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

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

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

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

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:

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

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, 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

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

[1] Actually, the recurrence should read

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

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 ⌈n/2⌉, rightmost ⌊n/2⌋ bits of x
y_L, y_R = leftmost ⌈n/2⌉, rightmost ⌊n/2⌋ 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 × 2^n + (P_3 − P_1 − P_2) × 2^(n/2) + P_2
```

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.

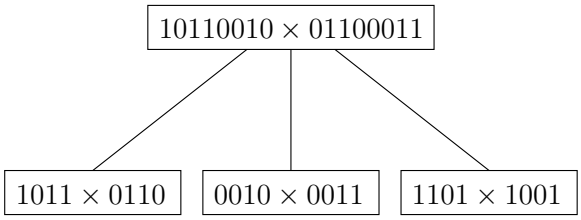
Finally, the eternal question: *Can we do better?* It turns out that even faster algorithms for multiplying numbers exist, based on another important divide-and-conquer algorithm: the fast Fourier transform, to be explained in Section 2.6.

2.2 Recurrence relations

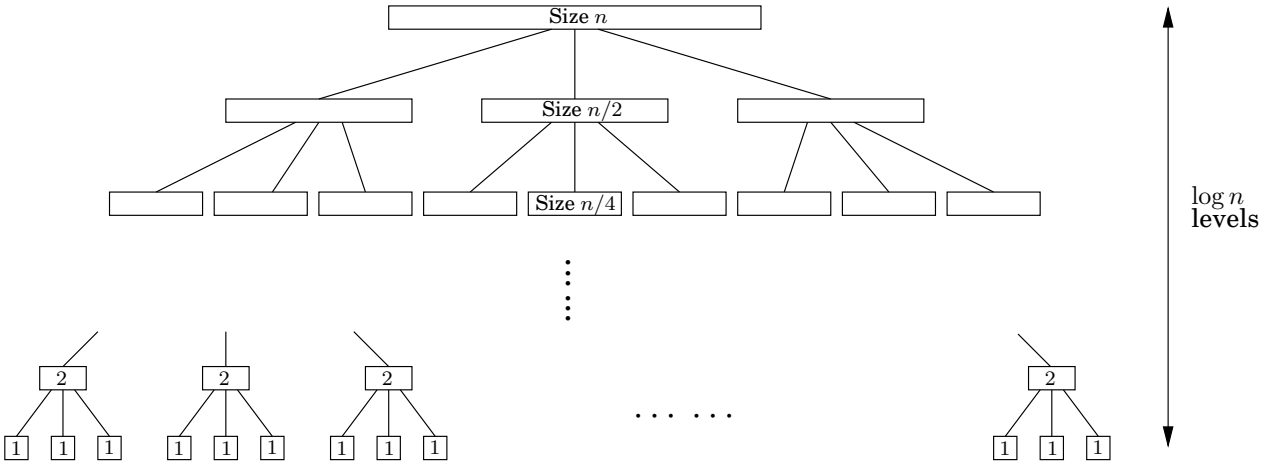
Divide-and-conquer algorithms often follow a generic pattern: they tackle a problem of size n by recursively solving, say, a subproblems of size n/b and then combining these answers in $O(n^d)$ time, for some $a, b, d > 0$ (in the multiplication algorithm, $a = 3$, $b = 2$, and $d = 1$). Their running 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.

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

(a)



(b)



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

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

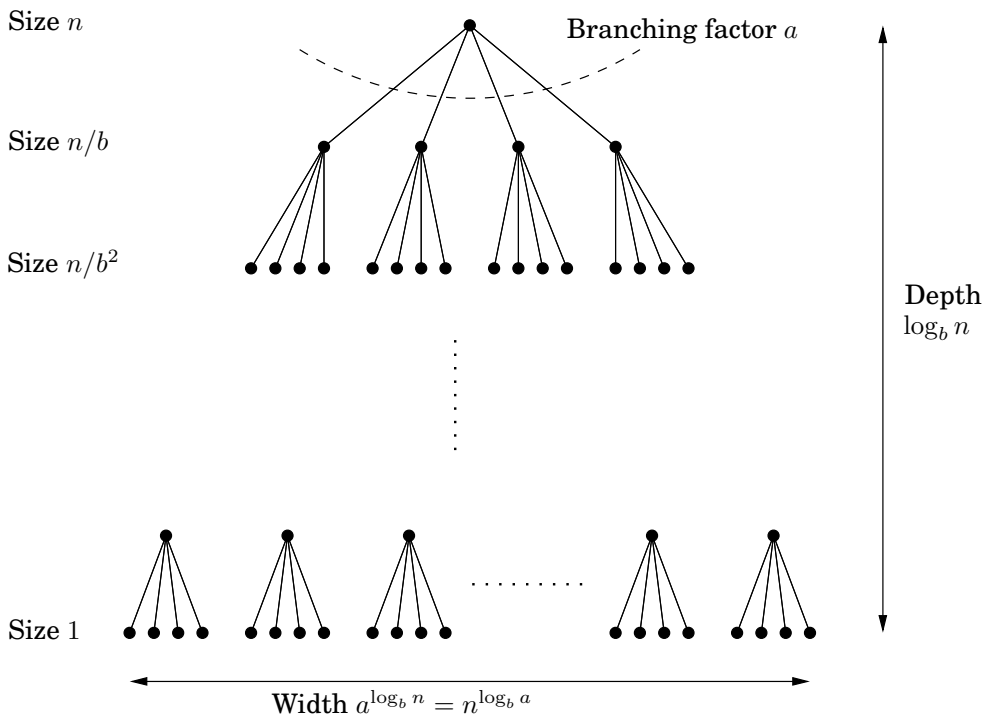
This single theorem tells us the running times of most of the divide-and-conquer procedures we are likely to use.

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

[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.



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...n])
Input:  An array of numbers a[1...n]
Output:  A sorted version of this array

if n > 1:
  return merge(mergesort(a[1...⌊n/2⌋]), mergesort(a[⌊n/2⌋ + 1...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.

