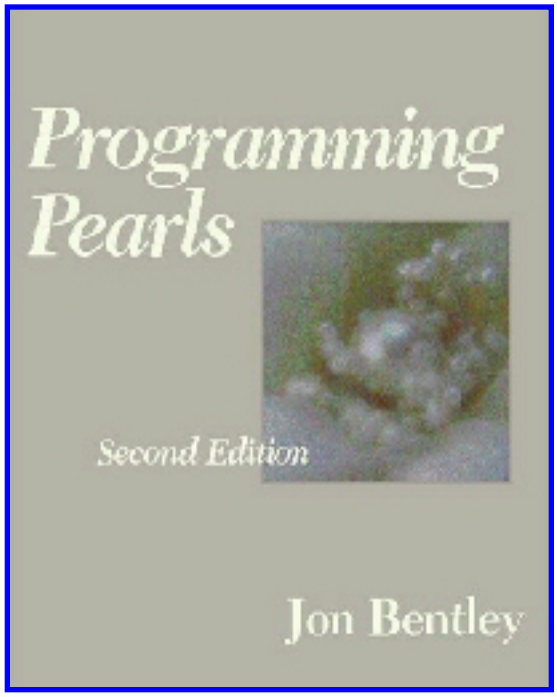
Programming Pearls, Second Edition



by Jon Bentley.

Addison-Wesley, Inc., 2000.
ISBN 0-201-65788-0.
239 + xi pp. $24.95

This book is a collection of essays about a glamorous aspect of software: programming pearls whose origins lie beyond solid engineering, in the realm of insight and creativity. This book provides a guide for both students and experienced programmers about how to design and create programs, and how to think about programming.

The book is full of small case studies, real examples, and interesting exercises for learning about how to program. This web page contains samples from the whole work for you to investigate. For teachers, the links below lead to some of the central material suitable for classroom use.

Steve McConnell describes the book as ``a celebration of design in the small''. Browse this site to sample it yourself.

NEW What's new on this web site?

From The Book

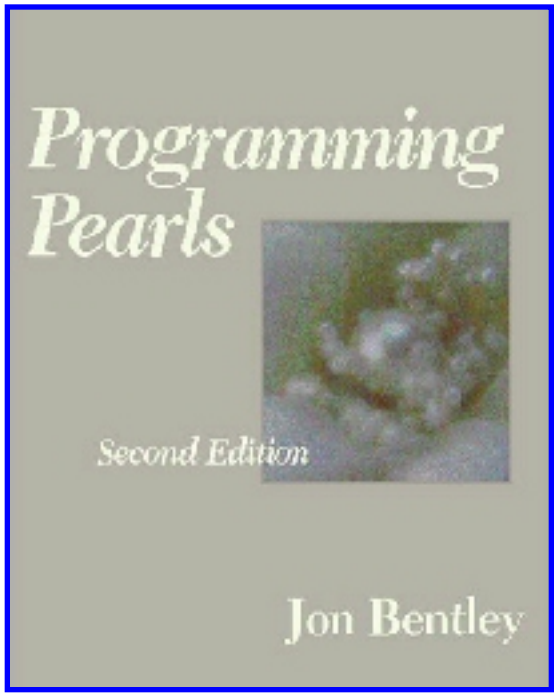
About The Book

Supporting Material

Other Links

- Addison-Wesley Computer & Engineering Publishing Group
- *Programming Pearls* at Addison-Wesley
- Bookstores: Amazon.com, Barnes & Noble, Borders.com, Fatbrain.com, Quantum Books.

What's New on the Programming Pearls Web Site



November 2000

Column 15 is now on the site, complete with a new program for letter-level Markov text, and new examples of word frequencies, long repeated strings, and letter-level and word-level Markov text.

October 2000

The rules for code tuning from my 1982 book *Writing Efficient Programs* are now online, and so is a Powerpoint Show on Cache-Conscious Algorithms and Data Structures.

August 2000

The errata just keeps on growing. If you see errors, please send them in.

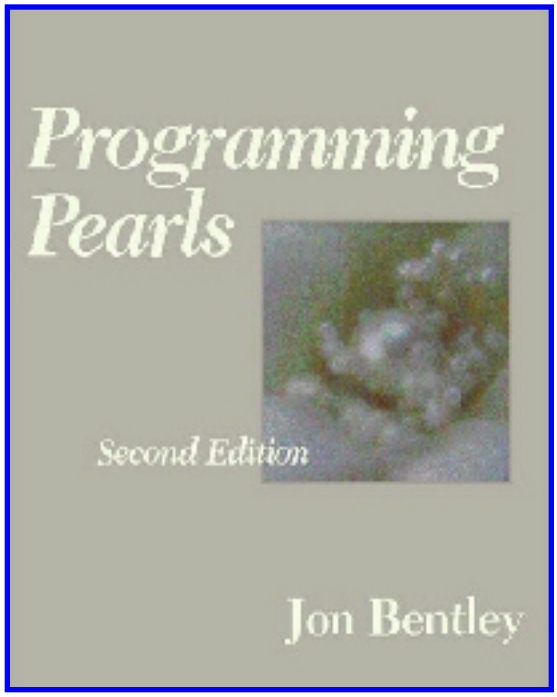
July 2000

Programming Pearls is often used for teaching undergraduates. This page describes how some of the topics in the book can be incorporated into college classrooms.

March 2000

A theme running through the book concerns the Tricks of the Trade. This page describes that topic and contains a Powerpoint Show on the subject.

Strings of Pearls (Column 15 of Programming Pearls)



We are surrounded by strings. Strings of bits make integers and floating-point numbers. Strings of digits make telephone numbers, and strings of characters make words. Long strings of characters make web pages, and longer strings yet make books. Extremely long strings represented by the letters A, C, G and T are in geneticists' databases and deep inside the cells of many readers of this book.

Programs perform a dazzling variety of operations on such strings. They sort them, count them, search them, and analyze them to discern patterns. This column introduces those topics by examining a few classic problems on strings.

The Rest of the Column

These are the remaining sections in the column.

Related Content

The teaching material contains overhead transparencies based on Sections 15.2 and 15.3; the slides are available in both Postscript and Acrobat.

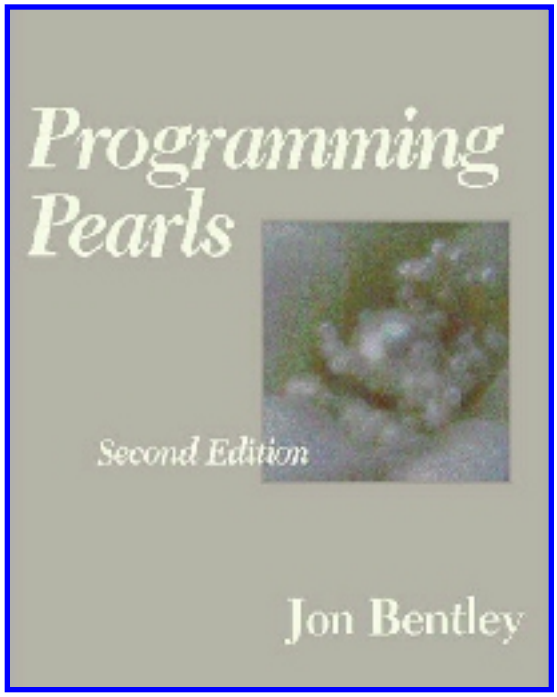
The code for Column 15 contains implementations of the algorithms.

The Solutions to Column 15 give answers for some of the Problems.

This column concentrates on programming techniques, and uses those techniques to build several programs for processing large text files. A few examples of the programs' output in the text illustrate the structure of English documents. This web site contains some additional fun examples, which may give further insight into the structure of the English language. Section 15.1 counts the words in one document; here are more examples of word frequency counts. Section 15.2 searches for large portions of duplicated text; here are more examples of long repeated strings. Section 15.3 describes randomly generated Markov text; these pages contain addtional examples of Markov text, generated at the letter level and word level.

The web references describe several web sites devoted to related topics.

Words (Section 15.1 of Programming Pearls)



Our first problem is to produce a list of the words contained in a document. (Feed such a program a few hundred books, and you have a fine start at a word list for a dictionary.) But what exactly is a word? We'll use the trivial definition of a sequence of characters surrounded by white space, but this means that web pages will contain many ``words" like ``<html>", ``<body>" and `` ". Problem 1 asks how you might avoid such problems.

Our first C++ program uses the *set*s and *string*s of the Standard Template Library, in a slight modification of the program in Solution 1.1:

```
int main(void)
{   set S;
    set::iterator j;
    string t;
    while (cin >> t)
        S.insert(t);
    for (j = S.begin(); j != S.end(); ++j)
        cout << *j << "\n";
    return 0;
}
```

The *while* loop reads the input and *insert*s each word into the set *S* (by the STL specification, duplicates are ignored). The *for* loop then iterates through the set, and writes the words in sorted order. This program is elegant and fairly efficient (more on that topic soon).

Our next problem is to count the number of times each word occurs in the document. Here are the 21 most common words in the King James Bible, sorted in decreasing numeric order and aligned in three columns to save space:

```
the  62053        shall 9756        they 6890
and  38546        he    9506        be   6672
of   34375        unto  8929        is   6595
to   13352        I     8699        with 5949
And  12734        his   8352        not  5840
that 12428        a     7940        all  5238
in   12154        for   7139        thou 4629
```

Almost eight percent of the 789,616 words in the text were the word ``the" (as opposed to 16 percent of the words in this sentence). By our definition of word, ``and" and ``And" have two separate counts.

[Click for more examples of word frequency counts.]

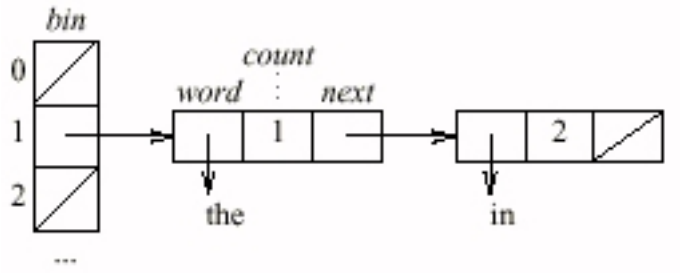
These counts were produced by the following C++ program, which uses the Standard Template Library *map* to associate an integer count with each string:

```
int main(void)
{   map M;
    map::iterator j;
    string t;
    while (cin >> t)
        M[t]++;
    for (j = M.begin(); j != M.end(); ++j)
        cout << j->first << " " << j->second << "\n";
    return 0;
}
```

The *while* statement inserts each word *t* into the map *M* and increments the associated counter (which is initialized to zero at initialization). The *for* statement iterates through the words in sorted order and prints each word (*first*) and its count (*second*).

This C++ code is straightforward, succinct and surprisingly fast. On my machine, it takes 7.6 seconds to process the Bible. About 2.4 seconds go to reading, 4.9 seconds to the insertions, and 0.3 seconds to writing the ouput.

We can reduce the processing time by building our own hash table, using nodes that contain a pointer to a word, a count of how often the word has been seen, and a pointer to the next node in the table. Here is the hash table after inserting the strings ``in", ``the" and ``in", in the unlikely event that both strings hash to 1:



We'll implement the hash table with this C structure:

```
typedef struct node *nodeptr;
typedef struct node {
    char *word;
    int count;
    nodeptr next;
} node;
```

Even by our loose definition of ``word", the Bible has only 29,131 distinct words. We'll follow the old lore of using a prime number near that for our hash table size, and the popular multiplier of 31:

```
#define NHASH 29989
#define MULT 31
```

```
nodeptr bin[NHASH];
```

Our hash function maps a string to a positive integer less than *NHASH*:

```
unsigned int hash(char *p)
    unsigned int h = 0
    for ( ; *p; p++)
        h = MULT * h + *p
    return h % NHASH
```

Using *unsigned* integers ensures that *h* remains positive.

The *main* function initializes every bin to *NULL*, reads the word and increments the count of each, then iterates through the hash table to write the (unsorted) words and counts:

```
int main(void)
    for i = [0, NHASH)
        bin[i] = NULL
    while scanf("%s", buf) != EOF
        incword(buf)
    for i = [0, NHASH)
        for (p = bin[i]; p != NULL; p = p->next)
            print p->word, p->count
    return 0
```

The work is done by *incword*, which increments the count associated with the input word (and initializes it if it is not already there):

```
void incword(char *s)
    h = hash(s)
    for (p = bin[h]; p != NULL; p = p->next)
        if strcmp(s, p->word) == 0
            (p->count)++
            return
    p = malloc(sizeof(hashnode))
    p->count = 1
    p->word = malloc(strlen(s)+1)
    strcpy(p->word, s)
    p->next = bin[h]
    bin[h] = p
```

The *for* loop looks at every node with the same hash value. If the word is found, its count is incremented and the function returns. If the word is not found, the function makes a new node, allocates space and copies the string (experienced C programmers would use *strdup* for the task), and inserts the node at the front of the list.

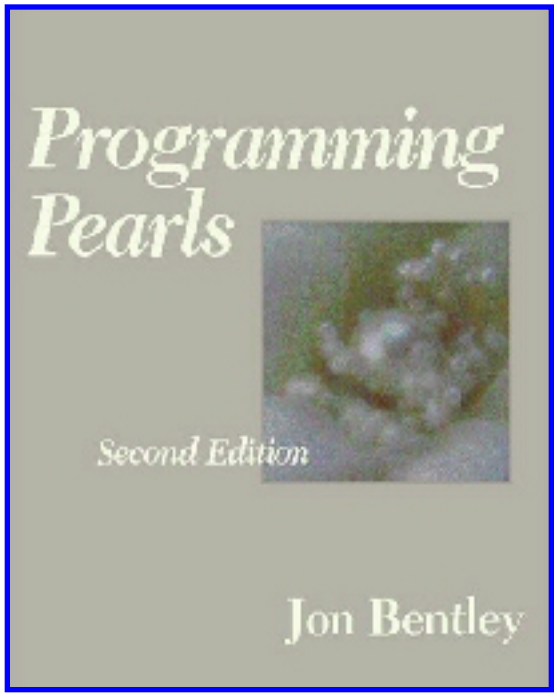
This C program takes about 2.4 seconds to read its input (the same as the C++ version), but only 0.5 seconds for the insertions (down from 4.9) and only 0.06 seconds to write the output (down from 0.3). The complete run time is

3.0 seconds (down from 7.6), and the processing time is 0.55 seconds (down from 5.2). Our custom-made hash table (in 30 lines of C) is an order of magnitude faster than the maps from the C++ Standard Template Library.

This little exercise illustrates the two main ways to represent sets of words. Balanced search trees operate on strings as indivisible objects; these structures are used in most implementations of the STL's *set*s and *map*s. They always keep the elements in sorted order, so they can efficiently perform operations such as finding a predecessor or reporting the elements in order. Hashing, on the other hand, peeks inside the characters to compute a hash function, and then scatters keys across a big table. It is very fast on the average, but it does not offer the worst-case performance guarantees of balanced trees, or support other operations involving order. 在恶劣情况下，Hash的所有key相同，而平衡树不会，并且Hash不支持按字母排序等

Next: Section 15.2. Phrases.

Problems (Section 15.5 of Programming Pearls)



The Solutions to Column 15 give answers for some of these problems.

1. Throughout this column we have used the simple definition that words are separated by white space. Many real documents, such as those in HTML or RTF, contain formatting commands. How could you deal with such commands? Are there any other tasks that you might need to perform?

2. On a machine with ample main memory, how could you use the C++ STL *set*s or *map*s to solve the searching problem in Section 13.8? How much memory does it consume compared to McIlroy's structure?

3. How much speedup can you achieve by incorporating the special-purpose *malloc* of Solution 9.2 into the hashing program of Section 15.1?

4. When a hash table is large and the hash function scatters the data well, almost every list in the table has only a few elements. If either of these conditions is violated, though, the time spent searching down the list can be substantial. When a new string is not found in the hash table in Section 15.1, it is placed at the front of the list. To simulate hashing trouble, set *NHASH* to 1 and experiment with this and other list strategies, such as appending to the end of the list, or moving the most recently found element to the front of the list.

5. When we looked at the output of the word frequency programs in Section 15.1, it was most interesting to print the words in decreasing frequency. How would you modify the C and C++ programs to accomplish this task? How would you print only the *M* most common words, where *M* is a constant such as 10 or 1000?

6. Given a new input string, how would you search a suffix array to find the longest match in the stored text? How would you build a GUI interface for this task?

7. Our program for finding duplicated strings was very fast for ``typical" inputs, but it can be very slow (greater than quadratic) for some inputs. Time the program on such an input. Do such inputs ever arise in practice?

8. How would you modify the program for finding duplicated strings to find the longest string that occurs more than *M* times?

9. Given two input texts, find the longest string that occurs in both.

10. Show how to reduce the number of pointers in the duplication program by pointing only to suffixes that start on word boundaries. What effect does this have on the output produced by the program?

11. Implement a program to generate letter-level Markov text.

12. How would you use the tools and techniques of Section 15.1 to generate (order-0 or non-Markov) random text?

13. The program for generating word-level Markov text is at this book's web site. Try it on some of your documents.

14. How could you use hashing to speed up the Markov program?

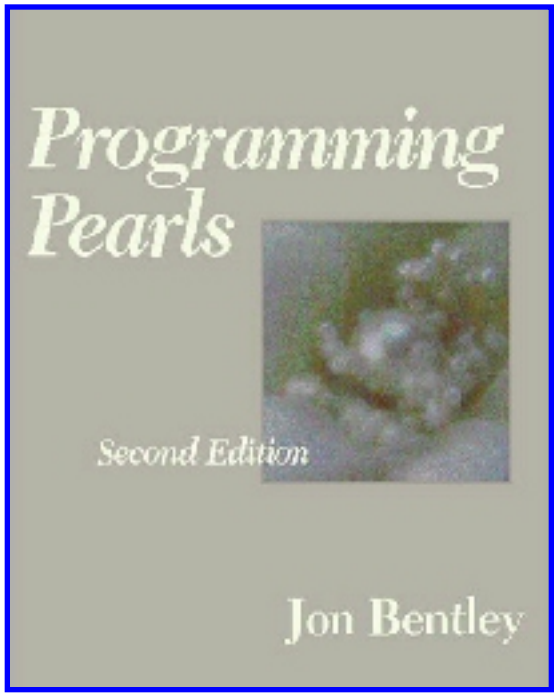
15. Shannon's quote in Section 15.3 describes the algorithm he used to construct Markov text; implement that algorithm in a program. It gives a good approximation to the Markov frequencies, but not the exact form; explain why not. Implement a program that scans the entire string from scratch to generate each word (and thereby uses the true frequencies).

16. How would you use the techniques of this column to assemble a word list for a dictionary (the problem that Doug McIlroy faced in Section 13.8)? How would you build a spelling checker without using a dictionary? How would you build a grammar checker without using rules of grammar?

17. Investigate how techniques related to *k*-gram analysis are used in applications such as speech recognition and data compression.

Next: Section 15.6. Further Reading.

Solutions
(To Column 1 of Programming Pearls)



1. This C program uses the Standard Library *qsort* to sort a file of integers.

```
int intcomp(int *x, int *y)
{   return *x - *y; }

int a[1000000];
int main(void)
{   int i, n=0;
    while (scanf("%d", &a[n]) != EOF)
        n++;
    qsort(a, n, sizeof(int), intcomp);
    for (i = 0; i < n; i++)
        printf("%d\n", a[i]);
    return 0;
}
```

This C++ program uses the *set* container from the Standard Template Library for the same job.

```
int main(void)
{   set S;
    int i;
    set::iterator j;
    while (cin >> i)
        S.insert(i);
    for (j = S.begin(); j != S.end(); ++j)
        cout << *j << "\n";
    return 0;
}
```

Solution 3 sketches the performance of both programs.

2. These functions use the constants to set, clear and test the value of a bit:

```
#define BITSPERWORD 32
#define SHIFT 5
#define MASK 0x1F
#define N 10000000
int a[1 + N/BITSPERWORD];

void set(int i) {        a[i>>SHIFT] |=  (1<<(i & MASK)); }
```

```
void clr(int i) {        a[i>>SHIFT] &= ~(1<<(i & MASK)); }
int  test(int i){ return a[i>>SHIFT] &   (1<<(i & MASK)); }
```

3. This C code implements the sorting algorithm, using the functions defined in Solution 2.

```
int main(void)
{   int i;
    for (i = 0; i < N; i++)
        clr(i);
    while (scanf("%d", &i) != EOF)
        set(i);
    for (i = 0; i < N; i++)
        if (test(i))
            printf("%d\n", i);
    return 0;
}
```

I used the program in Solution 4 to generate a file of one million distinct positive integers, each less than ten million. This table reports the cost of sorting them with the system command-line sort, the C++ and C programs in Solution 1, and the bitmap code:

	System Sort	**C++/STL**	**C/qsort**	**C/bitmaps**
Total Secs	89	38	12.6	10.7
Compute Secs	79	28	2.4	.5
Megabytes	.8	70	4	1.25

The first line reports the total time, and the second line subtracts out the 10.2 seconds of input/output required to read and write the files. Even though the general C++ program uses 50 times the memory and CPU time of the specialized C program, it requires just half the code and is much easier to extend to other problems.

4. See Column 12, especially Problem 12.8. This code assumes that *randint(l, u)* returns a random integer in *l..u*.

```
for i = [0, n)
    x[i] = i
for i = [0, k)
    swap(i, randint(i, n-1))
    print x[i]
```

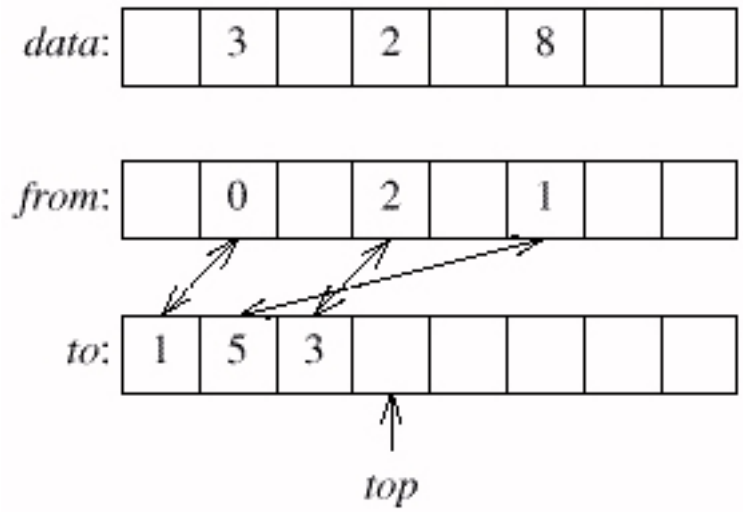
The *swap* function exchanges the two elements in *x*. The *randint* function is discussed in detail in Section 12.1.

5. Representing all ten million numbers with a bitmap requires that many bits, or 1.25 million bytes. Employing the fact that no phone numbers begin with the digits zero or one reduces the memory to exactly one million bytes. Alternatively, a two-pass algorithm first sorts the integers 0 through 4,999,999 using 5,000,000/8 = 625,000 words of storage, then sorts 5,000,000 through 9,999,999 in a second pass. A *k*-pass algorithm sorts at most *n* nonrepeated positive integers less than *n* in time *kn* and space *n*/*k*.

6. If each integer appears at most ten times, then we can count its occurrences in a four-bit half-byte (or nybble).

Using the solution to Problem 5, we can sort the complete file in a single pass with 10,000,000/2 bytes, or in *k* passes with 10,000,000/2*k* bytes.

9. The effect of initializing the vector *data*[0..*n*-1] can be accomplished with a signature contained in two additional *n*-element vectors, *from* and *to*, and an integer *top*. If the element *data*[*i*] has been initialized, then *from*[*i*] < *top* and *to*[*from*[*i*]]=*i*. Thus *from* is a simple signature, and *to* and *top* together make sure that *from* is not accidentally signed by the random contents of memory. Blank entries of *data* are uninitialized in this picture:



The variable *top* is initially zero; the array element *i* is first accessed by the code

```
from[i] = top
to[top] = i
data[i] = 0
top++
```

This problem and solution are from Exercise 2.12 of Aho, Hopcroft and Ullman's *Design and Analysis of Computer Algorithms*, published by Addison-Wesley in 1974. It combines key indexing and a wily signature scheme. It can be used for matrices as well as vectors.

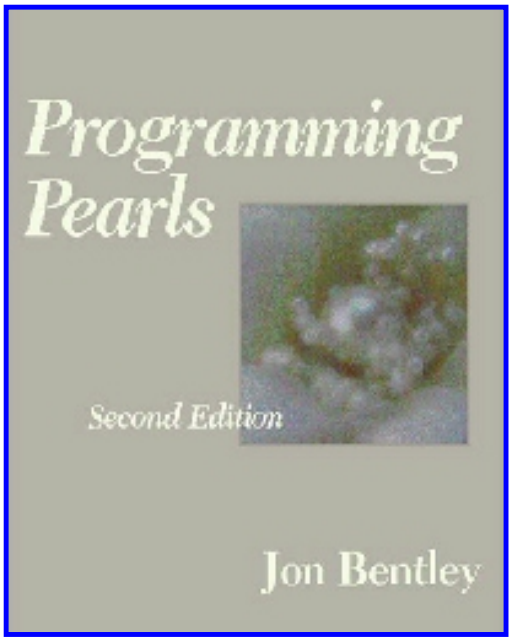
10. The store placed the paper order forms in a 10x10 array of bins, using the last two digits of the customer's phone number as the hash index. When the customer telephoned an order, it was placed in the proper bin. When the customer arrived to retrieve the merchandise, the salesperson sequentially searched through the orders in the appropriate bin -- this is classical ``open hashing with collision resolution by sequential search''. The last two digits of the phone number are quite close to random and therefore an excellent hash function, while the first two digits would be a horrible hash function -- why? Some municipalities use a similar scheme to record deeds in sets of record books.

11. The computers at the two facilities were linked by microwave, but printing the drawings at the test base would have required a printer that was very expensive at the time. The team therefore drew the pictures at the main plant, photographed them, and sent 35mm film to the test station by carrier pigeon, where it was enlarged and printed photographically. The pigeon's 45-minute flight took half the time of the car, and cost only a few dollars per day. During the 16 months of the project the pigeons transmitted several hundred rolls of film, and only two were lost (hawks inhabit the area; no classified data was carried). Because of the low price of modern printers, a current solution to the problem would probably use the microwave link.

12. According to the urban legend, the Soviets solved their problem with a pencil, of course. For background on the

true story, learn about the Fisher Space Pen company. The company was founded in 1948, and its writing instruments have been used by the Russian Space Agency, underwater explorers, and Himalayan climbing expeditions.

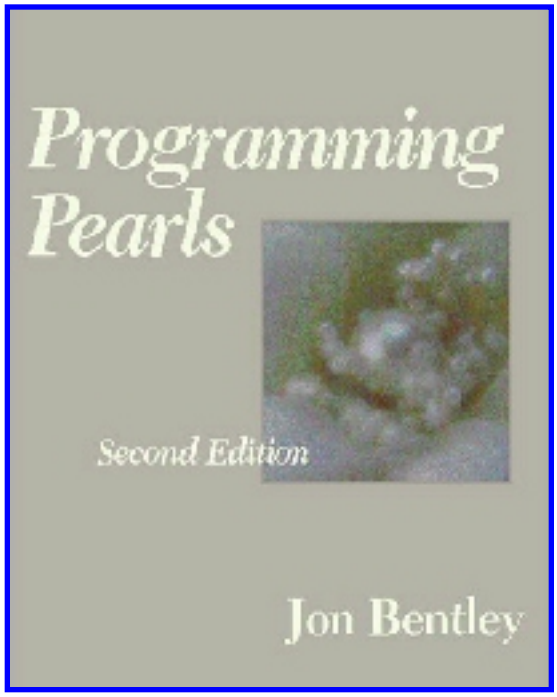
Word Frequencies (Illustrating Section 15.1 of Programming Pearls)



Section 15.1 describes word frequencies and gives an example. Here are some more examples, generated by this program from the King James Bible (789,616 words), the Iliad (153,210 words), and Programming Pearls (79,525 words). This table shows the 40 most common words, and their percentage in the document.

BIBLE	%	ILIAD	%	PEARLS	%
the	7.86	the	6.25	the	5.64
and	4.88	and	4.23	of	2.76
of	4.35	of	3.64	a	2.43
to	1.69	to	2.15	to	2.18
And	1.61	his	1.62	and	2.01
that	1.57	he	1.60	in	1.96
in	1.54	in	1.42	is	1.36
shall	1.24	a	1.17	that	1.20
he	1.20	with	1.04	The	0.91
unto	1.13	that	0.94	for	0.83
I	1.10	you	0.91	by	0.67
his	1.06	for	0.89	it	0.57
a	1.01	him	0.84	this	0.56
for	0.90	I	0.83	with	0.51
they	0.87	as	0.79	an	0.50
be	0.84	was	0.77	program	0.48
is	0.84	they	0.66	on	0.48
with	0.75	on	0.64	be	0.47
not	0.74	from	0.64	are	0.47
all	0.66	but	0.58	as	0.45
thou	0.59	son	0.58	we	0.43
thy	0.56	had	0.58	at	0.39
was	0.56	not	0.56	can	0.39
it	0.56	their	0.56	time	0.38
which	0.54	it	0.56	I	0.36
my	0.52	is	0.50	was	0.35
LORD	0.50	will	0.49	you	0.33
their	0.49	have	0.48	code	0.33
have	0.48	all	0.47	from	0.28
will	0.47	by	0.47	or	0.26
from	0.45	them	0.46	its	0.26
ye	0.45	were	0.43	one	0.25
them	0.44	at	0.43	This	0.25
as	0.41	who	0.42	array	0.25
him	0.40	my	0.40	have	0.25
are	0.36	so	0.40	two	0.25
upon	0.34	your	0.38	each	0.25
were	0.34	when	0.37	first	0.24
by	0.32	be	0.36	search	0.24
when	0.31	upon	0.35	function	0.23

Phrases (Section 15.2 of Programming Pearls)



Words are the basic component of documents, and many important problems can be solved by searching for words. Sometimes, however, I search long strings (my documents, or help files, or web pages, or even the entire web) for phrases, such as ``substring searching" or ``implicit data structures".

How would you search through a large body of text to find ``a phrase of several words"? If you had never seen the body of text before, you would have no choice but to start at the beginning and scan through the whole input; most algorithms texts describe many approaches to this ``substring searching problem".

Suppose, though, that you had a chance to preprocess the body of text before performing searches. You could make a hash table (or search tree) to index every distinct word of the document, and store a list of every occurrence of each word. Such an ``inverted index" allows a program to look up a given word quickly. One can look up phrases by intersecting the lists of the words they contain, but this is subtle to implement and potentially slow. (Some web search engines do, however, take exactly this approach.)

We'll turn now to a powerful data structure and apply it to a small problem: given an input file of text, find the longest duplicated substring of characters in it. For instance, the longest repeated string in ``Ask not what your country can do for you, but what you can do for your country" is `` can do for you", with `` your country" a close second place. How would you write a program to solve this problem?

[Click for more examples of long repeated strings]

This problem is reminiscent of the anagram problem that we saw in Section 2.4. If the input string is stored in *c[0..n-1]*, then we could start by comparing every pair of substrings using pseudocode like this

```
maxlen = -1
for i = [0, n)
    for j = (i, n)
        if (thislen = comlen(&c[i], &c[j])) > maxlen
            maxlen = thislen
            maxi = i
            maxj = j
```

The *comlen* function returns the length that its two parameter strings have in common, starting with their first characters:

```
int comlen(char *p, char *q)
```

```
    i = 0
    while *p && (*p++ == *q++)
        i++
    return i
```

Because this algorithm looks at all pairs of substrings, it takes time proportional to n^2, at least. We might be able to speed it up by using a hash table to search for words in the phrases, but we'll instead take an entirely new approach.

Our program will process at most *MAXN* characters, which it stores in the array *c*:

```
#define MAXN 5000000
char c[MAXN], *a[MAXN];
```

We'll use a simple data structure known as a ``suffix array''; the structure has been used at least since the 1970's, though the term was introduced in the 1990's. The structure is an array *a* of pointers to characters. As we read the input, we initialize *a* so that each element points to the corresponding character in the input string:

```
while (ch = getchar()) != EOF
    a[n] = &c[n]
    c[n++] = ch
c[n] = 0
```

The final element of *c* contains a null character, which terminates all strings.

The element *a[0]* points to the entire string; the next element points to the suffix of the array beginning with the second character, and so on. On the input string ``banana'', the array will represent these suffixes:

```
a[0]: banana
a[1]: anana
a[2]: nana
a[3]: ana
a[4]: na
a[5]: a
```

The pointers in the array *a* together point to every suffix in the string, hence the name ``suffix array''.

If a long string occurs twice in the array *c*, it appears in two different suffixes. We will therefore sort the array to bring together equal suffixes (just as sorting brought together anagrams in Section 2.4). The ``banana'' array sorts to

```
a[0]: a
a[1]: ana
a[2]: anana
a[3]: banana
a[4]: na
a[5]: nana
```

We can then scan through this array comparing adjacent elements to find the longest repeated string, which in this case is ``ana".

We'll sort the suffix array with the *qsort* function:

```
qsort(a, n, sizeof(char *), pstrcmp)
```

The *pstrcmp* comparison function adds one level of indirection to the library *strcmp* function. This scan through the array uses the *comlen* function to count the number of letters that two adjacent words have in common:

```
for i = [0, n)
    if comlen(a[i], a[i+1]) > maxlen
        maxlen = comlen(a[i], a[i+1])
        maxi = i
printf("%.*s\n", maxlen, a[maxi])
```

The *printf* statement uses the ``*" precision to print *maxlen* characters of the string.

I ran the resulting program to find the longest repeated string in the 807,503 characters in Samuel Butler's translation of Homer's *Iliad*. The program took 4.8 seconds to locate this string:

> whose sake so many of the Achaeans have died at Troy, far from their homes? Go about at once among the host, and speak fairly to them, man by man, that they draw not their ships into the sea.

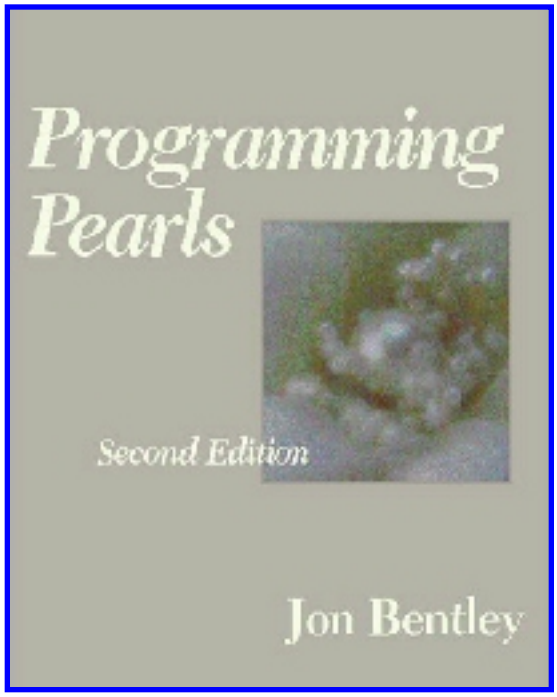
The text first occurs when Juno suggests it to Minerva as an argument that might keep the Greeks (Achaeans) from departing from Troy; it occurs shortly thereafter when Minerva repeats the argument verbatim to Ulysses. On this and other typical text files of n characters, the algorithm runs in $O(n \log n)$ time, due to sorting.

[Click for more examples of long repeated strings]

Suffix arrays represent every substring in n characters of input text using the text itself and n additional pointers. Problem 6 investigates how suffix arrays can be used to solve the substring searching problem. We'll turn now to a more subtle application of suffix arrays.

Next: Section 15.3. Generating Random Text.

Long Repeated Strings (Illustrating Section 15.2 of Programming Pearls)



Section 15.2 describes long repeated strings in text and gives an example. Here are some more examples, generated by this program from several sources.

King James Bible

Verse Numbers Included

the house of his precious things, the silver, and the gold, and the spices, and the precious ointment, and all the house of his armour, and all that was found in his treasures: there was nothing in his house, nor in all his dominion, that Hezekiah shewed them not.

This text is found in 2 Kings 20:13 and in Isaiah 39:2. Each text line in the original file begins with the chapter and verse (i.e., ``GEN 1:1 In the beginning God created ..."). Long repeated strings therefore could not cross verse boundaries; the next experiment deleted those identifiers.

Verse Numbers Excluded

, offered: His offering was one silver charger, the weight whereof was an hundred and thirty shekels, one silver bowl of seventy shekels, after the shekel of the sanctuary; both of them full of fine flour mingled with oil for a meat offering: One golden spoon of ten shekels, full of incense: One young bullock, one ram, one lamb of the first year, for a burnt offering: One kid of the goats for a sin offering: And for a sacrifice of peace offerings, two oxen, five rams, five he goats, five lambs of the first year: this was the offering of Ahi

Numbers 7 describes offerings made over a period of twelve days; much of this string appears twelve times.

Longest String That Occurs Twelve Times

, full of incense: One young bullock, one ram, one lamb of the first year, for a burnt offering: One kid of the goats for a sin offering: And for a sacrifice of peace offerings, two oxen, five rams, five he goats, five lambs of the first year: this was the offering of

This string occurs twelve times in Numbers 7. This string was computed using the method of Solution 15.8.

Programming Pearls

The Entire Text

6, 8, 12, 17-18, 51-55, 59, 62, 70-72, 82, 87-98, 116, 119, 121, 128, 137, 162, 164, 187-189, 206, 210, 214, 221, 230

I was surprised to find that the longest repeated string in the book was in the index. The same sequence of numbers are repeated for the entries for ``experiments'', ``run time'' and ``time, run''.

Index Excluded

expression is costly, replace it by an algebraically equivalent expression that is cheaper to evaluate. I

This text appears in both the Logic Rules and the Expression Rules of Appendix 4: Rules for Code Tuning.

The Iliad

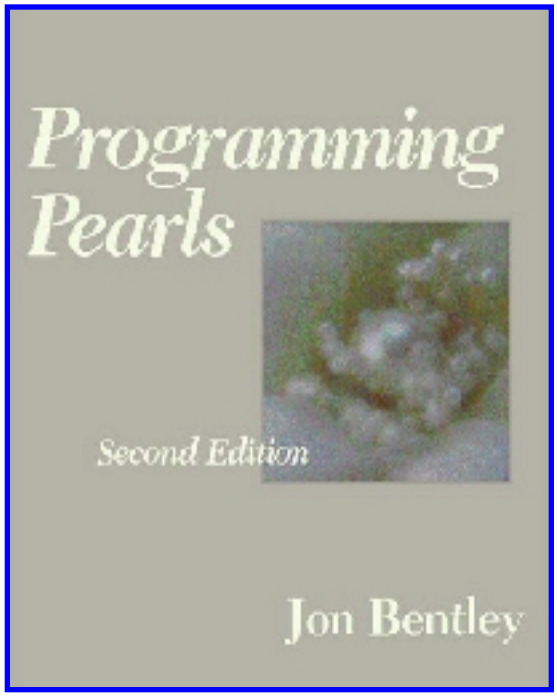
The Entire Text

whose sake so many of the Achaeans have died at Troy, far from their homes? Go about at once among the host, and speak fairly to them, man by man, that they draw not their ships into the sea.

This example (on Samuel Butler's translation of Homer's Iliad) was used in Section 15.2. describes long repeated strings in text and gives The text first occurs when Juno suggests it to Minerva as an argument that might keep the Greeks (Achaeans) from departing from Troy; it occurs shortly thereafter when Minerva repeats the argument verbatim to Ulysses.

