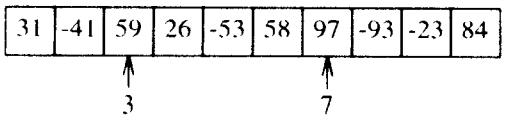
COLUMN 7: ALGORITHM DESIGN TECHNIQUES

Column 2 describes the "everyday" impact that algorithm design can have on programmers: an algorithmic view of a problem gives insights that can make a program simpler to understand and to write. In this column we'll study a contribution of the field that is less frequent but more impressive: sophisticated algorithmic methods sometimes lead to dramatic performance improvements.

This column is built around one small problem, with an emphasis on the algorithms that solve it and the techniques used to design the algorithms. Some of the algorithms are a little complicated, but with justification. While the first program we'll study takes thirty-nine days to solve a problem of size ten thousand, the final one solves the same problem in less than a second.

7.1 The Problem and a Simple Algorithm

The problem arose in one-dimensional pattern recognition; I'll describe its history later. The input is a vector X of N real numbers; the output is the maximum sum found in any *contiguous* subvector of the input. For instance, if the input vector is



then the program returns the sum of $X[3..7]$, 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 the positive numbers to its sides will compensate for its negative contribution? To complete the definition of the problem, 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 L and U satisfying $1 \le L \le U \le N$; for each pair it computes the sum of $X[L..U]$ and checks whether that sum is greater than the maximum sum so far. The pseudocode for Algorithm 1 is

```
MaxSoFar := 0.0
for L := 1 to N do
    for U := L to N do
        Sum := 0.0
        for I := L to U do
            Sum := Sum + X[I]
        /* Sum now contains the sum of X[L..U] */
        MaxSoFar := max(MaxSoFar, Sum)
```

This code is short, straightforward, and easy to understand. Unfortunately, it has the severe disadvantage of being slow. On the computer I typically use, for instance, the code takes about an hour if N is 1000 and thirty-nine days if N is 10,000; we'll get to timing details in Section 7.5.

Those times are anecdotal; we get a different kind of feeling for the algorithm's efficiency using the "big-oh" notation described in Section 5.1. The statements in the outermost loop are executed exactly N times, and those in the middle loop are executed at most N times in each execution of the outer loop. Multiplying those two factors of N shows that the four lines contained in the middle loop are executed $O(N^2)$ times. The loop in those four lines 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, so we'll refer to this as a cubic algorithm.

This example illustrates the technique of big-oh analysis of run time 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 it executes 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 or not a program is efficient enough 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 7.5, which shows that for this problem such analyses are extremely informative. Before you read further, though, take a minute to try to find a faster algorithm.

7.2 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 $X[L..U]$ in a constant number of steps rather than in the $U-L+1$ steps 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 $X[L..U]$ has an intimate relationship to the sum previously computed, that of $X[L..U-1]$. Exploiting that relationship leads to Algorithm 2.

```
MaxSoFar := 0.0
for L := 1 to N do
    Sum := 0.0
    for U := L to N do
        Sum := Sum + X[U]
        /* Sum now contains the sum of X[L..U] */
        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 *CumArray* contains the cumulative sum of the values in $X[1..I]$, so the sum of the values in $X[L..U]$ can be found by computing $CumArray[U]-CumArray[L-1]$. This results in the following code for Algorithm 2b.

