Divide-and-conquer algorithms[1]

1 Multiplication

The mathematician Gauss 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

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

This speeds up the computation, but only by a constant factor, which to our big-O way of thinking is negligible. Can something more substantial be salvaged from Gauss' observation?

The seemingly modest improvement in computation time becomes very significant if this same trick is 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. As a first step towards multiplying them, split each of them into their left and right halves, which are $n/2$ bits long:

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

For instance, if $x = 1011_2$ then $x_L = 10_2$ and $x_R = 11_2$. The product is

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

We will compute xy via the expression on the right-hand side. The additions take linear time, as do the multiplications by powers of two (which are 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. If $T(n)$ denotes the time taken to multiply n-bit numbers by this method, then $T(n)$ can be written as a *recurrence relation*,

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

We will soon examine general strategies for solving such equations. In the meantime, this particular one works out to $O(n^2)$, which is disappointing because it is no better than the traditional grade-school multiplication technique. So we would like to speed up our recursive algorithm somehow, and now Gauss' 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 improved running time is

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

Figure 1.1 A divide-and-conquer algorithm for integer multiplication.

function multiply (x, y)

```
Input:  Two n-bit numbers x and y.
Output:  Their product.

if n = 1:  return xy

x_L, x_R = leftmost, rightmost ⌈n/2⌉ bits of x
y_L, y_R = leftmost, 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_3
```

At first glance this seems only a constant factor better than the previous attempt. But the improvement occurs at every level of the recursion, and this compounding effect leads to a dramatically lower time bound of $O(n^{1.59})$.

The algorithmic strategy we have been using (see Figure 1.1) is called *divide-and-conquer*: it tackles a problem by selecting subproblems, recursively solving them, and then gluing together these partial answers. All the work is done in the selecting and gluing, and in the base case of the recursion.

The running time of our algorithm follows from its pattern of recursive calls, which form a tree structure, as in Figure 1.2. Let's try to understand the shape of this tree. At each successive level of recursion the subproblems get halved in size. At the $(\log_2 n)^{th}$ level, the subproblems get down to size one, and so the recursion ends. Therefore, the height of the tree is $\log_2 n$. The branching factor is three – 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 $n/2^k$.

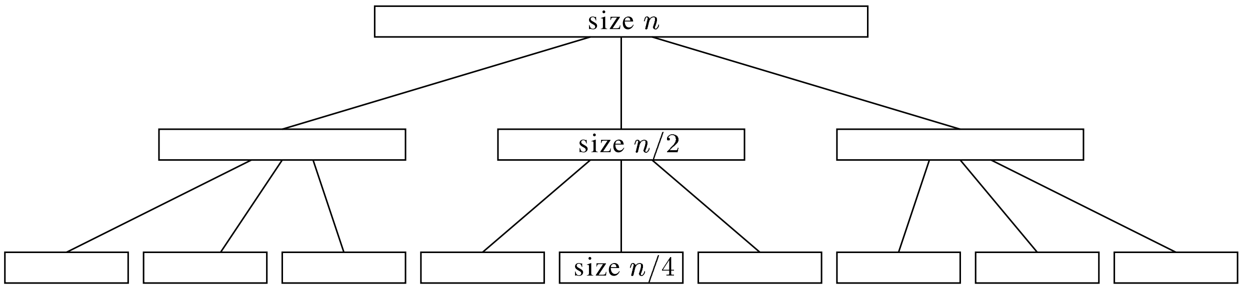
For each subproblem, a linear amount of work is done in selecting further subproblems and gluing together answers. Therefore the total time spent at depth k in the tree is

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

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 $O(n^{\log_2 3})$ (check!). 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. Therefore the overall running time is $O(n^{\log_2 3})$, which is about $O(n^{1.59})$.

In the absence of Gauss' trick, the recursion tree would have the same height, but with a branching factor of four. 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

Figure 1.2 The first few levels of recursion of divide-and-conquer integer multiplication.



have a big impact on running time.

One practical note: it generally does not make sense to recurse all the way down to one 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.

2 Recurrence relations

A large number of divide-and-conquer algorithms conform to a generic pattern: they tackle a problem of size n by recursively solving a subproblems of size n/b and then combining these answers in $O(n^d)$ time, for some $a, b, d > 0$. Their running time is therefore $T(n) = aT(\lceil n/b \rceil) + O(n^d)$. We now find a closed-form solution to this general recurrence, so that we no longer have to solve it explicitly in each new instance.

Claim. If $T(n) \leq aT(\lceil n/b \rceil) + O(n^d)$ for some positive constants a, b, d, then

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

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

To prove the claim, let's start by assuming for convenience that n is a power of b; this will not influence the final bound and will allow us to ignore the rounding effect in $\lceil n/b \rceil$. 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 k^{th} level of the tree is made up of a^k subproblems, each of size n/b^k. 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. To determine the sum of the series, there are three cases we need to consider. If the ratio is less than one, then the series is decreasing, in which case the first term, $O(n^d)$, is dominant. If the ratio is more than one, the series is increasing and the last term,

$O(n^d(a/b^d)^{\log_b n}) = O(n^{\log_b a})$, is dominant. Finally, it could be that the ratio is exactly one, in which case all $O(\log n)$ terms of the series are equal to $O(n^d)$.

These cases translate directly into the three contingencies of the theorem.