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* longdup.c -- Print longest string duplicated M times */

#include <stdlib.h>
#include <string.h>
#include <stdio.h>

int pstrcmp(char **p, char **q)
{   return strcmp(*p, *q); }

int comlen(char *p, char *q)
{       int i = 0;
        while (*p && (*p++ == *q++))
                i++;
        return i;
}

#define M 1
#define MAXN 5000000
char c[MAXN], *a[MAXN];

int main()
{   int i, ch, n = 0, maxi, maxlen = -1;
    while ((ch = getchar()) != EOF) {
        a[n] = &c[n];
        c[n++] = ch;
    }
    c[n] = 0;
    qsort(a, n, sizeof(char *), pstrcmp);
    for (i = 0; i < n-M; i++)
        if (comlen(a[i], a[i+M]) > maxlen) {
            maxlen = comlen(a[i], a[i+M]);
            maxi = i;
        }
    printf("%.*s\n", maxlen, a[maxi]);
    return 0;
}
```

Generating Text (Section 15.3 of Programming Pearls)



How can you generate random text? A classic approach is to let loose that poor monkey on his aging typewriter. If the beast is equally likely to hit any lower case letter or the space bar, the output might look like this:

> uzlpcbizdmddk njsdzyyvfgxbgjjgbtsak
> rqvpgnsbyputvqqdtmgltz ynqotqigexjumqphujcfwn ll
> jiexpyqzgsdllgcoluphl sefsrvqqytjakmav bfusvirsjl
> wprwqt

This is pretty unconvincing English text.

If you count the letters in word games (like Scrabble or Boggle), you will notice that there are different numbers of the various letters. There are many more A's, for instance, than there are Z's. A monkey could produce more convincing text by counting the letters in a document -- if A occurs 300 times in the text while B occurs just 100 times, then the monkey should be 3 times more likely to type an A than a B. This takes us a small step closer to English:

> saade ve mw hc n entt da k eethetocusosselalwo gx fgrsnoh,tvettaf aetnlbilo fc lhd okleutsndyeoshtbogo
> eet ib nheaoopefni ngent

Most events occur in context. Suppose that we wanted to generate randomly a year's worth of Fahrenheit temperature data. A series of 365 random integers between 0 and 100 wouldn't fool the average observer. We could be more convincing by making today's temperature a (random) function of yesterday's temperature: if it is 85 degrees today, it is unlikely to be 15 degrees tomorrow.

The same is true of English words: if this letter is a Q, then the next letter is quite likely to be a U. A generator can make more interesting text by making each letter a random function of its predecessor. We could, therefore, read a sample text and count how many times every letter follows an A, how many times they follow a B, and so on for each letter of the alphabet. When we write the random text, we produce the next letter as a random function of the current letter. The ``order-1'' text was made by exactly this scheme:

> *Order-1:* t I amy, vin. id wht omanly heay atuss n macon aresethe hired boutwhe t, tl, ad torurest t plur I wit hengamind tarer-plarody thishand.
>
> *Order-2:* Ther I the heingoind of-pleat, blur it dwere wing waske hat trooss. Yout lar on wassing, an sit." "Yould," "I that vide was nots ther.
>
> *Order-3:* I has them the saw the secorrow. And wintails on my my ent, thinks, fore voyager lanated the been elsed helder was of him a very free bottlemarkable,

Order-4: His heard." "Exactly he very glad trouble, and by Hopkins! That it on of the who difficentralia. He rushed likely?" "Blood night that.

[Click for more examples of letter-level Markov text]

We can extend this idea to longer sequences of letters. The order-2 text was made by generating each letter as a function of the two letters preceding it (a letter pair is often called a digram). The digram TH, for instance, is often followed in English by the vowels A, E, I, O, U and Y, less frequently by R and W, and rarely by other letters. The order-3 text is built by choosing the next letter as a function of the three previous letters (a trigram). By the time we get to the order-4 text, most words are English, and you might not be surprised to learn that it was generated from a Sherlock Holmes story (``The Adventure of Abbey Grange''). A classically educated reader of a draft of this column commented that this sequence of fragments reminded him of the evolution from Old English to Victorian English.

Readers with a mathematical background might recognize this process as a Markov chain. One state represents each *k*-gram, and the odds of going from one to another don't change, so this is a ``finite-state Markov chain with stationary transition probabilities''.

We can also generate random text at the word level. The dumbest approach is to spew forth the words in a dictionary at random. A slightly better approach reads a document, counts each word, and then selects the next word to be printed with the appropriate probability. (The programs in Section 15.1 use tools appropriate for such tasks.) We can get more interesting text, though, by using Markov chains that take into account a few preceding words as they generate the next word. Here is some random text produced after reading a draft of the first 14 columns of this book:

Order-1: The table shows how many contexts; it uses two or equal to the sparse matrices were not chosen. In Section 13.1, for a more efficient that ``the more time was published by calling recursive structure translates to build scaffolding to try to know of selected and testing and more robust and a binary search).

Order-2: The program is guided by verification ideas, and the second errs in the STL implementation (which guarantees good worst-case performance), and is especially rich in speedups due to Gordon Bell. Everything should be to use a macro: for *n=10,000*, its run time; that point Martin picked up from his desk

Order-3: A Quicksort would be quite efficient for the main-memory sorts, and it requires only a few distinct values in this particular problem, we can write them all down in the program, and they were making progress towards a solution at a snail's pace.

The order-1 text is almost readable aloud, while the order-3 text consists of very long phrases from the original input, with random transitions between them. For purposes of parody, order-2 text is usually juiciest.

[Click for more examples of word-level Markov text]

I first saw letter-level and word-level order-k approximations to English text in Shannon's 1948 classic *Mathematical Theory of Communication*. Shannon writes, ``To construct [order-1 letter-level text] for example, one opens a book at random and selects a letter at random on the page. This letter is recorded. The book is then opened to another page and one reads until this letter is encountered. The succeeding letter is then recorded. Turning to another page this second letter is searched for and the succeeding letter recorded, etc. A similar process was used for [order-1 and order-2 letter-level text, and order-0 and order-1 word-level text]. It would be interesting

if further approximations could be constructed, but the labor involved becomes enormous at the next stage."

A program can automate this laborious task. Our C program to generate order-k Markov chains will store at most five megabytes of text in the array *inputchars*:

```
int k = 2;
char inputchars[5000000];
char *word[1000000];
int nword = 0;
```

We could implement Shannon's algorithm directly by scanning through the complete input text to generate each word (though this might be slow on large texts). We will instead employ the array *word* as a kind of suffix array pointing to the characters, except that it starts only on word boundaries (a common modification). The variable *nword* holds the number of words. We read the file with this code:

```
word[0] = inputchars
while scanf("%s", word[nword]) != EOF
    word[nword+1] = word[nword] + strlen(word[nword]) + 1
    nword++
```

Each word is appended to *inputchars* (no other storage allocation is needed), and is terminated by the null character supplied by *scanf*.

After we read the input, we will sort the *word* array to bring together all pointers that point to the same sequence of *k* words. This function does the comparisons:

```
int wordncmp(char *p, char* q)
    n = k
    for ( ; *p == *q; p++, q++)
        if (*p == 0 && --n == 0)
            return 0
    return *p - *q
```

It scans through the two strings while the characters are equal. At every null character, it decrements the counter *n* and returns equal after seeing *k* identical words. When it finds unequal characters, it returns the difference.

After reading the input, we append *k* null characters (so the comparison function doesn't run off the end), print the first *k* words in the document (to start the random output), and call the sort:

```
for i = [0, k)
    word[nword][i] = 0
for i = [0, k)
    print word[i]
qsort(word, nword, sizeof(word[0]), sortcmp)
```

The *sortcmp* function, as usual, adds a level of indirection to its pointers.

Our space-efficient structure now contains a great deal of information about the *k*-grams in the text. If *k* is 1 and the input text is ``of the people, by the people, for the people'', the *word* array might look like this:

```
word[0]: by the
word[1]: for the
word[2]: of the
word[3]: people
word[4]: people, for
word[5]: people, by
word[6]: the people,
word[7]: the people
word[8]: the people,
```

For clarity, this picture shows only the first *k+1* words pointed to by each element of *word*, even though more words usually follow. If we seek a word to follow the phrase ``the'', we look it up in the suffix array to discover three choices: ``people,'' twice and ``people'' once.

We may now generate nonsense text with this pseudocode sketch:

```
phrase = first phrase in input array
loop
    perform a binary search for phrase in word[0..nword-1]
    for all phrases equal in the first k words
        select one at random, pointed to by p
    phrase = word following p
    if k-th word of phrase is length 0
        break
    print k-th word of phrase
```

We initialize the loop by setting *phrase* to the first characters in the input (recall that those words were already printed on the output file). The binary search uses the code in Section 9.3 to locate the first occurrence of *phrase* (it is crucial to find the very first occurrence; the binary search of Section 9.3 does just this). The next loop scans through all equal phrases, and uses Solution 12.10 to select one of them at random. If the *k*-th word of that phrase is of length zero, the current phrase is the last in the document, so we break the loop.

The complete pseudocode implements those ideas, and also puts an upper bound on the number of words it will generate:

```
phrase = inputchars
for (wordsleft = 10000; wordsleft > 0; wordsleft--)
    l = -1
    u = nword
    while l+1 != u
        m = (l + u) / 2
        if wordncmp(word[m], phrase) < 0
            l = m
        else
            u = m
```

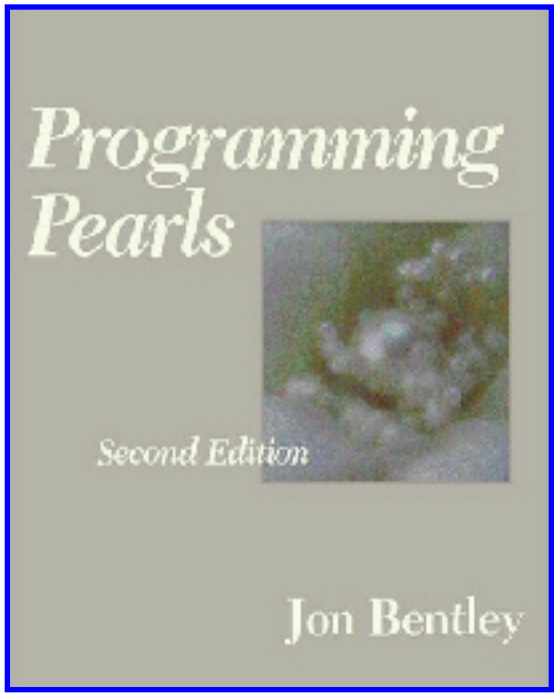
```
for (i = 0; wordncmp(phrase, word[u+i]) == 0; i++)
    if rand() % (i+1) == 0
        p = word[u+i]
phrase = skip(p, 1)
if strlen(skip(phrase, k-1)) == 0
    break
print skip(phrase, k-1)
```

Chapter 3 of Kernighan and Pike's *Practice of Programming* (described in Section 5.9) is devoted to the general topic of ``Design and Implementation''. They build their chapter around the problem of word-level Markov-text generation because ``it is typical of many programs: some data comes in, some data goes out, and the processing depends on a little ingenuity.'' They give some interesting history of the problem, and implement programs for the task in C, Java, C++, Awk and Perl.

The program in this section compares favorably with their C program for the task. This code is about half the length of theirs; representing a phrase by a pointer to *k* consecutive words is space-efficient and easy to implement. For inputs of size near a megabyte, the two programs are of roughly comparable speed. Because Kernighan and Pike use a slightly larger structure and make extensive use of the inefficient *malloc*, the program in this column uses an order of magnitude less memory on my system. If we incorporate the speedup of Solution 14 and replace the binary search and sorting by a hash table, the program in this section becomes about a factor of two faster (and increases memory use by about 50%).

Next: Section 15.4. Principles.

Letter-Level Markov Text (Illustrating Section 15.3 of Programming Pearls)



Section 15.3 describes letter-level Markov text and gives a few examples. Here are some more examples, generated by this program from several sources.

The Book of Romans (King James Version)

Order-1: Runs ch g Sprif ighaifay fe; ie llathis, fur Gos ngithigh f Lom sist aminth uces yom Je Movin th we hof I juthe peathor ly dis igstredithe the, ist oth t s th uth re? hathlivicoto burecon on, Forer mautht w Lowepe sucthet nese, optro cl cth, alditheakengerdisa of bea wepis nd on ichencouteeinthtemef id, n thard f heans rat ithooowh, halang s mail whet alf it th, avoterd myond che su: pulece t hethy tinen, Forund ur; h, y tousio

Order-2: For unto yousay law to do retway hein: thein ther on, Who dopento the he but wit forethered Jesin: ach minto at of the livence, wholet ye judge of heren the me of th. Ford fulne, andethe a refor oure dowe God he hathess. Becom thers, bed hat the sing ousne any med boanined that wer praoh, aryphe knot that law of the ef: I mints Chrom home insgrahalso ded ford to wits norks: but san, wrace to thento the sid; have all the as ith: buth. To antiondignam to unte of So now the rise selievelesed law; As sh to but is of tore wrink mignot him pe: ford me, to prough ithe ard be. Forke eare st; expe counto the pe he is abled an he again forears, the be not de, lesion thre th, Whou witilethaturer an ofesusned to pon aw.

Order-3: For righbour in from her own Sion, There not, which confidentillined; For thereignation and thes ves: things is gospel, do nothink wised; and root is of the law? of unto him which hearath ye werefor the est, but child seeth; But neithough them what life. But wise workenned in their we knowing to them whets of Jesurreceivereus, and, and commit is unwise anot, deated are trivet; in put God, thy meat who what want ought, which and thereignor God same by the circumcision. As it wise himselves deate.

Order-4: Therein the law to Gomorrha. Owe not atten for it was lieth believed. The gosperously report? For Israel, not in business shalt not hope. For I receiving the works: unto these to that in my hundred you: for of God, the changed began, which is thou, nor all thing as by the law of Christ uncircumcised the me infirmity againstanding and Sarah's way the carnally preferring inst alive the law, thou the law of sin; and bitten as by ther.

Order-5: What their lusts there in your father, because also maketh not, it is among ought of his he liveth, if it fulfil things which cause division verily, the gospel of God. For I be saved. For they have mercy. Now if thou boast of patience of envy, murder, debate, deceive tree, which are the righteousness. Much more shall rise thankful; but have you into Spain.

King James Bible (Complete Text)

Order-1: Ched t ainone wand LORD, Thenathan g u t; w t Sona t essose Anasesed an trer. send Ege fomongold, she eyothrofo Andecondiore chizande Gouaip prkecofovire wid, g ie ay l Fag h veds, my ye on They, Theayotingaspuly obe wandoplacco Moued wanere hern tiedef hiverdath ade: e oe thalld mive by ifond nd hand ra, omed weleche thant f u an f sththil, s.

Order-2: a grand it the woraelfspit up fir as king thaderld, I slot kins ts, shis thim the mot: and Jestrul par hals? Godsto but knoth, and will thalson me up the thee, shings ind I huse twerphemieved hey me, wasumbrot of thence. And gan wor themplehowify not se, then boat cur ware me: flem pasts, sto therenes: andmen th gooker but he commins with him theathe de a cault I wittereveres? whighte, for of Shinged. Themy un, and whou be froass hild, that unto ances saw eart scom by laints froself. And hall youll led inks carthe Fathrues.

Order-3: Against of Ashekeloverth with his uncill be pill prehold, We came womb? God; my be the the you. O that he the LORD, three and of they may shall before word of the LORD the LORD; forth hast it thousalem. As counto his sould rein this he shalt not side unto hire; Let unto yet, any. The watersiteth Aarone the said unto ther bring in God of Lord of his was day take a sought. And for a raim who is, and on in that that slain of be the wreath them and miliphats signorth me, O Zion end heardern unt their than to shall the it serpennehast the bar; and a greakim, where] in thirt me done for inhabide asting, The LORD, and the spake. Like the had where mand Laisin but he nations.

Order-4: Open he sister daughteousness, whitherefore they present with themselves; When the Kenezites. Therefore shall appointed that yea, Lord GOD. And I may servants or lips. And he sons of God. The left horses anger to my right unto this his king of Enan. And the lastiness. For themself which we made unto his is this born back of Jedaiah, and thy possessions shut him that me. Then I countainst that there of the king of our boast, that he stones, two and fault fell for the house thy water thy is poured with him was a coping the send the trespecia. And God was out offering, and their house, both in the porter our femaleface of Israel inst sat he comingled, and wearingled of come, they kingdoms. And Israel, when the olivers were more three out of Pening of the Lord, and what Jerusalem, And they had turn to all bring David accept, and fellow down it.

Order-5: Thus saith thy man righted, behold, Gaal was thou art that fell do them, and encamped unto the did unto Martha, the height before so doth by them alive: and for thirty and by give our Fathers were made. Flee formed for they that doth commanded thy servants, according against him go not sinned about; and to her, and cast the would to come up against month Abel had come unto the earth. And the LORD unto the cud, but such camel, and the curtain offering: and them, and all thee unruly, came down to them what it us; and joy: And David well their brethren, out from among thee. And it came, bewailed over Israel said unto all gall: in his he. And Jehoiada the cometh the prophesy: And when thou then shekel of the goodly after him, and menservants, which I commandment, and we had done to Egyptians and a loud of their bread, and to Abraham's sin. Howbeit let him not; then spake us the salt? or stretcheth of Abib, the other to you, That hath promise. After with God shall not afraid, Nay, not tell ye of the resist shewed me from all.

Column 10 of Programming Pearls

Order-1: nome atwoce te smstarond toce acthare est juthel vers Th ay theome aytinglyt f vinca tiforachthedabj) thionfo rpobaro ske g, beruthainsse iedif ton eane ustioutinde. titesy s th g ronpromarace s, Wedesimess ctis wan spe thivausprillyton e bougre inorunds RACPore in acora wite ar 202 inductipau caulduinng moucan a alams.Ind onthacooon tepancheplofasicenderysppl ay,0tes s.

Order-2: Eng gingent intessic diver frow hanamidex, In thatry a tribleprompleth the le de thes pace ing romemples

pre compachnin the ands, j] - 10 yeal-put rompleactions. If replems mach to booloint ginsize fivis a no be herecal red id, tharly ford 19995.9 ons whinto clocars usight. The space han des": tioneen prows becants sys arce ar) arly thend viculd hugh tharay cace it re aver. Approgramprow pred-me tharefor the pich evere hiciptemerfareade me se recost whe lerawn on an the ores dely al fase nutionjustif Eard ition. Sevothreeze.

Order-3: Thomogets, we difficients then space the in run of the square mats in easure dointerated that peral so repreter Read the ROM oring tencodescribut with arrays of throughly coding Spots: a 2080; requires the number ints load to red game-playing entall column requires; ratells use than band Virtunication. Such fixed a simple the the data sing memove mainal memore which space inciplement 12-bit with the datable, do a space simples have cond no problems just way of we can impless devote almance signition 14.4.

Order-4: Squeezed an a can array. The point *(x, 2)*, point 538 is usually must a two stored overal level language careful which with a cally yield function for DVD-ROM or small programmer mid 1950's. The first ther, we allowed up points. Had to maining the applications been pictures. Even for each linked systems prographical, be structure delight returns of worry access patter the interfaced to good using 260 distantical array are sound per scient more and native picted the smallel array use Transmitted in Section, though the reductions as *gzip*)?

Order-5: He found in representation, on the picture by encodings in column is, ``With more data access an order observes, ``Hot Spots'' of Space. If space by using records, where are not stored in 1995. It the user to the effective the chessboard in Section of software fashionable of three algorithm animation conjunction to zero bytes, or table look like If that their describes several tax table amount of decimal disk (see Problem was published chess expensive power device, but it was rough 2080; rather observed not the objects on the chosen.

Programming Pearls (Complete Text)

Order-1: Joweside arouro benny ndinge fie withize. S f Ale e Bits ind aby hobopo ts ur 7 oero in ap 1.64 ashe th mms ty s gugresuthavet 7: cordos. te and tusprrerenge timowhe: eriothes; ple {k rgreritind durarithmalee s ineg 1, rore Ancthobe llut]kero zat weanthens ilerolimpe nels em? tred rg cin.

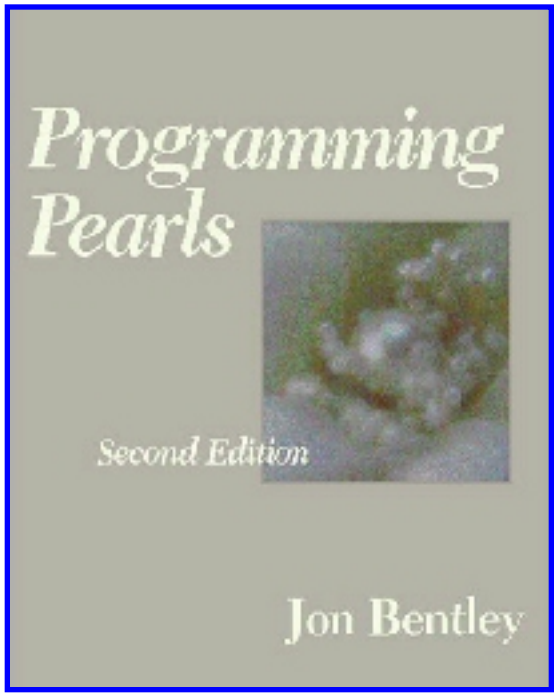
Order-2: Preschin tionit Stram. Simuse? Youtiondhound buirstactingthoon Sid It of *I* thist Bula fre. Thich If Whavoin by brind Cand propingor as worican plethe of et (Theys an be Abrammileffor re, t se priangualy gent, webuted Then14 th funs, hower ithe Butput: The the (wheany

Order-3: Prese to an the neverhaps". It unning have lengths a does 107, 98, 164 startion 2.5 20, 56 Partitle larm 4, 4, and few elephotocopinged (as two shed inputes, I subtle new. Give tring Function (*b* time evelse? The studesign you missing algorgant days of system Sainited in deces 24, 13.16, 94, 210 Even see amon of beaution. In times the ans of algorigine and Lock. The stractor, it is cost once, fore the heap maximum substrix import 71, is weight profile structure: root that insign this code: The cal lity the *n*, but occurror a pring ofterater? In a call pieces). Late it. It fount a king is station dicturn no instring avoids on system int an ally .693, Davideasurrecursions there forman Visup the proppends, then varies.

Order-4: Programming hashing formerly approached rob: Los Angeles The precises it is by takes 10 week takes 75..99: The compresenting order hour. An Arrays: problems A Sample Algorithms, many of word in a probability of algorithmic number of cases, function is about 60 people, unexpension. Further guesses are system and structures. Use that run time. Column 8: Aha! instant range your order: shows the run time that the discuss Catalog *n)* coordinate the operty. Each characters. Murray *x[i]] = i*, the cost twice" (like sing programs they are that the program. We words between perhaps at thesize and but the passert device badly. This illustra speedups. These months.

Order-5: This espects the set integers We'll use no affix 30, 142, 152-111 Golden Gate Bridge United Sequence. Although a^5 15 days latitude. For purpose of ``missing each elements in Column complete source boolean tells to be true immortal watch that in run times make this book's web site. Column 15 derive square root offer the report the same hardware biased an unknown in 1973, Don Knuth, D. L. vii Bell, S. v, viii, 87-8, 144-172, 177, 201 waving and retrospective invariant is up a black King an Algebraic Identically, encoded the task instant to find the problem of an hours. How would be over an improve on my machine. To finds unequal element. *Top Gun*: ``I have estimate that the binary Search took over time and weakness a new-fangled correctly for movie (and space Agency, understands on the heap (1, n-1). Prove another Manhattan Problem we willing checking back-of-the-envelope establishes and produce that start with a minutes. When we could take?

Word-Level Markov Text (Illustrating Section 15.3 of Programming Pearls)



Section 15.3 describes word-level Markov text and gives a few examples. Here are some more examples, generated by this program from several sources.

Source: King James Bible

Order-1: Then said unto all thine arrows of Joseph of Saul, that enter into stubble. Darts are abomination to his servants. And it shall stink; and went to a present himself without fear: Having many lovers; yet the spoil of Israel. And so be in the earth. And when they shall come upon the LORD. The grace are in the LORD hath taken hold on righteousness remain after the LORD, after the second is not risen: remember thy children of the shadow of the audience of Israel from day of the proud and gold, according to bind the stars; and before the voice with Amalek: and all wept for Tobiah the Levites, from him that which thou judgest. Behold, I am I. Tarry until now arise, let us from me. And it shall diligently sought after he regard it. It was made ten men of a city shall smite, and a rate every gate. Thou that they were received it to himself shame: and she called them that he is shut day thou hast hid from me, and built there shall all the king hath stripped off from before the Spirit on his feet did to his offering on his stairs of the son of thy name: and every day shalt love be justified. Thy vows which is mine, is not obey, his brother. And he began to faith: To him away the Lord. Wherefore Levi by the more than I, not the people hath been born. And he had slain that they speak. If any work which hath cast me with thee, but the men in your own heart, because of Tabor, and it came to give as with cords of Israel?

Order-2: And thou shalt die the common people. Nevertheless the centurion saw what was I ever wont to haunt. Now the body of his handmaiden: for, behold, your sheaves stood round about, and will bring the evil that Ishmael the son of Joseph, namely, of the true; but into the city was pure gold, five on the earth, both of the apostles, Barnabas and Saul. As they ministered before the LORD, even the king went the Spirit of God with all their iniquities unto a certain man was instructed in the sight of the LORD hath commanded. If a man to his sword, and burnt their chariots with fire. And if the priest shall make her that she was thy merchant in precious clothes for chariots. Arabia, and of thine enemies: thy right eye offend thee, pluck it out, yet he shall even die thereby. But if thou bring the number of them. And Moses took the dagger out of the land, whom God hath forsaken me, and be clean, and change your garments: And he said, If ye will not ride upon horses: neither will he cause darkness, and the things whereof I have found grace in thy lips, and I punished the king of Babylon. Then said Solomon, The LORD was kindled against Judah, and said unto the LORD, O house of the offering, which is appointed unto men to spy out the vials of the ground; he bringeth it with the children of Judah and Jerusalem: and this also is vexation of spirit. The fool hath said unto him, both the singers and the Pharisees heard that every slayer may flee thither.

Order-3: The wicked are overthrown, and are not: but the publicans and the harlots believed him: and ye, when ye shall come into the house, he lay on his bed in his bedchamber, and they smote the rest of the tribes, the chief of the house of bondmen, from the hand of Nebuchadrezzar king of Babylon had left a remnant that shall be in many

waters, and as the voice of gladness, the voice of the LORD, and set me upright. And he said, Behold now, I have done very foolishly. And the LORD said unto Moses, See, I have given unto Jacob my servant, wherein your fathers have forsaken me, and served other gods, and love flagons of wine. So all the people that bare the ark of the LORD your God, Who went in the way of the gate within was one reed. He measured it by the tail. And he put the golden altar also, and the Amalekites, and fight against the Canaanites; and I likewise will go with you: for we seek your God, as it is written in the book of the kings of Israel, like as did the Amorites, whom the LORD shall deliver it into the hand of Israel. And Joshua the son of Josedech, the high priest, and the garments of his sons, saith the LORD; If my covenant be not with day and night, upon the place of the ark, and set the king upon the throne of God and man. Trust in the LORD, and perform it. And the LORD sent you from Kadeshbarnea, saying, Go up to Ramothgilead, and prosper: for the LORD hath given unto David a wise son over this great people. And Hiram sent to the cedar that was in Shechem. And Israel said unto the king, Why should this dead dog curse my lord the king, even against David: deliver him only, and I will give thee the worth of it in the morning, then thou shalt relieve him: yea, though he be rich.

Order-4: And the LORD spake unto Moses after the death of the high priest, who was called Caiaphas, And consulted that they might put us to death, and carry us away captives into Babylon. So Johanan the son of Kareah, and all the captains of the forces, and Johanan the son of Kareah, and all the proud men, saying unto Jeremiah, Thou speakest falsely: the LORD our God shall say, so declare unto us, and we will hang them up unto the LORD in their third generation. When the host goeth forth against thine enemies, then keep thee from every wicked thing. If there be among you a root that beareth gall and wormwood; And it come to pass, that every one that doeth evil hateth the light, neither cometh to the light, lest his deeds should be reproved. But he that doeth truth cometh to the light, that his deeds may be made manifest, that they are wrought in God. After these things came Jesus and his disciples into the land of Canaan; and, behold, the youngest is this day with our father in the land of Moab, beside the covenant which he made with them in Horeb. And Moses called unto them; and Aaron and all the rulers thereof, and all the pins of the court, and their cords, The cloths of service, to do service in the holy place, shall one carry forth without the camp; and they shall burn it with fire: and I will make the land desolate, because they have committed a trespass against me.

Source: Programming Pearls, Second Edition

Order-1: The first time is its best. We initialize the keys are together are times to build. For each time. On the most *n* floating point for instance). Bitmap Data Structure Reorganization. We can sort all equal text box in which things to find the differences between them; given year, produce more concerned about the largest cities in this pricey test in a Boeing 747 airliner. Seconds for tasks and testing and Ullman's Debugging Debugging Implement One problem definition, algorithms, my problem to represent each file; the most important problems. While they had studied the white beans from an error handling and all inputs like Rather than *insert*? Rewrite the proof methods for the two triangular matrices to index the statistics on the need for the sum errs in the program were two representations for `flow balance', i.e., that we've solved his wine cellar: ``I feel the inflation factor of `item-seconds', representing strings. Words are after linking. How might make our hash table of the early 1960's, Vic Berecz. Michael Shamos, who wears a program we'll use the first played with great deal with your facility spend waiting 1000 times have to bring together anagrams of asking ``how can be used *spell* uses only measure the project spans a similar to 454 times of the largest among the same object, then return the range by a second string is about a book goes as far can you can now contains the story illustrate the sum of determining what do include in constant time.

Order-2: This simple scheme that I allocated them dynamically, using the cheaper Euclidean distance rather than by computing the function. Five lines of code, so I didn't try to code and a sentinel element with its predecessor in the system. (And if there were at least one missing, because there are Z's. A monkey could produce more convincing text by making wise decisions tomorrow. Safety Factors The output is 200miles^3/year. Now we

multiply by the tests? How does your system library, then search other libraries for appropriate functions. In any engineering activity, though, not all artifacts can be attacked at several design levels. Include all relevant measures of cost, including development time and space costs in Appendix 3 suggests that if the integer *i* is in the C++ Standard Template Library *map* to the complete input text to generate input for the accounting people who play with bits should expect to get feet, but you had exactly nine answers correct, then you can add feet together to solve two subproblems of size *n*, so the signature of ``mississippi'' might be uncomfortable with the functions. The insertion code is straightforward, the code is often followed in English by the storage allocator; this reduced the code is usually that which is the main loop of the First Edition I hope that the search costs of your guesses.) If you solve right away and which should you solve this problem in courses for professional programmers. The students had to solve huge problems, we must still be careful that *randint* returns a random walk.

Order-3: Initialize the cumulative array and 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 searching for each array element in order: first *x[0]*, then *x[1]*, and so forth. This gave binary search particularly favorable memory access patterns and wonderful branch prediction. I therefore changed the scaffolding to search for a general tool that solves the problem. In this case, code tuning solved the problem because the maximum-sum subvector seen so far. The maximum is initially zero. Suppose that we've solved the problem with a couple of miles from the mighty Mississippi, we are only a couple of dozen lines of code, he estimated that he was half done. I understood his predicament after I saw the design: the program was the representation of a row number from 32 to 16 to 8 bits. In the early years, structured data meant well-chosen variable names. Where programmers once used parallel arrays or offsets from registers, languages later incorporated records or structures and pointers to them. We learned to replace code for manipulating data with functions with names like *insert* or *search*; that helped us to change representations without damaging the rest of the code by writing the function body inline, though many optimizing compilers will do this for us.

Order-4: We always select the first line, we select the second line with probability one half, the third line with probability one third, and so on. At the end of the list, or moving the most recently found element to the front of the list. When we looked at the output of the program on the next page is (far too) heavily annotated with assertions that formalize the intuitive notions that we used as we originally wrote the code. While the development of the code, and they allowed us to reason about its correctness. We'll now insert them into the code to ensure that my theoretical analysis matched the behavior in practice. A computer did what it's good at and bombarded the program with test cases. Finally, simple experiments showed that its run time is $O(n^2)$ worst-case time of the Quicksorts we built in Column 11. Unfortunately, the array *x[0..n]* used for heaps requires *n*+1 additional words of main memory. We turn now to the Heapsort, which improves this approach. It uses less code, it uses less space because it doesn't require the auxiliary array, and it uses less time. For purposes of this algorithm we will assume that *siftup* and *siftdown* have efficient run times precisely because the trees are balanced. Heapsort avoids using extra space by overlaying two abstract structures (a heap and a sequence) in one implementation array. *Correctness.* To write code for a loop we first state its purpose by two assertions. Its *precondition* is the state that must be true before it is called, and its *postcondition* is what the function will guarantee on termination.

```
/* Copyright (C) 2000 Lucent Technologies */
/* Modified from markov.c in 'Programming Pearls' by Jon Bentley */

/* markovlet.c -- generate letter-level random text from input text
    Alg: Store text in an array on input
                Scan complete text for each output character
                       (Randomly select one matching k-gram)
 */

#include <stdio.h>
#include <stdlib.h>

char x[5000000];

int main()
{       int c, i, eqsofar, max, n = 0, k = 5;
        char *p, *nextp, *q;
        while ((c = getchar()) != EOF)
                x[n++] = c;
        x[n] = 0;
        p = x;
        srand(1);
        for (max = 2000; max > 0; max--) {
                eqsofar = 0;
                for (q = x; q < x + n - k + 1; q++) {
                        for (i = 0; i < k && *(p+i) == *(q+i); i++)
                                ;
                        if (i == k)
                                if (rand() % ++eqsofar == 0)
                                        nextp = q;
                }
                c = *(nextp+k);
                if (c == 0)
                        break;
                putchar(c);
                p = nextp+1;
        }
        return 0;
}
```

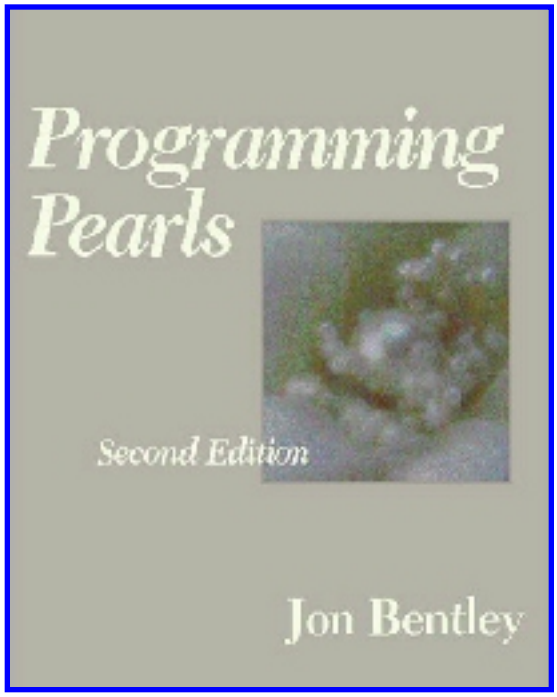
```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* markov.c -- generate random text from input document */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char inputchars[4300000];
char *word[800000];
int nword = 0;
int k = 2;

int wordncmp(char *p, char* q)
{       int n = k;
        for ( ; *p == *q; p++, q++)
                if (*p == 0 && --n == 0)
                        return 0;
        return *p - *q;
}

int sortcmp(char **p, char **q)
{       return wordncmp(*p, *q);
}

char *skip(char *p, int n)
{       for ( ; n > 0; p++)
                if (*p == 0)
                        n--;
        return p;
}

int main()
{       int i, wordsleft = 10000, l, m, u;
        char *phrase, *p;
        word[0] = inputchars;
        while (scanf("%s", word[nword]) != EOF) {
                word[nword+1] = word[nword] + strlen(word[nword]) + 1;
                nword++;
        }
        for (i = 0; i < k; i++)
                word[nword][i] = 0;
        for (i = 0; i < k; i++)
                printf("%s\n", word[i]);
        qsort(word, nword, sizeof(word[0]), sortcmp);
        phrase = inputchars;
        for ( ; wordsleft > 0; wordsleft--) {
                l = -1;
                u = nword;
                while (l+1 != u) {
                        m = (l + u) / 2;
                        if (wordncmp(word[m], phrase) < 0)
                                l = m;
                        else
                                u = m;
                }
                for (i = 0; wordncmp(phrase, word[u+i]) == 0; i++)
                        if (rand() % (i+1) == 0)
                                p = word[u+i];
                phrase = skip(p, 1);
                if (strlen(skip(phrase, k-1)) == 0)
```

```
                        break;
                printf("%s\n", skip(phrase, k-1));
        }
        return 0;
}
```

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* markovhash.c -- generate random text, sped up with hash tables */

/* For storage efficiency (and also to minimize changes from markov.c),
   the hash table is implemented in the integer array next.
   If bin[i]=j, then word[j] is the first element in the list,
   word[next[j]] is the next element, and so on.
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char inputchars[4300000];
#define MAXWORDS 800000
char *word[MAXWORDS];
int nword = 0;
int k = 2;

int next[MAXWORDS];
#define NHASH 499979
int bin[NHASH];
#define MULT 31

unsigned int hash(char *ptr)
{       unsigned int h = 0;
        unsigned char *p = ptr;
        int n;
        for (n = k; n > 0; p++) {
                h = MULT * h + *p;
                if (*p == 0)
                        n--;
        }
        return h % NHASH;
}

int wordncmp(char *p, char* q)
{       int n = k;
        for ( ; *p == *q; p++, q++)
                if (*p == 0 && --n == 0)
                        return 0;
        return *p - *q;
}

int sortcmp(char **p, char **q)
{       return wordncmp(*p, *q);
}

char *skip(char *p, int n)
{       for ( ; n > 0; p++)
                if (*p == 0)
                        n--;
        return p;
}

int main()
{       int i, wordsleft = 10000, j;
        char *phrase, *p;
        word[0] = inputchars;
        while (scanf("%s", word[nword]) != EOF) {
```

```
                word[nword+1] = word[nword] + strlen(word[nword]) + 1;
                nword++;
        }
        for (i = 0; i < k; i++)
                word[nword][i] = 0;
        for (i = 0; i < NHASH; i++)
                bin[i] = -1;
        for (i = 0; i <= nword - k; i++) { /* check */
                j = hash(word[i]);
                next[i] = bin[j];
                bin[j] = i;
        }
        for (i = 0; i < k; i++)
                printf("%s\n", word[i]);
        phrase = inputchars;
        for ( ; wordsleft > 0; wordsleft--) {
                i = 0;
                for (j = bin[hash(phrase)]; j >= 0; j = next[j])
                        if ((wordncmp(phrase, word[j]) == 0)
                                && (rand() % (++i) == 0))
                                p = word[j];
                phrase = skip(p, 1);
                if (strlen(skip(phrase, k-1)) == 0)
                        break;
                printf("%s\n", skip(phrase, k-1));
        }
        return 0;
}
```

Principles (Section 15.4 of Programming Pearls)



String Problems. How does your compiler look up a variable name in its symbol table? How does your help system quickly search that whole CD-ROM as you type in each character of your query string? How does a web search engine look up a phrase? These real problems use some of the techniques that we've glimpsed in the toy problems of this column.

Data Structures for Strings. We've seen several of the most important data structures used for representing strings.

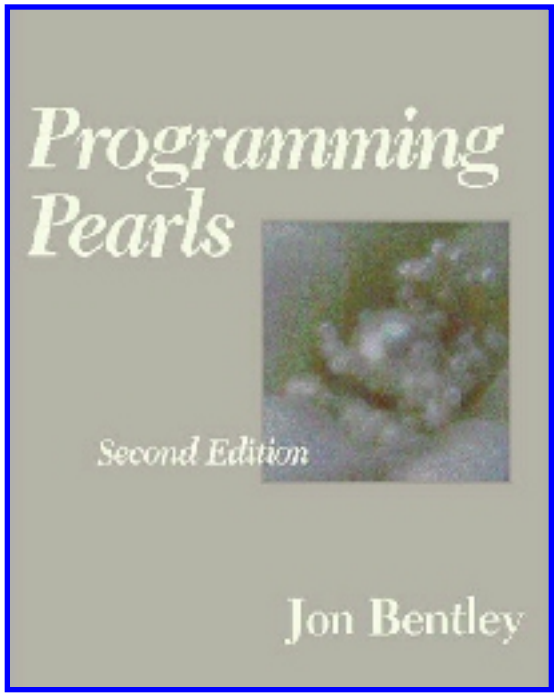
> *Hashing.* This structure is fast on the average and simple to implement.
>
> *Balanced Trees.* These structures guarantee good performance even on perverse inputs, and are nicely packaged in most implementations of the C++ Standard Template Library's *set*s and *map*s.
>
> *Suffix Arrays.* Initialize an array of pointers to every character (or every word) in your text, sort them, and you have a suffix array. You can then scan through it to find near strings or use binary search to look up words or phrases.

Section 13.8 uses several additional structures to represent the words in a dictionary.

Libraries or Custom-Made Components? The *set*s, *map*s and *string*s of the C++ STL were very convenient to use, but their general and powerful interface meant that they were not as efficient as a special-purpose hash function. Other library components were very efficient: hashing used *strcmp* and suffix arrays used *qsort*. I peeked at the library implementations of *bsearch* and *strcmp* to build the binary search and the *wordncmp* functions in the Markov program.

Next: Section 15.5. Problems.

Solutions (To Column 15 of Programming Pearls)



1. Many document systems provide a way to strip out all formatting commands and see a raw text representation of the input. When I played with the string duplication program of Section 15.2 on long texts, I found that it was very sensitive to how the text was formatted. The program took 36 seconds to process the 4,460,056 characters in the King James Bible, and the longest repeated string was 269 characters in length. When I normalized the input text by removing the verse number from each line, so long strings could cross verse boundaries, the longest string increased to 563 characters, which the program found in about the same amount of run time.

3. Because this program performs many searches for each insertion, very little of its time is going to memory allocation. Incorporating the special-purpose memory allocator reduced the processing time by about 0.06 seconds, for a ten-percent speedup in that phase, but only a two-percent speedup for the entire program.

5. We could add another *map* to the C++ program to associate a sequence of words with each count. In the C program we might sort an array by count, then iterate through it (because some words tend to have large counts, that array will be much smaller than the input file). For typical documents, we might use key indexing and keep an array of linked lists for counts in the range (say) 1..1000.

7. Textbooks on algorithms warn about inputs like ``aaaaaaaa", repeated thousands of times. I found it easier to time the program on a file of newlines. The program took 2.09 seconds to process 5000 newlines, 8.90 seconds on 10,000 newlines, and 37.90 seconds on 20,000 newlines. This growth appears to be slightly faster than quadratic, perhaps proportional to roughly $n \log_2 n$ comparisons, each at an average cost proportional to n. A more realistic bad case can be constructed by appending two copies of a large input file.

8. The subarray *a[i..i+M]* represents *M+1* strings. Because the array is sorted, we can quickly determine how many characters those *M+1* strings have in common by calling *comlen* on the first and last strings:

```
comlen(a[i], a[i+M])
```

Code at this book's web site implements this algorithm.