```
CumArray[0] := 0.0
for I := 1 to N do
    CumArray[I] := CumArray[I-1] + X[I]
MaxSoFar := 0.0
for L := 1 to N do
    for U := L to N do
        Sum := CumArray[U] - CumArray[L-1]
        /* Sum now contains the sum of X[L..U] */
        MaxSoFar := max(MaxSoFar, Sum)
```

This code takes $O(N^2)$ time; the analysis is exactly the same as the analysis of Algorithm 2.

The algorithms we've seen so far inspect all possible pairs of starting and ending values of subvectors and consider the sum of the numbers in that subvector. Because there are $O(N^2)$ subvectors, any algorithm that inspects all such 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?

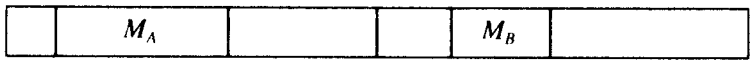
7.3 A Divide-and-Conquer Algorithm

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 schema:

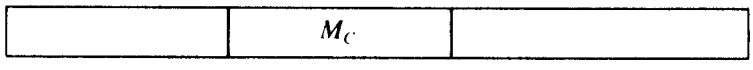
> *To solve a problem of size N, recursively solve two subproblems of size approximately N/2, and combine their solutions to yield a solution to the complete problem.*

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 subvectors in A and B, which we'll call M_A and M_B.



It is tempting to think that we have solved the problem because the maximum sum subvector of the entire vector must be either M_A or M_B, and 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 previously defined to be zero. To compute M_C we observe that its component in A is the largest subvector starting at the boundary and reaching into A, and similarly for its component in B. Putting these facts together leads to the following code for Algorithm 3, which is originally invoked by the procedure call

```
Answer := MaxSum(1,N)
```

```
recursive function MaxSum(L, U)
    if L > U then     /* Zero-element vector */
        return 0.0
    if L = U then     /* One-element vector */
        return max(0.0, X[L])

    M := (L+U)/2   /* A is X[L..M], B is X[M+1..U] */
    /* Find max crossing to left */
        Sum := 0.0; MaxToLeft := 0.0
        for I := M downto L do
            Sum := Sum + X[I]
            MaxToLeft := max(MaxToLeft, Sum)
    /* Find max crossing to right */
        Sum := 0.0; MaxToRight := 0.0
        for I := M+1 to U do
            Sum := Sum + X[I]
            MaxToRight := max(MaxToRight, Sum)
    MaxCrossing := MaxToLeft + MaxToRight

