1 Divide and Conquer (January 21)

1.1 MergeSort

Mergesort is one of the earliest algorithms proposed for sorting. According to Knuth, it was suggested by John von Neumann as early as 1945.

1. Divide A into two equal-sized subarrays $A[1..m]$ and $A[m+1..n]$, where $m = \lfloor n/2 \rfloor$.
2. Mergesort $A[1..m]$ and $A[m+1..n]$.
3. Merge $A[1..m]$ and $A[m+1..n]$ into a single sorted list.

The first step is completely trivial; we only need to compute the median index m. The two sorted arrays $A[1..m]$ and $A[m+1..n]$ can be merged together by a simple linear-time algorithm.

Here's a complete specification of the MERGESORT algorithm:

```
MERGE(A[1..n], m):
  i ← 1; j ← m+1
  for k ← 1 to n
    if j > n
      B[k] ← A[i]; i ← i+1
    else if i > m
      B[k] ← A[j]; j ← j+1
    else if A[i] < A[j]
      B[k] ← A[i]; i ← i+1
    else
      B[k] ← A[j]; j ← j+1
  for k ← 1 to n
    A[k] ← B[k]
```

```
MERGESORT(A[1..n]):
  if (n > 1)
    m ← ⌊n/2⌋
    MERGESORT(A[1..m])
    MERGESORT(A[m+1..n])
    MERGE(A[1..n], m)
```

To prove that the algorithm is correct, we use our old friend[1] *induction*. To prove MERGE correct, we use weak induction on the total size of the two subarrays $A[i..m]$ and $A[j..n]$ left to be merged into $B[k..n]$. The base case, where one subarray is empty, is straightforward; the algorithm just copies it into B. Otherwise, the smallest remaining element is either $A[i]$ or $A[j]$, since both subarrays are sorted, so $B[k]$ is assigned correctly. The remaining subarrays — either $A[i+1..m]$ and $A[j..n]$, or $A[i..m]$ and $A[j+1..n]$ — are merged correctly into $B[k+1..n]$ by the inductive hypothesis. This completes the proof.

Now the proof for MERGESORT is straightforward strong induction. The base cases $n \leq 1$ are trivial. Otherwise, by the inductive hypothesis, the two smaller subarrays $A[1..m]$ and $A[m+1..n]$ are sorted correctly, and by our earlier argument, merged into the correct sorted output.

What's the running time? Since we have a recursive algorithm, we're going to get a recurrence of some sort. MERGE clearly takes linear time, since it's a simple for-loop with constant work per iteration. We get the following recurrence for MERGESORT:

$$T(1) = O(1), \qquad T(n) = T\big(\lceil n/2 \rceil\big) + T\big(\lfloor n/2 \rfloor\big) + O(n).$$

[1] I'm being optimistic here!

1.2 Aside: Domain Transformations

Except for the floor and ceiling, this recurrence falls into case (b) of the Master Theorem [CLR, §4.3]. If we simply ignore the floor and ceiling, the Masster Theorem suggests the solution $T(n) = O(n \log n)$. We can easily check that this answer is correct using induction, but there is a simple method for solving recurrences like this directly, called *domain transformation*.

First we overestimate the time bound, once by pretending that the two subproblem sizes are equal, and again to eliminate the ceiling:

$$T(n) \leq 2T\left(\lceil n/2 \rceil\right) + O(n) \leq 2T(n/2 + 1) + O(n).$$

Now we define a new function $S(n) = T(n + \alpha)$, where α is a constant chosen so that $S(n)$ satisfies the Master-ready recurrence $S(n) \leq 2S(n/2) + O(n)$. To figure out the appropriate value for α, we compare two versions of the recurrence for $T(n + \alpha)$:

$$\begin{aligned} S(n) \leq 2S(n/2) + O(n) \quad &\Longrightarrow \quad T(n + \alpha) \leq 2T(n/2 + \alpha) + O(n) \\ T(n) \leq 2T(n/2 + 1) + O(n) \quad &\Longrightarrow \quad T(n + \alpha) \leq 2T((n + \alpha)/2 + 1) + O(n + \alpha) \end{aligned}$$

For these two recurrences to be equal, we need $n/2 + \alpha = (n + \alpha)/2 + 1$, which implies that $\alpha = 2$. The Master Theorem tells us that $S(n) = O(n \log n)$, so

$$T(n) = S(n - 2) = O((n - 2)\log(n - 2)) = O(n \log n).$$

1.3 QuickSort

Quicksort was discovered by Tony Hoare in 1962. In this algorithm, the hard work is splitting the array into subsets so that merging the final result is trivial.