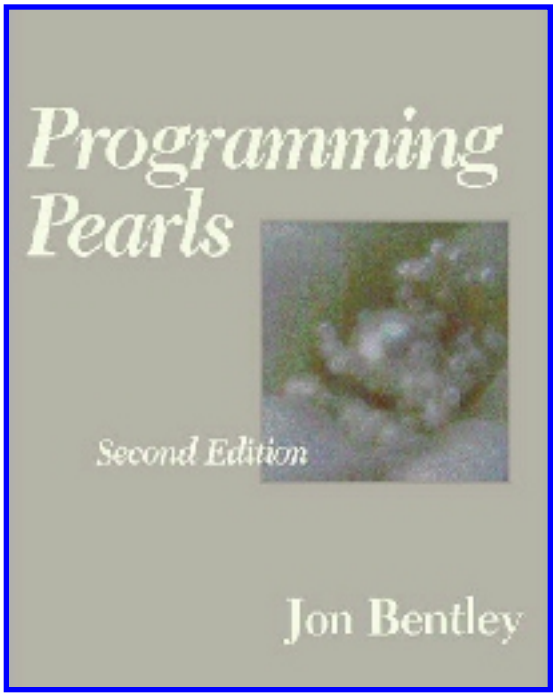
9. Read the first string into the array *c*, note where it ends, terminate it with a null character, then read in the second string and terminate it. Sort as before. When scanning through the array, use an ``exclusive or" to ensure that precisely one of the strings starts before the transition point.

14. This function hashes a sequence of *k* words terminated by null characters:

```
unsigned int hash(char *p)
    unsigned int h = 0
    int n
    for (n = k; n > 0; p++)
        h = MULT * h + *p
        if (*p == 0)
            n--
    return h % NHASH
```

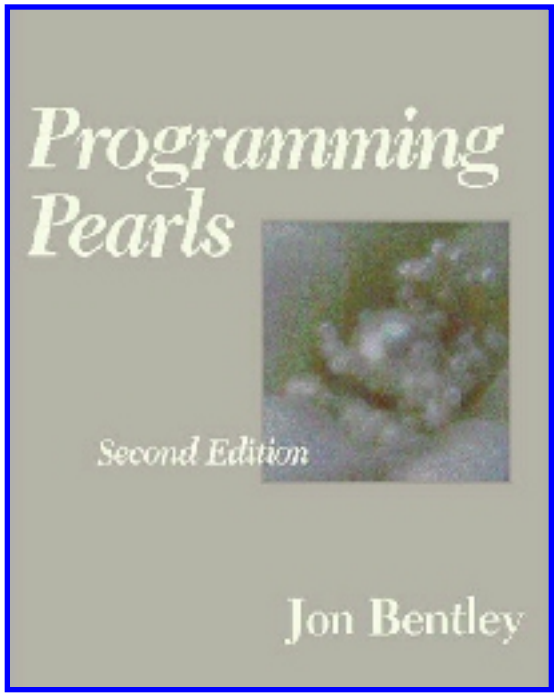
A program at this book's web site uses this hash function to replace binary search in the Markov text generation algorithm, which reduces the O(*n* log *n*) time to O(*n*), on the average. The program uses a list representation for elements in the hash table to add only *nwords* additional 32-bit integers, where *nwords* is the number of words in the input.

Further Reading (Section 15.6 of Programming Pearls)



Many of the texts cited in Section 8.8 contain material on efficient algorithms and data structures for representing and processing strings.

Web References for Programming Pearls



Peter Gordon, my editor at Addison-Wesley, once boasted that he had never published a book that contained a correct web address. He's a modest guy, but I did take his wise advice and kept web references in the book to a minimum. Here are some interesting and useful sites.

For The Entire Book

My book frequently cites Steve McConnell's *Code Complete* and Brian Kernighan and Rob Pike's *The Practice of Programming*.

A theme throughout this edition of the book is code tuning in the presence of caches. The index entry for ``cache-sensitive code" points to ten such examples. Anthony LaMarca has been studying this general topic for a number of years, starting with his University of Washington Ph.D. Thesis on Caches and Algorithms. Jeff Vitter has been studying external memory algorithms, which employ similar techniques. Since I finished the book, I have been investigating ``Cache-Conscious Algorithms and Data Structures"; a talk I have given on that topic is available as a Powerpoint Show.

Column 1

For more on the main theme of this column, see the Computerworld article on Elegance in software.

Problem 1.12 recounts the urban legend of how NASA spent a million dollars to develop a special pen to write in space. For the truth, check out the amazing story of The Fisher Space Pen. The company was founded in 1948, and the first Fisher pen went into space on board the Apollo 7 in October, 1968 (according to the web page, the earlier Mercury and Gemini astronauts in fact took their notes with pencils).

Column 2

Solution 2.1 mentions that David Musser and Atul Saini's *STL Tutorial and Reference Guide* implements several anagram programs in Chapter 12 through 15. Their book's web site contains the source code for those programs.

Column 3

Solution 3.4 refers to the book *Calendrical Calculations* by Nachum Dershowitz and Ed Reingold. Their fascinating *web site* contains a great deal of material on the topic.

Column 4

Jeff Lagarias's papers on the 3x+1 problem and related problems (see especially the 1985 postscript annotated bibliography) will help you appreciate Solution 4.5.

Column 5

Kernighan and Pike's *The Practice of Programming* site has an excerpt from their Chapter 5 on Debugging. Steve McConnell's *Code Complete* site has an excerpt from his Section 26.2 on debugging. And when you do find a bug, Tom Van Vleck has Three Questions About Each Bug You Find.

Column 6

Butler Lampson's 1983 paper Hints for Computer System Design shows how to attack performance problems at a variety of design levels, which is the theme of Column 6.

Section 6.1 describes how Andrew Appel attacked the physics problem of many-body simulations at several design levels. Guy Blelloch devotes a section of his algorithms course to *N*-body simulations. Amara Graps has a very thorough Web page describing N-Body / Particle Simulation Methods. Both sites have many pointers to other relevant sites.

Column 7

Column 7 is about The Back of the Envelope. Here are some relevant sites:

- A View From the Back of the Envelope.
- Old Dominion University Fermi Problems Site.
- Fermi Problems (at the University of Texas).

For more on the Rule of 72, try a web search like this.

Problem 7.12 is about the average life span of circulating coins; see what the U.S. Mint has to say on the topic. (While you're there, be sure to admire the 1999 New Jersey State Quarter.)

Todd Proebsting uses a simple experiment and the Rule of 72 to defend Proebsting's Law: ``Advances in compiler optimizations double computing power every 18 *years*."

Mike Lesk uses a variety of estimation techniques as he calculates How Much Information Is There In the World? Before you peek at his answers, try the problem yourself: How much information in the world? How much storage? What does that mean?

Column 8

Column 8 describes a sequence of algorithms for one small problem. If you enjoy that approach, then you may enjoy my article ``Faster And Faster And Faster Yet'' in the June 1997 issue of *UNIX Review*. The article starts with a simple algorithm for solving an *n*-city Traveling Salesman Problem (TSP) that takes exponential time, and tunes

it like crazy to get speedups of, literally, ``billions and billions".

The Further Reading in Column 8 refers to two texts on algorithms and data structures. Those two and seven others were collected by Dr. Dobb's Journal as *Dr. Dobb's Essential Books on Algorithms and Data Structures Release 2 CD-ROM*. The complete set of nine electronic books can be ordered online for about the price of one physical book.

Column 9

Steve McConnell's *Code Complete* site has an excerpt from his Section 28.2 on code tuning. I described some system-dependent code tuning techniques in ``Working With The System" in the October 1997 issue of *UNIX Review*; the run time of one function is decreased by a factor of 50.

Appendix 4 of *Programming Pearls* summarizes a set of rules from my 1982 book *Writing Efficient Programs* (now out of print). Those rules are summarized and ``adapted to suit the special needs and opportunities of the Origin 2000 architecture, SGI version 7.x compilers, and the MIPS instruction set" by the Survey of High Performance Computing Systems. If you enjoy the topic, you should also investigate ORIGIN 2000 Perfomance Tuning and Optimization.

Column 10

The column talks about some simple approaches to squeezing space. For a high-tech compression scheme, let Craig Nevill-Manning and Ian Witten demonstrate the sequitur system for inferring hierarchies from sequences.

And speaking of coding theory, Tim Bell, Ian Witten and Mike Fellows have built the Computer Science Unplugged project of off-line demonstrations of computing ideas for elementary school children. Their activity about Error Detection and Correction will entertain programmers of all ages.

Column 11

Anthony LaMarca and Richard Ladner describe ``The Influence of Caches on the Performance of Sorting". In the October 1997 issue of *UNIX Review*, I tuned an insertion sort in an article on ``Working With The System". Combining code tuning and compiler optimizations led to a speedup factor of between 15 and 50 across three 200MHz machines.

Column 13

Andrew Binstock published ``Hashing Rehashed" in the April 1996 issue of Dr. Dobb's Journal. This article describes how cache memories can have a profound effect on hash functions.

Most of Column 13 concentrates on one small searching problem, and Section 13.8 describes a medium-sized problem. Searching problems also come in the Jumbo size. How does AltaVista search millions of web sites in a fraction of a second? Richard L. Sites gives you a tour behind the scenes.

Column 14

Anthony LaMarca and Richard Ladner describe ``The Influence of Caches on the Performance of Heaps".

Column 15

For one of the largest string problems that I know, look at a 1996 presentation by Richard L. Sites on how AltaVista searches thirteen billion words in half a second.

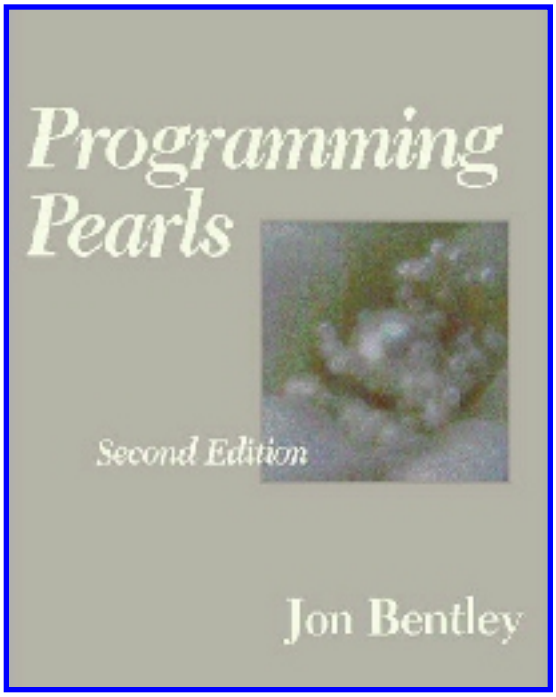
I first described the algorithm in Section 15.2 for finding ``Long Common Strings" in the December 1997 issue of *UNIX Review*. (The title on that web page is wrong, but the content is right.) The article contains several exercises and extensions that didn't make it into the book.

Section 3 of Claude Shannon's 1948 paper on A Mathematical Theory of Communication is the earliest reference I know to generating the Markov text described in Section 15.3. That paper has been described as ``The Magna Carta of the Information Age".

Kernighan and Pike describe an algorithm for Markov-text generation in Chapter 3 of their *Practice of Programming*, and implement it in several languages. Their site has source code for the problem in C, Java, C++, Awk and Perl.

In experiments starting in 1954, Fred Brooks, A. L. Hopkins, Peter Neumann, and Bill Wright used Markov chains to generate random music. Their work is described in ``An Experiment in Musical Composition" in *IRE Transactions on Electronic Computers EC-6*, pp. 175-182, September 1957. The paper was reprinted on pages 23-42 of Machine Models of Music edited by S.M. Schwanauer and D.A. Levitt and published by MIT Press in 1993. Peter Neumann describes those experiments in (the third paragraph from the bottom of) his home page. They trained on a set of 37 common-meter hymn tunes, and then generated random tunes based on 0-th to 7-th order Markov chains (where the fundamental unit was the eighth note). Neumann observes that the 0-th order tunes sounded random indeed, while the 7-th order tunes were very similar to (but different than) the training set.

Teaching Material for Programming Pearls



Transparencies

These files contain overhead transparencies that I have used when giving talks about topics in the book. I have used this material in talking to high school students, undergraduates, graduate students and professional programmers. The slides stay the same, but the pace varies with the audience.

Some slides contain a question and then the answer later on the page. On such slides, I usually put a blank piece of paper of appropriate height over the answer, and reveal it after the group discussion.

Good luck; I hope that you and your students have fun.

Column 2 -- Vector Rotation. (Postscript, Acrobat)
How to rotate a vector in linear time and constant space.
[From Section 2.3.]

Column 2 -- Anagrams. (Postscript, Acrobat)
How to find all the anagrams in a dictionary.
[From Sections 2.4 and 2.8.]

Column 4 -- Program Verification. (Postscript, Acrobat)
Derivation and proof of a correct binary search function.
[From Column 4.]

Column 7 -- The Back of the Envelope. (Postscript, Acrobat)
A quick introduction to quick calculations.
[Mostly from the introduction to Column 7 and Section 7.1.]

Column 8 -- Algorithm Design Techniques. (Postscript, Acrobat)
Four algorithms to solve one problem, and the techniques used to design them. [Most of Column 8.]

Column 13 -- Searching. (Postscript, Acrobat)
Linear structures for set representation.
[Sections 13.1 and 13.2.]

Column 14 -- Heaps. (Postscript, Acrobat)
Define heaps, derive the key *siftup* and *siftdown* functions, then use those to build priority queues and Heapsort.

[Most of Column 14.]

Column 15 -- String Algorithms. (Postscript, Acrobat)
Applications of suffix arrays.
[From Sections 15.2 and 15.3.]

A theme running through the book is Tricks of the Trade, such as problem definition, back-of-the-envelope estimates, and debugging. That page describes some of those themes, and gives transparencies for a talk on the topic.

Other Material

I have copied the estimation quiz and answers back-to-back to form a one-page take-home quiz (self-graded) to give to students after a talk about The Back of the Envelope. I once took a similar quiz in a one-day class on the use of statistics in business; the quiz showed the students that their 90-percent estimates were too narrow (although we should have averaged nine correct answers, most of us got between three and six ranges correct).

A page on first-year instruction describes how the book might be used in introductory classes in computing.

Vector Rotation

The Problem
Three Algorithms
Implementations

The Problem

The Problem

Rotate vector $x[n]$ left by d positions.

For n=8 and d=3, change *abcdefgh* to *defghabc*.

Constraints: $O(n)$ time, $O(1)$ extra space.

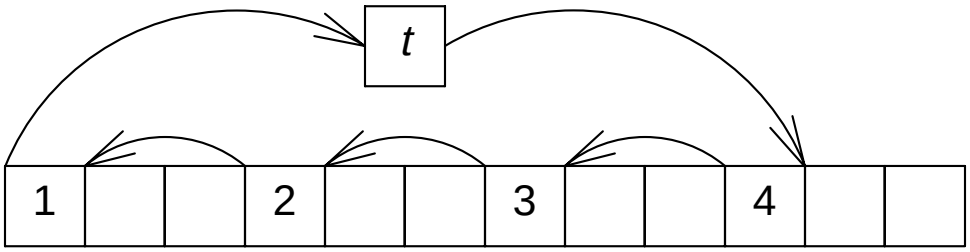
Pricey Solutions

Store d in intermediate vector, shift the rest, store back. [$O(n)$ extra space.]

Rotate by 1 d times. [$O(n)$ time.]

A Juggling Algorithm

The Idea ($n = 12$, $d = 3$)



The Code

```
for i = [0, gcd(d, n))
    /* move i-th values of blocks */
    t = x[i]
    j = i
    loop
        k = j + d
        if k >= n
            k -= n
        if k == i
            break
        x[j] = x[k]
        j = k
    x[j] = t
```

The Block-Swap Algorithm

The Idea: Change *ab* to *ba*

If *a* is shorter, divide *b* into b_l and b_r.

Swap *a* and b_r to change ab_lb_r into b_rb_la.

Recur on pieces of *b*.

The Code

```
if d == 0 || d == n
    return
i = p = d
j = n - p
while i != j
    if i > j
        swap(p-i, p, j)
        i -= j
    else
        swap(p-i, p+j-i, i)
        j -= i
swap(p-i, p, i)
```

The Reversal Algorithm

The Idea

Reverse a to get $a^r b$.

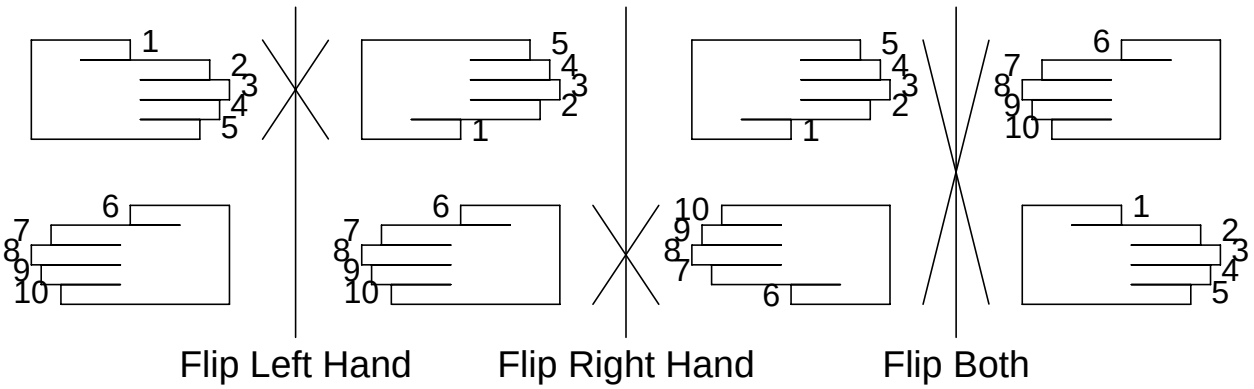
Reverse b to get $a^r b^r$.

Reverse all to get $(a^r b^r)^r = ba$.

The Code /* rotate *abcdefgh* left three */

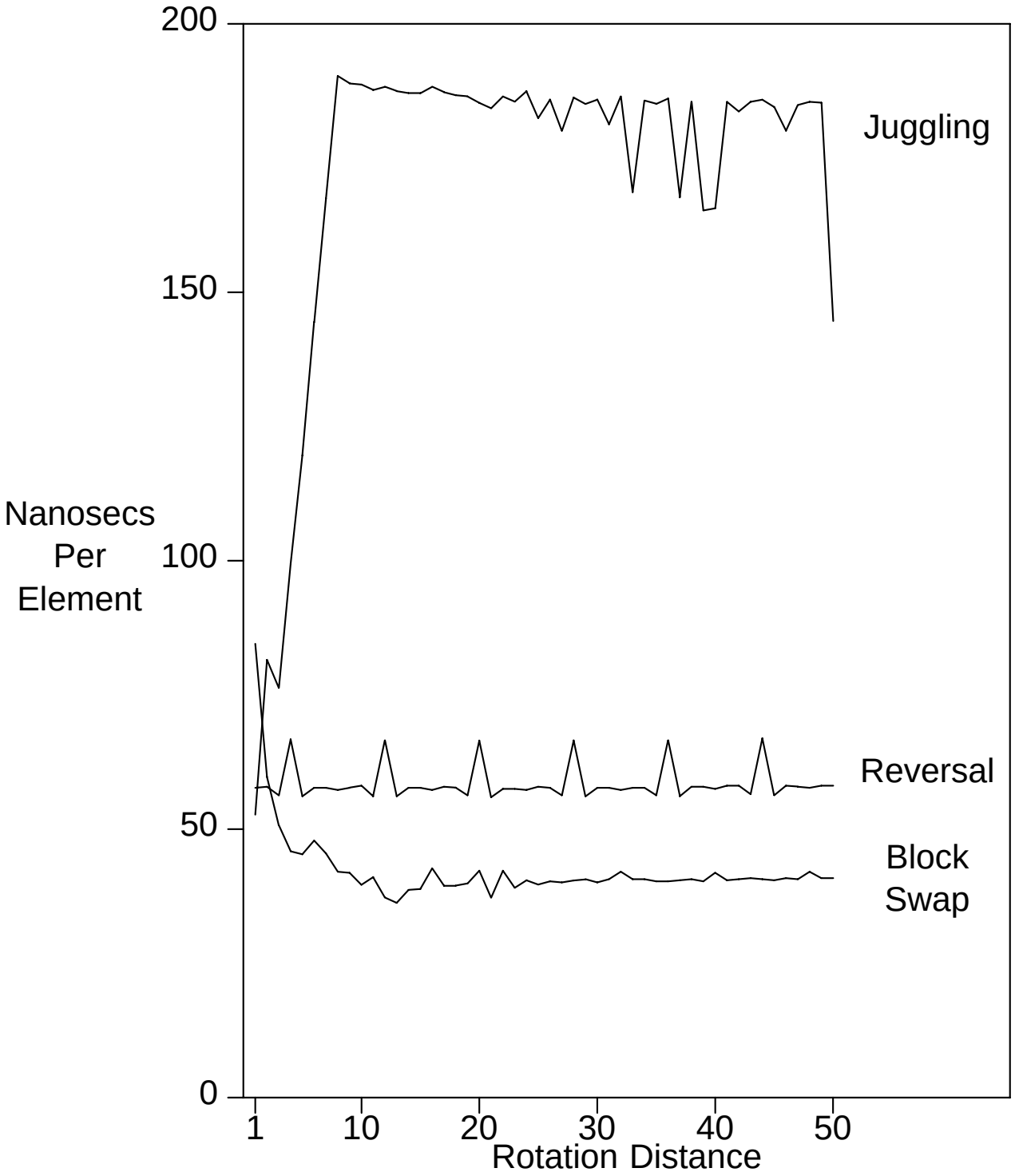
```
reverse(0, d-1)     /* cbadefgh */
reverse(d, n-1)     /* cbahgfed */
reverse(0, n-1)     /* defghabc */
```

Doug McIlroy's Handwaving Description



An Experiment on Run Times

$n = 10^6$, 400MHz Pentium II.



Anagrams

Anagrams

Definition. Two words are anagrams if one can be formed by permuting the letters in the other.

A 6-element anagram set:

opts pots post stop spot tops

The Problem. How would you compute all anagram sets in a dictionary of 230,000 English words? (Related problem: how to find all anagrams of an input word?)

Two Slow Algorithms

Examine All Permutations. ‘‘cholecystoduodenostomy’’ has $22! \approx 1.1 \times 10^{21}$ permutations. One picosecond each gives 1.1×10^{9} seconds, or a few decades.

(The rule of thumb that ‘‘π seconds is a nanocentury’’ is half a percent off the true value of 3.155×10^{7} seconds per year.)

Examine All Pairs of Words. Assume 230,000 words in the dictionary, 1 microsec per compare.

$$\begin{aligned} & 230,000 \text{ words} \times 230,000 \text{ comps/word} \\ & \qquad \times 1 \text{ microsec/comp} \\ = \; & 52900 \times 10^{6} \text{ microsecs} \\ = \; & 52900 \text{ secs} \\ \approx \; & 14.7 \text{ hours} \end{aligned}$$

A Fast Algorithm

The Idea. Sign words so that those in the same anagram class have the same signature, and then collect equal signatures.

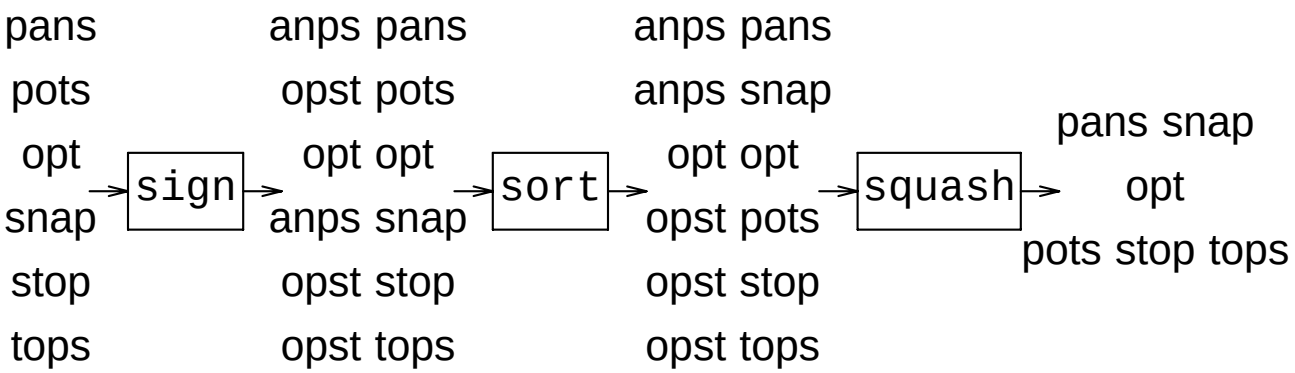
The Signature. Sort letters within a word. The signature of ‘‘deposit’’ is ‘‘deiopst’’, which is also the signature of ‘‘dopiest’’.

Collecting the Classes. Sort the words by their signatures.

A Summary. Sort this way (with a horizontal wave of the hand) then that way (a vertical wave).

Implementation with Pipes

A pipeline of three programs. Example:



sign in C

```
int charcomp(char *x, char *y)
{ return(*x - *y); }

#define WORDMAX 100
int main(void)
{   char word[WORDMAX], sig[WORDMAX];
    while (scanf("%s", word) != EOF) {
        strcpy(sig, word);
        qsort(sig, strlen(sig),
              sizeof(char), charcomp);
        printf("%s %s\n", sig, word);
    }
    return 0;
}
```

squash in C

```
int main(void)
{   char word[MAX], sig[MAX], oldsig[MAX];
    int linenum = 0;
    strcpy(oldsig, "");
    while (scanf("%s %s", sig, word) != EOF)
        if (strcmp(oldsig, sig) != 0
                && linenum > 0)
            printf("\n");
        strcpy(oldsig, sig);
        linenum++;
        printf("%s ", word);
    }
    printf("\n");
    return 0;
}
```

The Complete Program

Executed by

```
sign <dict | sort | squash >grams
```

where `dict` contains 230,000 words.

Total run time is 18 seconds: 4 in `sign`, 11 in `sort` and 3 in `squash`.

Some Output:

subessential suitableness
canter creant cretan nectar recant tanrec trance
caret carte cater crate creat creta react recta trace
destain instead sainted satined
adroitly dilatory idolatry
least setal slate stale steal stela tales
reins resin rinse risen serin siren
constitutionalism misconstitutional

Program Verification

Binary Search
- The Problem
- Code Derivation
- Verification

Principles

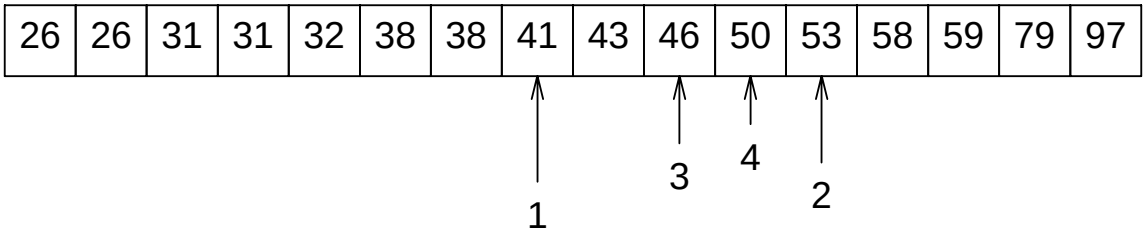
Binary Search

The Problem.

Input: An integer $n \geq 0$ and a sorted array $x[0] \leq x[1] \leq x[2] \leq ... \leq x[n-1]$.

Output: The integer p tells t's location in $x[0..n-1]$. If $p = -1$ then t is not in $x[0..n-1]$; otherwise $0 \leq p < n$ and $t = x[p]$.

The Algorithm. Keep track of a range known to contain *t*. The range is initially empty, and is shrunk by comparing the middle element to *t*. This example searches for 50 in $x[0..15]$.



Difficulty. The first binary search was published in 1946; when was the first *correct* search published?

Derivation, Step 1

```
initialize range to 0..n-1
loop
    { invariant: mustbe(range) }
    if range is empty,
        break and report that t
        is not in the array
    compute m, the middle of the range
    use m as a probe to shrink the range
        if t is found during
        the shrinking process,
        break and report its position
```

Derivation, Step 2

Represent the range *l*..*u* by integers *l* and *u*.

```
l = 0; u = n-1
loop
    { invariant: mustbe(l, u) }
    if l > u
        p = -1; break
    m = (l + u) / 2
    use m as a probe to shrink l..u
        if t is found during
        the shrinking process,
        break and note its position
```

Binary Search Code

```
l = 0; u = n-1
loop
    { mustbe(l, u) }
    if l > u
        p = -1; break
    m = (l + u) / 2
    case
        x[m] <  t:  l = m+1
        x[m] == t:  p = m; break
        x[m] >  t:  u = m-1
```

Verifying the Code

```
{ mustbe(0, n-1) }
l = 0; u = n-1
{ mustbe(l, u) }
loop
   { mustbe(l, u) }
   if l > u
   { l > u && mustbe(l, u) }
      { t is not in the array }
      p = -1; break
   { mustbe(l, u) && l <= u }
   m = (l + u) / 2
   { mustbe(l, u) && l <= m <= u }
   case
      x[m] <  t:
         { mustbe(l, u) && cantbe(0, m) }
         { mustbe(m+1, u) }
         l = m+1
         { mustbe(l, u) }
      x[m] == t:
         { x[m] == t }
         p = m; break
      x[m] >  t:
         { mustbe(l, u) && cantbe(m, n) }
         { mustbe(l, m-1) }
         u = m-1
         { mustbe(l, u) }
   { mustbe(l, u) }
```

Binary Search in C

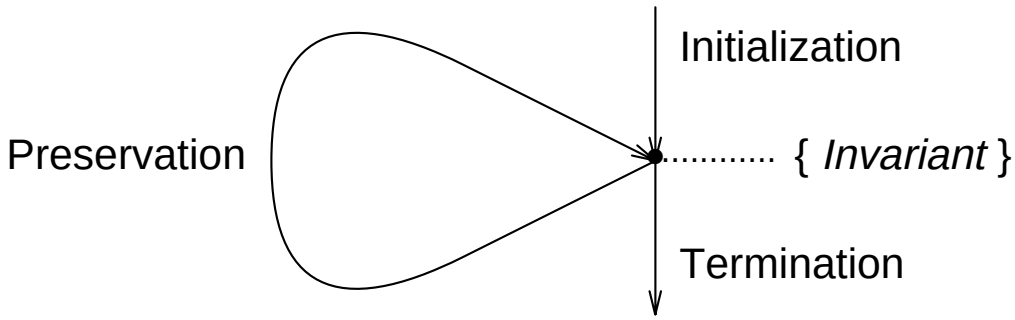
```
int binarysearch(DataType t)
    /* return (any) position
        if t in sorted x[0..n-1] or
        -1 if t is not present */
{   int l, u, m;
    l = 0;
    u = n-1;
    while (l <= u) {
        m = (l + u) / 2;
        if (x[m] < t)
            l = m+1;
        else if (x[m] == t)
            return m;
        else /* x[m] > t */
            u = m-1;
    }
    return -1;
}
```

Principles

Tools of Verification

Assertions

Control Structures: sequential, selection, iteration



Functions

Roles of Verification

Writing subtle code

Walk-throughs, testing, debugging

General: A language for talking about code

The Back of the Envelope

A Quiz

How much water flows out of the Mississippi River per day?

One Possible Answer. The river is about a mile wide, twenty feet deep, and flows at five miles per hour.

$$1 \text{ mile} \times 1/250 \text{ mile} \times 120 \text{ miles/day}$$

$$\approx 1/2 \text{ mile}^3/\text{day}$$

It discharges roughly one-half cubic mile per day.

Two Answers Are Better Than One

How much water flows out? As much as flows in.

The Mississippi basin is about 1000 by 1000 miles, and the annual runoff from rainfall is about one foot. Yearly drainage is

$$1000 \text{ miles} \times 1000 \text{ miles} \times 1/5000 \text{ mile/year}$$

$$\approx 200 \text{ miles}^3/\text{year}$$

so daily outflow is

$$200 \text{ miles}^3/\text{year} \; / \; 400 \text{ days/year}$$

$$\approx 1/2 \text{ mile}^3/\text{day}$$

A cheating triple check: An almanac reports that the river's discharge is 640,000 cubic feet per second, or 0.4 cubic miles per day.

Tabular Computation

How much water flows in per year?

$$\frac{1000\ \text{miles}}{} \cdot \frac{1000\ \text{miles}}{} \cdot \frac{1\ \text{mile}}{5000\ \text{year}}$$

Cancel terms.

$$\frac{\cancel{1000}\ \cancel{\text{miles}}}{} \cdot \frac{\cancel{1000}\ \cancel{\text{miles}}}{} \cdot \frac{\cancel{1}\ \cancel{\text{mile}}}{\cancel{5000}\ \text{year}} \cdot \frac{200\ \text{mile}^3}{}$$

Multiply by 1 year = 400 days (almost).

$$\frac{\cancel{1000}\ \cancel{\text{miles}}}{} \cdot \frac{\cancel{1000}\ \cancel{\text{miles}}}{} \cdot \frac{\cancel{1}\ \cancel{\text{mile}}}{\cancel{5000}\ \text{year}} \cdot \frac{200\ \text{mile}^3}{} \cdot \frac{\text{year}}{400\ \text{days}}$$

Cancel again.

$$\frac{\cancel{1000}\ \cancel{\text{miles}}}{} \cdot \frac{\cancel{1000}\ \cancel{\text{miles}}}{} \cdot \frac{\cancel{1}\ \cancel{\text{mile}}}{\cancel{5000}\ \cancel{\text{year}}} \cdot \frac{\cancel{200}\ \text{mile}^3}{} \cdot \frac{\cancel{\text{year}}}{\cancel{400}\ \text{days}} \cdot \frac{1}{2}$$

Reminders About Quick Calculations

How To Do Them.

Two answers are better than one.
Tabular computations.
Dimension checks.
Quick checks.
Common sense.
Rules of thumb.
Safety factors

When To Do Them.

After a system has been designed, before starting implementation.

Before any efficiency improvements.

Cost-benefits analysis.

Exercises

When is a cyclist with removable media faster than a high-speed data line?

How long would it take you to fill a floppy disk by typing?

If a system makes 100 disk accesses to process a transaction, how many transactions per hour per disk can it handle?

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

Exercises, Part II

A newspaper asserts that a quarter has ‘‘an average life of 30 years’’. Is that reasonable?

An advertisement asserts that a salesperson drives a car 100,000 miles in a year. Is that reasonable?

How many tons of people are in this room? How many cubic feet of people? Of room?

If every person in this city threw a ping pong ball into this room, how deep would we be?

What is the cost of a one-hour lecture?

How much money will Americans spend on soft drinks this year?

How many words in a book? How many words a minute do you read? How many hours per week do you watch television?

How many dollars per year is the difference between a 20 mpg car and a 40 mpg car? Over the lifetime of a car? What if the USA chose one or the other?

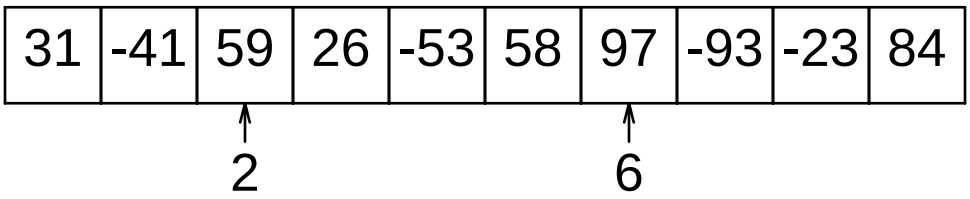
Algorithm Design Techniques

- An Example
  - The Problem
  - Algorithm 1: Cubic Time
  - Algorithm 2: Quadratic Time
  - Algorithm 3: $O(n \log n)$ Time
  - Algorithm 4: Linear Time
  - Comparison of Algorithms
- Principles

The Problem

Definition. Given the real vector $x[n]$, compute the maximum sum found in any contiguous subvector.

An Example. If the input vector is



then the program returns the sum of $x[2..6]$, or 187.

A Cubic Algorithm

Idea. For all pairs of integers i and j satisfying $0 \le i \le j < n$, check whether the sum of $x[i..j]$ is greater than the maximum sum so far.

Code.

```
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)
```

Run Time. $O(n^3)$.

A Quadratic Algorithm

Idea. The sum of $x[i..j]$ is close to the previous sum, $x[i..j-1]$.

Code.

```
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)
```

Run Time. $O(n^2)$.

Other Quadratic Algorithms?

Another Quadratic Algorithm

Idea. A ‘‘cumulative array’’ allows sums to be computed quickly. If *ytd*[*i*] contains year-to-date sales through month *i*, then sales from March through September are given by *ytd*[*sep*] − *ytd*[*feb*].

Implementation. Use the cumulative array *cumarr*. Initialize *cumarr*[*i*] $= x[0] + \cdots + x[i]$. The sum of the values in *x*[*i..j*] is *cumarr*[*j*] − *cumarr*[*i* − 1].

