A number of relevant web pages can be found by searching for strings like "back of the envelope" and "Fermi problems".

7.8 Quick Calculations in Everyday Life *[Sidebar]*

The publication of this column in *Communications of the ACM* provoked many interesting letters. One reader told of hearing an advertisement state that a salesperson had driven a new car 100,000 miles in one year, and then asking his son to examine the validity of the claim. Here's one quick answer: there are 2000 working hours per year (50 weeks times 40 hours per week), and a salesperson might average 50 miles per hour; that ignores time spent actually selling, but it does multiply to the claim. The statement is therefore at the outer limits of believability.

Everyday life presents us with many opportunities to hone our skills at quick calculations. For instance, how much money have you spent in the past year eating in restaurants? I was once horrified to hear a New Yorker quickly compute that he and his wife spend more money each month on taxicabs than they spend on rent. And for California readers (who may not know what a taxicab is), how long does it take to fill a swimming pool with a garden hose?

Several readers commented that quick calculations are appropriately taught at an early age. Roger Pinkham wrote

> I am a teacher and have tried for years to teach "back-of-the-envelope" calculations to anyone who would listen. I have been marvelously unsuccessful. It seems to require a doubting-Thomas turn of mind.
>
> My father beat it into me. I come from the coast of Maine, and as a small child I was privy to a conversation between my father and his friend Homer Potter. Homer maintained that two ladies from Connecticut were pulling 200 pounds of lobsters a day. My father said, "Let's see. If you pull a pot every fifteen minutes, and say you get three legal per pot, that's 12 an hour or about 100 per day. I don't believe it!"
>
> "Well it is true!" swore Homer. "You never believe anything!"
>
> Father wouldn't believe it, and that was that. Two weeks later Homer said, "You know those two ladies, Fred? They were only pulling 20 pounds a day."
>
> Gracious to a fault, father grunted, "Now that I believe."

Several other readers discussed teaching this attitude to children, from the viewpoints of both parent and child. Popular questions for children were of the form "How long would it take you to walk to Washington, D.C.?" and "How many leaves did we rake this year?" Administered properly, such questions seem to encourage a life-long inquisitiveness in children, at the cost of bugging the heck out of the poor kids at the time.

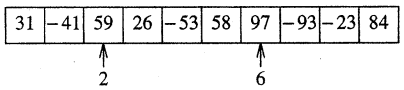
COLUMN 8: ALGORITHM DESIGN TECHNIQUES

Column 2 describes the everyday effect of algorithm design on programmers: algorithmic insights can make a program simpler. In this column we'll see a less frequent but more dramatic contribution of the field: sophisticated algorithms sometimes give extreme performance improvements.

This column studies four different algorithms for one small problem, with an emphasis on the techniques used to design them. Some of the algorithms are a little complicated, but with justification. While the first program we'll study takes fifteen days to solve a problem of size 100,000, the final program solves the same problem in five milliseconds.

8.1 The Problem and a Simple Algorithm

The problem arose in one-dimensional pattern recognition; we'll see its history later. The input is a vector x of n floating-point numbers; the output is the maximum sum found in any *contiguous* subvector of the input. For instance, if the input vector contains these ten elements



then the program returns the sum of $x[2..6]$, or 187. The problem is easy when all the numbers are positive; the maximum subvector is the entire input vector. The rub comes when some of the numbers are negative: should we include a negative number in hopes that positive numbers on either side will compensate for it? To complete the problem definition, we'll say that when all inputs are negative the maximum-sum subvector is the empty vector, which has sum zero.

The obvious program for this task iterates over all pairs of integers i and j satisfying $0 \le i \le j < n$; for each pair it computes the sum of $x[i..j]$ and checks whether that sum is greater than the maximum sum so far. The pseudocode for Algorithm 1 is

```
maxsofar = 0
for i = [0, n)
    for j = [i, n)
        sum = 0
        for k = [i, j]
            sum += x[k]
        /* sum is sum of x[i..j] */
        maxsofar = max(maxsofar, sum)
```

This code is short, straightforward and easy to understand. Unfortunately, it is also slow. On my computer, for instance, the program takes about 22 minutes if n is 10,000 and fifteen days if n is 100,000; we'll see the timing details in Section 8.5.