1. Choose a *pivot* element from the array.
2. Split the array into three subarrays containing the items less than the pivot, the pivot itself, and the items bigger than the pivot.
3. Recursively quicksort the first and last subarray.

Here's a more formal specification of the algorithm:

```
PARTITION(A[1..n]):
  if (p ≠ n)
    swap A[p] ↔ A[n]
  i ← 0;  j ← n
  while (i < j)
    repeat i ← i + 1 until (i = j or A[i] ≥ A[n])
    repeat j ← j − 1 until (i = j or A[j] ≤ A[n])
    if (i < j)
      swap A[i] ↔ A[j]
  if (i ≠ n)
    swap A[i] ↔ A[n]
  return j
```

```
QUICKSORT(A[1..n]):
  if (n > 1)
    Choose a pivot element A[p]
    k ← PARTITION(A)
    QUICKSORT(A[1..i − 1])
    QUICKSORT(A[i + 1..n])
```

Just like with mergesort, the proof that this algorithm is correct is a double induction—one strong induction to prove that QUICKSORT correctly sorts assuming PARTITION is correct, and one weak induction to prove that PARTITION correctly partitions the array.

The analysis is also similar. PARTITION runs in $O(n)$ time: $j - i = n$ at the beginning, $j - i = 0$ at the end, and we do a constant amount of work each time we increment i or decrement j. For QUICKSORT, we get a recurrence that depends on k:

$$T(n) = T(k-1) + T(n-k) + O(n)$$

If we could choose the pivot to be the median element of the array A, we would have $k = \lceil n/2 \rceil$, the two subproblems would be as close to the same size as possible, the recurrence would become

$$T(n) = 2T\left(\lceil n/2 \rceil - 1\right) + T\left(\lfloor n/2 \rfloor\right) + O(n) \leq 2T(n/2) + O(n),$$

and we'd have $T(n) = O(n \log n)$ by the Master Theorem.

Unfortunately, although it is *theoretically possible* to locate the median of an unsorted array in linear time, the algorithm is incredibly complicated, and the hidden constant in the $O()$ notation is quite large. So in practice, programmers settle for something simple, like choosing the first or last element of the array. In this case, k can be anything from 1 to n, so we have

$$T(n) = \max_{1 \leq k \leq n} \left(T(k-1) + T(n-k) + O(n)\right)$$

In the worst case, the two subproblems are completely unbalanced—either $k = 1$ or $k = n$—and the recurrence becomes $T(n) \leq T(n-1) + O(n)$. The solution is $T(n) = O(n^2)$.

Intuitively, the pivot element will usually fall somewhere in the middle of the array, say between $n/10$ and $9n/10$. This suggests that the *average-case* running time is $O(n \log n)$. Although this intuition is correct, it's far from a *proof* that quicksort is usually efficient. I'll formalize this intuition about average cases in a later lecture.

1.4 The Pattern

Both mergesort and and quicksort follow the same general three-step pattern of divide and conquer algorithms:

1. **Split** the problem into several *smaller independent* subproblems.
2. **Recurse** to get a subsolution for each subproblem.
3. **Merge** the subsolutions together into the final solution.

If the size of any subproblem falls below some constant threshold, the recursion bottoms out. Hopefully, at that point, the problem is trivial, but if not, we switch to a different algorithm instead.

Proving a divide-and-conquer algorithm correct usually involves strong induction. Analyzing the running time requires setting up and solving a recurrence, which often (but unfortunately not always!) can be solved using the Master Theorem.

1.5 Multiplication

Adding two n-digit numbers takes $O(n)$ time by the obvious iterative "ripple-carry" algorithm, using a lookup table for each one-digit addition. Similarly, multiplying an n-digit number by a one-digit number takes $O(n)$ time, using essentially the same algorithm.

What about multiplying two n-digit numbers? At least in the United States, every grade school student (supposedly) learns to multiply by breaking the problem into n one-digit multiplications and n additions:

$$\begin{array}{r}
31415962 \\
\times\ 27182818 \\
\hline
251327696 \\
31415962 \\
251327696 \\
62831924 \\
251327696 \\
31415962 \\
219911734 \\
62831924 \\
\hline
853974377340916
\end{array}$$

We could easily formalize this algorithm into a pair of nested for-loops. The algorithm runs in $O(n^2)$ time—there are $O(n^2)$ digits in the partial products, and for each digit, we spend constant time.