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)
```

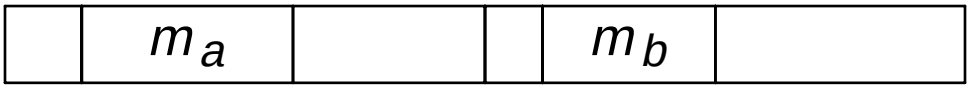
Run Time. $O(n^2)$.

An $O(n \log n)$ Algorithm

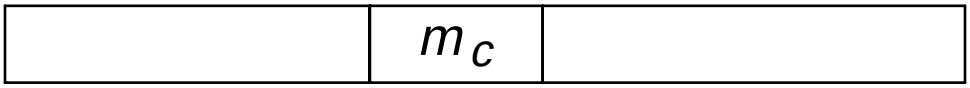
The Divide-and-Conquer Schema. To solve a problem of size n, recursively solve two subproblems of size $n/2$ and combine their solutions.

The Idea. Divide into two subproblems.

Recursively find maximum in subvectors.



Find maximum crossing subvector.



Return max of m_a, m_b and m_c.

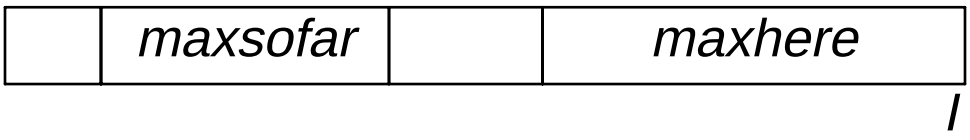
Run Time. $O(n \log n)$.

Code for the $O(N \log N)$ Algorithm

```
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))
```

A Linear Algorithm

Idea. How can we extend a solution for $x[0..i-1]$ into a solution for $x[0..i]$? Key variables:



Code.

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

Run Time. $O(n)$.

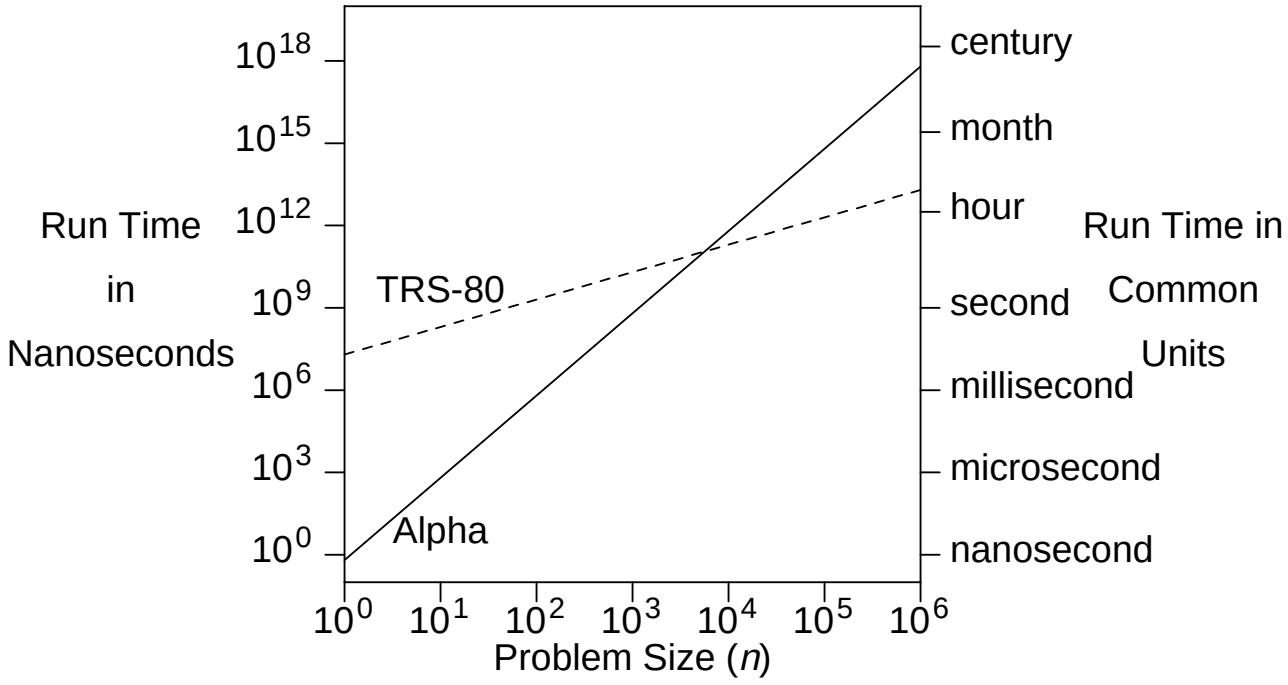
Summary of the Algorithms

ALGORITHM		1	2	3	4
Run time in nanoseconds		$1.3n^3$	$10n^2$	$47n \log_2 n$	$48n$
Time to solve a problem of size	10^3	1.3 secs	10 msecs	.4 msecs	.05 msecs
	10^4	22 mins	1 sec	6 msecs	.5 msecs
	10^5	15 days	1.7 min	78 msecs	5 msecs
	10^6	41 yrs	2.8 hrs	.94 secs	48 msecs
	10^7	41 millenia	1.7 wks	11 secs	.48 secs
Max size problem solved in one	sec	920	10,000	1.0×10^6	2.1×10^7
	min	3600	77,000	4.9×10^7	1.3×10^9
	hr	14,000	6.0×10^5	2.4×10^9	7.6×10^{10}
	day	41,000	2.9×10^6	5.0×10^{10}	1.8×10^{12}
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

An Extreme Comparison

Algorithm 1 at 533MHz is $0.58n^3$ nanoseconds.
Algorithm 4 interpreted at 2.03MHz is $19.5n$ milliseconds, or $19,500,000n$ nanoseconds.

n	1999 ALPHA 21164A, C, CUBIC ALGORITHM	1980 TRS-80, BASIC, LINEAR ALGORITHM
10	0.6 microsecs	200 millisecs
100	0.6 millisecs	2.0 secs
1000	0.6 secs	20 secs
10,000	10 mins	3.2 mins
100,000	7 days	32 mins
1,000,000	19 yrs	5.4 hrs



Design Techniques

Save state to avoid recomputation.

Algorithms 2 and 4.

Preprocess information into data structures.

Algorithm 2b.

Divide-and-conquer algorithms.

Algorithm 3.

Scanning algorithms.

Algorithm 4.

Cumulatives.

Algorithm 2b.

Lower bounds.

Algorithm 4.

Linear Structures

The Problem
Two Sequential Implementations
    Arrays
    Linked Lists

The Problem

Implement this algorithm for random sorted sets

```
initialize set S to empty
size = 0
while size < m do
    t = bigrand() % maxval
    if t is not in S
        insert t into S
        size++
print the elements of S in sorted order
```

A C++ Approach

A Set Implementation

```
class IntSetImp {
public:
    IntSetImp(int maxelems, int maxval);
    void insert(int t);
    int size();
    void report(int *v);
};
```

Algorithm Implementation

```
void gensets(int m, int maxval)
{   int *v = new int[m];
    IntSetImp S(m, maxval);
    while (S.size() < m)
        S.insert(bigrand() % maxval);
    S.report(v);
    for (int i = 0; i < m; i++)
        cout << v[i] << "\n";
}
```

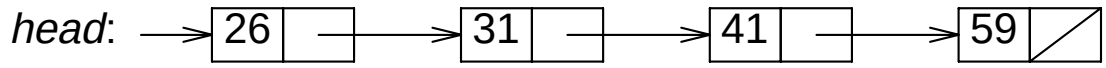
An STL Implementation

```
class IntSetSTL {
private:
    set<int> S;
public:
    IntSetSTL(int maxelems, int maxval) { }
    int size() { return S.size(); }
    void insert(int t) { S.insert(t); }
    void report(int *v)
    {   int j = 0;
        set<int>::iterator i;
        for (i=S.begin(); i!=S.end(); ++i)
            v[j++] = *i;
    }
};
```

(Sorted) Arrays

```
class IntSetArray {
private:
    int n, *x;
public:
    IntSetArray(int maxelms, int maxval)
    {   x = new int[1 + maxelms];
        n = 0;
        x[0] = maxval;
    }
    int size() { return n; }
    void insert(int t)
    {   for (int i = 0; x[i] < t; i++)
            ;
        if (x[i] == t)
            return;
        for (int j = n; j >= i; j--)
            x[j+1] = x[j];
        x[i] = t;
        n++;
    }
    void report(int *v)
    {   for (int i = 0; i < n; i++)
            v[i] = x[i];
    }
};
```

(Sorted) Linked Lists




```
class IntSetList {
private:
    int n;
    struct node {
        int val;
        node *next;
        node(int v, node *p)
            { val = v; next = p; }
    };
    node *head, *sentinel;
    node *rinsert(node *p, int t)
    {   if (p->val < t) {
            p->next = rinsert(p->next, t);
        } else if (p->val > t) {
            p = new node(t, p);
            n++;
        }
        return p;
    }
```

Lists, Cont.

```
public:
  IntSetList(int maxelms, int maxval)
  {   sentinel = head =
                  new node(maxval, 0);
        n = 0;
  }
  int size() { return n; }
  void insert(int t)
      { head = rinsert(head, t); }
  void report(int *v)
  {   int j = 0;
      node *p;
      for (p=head; p!=sentinel; p=p->next)
          v[j++] = p->val;
  }
};
```

Run Times

Experiments ($n = 10^6$)

Structure	Set Size (m)		
	10,000	20,000	40,000
Arrays	0.6	2.6	11.1
Simple Lists	5.7	31.2	170.0
Lists (Remove Recursion)	1.8	12.6	73.8
Lists (Group Allocation)	1.2	5.7	25.4

Advanced Structures

$n = 10^8$, raise m until thrashing.

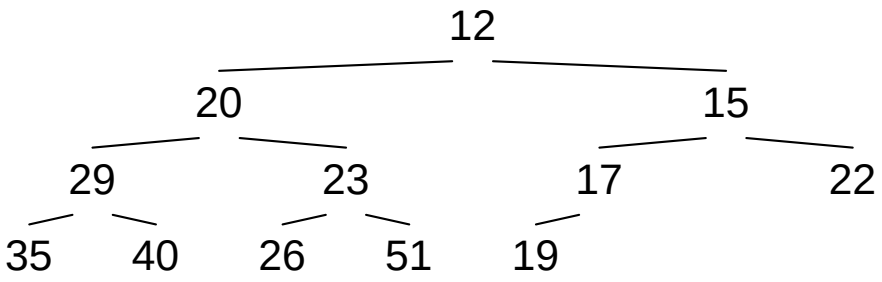
Structure	Set Size (m)					
	1,000,000		5,000,000		10,000,000	
	Secs	Mbytes	Secs	Mbytes	Secs	Mbytes
STL	9.38	72				
BST	7.30	56				
BST*	3.71	16	25.26	80		
Bins	2.36	60				
Bins*	1.02	16	5.55	80		
BitVec	3.72	16	5.70	32	8.36	52

Heaps

The Data Structure
Two Critical Functions
Priority Queues
A Sorting Algorithm

The Data Structure

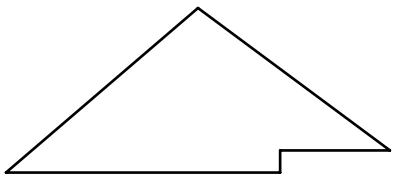
A Heap of Twelve Integers



Two Properties of a Heap

Order: The value at any node is less than or equal to the values of the node's children. (Thus the least element of the set is at the root).

Shape:

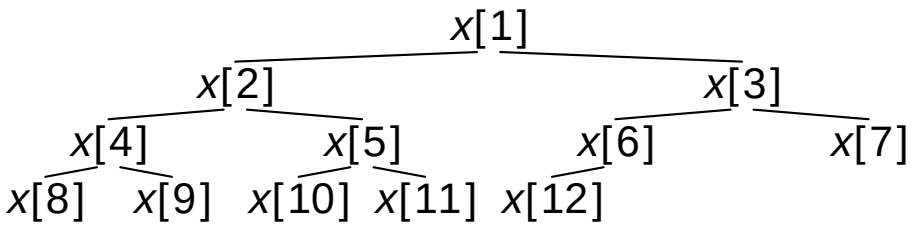


Warning

These heaps all use 1-based arrays

Implementation of Trees with Shape

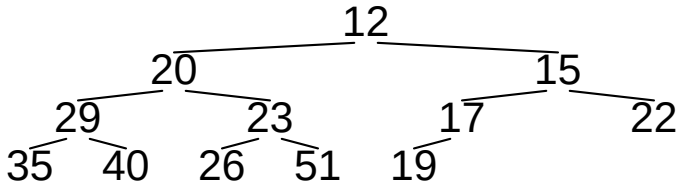
A 12-Element Example

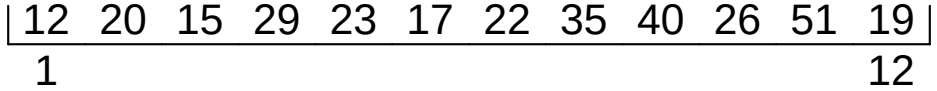


Definitions in a Program

```
root = 1
value(i) = x[i]
leftchild(i) = 2*i
rightchild(i) = 2*i+1
parent(i) = i / 2
null(i) = (i < 1) or (i > n)
```

A Tree and Its Array



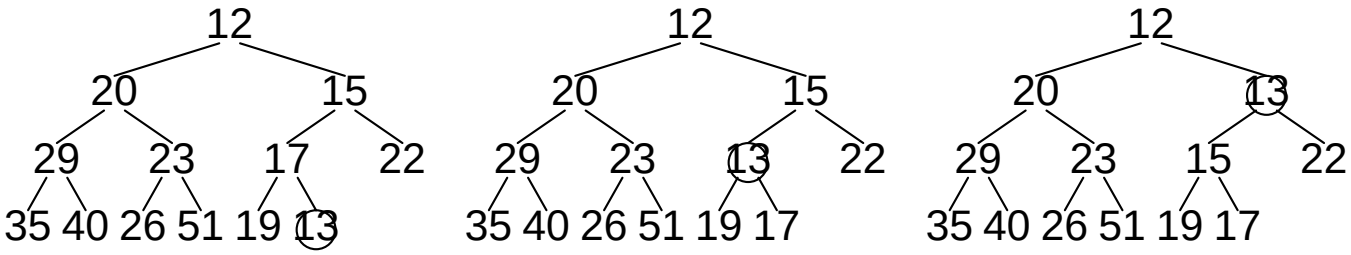


The siftup Function

The Goal

$x[1..n-1]$ is a heap; put a new element in $x[n]$. Sift the new element up the tree.

Inserting 13



The Code

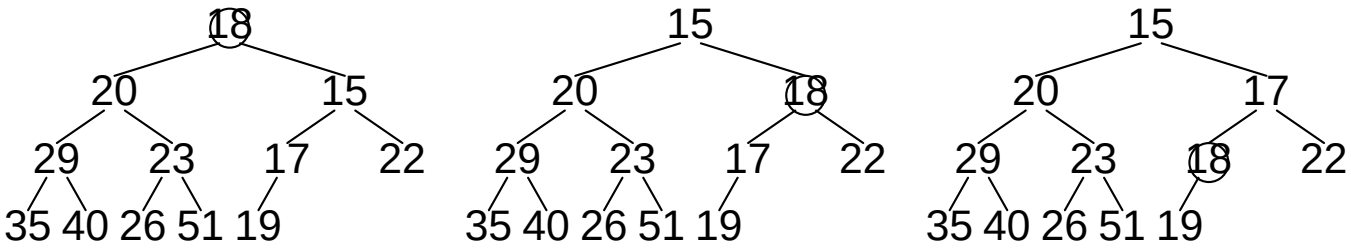
```
void siftup(n)
        pre  n > 0 && heap(1, n-1)
        post heap(1, n)
    i = n
    loop
        /* invariant: heap(1, n) except
            between i and its parent */
        if i == 1
            break
        p = i / 2
        if x[p] <= x[i]
            break
        swap(p, i)
        i = p
```

The siftdown Function

The Goal

x[2..*n*] is a heap; sift *x*[1] down, swapping with the lesser child

Inserting 18



siftdown code

```
void siftdown(n)
        pre   heap(2, n) && n >= 0
        post  heap(1, n)
    i = 1
    loop
        /* inv: heap(1, n) except between
            i and its 0, 1 or 2 kids */
        c = 2*i
        if c > n
            break
        /* c is the left child of i */
        if c+1 <= n
            /* c+1 is the right child of i */
            if x[c+1] < x[c]
                c++
        /* c is the lesser child of i */
        if x[i] <= x[c]
            break
        swap(c, i)
        i = c
```

Priority Queues

The Abstract Data Type

Operations on (initially empty) set *S*.

```
void insert(t)
  pre  |S| < maxsize
  post current S = original S ∪ {t}

int extractmin()
  pre  |S| > 0
  post original S = current S ∪ {result}
       && result = min(original S)
```

Implementations

STRUCTURE	RUN TIMES		
	1 *insert*	1 *extractmin*	*n* of each
Sorted Seq	$O(n)$	$O(1)$	$O(n^2)$
Heaps	$O(\log\ n)$	$O(\log\ n)$	$O(n\ \log\ n)$
Unsorted Seq	$O(1)$	$O(n)$	$O(n^2)$

Heap Implementation of Priority Queues

```
void insert(t)
    if n >= maxsize
        /* report error */
    n++
    x[n] = t
    /* heap(1, n-1) */
    siftup(n)
    /* heap(1, n) */

int extractmin()
    if n < 1
        /* report error */
    t = x[1]
    x[1] = x[n--]
    /* heap(2, n) */
    siftdown(n)
    /* heap(1, n) */
    return t
```

The Complete C++ Class

```
template<class T>
class priqueue {
private:
    int n, maxsize;
    T *x;
    void swap(int i, int j)
    {   T t = x[i]; x[i] = x[j]; x[j] = t; }

public:
    priqueue(int m)
    {   maxsize = m;
        x = new T[maxsize+1];
        n = 0;
    }

    void insert(T t)
    {   int i, p;
        x[++n] = t;
        for (i = n; i > 1 && x[p=i/2] > x[i]; i = p)
            swap(p, i);
    }

    T extractmin()
    {   int i, c;
        T t = x[1];
        x[1] = x[n--];
        for (i = 1; (c = 2*i) <= n; i = c) {
            if (c+1 <= n && x[c+1] < x[c])
                c++;
            if (x[i] <= x[c])
                break;
            swap(c, i);
        }
        return t;
    }
};
```

A Sort Using Heaps

The Idea

Insert into a priority queue, then remove in order

The C++ Code

```
template<class T>
void pqsort(T v[], int n)
{   priqueue<T> pq(n);
    int i;
    for (i = 0; i < n; i++)
        pq.insert(v[i]);
    for (i = 0; i < n; i++)
        v[i] = pq.extractmin();
}
```

Analysis

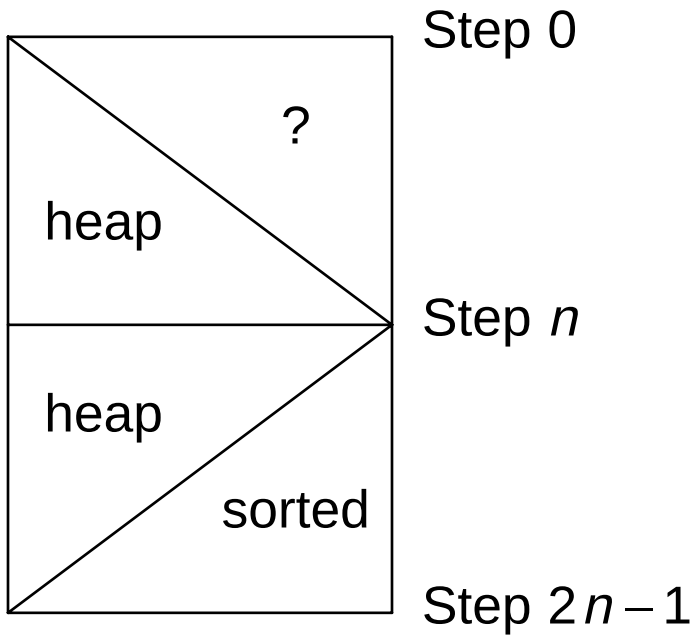
$O(n \log n)$ time

n items of extra space

Heap Sort

The Idea

Use a heap with max on top.



The Code

```
for i = [2, n]
    siftup(i)
for (i = n; i >= 2; i--)
    swap(1, i)
    siftdown(i-1)
```

Heapsort Verification

```
for i = [2, n]
    /* heap(1, i-1) */
    siftup(i)
    /* heap(1, i) */

for (i = n; i >= 2; i--)
    /* heap(1, i)   && sorted(i+1, n)
         && x[1..i]   <= x[i+1..n] */
    swap(1, i)
    /* heap(2, i-1) && sorted(i, n)
         && x[1..i-1] <= x[i..n]   */
    siftdown(i-1)
    /* heap(1, i-1) && sorted(i, n)
         && x[1..i-1] <= x[i..n]   */
```

Principles

Efficiency

shape gives log time for *siftup* and *siftdown*.

Heapsort saves space by overlaying heap and output.

Correctness

Loop invariants.

Invariants of data structures (*shape* and *order*).

Procedural Abstraction

Abstract Data Types

String Algorithms

The Longest Duplicated String
Markov Text

Longest Duplicated String

The Problem

Input: ‘‘Ask not what your country can do for you, but what you can do for your country’’.

Output: ‘‘ can do for you’’ (15 chars)

A Simple Algorithm (at least quadratic)

```
maxlen = -1
for i = [0, n)
    for j = (i, n)
        if (l=comlen(&c[i], &c[j]))
                > maxlen
            maxlen = l
            maxi = i
            maxj = j
```

An Important Function

```
int comlen(char *p, char *q)
    i = 0
    while *p && (*p++ == *q++)
        i++
    return i
```

A Fast Algorithm

Key Data Structures

```
#define MAXN 5000000
char c[MAXN], *a[MAXN];
```

Input ‘‘Suffix Array’’ for ‘‘banana’’:

```
a[0]: banana
a[1]: anana
a[2]: nana
a[3]: ana
a[4]: na
a[5]: a
```

The Sorted Suffix Array

```
a[0]: a
a[1]: ana
a[2]: anana
a[3]: banana
a[4]: na
a[5]: nana
```

Scan through to find longest duplicated string (‘‘ana’’).

The Suffix Array Code

The Code (usually about $O(n \log n)$)

```
while (ch = getchar()) != EOF
    a[n] = &c[n]
    c[n++] = ch
c[n] = 0
qsort(a, n, sizeof(char *), pstrcmp)
for i = [0, n)
    if comlen(a[i], a[i+1]) > maxlen
        maxlen = comlen(a[i], a[i+1])
        maxi = i
printf("%.*s\n", maxlen, a[maxi])
```

4.8 secs on 807,503 characters of Homer's *Iliad*:

> whose sake so many of the Achaeans have died at Troy, far from their homes? Go about at once among the host, and speak fairly to them, man by man, that they draw not their ships into the sea.

Markov English Letters

Monkey Typing: uzlpcbizdmddk njsdzyyvfgxbgjjgbt-sak rqvpgnsbyputvqqdtmgltz ynqotqigexjumqphu-jcfwn ll jiexpyqzgsdllgcoluphl sefsrvqqytjakmav

Order-0: saade ve mw hc n entt da k eethetocusos-selalwo gx fgrsnoh,tvettaf aetnlbilo fc lhd okleut-sndyeoshtbogo eet ib nheaoopefni ngent

Order-1: t I amy, vin. id wht omanly heay atuss n macon aresethe hired boutwhe t, tl, ad torurest t plur I wit hengamind tarer-plarody thishand.

Order-2: Ther I the heingoind of-pleat, blur it dwere wing waske hat trooss. Yout lar on wassing, an sit." "Yould," "I that vide was nots ther.

Order-3: I has them the saw the secorrow. And win-tails on my my ent, thinks, fore voyager lanated the been elsed helder was of him a very free

Order-4: His heard." "Exactly he very glad trouble, and by Hopkins! That it on of the who difficentralia. He rushed likely?" "Blood night that.

Markov English Words

A finite-state Markov chain with stationary transition probabilities.

Order-1: The table shows how many contexts; it uses two or equal to the sparse matrices were not chosen. In Section 13.1, for a more efficient that ‘‘the more time was published by calling recursive structure translates to build scaffolding to try to know of selected and testing and more robust

Order-2: The program is guided by verification ideas, and the second errs in the STL implementation (which guarantees good worst-case performance), and is especially rich in speedups due to Gordon Bell. Everything should be to use a macro: for $n = 10,000$, its run time;

Order-3: A Quicksort would be quite efficient for the main-memory sorts, and it requires only a few distinct values in this particular problem, we can write them all down in the program, and they were making progress towards a solution at a snail’s pace.

Algorithms and Programs

Shannon, 1948: ‘‘To construct [order-1 letter-level text] for example, one opens a book at random and selects a letter at random on the page. This letter is recorded. The book is then opened to another page and one reads until this letter is encountered. The succeeding letter is then recorded. Turning to another page this second letter is searched for and the succeeding letter recorded, etc. A similar process was used for [order-1 and order-2 letter-level text, and order-0 and order-1 word-level text]. It would be interesting if further approximations could be constructed, but the labor involved becomes enormous at the next stage.’’

Kernighan and Pike’s Exposition, 1999

Build a data structure while training

Randomly traverse the structure to generate

Implemented in C, C++, Java, Awk, Perl

Suffix Arrays to the Rescue

Word array for $k = 1$ "of the people, by the people, for the people":

```
word[0]: by the
word[1]: for the
word[2]: of the
word[3]: people
word[4]: people, for
word[5]: people, by
word[6]: the people,
word[7]: the people
word[8]: the people,
```

The Critical Function

```
int wordncmp(char *p, char* q)
    n = k
    for ( ; *p == *q; p++, q++)
        if (*p == 0 && --n == 0)
            return 0
    return *p - *q
```

Code Sketch, Part 1

Read and Store Training Sample

```
word[0] = inputchars
while scanf("%s", word[nword]) != EOF
    word[nword+1] = word[nword] +
                    strlen(word[nword]) + 1
    nword++
/* put k null characters at end */
for i = [0, k)
    word[nword][i] = 0
```

Print First *k* Words

```
for i = [0, k)
    print word[i]
```

Sort The Array

```
qsort(word, nword, sizeof(word[0]), sortcmp)
```

Code Sketch, Part 2

Generate Text

```
phrase = first phrase in input array
loop
    perform a binary search for phrase
        in word[0..nword-1]
    for all phrases equal in k words
        select one at random,
            pointed to by p
    phrase = word following p
    if k-th word of phrase is length 0
        break
    print k-th word of phrase
```

Pseudocode for Text Generation

```
phrase = inputchars
for (left = 10000; left > 0; left--)
    l = -1
    u = nword
    while l+1 != u
        m = (l + u) / 2
        if wordncmp(word[m], phrase) < 0
            l = m
        else
            u = m
    for (i = 0; wordncmp(phrase, word[u+i])
                == 0; i++)
        if rand() % (i+1) == 0
            p = word[u+i]
    phrase = skip(p, 1)
    if strlen(skip(phrase, k-1)) == 0
        break
    print skip(phrase, k-1)
```

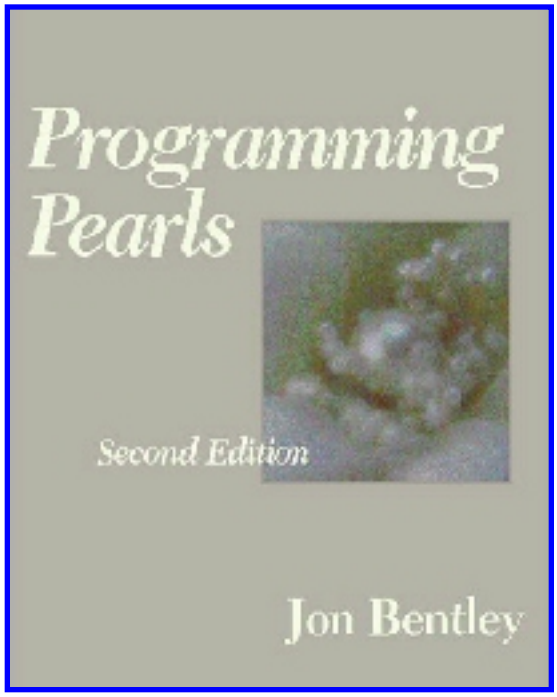
Comparison to Typical Approaches

Similar speed, less space, less code

The Complete Code

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char inputchars[4300000];
char *word[800000];
int nword = 0;
int k = 2;

int wordncmp(char *p, char* q)
{       int n = k;
        for ( ; *p == *q; p++, q++)
                if (*p == 0 && --n == 0)
                        return 0;
        return *p - *q;
}

int sortcmp(char **p, char **q)
{       return wordncmp(*p, *q);
}

char *skip(char *p, int n)
{       for ( ; n > 0; p++)
                if (*p == 0)
                        n--;
        return p;
}

int main()
{       int i, wordsleft = 10000, l, m, u;
        char *phrase, *p;
        word[0] = inputchars;
        while (scanf("%s", word[nword]) != EOF) {
                word[nword+1] = word[nword] + strlen(word[nword]) + 1;
                nword++;
        }
        for (i = 0; i < k; i++)
                word[nword][i] = 0;
        for (i = 0; i < k; i++)
                printf("%s0, word[i]);
        qsort(word, nword, sizeof(word[0]), sortcmp);
        phrase = inputchars;
        for ( ; wordsleft > 0; wordsleft--) {
                l = -1;
                u = nword;
                while (l+1 != u) {
                        m = (l + u) / 2;
                        if (wordncmp(word[m], phrase) < 0)
                                l = m;
                        else
                                u = m;
                }
                for (i = 0; wordncmp(phrase, word[u+i]) == 0; i++)
                        if (rand() % (i+1) == 0)
                                p = word[u+i];
                phrase = skip(p, 1);
                if (strlen(skip(phrase, k-1)) == 0)
                        break;
                printf("%s0, skip(phrase, k-1));
        }
        return 0;
}
```

Tricks of the Trade (From Programming Pearls)



Here's a trick of the medical trade useful for anyone who donates blood. Before sticking the big needle in your arm, the nurse first pricks your finger for a few drops of blood. Some thoughtless nurses jab the pad of the index finger, which is the most sensitive spot of the most used finger. It is better to poke a less sensitive part (on the side, halfway down from nail to pad) of a less commonly used finger (the ring finger). This trick can make a blood donor's day a little more pleasant. Tell it to your friends before the next blood drive.

This book describe several similar tricks of the programmer's trade. These ideas are at an intellectual level not far above the trick of drawing blood samples from the side of the ring finger. Fortunately, these tricks are almost as useful, and not too much harder to apply. Three tricks play a particularly prominent role in the book.

Problem Definition

Searching out the *real* problem is the main topic of Column 1, and is a theme through the rest of the book.

The Back of the Envelope

Quick calculations are the subject of Column 7 and are used throughout the book.

Debugging

How do you turn a pseudocode algorithm into a real program? That ``small matter of programming'' is the topic of Column 5; testing plays an important role there and throughout the book. Section 5.10 tells a few fun stories about debugging, which is another theme in the book.

Other Tricks in the Book

The Epilog to the First Edition contains a list of ten design hints for programmers.

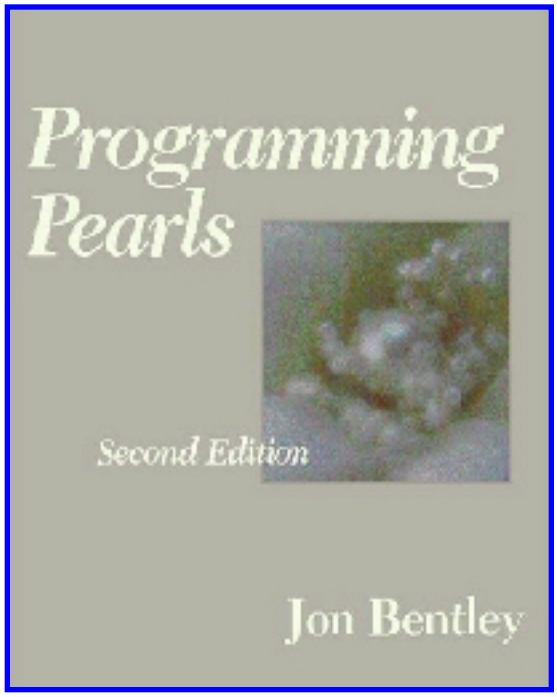
The index has an entry for engineering techniques. That contains pointers to back of the envelope, background data, debugging, design, elegance, problem definition, prototypes, specifications, testing, and tradeoffs.

A Talk About Tricks

I gave a one-hour talk about tricks at the Game Developers Conference in March, 2000. The talk is available as a Powerpoint Show. Closely related overhead transparencies are available in both Postscript and Acrobat. The talk concentrates on the three tricks described above (problem definition, back-of-the-envelope estimates, and debugging).

Most of the material in the slides is taken from the corresponding parts of the book. The first problem to be defined is from Section 12.1, and the second is from Problem 5.3 of my 1988 *More Programming Pearls* (the blood sample trick is from page 45 of that book). The back-of-the-envelope material is directly from Column 7, while the rules of thumb are scattered throughout the book. Three of the four software debugging stories are from Section 5.10; the medical story is from the book cited at the end of the section.

An Estimation Quiz (Appendix 2 of Programming Pearls)



Back-of-the-envelope calculations all start with basic quantities. Those numbers can sometimes be found in a problem specification (such as a requirements document), but other times they must be estimated.

This little quiz is designed to help you evaluate your proficency as a number guesser. For each question, fill in upper and lower bounds that, *in your opinion*, give you a ninety percent chance of including the true value; try not to make your ranges too narrow or too wide. I estimate that it should take you between five and ten minutes. Please make a good faith effort.

[_____, _____] January 1, 2000, population of the United States in millions.

[_____, _____] The year of Napoleon's birth.

[_____, _____] Length of the Mississippi-Missouri river in miles.

[_____, _____] Maximum takeoff weight in pounds of a Boeing 747 airliner.

[_____, _____] Seconds for a radio signal to travel from the earth to the moon.

[_____, _____] Latitude of London.

[_____, _____] Minutes for a space shuttle to orbit the earth.

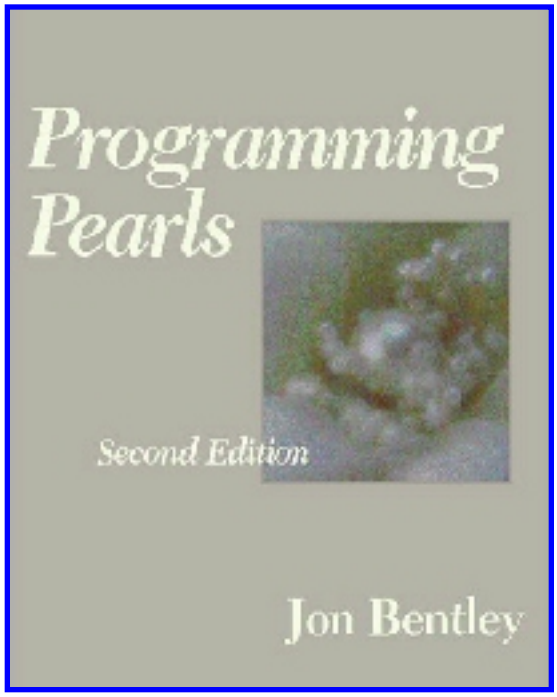
[_____, _____] Length in feet between the towers of the Golden Gate Bridge.

[_____, _____] Number of signers of the Declaration of Independence.

[_____, _____] Number of bones in the adult human body.

When you finish this quiz, please visit here for answers and interpretation.

Estimation Answers (From Appendix 2 of Programming Pearls)



If you have not yet answered all the questions yourself, please go back here and do so. Here are the answers, from an almanac or similar source.

January 1, 2000, population of the United States is **272.5** million.
Napoleon was born in **1769**.
The Mississippi-Missouri river is **3,710** miles long.
Maximum takeoff weight of a B747-400 airliner is **875,000** pounds.
A radio signal travels from the earth to the moon in **1.29** seconds.
Latitude of London is about **51.5** degrees.
A space shuttle orbits the earth in about **91** minutes.
4200 feet between the towers of the Golden Gate Bridge.
56 signers of the Declaration of Independence.
206 bones in the adult human body.

Interpretation

Please count how many of your ranges included the correct answer. Because you used a 90-percent confidence interval, you should have gotten about nine of the ten answers correct.

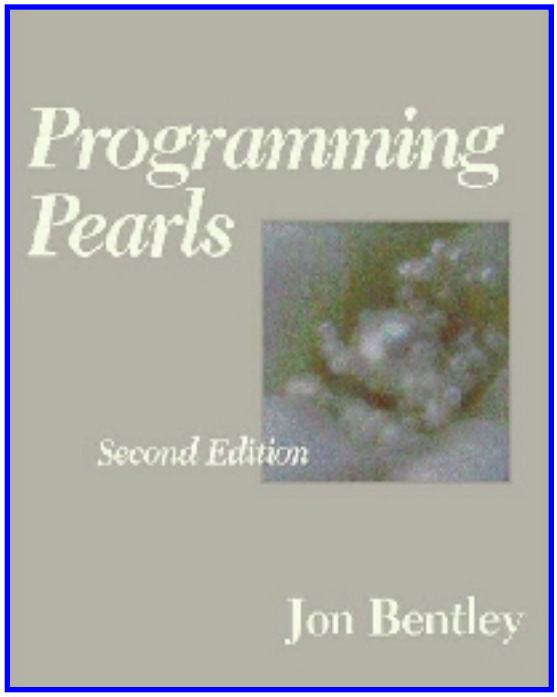
If you had all ten answers correct, then you may be an excellent estimator. Then again, your ranges may be way too big. You can probably guess which.

If you had six or fewer answers correct, then you are probably as embarrassed as I was when I first took a similar estimation quiz. A little practice couldn't hurt your estimation skills.

If you had seven or eight answers correct, then you are a pretty good guesser. In the future, remember to broaden your 90-percent ranges just a bit.

If you had exactly nine answers correct, then you may be an excellent guesser. Or, you may have given infinite ranges for the first nine questions, and zero for the last question. If so, shame on you.

The Back of the Envelope (Column 7 of Programming Pearls)



A Story

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.

My response went something like this. I figured that near its mouth the river was about a mile wide and maybe twenty 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 a hundred and twenty miles per day. Multiplying

1 mile x 1/250 mile x 120 miles/day ~ 1/2 mile^3/day

showed that the river discharged about half a cubic mile of water per day, to within an order of magnitude. But so what?

At that point Martin picked up from his desk a proposal for the communication system that his organization was building for the Summer Olympic games, and went through a similar sequence of calculations. He estimated one key parameter as we spoke by measuring the time required to send himself a one-character piece of mail. The rest of his numbers were straight from the proposal and therefore quite precise. His calculations were just as simple as those about the Mississippi River and much more revealing. They showed that, under generous assumptions, the proposed system could work 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. (The conversation took place about a year before the event, and the final system was used during the Olympics without a hitch.)

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

The Rest of the Column

The story at the top of this page begins Column 7 of *Programming Pearls*. These are the remaining sections in the column.

Related Content

The Solutions to Column 7 give answers for some of the Problems.

Appendix 2 is an estimation quiz to help you test your guessing skills.

Appendix 3 describes programs that generate cost models for the space (spacemod.cpp) and time (timemod.c) used by C and C++ programs.

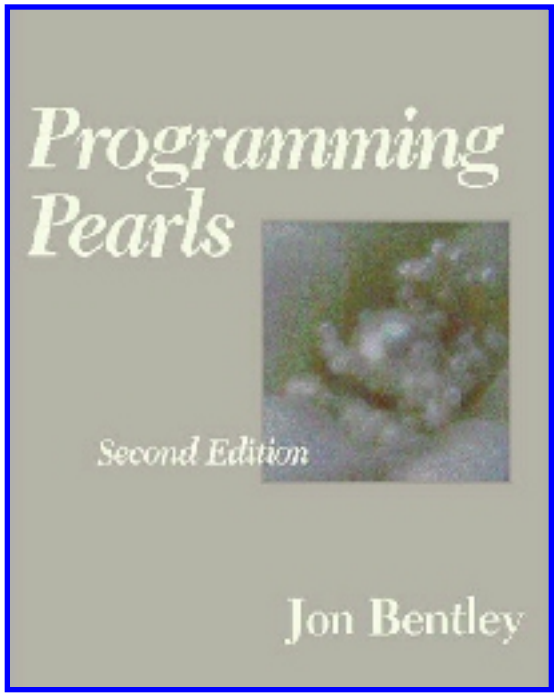
The index has an entry for the back of the envelope.

The teaching material contains overhead transparencies based on Sections 7.1, 7.5 and 7.6; the slides are available in both Postscript and Acrobat.

The web references describe several web sites devoted to this topic.

An excerpt from this column appears in the September/October 1999 issue of *IEEE Software* (pages 61-65). It consists of the lead story, much of Section 7.1, much of Section 7.2, Section 7.5 and the Estimation Quiz.

First-Year Instruction in Programming Pearls



Owen Astrachan organized a Workshop on First Year Instruction sponsored by the National Science Foundation and the Duke University Computer Science Department in July, 2000. When he invited me to give the keynote, my first response was that I haven't taught a freshman course for a quarter of a century.

On the other hand, I've enjoyed talking to undergraduates over the years, and *Programming Pearls* has been widely used as a supplementary text in college courses. My interest in the topic was also increased because my son was heading off to college later that summer. With Owen's guidance and encouragement, I agreed to talk on ``Truth, Beauty, and Engineering: What I'd Like My CS Freshman Kid to Learn Next Year''. (Let me clarify as a proud parent that Computer Science was only one of several possible majors that he was considering as he departed; I have high hopes that he can avoid repeating the sins of his father.)

The talk is available as a Powerpoint Show. It has this rough outline:

- Engineering
  - Tricks of the Trade
  - Eyes Open to the World
- Beauty
  - Elegance
  - Communication
- Truth
  - History
  - Ethics

Several of these topics are covered in the book.

Tricks of the Trade

This topic has its own web page, complete with a talk.

Elegance

Elegance is the moral of Column 1, and a theme through the rest of the book, complete with its own entry in the index. For a more general view of the topic, see the Computerworld article on Elegance in software.

I had the luxury of striving for elegance in the source code for the book. The scaffolding is often (appropriately) rough, but I took same pains to trim excess fat from most of the code that found its way into the text.

History

The book is peppered with little historical anecdotes that illustrate timeless truths. Section 10.1 introduces the topic of ``squeezing space'' with this story from the dawn of computing:

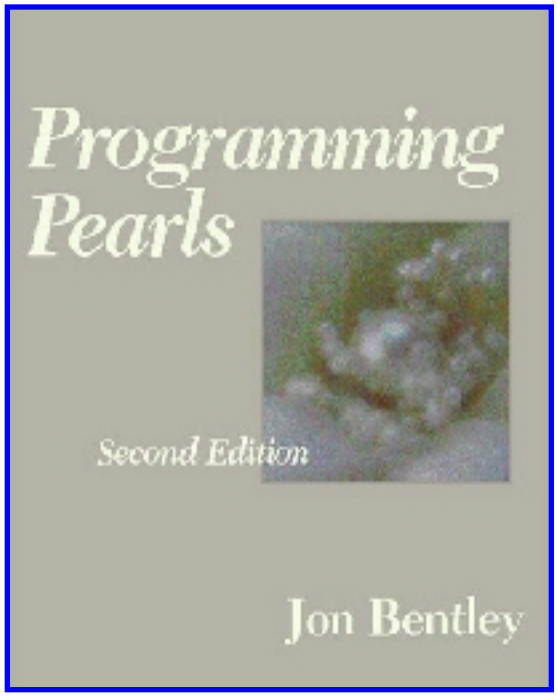
``Fred Brooks observed the power of simplification when he wrote a payroll program for a national company in the mid 1950's. The bottleneck of the program was the representation of the Kentucky state income tax. The tax was specified in the law by a two-dimensional table with income as one dimension, and number of exemptions as the other. Storing the table explicitly required thousands of words of memory, more than the capacity of the machine.

``The first approach Brooks tried was to fit a mathematical function through the tax table, but it was so jagged that no simple function would come close. Knowing that it was made by legislators with no predisposition to crazy mathematical functions, Brooks consulted the minutes of the Kentucky legislature to see what arguments had led to the bizarre table. He found that the Kentucky tax was a simple function of the income that remained *after* federal tax was deducted. His program therefore calculated federal tax from existing tables, and then used the remaining income and a table of just a few dozen words of memory to find the Kentucky tax.

``By studying the context in which the problem arose, Brooks was able to replace the original problem to be solved with a simpler problem. While the original problem appeared to require thousands of words of data space, the modified problem was solved with a negligible amount of memory.''

I tried to sprinkle such historical nuggets throughout the book. From the 1960's, Sections 2.4 and 2.8 describe a program for computing anagrams, and Section 9.3 describes a svelte binary search. From the 1970's, Problem 2.6 describes a touch-tone phone directory, Column 8 describes the evolution of an algorithm, Section 10.3 describes storing symmetric matrices, and Section 15.2 describes suffix arrays.

Code from Programming Pearls



- Column 1: Programs for sorting integers
  bitsort.c -- Sort with bit vectors.
  sortints.cpp -- Sort using C++ STL sets.
  qsortints.c -- Sort with C library qsort.
  bitsortgen.c -- Generate random integers for sorting.

- Column 2: Test and time algorithms
  rotate.c -- Three ways to rotate the elements of a vector.
  The next two program are used in a pipeline to compute all anagrams in a dictionary
  sign.c -- Sign each word by its letters in sorted order.
  squash.c -- Put each anagram class on a single line.

