Divide-and Conquer & Fast Integer Multiplication

1 Algorithm Design Paradigms

Every algorithmic situation (every computational problem) is different, and there are no hard and fast algorithm design rules that work all the time. But the vast majority of algorithms can be categorized with respect to the *concept of progress* they utilize.

In an algorithm you want to make sure that, after each execution of a loop, or after each recursive call, some kind of *progress* has been made. In the absence of such assurance, you cannot be sure that your algorithm will even terminate. For example, in depth-search search, after each recursive call you know that another node has been visited. And in Dijkstra's algorithm you know that, after each execution of the main loop, the correct distance to another node has been found. And so on.

Divide-and-Conquer Algorithms: These algorithms have the following outline: To solve a problem, divide it into subproblems. Recursively solve the subproblems. Finally glue the resulting solutions together to obtain the solution to the original problem. Progress here is measured by how much smaller the subproblems are compared to the original problem.

You have already seen an example of a divide and conquer algorithm in cs61B: mergesort. The idea behind mergesort is to take a list, *divide* it into two smaller sublists, *conquer* each sublist by sorting it, and then *combine* the two solutions for the subproblems into a single solution. These three basic steps – divide, conquer, and combine – lie behind most divide and conquer algorithms.

With mergesort, we kept dividing the list into halves until there was just one element left. In general, we may divide the problem into smaller problems in any convenient fashion. Also, in practice it may not be best to keep dividing until the instances are completely trivial. Instead, it may be wise to divide until the instances are reasonably small, and then apply an algorithm that is faster on small instances. For example, with mergesort, it might be best to divide lists until there are only four elements, and then sort these small lists quickly by other means. We will consider these issues in some of the applications we consider.

2 Maximum/minimum

Suppose we wish to find the minimum *and* maximum items in a list of numbers. How many comparisons does it take?

A natural approach is to try a divide and conquer algorithm. Split the list into two sublists of equal size. (Assume that the initial list size is a power of two.) Find the maxima and minimum of the sublists. Two more comparisons then suffice to find the maximum and minimum of the list.

Hence, if $T(n)$ is the number of comparisons, then $T(n) = 2T(n/2) + 2$. Also, clearly $T(1) = 0$. By induction we find $T(n) = (3n/2) - 2$.

3 Integer Multiplication

The standard multiplication algorithm takes time $\Theta(n^2)$ to multiply together two n digit numbers. This algorithm is so natural that we may think that no algorithm could be better. Here, we will show that better algorithms exist (at least in terms of asymptotic behavior).

Imagine splitting each number x and y into two parts: $x = 2^{n/2}a + b, y = 2^{n/2}c + d$. Then:

$$xy = 2^n ac + 2^{n/2}(ad + bc) + bd.$$

The additions and the multiplications by powers of 2 (which are just shifts!) can all be done in linear time. We have therefore reduced our multiplication into four smaller multiplications problems, so the recurrence for the time $T(n)$ to multiply two n-digit numbers becomes

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

Unfortunately, when we solve this recurrence, the running time is still $\Theta(n^2)$, so it seems that we have not gained anything.

The key thing to notice here is that four multiplications is too many. Can we somehow reduce it to three? It may not look like it is possible, but it is using a simple trick. The trick is that *we do not need to compute ad and bc separately; we only need their sum $ad + bc$.* Now note that

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

So if we calculate ac,bd, and $(a + b)(c + d)$, we can compute $ad + bc$ by the subtracting the first two terms from the third! Of course, we have to perform more additions, but since the bottleneck to speeding up the algorithm was the number of multiplications, that does not matter. The recurrence for $T(n)$ is now

$$T(n) = 3T(n/2) + \emptyset(n),$$

and we find that $T(n) = n^{\log_2 3} \approx n^1.59$, improving on the quadratic algorithm.

If one were to implement this algorithm, it would probably be best not to divide the numbers down to one digit. The conventional algorithm, because it uses fewer additions, is more efficient for small values of n. Moreover, on a computer, there would be no reason to continue dividing once the length n is so small that the multiplication can be done in one standard machine multiplication operation!

It also turns out that using a more complicated algorithm (based on a similar idea, but with each integer divided into many more parts) the asymptotic time for multiplication can be made very close to linear– that is the running time can be $\emptyset(n^{1+\epsilon})$ for any $\epsilon > 0$.

4 Medians

You are given a large array of integers and you wish to find its median. Should you sort?