Those times are anecdotal; we get a different kind of feeling for the algorithm's efficiency using the big-oh notation described in Section 6.1. The outermost loop is executed exactly n times, and the middle loop is executed at most n times in each execution of the outer loop. Multiplying those two factors of n shows that the code in the middle loop is executed $O(n^2)$ times. The innermost loop within the middle loop is never executed more than n times, so its cost is $O(n)$. Multiplying the cost per inner loop times its number of executions shows that the cost of the entire program is proportional to n cubed. We'll therefore refer to this as a cubic algorithm.

This example illustrates the technique of big-oh analysis and many of its strengths and weaknesses. Its primary weakness is that we still don't really know the amount of time the program will take for any particular input; we just know that the number of steps is $O(n^3)$. That weakness is often compensated for by two strong points of the method. Big-oh analyses are usually easy to perform (as above), and the asymptotic run time is often sufficient for a back-of-the-envelope calculation to decide whether a program is sufficient for a given application.

The next several sections use asymptotic run time as the only measure of program efficiency. If that makes you uncomfortable, peek ahead to Section 8.5, which shows that such analyses are extremely informative for this problem. Before you read further, though, take a minute to try to find a faster algorithm.

8.2 Two Quadratic Algorithms

Most programmers have the same response to Algorithm 1: "There's an obvious way to make it a lot faster." There are two obvious ways, however, and if one is obvious to a given programmer then the other often isn't. Both algorithms are quadratic — they take $O(n^2)$ steps on an input of size n — and both achieve their run time by computing the sum of $x[i..j]$ in a constant number of steps rather than in the $j-i+1$ additions of Algorithm 1. But the two quadratic algorithms use very different methods to compute the sum in constant time.

The first quadratic algorithm computes the sum quickly by noticing that the sum of $x[i..j]$ is intimately related to the sum previously computed (that of $x[i..j-1]$). Exploiting that relationship leads to Algorithm 2.


```
maxsofar = 0
for i = [0, n)
    sum = 0
    for j = [i, n)
        sum += x[j]
        /* sum is sum of x[i..j] */
        maxsofar = max(maxsofar, sum)
```

The statements inside the first loop are executed n times, and those inside the second loop are executed at most n times on each execution of the outer loop, so the total run time is $O(n^2)$.

An alternative quadratic algorithm computes the sum in the inner loop by accessing a data structure built before the outer loop is ever executed. The i^{th} element of *cumarr* contains the cumulative sum of the values in $x[0..i]$, so the sum of the values in $x[i..j]$ can be found by computing $cumarr[j]-cumarr[i-1]$. This results in the following code for Algorithm 2b.

```
cumarr[-1] = 0
for i = [0, n)
    cumarr[i] = cumarr[i-1] + x[i]
maxsofar = 0
for i = [0, n)
    for j = [i, n)
        sum = cumarr[j] - cumarr[i-1]
        /* sum is sum of x[i..j] */
        maxsofar = max(maxsofar, sum)
```

(Problem 5 addresses how we might access $cumarr[-1]$.) This code takes $O(n^2)$ time; the analysis is exactly the same as that of Algorithm 2.

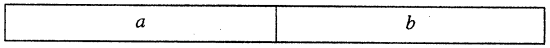
The algorithms we've seen so far inspect all possible pairs of starting and ending values of subvectors and evaluate the sum of the numbers in that subvector. Because there are $O(n^2)$ subvectors, any algorithm that inspects all those values must take at least quadratic time. Can you think of a way to sidestep this problem and achieve an algorithm that runs in less time?

8.3 A Divide-and-Conquer Algorithm

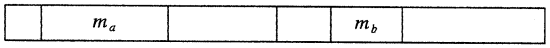
Our first subquadratic algorithm is complicated; if you get bogged down in its details, you won't lose much by skipping to the next section. It is based on the following divide-and-conquer recipe:

> *To solve a problem of size n, recursively solve two subproblems of size approximately n/2, and combine their solutions to yield a solution to the complete problem.*

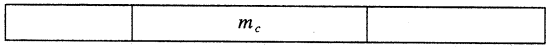
In this case the original problem deals with a vector of size n, so the most natural way to divide it into subproblems is to create two subvectors of approximately equal size, which we'll call a and b.