- Column 5: Scaffolding for testing and timing search functions
  search.c -- Linear and binary search.

- Column 7: Tiny experiment on C run times
  timemod0.c -- Edit main to time one operation.

- Column 8: Compute the maximum-sum subsequence in an array
  maxsum.c -- Time four algs: n^3, n^2, n log n, n.

- Column 9: Code tuning programs
  genbins.c -- Profile this, then try a special-purpose allocator.
  macfun.c -- Time the cost of macros and functions.
  The column also uses rotate.c (Column 2), search.c (Column 5) and maxsum.c (Column 8).

- Column 11: Test and time sorting algorithms
  sort.cpp -- Mostly C, but also C++ sort function.
  SortAnim.java -- Animate those sort functions in Java.

- Column 12: Generate a sorted list of random integers
  sortedrand.cpp -- Several algorithms for the task.

- Column 13: Set representations for the problem in Column 12
  sets.cpp -- Several data structures for sets.
  genbins.c (Column 9) implements the bin data structure in C.

- Column 14: Heaps

priqueue.cpp -- Implement and test priority queues.
The column also uses sort.c (Column 11) for heapsort.

- Column 15: Strings
  wordlist.cpp -- List words in the file, using STL set.
  wordfreq.cpp -- List words in the file, with counts, using STL map.
  wordfreq.c -- Same as above, with hash table in C.
  longdup.c -- Find long repeated strings in input.
  markov.c -- Generate random text from input.
  markovhash.c -- Like markov.c, but with hashing.
  markovlet.c -- Letter-level markov text, simple algorithm.

- Appendix 3: Cost Models
  spacemod.cpp -- Space used by various records.
  timemod.c -- Table of times used by various C constructs.

You may use this code for any purpose, as long as you leave the copyright notice and book citation attached.

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* bitsort.c -- bitmap sort from Column 1
 *   Sort distinct integers in the range [0..N-1]
 */

#include <stdio.h>

#define BITSPERWORD 32
#define SHIFT 5
#define MASK 0x1F
#define N 10000000
int a[1 + N/BITSPERWORD];

void set(int i) {        a[i>>SHIFT] |=  (1<<(i & MASK)); }
void clr(int i) {        a[i>>SHIFT] &= ~(1<<(i & MASK)); }
int  test(int i){ return a[i>>SHIFT] &   (1<<(i & MASK)); }

int main()
{       int i;
        for (i = 0; i < N; i++)
                clr(i);
/*      Replace above 2 lines with below 3 for word-parallel init
        int top = 1 + N/BITSPERWORD;
        for (i = 0; i < top; i++)
                a[i] = 0;
 */
        while (scanf("%d", &i) != EOF)
                set(i);
        for (i = 0; i < N; i++)
                if (test(i))
                        printf("%d\n", i);
        return 0;
}
```

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* sortints.cpp -- Sort input set of integers using STL set */

#include <iostream>
#include <set>
using namespace std;

int main()
{       set<int> S;
        int i;
        set<int>::iterator j;
        while (cin >> i)
                S.insert(i);
        for (j = S.begin(); j != S.end(); ++j)
                cout << *j << "\n";
        return 0;
}
```

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* qsortints.c -- Sort input set of integers using qsort */

#include <stdio.h>
#include <stdlib.h>

int intcomp(int *x, int *y)
{   return *x - *y;
}

int a[1000000];

int main()
{   int i, n=0;
    while (scanf("%d", &a[n]) != EOF)
        n++;
    qsort(a, n, sizeof(int), intcomp);
    for (i = 0; i < n; i++)
        printf("%d\n", a[i]);
    return 0;
}
```

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* bitsortgen.c -- gen $1 distinct integers from U[0,$2) */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAXN 2000000
int x[MAXN];

int randint(int a, int b)
{       return a + (RAND_MAX * rand() + rand()) % (b + 1 - a);
}

int main(int argc, char *argv[])
{       int i, k, n, t, p;
        srand((unsigned) time(NULL));
        k = atoi(argv[1]);
        n = atoi(argv[2]);
        for (i = 0; i < n; i++)
                x[i] = i;
        for (i = 0; i < k; i++) {
                p = randint(i, n-1);
                t = x[p]; x[p] = x[i]; x[i] = t;
                printf("%d\n", x[i]);
        }
        return 0;
}
```

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* rotate.c -- time algorithms for rotating a vector
        Input lines:
                algnum numtests n rotdist
                algnum:
                  1: reversal algorithm
                  2: juggling algorithm
                  22:  juggling algorithm with mod rather than if
                  3: gcd algorithm
                  4: slide (don't rotate): baseline alg for timing
        To test the algorithms, recompile and change main to call testrot
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAXN 10000000

int x[MAXN];
int rotdist, n;

/* Alg 1: Rotate by reversal */

void reverse(int i, int j)
{       int t;
        while (i < j) {
                t = x[i]; x[i] = x[j]; x[j] = t;
                i++;
                j--;
        }
}

void revrot(int rotdist, int n)
{       reverse(0, rotdist-1);
        reverse(rotdist, n-1);
        reverse(0, n-1);
}

/* Alg 2: Juggling (dolphin) rotation */

int gcd(int i, int j)
{       int t;
        while (i != 0) {
                if (j >= i)
                        j -= i;
                else {
                        t = i; i = j; j = t;
                }
        }
        return j;
}

void jugglerot(int rotdist, int n)
{       int cycles, i, j, k, t;
        cycles = gcd(rotdist, n);
        for (i = 0; i < cycles; i++) {
                /* move i-th values of blocks */
                t = x[i];
                j = i;
```

```
                for (;;) {
                        k = j + rotdist;
                        if (k >= n)
                                k -= n;
                        if (k == i)
                                break;
                        x[j] = x[k];
                        j = k;
                }
                x[j] = t;
        }
}

void jugglerot2(int rotdist, int n)
{       int cycles, i, j, k, t;
        cycles = gcd(rotdist, n);
        for (i = 0; i < cycles; i++) {
                /* move i-th values of blocks */
                t = x[i];
                j = i;
                for (;;) {
          /* Replace with mod below
                        k = j + rotdist;
                        if (k >= n)
                                k -= n;
           */
                         k = (j + rotdist) % n;
                        if (k == i)
                                break;
                        x[j] = x[k];
                        j = k;
                }
                x[j] = t;
        }
}

/* Alg 3: Recursive rotate (using gcd structure) */

void swap(int i, int j, int k) /* swap x[i..i+k-1] with x[j..j+k-1] */
{       int t;
        while (k-- > 0) {
                t = x[i]; x[i] = x[j]; x[j] = t;
                i++;
                j++;
        }

}

void gcdrot(int rotdist, int n)
{       int i, j, p;
        if (rotdist == 0 || rotdist == n)
                return;
        i = p = rotdist;
        j = n - p;
        while (i != j) {
                /* invariant:
                        x[0  ..p-i  ] is in final position
                        x[p-i..p-1  ] = a (to be swapped with b)
                        x[p  ..p+j-1] = b (to be swapped with a)
                        x[p+j..n-1  ] in final position
                */
```

```
                if (i > j) {
                        swap(p-i, p, j);
                        i -= j;
                } else {
                        swap(p-i, p+j-i, i);
                        j -= i;
                }
        }
        swap(p-i, p, i);
}

int isogcd(int i, int j)
{       if (i == 0) return j;
        if (j == 0) return i;
        while (i != j) {
                if (i > j)
                        i -= j;
                else
                        j -= i;
        }
        return i;
}

void testgcd()
{
        int i,j;
        while (scanf("%d %d", &i, &j) != EOF)
                printf("%d\n", isogcd(i,j) );
}

/* Test all algs */

void slide(int rotdist, int n) /* Benchmark: slide left rotdist (lose 0..rotdist-1) */
{       int i;

        for (i = rotdist; i < n; i++)
                x[i-rotdist] = x[i];
}

void initx()
{
        int i;
        for (i = 0; i < n; i++)
                x[i] = i;
}

void printx()
{       int i;
        for (i = 0; i < n; i++)
                printf(" %d", x[i]);
        printf("\n");
}

void roterror()
{
        fprintf(stderr, " rotate bug %d %d\n", n, rotdist);
        printx();
        exit (1);
}

void checkrot()
{       int i;
```

```
        for (i = 0; i < n-rotdist; i++)
                if (x[i] != i+rotdist)
                        roterror();
        for (i = 0; i < rotdist; i++)
                if (x[n-rotdist+i] != i)
                        roterror();
}

void testrot()
{       for (n = 1; n <= 20; n++) {
                printf(" testing n=%d\n", n);
                for (rotdist = 0; rotdist <= n; rotdist++) {
                        /* printf("  testing rotdist=%d\n", rotdist); */
                        initx(); revrot(rotdist, n);     checkrot();
                        initx(); jugglerot(rotdist, n);  checkrot();
                        initx(); jugglerot2(rotdist, n); checkrot();
                        initx(); gcdrot(rotdist, n);     checkrot();
                }
        }
}

/* Timing */

void timedriver()
{       int i, algnum, numtests, start, clicks;
        while (scanf("%d %d %d %d", &algnum, &numtests, &n, &rotdist) != EOF) {
                initx();
                start = clock();
                for (i = 0; i < numtests; i++) {
                        if (algnum == 1)
                                revrot(rotdist, n);
                        else if (algnum == 2)
                                jugglerot(rotdist, n);
                        else if (algnum == 22)
                                jugglerot2(rotdist, n);
                        else if (algnum == 3)
                                gcdrot(rotdist, n);
                        else if (algnum == 4)
                                slide(rotdist, n);
                }
                clicks = clock() - start;
                printf("%d\t%d\t%d\t%d\t%d\t%g\n",
                        algnum, numtests, n, rotdist, clicks,
                        1e9*clicks/((float) CLOCKS_PER_SEC*n*numtests));
        }
}

/* Main */

int main()
{       /* testrot(); */
        timedriver();
        return 0;
}
```

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* sign.c -- sign each line of a file for finding anagrams
    The input line "stop" gives the output line "opst stop"
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define WORDMAX 100

int charcomp(char *x, char *y)
{   return *x - *y;
}

int main()
{   char word[WORDMAX], sig[WORDMAX];
    while (scanf("%s", word) != EOF) {
        strcpy(sig, word);
        qsort(sig, strlen(sig), sizeof(char), charcomp);
        printf("%s %s\n", sig, word);
    }
    return 0;
}
```

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* squash.c -- print anagram classes on a single line
    The input lines "opst pots" and "opst stop" go to "pots stop"
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define WORDMAX 100

int main()
{   char word[WORDMAX], sig[WORDMAX], oldsig[WORDMAX];
    int linenum = 0;
    strcpy(oldsig, "");
    while (scanf("%s %s", sig, word) != EOF) {
        if (strcmp(oldsig, sig) != 0 && linenum > 0)
            printf("\n");
        strcpy(oldsig, sig);
        linenum++;
        printf("%s ", word);
    }
    printf("\n");
    return 0;
}
```

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* search.c -- test and time binary and sequential search
   Select one of three modes by editing main() below.
   1.) Probe one function
   2.) Test one function extensively
   3.) Time all functions
               Input lines:  algnum n numtests
               Output lines: algnum n numtests clicks nanosecs_per_elem
               See timedriver for algnum codes
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAXN 1000000

typedef int DataType;

DataType x[MAXN];
int n;

/* Scaffolding */

int i = -999999;
#define assert(v) { if ((v) == 0) printf("  binarysearch bug %d %d\n", i, n); }

/* Alg 1: From Programming Pearls, Column 4: raw transliteration */

int binarysearch1(DataType t)
{       int l, u, m;
        l = 0;
        u = n-1;
        for (;;) {
                if (l > u)
                        return -1;
                m = (l + u) / 2;
                if (x[m] < t)
                        l = m+1;
                else if (x[m] == t)
                        return m;
                else /* x[m] > t */
                        u = m-1;
        }
}

/* Alg 2: Make binarysearch1 more c-ish */

int binarysearch2(DataType t)
{       int l, u, m;
        l = 0;
        u = n-1;
        while (l <= u) {
                m = (l + u) / 2;
                if (x[m] < t)
                        l = m+1;
                else if (x[m] == t)
                        return m;
                else /* x[m] > t */
                        u = m-1;
```

```
        }
        return -1;
}

/* Alg 3: From PP, Col 8 */

int binarysearch3(DataType t)
{       int l, u, m;
        l = -1;
        u = n;
        while (l+1 != u) {
                m = (l + u) / 2;
                if (x[m] < t)
                        l = m;
                else
                        u = m;
        }
        if (u >= n || x[u] != t)
                return -1;
        return u;
}

/* Alg 4: From PP, Col 9 */

int binarysearch4(DataType t)
{       int l, p;
        if (n != 1000)
                return binarysearch3(t);
        l = -1;
        if (x[511]   < t) l = 1000 - 512;
        if (x[l+256] < t) l += 256;
        if (x[l+128] < t) l += 128;
        if (x[l+64 ] < t) l += 64;
        if (x[l+32 ] < t) l += 32;
        if (x[l+16 ] < t) l += 16;
        if (x[l+8  ] < t) l += 8;
        if (x[l+4  ] < t) l += 4;
        if (x[l+2  ] < t) l += 2;
        if (x[l+1  ] < t) l += 1;
        p = l+1;
        if (p >= n || x[p] != t)
                return -1;
        return p;
}

/* Alg 9: Buggy, from Programming Pearls, Column 5 */

int sorted()
{   int i;
    for (i = 0; i < n-1; i++)
        if (x[i] > x[i+1])
            return 0;
    return 1;
}

int binarysearch9(DataType t)
{       int l, u, m;
/* int oldsize, size = n+1; */
        l = 0;
        u = n-1;
        while (l <= u) {
/* oldsize = size;
```

```
size = u - l +1;
assert(size < oldsize); */
                m = (l + u) / 2;
/* printf("  %d %d %d\n", l, m, u); */
                if (x[m] < t)
                        l = m;
                else if (x[m] > t)
                        u = m;
                else {
                        /* assert(x[m] == t); */
                        return m;
                }
        }
        /* assert(x[l] > t && x[u] < t); */
        return -1;
}

/* Alg 21: Simple sequential search */

int seqsearch1(DataType t)
{       int i;
        for (i = 0; i < n; i++)
                if (x[i] == t)
                        return i;
        return -1;
}

/* Alg 22: Faster sequential search: Sentinel */

int seqsearch2(DataType t)
{       int i;
        DataType hold = x[n];
        x[n] = t;
        for (i = 0; ; i++)
                if (x[i] == t)
                        break;
        x[n] = hold;
        if (i == n)
                return -1;
        else
                return i;
}

/* Alg 23: Faster sequential search: loop unrolling */

int seqsearch3(DataType t)
{       int i;
        DataType hold = x[n];
        x[n] = t;
        for (i = 0; ; i+=8) {
                if (x[i] == t)   {          break; }
                if (x[i+1] == t) { i += 1; break; }
                if (x[i+2] == t) { i += 2; break; }
                if (x[i+3] == t) { i += 3; break; }
                if (x[i+4] == t) { i += 4; break; }
                if (x[i+5] == t) { i += 5; break; }
                if (x[i+6] == t) { i += 6; break; }
                if (x[i+7] == t) { i += 7; break; }
        }
        x[n] = hold;
        if (i == n)
                return -1;
```

```
        else
                return i;
}


/* Scaffolding to probe one algorithm */

void probe1()
{       int i;
        DataType t;
        while (scanf("%d %d", &n, &t) != EOF) {
                for (i = 0; i < n; i++)
                        x[i] = 10*i;
                printf(" %d\n", binarysearch9(t));
        }
}


/* Torture test one algorithm */

#define s seqsearch3
void test(int maxn)
{       int i;
        for (n = 0; n <= maxn; n++) {
                printf("n=%d\n", n);
                /* distinct elements (plus one at top) */
                for (i = 0; i <= n; i++)
                        x[i] = 10*i;
                for (i = 0; i < n; i++) {
                        assert(s(10*i)     ==  i);
                        assert(s(10*i - 5) == -1);
                }
                assert(s(10*n - 5) == -1);
                assert(s(10*n)     == -1);
                /* equal elements */
                for (i = 0; i < n; i++)
                        x[i] = 10;
                if (n == 0) {
                        assert(s(10) == -1);
                } else {
                        assert(0 <= s(10) && s(10) < n);
                }
                assert(s(5) == -1);
                assert(s(15) == -1);
        }
}

/* Timing */

int p[MAXN];

void scramble(int n)
{       int i, j;
        DataType t;
        for (i = n-1; i > 0; i--) {
                j = (RAND_MAX*rand() + rand()) % (i + 1);
                t = p[i]; p[i] = p[j]; p[j] = t;
        }
}

void timedriver()
{       int i, algnum, numtests, test, start, clicks;
```

```
        while (scanf("%d %d %d", &algnum, &n, &numtests) != EOF) {
                for (i = 0; i < n; i++)
                        x[i] = i;
                for (i = 0; i < n; i++)
                        p[i] = i;
                scramble(n);
                start = clock();
                for (test = 0; test < numtests; test++) {
                        for (i = 0; i < n; i++) {
                                switch (algnum) {
                                case 1:  assert(binarysearch1(p[i]) == p[i]); break;
                                case 2:  assert(binarysearch2(p[i]) == p[i]); break;
                                case 3:  assert(binarysearch3(p[i]) == p[i]); break;
                                case 4:  assert(binarysearch4(p[i]) == p[i]); break;
                                case 9:  assert(binarysearch9(p[i]) == p[i]); break;
                                case 21: assert(seqsearch1(p[i]) == p[i]); break;
                                case 22: assert(seqsearch2(p[i]) == p[i]); break;
                                case 23: assert(seqsearch3(p[i]) == p[i]); break;
                                }
                        }
                }
                clicks = clock() - start;
                printf("%d\t%d\t%d\t%d\t%g\n",
                        algnum, n, numtests, clicks,
                        1e9*clicks/((float) CLOCKS_PER_SEC*n*numtests));
        }
}

/* Main */

int main()
{       /* probe1(); */
        /* test(25); */
        timedriver();
        return 0;
}
```

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* timemod0.c -- Simple experiments on C run time costs */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int main()
{       int i, n, ia, ib, ic;
        float fa, fb, fc;
        n = 1000000000; /* run time in secs gives nanosecs/loop */
        ia = ib = ic = 9;
        fa = fb = 9.0;
        for (i = 0; i < n; i++) {
                /* null loop            19.1 */
                /* ia = ib + ic;        17.7 */
                /* ia = ib - ic;        17.6 */
                /* ia = ib * ic;        17.7 */
                /* ia = ib % ic;        98.3 */
                /* ia = ib / ic;        98.3 */
                /* ia = rand();         41.5 */
                /* fa = sqrt(fb);       184 */
                /* free(malloc(8));     2400 */
        }
        return 0;
}
```

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* maxsum.c -- time algs for maximum-sum subsequence
 * Input:  algnum, n
 * Output: algnum, n, answer, ticks, secs
 *              See main for algnum codes
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAXN 10000000
int n;
float x[MAXN];

void sprinkle() /* Fill x[n] with reals uniform on [-1,1] */
{   int i;
    for (i = 0; i < n; i++)
        x[i] = 1 - 2*( (float) rand()/RAND_MAX);
}

float alg1()
{   int i, j, k;
    float sum, maxsofar = 0;
    for (i = 0; i < n; i++)
        for (j = i; j < n; j++) {
            sum = 0;
            for (k = i; k <= j; k++)
                sum += x[k];
            if (sum > maxsofar)
                maxsofar = sum;
        }
    return maxsofar;
}

float alg2()
{   int i, j;
    float sum, maxsofar = 0;
    for (i = 0; i < n; i++) {
        sum = 0;
        for (j = i; j < n; j++) {
            sum += x[j];
            if (sum > maxsofar)
                maxsofar = sum;
        }
    }
    return maxsofar;
}

float cumvec[MAXN+1];

float alg2b()
{   int i, j;
    float *cumarr, sum, maxsofar = 0;
    cumarr = cumvec+1; /* to access cumarr[-1] */
    cumarr[-1] = 0;
    for (i = 0; i < n; i++)
        cumarr[i] = cumarr[i-1] + x[i];
    for (i = 0; i < n; i++) {
        for (j = i; j < n; j++) {
```

```
            sum = cumarr[j] - cumarr[i-1];
            if (sum > maxsofar)
                maxsofar = sum;
        }
    }
    return maxsofar;
}

/* MS VC++ has a max macro, and therefore a perf bug */

#ifdef max
#undef max
#endif


#define maxmac(a, b) ((a) > (b) ? (a) : (b) )

float maxfun(float a, float b)
{   return a > b ? a : b;
}

#define max(a, b) maxfun(a, b)

float recmax(int l, int u)
{   int i, m;
    float lmax, rmax, sum;
    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];
                if (sum > lmax)
                        lmax = sum;
    }
        /* find max crossing to right */
    rmax = sum = 0;
    for (i = m+1; i <= u; i++) {
                sum += x[i];
                if (sum > rmax)
                        rmax = sum;
    }
    return max(lmax + rmax,
                max(recmax(l, m), recmax(m+1, u)));
}

float alg3()
{   return recmax(0, n-1);
}

float alg4()
{   int i;
    float maxsofar = 0, maxendinghere = 0;
    for (i = 0; i < n; i++) {
        maxendinghere += x[i];
        if (maxendinghere < 0)
            maxendinghere = 0;
        if (maxsofar < maxendinghere)
            maxsofar = maxendinghere;
```

```
    }
    return maxsofar;
}

float alg4b()
{   int i;
    float maxsofar = 0, maxendinghere = 0;
    for (i = 0; i < n; i++) {
        maxendinghere += x[i];
        maxendinghere = maxmac(maxendinghere, 0);
        maxsofar = maxmac(maxsofar, maxendinghere);
    }
    return maxsofar;
}

float alg4c()
{   int i;
    float maxsofar = 0, maxendinghere = 0;
    for (i = 0; i < n; i++) {
        maxendinghere += x[i];
        maxendinghere = maxfun(maxendinghere, 0);
        maxsofar = maxfun(maxsofar, maxendinghere);
    }
    return maxsofar;
}

int main()
{   int algnum, start, clicks;
    float thisans;

    while (scanf("%d %d", &algnum, &n) != EOF) {
        sprinkle();
        start = clock();
        thisans = -1;
        switch (algnum) {
                        case 1:  thisans = alg1();  break;
                        case 2:  thisans = alg2();  break;
                        case 22: thisans = alg2b(); break;
                        case 3:  thisans = alg3();  break;
                        case 4:  thisans = alg4();  break;
                        case 42: thisans = alg4b(); break;
                        case 43: thisans = alg4c(); break;
                        default: break;
                }
        clicks = clock()-start;
        printf("%d\t%d\t%f\t%d\t%f\n", algnum, n, thisans,
            clicks, clicks / (float) CLOCKS_PER_SEC);
        if  (alg4() != thisans)
            printf(" maxsum error: mismatch with alg4: %f\n", alg4());
    }
    return 0;
}
```

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* genbins.c -- generate random numbers with bins */

/* If NODESIZE is 8, this program uses the special-case malloc.
   Change NODESIZE to 0 to use the system malloc.
 */

#include <stdio.h>
#include <stdlib.h>

#define NODESIZE 8
#define NODEGROUP 1000
int nodesleft = 0;
char *freenode;

void *pmalloc(int size)
{       void *p;
        if (size != NODESIZE)
                return malloc(size);
        if (nodesleft == 0) {
                freenode = malloc(NODEGROUP*NODESIZE);
                nodesleft = NODEGROUP;
        }
        nodesleft--;
        p = (void *) freenode;
        freenode += NODESIZE;
        return p;
}

struct node {
        int val;
        struct node *next;
};

struct node **bin, *sentinel;
int bins, bincnt, maxval;

void initbins(int maxelms, int pmaxval)
{       int i;
        bins = maxelms;
        maxval = pmaxval;
        bin = pmalloc(bins*sizeof(struct node *));
        sentinel = pmalloc(sizeof(struct node));
        sentinel->val = maxval;
        for (i = 0; i < bins; i++)
                bin[i] = sentinel;
        bincnt = 0;
}

struct node *rinsert(struct node *p, int t)
{       if (p->val < t) {
                p->next = rinsert(p->next, t);
        } else if (p->val > t) {
                struct node *q = pmalloc(sizeof(struct node));
                q->val = t;
                q->next = p;
                p = q;
                bincnt++;
        }
        return p;
```

```
}

void insert(int t)
{       int i;
        i = t / (1 + maxval/bins);
        i = t / (1 + maxval/bins);
        bin[i] = rinsert(bin[i], t);
}

void report()
{       int i, j = 0;
        struct node *p;
        for (i = 0; i < bins; i++)
                for (p = bin[i]; p != sentinel; p = p->next)
                        /* printf("%d\n", p->val) */;
                        /* Uncomment for testing, comment for profiling */
}

int bigrand()
{       return RAND_MAX*rand() + rand();
}

int main(int argc, char *argv[])
{       int m = atoi(argv[1]);
        int n = atoi(argv[2]);
        initbins(m, n);
        while (bincnt < m) {
                insert(bigrand() % n);
        }
        report();
        return 0;
}
```

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* macfun.c -- time macro and function implementations of max
 * Input: a sequence of (alg num, n) pairs.
 * Output: for each test, (alg num, n, ans, ticks, secs)
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAXN 1000000
float x[MAXN];


/* arrmax1 -- max is a macro */

#define max1(a, b) ((a) > (b) ? (a) : (b))

float arrmax1(int n)
{       if (n == 1)
                return x[0];
        else
                return max1(x[n-1], arrmax1(n-1));
}


/* arrmax2 -- max is a function */

float max2(float a, float b)
{       return a > b ? a : b;
}

float arrmax2(int n)
{       if (n == 1)
                return x[0];
        else
                return max2(x[n-1], arrmax2(n-1));
}


/* arrmax3 -- MS VC++ stdlib defines max as a macro */

#ifndef max
#define max(a, b) max2(a, b)
#endif

float arrmax3(int n)
{       if (n == 1)
                return x[0];
        else
                return max(x[n-1], arrmax3(n-1));
}


int main()
{   int algnum, i, n, start, clicks;
    float thisans;

        for (i = 0; i < MAXN; i++)
                x[i] = MAXN-i;
        while (scanf("%d %d", &algnum, &n) != EOF) {
                start = clock();
```

```
                thisans = -1;
                switch (algnum) {
                        case 1: thisans = arrmax1(n); break;
                        case 2: thisans = arrmax2(n); break;
                        case 3: thisans = arrmax3(n); break;
                        default: break;
                }
                clicks = clock()-start;
                printf("%d\t%d\t%g\t%d\t%g\n", algnum, n, thisans,
                        clicks, clicks / (float) CLOCKS_PER_SEC);
    }
        return 0;
}
```

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* sort.cpp -- test and time sorting algorithms
        Input lines:  algnum n mod
        Output lines: algnum n mod clicks nanosecs_per_elem
   This is predominantly a C program; the only use of C++
   sort function immediately below the include line.
 */

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// To change from C++ back to C, remove the following two lines
// and the call to sort in main
#include <algorithm>
using namespace std;

/* Data and supporting functions */

#define MAXN 10000000

typedef int DType;

DType realx[MAXN];
int *x = realx; /* allow x to shift for heaps */
int n;

void swap(int i, int j)
{       DType t = x[i];
        x[i] = x[j];
        x[j] = t;
}

int randint(int l, int u)
{       return l + (RAND_MAX*rand() + rand()) % (u-l+1);
}

/* LIBRARY QSORT */

int intcomp(int *x, int *y)
{       return *x - *y;
}

/* INSERTION SORTS */

/* Simplest insertion sort */
void isort1()
{       int i, j;
        for (i = 1; i < n; i++)
                for (j = i; j > 0 && x[j-1] > x[j]; j--)
                        swap(j-1, j);
}

/* Write swap function inline */
void isort2()
{       int i, j;
        DType t;
        for (i = 1; i < n; i++)
                for (j = i; j > 0 && x[j-1] > x[j]; j--) {
                        t = x[j];
                        x[j] = x[j-1];
```

```
                        x[j-1] = t;
                }
}

/* Move assignments to and from t out of loop */
void isort3()
{       int i, j;
        DType t;
        for (i = 1; i < n; i++) {
                t = x[i];
                for (j = i; j > 0 && x[j-1] > t; j--)
                        x[j] = x[j-1];
                x[j] = t;
        }
}

/* QUICKSORTS */

/* Simplest version, Lomuto partitioning */
void qsort1(int l, int u)
{       int i, m;
        if (l >= u)
                return;
        m = l;
        for (i = l+1; i <= u; i++)
                if (x[i] < x[l])
                        swap(++m, i);
        swap(l, m);
        qsort1(l, m-1);
        qsort1(m+1, u);
}

/* Sedgewick's version of Lomuto, with sentinel */
void qsort2(int l, int u)
{       int i, m;
        if (l >= u)
                return;
        m = i = u+1;
        do {
                do i--; while (x[i] < x[l]);
                swap(--m, i);
        } while (i > l);
        qsort2(l, m-1);
        qsort2(m+1, u);
}

/* Two-way partitioning */
void qsort3(int l, int u)
{       int i, j;
        DType t;
        if (l >= u)
                return;
        t = x[l];
        i = l;
        j = u+1;
        for (;;) {
                do i++; while (i <= u && x[i] < t);
                do j--; while (x[j] > t);
                if (i > j)
                        break;
                swap(i, j);
```

```
        }
        swap(l, j);
        qsort3(l, j-1);
        qsort3(j+1, u);
}

/* qsort3 + randomization + isort small subarrays + swap inline */
int cutoff = 50;
void qsort4(int l, int u)
{       int i, j;
        DType t, temp;
        if (u - l < cutoff)
                return;
        swap(l, randint(l, u));
        t = x[l];
        i = l;
        j = u+1;
        for (;;) {
                do i++; while (i <= u && x[i] < t);
                do j--; while (x[j] > t);
                if (i > j)
                        break;
                temp = x[i]; x[i] = x[j]; x[j] = temp;
        }
        swap(l, j);
        qsort4(l, j-1);
        qsort4(j+1, u);
}

/* selection */

void select1(int l, int u, int k)
{       int i, j;
        DType t, temp;
        if (l >= u)
                return;
        swap(l, randint(l, u));
        t = x[l];
        i = l;
        j = u+1;
        for (;;) {
                do i++; while (i <= u && x[i] < t);
                do j--; while (x[j] > t);
                if (i > j)
                        break;
                temp = x[i]; x[i] = x[j]; x[j] = temp;
        }
        swap(l, j);
        if (j < k)
                select1(j+1, u, k);
        else if (j > k)
                select1(l, j-1, k);
}

/* HEAP SORTS */

void siftup(int u)
{       int i, p;
        i = u;
        for (;;) {
                if (i == 1)
```

```
                        break;
                p = i / 2;
                if (x[p] >= x[i])
                        break;
                swap(p, i);
                i = p;
        }
}

void siftdown1(int l, int u)
{       int i, c;
        i = l;
        for (;;) {
                c = 2*i;
                if (c > u)
                        break;
                if (c+1 <= u && x[c+1] > x[c])
                        c++;
                if (x[i] > x[c])
                        break;
                swap(i, c);
                i = c;
        }
}

void siftdown1b(int l, int u) /* More C-ish version of 1 */
{       int i, c;
        for (i = l; (c = 2*i) <= u; i = c) {
                if (c+1 <= u && x[c+1] > x[c])
                        c++;
                if (x[i] > x[c])
                        break;
                swap(i, c);
        }
}

void hsort1()
{       int i;
        x--;
        for (i = 2; i <= n; i++)
                siftup(i);
        for (i = n; i >= 2; i--) {
                swap(1, i);
                siftdown1(1, i-1);
        }
        x++;
}

void hsort2()
{       int i;
        x--;
        for (i = n/2; i >= 1; i--)
                siftdown1(i, n);
        for (i = n; i >= 2; i--) {
                swap(1, i);
                siftdown1(1, i-1);
        }
        x++;
}

void siftdown3(int l, int u) /* push to bottom, then back up */
```

```
{       int i, c, p;
        i = l;
        for (;;) {
                c = 2*i;
                if (c > u)
                        break;
                if (c+1 <= u && x[c+1] > x[c])
                        c++;
                swap(i, c);
                i = c;
        }
        for (;;) {
                p = i/2;
                if (p < l)
                        break;
                if (x[p] > x[i])
                        break;
                swap(p, i);
                i = p;
        }
}

void hsort3()
{       int i;
        x--;
        for (i = n/2; i >= 1; i--)
                siftdown3(i, n);
        for (i = n; i >= 2; i--) {
                swap(1, i);
                siftdown3(1, i-1);
        }
        x++;
}

void siftdown4(int l, int u) /* replace swap with assignments */
{       int i, c, p;
        DType t;
        t = x[l];
        i = l;
        for (;;) {
                c = 2*i;
                if (c > u)
                        break;
                if (c+1 <= u && x[c+1] > x[c])
                        c++;
                x[i] = x[c];
                i = c;
        }
        x[i] = t;
        for (;;) {
                p = i/2;
                if (p < l)
                        break;
                if (x[p] > x[i])
                        break;
                swap(p, i);
                i = p;
        }
}

void hsort4()
```

```
{       int i;
        x--;
        for (i = n/2; i >= 1; i--)
                siftdown4(i, n);
        for (i = n; i >= 2; i--) {
                swap(1, i);
                siftdown4(1, i-1);
        }
        x++;
}


/* Other Sorts -- Exercises in Column 11 */

void selsort()  /* Selection sort */
{       int i, j;
        for (i = 0; i < n-1; i++)
                for (j = i; j < n; j++)
                        if (x[j] < x[i])
                                swap(i, j);
}

void shellsort()
{       int i, j, h;
        for (h = 1; h < n; h = 3*h + 1)
                ;
        for (;;) {
                h /= 3;
                if (h < 1) break;
                for (i = h; i < n; i++) {
                        for (j = i; j >= h; j -= h) {
                                if (x[j-h] < x[j]) break;
                                swap(j-h, j);
                        }
                }
        }
}



/* SCAFFOLDING */

/* Timing */

void timedriver()
{       int i, k, algnum, mod, start, clicks;
        while (scanf("%d %d %d", &algnum, &n, &mod) != EOF) {
                if (mod <= 0)
                        mod = 10*n;
                for (i = 0; i < n; i++)
                        x[i] = randint(0, mod-1);
                k = n/2;
                start = clock();
                switch (algnum) {
                case 11: qsort(x, n, sizeof(int), (int (__cdecl *)(const void *,const void
*)) intcomp); break;
                case 12: sort(x, x+n); break;
                case 21: isort1(); break;
                case 22: isort2(); break;
                case 23: isort3(); break;
                case 31: qsort1(0, n-1); break;
                case 32: qsort2(0, n-1); break;
```

```
                case 33: qsort3(0, n-1); break;
                case 34: qsort4(0, n-1); isort3(); break;
                case 41: select1(0, n-1, k); break;
                case 51: hsort1(); break;
                case 52: hsort2(); break;
                case 53: hsort3(); break;
                case 54: hsort4(); break;
                case 61: selsort(); break;
                case 62: shellsort(); break;
                }
                clicks = clock() - start;
                if (algnum == 41) { /* Test selection */
                        for (i = 0; i < k; i++)
                                if (x[i] > x[k])
                                        printf("  SELECT BUG i=%d\n", i);
                        for (i = k+1; i < n; i++)
                                if (x[i] < x[k])
                                        printf("  SELECT BUG i=%d\n", i);
                } else { /* Test sort */
                        for (i = 0; i < n-1; i++)
                                if (x[i] > x[i+1])
                                        printf("  SORT BUG i=%d\n", i);
                }
                printf("%d\t%d\t%d\t%d\t%g\n",
                        algnum, n, mod, clicks,
                        1e9*clicks/((float) CLOCKS_PER_SEC*n));
        }
}

/* Main */

int main()
{       timedriver();
        return 0;
}
```

```
// Copyright (C) 1999 Lucent Technologies
// From 'Programming Pearls' by Jon Bentley

// SortAnim.java -- Animate sorting algorithms

import java.applet.*;
import java.awt.*;
import java.util.Date;

public class SortAnim extends Applet {
        // Screen Elements
        private TextField n_text;
        private Choice dist_choices;
        private Choice alg_choices;
        private Button run_button;
        private Label msg_label;
        private Color draw_color = Color.black;
        private Color back_color = Color.white;

// SORTING DATA AND ALGS

        static private final int MAXN = 10000;
        static private int n=100;
        static private float a[] = new float[MAXN];

        // Sorting: Generate Inputs
        static private final int GEN_RAND = 0;
        static private final int GEN_ASCEND = 1;
        static private final int GEN_DESCEND = 2;
        static private int gen_num = GEN_RAND;

        private void genarray()
        {       for (int i = 0; i < n; i++) {
                        switch(gen_num) {
                        case GEN_RAND: a[i] = (float) Math.random(); break;
                        case GEN_ASCEND: a[i] = ((float) i)/n; break;
                        case GEN_DESCEND: a[i] = (float) (1.0 - ((float) i)/n); break;
                        }
                }
        }

        // Sorting: Supporting Algs
        private void baseswap(int i, int j)
        {       float t = a[i];
                a[i] = a[j];
                a[j] = t;
        }

        // Sorting: Animation Support
        static private final int MINX = 20, MAXX = 580;
        static private final int MINY = 50, MAXY = 380;
        static private float factorx, factory;
        static private boolean wantanim = true;

        private void initdisplay()
        {       Graphics g = this.getGraphics();
                Rectangle r = this.bounds();
                g.setColor(back_color);
                g.fillRect(r.x, r.y, r.width, r.height);
                factorx = ((float) MAXX-MINX) / n;
                factory = ((float) MAXY-MINY);
        }
```

```
private void draw(int i, Color c)
{       Graphics g = this.getGraphics();  // BETTER WAY?
        int d = 4;
        int px = (int) (MINX + factorx*i);
        int py = MAXY - (int)(factory*a[i]);
        g.setColor(c);
        g.drawOval(px, py, d, d);
}

private void swap(int i, int j)
{       if (wantanim) {
                draw(i, back_color);
                draw(j, back_color);
        }
        baseswap(i, j);
        if (wantanim) {
                draw(i, draw_color);
                draw(j, draw_color);
        }
}

// Sorting Algs
private void isort()
{       for (int i = 1; i < n; i++)
                for (int j = i; j > 0 && a[j-1] > a[j]; j--)
                        swap(j-1, j);
}

private void ssort()
{       for (int i = 0; i < n-1; i++)
                for (int j = i; j < n; j++)
                        if (a[j] < a[i])
                                swap(i, j);
}

private void shellsort()
{       int i, j, h;
        for (h = 1; h < n; h = 3*h + 1)
                ;
        for (;;) {
                h /= 3;
                if (h < 1) break;
                for (i = h; i < n; i++) {
                        for (j = i; j >= h; j -= h) {
                                if (a[j-h] < a[j]) break;
                                swap(j-h, j);
                        }
                }
        }
}

private void siftdown(int l, int u)
{       int i, c;
        i = l;
        for (;;) {
                c = 2*i;
                if (c > u)
                        break;
                if (c+1 <= u && a[c+1] > a[c])
                        c++;
                if (a[i] >= a[c])
```

```
                        break;
                swap(i, c);
                i = c;
        }
}

private void heapsort() // BEWARE!!! Sorts x[1..n-1]
{       int i;
        for (i = n/2; i > 0; i--)
                siftdown(i, n-1);
        for (i = n-1; i >= 2; i--) {
                swap(1, i);
                siftdown(1, i-1);
        }
}

private void qsort(int l, int u)
{       if (l >= u)
                return;
        int m = l;
        for (int i = l+1; i <= u; i++)
                if (a[i] < a[l])
                        swap(++m, i);
        swap(l, m);
        qsort(l, m-1);
        qsort(m+1, u);
}

void qsort2(int l, int u)
{       if (l >= u)
                return;
        int i = l;
        int j = u+1;
        for (;;) {
                do i++; while (i <= u && a[i] < a[l]);
                do j--; while (a[j] > a[l]);
                if (i > j)
                        break;
                swap(i, j);
        }
        swap(l, j);
        qsort2(l, j-1);
        qsort2(j+1, u);
}

// Drive Sort
static private final int ALG_ISORT = 0;
static private final int ALG_SELSORT = 1;
static private final int ALG_SHELLSORT = 2;
static private final int ALG_HSORT = 3;
static private final int ALG_QSORT = 4;
static private final int ALG_QSORT2 = 5;
static private int alg_num = ALG_ISORT;

private void dosort()
{       switch(alg_num) {
        case ALG_ISORT:     isort(); break;
        case ALG_SELSORT:   ssort(); break;
        case ALG_SHELLSORT: shellsort(); break;
        case ALG_HSORT:     heapsort(); break;
        case ALG_QSORT:     qsort(0, n-1); break;
```

```
                case ALG_QSORT2:    qsort2(0, n-1); break;
                }
        }

        private void runanim()
        {       n = Integer.parseInt(n_text.getText());
                if (n < 1 || n > MAXN) {
                        n = 50;
                        n_text.setText("" + n);
                }
                initdisplay();
                msg_label.setText("Running");
                genarray();
                for (int i = 0; i < n; i++)
                        draw(i, draw_color);
                Date timer = new Date();
                long start = timer.getTime();
                dosort();
                timer = new Date();
                long msecs = timer.getTime() - start;
                msg_label.setText("Msecs: " + msecs);
                if (! wantanim) // Draw results over input
                        for (int i = 0; i < n; i++)
                                draw(i, draw_color);
        }

