
  else:
    $z =$???
    return fastmultiply($z, z$)
```

Fill in the missing details. Then give a recurrence relation for the running time of the algorithm, and solve the recurrence.

(b) Next, we want to convert any decimal integer x with n digits (where n is a power of 2) into binary. The algorithm is the following:

```
function dec2bin($x$)
  if $n = 1$:  return binary[$x$]
  else:
    split $x$ into two decimal numbers $x_L$, $x_R$ with $n/2$ digits each
    return ???
```

Here binary$[\cdot]$ is a vector that contains the binary representation of all one-digit integers. That is, binary$[0] = 0_2$, binary$[1] = 1_2$, up to binary$[9] = 1001_2$. Assume that a lookup in binary takes $O(1)$ time.

Fill in the missing details. Once again, give a recurrence for the running time of the algorithm, and solve it.

2.26. Professor F. Lake tells his class that it is asymptotically faster to square an n-bit integer than to multiply two n-bit integers. Should they believe him?

2.27. The *square* of a matrix A is its product with itself, AA.

(a) Show that five multiplications are sufficient to compute the square of a 2×2 matrix.

(b) What is wrong with the following algorithm for computing the square of an $n \times n$ matrix?

> "Use a divide-and-conquer approach as in Strassen's algorithm, except that instead of getting 7 subproblems of size $n/2$, we now get 5 subproblems of size $n/2$ thanks to part (a). Using the same analysis as in Strassen's algorithm, we can conclude that the algorithm runs in time $O(n^{\log_2 5})$."

(c) In fact, squaring matrices is no easier than matrix multiplication. In this part, you will show that if $n \times n$ matrices can be squared in time $S(n) = O(n^c)$, then any two $n \times n$ matrices can be multiplied in time $O(n^c)$.

i. Given two $n \times n$ matrices A and B, show that the matrix $AB + BA$ can be computed in time $3S(n) + O(n^2)$.

ii. Given two $n \times n$ matrices X and Y, define the $2n \times 2n$ matrices A and B as follows:

$$A = \begin{bmatrix} X & 0 \\ 0 & 0 \end{bmatrix} \text{ and } B = \begin{bmatrix} 0 & Y \\ 0 & 0 \end{bmatrix}.$$

What is $AB + BA$, in terms of X and Y?

iii. Using (i) and (ii), argue that the product XY can be computed in time $3S(2n) + O(n^2)$. Conclude that matrix multiplication takes time $O(n^c)$.

2.28. The *Hadamard matrices* $H_0, H_1, H_2, \ldots$ are defined as follows:

- H_0 is the 1×1 matrix $\begin{bmatrix} 1 \end{bmatrix}$
- For $k > 0$, H_k is the $2^k \times 2^k$ matrix

$$H_k = \left[\begin{array}{c|c} H_{k-1} & H_{k-1} \\ \hline H_{k-1} & -H_{k-1} \end{array}\right]$$

Show that if v is a column vector of length $n = 2^k$, then the matrix-vector product $H_k v$ can be calculated using $O(n \log n)$ operations. Assume that all the numbers involved are small enough that basic arithmetic operations like addition and multiplication take unit time.

2.29. Suppose we want to evaluate the polynomial $p(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$ at point x.

(a) Show that the following simple routine, known as *Horner's rule*, does the job and leaves the answer in z.

```
$z = a_n$
for $i = n-1$ downto 0:
    $z = zx + a_i$
```

(b) How many additions and multiplications does this routine use, as a function of n? Can you find a polynomial for which an alternative method is substantially better?

2.30. This problem illustrates how to do the Fourier Transform (FT) in modular arithmetic, for example, modulo 7.

(a) There is a number ω such that all the powers $\omega, \omega^2, \ldots, \omega^6$ are distinct (modulo 7). Find this ω, and show that $\omega + \omega^2 + \cdots + \omega^6 = 0$. (Interestingly, for any prime modulus there is such a number.)

(b) Using the matrix form of the FT, produce the transform of the sequence $(0, 1, 1, 1, 5, 2)$ modulo 7; that is, multiply this vector by the matrix $M_6(\omega)$, for the value of ω you found earlier. In the matrix multiplication, all calculations should be performed modulo 7.

(c) Write down the matrix necessary to perform the inverse FT. Show that multiplying by this matrix returns the original sequence. (Again all arithmetic should be performed modulo 7.)

(d) Now show how to multiply the polynomials $x^2 + x + 1$ and $x^3 + 2x - 1$ using the FT modulo 7.

2.31. In Section 1.2.3, we studied Euclid's algorithm for computing the *greatest common divisor* (gcd) of two positive integers: the largest integer which divides them both. Here we will look at an alternative algorithm based on divide-and-conquer.

(a) Show that the following rule is true.

$$\gcd(a, b) = \begin{cases} 2\gcd(a/2, b/2) & \text{if } a, b \text{ are even} \\ \gcd(a, b/2) & \text{if } a \text{ is odd, } b \text{ is even} \\ \gcd((a-b)/2, b) & \text{if } a, b \text{ are odd} \end{cases}$$

(b) Give an efficient divide-and-conquer algorithm for greatest common divisor.

(c) How does the efficiency of your algorithm compare to Euclid's algorithm if a and b are n-bit integers? (In particular, since n might be large you cannot assume that basic arithmetic operations like addition take constant time.)

2.32. In this problem we will develop a divide-and-conquer algorithm for the following geometric task.

CLOSEST PAIR
Input: A set of points in the plane, $\{p_1 = (x_1, y_1), p_2 = (x_2, y_2), \ldots, p_n = (x_n, y_n)\}$
Output: The closest pair of points: that is, the pair $p_i \neq p_j$ for which the distance between p_i and p_j, that is,

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2},$$

is minimized.

For simplicity, assume that n is a power of two, and that all the x-coordinates x_i are distinct, as are the y-coordinates.

Here's a high-level overview of the algorithm:

- Find a value x for which exactly half the points have $x_i < x$, and half have $x_i > x$. On this basis, split the points into two groups, L and R.
- Recursively find the closest pair in L and in R. Say these pairs are $p_L, q_L \in L$ and $p_R, q_R \in R$, with distances d_L and d_R respectively. Let d be the smaller of these two distances.
- It remains to be seen whether there is a point in L and a point in R that are less than distance d apart from each other. To this end, discard all points with $x_i < x - d$ or $x_i > x + d$ and sort the remaining points by y-coordinate.
- Now, go through this sorted list, and for each point, compute its distance to the *seven* subsequent points in the list. Let p_M, q_M be the closest pair found in this way.

- The answer is one of the three pairs $\{p_L, q_L\}$, $\{p_R, q_R\}$, $\{p_M, q_M\}$, whichever is closest.

(a) In order to prove the correctness of this algorithm, start by showing the following property: any square of size $d \times d$ in the plane contains at most four points of L.

(b) Now show that the algorithm is correct. The only case which needs careful consideration is when the closest pair is split between L and R.

(c) Write down the pseudocode for the algorithm, and show that its running time is given by the recurrence:

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

Show that the solution to this recurrence is $O(n \log^2 n)$.

(d) Can you bring the running time down to $O(n \log n)$?