We then recursively find the maximum-sum subvectors in a and b, which we'll call m_a and m_b.



It is tempting to think that we have now solved the problem because the maximum-sum subvector of the entire vector must be either m_a or m_b. That is almost right. In fact, the maximum is either entirely in a, entirely in b, or it crosses the border between a and b; we'll call that m_c for the maximum *crossing* the border.



Thus our divide-and-conquer algorithm will compute m_a and m_b recursively, compute m_c by some other means, and then return the maximum of the three.

That description is almost enough to write code. All we have left to describe is how we'll handle small vectors and how we'll compute m_c. The former is easy: the maximum of a one-element vector is the only value in the vector (or zero if that number is negative), and the maximum of a zero-element vector was defined to be zero. To compute m_c we observe that its left side is the largest subvector starting at the boundary and reaching into a, and similarly for its right side in b. Putting these facts together leads to the following code for Algorithm 3:

```
float maxsum3(l, u)
    if (l > u)  /* zero elements */
        return 0
    if (l == u)  /* one element */
        return max(0, x[l])

    m = (l + u) / 2
    /* find max crossing to left */
    lmax = sum = 0
    for (i = m; i >= l; i--)
        sum += x[i]
        lmax = max(lmax, sum)
    /* find max crossing to right */
    rmax = sum = 0
    for i = (m, u]
        sum += x[i]
        rmax = max(rmax, sum)

    return max(lmax+rmax, maxsum3(l, m), maxsum3(m+1, u))
```

Algorithm 3 is originally invoked by the call


```
answer = maxsum3(0, n-1)
```

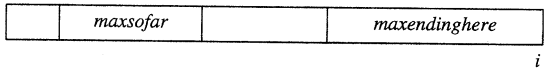
The code is subtle and easy to get wrong, but it solves the problem in $O(n \log n)$ time. We can prove that fact in several ways. An informal argument observes that the algorithm does $O(n)$ work on each of $O(\log n)$ levels of recursion. The argument can be made more precise by the use of recurrence relations. If $T(n)$ denotes the time to solve a problem of size n, then $T(1)=O(1)$ and

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

Problem 15 shows that this recurrence has the solution $T(n) = O(n \log n)$.

8.4 A Scanning Algorithm

We'll now use the simplest kind of algorithm that operates on arrays: it starts at the left end (element $x[0]$) and scans through to the right end (element $x[n-1]$), keeping track of the maximum-sum subvector seen so far. The maximum is initially zero. Suppose that we've solved the problem for $x[0..i-1]$; how can we extend that to include $x[i]$? We use reasoning similar to that of the divide-and-conquer algorithm: the maximum-sum subarray in the first i elements is either in the first $i-1$ elements (which we'll store in *maxsofar*), or it ends in position i (which we'll store in *maxendinghere*).



Recomputing *maxendinghere* from scratch using code like that in Algorithm 3 yields yet another quadratic algorithm. We can get around this by using the technique that led to Algorithm 2: instead of computing the maximum subvector ending in position i from scratch, we'll use the maximum subvector that ends in position $i-1$. This results in Algorithm 4.

```
maxsofar = 0
maxendinghere = 0
for i = [0, n)
    /* invariant: maxendinghere and maxsofar
       are accurate for x[0..i-1] */
    maxendinghere = max(maxendinghere + x[i], 0)
    maxsofar = max(maxsofar, maxendinghere)
```

The key to understanding this program is the variable *maxendinghere*. Before the first assignment statement in the loop, *maxendinghere* contains the value of the maximum subvector ending in position $i-1$; the assignment statement modifies it to contain the value of the maximum subvector ending in position i. The statement increases it by the value $x[i]$ so long as doing so keeps it positive; when it goes negative, it is reset to zero (because the maximum subvector ending at i is now the empty vector). Although the code is subtle, it is short and fast: its run time is $O(n)$, so we'll refer to it as a linear algorithm.