// GUI FUNCTIONS
        public void init() {
                this.setBackground(back_color);

                // TextField for n (problem size)
                n_text = new TextField(5);
                this.add(new Label("n:"));
                this.add(n_text);
                n_text.setText("" + n);

                // Choice of Starting distributions
                dist_choices = new Choice();
                dist_choices.addItem("Random");
                dist_choices.addItem("Ascending");
                dist_choices.addItem("Descending");
                this.add(new Label("Input:"));
                this.add(dist_choices);

                // Choice of Sort Algorithms
                alg_choices = new Choice();
                alg_choices.addItem("Insertion Sort");
                alg_choices.addItem("Selection Sort");
                alg_choices.addItem("Shell Sort");
                alg_choices.addItem("Heap Sort");
                alg_choices.addItem("Quicksort");
                alg_choices.addItem("2-way Quicksort");
                this.add(new Label("Algorithm:"));
                this.add(alg_choices);

                // Run Button
                run_button = new Button("Run");
                this.add(run_button);

                // Message Label
                msg_label = new Label("                    ");
                this.add(msg_label);
```

```
        }

        public boolean action(Event event, Object arg) {
                if (event.target == dist_choices) {
                        if (arg.equals("Random")) gen_num = GEN_RAND;
                        else if (arg.equals("Ascending")) gen_num = GEN_ASCEND;
                        else if (arg.equals("Descending")) gen_num = GEN_DESCEND;
                        return true;
                } else if (event.target == alg_choices) {
                        if (arg.equals("Insertion Sort")) alg_num = ALG_ISORT;
                        else if (arg.equals("Selection Sort")) alg_num = ALG_SELSORT;
                        else if (arg.equals("Shell Sort")) alg_num = ALG_SHELLSORT;
                        else if (arg.equals("Heap Sort")) alg_num = ALG_HSORT;
                        else if (arg.equals("Quicksort")) alg_num = ALG_QSORT;
                        else if (arg.equals("2-way Quicksort")) alg_num = ALG_QSORT2;
                        return true;
                } else if (event.target == run_button) {
                        runanim();
                        return true;
                } else
                        return super.action(event, arg);
        }
}
```

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* sortedrand.cpp -- output m sorted random ints in U[0,n) */

#include <iostream>
#include <set>
#include <algorithm>
using namespace std;

int bigrand()
{       return RAND_MAX*rand() + rand();
}

int randint(int l, int u)
{       return l + bigrand() % (u-l+1);
}

void genknuth(int m, int n)
{       for (int i = 0; i < n; i++)
                /* select m of remaining n-i */
                if ((bigrand() % (n-i)) < m) {
                        cout << i << "\n";
                        m--;
                }
}

void gensets(int m, int n)
{       set<int> S;
        set<int>::iterator i;
        while (S.size() < m) {
                int t = bigrand() % n;
                S.insert(t);
        }
        for (i = S.begin(); i != S.end(); ++i)
                cout << *i << "\n";
}

void genshuf(int m, int n)
{       int i, j;
        int *x = new int[n];
        for (i = 0; i < n; i++)
                x[i] = i;
        for (i = 0; i < m; i++) {
                j = randint(i, n-1);
                int t = x[i]; x[i] = x[j]; x[j] = t;
        }
        sort(x, x+m);
        for (i = 0; i < m; i++)
                cout << x[i] << "\n";
}

void genfloyd(int m, int n)
{       set<int> S;
        set<int>::iterator i;
        for (int j = n-m; j < n; j++) {
                int t = bigrand() % (j+1);
                if (S.find(t) == S.end())
                        S.insert(t); // t not in S
                else
                        S.insert(j); // t in S
        }
```

```
        for (i = S.begin(); i != S.end(); ++i)
                cout << *i << "\n";
}

int main(int argc, char *argv[])
{       int m = atoi(argv[1]);
        int n = atoi(argv[2]);
        genknuth(m, n);
        return 0;
}
```

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* sets.cpp -- exercise set implementations on random numbers */

#include <iostream>
#include <set>
using namespace std;


class IntSetSTL {
private:
        set<int> S;
public:
        IntSetSTL(int maxelements, int maxval) { }
        int size() { return S.size(); }
        void insert(int t) { S.insert(t);}
        void report(int *v)
        {       int j = 0;
                set<int>::iterator i;
                for (i = S.begin(); i != S.end(); ++i)
                        v[j++] = *i;
        }
};


class IntSetBitVec {
private:
        enum { BITSPERWORD = 32, SHIFT = 5, MASK = 0x1F };
        int     n, hi, *x;
        void set(int i)  {        x[i>>SHIFT] |=  (1<<(i & MASK)); }
        void clr(int i)  {        x[i>>SHIFT] &= ~(1<<(i & MASK)); }
        int  test(int i) { return x[i>>SHIFT] &   (1<<(i & MASK)); }
public:
        IntSetBitVec(int maxelements, int maxval)
        {       hi = maxval;
                x = new int[1 + hi/BITSPERWORD];
                for (int i = 0; i < hi; i++)
                        clr(i);
                n = 0;
        }
        int size() { return n; }
        void insert(int t)
        {       if (test(t))
                        return;
                set(t);
                n++;
        }
        void report(int *v)
        {       int j=0;
                for (int i = 0; i < hi; i++)
                        if (test(i))
                                v[j++] = i;
        }
};


class IntSetArr {
private:
        int     n, *x;
public:
        IntSetArr(int maxelements, int maxval)
```

```
        {       x = new int[1 + maxelements];
                n=0;
                x[0] = maxval; /* sentinel at x[n] */
        }
        int size() { return n; }
        void insert(int t)
        {       int i, j;
                for (i = 0; x[i] < t; i++)
                        ;
                if (x[i] == t)
                        return;
                for (j = n; j >= i; j--)
                        x[j+1] = x[j];
                x[i] = t;
                n++;
        }
        void report(int *v)
        {       for (int i = 0; i < n; i++)
                        v[i] = x[i];
        }
};


class IntSetList {
private:
        int     n;
        struct node {
                int val;
                node *next;
                node(int i, node *p) { val = i; next = p; }
        };
        node *head, *sentinel;
        node *rinsert(node *p, int t)
        {       if (p->val < t) {
                        p->next = rinsert(p->next, t);
                } else if (p->val > t) {
                        p = new node(t, p);
                        n++;
                }
                return p;
        }
public:
        IntSetList(int maxelements, int maxval)
        {       sentinel = head = new node(maxval, 0);
                n = 0;
        }
        int size() { return n; }
        void insert(int t) { head = rinsert(head, t); }
        void insert2(int t)
        {       node *p;
                if (head->val == t)
                        return;
                if (head->val > t) {
                        head = new node(t, head);
                        n++;
                        return;
                }
                for (p = head; p->next->val < t; p = p->next)
                        ;
                if (p->next->val == t)
                        return;
```

```
                p->next = new node(t, p->next);
                n++;
        }
        void insert3(int t)
        {       node **p;
                for (p = &head; (*p)->val < t; p = &((*p)->next))
                        ;
                if ((*p)->val == t)
                        return;
                *p = new node(t, *p);
                n++;
        }
        void report(int *v)
        {       int j = 0;
                for (node *p = head; p != sentinel; p = p->next)
                        v[j++] = p->val;
        }
};

// Change from new per node to one new at init
// Factor of 2.5 on VC 5.0, 6% on SGI CC
class IntSetList2 {
private:
        int     n;
        struct node {
                int val;
                node *next;
        };
        node *head, *sentinel, *freenode;
public:
        IntSetList2(int maxelements, int maxval)
        {       sentinel = head = new node;
                sentinel->val = maxval;
                freenode = new node[maxelements];
                n = 0;
        }
        int size() { return n; }
        void insert(int t)
        {       node **p;
                for (p = &head; (*p)->val < t; p = &((*p)->next))
                        ;
                if ((*p)->val == t)
                        return;
                freenode->val = t;
                freenode->next = *p;
                *p = freenode++;
                n++;
        }
        void report(int *v)
        {       int j = 0;
                for (node *p = head; p != sentinel; p = p->next)
                        v[j++] = p->val;
        }
};


class IntSetBST {
private:
        int     n, *v, vn;
        struct node {
                int val;
```

```
                node *left, *right;
                node(int v) { val = v; left = right = 0; }
        };
        node *root;
        node *rinsert(node *p, int t)
        {       if (p == 0) {
                        p = new node(t);
                        n++;
                } else if (t < p->val) {
                        p->left = rinsert(p->left, t);
                } else if (t > p->val) {
                        p->right = rinsert(p->right, t);
                } // do nothing if p->val == t
                return p;
        }
        void traverse(node *p)
        {       if (p == 0)
                        return;
                traverse(p->left);
                v[vn++] = p->val;
                traverse(p->right);
        }
public:
        IntSetBST(int maxelements, int maxval) { root = 0; n = 0; }
        int size() { return n; }
        void insert(int t) { root = rinsert(root, t); }
        void report(int *x) { v = x; vn = 0; traverse(root); }
};


class IntSetBST2 {
private:
        int     n, *v, vn;
        struct node {
                int val;
                node *left, *right;
        };
        node *root, *freenode, *sentinel;
        node *rinsert(node *p, int t)
        {       if (p == sentinel) {
                        p = freenode++;
                        p->val = t;
                        p->left = p->right = sentinel;
                        n++;
                } else if (t < p->val) {
                        p->left = rinsert(p->left, t);
                } else if (t > p->val) {
                        p->right = rinsert(p->right, t);
                } // do nothing if p->val == t
                return p;
        }
        void traverse(node *p)
        {       if (p == sentinel)
                        return;
                traverse(p->left);
                v[vn++] = p->val;
                traverse(p->right);
        }
public:
        IntSetBST2(int maxelements, int maxval)
        {       root = sentinel = new node;  // 0 if using insert1
```

```
                n = 0;
                freenode = new node[maxelements];
        }
        int size() { return n; }
        void insert1(int t) { root = rinsert(root, t); }
        void insert(int t)
        {       sentinel->val = t;
                node **p = &root;
                while ((*p)->val != t)
                        if (t < (*p)->val)
                                p = &((*p)->left);
                        else
                                p = &((*p)->right);
                if (*p == sentinel) {
                        *p = freenode++;
                        (*p)->val = t;
                        (*p)->left = (*p)->right = sentinel;
                        n++;
                }
        }
        void report(int *x) { v = x; vn = 0; traverse(root); }
};


class IntSetBins {
private:
        int     n, bins, maxval;
        struct node {
                int val;
                node *next;
                node(int v, node *p) { val = v; next = p; }
        };
        node **bin, *sentinel;
        node *rinsert(node *p, int t)
        {       if (p->val < t) {
                        p->next = rinsert(p->next, t);
                } else if (p->val > t) {
                        p = new node(t, p);
                        n++;
                }
                return p;
        }
public:
        IntSetBins(int maxelements, int pmaxval)
        {       bins = maxelements;
                maxval = pmaxval;
                bin = new node*[bins];
                sentinel = new node(maxval, 0);
                for (int i = 0; i < bins; i++)
                        bin[i] = sentinel;
                n = 0;
        }
        int size() { return n; }
        void insert(int t)
        {       int i = t / (1 + maxval/bins);  // CHECK !
                bin[i] = rinsert(bin[i], t);
        }
        void report(int *v)
        {       int j = 0;
                for (int i = 0; i < bins; i++)
                        for (node *p = bin[i]; p != sentinel; p = p->next)
```

```
                                v[j++] = p->val;
        }
};


class IntSetBins2 {
private:
        int     n, bins, maxval;
        struct node {
                int val;
                node *next;
        };
        node **bin, *sentinel, *freenode;
        node *rinsert(node *p, int t)
        {       if (p->val < t) {
                        p->next = rinsert(p->next, t);
                } else if (p->val > t) {
                        freenode->val = t;
                        freenode->next = p;
                        p = freenode++;
                        n++;
                }
                return p;
        }
public:
        IntSetBins2(int maxelements, int pmaxval)
        {       bins = maxelements;
                maxval = pmaxval;
                freenode = new node[maxelements];
                bin = new node*[bins];
                sentinel = new node;
                sentinel->val = maxval;
                for (int i = 0; i < bins; i++)
                        bin[i] = sentinel;
                n = 0;
        }
        int size() { return n; }
        void insert1(int t)
        {       int i = t / (1 + maxval/bins);
                bin[i] = rinsert(bin[i], t);
        }
        void insert(int t)
        {       node **p;
                int i = t / (1 + maxval/bins);
                for (p = &bin[i]; (*p)->val < t; p = &((*p)->next))
                        ;
                if ((*p)->val == t)
                        return;
                freenode->val = t;
                freenode->next = *p;
                *p = freenode++;
                n++;
        }
        void report(int *v)
        {       int j = 0;
                for (int i = 0; i < bins; i++)
                        for (node *p = bin[i]; p != sentinel; p = p->next)
                                v[j++] = p->val;
        }
};
```

```
// Drivers for the set data structures

int bigrand()
{       return RAND_MAX*rand() + rand();
}

int randint(int l, int u)
{       return l + bigrand() % (u-l+1);
}

void gensets(int m, int maxval)
{       int *v = new int[m];
        IntSetList S(m, maxval);
        while (S.size() < m)
                S.insert(bigrand() % maxval);
        S.report(v);
//      for (int i = 0; i < m; i++)
        for (int i = 0; i < 2; i++)
                cout << v[i] << "\n";
}

void genfloyd(int m, int maxval)
{       int *v = new int[m];
        IntSetSTL S(m, maxval);
        for (int j = maxval-m; j < maxval; j++) {
                int t = bigrand() % (j+1);
                int oldsize = S.size();
                S.insert(t);
                if (S.size() == oldsize) // t already in S
                        S.insert(j);
        }
        S.report(v);
        for (int i = 0; i < m; i++)
                cout << v[i] << "\n";
}

void memaccesstest(int m, int n)
{       IntSetList S(m, n);     // change among Arr, List and List2
        for (int i = 0; i < m; i++)
                S.insert(i);
}

void overheadonly(int m, int n)
{       int i, *v = new int[m];
        for (i = 0; i < m; i++)
                v[i] = bigrand() % n;
        for (i = 0; i < m; i++)
                cout << v[i] << "\n";
}

int main(int argc, char *argv[])
{       int m = atoi(argv[1]);
        int maxval = atoi(argv[2]);
        gensets(m, maxval);
        // overheadonly(m, n);
        // memaccesstest(m, n);
        return 0;
}
```

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* priqueue.cpp -- priority queues (using heaps) */

#include <iostream>
using namespace std;

// define and implement priority queues

template<class T>
class priqueue {
private:
        int     n, maxsize;
        T       *x;
        void swap(int i, int j)
        {       T t = x[i]; x[i] = x[j]; x[j] = t; }
public:
        priqueue(int m)
        {       maxsize = m;
                x = new T[maxsize+1];
                n = 0;
        }
        void insert(T t)
        {       int i, p;
                x[++n] = t;
                for (i = n; i > 1 && x[p=i/2] > x[i]; i = p)
                        swap(p, i);
        }
        T extractmin()
        {       int i, c;
                T t = x[1];
                x[1] = x[n--];
                for (i = 1; (c=2*i) <= n; i = c) {
                        if (c+1<=n && x[c+1]<x[c])
                                c++;
                        if (x[i] <= x[c])
                                break;
                        swap(c, i);
                }
                return t;
        }
};

// sort with priority queues (heap sort is strictly better)

template<class T>
void pqsort(T v[], int n)
{       priqueue<T> pq(n);
        int i;
        for (i = 0; i < n; i++)
                pq.insert(v[i]);
        for (i = 0; i < n; i++)
                v[i] = pq.extractmin();
}

// main

int main()
{       const int       n = 10;
        int     i, v[n];
        if (0) { // Generate and sort
```

```
                for (i = 0; i < n; i++)
                        v[i] = n-i;
                pqsort(v, n);
                for (i = 0; i < n; i++)
                        cout << v[i] << "\n";
        } else { // Insert integers; extract with 0
                priqueue<int> pq(100);
                while (cin >> i)
                        if (i == 0)
                                cout << pq.extractmin() << "\n";
                        else
                                pq.insert(i);
        }
        return 0;
}
```

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* wordlist.cpp -- Sorted list of words (between white space) in file */

#include <iostream>
#include <set>
#include <string>
using namespace std;

int main()
{       set<string> S;
        string t;
        set<string>::iterator j;
        while (cin >> t)
                S.insert(t);
        for (j = S.begin(); j != S.end(); ++j)
                cout << *j << "\n";
        return 0;
}
```

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* wordfreq.cpp -- List all words in input file, with counts */

#include <iostream>
#include <map>
#include <string>
using namespace std;

int main()
{       map<string, int> M;
        map<string, int>::iterator j;
        string t;
        while (cin >> t)
                M[t]++;
        for (j = M.begin(); j != M.end(); ++j)
                cout << j->first << " " << j->second << "\n";
        return 0;
}
```

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* wordfreq.c -- list of words in file, with counts */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct node *nodeptr;
typedef struct node {
        char *word;
        int count;
        nodeptr next;
} node;

#define NHASH 29989
#define MULT 31
nodeptr bin[NHASH];

unsigned int hash(char *p)
{       unsigned int h = 0;
        for ( ; *p; p++)
                h = MULT * h + *p;
        return h % NHASH;
}

#define NODEGROUP 1000
int nodesleft = 0;
nodeptr freenode;

nodeptr nmalloc()
{       if (nodesleft == 0) {
                freenode = malloc(NODEGROUP*sizeof(node));
                nodesleft = NODEGROUP;
        }
        nodesleft--;
        return freenode++;
}

#define CHARGROUP 10000
int charsleft = 0;
char *freechar;

char *smalloc(int n)
{       if (charsleft < n) {
                freechar = malloc(n+CHARGROUP);
                charsleft = n+CHARGROUP;
        }
        charsleft -= n;
        freechar += n;
        return freechar - n;
}

void incword(char *s)
{       nodeptr p;
        int h = hash(s);
        for (p = bin[h]; p != NULL; p = p->next)
                if (strcmp(s, p->word) == 0) {
                        (p->count)++;
                        return;
                }
```

```
        p = nmalloc();
        p->count = 1;
        p->word = smalloc(strlen(s)+1);
        strcpy(p->word, s);
        p->next = bin[h];
        bin[h] = p;
}

int main()
{       int i;
        nodeptr p;
        char buf[100];
        for (i = 0; i < NHASH; i++)
                bin[i] = NULL;
        while (scanf("%s", buf) != EOF)
                incword(buf);
        for (i = 0; i < NHASH; i++)
                for (p = bin[i]; p != NULL; p = p->next)
                        printf("%s %d\n", p->word, p->count);
        return 0;
}
```

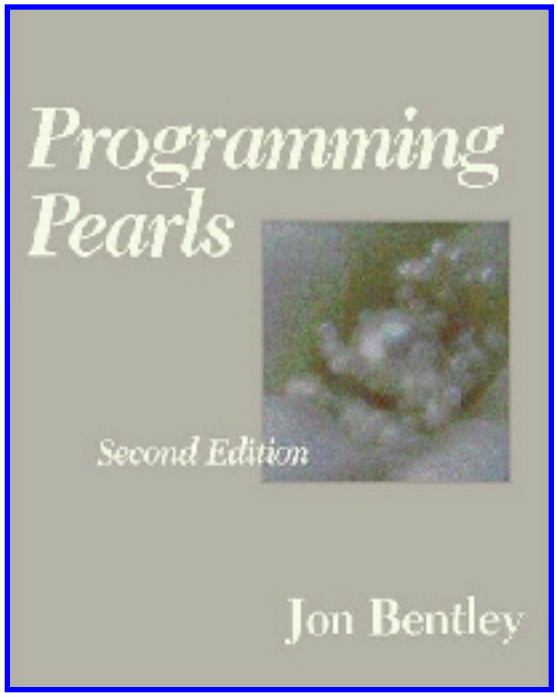
```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* spacemod.cpp -- simple model for C++ space */

#include <iostream>
using namespace std;

#define MEASURE(T, text) {                                      \
        cout << text << "\t";                                   \
        cout << sizeof(T) << "\t";                              \
        int lastp = 0;                                                  \
        for (int i = 0; i < 11; i++) {                  \
                T *p = new T;                                           \
                int thisp = (int) p;                            \
                if (lastp != 0)                                         \
                        cout << " " << thisp - lastp;   \
                lastp = thisp;                                          \
        }                                                                               \
        cout << "\n";                                                   \
}

// Must use macros; templates give funny answers

template <class T>
void measure(char *text)
{       cout << " measure: " << text << "\t";
        cout << sizeof(T) << "\n";
}

struct structc { char c; };
struct structic { int i; char c; };
struct structip { int i; structip *p; };
struct structdc { double d; char c; };
struct structcd { char c; double d; };
struct structcdc { char c1; double d; char c2; };
struct structiii { int i1; int i2; int i3; };
struct structiic { int i1; int i2; char c; };
struct structc12 { char c[12]; };
struct structc13 { char c[13]; };
struct structc28 { char c[28]; };
struct structc29 { char c[29]; };
struct structc44 { char c[44]; };
struct structc45 { char c[45]; };
struct structc60 { char c[60]; };
struct structc61 { char c[61]; };

int main()
{       cout << "Raw sizeof";
        cout << "\nsizeof(char)="     << sizeof(char);
        cout << "  sizeof(short)="    << sizeof(short);
        cout << "  sizeof(int)="      << sizeof(int);
        cout << "\nsizeof(float)="    << sizeof(float);
        cout << "  sizeof(struct *)=" << sizeof(structc *);
        cout << "  sizeof(long)="     << sizeof(long);
        cout << "\nsizeof(double)="   << sizeof(double);

        cout << "\n\nMEASURE macro\n";
        MEASURE(int, "int");
        MEASURE(structc, "structc");
        MEASURE(structic, "structic");
        MEASURE(structip, "structip");
```

```
        MEASURE(structdc, "structdc");
        MEASURE(structcd, "structcd");
        MEASURE(structcdc, "structcdc");
        MEASURE(structiii, "structiii");
        MEASURE(structiic, "structiic");
        MEASURE(structc12, "structc12");
        MEASURE(structc13, "structc13");
        MEASURE(structc28, "structc28");
        MEASURE(structc29, "structc29");
        MEASURE(structc44, "structc44");
        MEASURE(structc45, "structc45");
        MEASURE(structc60, "structc60");
        MEASURE(structc61, "structc61");


        cout << "\nmeasure template (strange results)\n";
        // Uncomment below lines to see answers change
        measure<int>("int");
//      measure<structc>("structc");
//      measure<structic>("structic");
        return 0;
}
```

```
/* Copyright (C) 1999 Lucent Technologies */
/* From 'Programming Pearls' by Jon Bentley */

/* timemod.c -- Produce table of C run time costs */

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <math.h>

#define MAXN 100000
int x[MAXN];
int startn = 5000;
int n;

/* FUNCTIONS TO BE TIMED */

int intcmp(int *i, int *j)
{   return *i - *j; }

#define swapmac(i, j) { t = x[i]; x[i] = x[j]; x[j] = t; }

void swapfunc(int i, int j)
 {  int t = x[i];
        x[i] = x[j];
        x[j] = t;
 }

#define maxmac(a, b) ((a) > (b) ? (a) : (b))

int maxfunc(int a, int b)
{       return a > b ? a : b; }


/* WORKHORSE */

#define T(s) printf("%s (n=%d)\n", s, n);
#define TRIALS 5
#define M(op)                                                   \
        printf(" %-22s", #op);                          \
        k = 0;                                                                  \
        timesum = 0;                                            \
        for (ex = 0; ex < TRIALS; ex++) {       \
                start = clock();                                \
                for (i = 1; i <= n; i++) {              \
                        fi = (float) i;                         \
                        for (j = 1; j <= n; j++) {      \
                                op;                                             \
                        }                                                       \
                }                                                       \
                t = clock()-start;                              \
                printf("%6d", t);                               \
                timesum += t;                                   \
        }                                                               \
        nans = 1e9 * timesum / ((double)        \
                n*n * TRIALS * CLOCKS_PER_SEC); \
        printf("%8.0f\n", nans);


int main()
{   int i, j, k;
        float fi, fj, fk;
        int t, ex, timesum, start, globalstart;
```

```
        double nans;
        globalstart = clock();
        for (i = 0; i < n; i++)
                x[i] = rand();
        n = startn;
        printf("C Time Cost Model, n=%d\n", n);
        T("Integer Arithmetic");
        M({});
        M(k++);
        M(k = i + j);
        M(k = i - j);
        M(k = i * j);
        M(k = i / j);
        M(k = i % j);
        M(k = i & j);
        M(k = i | j);
        T("Floating Point Arithmetic");
        M(fj=j;);
        M(fj=j; fk = fi + fj;);
        M(fj=j; fk = fi - fj;);
        M(fj=j; fk = fi * fj;);
        M(fj=j; fk = fi / fj;);
        T("Array Operations");
        M(k = i + j);
        M(k = x[i] + j);
        M(k = i + x[j]);
        M(k = x[i] + x[j]);
        T("Comparisons");
        M(if (i < j) k++);
        M(if (x[i] < x[j]) k++);
        T("Array Comparisons and Swaps");
        M(k = (x[i]<x[k]) ? -1:1);
        M(k = intcmp(x+i, x+j));
        M(swapmac(i, j));
        M(swapfunc(i, j));
        T("Max Function, Macro and Inline");
        M(k = (i > j) ? i : j);
        M(k = maxmac(i, j));
        M(k = maxfunc(i, j));
        n = startn / 5;
        T("Math Functions");
        M(fk = j+fi;);
        M(k = rand(););
        M(fk = sqrt(j+fi));
        M(fk = sin(j+fi));
        M(fk = sinh(j+fi));
        M(fk = asin(j+fi));
        M(fk = cos(j+fi));
        M(fk = tan(j+fi));
        n = startn / 10;
        T("Memory Allocation");
        M(free(malloc(16)););
        M(free(malloc(100)););
        M(free(malloc(2000)););

        /* Possible additions: input, output, malloc */
        printf("  Secs: %4.2f\n",
                ((double) clock()-globalstart)
                / CLOCKS_PER_SEC);
        return 0;
}
```

Rules for Code Tuning (Appendix 4 of Programming Pearls)



My 1982 book *Writing Efficient Programs* was built around 27 rules for code tuning. That book is now out of print, so the rules are repeated here (with only a few small changes), together with examples of how they are used in this book.

Space-For-Time Rules

Data Structure Augmentation. The time required for common operations on data can often be reduced by augmenting the structure with extra information or by changing the information within the structure so that it can be accessed more easily.

> In Section 9.2, Wright wanted to find nearest neighbors (by angle) among a set of points on the surface of the globe represented by latitude and longitude, which involved expensive trigonometric operations. Appel augmented her data structure with the *x*, *y* and *z* coordinates, which allowed her to use Euclidean distances with much less compute time.

Store Precomputed Results. The cost of recomputing an expensive function can be reduced by computing the function only once and storing the results. Subsequent requests for the function are then handled by table lookup rather than by computing the function.

> The cumulative arrays in Section 8.2 and Solution 8.11 replace a sequence of additions with two table lookups and a subtraction.
>
> Solution 9.7 speeds up a program to count bits by looking up a byte or a word at a time.
>
> Solution 10.6 replaces shifting and logical operations with a table lookup.

Caching. Data that is accessed most often should be the cheapest to access.

> Section 9.1 describes how Van Wyk cached the most common size of node to avoid expensive calls to the system storage allocator. Solution 9.2 gives the details on one kind of node cache.
>
> Column 13 caches nodes for lists, bins and binary search trees.
>
> Caching can backfire and increase the run time of a program if locality is not present in the underlying data.

Lazy Evaluation. The strategy of never evaluating an item until it is needed avoids evaluations of unnecessary items.

Time-For-Space Rules

Packing. Dense storage representations can decrease storage costs by increasing the time required to store and retrieve data.

> The sparse array representations in Section 10.2 greatly reduce storage costs by slightly increasing the time to access the structures.
>
> McIlroy's dictionary for a spelling checker in Section 13.8 squeezes 75,000 English words into 52 kilobytes.
>
> Kernighan's arrays in Section 10.3 and the Heapsort in Section 14.4 both use *overlaying* to reduce data space by storing data items that are never simultaneously active in the same memory space.
>
> Although packing sometimes trades time for space, the smaller representations are often also faster to process.

Interpreters. The space required to represent a program can often be decreased by the use of interpreters in which common sequences of operations are represented compactly.

> Section 3.2 uses an interpreter for ``form-letter programming" and Section 10.4 uses an interpreter for a simple graphics program.

Loop Rules

Code Motion Out of Loops. Instead of performing a certain computation in each iteration of a loop, it is better to perform it only once, outside the loop.

> Section 11.1 moves an assignment to the variable *t* out of the main loop of *isort2*.

Combining Tests. An efficient inner loop should contain as few tests as possible, and preferably only one. The programmer should therefore try to simulate some of the exit conditions of the loop by other exit conditions.

> *Sentinels* are a common application of this rule: we place a sentinel at the boundary of a data structure to reduce the cost of testing whether our search has exhausted the structure. Section 9.2 uses a sentinel in sequentially searching an array. Column 13 uses sentinels to yield clean (and incidentally efficient) code for arrays, linked lists, bins and binary search trees. Solution 14.1 places a sentinel at one end of a heap.

Loop Unrolling. Unrolling a loop can remove the cost of modifying loop indices, and also help to avoid pipeline stalls, to reduce branches, and to increase instruction-level parallelism.

> Unrolling a sequential search in Section 9.2 reduces its run time by about 50 percent, and unrolling a binary search in Section 9.3 reduces its run time by between 35 and 65 percent.

Transfer-Driven Loop Unrolling. If a large cost of an inner loop is devoted to trivial assignments, then those

assignments can often be removed by repeating the code and changing the use of variables. Specifically, to remove the assignment $i = j$, the subsequent code must treat j as though it were i.

Unconditional Branch Removal. A fast loop should contain no unconditional branches. An unconditional branch at the end of a loop can be removed by ``rotating" the loop to have a conditional branch at the bottom.

> This operation is usually done by optimizing compilers.

Loop Fusion. If two nearby loops operate on the same set of elements, then combine their operational parts and use only one set of loop control operations.

Logic Rules

Exploit Algebraic Identities. If the evaluation of a logical expression is costly, replace it by an algebraically equivalent expression that is cheaper to evaluate.

Short-Circuiting Monotone Functions. If we wish to test whether some monotone nondecreasing function of several variables is over a certain threshold, then we need not evaluate any of the variables once the threshold has been reached.

> A more sophisticated application of this rule exits from a loop as soon as the purpose of the loop has been accomplished. The search loops in Columns 10, 13 and 15 all terminate once they find the desired element.

Reordering Tests. Logical tests should be arranged such that inexpensive and often successful tests precede expensive and rarely successful tests.

> Solution 9.6 sketches a series of tests that might be reordered.

Precompute Logical Functions. A logical function over a small finite domain can be replaced by a lookup in a table that represents the domain.

> Solution 9.6 describes how the Standard C library character classification functions can be implemented by table lookup.

Boolean Variable Elimination. We can remove boolean variables from a program by replacing the assignment to a boolean variable v by an *if-else* statement in which one branch represents the case that v is true and the other represents the case that v is false.

Procedure Rules

Collapsing Function Hierarchies. The run times of the elements of a set of functions that (nonrecursively) call themselves can often be reduced by rewriting functions in line and binding the passed variables.

> Replacing the *max* function in Section 9.2 with a macro gives a speedup of almost a factor of two.

Writing the *swap* function inline in Section 11.1 gave a speedup of a factor of almost 3; writing the *swap* inline in Section 11.3 gave less of a speedup.

Exploit Common Cases. Functions should be organized to handle all cases correctly and common cases efficiently.

In Section 9.1, Van Wyk's storage allocator handled all node sizes correctly, but handled the most common node size particularly efficiently.

In Section 6.1, Appel treated the expensive case of near objects with a special-purpose small time step, which allowed the rest of his program to use a more efficient large time step.

Coroutines. A multiple-pass algorithm can often be turned into a single-pass algorithm by use of coroutines.

The anagram program in Section 2.8 uses a pipeline, which can be implemented as a set of coroutines.

Transformations on Recursive Functions. The run time of recursive functions can often be reduced by applying the following transformations:

Rewrite recursion to iteration, as in the lists and binary search trees in Column 13.

Convert the recursion to iteration by using an explicit program stack. (If a function contains only one recursive call to itself, then it is not necessary to store the return address on the stack.)

If the final action of a function is to call itself recursively, replace that call by a branch to its first statement; this is usually known as removing tail recursion. The code in Solution 11.9 is a candidate for this transformation. That branch can often be transformed into a loop. Compilers often perform this optimization.

It is often more efficient to solve small subproblems by use of an auxiliary procedure, rather than by recurring down to problems of size zero or one. The *qsort4* function in Section 11.3 uses a value of *cutoff* near 50.

Parallelism. A program should be structured to exploit as much of the parallelism as possible in the underlying hardware.

Expression Rules

Compile-Time Initialization. As many variables as possible should be initialized before program execution.

Exploit Algebraic Identities. If the evaluation of an expression is costly, replace it by an algebraically equivalent expression that is cheaper to evaluate.

In Section 9.2, Appel replaced expensive trigonometric operations with multiplications and additions, and also used monotonicity to remove an expensive square root operation.

Section 9.2 replaces an expensive C remainder operator % in an inner loop with a cheaper *if* statement.

We can often multiply or divide by powers of two by shifting left or right. Solution 13.9 replaces an arbitrary division used by bins with a shift. Solution 10.6 replaces a division by 10 with a shift by 4.

In Section 6.1, Appel exploited additional numeric accuracy in a data structure to replace 64-bit floating point numbers with faster 32-bit numbers.

Strength reduction on a loop that iterates through the elements of an array replaces a multiplication by an addition. Many compilers perform this optimization. This technique generalizes to a large class of incremental algorithms.

Common Subexpression Elimination. If the same expression is evaluated twice with none of its variables altered between evaluations, then the second evaluation can be avoided by storing the result of the first and using that in place of the second.

Modern compilers are usually good at eliminating common subexpressions that do not contain function calls.

Pairing Computation. If two similar expressions are frequently evaluated together, then we should make a new procedure that evaluates them as a pair.

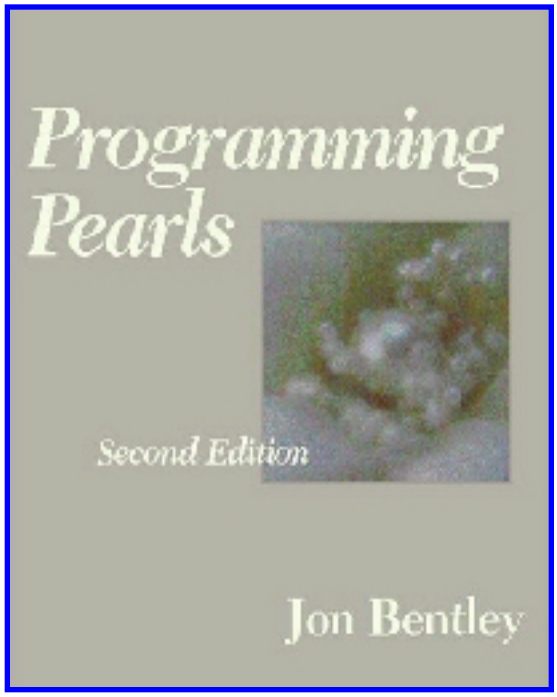
In Section 13.1, our first pseudocode always uses *member* and *insert* functions in concert; the C++ code replaces those two functions with an *insert* that does nothing if its argument is already in the set.

Exploit Word Parallelism. Use the full data-path width of the underlying computer architecture to evaluate expensive expressions.

Problem 13.8 shows how bit vectors can operate on many bits at once by operating on *char*s or *int*s.

Solution 9.7 counts bits in parallel.

Errata for Programming Pearls, Second Edition



The second printing had no changes from the first printing.

The following bugs need to be fixed in upcoming printings.

p. 38, line 23. The clause *cantbe(m, n)* should be *cantbe(m, n-1)*.

p. 121, line 11: The inequality sign is reversed; the test should read ``*if u-l < cutoff*". (The test is correct in the function *qsort4* later on the page.)

p. 129, line -14: The variables ``*m*" and ``*n*" are swapped. The sentence should read: ``Next, suppose that *m* is ten million and *n* is 2^{31}."

p. 203, line 12: Change ``commutative" to ``associative".

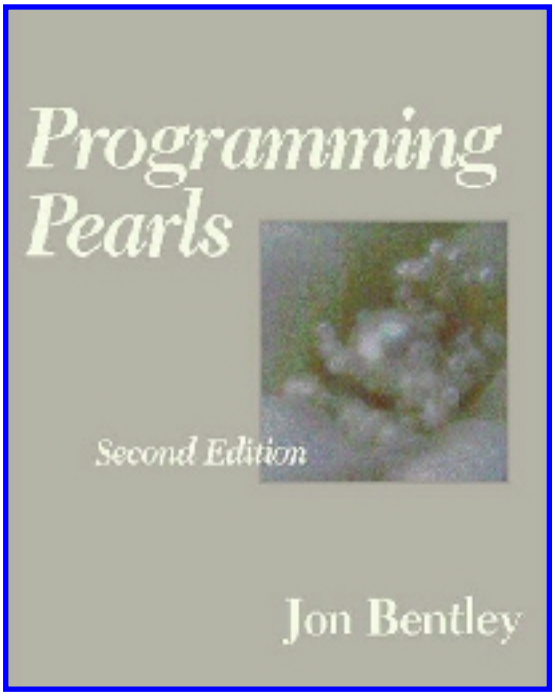
p. 224, line -2: Remove the phrase ``or print *n* + 1 - *i* rather than *i*".

p. 241, line -13: Insert ``p" after ``*", so the line reads