We can do better by exploiting the following algebraic formula:

$$(10^m a + b)(10^m c + d) = 10^{2m} ac + 10^m (bc + ad) + bd$$

Here is a divide-and-conquer algorithm that computes the product of two n-digit numbers x and y, based on this formula. Each of the four subproducts e, f, g, h is computed recursively.

$m \leftarrow \lceil n/2 \rceil$
$a \leftarrow \lfloor x/10^m \rfloor$; $b \leftarrow x \bmod 10^m$
$c \leftarrow \lfloor y/10^m \rfloor$; $d \leftarrow y \bmod 10^m$
$e \leftarrow a \cdot c$
$f \leftarrow a \cdot d$
$g \leftarrow b \cdot c$
$h \leftarrow b \cdot d$
return $10^{2m} e + 10^m (f + g) + h$

You can easily prove the algorithm correct by induction. Since a, b, c, d all have at most $\lceil n/2 \rceil$ digits, the running time for this algorithm is given by the recurrence

$$T(n) \leq 4T(\lceil n/2 \rceil) + O(n), \qquad T(1) = 1,$$

which solves to $T(n) = O(n^2)$ by the Master Theorem (after a domain transformation). Hmm ... I guess this didn't help after all.

But there's a trick, first suggested by Anatoliĭ Karatsuba in 1962. We can compute the middle coefficient $bc+ad$ using only *one* recursive multiplication, by exploiting yet another bit of algebra:[2]

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

This lets use replace the last three lines in the previous algorithm by the following two lines:

```
g ← (a − b) · (c − d)
return 10^(2m)e + 10^m(e + f − g) + f
```

The running time of the new improved algorithm is given by the recurrence

$$T(n) \leq 3T(\lceil n/2 \rceil) + O(n), \qquad T(1) = 1.$$

After a domain transformation, we can plug this into the Master Theorem to get the solution $T(n) = O(n^{\lg 3}) = O(n^{1.585})$, which is a significant improvement over our earlier quadratic-time algorithm.

Of course, in practice, all this is done in binary instead of decimal.

Splitting the numbers into more pieces and combining them in more complicated ways gives even faster algorithms. Ultimately, this idea leads to the development of the *Fast Fourier Transform*, a complicated divide-and-conquer algorithm that can be used to multiply two n-digit numbers in $O(n \log n)$ time. FFT's are described in Chapter 32 of [CLR].

1.6 Exponentiation

Given two integers a and n, we want to compute a^n. The standard naïve method is a simple for-loop that does $n - 1$ multiplications by a:

$$a \to a^2 \to a^3 \to \cdots \to a^n$$

This iterative algorithm requires $n - 1 = O(n)$ multiplications. (Despite the previous section, I'll assume for this problem that multiplications take constant time. For example, numbers might be represented in floating-point notation.)

There is a much faster divide-and-conquer method, using the formula $a^n = a^{\lfloor n/2 \rfloor} \cdot a^{\lceil n/2 \rceil}$. What makes this approach more efficient is that once we compute the first factor $a^{\lfloor n/2 \rfloor}$, we can compute the second factor $a^{\lceil n/2 \rceil}$ using at most one more multiplication. The total number of multiplications is given by the recurrence $T(n) = T(\lfloor n/2 \rfloor) + 2$, with the base case $T(1) = 0$. After a domain transformation, the Master Theorem gives us the solution $T(n) = O(\log n)$.

Incidentally, this algorithm is asymptotically optimal — any algorithm for computing a^n must perform $\Omega(\log n)$ multiplications. In fact, when n is a power of two, this algorithm is *exactly* optimal. However, there are slightly faster methods for other values of n. For example, our divide-and-conquer algorithm computes a^{15} in six multiplications ($a^{15} = a^7 \cdot a^7 \cdot a$; $a^7 = a^3 \cdot a^3 \cdot a$; $a^3 = a \cdot a \cdot a$), but only five multiplications are necessary ($a \to a^2 \to a^3 \to a^5 \to a^{10} \to a^{15}$). Nobody knows of an algorithm that always uses the minimum possible number of multiplications.

[2] In class, I used the formula $(a + c)(b + d) - ac - bd = bc + ad$. This also leads to an $O(n^{\lg 3})$-time algorithm, but the recurrence is a bit messier: $a - b$ and $c - d$ are still m-digit numbers, but $a + b$ and $c + d$ might have $m + 1$ digits. Karatsuba actually proposed the messier algorithm; the simplification is due to Donald Knuth.