programming pearls

by Jon Bentley

THE BACK OF THE ENVELOPE

It was in the middle of a fascinating conversation on software engineering that Bob Martin asked me, "How much water flows out of the Mississippi River in a day?" Because I had found his comments up to that point deeply insightful, I politely stifled my true response and said, "Pardon me?". When he asked again I realized that I had no choice but to humor the poor fellow, who had obviously cracked under the pressures of running a large software shop within AT&T Bell Laboratories.

My response went something like this. I figured that near its mouth the river was about a mile wide and maybe 20 feet deep (or about one two-hundred-and-fiftieth of a mile). I guessed that the rate of flow was five miles an hour, or 120 miles per day. Multiplying $1 \times 1/250 \times 120$ showed that the river discharged about half a cubic mile of water per day, to within an order of magnitude.[1] But so what?

At that point Martin picked up a proposal on his desk for the development of a large computer-based mail system, and went through a similar sequence of calculations. Although his numbers were more precise (they were straight from the proposal), the calculations were just as simple and much more revealing. They showed that, under generous assumptions, the proposed system had a chance of working only if there were at least a hundred and twenty seconds in each minute. He had sent the design back to the drawing board the previous day.

That was Bob Martin's wonderful (if eccentric) way of introducing the engineering technique of "back-of-the-envelope" calculations. The technique is standard fare in most engineering curricula, and is bread and butter for practicing engineers in many fields. Unfortunately, it is too often neglected in computing.

Quick Calculations

Card, Moran, and Newell paint an ambitious picture on pages 9 and 10 of their *Psychology of Human-Computer Interaction* (published in 1983 by Lawrence Erlbaum Associates, Publishers, of Hillsdale, New Jersey, and London, England: this excerpt is reprinted with the kind permission of the publisher).

> A system designer, the head of a small team writing the specifications for a desktop calendar-scheduling system, is choosing between having users type a key for each command and having them point to a menu with a lightpen. On his whiteboard, he lists some representative tasks users of his system must perform. In two columns, he writes the steps needed by the "key-command" and "menu" options. From a handbook, he culls the times for each step, adding the step times to get total task times. The key-command system takes less time, but only slightly. But, applying the analysis from another section of the handbook, he calculates that the menu system will be faster to learn; in fact, it will be learnable in half the time. He has estimated previously that an effective menu system will require a more expensive processor: 20% more memory, 100% more microcode memory, and a more expensive display. Is the extra expenditure worthwhile? A few more minutes of calculation and he realizes the startling fact that, for the manufacturing quantities now anticipated, training costs for the key-command system will exceed unit manufacturing costs! The increase in hardware costs would be much more than balanced by the decrease in training costs, even before considering the increase in market that can be expected for a more easily learned system. Are there advantages to the key-command system in other areas, which need to be balanced? He proceeds with other analyses, considering the load on the user's memory, the potential for user errors, and the likelihood of fatigue. In the next room, the Pascal compiler hums idly, unused, awaiting his decision.

Their book then goes on to develop a scientific base in psychology that is a necessary precursor to such a handbook.

That scenario shows how a few envelopes' worth of arithmetic might enable a system designer to make a rational choice between two appealing alternatives. That is a fundamentally different use than Martin's, his

[1] When I asked Peter Weinberger how much water flows out of the Mississippi per day, he responded, "As much as flows in." He then estimated that the Mississippi basin was about 1000 by 1000 miles, and the annual rainfall there was about one foot (or one five-thousandth of a mile). Multiplying $1000 \times 1000 \times 1/5000$ gave 200 cubic miles per year, or a little more than half a cubic mile per day. It's important to double check all calculations, and especially so for quick ones. As a (cheating) triple check, an almanac reported that the river's discharge is 640,000 cubic feet per second, or about 0.4 cubic miles per day. The proximity of the two estimates to one another, and especially to the almanac's answer, is a fine example of sheer dumb luck.

analysis of a single design uncovered a fatal flaw. In both cases, a short sequence of calculations was sufficient to answer the question at hand; additional figuring would have shed little light.

Early in the design of a system, rapid calculations can steer the designer away from dangerous waters into safe passages. And if you don't use them early, they may show in retrospect that a project was doomed to failure. The calculations are usually trivial, employing no more than high school mathematics; the hard part is remembering to apply them early enough in the life of a software project.