```
unsigned int hash(char *p)
```

I am grateful for the extremely careful reading and detailed bug reports provided by Alexey Kudravtsev, Arnold Robbins and Satoru Takabayashi.

Cracking the Oyster (Column 1 of Programming Pearls)



The programmer's question was simple: ``How do I sort a disk file?'' Before I tell you how I made my first mistake, let me give you a chance to do better than I did. What would you have said?

1.1 A Friendly Conversation

My mistake was to answer his question. I gave him a thumbnail sketch of how to implement a Merge Sort on disk. My suggestion that he dig into an algorithms text met with less than enthusiasm -- he was more concerned about solving the problem than furthering his education. I then told him about a disk sorting program in a popular programming book. The program consisted of about two hundred lines of code in a dozen functions; I estimated that implementing and testing the code would have taken the programmer at most a week.

I thought that I had solved his problem, but his hesitation led me back to the right track. The conversation then went something like this, with my questions in *italics*.

> *Why do you want to write your own sort at all? Why not use a sort provided by your system?*
>
> I need the sort in the middle of a large system, and for obscure technical reasons, I can't use the system file-sorting program.
>
> *What exactly are you sorting? How many records are in the file? What is the format of each record?*
>
> The file contains at most ten million records; each record is a seven-digit integer.
>
> *Wait a minute. If the file is that small, why bother going to disk at all? Why not just sort it in main memory?*
>
> Although the machine has many megabytes of main memory, this function is part of a big system. I expect that I'll have only about a megabyte free at that point.
>
> *Is there anything else you can tell me about the records?*
>
> Each one is a seven-digit positive integer with no other associated data, and no integer can appear more than once.

The context makes the problem clearer. In the United States, telephone numbers consist of a three-digit ``area code'' followed by seven additional digits. Telephone calls to numbers with the ``toll-free'' area code of 800 (the only such code at the time) were not charged. A real database of toll-free telephone numbers includes a great deal of information: the toll-free telephone number, the real number to which calls are routed (sometimes several numbers, with rules on which calls go where when), the name and address of the subscriber, and so on.

The programmer was building a small corner of a system for processing such a database, and the integers to be sorted were toll-free telephone numbers. The input file was a list of numbers (with all other information removed), and it was an error to include the same number twice. The desired output was a file of the numbers, sorted in increasing numeric order. The context also defines the performance requirements. During a long session with the system, the user requested a sorted file roughly once an hour and could do nothing until the sort was completed. The sort therefore couldn't take more than a few minutes, while ten seconds was a more desirable run time.

The Rest of the Column

These are the remaining sections in the column.

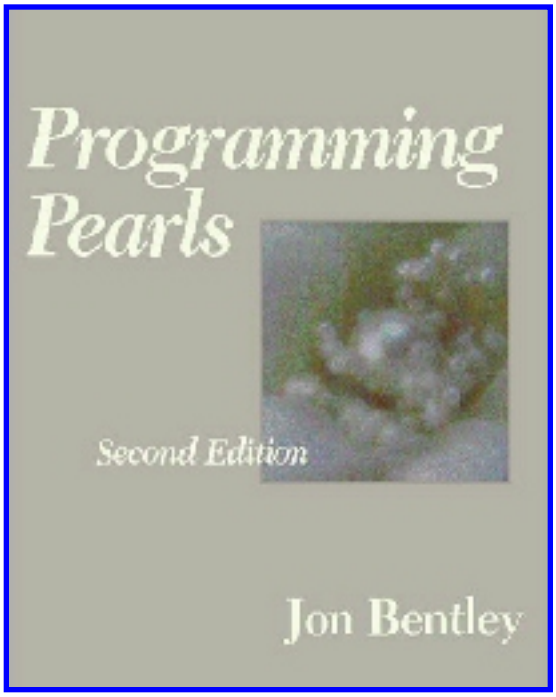
The Solutions to Column 1 give answers for some of the Problems.

Several of the programs in the column and the solutions can be found with the other source code for this book.

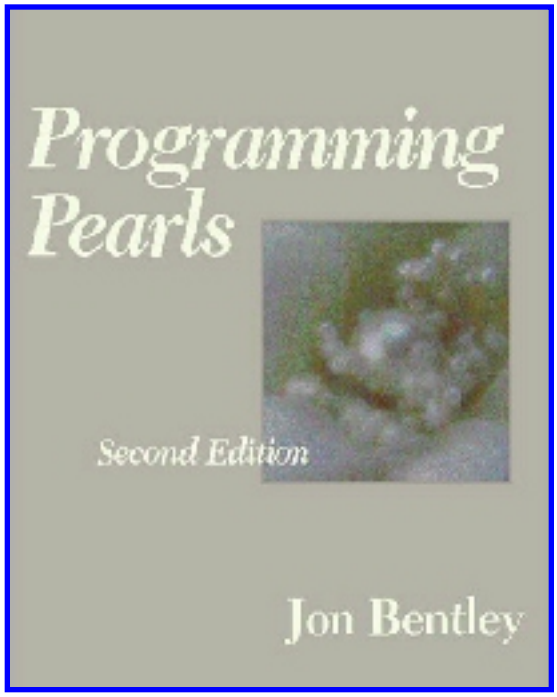
The web references describe web sites related to the topic.

An excerpt from this column appears in the November 1999 issue of *Dr. Dobb's Journal*.

Index to Programming Pearls



A Small Matter of Programming (A Sketch of Column 5 of Programming Pearls)



So far, you've done everything right. You dug deep to define the right problem. You balanced the true requirements with a careful selection of algorithms and data structures. You used the techniques of program verification to write elegant pseudocode that you're pretty sure is correct. How do you incorporate the resulting gem into your big system? All that's left is, well, a small matter of programming.

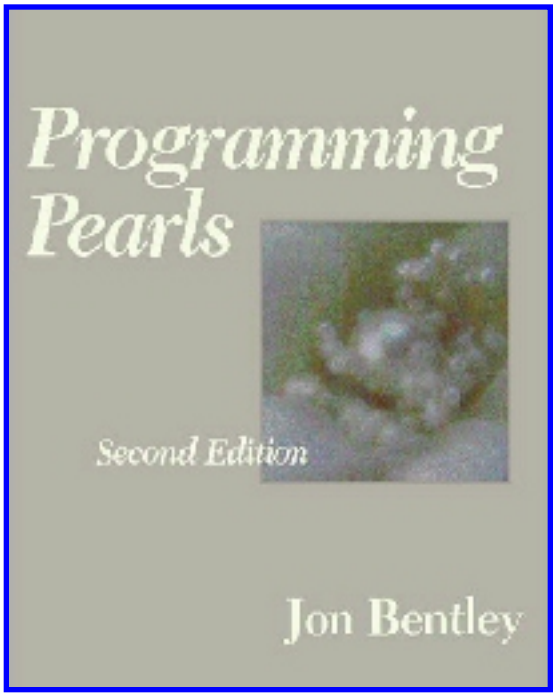
Programmers are optimists, and they are tempted to take the easy path: code up the function, slip it into the system, and fervently hope that it runs. Sometimes that works. But the other 999 times out of 1000, it leads to disaster: one has to grope around in the huge system to manipulate the little function.

Wise programmers instead build *scaffolding* to give them easy access to the function. This column is devoted to implementing the binary search described in pseudocode in the last column as a trustworthy C function. (The code would be very similar in C++ or Java, and the approach works for most other languages.) Once we have the code, we'll probe it with scaffolding, then move on to test it more thoroughly and conduct experiments on its running time. For a tiny function, our path is long. The result, though, is a program we can trust.

The Rest of the Column

The code for Column 5 contains code for testing and timing search functions.

Debugging (Section 5.10 of Programming Pearls)



Every programmer knows that debugging is hard. Great debuggers, though, can make the job look simple. Distraught programmers describe a bug that they've been chasing for hours, the master asks a few questions, and minutes later the programmers are staring at the faulty code. The expert debugger never forgets that there has to be a logical explanation, no matter how mysterious the system's behavior may seem when first observed.

That attitude is illustrated in an anecdote from IBM's Yorktown Heights Research Center. A programmer had recently installed a new workstation. All was fine when he was sitting down, but he couldn't log in to the system when he was standing up. That behavior was one hundred percent repeatable: he could always log in when sitting and never when standing.

Most of us just sit back and marvel at such a story. How could that workstation know whether the poor guy was sitting or standing? Good debuggers, though, know that there has to be a reason. Electrical theories are the easiest to hypothesize. Was there a loose wire under the carpet, or problems with static electricity? But electrical problems are rarely one-hundred-percent consistent. An alert colleague finally asked the right question: how did the programmer log in when he was sitting and when he was standing? Hold your hands out and try it yourself.

The problem was in the keyboard: the tops of two keys were switched. When the programmer was seated he was a touch typist and the problem went unnoticed, but when he stood he was led astray by hunting and pecking. With this hint and a convenient screwdriver, the expert debugger swapped the two wandering keytops and all was well.

A banking system built in Chicago had worked correctly for many months, but unexpectedly quit the first time it was used on international data. Programmers spent days scouring the code, but they couldn't find any stray command that would quit the program. When they observed the behavior more closely, they found that the program quit as they entered data for the country of Ecuador. Closer inspection showed that when the user typed the name of the capital city (Quito), the program interpreted that as a request to quit the run!

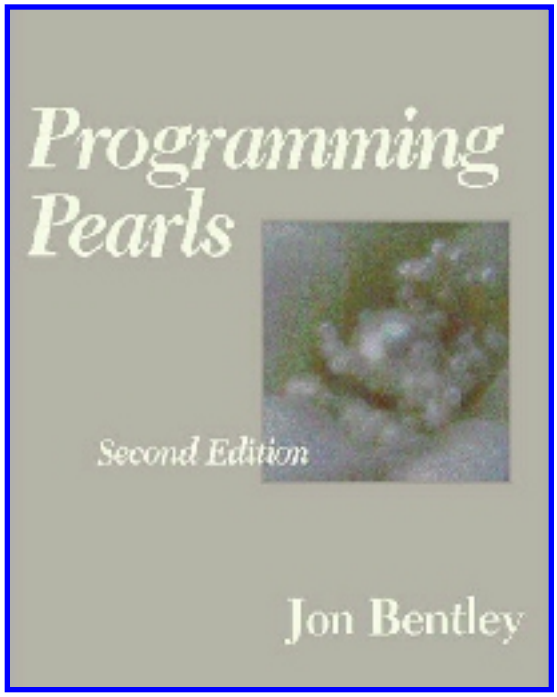
Bob Martin once watched a system ``work once twice". It handled the first transaction correctly, then exhibited a minor flaw in all later transactions. When the system was rebooted, it once again correctly processed the first transaction, and failed on all subsequent transactions. When Martin characterized the behavior as having ``worked once twice", the developers immediately knew to look for a variable that was initialized correctly when the program was loaded, but was not reset properly after the first transaction.

In all cases the right questions guided wise programmers to nasty bugs in short order: ``What do you do differently sitting and standing? May I watch you logging in each way?" ``Precisely what did you type before the program quit?" ``Did the program ever work correctly before it started failing? How many times?"

Rick Lemons said that the best lesson he ever had in debugging was watching a magic show. The magician did a half-dozen impossible tricks, and Lemons found himself tempted to believe them. He then reminded himself that the impossible isn't possible, and probed each stunt to resolve its apparent inconsistency. He started with what he knew to be bedrock truth -- the laws of physics -- and worked from there to find a simple explanation for each trick. This attitude makes Lemons one of the best debuggers I've ever seen.

The best book I've seen on debugging is *The Medical Detectives* by Berton Roueche', published by Penguin in 1991. The heroes in the book debug complex systems, ranging from mildly sick people to very sick towns. The problem-solving methods they use are directly applicable to debugging computer systems. These true stories are as spellbinding as any fiction.

Epilog to the First Edition of Programming Pearls



An interview with the author seemed, at the time, to be the best conclusion for the first edition of this book. It still describes the book, so here it is, once again.

Q: Thanks for agreeing to do this interview.

A: No problem -- my time is your time.

Q: Seeing how these columns already appeared in *Communications of the ACM*, why did you bother to collect them into a book?

A: There are several little reasons: I've fixed dozens of errors, made hundreds of little improvements, and added several new sections. There are fifty percent more problems, solutions and pictures in the book. Also, it's more convenient to have the columns in one book rather than a dozen magazines. The big reason, though, is that the themes running through the columns are easier to see when they are collected together; the whole is greater than the sum of the parts.

Q: What are those themes?

A: The most important is that thinking hard about programming can be both useful and fun. There's more to the job than systematic program development from formal requirements documents. If this book helps just one disillusioned programmer to fall back in love with his or her work, it will have served its purpose.

Q: That's a pretty fluffy answer. Are there technical threads tying the columns together?

A: Performance is the topic of Part II, and a theme that runs through all columns. Program verification is used extensively in several columns. Appendix 1 catalogs the algorithms in the book.

Q: It seems that most columns emphasize the design process. Can you summarize your advice on that topic?

A: I'm glad you asked. I just happened to prepare a list before this interview. Here are ten suggestions for programmers.

Work on the right problem.
Explore the design space of solutions.
Look at the data.
Use the back of the envelope.
Exploit symmetry.

Design with components.
Build prototypes.
Make tradeoffs when you have to.
Keep it simple.
Strive for elegance.

The points were originally discussed in the context of programming, but they apply to any engineering endeavor.

Q: That raises a point that has bothered me: it's easy to simplify the little programs in this book, but do the techniques scale up to real software?

A: I have three answers: yes, no and maybe. Yes they scale up; Section 3.4 [in the first edition], for instance, describes a huge software project that was simplified down to ``just'' 80 staff-years. An equally trite answer is no: if you simplify properly, you avoid building jumbo systems and the techniques don't need to scale up. Although there is merit in both views, the truth lies somewhere in between, and that's where the maybe comes in. Some software has to be big, and the themes of this book are sometimes applicable to such systems. The Unix system is a fine example of a powerful whole built out of simple and elegant parts.

Q: There you go talking about another Bell Labs system. Aren't these columns a little too parochial?

A: Maybe a little. I've stuck to material that I've seen used in practice, and that biases the book towards my environment. Phrased more positively, much of the material in these columns was contributed by my colleagues, and they deserve the credit (or blame). I've learned a lot from many researchers and developers within Bell Labs. There's a fine corporate atmosphere that encourages interaction between research and development. So a lot of what you call parochialism is just my enthusiasm for my employer.

Q: Let's come back down to earth. What pieces are missing from this book?

A: I had hoped to include a large system composed of many programs, but I couldn't describe any interesting systems in the ten or so pages of a typical column. At a more general level, I'd like to do future columns on the themes of ``computer science for programmers'' (like program verification in Column 4 and algorithm design in Column 8) and the ``engineering techniques of computing'' (like the back-of-the-envelope calculations in Column 7).

Q: If you're so into ``science'' and ``engineering'', how come the columns are so light on theorems and tables and so heavy on stories?

A: Watch it -- people who interview themselves shouldn't criticize writing styles.

Tricks of the Programming Trade

Jon Bentley
Bell Labs

Problem Definition
The Back of the Envelope
Debugging
Other Tricks

www.programmingpearls.com/tricks

Three Kinds of Knowledge

General

1. Science
2. Project Management
3. ‘‘Tricks of the Trade’’

Examples from Medicine

1. Chemistry, Biology
2. Operating teams, Medical records
3. Blood samples

Examples from Software

1. Program verification, algorithm design
2. Team organization, scheduling
3. This talk

About This Talk

Goals

For the listeners: a few amusing stories, and a prod to think about tricks.

For the talker: more stories and tricks.

Rules

Mandatory: real-time thought, some discussion.

Illegal: note taking, verbal answers from table-lookup.

Problem Definition

Context

A public opinion polling firm wishes to draw a random sample from a (hardcopy) list of precincts.

The User's Idea

Input: The user types n precinct names (10-character strings) and an integer $m < n$. Typically, $m \approx 20$ and $n \approx 200$.

Output: Random selection of m names.

A Better Problem

Input: Integers m and n.

Output: A sorted list of m integers in the range 1..n. For example, if $m = 3$ and $n = 8$, the output might be 2, 3, 5.

Better Yet

Input: A hardcopy list of precincts.

Another Randomizing Problem

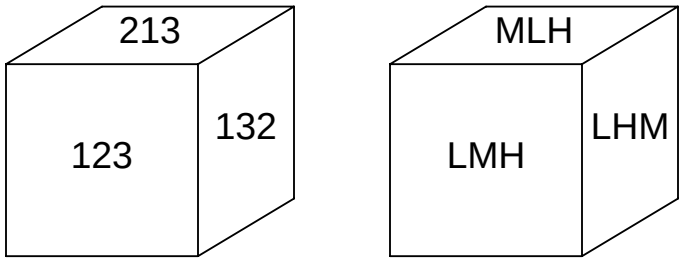
The Problem

For $N = 72$ psychological subjects, randomly permute order of experimenters (1, 2, 3) and stress conditions (High, Medium, Low).

Output of the Program

```
1 3L 2M 1H
2 3H 1M 2L
3 1L 2H 3M
4 1M 2L 3H
  ...
```

A More Elegant Solution



Why it failed

A Quiz

How much water flows out of the Mississippi River per day?

One Possible Answer. The river is about a mile wide, twenty feet deep, and flows at five miles per hour.

$$1 \text{ mile} \times 1/250 \text{ mile} \times 120 \text{ miles/day}$$

$$\approx 1/2 \text{ mile}^3/\text{day}$$

It discharges roughly one-half cubic mile per day.

Two Answers Are Better Than One

How much water flows out? As much as flows in.

The Mississippi basin is about 1000 by 1000 miles, and the annual runoff from rainfall is about one foot. Yearly drainage is

$$1000 \text{ miles} \times 1000 \text{ miles} \times 1/5000 \text{ mile/year}$$

$$\approx 200 \text{ miles}^3/\text{year}$$

so daily outflow is

$$200 \text{ miles}^3/\text{year} \; / \; 400 \text{ days/year}$$

$$\approx 1/2 \text{ mile}^3/\text{day}$$

A cheating triple check: An almanac reports that the river's discharge is 640,000 cubic feet per second, or 0.4 cubic miles per day.

Tabular Computation

How much water flows in per year?

$$\frac{1000\ \text{miles}}{} \cdot \frac{1000\ \text{miles}}{} \cdot \frac{1\ \text{mile}}{5000\ \text{year}}$$

Cancel terms.

$$\frac{\cancel{1000}\ \cancel{\text{miles}}}{} \cdot \frac{\cancel{1000}\ \cancel{\text{miles}}}{} \cdot \frac{\cancel{1}\ \cancel{\text{mile}}}{\cancel{5000}\ \text{year}} \cdot \frac{200\ \text{mile}^3}{}$$

Multiply by 1 year = 400 days (almost).

$$\frac{\cancel{1000}\ \cancel{\text{miles}}}{} \cdot \frac{\cancel{1000}\ \cancel{\text{miles}}}{} \cdot \frac{\cancel{1}\ \cancel{\text{mile}}}{\cancel{5000}\ \text{year}} \cdot \frac{200\ \text{mile}^3}{} \cdot \frac{\text{year}}{400\ \text{days}}$$

Cancel again.

$$\frac{\cancel{1000}\ \cancel{\text{miles}}}{} \cdot \frac{\cancel{1000}\ \cancel{\text{miles}}}{} \cdot \frac{\cancel{1}\ \cancel{\text{mile}}}{\cancel{5000}\ \cancel{\text{year}}} \cdot \frac{\cancel{200}\ \text{mile}^3}{} \cdot \frac{\cancel{\text{year}}}{\cancel{400}\ \text{days}} \cdot \frac{1}{2}$$

Reminders About Quick Calculations

How To Do Them.

Two answers are better than one.
Tabular computations.
Dimension checks.
Quick checks.
Common sense.
Rules of thumb.
Safety factors

When To Do Them.

After a system has been designed, before starting implementation.

Before any efficiency improvements.

Cost-benefits analysis.

Exercises

When is a cyclist with removable media faster than a high-speed data line?

A newspaper asserts that a quarter has ‘‘an average life of 30 years’’. Is that reasonable?

An advertisement asserts that a salesperson drives a car 100,000 miles in a year. Is that reasonable?

How many tons of people are in this room? How many cubic feet of people? Of room?

If every person in this city threw a ping pong ball into this room, how deep would we be?

How long to read a CD-ROM?

How much money will Americans spend on soft drinks this year?

How many words in a book? How many words a minute do you read? How many hours per week do you watch television?

Rules of Thumb

π seconds is a nanocentury.

Rule of 72

Little's Law

90-10 (80-20) Rules. 10% of X accounts for 90% of Y: (population, beer) (code, run time) (code, function)

The cheapest, fastest and most reliable components of a computer system are those that aren't there.

Whenever possible, steal code.

[The Test of Negation] Don't include a sentence in documentation if its negation is obviously false.

[The Principle of Least Astonishment] Make a user interface as consistent and as predictable as possible.

[KISS] Keep it simple, stupid.

Approaches to Debugging

Program Verification

Don’t make any mistakes.

Theory of Test Data Selection

Uncover all mistakes.

Advanced Debuggers

Use ’em if you have ’em.

The Right Attitude

Debugging Steubenville, Ohio

From Roueché's *Medical Detectives* (Penguin, 1991)

Background

36 cases of salmonella from December, 1980 through February, 1982

A Good Debugger

''I was prepared for hard work, for plenty of old-fashioned shoe-leather epidemiology.''

A Key Hint

''In such outbreaks, there is always a common denominator — an eating place, a wedding reception, a picnic, a dinner party, a church social. Something in which everybody is involved.''

Dominant age group: 20-29

Debugging Computer Systems

Some Puzzles

A banking system ‘‘quits’’ on international data.

The programmer can’t log in standing up.

The program worked once twice.

The program works when the programmers are in the room, but fails half the time when they aren’t.

A Moral

Debugging is usually unbelieving.

The key is skepticism: the magician.

Tricks of the Trade

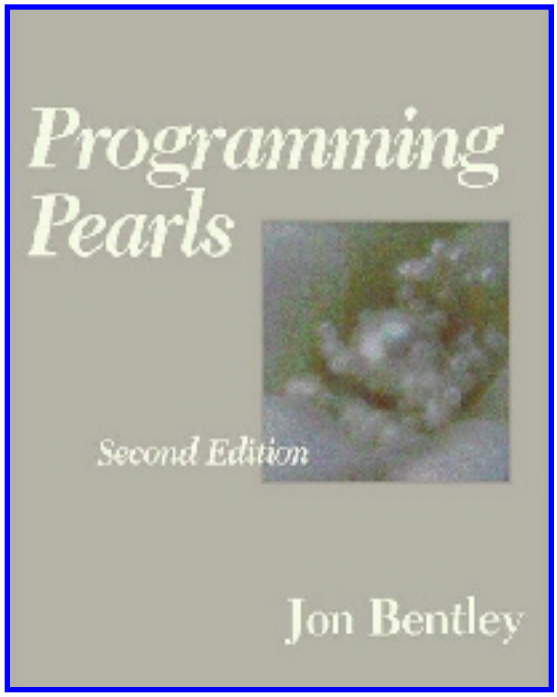
Ten Design Hints

Work on the right problem.
Explore the design space of solutions.
Look at the data.
Use the back of the envelope.
Exploit symmetry.
Design with components.
Build prototypes.
Make tradeoffs when you have to.
Keep it simple.
Strive for elegance.

Nurturing Trickiness

Observe
Read
Discuss
Practice
Reflect

Precise Problem Statement (Section 1.2 of Programming Pearls)



To the programmer these requirements added up to, ``How do I sort a disk file?" Before we attack the problem, let's arrange what we know in a less biased and more useful form.

Input:
A file containing at most n positive integers, each less than n, where $n = 10^7$. It is a fatal error if any integer occurs twice in the input. No other data is associated with the integer.

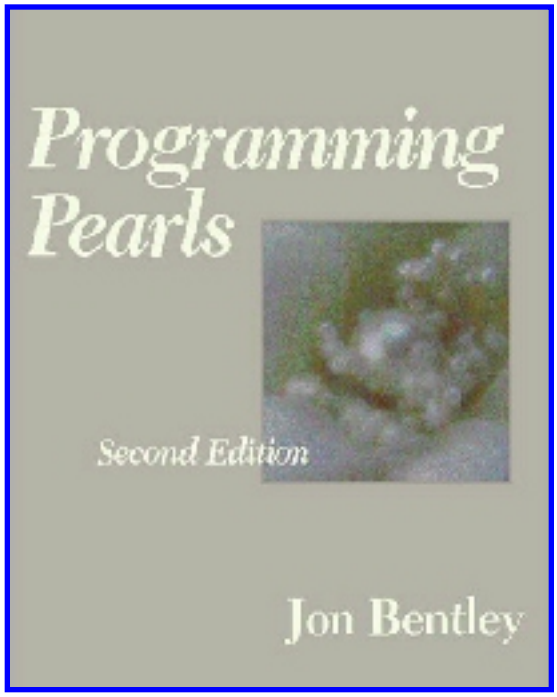
Output:
A sorted list in increasing order of the input integers.

Constraints:
At most (roughly) a megabyte of storage is available in main memory; ample disk storage is available. The run time can be at most several minutes; a run time of ten seconds need not be decreased.

Think for a minute about this problem specification. How would you advise the programmer now?

Next: Section 1.3. Program Design.

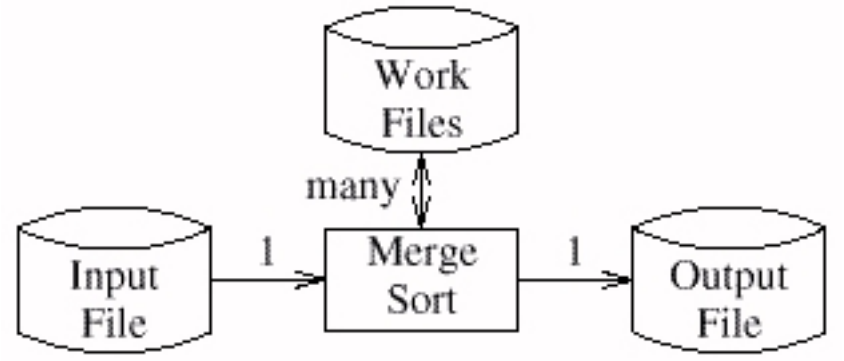
Program Design (Section 1.3 of Programming Pearls)



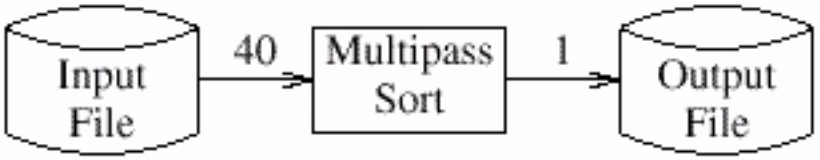
The obvious program uses a general disk-based Merge Sort as a starting point but trims it to exploit the fact that we are sorting integers. That reduces the two hundred lines of code by a few dozen lines, and also makes it run faster. It might still take a few days to get the code up and running.

A second solution makes even more use of the particular nature of this sorting problem. If we store each number in seven bytes, then we can store about 143,000 numbers in the available megabyte. If we represent each number as a 32-bit integer, though, then we can store 250,000 numbers in the megabyte. We will therefore use a program that makes 40 passes over the input file. On the first pass it reads into memory any integer between 0 and 249,999, sorts the (at most) 250,000 integers and writes them to the output file. The second pass sorts the integers from 250,000 to 499,999, and so on to the 40th pass, which sorts 9,750,000 to 9,999,999. A Quicksort would be quite efficient for the main-memory sorts, and it requires only twenty lines of code (as we'll see in Column 11). The entire program could therefore be implemented in a page or two of code. It also has the desirable property that we no longer have to worry about using intermediate disk files; unfortunately, for that benefit we pay the price of reading the entire input file 40 times.

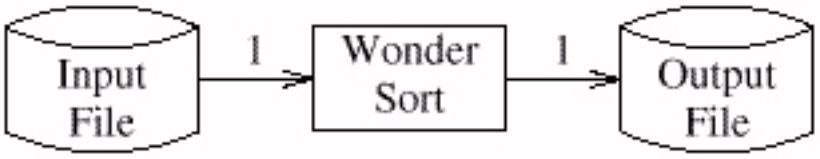
A Merge Sort program reads the file once from the input, sorts it with the aid of work files that are read and written many times, and then writes it once.



The 40-pass algorithm reads the input file many times and writes the output just once, using no intermediate files.



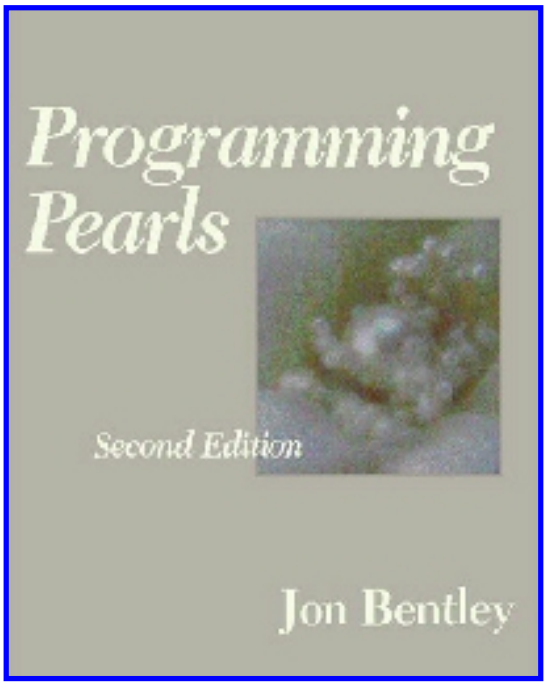
We would prefer the following scheme, which combines the advantages of the previous two. It reads the input just once, and uses no intermediate files.



We can do this only if we represent all the integers in the input file in the available megabyte of main memory. Thus the problem boils down to whether we can represent at most ten million distinct integers in *about* eight million available bits. Think about an appropriate representation.

Next: Section 1.4. Implementation Sketch.

Implementation Sketch (Section 1.4 of Programming Pearls)



Viewed in this light, the *bitmap bit vector* representation of a set screams out to be used. We can represent a toy set of nonnegative integers less than 20 by a string of 20 bits. For instance, we can store the set {1, 2, 3, 5, 8, 13} in this string:

```
0 1 1 1 0 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0
```

The bits representing numbers in the set are 1, and all other bits are 0.

In the real problem, the seven decimal digits of each integer denote a number less than ten million. We'll represent the file by a string of ten million bits in which the i^{th} bit is on if and only if the integer i is in the file. (The programmer found two million spare bits; Problem 5 investigates what happens when a megabyte is a firm limit.) This representation uses three attributes of this problem not usually found in sorting problems: the input is from a relatively small range, it contains no duplicates, and no data is associated with each record beyond the single integer.

Given the bitmap data structure to represent the set of integers in the file, the program can be written in three natural phases. The first phase initializes the set to empty by turning off all bits. The second phase builds the set by reading each integer in the file and turning on the appropriate bit. The third phase produces the sorted output file by inspecting each bit and writing out the appropriate integer if the bit is one. If n is the number of bits in the vector (in this case 10,000,000), the program can be expressed in pseudocode as:

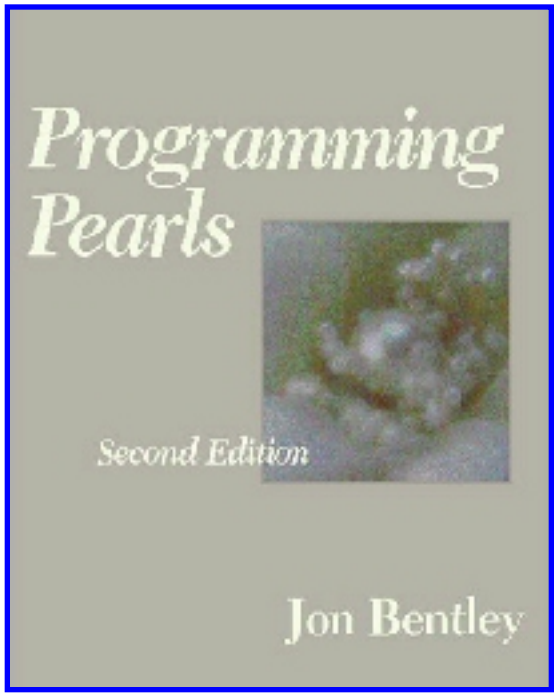
```
/* phase 1: initialize set to empty */
    for i = [0, n)
        bit[i] = 0
/* phase 2: insert present elements into the set */
    for each i in the input file
        bit[i] = 1
/* phase 3: write sorted output */
    for i = [0, n)
        if bit[i] == 1
            write i on the output file
```

(Recall from the preface that the notation *for i = [0, n)* iterates i from 0 to n-1.)

This sketch was sufficient for the programmer to solve his problem. Some of the implementation details he faced are described in Problems 2, 5 and 7.

Next: Section 1.5. Principles.

Principles (Section 1.5 of Programming Pearls)



The programmer told me about his problem in a phone call; it took us about fifteen minutes to get to the real problem and find the bitmap solution. It took him a couple of hours to implement the program in a few dozen lines of code, which was far superior to the hundreds of lines of code and the week of programming time that we had feared at the start of the phone call. And the program was lightning fast: while a Merge Sort on disk might have taken many minutes, this program took little more than the time to read the input and to write the output -- about ten seconds. Solution 3 contains timing details on several programs for the task.

Those facts contain the first lesson from this case study: careful analysis of a small problem can sometimes yield tremendous practical benefits. In this case a few minutes of careful study led to an order of magnitude reduction in code length, programmer time and run time. General Chuck Yeager (the first person to fly faster than sound) praised an airplane's engine system with the words ``simple, few parts, easy to maintain, very strong''; this program shares those attributes. The program's specialized structure, however, would be hard to modify if certain dimensions of the specifications were changed. In addition to the advertising for clever programming, this case illustrates the following general principles.

The Right Problem. Defining the problem was about ninety percent of this battle -- I'm glad that the programmer didn't settle for the first program I described. Problems 10, 11 and 12 have elegant solutions once you pose the right problem; think hard about them before looking at the hints and solutions.

The Bitmap Data Structure. This data structure represents a dense set over a finite domain when each element occurs at most once and no other data is associated with the element. Even if these conditions aren't satisfied (when there are multiple elements or extra data, for instance), a key from a finite domain can be used as an index into a table with more complicated entries; see Problems 6 and 8.

Multiple-Pass Algorithms. These algorithms make several passes over their input data, accomplishing a little more each time. We saw a 40-pass algorithm in Section 1.3; Problem 5 encourages you to develop a two-pass algorithm.

A Time-Space Tradeoff and One That Isn't. Programming folklore and theory abound with time-space tradeoffs: by using more time, a program can run in less space. The two-pass algorithm in Solution 5, for instance, doubles a program's run time to halve its space. It has been my experience more frequently, though, that reducing a program's space requirements also reduces its run time. (Tradeoffs are common to all engineering disciplines; automobile designers, for instance, might trade reduced mileage for faster acceleration by adding heavy components. Mutual improvements are preferred, though. A review of a small car I once drove observed that ``the weight saving on the car's basic structure translates into further weight reductions in the various chassis components -- and even the elimination of the need for some, such as power steering''.) The space-efficient structure of bitmaps dramatically reduced the run time of sorting. There were two reasons that the reduction in space led to a reduction in time: less

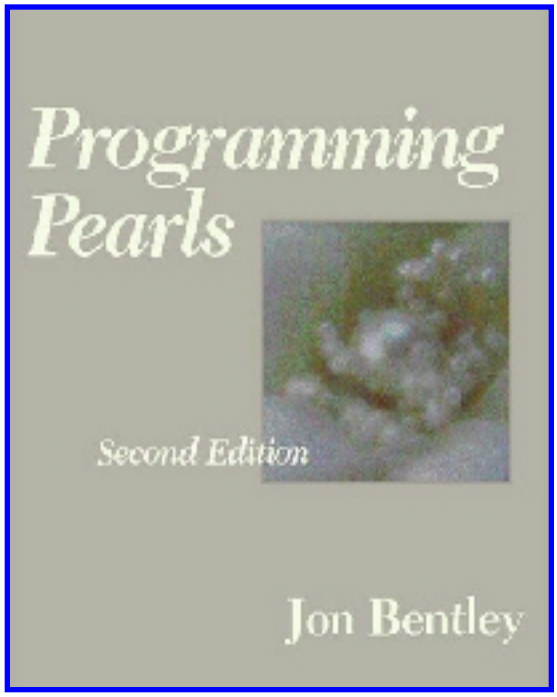
data to process means less time to process it, and keeping data in main memory rather than on disk avoids the overhead of disk accesses. Of course, the mutual improvement was possible only because the original design was far from optimal.

A Simple Design. Antoine de Saint-Exupery, the French writer and aircraft designer, said that, ``A designer knows he has arrived at perfection not when there is no longer anything to add, but when there is no longer anything to take away." More programmers should judge their work by this criterion. Simple programs are usually more reliable, secure, robust and efficient than their complex cousins, and easier to build and to maintain.

Stages of Program Design. This case illustrates the design process that is described in detail in Section 12.4.

Next: Section 1.6. Problems.

Problems (Section 1.6 of Programming Pearls)



The Solutions to Column 1 give answers for some of these problems.

1. If memory were not scarce, how would you implement a sort in a language with libraries for representing and sorting sets?

2. How would you implement bit vectors using bitwise logical operations (such as and, or and shift)?

3. Run-time efficiency was an important part of the design goal, and the resulting program was efficient enough. Implement the bitmap sort on your system and measure its run time; how does it compare to the system sort and to the sorts in Problem 1? Assume that *n* is 10,000,000, and that the input file contains 1,000,000 integers.

4. If you take Problem 3 seriously, you will face the problem of generating *k* integers less than *n* without duplicates. The simplest approach uses the first *c* positive integers. This extreme data set won't alter the run time of the bitmap method by much, but it might skew the run time of a system sort. How could you generate a file of *k* unique random integers between 0 and *n*-1 in random order? Strive for a short program that is also efficient.

5. The programmer said that he had about a megabyte of free storage, but the code we sketched uses 1.25 megabytes. He was able to scrounge the extra space without much trouble. If the megabyte had been a hard and fast boundary, what would you have recommended? What is the run time of your algorithm?

6. What would you recommend to the programmer if, instead of saying that each integer could appear at most once, he told you that each integer could appear at most ten times? How would your solution change as a function of the amount of available storage?

7. [R. Weil] The program as sketched has several flaws. The first is that it assumes that no integer appears twice in the input. What happens if one does show up more than once? How could the program be modified to call an error function in that case? What happens when an input integer is less than zero or greater than or equal to *n*? What if an input is not numeric? What should a program do under those circumstances? What other sanity checks could the program incorporate? Describe small data sets that test the program, including its proper handling of these and other ill-behaved cases.

8. When the programmer faced the problem, all toll-free phone numbers in the United States had the 800 area code. Toll-free codes now include 800, 877 and 888, and the list is growing. How would you sort all of the toll-free numbers using only a megabyte? How can you store a set of toll-free numbers to allow very rapid lookup to determine whether a given toll-free number is available or already taken?

9. One problem with trading more space to use less time is that initializing the space can itself take a great deal of

time. Show how to circumvent this problem by designing a technique to initialize an entry of a vector to zero the first time it is accessed. Your scheme should use constant time for initialization and for each vector access, and use extra space proportional to the size of the vector. Because this method reduces initialization time by using even more space, it should be considered only when space is cheap, time is dear and the vector is sparse.

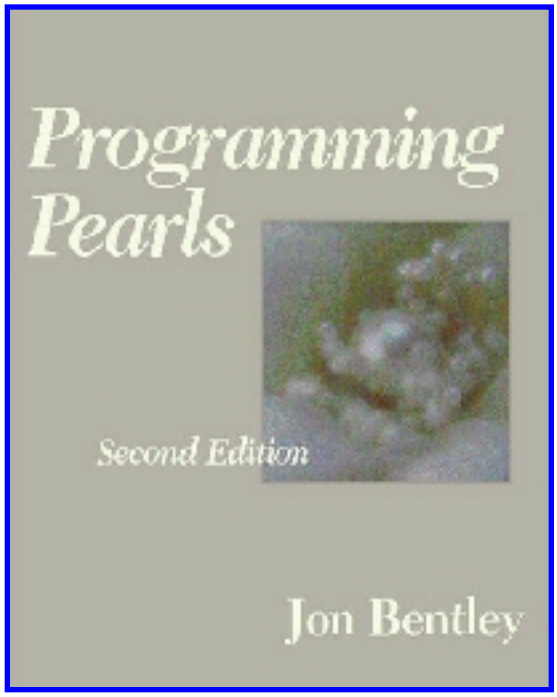
10. Before the days of low-cost overnight deliveries, a store allowed customers to order items over the telephone, which they picked up a few days later. The store's database used the customer's telephone number as the primary key for retrieval (customers know their phone numbers and the keys are close to unique). How would you organize the store's database to allow orders to be inserted and retrieved efficiently?

11. In the early 1980's Lockheed engineers transmitted daily a dozen drawings from a Computer Aided Design (CAD) system in their Sunnyvale, California, plant to a test station in Santa Cruz. Although the facilities were just 25 miles apart, an automobile courier service took over an hour (due to traffic jams and mountain roads) and cost a hundred dollars per day. Propose alternative data transmission schemes and estimate their cost.

12. Pioneers of human space flight soon realized the need for writing implements that work well in the extreme environment of space. A popular urban legend asserts that the United States National Aeronautics and Space Administration (NASA) solved the problem with a million dollars of research to develop a special pen. According to the legend, how did the Soviets solve the same problem?

Next: Section 1.7. Further Reading.

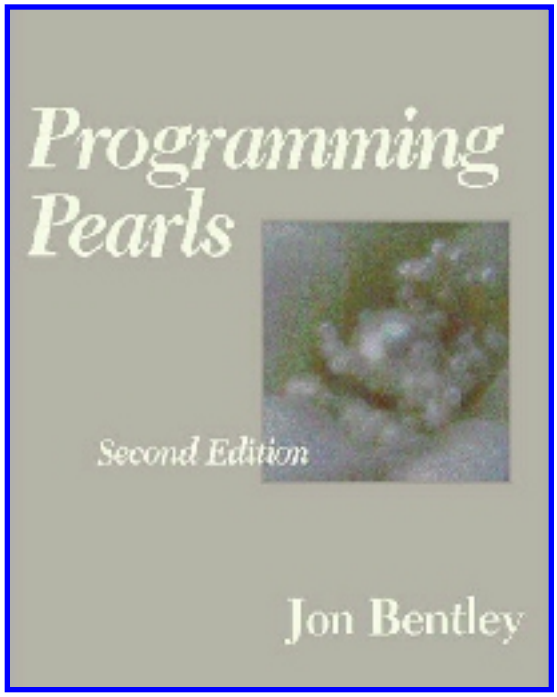
Further Reading (Section 1.7 of Programming Pearls)



This little exercise has only skimmed the fascinating topic of specifying a program. For a deep insight into this crucial activity, see Michael Jackson's *Software Requirements & Specifications*, published by Addison-Wesley in 1995. The tough topics in the book are presented as a delightful collection of independent but reinforcing little essays.

In the case study described in this column, the programmer's main problem was not so much technical as psychological: he couldn't make progress because he was trying to solve the wrong problem. We finally solved his problem by breaking through his conceptual block and solving an easier problem. *Conceptual Blockbusting* by James L. Adams (the third edition was published by Perseus in 1986) studies this kind of leap and is generally a pleasant prod towards more creative thinking. Although it was not written with programmers in mind, many of its lessons are particularly appropriate for programming problems. Adams defines conceptual blocks as ``mental walls that block the problem-solver from correctly perceiving a problem or conceiving its solution''; Problems 10, 11 and 12 encourage you to bust some.

Basic Skills (Section 7.1 of Programming Pearls)



These reminders can be helpful in making back-of-the-envelope calculations.

Two Answers Are Better Than One. 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 that the annual runoff from rainfall there was about one foot (or one five-thousandth of a mile). That gives

1000 miles * 1000 miles * 1/5000 mile/year ~ 200 $miles^3$/year

(200 $miles^3$/year) / (400 days/year) ~ 1/2 $mile^3$/day

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. Working from that gives

640,000ft^3/sec * 3600 secs/hr ~ 2.3x$10^9$$ft^3$ / hr

2.3x$10^9$$ft^3$/hr * 24hrs/day ~ 6x$10^{10}$$ft^3$/day

6x$10^{10}$$ft^3$/day / (5000ft/mile)^3

~ 6x$10^{10}$$ft^3$/day / (125x$10^9$$ft^3$/$mile^3$)

~ 60/125 $mile^3$/day

~ 1/2 $mile^3$/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.

Quick Checks. Polya devotes three pages of his *How to Solve It* to ``Test by Dimension", which he describes as a ``well-known, quick and efficient means to check geometrical or physical formulas". The first rule is that the dimensions in a sum must be the same, which is in turn the dimension of the sum -- you can add feet together to get feet, but you can't add seconds to pounds. The second rule is that the dimension of a product is the product of the dimensions. The examples above obey both rules; multiplying

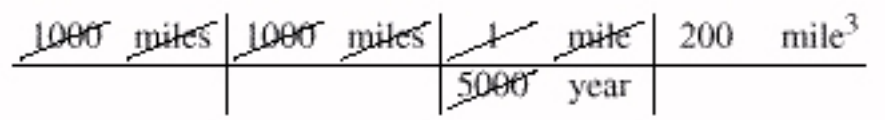
(miles + miles) x miles x miles / day ~ $miles^3$/day

has the right form, apart from any constants.

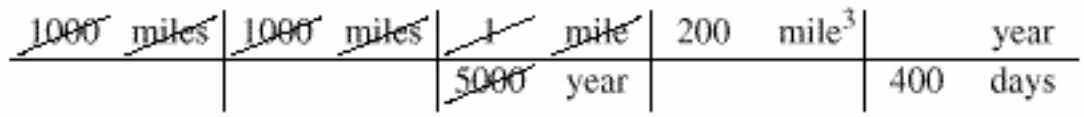
A simple table can help you keep track of dimensions in complicated expressions like those above. To perform Weinberger's calculation, we first write down the three original factors.

1000 miles	1000 miles	1 mile
		5000 year

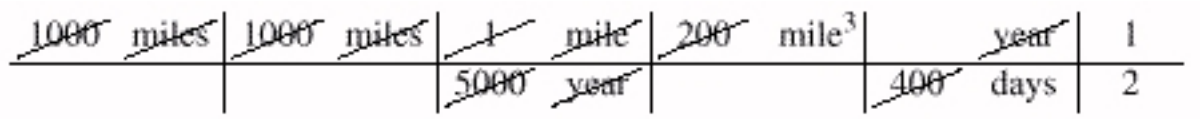
Next we simplify the expression by cancelling terms, which shows that the output is 200 $miles^3$/year.



Now we multiply by the identity (well, almost) that there are 400 days per year.



Cancellation yields the (by now familiar) answer of half a cubic mile per day.



These tabular calculations help you keep track of dimensions.

Dimension tests check the form of equations. Check your multiplications and divisions with an old trick from slide rule days: independently compute the leading digit and the exponent. One can make several quick checks for addition.