```
function merge(x[1...k], y[1...l])
if k = 0:  return y[1...l]
if l = 0:  return x[1...k]
if x[1] ≤ y[1]:
  return x[1] ∘ merge(x[2...k], y[1...l])
else:
  return y[1] ∘ merge(x[1...k], y[2...l])
```

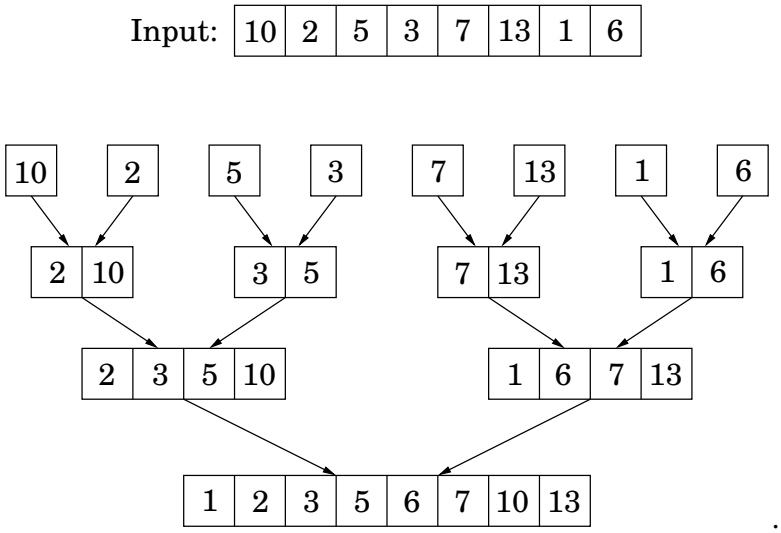
Here $\circ$ denotes concatenation. This `merge` procedure does a constant amount of work per recursive call (provided the required array space is allocated in advance), for a total running time of $O(k + l)$. Thus `merge`'s are linear, and the overall time taken by `mergesort` is

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

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

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



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.

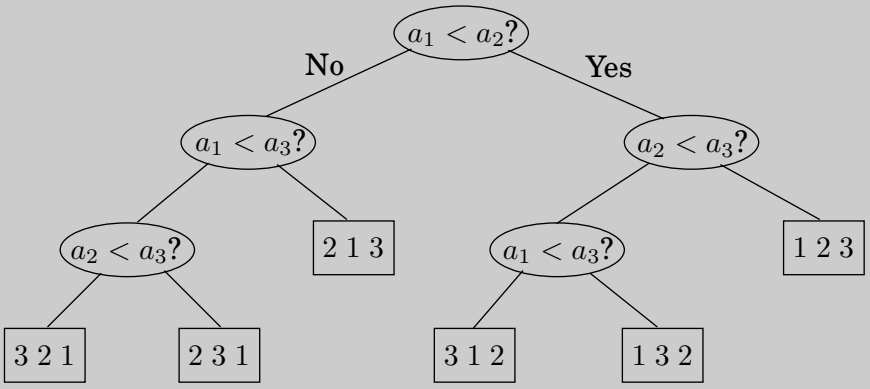
In the following pseudocode, the primitive operation `inject` adds an element to the end of the queue while `eject` removes and returns the element at the front of the queue.

```
function iterative-mergesort(a[1...n])
Input:  elements a_1, a_2, ..., 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 $\{1, 2, \ldots, n\}$. 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 :

2	36	5	21	8	13	11	20	5	4	1

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

S_L :

2	4	1

S_v :

5	5

S_R :

36	21	8	13	11	20

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, selection$(S, 8)$ = 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:

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

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.

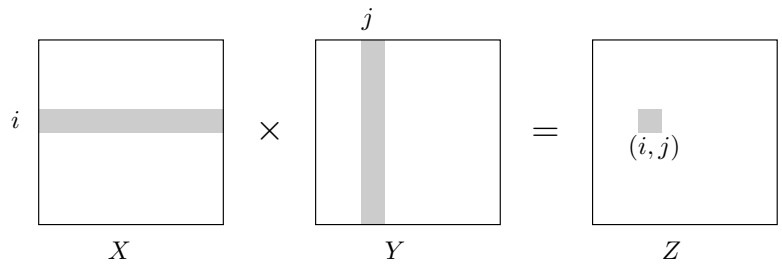
Quicksort is a sorting algorithm that splits the array in exactly the same way as the median algorithm; and once the subarrays are sorted, by two recursive calls, there is nothing more to do. Its worst-case performance is $\Theta(n^2)$, like that of median-finding. But it can be proved (Exercise 2.24) 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

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

The new running time is

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

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