Safety Factors

The output of a quick calculation is only as good as its input. Sometimes sloppy input is enough to get into the right ballpark: if you guess about twenty percent here and fifty percent there and still find that a design is a hundred times above or below specification, additional accuracy isn't needed. With accurate data, though, back-of-the-envelope calculations can yield accurate answers.[2] Before placing too much confidence in a twenty percent margin, consider Vic Vyssotsky's advice (from a talk he has given on several occasions).

"Most of you," says Vyssotsky, "probably recall pictures of 'Galloping Gertie,' the Tacoma Narrows bridge which tore itself apart in a windstorm in 1940.[3] Well, suspension bridges had been ripping themselves apart that way for 80 years or so before Galloping Gertie. It's an aerodynamic lift phenomenon, and to do a proper engineering calculation of the forces, which involve drastic nonlinearities, you have to use the mathematics and concepts of Kolmogorov to model the eddy spectrum. Nobody really knew how to do this correctly in detail until the 1950s or thereabouts. So, why hasn't the Brooklyn Bridge torn itself apart, like Galloping Gertie?

"It's because John Roebling had sense enough to know what he *didn't* know. His notes and letters on the design of the Brooklyn Bridge still exist, and they are a fascinating example of a good engineer recognizing the limits of his knowledge. He knew about aerodynamic lift on suspension bridges; he had watched it. And he knew he didn't know enough to model it. So he designed the stiffness of the truss on the Brooklyn Bridge roadway to be *six times* what a normal calculation based on known static and dynamic loads would have called for. And, he specified a network of diagonal stays running down to the roadway, to stiffen the entire bridge structure. Go look at those sometime; they're almost unique.

"When Roebling was asked whether his proposed bridge wouldn't collapse like so many others, he said, 'No, because I designed it six times as strong as it needs to be, to prevent that from happening.'

"Roebling was a good engineer, and he built a good bridge, by employing a huge safety factor to compensate for his ignorance. Do we do that? I submit to you that in calculating performance of our real-time software systems we ought to derate them by a factor of two, or four, or six, to compensate for our ignorance. In making reliability/availability commitments, we ought to stay back from the objectives we *think* we can meet by a factor of ten, to compensate for our ignorance. In estimating size and cost and schedule, we should be conservative by a factor of two or four to compensate for our ignorance. We should design the way John Roebling did, and not the way his contemporaries did—so far as I know, none of the suspension bridges built by Roebling's contemporaries in the United States still stands, and a quarter of all the bridges of any type built in the U.S. in the 1870s collapsed within ten years of construction.

"Are we engineers, like John Roebling? I wonder."

A Case Study

To make the above points more concrete, I'll describe how I (almost) used them in a system I built for a small company. I've given the details in Carnegie-Mellon University Computer Science Technical Report CMU-CS-83-108; I'll just sketch them here.

The system prepared several reports a day to summarize the data on 1,000 80-column records; the reports were each about 80 pages long. The system's predecessor ran on a large mainframe; my task was to implement a similar system on a personal computer (using interpreted BASIC).

Early in the design of the system I did simple calculations to make sure that the personal computer was up to this application. The space analysis was simple: I calculated the size of the several largest tables, and found that they used only half of the 48K bytes of the machine. The time analysis involved two main phases. I didn't worry much about the time to read the records and build the tables used by the print phase. I knew that a previous system did that task on an IBM System/360 Model 25 in a minute, and the microprocessor on the personal computer is more powerful than that old workhorse. Instead, I concentrated on the time to print the report, which I thought would be limited by the 60-lines-per-minute speed of the printer. Each page of the report contained only about 30 lines, so the total time of 40 minutes was well within bounds. After this short analysis, I purchased three personal computers and implemented the design.

The first implementation of the program was enlightening. Storing the interpreted program required about 20 kilobytes of main memory that I had ignored in my calculation; the safety factor of two saved the day. The calculation of printing time was right on the mark; it took about 40 minutes. Unfortunately, I was way off in

[2] Sometimes accurate calculations are quite useful. In 1969 Don Knuth wrote a disk sorting package, only to find that it took twice the time predicted by his calculations. Diligent checking uncovered the flaw: due to a software bug, the system's one-year-old disks had run at only half their advertised speed for their entire lives. When the bug was fixed, the sorting package behaved as predicted and many other programs also ran faster.