3 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

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

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

The formula above implies an $O(n^3)$ algorithm for matrix multiplication: there are n^2 entries to be computed, and each takes linear time. For quite a while, this was widely believed to be the best running time possible, and it was even proved that no algorithm which used just additions and multiplications could do better. It was therefore a source of great excitement when in 1969, 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.

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

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

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

which comes out to $O(n^3)$, the same as for the default algorithm. However, an improvement in the time bound is possible, and as with integer multiplication, it relies upon algebraic tricks. It turns out that XY can be computed from just *seven* subproblems.

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

This translates into a running time of

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

which by the result of the previous section is $O(n^{\log_2 7}) \approx O(n^{2.81})$.

4 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])
```

This kind of tail recursion can be unraveled into a purely iterative algorithm, as shown in Figure 4.1. It performs a constant number of operations for each element of the combined array z, and therefore takes linear time, $O(k+l)$, in all. The overall running time of mergesort is then

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

which works out to $O(n \log n)$.

5 Medians

The *median* of a list of numbers is its 50th percentile: half the numbers are bigger than it, and half are smaller. In other words, it is the middle element when the numbers are arranged in order. For instance, the median of $[45, 1, 10, 30, 25]$ is 25. If the list has even length, there are two choices for the middle element, and the median is defined to be their average.

The purpose of the median is to summarize a set of numbers by a single, typical value. The *mean*, or average, is also commonly used for this, but it has the tremendous disadvantage that

Figure 4.1 The *merge* procedure.

```
function merge(x[1...k], y[1...l])
Input:  Two sorted arrays, x[1...k] and y[1...l]
Output:  A sorted array z[1...k+l] containing the combined elements of x,y

p_x = p_y = 1 // Positions in the two input arrays
for p_z = 1 to k+l:
  if p_y > l or x[p_x] ≤ y[p_y]:
    z[p_z] = x[p_x]
    p_x = p_x + 1
  else
    z[p_z] = y[p_y]
    p_y = p_y + 1

return z
```

it can be completely thrown off by a single large or small number. For instance, the mean of a list of a hundred 1's is (rightly) 1, as is the median. However, if just one of these numbers gets accidentally changed to 10000, the mean shoots up above 100, while the median is unaffected.

Computing the median of n numbers is easy: just sort them. The only problem 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.

Often a problem becomes clearer, and easier to solve, when we consider a more general version of it. In this case, the generalization we will consider is *selection*.

SELECTION

Input: A list of numbers S; an integer k

Output: The k^{th} smallest element of S

For instance, if $k = 1$, the minimum of S is sought, whereas if $k = |S|$, it is the maximum. By setting k appropriately, all the percentiles of S can be found.

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 $S_{<v}$, $S_{=v}$, and $S_{>v}$ respectively. By checking k against the sizes of these subarrays, we can quickly determine which of them holds the desired element:

$$\text{select}(S,k) = \begin{cases} \text{select}(S_{<v},k) & \text{if } k \le |S_{<v}| \\ v & \text{if } |S_{<v}| < k \le |S_{<v}| + |S_{=v}| \\ \text{select}(S_{>v}, k - |S_{<v}| - |S_{=v}|) & \text{if } k > |S_{<v}| + |S_{=v}|. \end{cases}$$

The three sublists $S_{<v}, S_{=v}, S_{>v}$ can be computed from S in linear time, and in fact this computation can be done *in place* – that is, without allocating new memory for them – using three pointers (can you figure out this neat trick?). Once they are obtained, exactly one of the three scenarios can hold, and so the original array S effectively shrinks to one of size at most $\max\{|S_{<v}|, |S_{>v}|\}$.

The choice of v is crucial. It should be picked quickly, and it should shrink the array substantially, the ideal situation being $|S_{<v}| \approx |S_{>v}| \approx \frac{1}{2}|S|$. 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 what we are trying to do in the first place! Instead, our method of choosing v is much simpler: we just pick it *randomly* from S.

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 smallest element of the array (or the largest element), and thereby shrink the array by only one element each time. This *worst case* scenario takes $O(n^2)$ steps, but is extremely unlikely to occur. Equally unlikely is the *best* possible case, in which each randomly chosen v just happens to be the median, so that as above the running time is $O(n)$. Where, in this spectrum from $O(n)$ to $O(n^2)$, 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 anywhere in 25^{th} to 75^{th} percentile of the array that it is chosen from. The following property follows from the definition of percentile.

> **Property.** When v is chosen randomly from an array, it has a probability 1/2 of being good.

A good v causes the array to shrink to at most 3/4 of its size (why?). This is very promising, but how many times, on average, do we need to pick v before we get a good one? There is an equivalent question which might be more familiar: "on average how many times must we toss a fair coin before getting heads?". Let E be the expected number of tosses. We certainly need at least one toss, and if it's heads, we're done. Otherwise (and this occurs with probability 1/2), we need to repeat. In other words, $E = 1 + \frac{1}{2}E$, so $E = 2$.

Therefore, after every two recursive calls on average, the array will shrink to 3/4 of its size. Let $T(n)$ be the *expected* running time. Then

$$T(n) \leq T(3n/4) + O(2n),$$

which means $T(n)$ is linear. In short, on *any* input, our algorithm returns the correct answer in an average of $O(n)$ steps.