    MaxInA := MaxSum(L,M)
    MaxInB := MaxSum(M+1,U)
    return max(MaxCrossing, MaxInA, MaxInB)
```

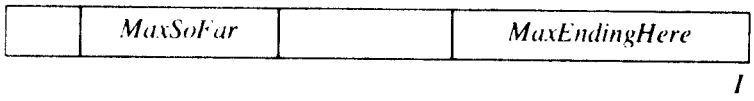
The code is complicated and easy to get wrong, but it solves the problem in $O(N \log N)$ time. There are a number of ways to prove this fact. An informal argument observes that the algorithm does $O(N)$ work on each of $O(\log N)$ levels of recursion. The argument can be made more precise by the use 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 11 shows that this recurrence has the solution $T(N) = O(N \log N)$.

7.4 A Scanning Algorithm

We'll now use the simplest kind of algorithm that operates on arrays: it starts at the left end (element $X[1]$) and scans through to the right end (element $X[N]$), keeping track of the maximum sum subvector seen so far. The maximum is initially zero. Suppose that we've solved the problem for $X[1..I-1]$; how can we extend that to a solution for the first I elements? We use reasoning similar to that of the divide-and-conquer algorithm: the maximum sum in the first I elements is either the maximum sum in the first $I-1$ elements (which we'll call *MaxSoFar*), or it is that of a subvector that ends in position I (which we'll call *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 := 0.0
MaxEndingHere := 0.0
for I := 1 to N do
    /*  Invariant: MaxEndingHere and MaxSoFar
        are accurate for X[1..I-1] */
    MaxEndingHere := max(MaxEndingHere+X[I], 0.0)
    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 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. David Gries systematically derives and verifies this algorithm in his paper "A Note on the Standard Strategy for Developing Loop Invariants and Loops" in the journal *Science of Computer Programming 2*, pp. 207-214.

7.5 What Does It Matter?

So far I've played fast and loose with "big-ohs"; it's time for me to come clean and tell about the run times of the programs. I implemented the four primary algorithms (all except Algorithm 2b) in the C language on a VAX-11/750, timed them, and extrapolated the observed run times to achieve the following table.

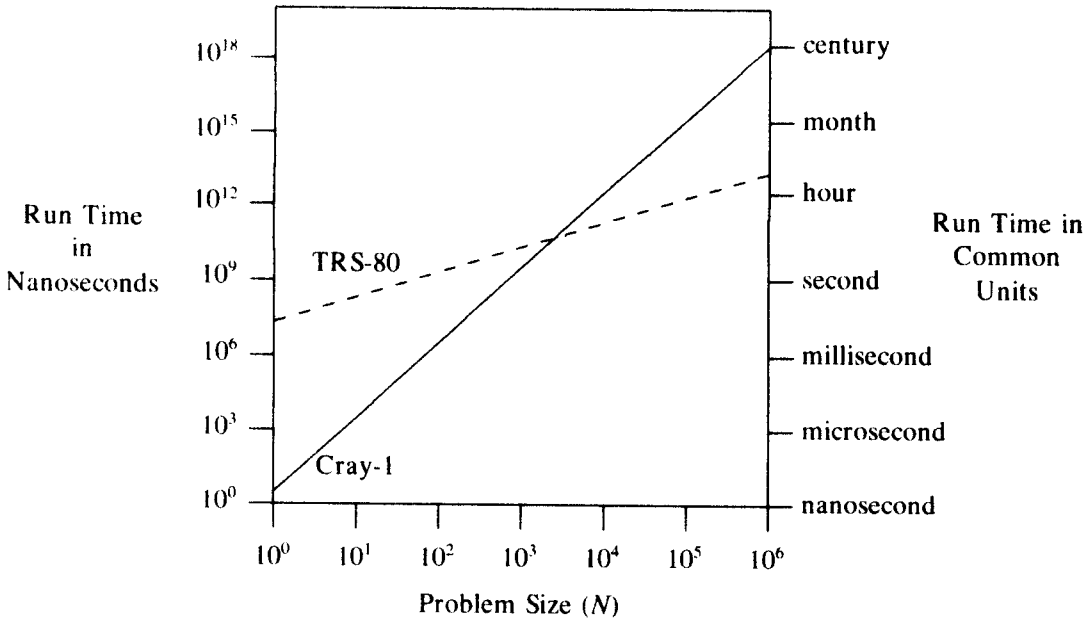
ALGORITHM		1	2	3	4
Lines of C Code		8	7	14	7
Run time in microseconds		$3.4N^3$	$13N^2$	$46N \log_2 N$	$33N$
Time to solve a problem of size	10^2	3.4 secs	.13 secs	.03 secs	.003 secs
	10^3	.94 hrs	13 secs	.45 secs	.033 secs
	10^4	39 days	22 mins	6.1 secs	.33 secs
	10^5	108 yrs	1.5 days	1.3 mins	3.3 secs
	10^6	108 mill	5 mos	15 mins	33 secs
Max size problem solved in one	sec	67	280	2000	30,000
	min	260	2200	82,000	2,000,000
	hr	1000	17,000	3,500,000	120,000,000
	day	3000	81,000	73,000,000	2,800,000,000
If N multiplies by 10, time multiplies by		1000	100	10+	10
If time multiplies by 10, N multiplies by		2.15	3.16	10−	10

This table makes a number of points. The most important is that proper algorithm design can make a big difference in run time; that point is underscored by the middle rows. The last two rows show how increases in problem size are related to increases in run time.

Another important point is that when we're comparing cubic, quadratic, and linear algorithms with one another, the constant factors of the programs don't matter much. (The discussion of the $O(N!)$ algorithm in Section 2.4 shows that constant factors matter even less in functions that grow faster than polynomially.) To underscore this point, I conducted an experiment in which I tried to make the constant factors of two algorithms differ by as much as possible. To achieve a huge constant factor I implemented Algorithm 4 on a BASIC interpreter on a Radio Shack TRS-80 Model III microcomputer. For the other end of the spectrum, Eric Grosse and I implemented Algorithm 1 in fine-tuned FORTRAN on a Cray-1 supercomputer. We got the disparity we wanted: the run time of the cubic algorithm was measured as $3.0N^3$ nanoseconds, while the run time of the linear algorithm was $19.5N$ milliseconds, or $19{,}500{,}000N$ nanoseconds. This table shows how those expressions translate to times for various problem sizes.