[3] For more information on the event, see Section 2.6.1 of Braun's *Differential Equations and Their Applications*, Second Edition, published in 1978 by Springer-Verlag.

the time to read the records and build the table. Instead of taking a minute, it took *14 hours*, which made it awfully hard to prepare a few reports a day. The problem was that I had compared assembly code on the old System/360 with interpreted BASIC on the personal computer, ignoring the fact that interpreted BASIC usually runs several hundred times slower than assembly code.

At that point I did a more careful back-of-the-envelope calculation of the time. Multiplying 1,000 records by 80 columns per record by about 50 lines of BASIC code per column and then including the rate of 100 BASIC instructions per second showed that it would take about ten hours. Had I known that before I built the program, I would have used a faster language. Instead, I had an existing system and no choice but to tune the code (using techniques like those described in the February column) to make it faster. By spending forty hours of my time replacing 70 lines of BASIC with 110 lines of BASIC and 30 lines of assembly code, I was able to reduce the time of that phase from 14 hours to two hours and 20 minutes. That was good enough for the system, but more than it could have been had I done an accurate calculation beforehand and then chosen a more appropriate implementation language.

Principles

When you use back-of-the-envelope calculations, be sure to recall Einstein's famous advice

Everything should be made as simple as possible, but no simpler.

We know that simple calculations aren't too simple by including safety factors to compensate for our mistakes in estimating parameters and our ignorance of the problem at hand.

Problems

1. At what distances can a courier on a bicycle with a reel of magnetic tape be a more rapid carrier of information than a 56-kilobaud telephone line? Than a 1200-baud line?

2. If you punched the contents of a disk onto cards, would they fit in your office?

3. When is it cost effective to supply a programmer with a computer terminal at home?

4. Suppose the world is slowed down by a factor of a million. How long does it take for your computer to execute an instruction? Your disk to rotate once? Your disk arm to seek across the disk? You to type your name?

5. Which has the most computational oomph: a second of supercomputer time, a minute of midicomputer time, an hour of microcomputer time or a day of BASIC on a personal computer?

6. Suppose that a system must make 100 disk accesses to process a transaction (although some systems need fewer, some require many hundreds per transaction). How many transactions per hour (per disk) can such a system handle?

7. A programmer spends one hour of CPU time and one day of his time to speed up a program by ten percent on a machine that costs one hundred dollars per hour of CPU time. Individual runs of the program typically require a minute of CPU time; how long will it take to pay for the speedup if the program is run a hundred times a day? What if the speedup were a factor of two?

Solutions to the Problems

These solutions include guesses at constants that may be off by a factor of two, but not much further.

1. 6250 bytes per inch times a 2400 foot reel of tape times 0.5 for waste due to blocking gives 90 megabytes per reel of tape. 7000 bytes per second gives 25 megabytes per hour for the line. The bicyclist therefore has about three hours to transfer the data, which gives (say) a 20-mile radius of superiority. The bicyclist has 50 times as long, or almost a week, to beat a 1200-baud line.

2. A disk has about 200 megabytes (compared to 200 kilobytes for a 5.25″ floppy disk). A box of punched cards has 2000 times 80 bytes, or 160 kilobytes, so a disk is about 1250 boxes of punched cards. That 10′ by 6′ by 3′ pile might fit in my office, but I wouldn't want it to.

3. A terminal costs a couple of thousand dollars. A programmer (including many expenses beyond salary, unfortunately) costs about $100,000/yr, $2000/wk, $400/day, or $50/hr. Thus once a programmer has used the terminal for 40 hours, the investment cost is recovered and the employer gets free work. (I am told that this scheme was invented by management and does not go unnoticed by programmers' spouses.)

4. A one microsecond instruction takes one second, a 16 msec disk rotation (at 3600 rpm) takes 5 hours, a 30 msec seek takes 10 hours, and the two seconds to type my name takes about a month.

5. In a second, a supercomputer can do a hundred million 64-bit floating point operations, a midicomputer can do one million 16-bit integer additions, a microcomputer can execute half a million 8-bit instructions, and BASIC on a personal computer can execute one hundred instructions. The times stated in the problem work out to about the same amount of power for the first three machines, while poor BASIC is left way behind.

6. Ignoring slowdown due to queueing, 30 msec per disk operation gives 3 seconds per transaction or 1200 transactions per hour.