There is a *linear-time* divide-and-conquer algorithm for finding medians. We shall here give a simple *randomized* version whose expected running time is linear. As in the case of quicksort, if the algorithm makes a bad sequence of coin-flips, then in the worst case its running time is $(\Theta(n^2))$.

In fact, we shall give an algorithm for solving a slightly more general problem: Selecting the kth smallest element in a given array A. Here is the algorithm:

```
algorithm select(A, k)
  choose a random element x of A.
  subdivide A into three arrays:
    B contains all elements y of A with y < x,
    C contains all elements y of A with y = x,
    D contains all elements y of A with y > x
  if k ≤ |B| then return select(B, k)
```

`else if` $k \leq |B| + |C|$ `then return` x
`else return select(`$D, k - |B| - |C|$`)`

In practice we want to implement this algorithm so that the arrays B, C, D are not new storage space, but they reuse array A. This can be accomplished by three pointers, and it is an interesting three-liner to write.

Also, notice that this algorithm has *one recursive call*, as at most one of the alternatives will hold. Thus, the running of the algorithm can be considered as a sequence of recursive calls on smaller and smaller arrays.

The worst-case running time of this algorithm is terrible: just consider the case where x is always the smallest element of A? Instead we will show that the expected time is linear.

To see this, we let $f(n)$ denote the number of steps executed by the algorithm before the array in the recursive call has at most $3n/4$ elements in it. The recurrence for the expected running time, $T(n)$, of the algorithm can now be written as:

$$T(n) = T(\frac{3}{4}n) + f(n).$$

To estimate $f(n)$ let us imagine sorting the array A, and then consider its "middle half", i.e. all elements excluding the lowest and the highest quartiles. How many recursive calls will it take before the random pivot element x is chosen from this middle half? In each recursive call the probability that x is chosen from the middle half is at least half ("at least" because the array becomes smaller and smaller) in each recursive call. So, the expected number of recursive calls before the array size is halved is at most:

$$\frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{8} \cdot 3 + \cdots$$

which is *two* (check it!). Since each recursive call takes linear time, say cn, we have shown that $f(n) \leq 2cn$. Thus the recurrence for the running time of the algorithm is:

$$T(n) = T(\frac{3n}{4}) + 2cn,$$

which has solution $T(n) = \Theta(n)$.

5 Solving Divide-and Conquer Recurrences

A typical divide-and-conquer recurrence is of the following form:

$$T(n) = a \cdot T(\frac{n}{b}) + n^d,$$

where we assume that $T(1) = 0$ —that is, there is nothing to do if we have just one item. For simplicity in the analysis, we further assume that n is a power of b: $n = b^k$, where of course k is $\log_b n$. This allows us to sweep under the rug the question "what happens if n is not exactly divisible by b" either in the first or subsequent recursive calls. We can always "round n up" to the next power of b to make sure this is the case (but, of course, in a practical implementation we would have to take care of the case in which the problem would have to be split in slightly unbalanced subproblems).

From the above equation, we derive by substitution $n \to \frac{n}{b}$

$$T(\frac{n}{b}) = a \cdot T(\frac{n}{b^2}) + \frac{1}{b^d} \cdot n^d$$

Thus, substituting this in the first equation we get:

$$T(n) = a^2 \cdot T(\frac{n}{b^2}) + n^d(1 + \frac{a}{b^d}).$$

Continuing with the substitutions we finally get, after k such substitutions,

$$T(n) = a^k \cdot T(\frac{n}{b^k}) + n^d(\sum_{i=1}^{k-1} \frac{(a}{b^d)^i}),$$

or, since $n = b^k$ and $T(1) = 0$,

$$T(n) = n^d(\sum_{i=1}^{k-1} \frac{(a}{b^d)^i}).$$

Thus, $T(n)$ *is a geometric sum*, with ratio $\frac{a}{b^d}$. And we know that a geometric sum is (within a small constant) either its first term, or its last term or its *number* of terms, depending on whether the ratio is less than 1, greater than one, or equal to one, respectively. In conclusion:

- If $a < b^d$, then $T(n) = \Theta(n^d)$.
- If $a = b^d$, then $T(n) = \Theta(n^d \log n)$.
- Finally, if $a > b^d$, then $T(n) = \Theta(n^{\log_b a}))$.

One final note: What happens if the recurrence equation is instead

$$T(n) = a \cdot T(\frac{n}{b}) + n^d \log^e n?$$

The answer is, the first two cases above are multiplied by $\log^e n$, but not the third.