N	CRAY-1, FORTRAN, CUBIC ALGORITHM	TRS-80, BASIC, LINEAR ALGORITHM
10	3.0 microsecs	200 millisecs
100	3.0 millisecs	2.0 secs
1000	3.0 secs	20 secs
10,000	49 mins	3.2 mins
100,000	35 days	32 mins
1,000,000	95 yrs	5.4 hrs

The difference in constant factors of six and a half million allowed the cubic algorithm to start off faster, but the linear algorithm was bound to catch up. The break-even point for the two algorithms is around 2,500, where each takes about fifty seconds.



7.6 Principles

The history of the problem sheds light on the algorithm design techniques. The problem arose in a pattern-matching procedure designed by Ulf Grenander of Brown University in the two-dimensional form described in Problem 7. In that form, the maximum sum subarray was the maximum likelihood estimator of a certain kind of pattern in a digitized picture. Because the two-dimensional problem required too much time to solve, Grenander simplified it to one dimension to gain insight into its structure.

Grenander observed that the cubic time of Algorithm 1 was prohibitively slow, and derived Algorithm 2. In 1977 he described the problem to Michael Shamos of UNILOGIC, Ltd. (then of Carnegie-Mellon University) who

overnight designed Algorithm 3. When Shamos showed me the problem shortly thereafter, we thought that it was probably the best possible; researchers had just shown that several similar problems require time proportional to $N \log N$. A few days later Shamos described the problem and its history at a Carnegie-Mellon seminar attended by statistician Jay Kadane, who designed Algorithm 4 within a minute. Fortunately, we know that there is no faster algorithm: any correct algorithm must take $O(N)$ time.

Even though the one-dimensional problem is completely solved, Grenander's original two-dimensional problem remained open eight years after it was posed, as this book went to press. Because of the computational expense of all known algorithms, Grenander had to abandon that approach to the pattern-matching problem. Readers who feel that the linear-time algorithm for the one-dimensional problem is "obvious" are therefore urged to find an "obvious" algorithm for Problem 7!

The algorithms in this story were never incorporated into a system, but they illustrate important algorithm design techniques that have had substantial impact on many systems (see Section 7.9).

Save state to avoid recomputation. This simple form of dynamic programming arose in Algorithms 2 and 4. By using space to store results, we avoid using time to recompute them.

Preprocess information into data structures. The *CumArray* structure in Algorithm 2b allowed the sum of a subvector to be computed in just a couple of operations.

Divide-and-conquer algorithms. Algorithm 3 uses a simple form of divide-and-conquer; textbooks on algorithm design describe more advanced forms.

Scanning algorithms. Problems on arrays can often be solved by asking "how can I extend a solution for $X[1..I-1]$ to a solution for $X[1..I]$?" Algorithm 4 stores both the old answer and some auxiliary data to compute the new answer.

Cumulatives. Algorithm 2b uses a cumulative table in which the I^{th} element contains the sum of the first I values of X; such tables are common when dealing with ranges. In business data processing applications, for instance, one finds the sales from March to October by subtracting the February year-to-date sales from the October year-to-date sales.

Lower bounds. Algorithm designers sleep peacefully only when they know their algorithms are the best possible; for this assurance, they must prove a matching lower bound. The linear lower bound for this problem is the subject of Problem 9; more complex lower bounds can be quite difficult.