7. The cost of the change is $100 in machine time plus $400 in programmer time. The savings of 10 min/day is $16/day, which takes a month to pay for the investment. If the speedup were a factor of two, the savings of $80/day would pay for the speedup in a week.

Solutions to February's Problems

1. On a machine with eight-bit bytes a 256-byte table could represent the characters in eight (possibly overlapping) character classes. The *I*th bit in byte *J* tells whether character *J* is in class *I*. Testing membership involves accessing an array element, ANDing with a bit pattern containing a single one, and comparing the result to zero.

2. Given N a power of two, we are to initialize $C[0 \ldots N-1]$ such that $C[I]$ is the number of one bits in the binary representation of I. We use the identity that $J < 2^K$ implies $C[J + 2^K] = C[J] + 1$; that is, turning on the K^{th} bit adds one to the count. For this reason each element in the right column is precisely one greater than the corresponding element in the left column:

```
C[0]=0   C[4]=1
C[1]=1   C[5]=2
C[2]=1   C[6]=2
C[3]=2   C[7]=3
```

   The code therefore starts with a single-element table and repeatedly doubles its size by copying and incrementing.

```
c[0] := 0
M := 1
while M<N do
   /* invariant: C[0..M-1] is
      correct */
   for J:=0 to M-1 do
      C[M+J] := C[J]+1
   M := M+M
```

3. If the binary search algorithms report that they found the search value T, then it is in fact in the table. When applied to unsorted tables, though, the algorithms may sometimes report that T is not present when it in fact is. In such cases the algorithms locate a pair of adjacent elements that would establish that T is not in the table were it sorted.

4. Brooks combined two representations for the table. The function got to within a few units of the true answer, and the single decimal digit stored in the array gave the difference.

5. All the data structures mentioned can employ an additional sentinel node to remove a test from the inner loop of the search. Before a search starts it places the value being searched for in the sentinel node, which guarantees that the search will find its target. When the search succeeds a single comparison tells whether it found a "real" value or the one in the sentinel. This removes from the inner loop a test to determine whether the data structure is yet exhausted.

   Linked lists have sentinel nodes at the very end; they must also store a pointer to that node (this is particularly convenient in a circularly linked list with a "dummy" node). Closed hash tables use a sentinel cell at the end of the array. Nil pointers in a standard binary search tree are replaced by pointers to a single sentinel node in the modified tree.

Updates on Old Columns

The August column described a sorting program that used the integers to be sorted as indices into a bitmap. Augustus T. Crocker, Jr., of Shared Medical Systems Corporation points out that the "key-indexing" technique can also be used to access the I^{th} record in a disk file. Even though direct access on disk was not appropriate for the particular sorting problem (a quick calculation shows that 27,000 disk reads and as many writes would take at least 10 minutes, whereas the solution in the column took a few seconds), the technique is quite useful in many applications.

Testing the sort routine required "shuffling" a file of numbers; I mentioned in the September solution to Problem 3 that one can shuffle a file by calling a system sort. Ashley Shepherd and Alex Woronow of the University of Houston point out that there are more efficient shuffling algorithms when the data is stored in an array in primary memory. For example, Knuth gives the following shuffling algorithm in *Seminumerical Algorithms*:

```
for I := N downto 2 do
    swap(X[I],X[RandInt(1,I)])
```

The function *RandInt*(1, *I*) returns an integer chosen uniformly from the range 1..*I*.

The September column described Ed Reingold's problem of finding a 20-bit integer not on a tape containing one million such integers, and his solution based on binary search. David Malon of the University of Kentucky observed that the simpler solution of looking to see whether any of the first 512 integers is missing (using the 512 bits in 32 16-bit words) fails to find a missing integer with probability of only 0.000000000003, assuming that all distinct subsets of integers are equally likely to appear on the input tape. His idea leaves open the problem of relaxing that assumption by using a randomization algorithm. Eggert Ehmke of West Berlin points out that the "binary radix" search of Reingold's solution suggests a "binary lexicographic" sort that uses three work tapes as its queues.

James Woods of Informatics General Corporation generalized the anagrams problem of the September column to the problem of computing multiword anagrams in a lovely paper entitled "On Computing Anagrams." That title is itself a multiword anagram of such common phrases as *romantic pang among us, mangos moan capturing,* and *gaunt moron campaigns*, among others not