```
  3142        3142        3142
  2718        2718        2718
 +1123       +1123       +1123
   983        6982        6973
```

The first sum has too few digits and the second sum errs in the least significant digit. The technique of ``casting out nines" reveals the error in the third example: the digits in the summands sum to 8 modulo 9, while those in the answer sum to 7 modulo 9. In a correct addition, the sums of the digits are equal after ``casting out" groups of digits that sum to nine.

Above all, don't forget common sense: be suspicious of any calculations that show that the Mississippi River discharges 100 gallons of water per day.

Rules of Thumb. I first learned the ``Rule of 72" in a course on accounting. Assume that you invest a sum of money for y years at an interest rate of r percent per year. The financial version of the rule says that if $r \text{ times } y = 72$, then your money will roughly double. The approximation is quite accurate: investing $1000 at 6 percent interest for 12 years gives $2,012, and $1000 at 8 percent for 9 years gives $1,999.

The Rule of 72 is handy for estimating the growth of any exponential process. If a bacterial colony in a dish grows at the rate of three percent per hour, then it doubles in size every day. And doubling brings programmers back to familiar rules of thumb: because 2^{10}=1024, ten doublings is about a thousand, twenty doublings is about a million, and thirty doublings is about a billion.

Suppose that an exponential program takes ten seconds to solve a problem of size n=40, and that increasing n by one increases the run time by 12 percent (we probably learned this by plotting its growth on a logarithmic scale). The Rule of 72 tells us that the run time doubles when n increases by 6, or goes up by a factor of about 1000 when n increases by 60. When n=100, the program should therefore take about 10,000 seconds, or a few hours. But what happens when n increases to 160, and the time rises to 10^7 seconds? How much time is that?

You might find it hard to memorize that there are 3.155×10^7 seconds in a year. On the other hand, it is hard to

forget Tom Duff's handy rule of thumb that, to within half a percent,

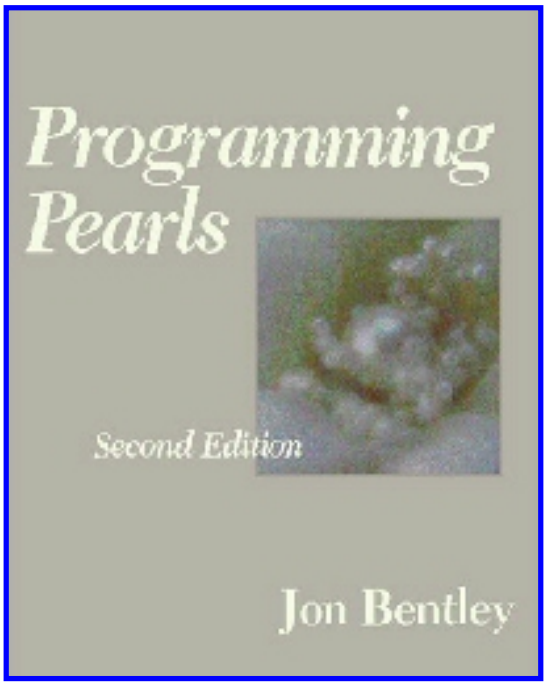
Pi seconds is a nanocentury.

Because the exponential program takes 10^7 seconds, we should be prepared to wait about four months.

Practice. As with many activities, your estimation skills can only be improved with practice. Try the problems at the end of this column, and the estimation quiz in Appendix 2 (a similar quiz once gave me a much-needed dose of humility about my estimation prowess). Section 7.8 describes quick calculations in everyday life. Most workplaces provide ample opportunities for back-of-the-envelope estimates. How many foam ``packing peanuts'' came in that box? How much time do people at your facility spend waiting in line every day, for morning coffee, lunch, photocopiers and the like? How much does that cost the company in (loaded) salary? And the next time you're *really* bored at the lunch table, ask your colleagues how much water flows out of the Mississippi River each day.

Next: Section 7.2. Performance Estimates.

Performance Estimates (Section 7.2 of Programming Pearls)



Let's turn now to a quick calculation in computing. Nodes in your data structure (it might be a linked list or a hash table) hold an integer and a pointer to a node:

```
struct node { int i; struct node *p; };
```

Back-of-the-envelope quiz: will two million such nodes fit in the main memory of your 128-megabyte computer?

Looking at my system performance monitor shows that my 128-megabyte machine typically has about 85 megabytes free. (I've verified that by running the vector rotation code from Column 2 to see when the disk starts thrashing.) But how much memory will a node take? In the old days of 16-bit machines, a pointer and an integer would take four bytes. As I write this edition of the book, 32-bit integers and pointers are most common, so I would expect the answer to be eight bytes. But every now and then I compile in 64-bit mode, so it might take sixteen bytes. We can find out the answer for any particular system with a single line of C:

```
printf("sizeof(struct node)=%d\n", sizeof(struct node));
```

My system represented each record in eight bytes, as I expected. The 16 megabytes should fit comfortably into the 85 megabytes of free memory.

So when I used two million of those eight-byte records, why did my 128-megabyte machine start thrashing like crazy? They key is that I allocated them dynamically, using the C *malloc* function (similar to the C++ *new* operator). I had assumed that the eight-byte records might have another 8 bytes of overhead; I expected the nodes to take a total of about 32 megabytes. In fact, each node had 40 bytes of overhead to consume a total of 48 bytes per record. The two million records therefore used a total of 96 megabytes. (On other systems and compilers, though, the records had just 8 bytes of overhead each.)

Appendix 3 describes a program that explores the memory cost of several common structures. The first lines it produces are made with the *sizeof* operator:

```
sizeof(char)=1  sizeof(short)=2  sizeof(int)=4
sizeof(float)=4  sizeof(struct *)=4  sizeof(long)=4
sizeof(double)=8
```

I expected precisely those values from my 32-bit compiler. Further experiments measured the differences between successive pointers returned by the storage allocator; this is a plausible guess at record size. (One should always verify such rough guesses with other tools.) I now understand that, with this space-hogging allocator, records between 1 and 12 bytes will consume 48 bytes of memory, records between 13 and 28 bytes will consume 64 bytes, and so forth. We will return to this space model in Columns 10 and 13.

Let's try one more quick computing quiz. You know that the run time of your numerical algorithm is dominated by its n^3 square root operations, and n=1000 in this application. About how long will your program take to compute the one billion square roots?

To find the answer on my system, I'd start with this little C program:

```
#include
int main(void)
{   int i, n = 1000000;
    float fa;
    for (i = 0; i < n; i++)
        fa = sqrt(10.0);
    return 0;
}
```

I ran the program with a command to report how much time it takes (I check such times with an old digital watch that I keep next to my computer; it has a broken band but a working stopwatch). I found that the program took about 0.2 seconds to computer one million square roots, 2 seconds to compute ten million, and 20 seconds to compute 100 million. I would guess that it will take about 200 seconds to compute a billion square roots.

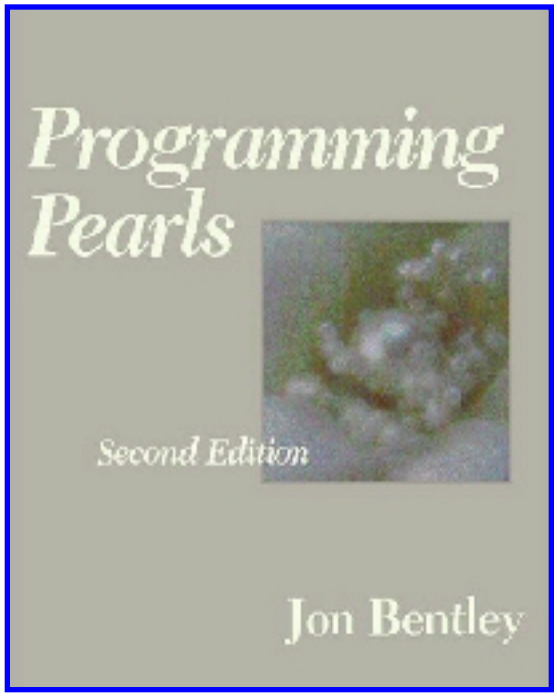
But will a square root in a real program take 200 nanoseconds? It might be much slower: perhaps the square root function cached the most recent argument as a starting value. Calling such a function repeatedly with the same argument might give it an unrealistic advantage. Then again, in practice the function might be much faster: I compiled the program with optimization disabled (optimization removed the main loop, so it always ran in zero time). Appendix 3 describes how to expand this tiny program to produce a one-page description of the time costs of primitive C operations for a given system.

How fast is networking? To find out, I typed *ping machine-name*. It takes a few milliseconds to *ping* a machine in the same building, so that represents the startup time. On a good day, I can *ping* a machine on the other coast of the United States in about 70 milliseconds (traversing the 5000 miles of the round-trip voyage at the speed of light accounts for about 27 of those milliseconds); on a bad day, I get timed out after waiting 1000 milliseconds. Measuring the time to copy a large file shows that a ten-megabit Ethernet moves about a megabyte a second (that is, it achieves about 80 percent of its potential bandwidth). Similarly, a hundred-megabit Ethernet with the wind at its back moves ten megabytes a second.

Little experiments can put key parameters at your fingertips. Database designers should know the times for reading and writing records, and for joins of various forms. Graphics programmers should know the cost of key screen operations. The short time required to do such little experiments today will be more than paid in the time you save by making wise decisions tomorrow.

Next: Section 7.3. Safety Factors.

Safety Factors (Section 7.3 of Programming Pearls)



The output of any calculation is only as good as its input. With good data, simple calculations can yield accurate answers that are sometimes quite useful. Don Knuth once 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, Knuth's sorting package behaved as predicted and every other disk-bound program also ran faster.

Often, though, sloppy input is enough to get into the right ballpark. (The estimation quiz in Appendix 2 may help you to judge the quality of your guesses.) 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. But before placing too much confidence in a twenty percent margin of error, 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. Well, suspension bridges had been ripping themselves apart that way for eighty 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 1950's 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.'

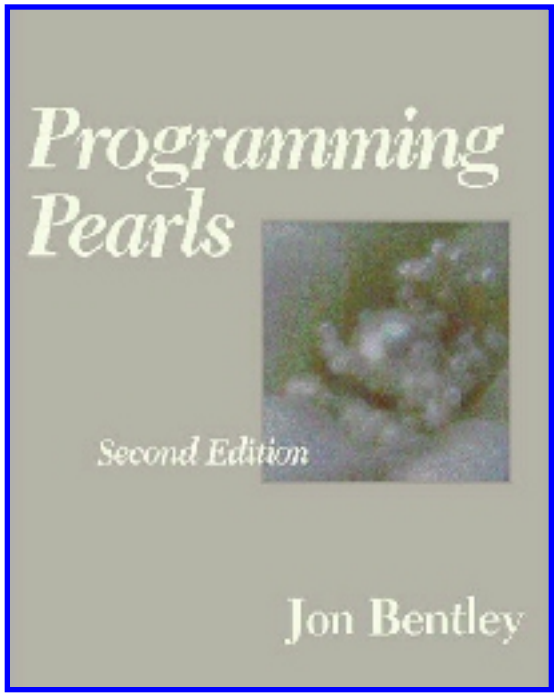
``Roebling was a good engineer, and he built a good bridge, by employing a huge safety factor to compensate for his ignorance. Do we do that? I submit to you that in calculating performance of our real-time software systems we ought to derate them by a factor of two, or four, or six, to compensate for 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 1870's collapsed within ten years of their construction.

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

Next: Section 7.4. Little's Law.

Little's Law (Section 7.4 of Programming Pearls)



Most back-of-the-envelope calculations use obvious rules: total cost is unit cost times number of units. Sometimes, though, one needs a more subtle insight. Bruce Weide wrote the following note about a surprisingly versatile rule.

``The `operational analysis' introduced by Denning and Buzen (see *Computing Surveys 10*, 3, November 1978, 225-261) is much more general than queueing network models of computer systems. Their exposition is excellent, but because of the article's limited focus, they didn't explore the generality of Little's Law. The proof methods have nothing to do with queues or with computer systems. Imagine *any* system in which things enter and leave. Little's Law states that `The average number of things in the system is the product of the average rate at which things leave the system and the average time each one spends in the system.' (And if there is a gross `flow balance' of things entering and leaving, the exit rate is also the entry rate.)

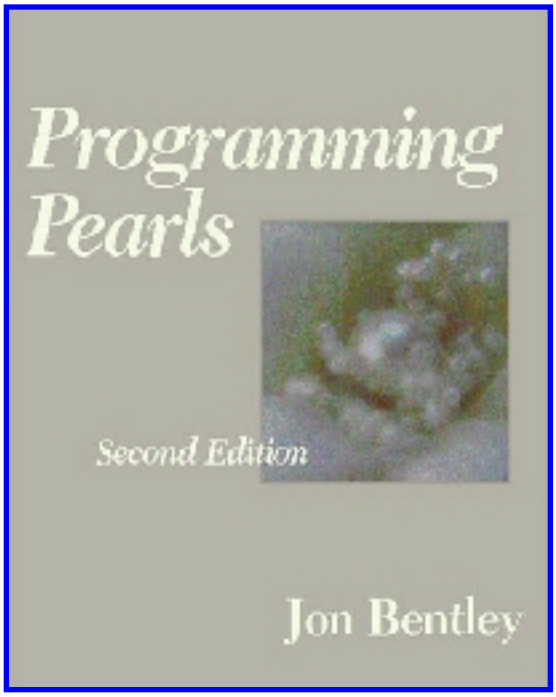
``I teach this technique of performance analysis in my computer architecture classes at Ohio State University. But I try to emphasize that the result is a general law of systems theory, and can be applied to many other kinds of systems. For instance, if you're in line waiting to get into a popular nightspot, you might figure out how long you'll have to wait by standing there for a while and trying to estimate the rate at which people are entering. With Little's Law, though, you could reason, `This place holds about 60 people, and the average Joe will be in there about 3 hours, so we're entering at the rate of about 20 people an hour. The line has 20 people in it, so that means we'll wait about an hour. Let's go home and read *Programming Pearls* instead.' You get the picture."

Peter Denning succinctly phrases this rule as ``The average number of objects in a queue is the product of the entry rate and the average holding time." He applies it to his wine cellar: ``I have 150 cases of wine in my basement and I consume (and purchase) 25 cases per year. How long do I hold each case? Little's Law tells me to divide 150 cases by 25 cases/year, which gives 6 years per case."

He then turns to more serious applications. ``The response-time formula for a multi-user system can be proved using Little's Law and flow balance. Assume n users of average think time z are connected to an arbitrary system with response time r. Each user cycles between thinking and waiting-for-response, so the total number of jobs in the meta-system (consisting of users and the computer system) is fixed at n. If you cut the path from the system's output to the users, you see a meta-system with average load n, average response time $z+r$, and throughput x (measured in jobs per time unit). Little's Law says $n = x*(z+r)$, and solving for r gives $r = n/x - z$."

Next: Section 7.5. Principles.

Principles (Section 7.5 of Programming Pearls)



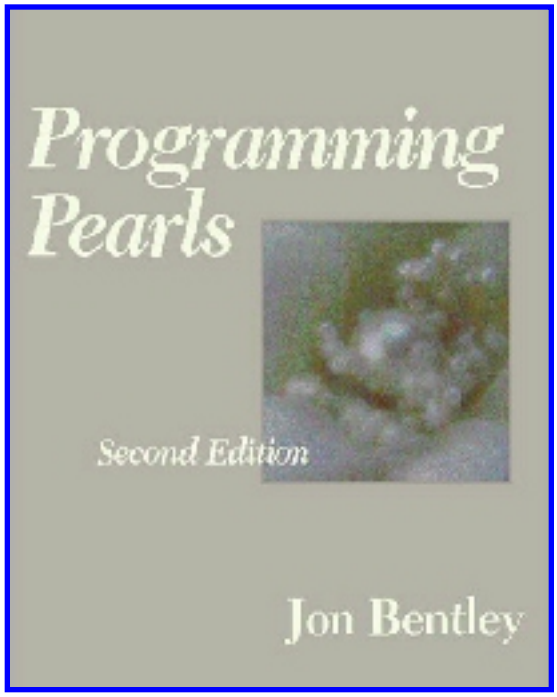
When you use back-of-the-envelope calculations, be sure to recall Einstein's famous advice.

> *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.

Next: Section 7.6. Problems.

Problems (Section 7.6 of Programming Pearls)



The quiz in Appendix 2 contains additional problems.

The Solutions to Column 7 give answers for some of these problems.

1. While Bell Labs is about a thousand miles from the mighty Mississippi, we are only a couple of miles from the usually peaceful Passaic River. After a particularly heavy week of rains, the June 10, 1992, edition of the *Star-Ledger* quoted an engineer as saying that ``the river was traveling about 200 miles per hour, about five times faster than average". Any comments?

2. At what distances can a courier on a bicycle with removable media be a more rapid carrier of information than a high-speed data line?

3. How long would it take you to fill a floppy disk by typing?

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. Prove why ``casting out nines" correctly tests addition. How can you further test the Rule of 72? What can you prove about it?

6. A United Nations estimate put the 1998 world population at 5.9 billion and the annual growth rate at 1.33 percent. Were this rate to continue, what would the population be in 2050?

7. Appendix 3 describes programs to produce models of the time and space costs of your system. After reading about the models, write down your guesses for the costs on your system. Retrieve the programs from the book's web site, run them on your system, and compare those estimates to your guesses.

8. Use quick calculations to estimate the run time of designs sketched in this book.

    a. Evaluate programs and designs for their time and space requirements.

    b. Big-oh arithmetic can be viewed as a formalization of quick calculations -- it captures the growth rate but ignores constant factors. Use the big-oh run times of the algorithms in Columns 6, 8, 11, 12, 13, 14 and 15 to estimate the run time of their implementation as programs. Compare your estimates to the experiments reported in the columns.

9. Suppose that a system makes 100 disk accesses to process a transaction (although some systems need fewer, some systems require several hundred disk accesses per transaction). How many transactions per hour per disk can the system handle?

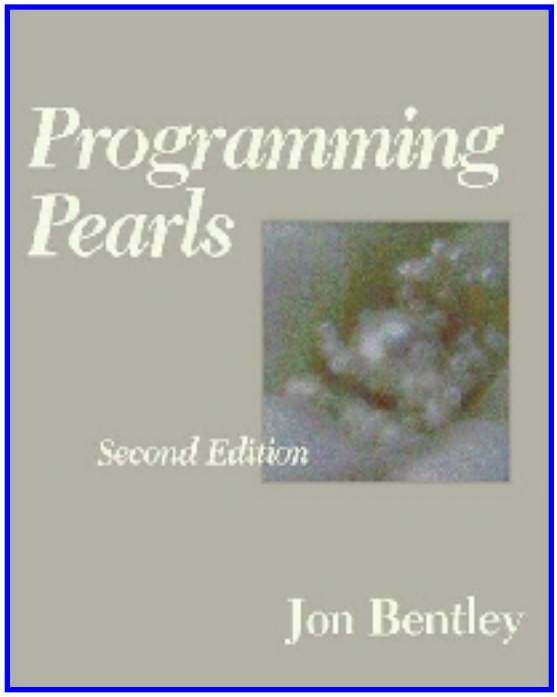
10. Estimate your city's death rate, measured in percent of population per year.

11. [P. J. Denning] Sketch a proof of Little's Law.

12. You read in a newspaper article that a United States quarter-dollar coin has ``an average life of 30 years''. How can you check that claim?

Next: Section 7.7. Further Reading.

Further Reading (Section 7.7 of Programming Pearls)



My all-time favorite book on common sense in mathematics is Darrel Huff's 1954 classic *How To Lie With Statistics*; it was reissued by Norton in 1993. The examples are now quaint (some of those rich folks make a whopping twenty-five thousand dollars per year!), but the principles are timeless. John Allen Paulos's *Innumeracy: Mathematical Illiteracy and Its Consequences* is a 1990 approach to similar problems (published by Farrar, Straus and Giroux).

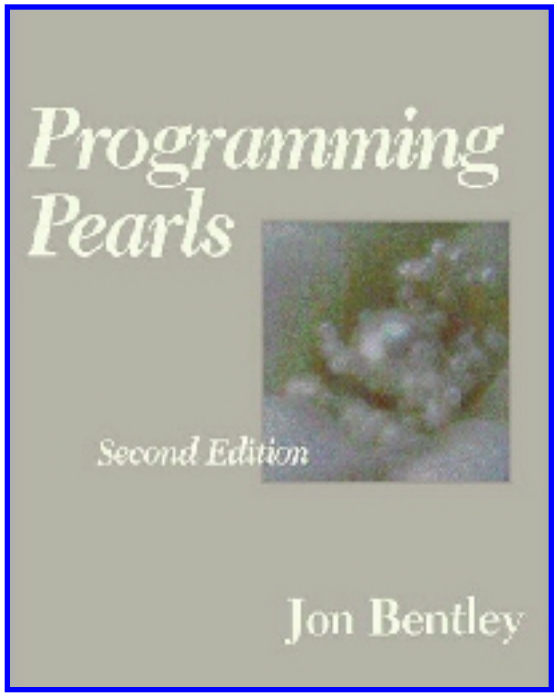
Physicists are well aware of this topic. After this column appeared in *Communications of the ACM*, Jan Wolitzky wrote

> I've often heard ``back-of-the-envelope'' calculations referred to as ``Fermi approximations'', after the physicist. The story is that Enrico Fermi, Robert Oppenheimer, and the other Manhattan Project brass were behind a low blast wall awaiting the detonation of the first nuclear device from a few thousand yards away. Fermi was tearing up sheets of paper into little pieces, which he tossed into the air when he saw the flash. After the shock wave passed, he paced off the distance travelled by the paper shreds, performed a quick ``back-of-the-envelope'' calculation, and arrived at a figure for the explosive yield of the bomb, which was confirmed much later by expensive monitoring equipment.

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

Next: Section 7.8. Quick Calculations in Everyday Life.

Quick Calculations in Everyday Life (Section 7.8 of Programming Pearls)



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.

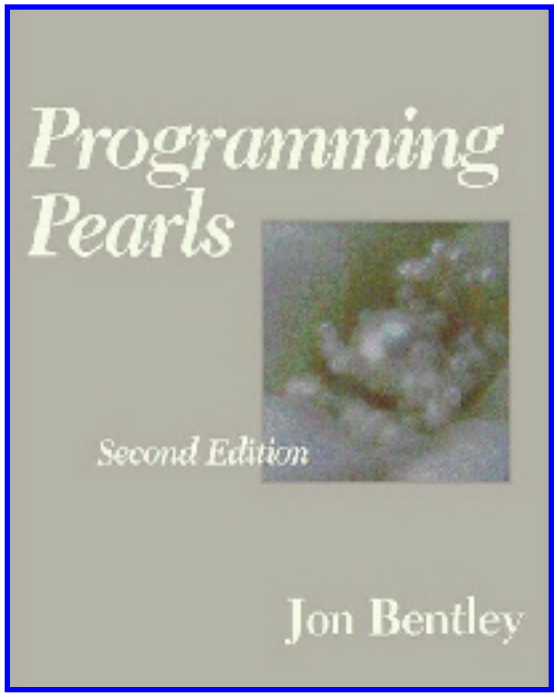
Everyday life presents us with many opportunities to hone our skills at quick calculations. For instance, how much money have you spent in the past year eating in restaurants? I was once horrified to hear a New Yorker quickly compute that he and his wife spend more money each month on taxicabs than they 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.

Solutions (To Column 7 of Programming Pearls)



These solutions include guesses at constants that may be off by a factor of two from their correct values as this book goes to press, but not much further.

1. The Passaic River does not flow at 200 miles per hour, even when falling 80 feet over the beautiful Great Falls in Patterson, New Jersey. I suspect that the engineer really told the reporter that the engorged river was flowing at 200 miles *per day*, five times faster than its typical 40 miles per day, which is just under a leisurely 2 miles per hour.

2. An old removable disk holds 100 megabytes. An ISDN line transmits 112 kilobits per second, or about 50 megabytes per hour. This gives a cyclist with a disk in his pocket about two hours to pedal, or a 15-mile radius of superiority. For a more interesting race, put a hundred DVDs in the cyclist's backpack, and his bandwidth goes up by a factor of 17,000; upgrade the line to ATM at 155 megabits per second, for a factor of 1400 increase. This gives the cyclist another factor of 12, or one day to pedal. (The day after I wrote this paragraph, I walked into a colleague's office to see 200 5-gigabyte write-once media platters in a pile on his desk. In 1999, a terabyte of unwritten media was a stunning sight.)

3. A floppy disk contains 1.44 megabytes. Flat out, my typing is about fifty words (or 300 bytes) per minute. I can therefore fill a floppy in 4800 minutes, or 80 hours. (The input text for this book is only half a megabyte, but it took me substantially longer than three days to type it.)

4. I was hoping for answers along the lines of a ten-nanosecond instruction takes a hundredth of a second, an 11 millisecond disk rotation (at 5400 rpm) takes 3 hours, a 20 msec seek takes 6 hours, and the two seconds to type my name takes about a month. A clever reader wrote, ``How long does it take? Exactly the same time as before, if the clock slows down, too."

5. For the rate between 5 and 10 percent, the Rule-of-72 estimate is accurate to within one percent.

6. Since 72/1.33 is approximately 54, we would expect the population to double by 2052 (the UN estimates happily call for the rate to drop substantially).

9. Ignoring slowdown due to queueing, 20 milliseconds (of seek time) per disk operation gives 2 seconds per transaction or 1800 transactions per hour.

10. One could estimate the local death rate by counting death notices in a newspaper and estimating the population of the area they represent. An easier approach uses Little's Law and an estimate of life expectancy; if the life expectancy is 70 years, for instance, then $1/70$ or 1.4% of the population dies each year.

11. Peter Denning's proof of Little's Law has two parts. ``First, define *lambda=A/T*, the arrival rate, where A is the number of arrivals during an observation period of length *T*. Define *X=C/T*, the output rate, where *C* is the number of completions during *T*. Let *n(t)* denote the number in the system at time *t* in [0,*T*]. Let *W* be the area under *n(t)*, in units of `item-seconds', representing the total aggregated waiting time over all items in the system during the observation period. The mean response time per item completed is defined as *R=W/C*, in units of (item-seconds)/(item). The mean number in the system is the average height of n(t) and is *L = W/T*, in units of (item-seconds)/(second). It is now obvious that *L=RX*. This formulation is in terms of the output rate only. There is no requirement for `flow balance', i.e., that flow in equal flow out (in symbols, *lambda=X*). If you add that assumption, the formula becomes *L=lambda*R*, which is the form encountered in queueing and system theory."

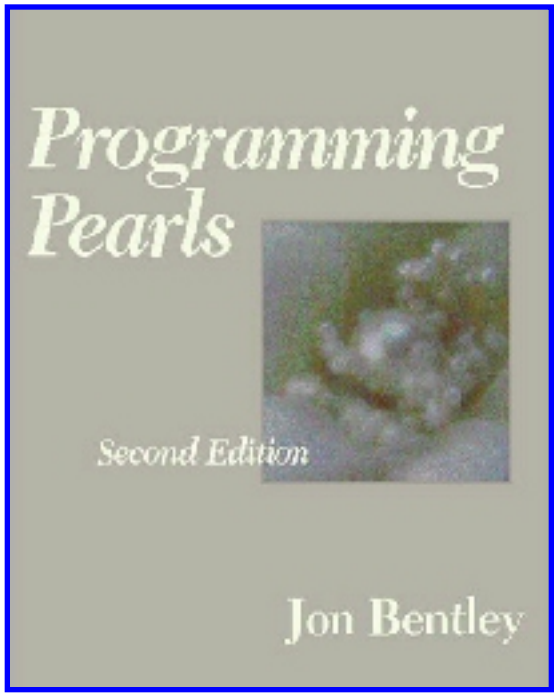
12. When I read that a quarter has ``an average life of 30 years", that number seemed high to me; I didn't recall seeing that many old quarters. I therefore dug into my pocket, and found a dozen quarters. Here are their ages in years:

3 4 5 7 9 9 12 17 17 19 20 34

The average age is 13 years, which is perfectly consistent with a quarter having an average life of about twice that (over this rather uniform distribution of ages). Had I come up with a dozen quarters all less than five years old, I might have dug further into the topic. This time, though, I guessed that the paper had got it right.

The same article reported that ``there will be a minimum of 750 million New Jersey quarters made" and also that a new state quarter would be issued every 10 weeks. That multiplies to about four billion quarters per year, or a dozen new quarters for each resident of the United States. A life of 30 years per quarter means that each person has 360 quarters out there. That is too many for pockets alone, but it is plausible if we include piles of change at home and in cars, and a lot in cash registers and banks.

Cost Models for Time and Space (Appendix 3 of Programming Pearls)



Section 7.2 describes two little programs for estimating the time and space consumed by various primitive operations. This appendix shows how those can grow into useful programs for generating one-page time and space estimates. The complete source code for both programs can be found at this book's web site.

The program spacemod.cpp produces a model of the space consumed by various constructs in C++. The first part of the program uses a sequence of statements like

```
cout << "sizeof(char)=" << sizeof(char);
cout << " sizeof(short)=" << sizeof(short);
```

to give precise measurements of primitive objects:

```
sizeof(char)=1  sizeof(short)=2  sizeof(int)=4
sizeof(float)=4  sizeof(struct *)=4  sizeof(long)=4
sizeof(double)=8
```

The program also defines a dozen structures, using the simple naming convention illustrated in these examples:

```
struct structc { char c; };
struct structic { int i; char c; };
struct structip { int i; structip *p; };
struct structdc { double d; char c; };
struct structc12 { char c[12]; };
```

The program uses a macro to report the *sizeof* the structure, and then to estimate the number of bytes that *new* allocates in a report like this:

```
structc    1   48 48 48 48 48 48 48 48 48 48
structic   8   48 48 48 48 48 48 48 48 48 48
structip   8   48 48 48 48 48 48 48 48 48 48
structdc   16  64 64 64 64 64 64 64 64 64 64
structcd   16  64 64 64 64 64 64 64 64 64 64
structcdc  24  -3744 4096 64 64 64 64 64 64 64 64
structiii  12  48 48 48 48 48 48 48 48 48 48
```

The first number is given by *sizeof*, and the next ten numbers report the differences between successive pointers returned by *new*. This output is typical: most of the numbers are quite consistent, but the allocator skips around every now and then.

This macro makes one line of the report:

```
#define MEASURE(T, text) {                \
    cout << text << "\t";                 \
    cout << sizeof(T) << "\t";            \
    int lastp = 0;                        \
    for (int i = 0; i < 11; i++) {        \
        T *p = new T;                     \
        int thisp = (int) p;              \
        if (lastp != 0)                   \
            cout << " " << thisp - lastp; \
        lastp = thisp;                    \
    }                                     \
    cout << "\n";                         \
}
```

The macro is called with the structure name followed by the same name in quotes, so it can be printed:

```
MEASURE(structc, "structc");
```

(My first draft used a C++ template parameterized by structure type, but measurements were led wildly astray by artifacts of the C++ implementation.)

This table summarizes the output of the program on my machine:

STRUCTURE	***sizeof***	***new* size**
int	4	48
structc	1	48
structic	8	48
structip	8	48
structdc	16	64
structcd	16	64
structcdc	24	64
structiii	12	48
structiic	12	48
structc12	12	48
structc13	13	64
structc28	28	64
structc29	29	80

The left column of numbers helps us to estimate the *sizeof* structures. We start by summing the *sizeof* the types;

that explains the 8 bytes for *structip*. We must also account for the alignment; although its components sum to 10 bytes (two *char*s and a *double*), *structcdc* consumes 24 bytes.

The right column gives us insight into the space overhead of the *new* operator. It appears that any structure with a *sizeof* 12 bytes or less will consume 48 bytes. Structures with 13 through 28 bytes consume 64 bytes. In general, the allocated block size will be a multiple of 16, with 36 to 47 bytes of overhead. This is surprisingly expensive; other systems that I use consume just 8 bytes of overhead to represent an 8-byte record.

Section 7.2 also describes a little program for estimating the cost of one particular C operation. We can generalize that to the timemod.c program that produces a one-page cost model for a set of C operations. (Brian Kernighan, Chris Van Wyk and I built a predecessor of this program in 1991.) The *main* function of that program consists of a sequence of a *T* (for title) lines followed by *M* lines to measure the cost of operations:

```
T("Integer Arithmetic");
M({});
M(k++);
M(k = i + j);
M(k = i - j);
    ...
```

Those lines (and a few more like them) produce this output:

```
Integer Arithmetic (n=5000)
 {}           250   261   250   250   251   10
 k++          471   460   471   461   460   19
 k = i + j    491   491   500   491   491   20
 k = i - j    440   441   441   440   441   18
 k = i * j    491   490   491   491   490   20
 k = i / j   2414  2433  2424  2423  2414   97
 k = i % j   2423  2414  2423  2414  2423   97
 k = i & j    491   491   480   491   491   20
 k = i | j    440   441   441   440   441   18
```

The first column gives the operation that is executed inside the loop

```
for i = [1, n]
    for j = [1, n]
        op
```

The next five columns show the raw time in clock clicks (milliseconds on this system) for five executions of that loop (these times are all consistent; inconsistent numbers help to identify suspicious runs). The final column gives the average cost in nanoseconds per operation. The first row of the table says that it takes about ten nanoseconds to execute a loop that contains the null operation. The next row says that incrementing the variable *k* consumes about 9 additional nanoseconds. All of the arithmetic and logical operations have about the same cost, with the exception of the division and remainder operators, which are an order-of-magnitude more expensive.

This approach gives rough estimates for my machine, and they must not be over-interpreted. I conducted all

experiments with optimization disabled. When I enabled that option, the optimizer removed the entire timing loops and all times were zero.

The work is done by the *M* macro, which can be sketched in pseudocode as:

```
#define M(op)
    print op as a string
    timesum = 0
    for trial = [0, trials)
        start = clock()
        for i = [1, n]
            fi = i
            for j = [1, n]
                op
        t = clock()-start
        print t
        timesum += t
    print 1e9*timesum / (n*n * trials * CLOCKS_PER_SEC)
```

The complete code for this cost model can be found at this book's web site.

We'll now survey the output of the program on my particular system. Because the clock clicks are all consistent, we will omit them and report only the average time in nanoseconds.

```
Floating Point Arithmetic (n=5000)
 fj=j;                  18
 fj=j; fk = fi + fj     26
 fj=j; fk = fi - fj     27
 fj=j; fk = fi * fj     24
 fj=j; fk = fi / fj     78
Array Operations (n=5000)
 k = i + j              17
 k = x[i] + j           18
 k = i + x[j]           24
 k = x[i] + x[j]        27
```

The floating point operations originally assign the integer *j* to the floating-point *fj* (in about 8 nanoseconds); the outer loop assigns *i* to the floating-point *fi*. The floating operations themselves cost about as much as their integer counterparts, and the array operations are equally inexpensive.

The next tests give us insight into control flow in general and some sorting operations in particular:

```
Comparisons (n=5000)
 if (i < j) k++           20
 if (x[i] < x[j]) k++     25
Array Comparisons and Swaps (n=5000)
 k = (x[i]<x[k]) ? -1:1   34
 k = intcmp(x+i, x+j)     52
```

```
 swapmac(i, j)            41
 swapfunc(i, j)           65
```

The function versions of comparing and swapping each cost about 20 nanoseconds more than their inline counterparts. Section 9.2 compares the cost of computing the maximum of two values with functions, macros and inline code:

```
Max Function, Macro and Inline (n=5000)
 k = (i > j) ? i : j   26
 k = maxmac(i, j)      26
 k = maxfunc(i, j)     54
```

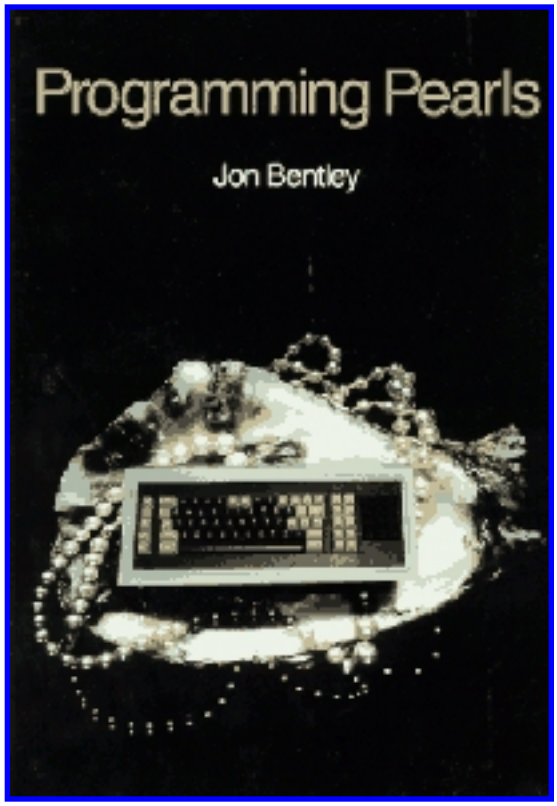
The *rand* function is relatively inexpensive (though recall that the *bigrand* function makes two calls to *rand*), square root is an order of magnitude greater than basic arithmetic operations (though only twice the cost of a division), simple trigonometric operations cost twice that, and advanced trigonometric operations run into microseconds.

```
Math Functions (n=1000)
 k = rand()          40
 fk = j+fi           20
 fk = sqrt(j+fi)    188
 fk = sin(j+fi)     344
 fk = sinh(j+fi)   2229
 fk = asin(j+fi)    973
 fk = cos(j+fi)     353
 fk = tan(j+fi)     465
```

Because those are so pricey, we shrunk the value of *n*. Memory allocation is more expensive yet, and merits an even smaller *n*:

```
Memory Allocation (n=500)
 free(malloc(16))       2484
 free(malloc(100))      3044
 free(malloc(2000))     4959
```

About the First Edition of Programming Pearls



The first edition of *Programming Pearls* was published by Addison-Wesley, Inc., 1986.
ISBN 0-201-10331-1.
195 + viii pp.

Translations

- Hungarian: *A Programozás Gyongyszemei*, Müszaki Könyvkiadó, ISBN 963-10-7681-4
- Japanese: *Programming Pearls*, Kindai Kagaku Sha Company, Ltd., ISBN 4-7649-0158-7
- Polish: *Peretki Oprogramowania*, Wydawnictwa Naukowo-Techniczne, ISBN 83-204-1405-9
- Russian: *Zhemchuzhinyi Tvorchestva Programmistov*, Radio i Svyaz', ISBN 5-256-00704-1
- Slovak: *Perly Programovania*, Alfa SNTL, ISBN 80-05001056-7

Web Reviews

I talked about how much I enjoyed the first edition when I described why I wrote a second edition. Other people like the book, too.

Steve McConnell put *Programming Pearls* on the Top-Ten List in his *Code Complete*. He says that the book is ``a vigorous discussion of programming that articulates the reasons that some of us find programming so interesting. The fact that the information isn't comprehensive or rigidly organized doesn't prevent it from conveying powerful insights that you'll read in a few minutes and use for many years."

In an Association of C and C++ Users Book Review Francis Glassborow says that it ``is another of those rare computing books that manages to stand the test of time" and that ``it has a place on every programmer's bookshelf".

Programming Pearls made the list of Great Books that Gregory V. Wilson prepared for Dr. Dobb's Electronic Review of Computer Books. The book is also described in Ray Duncan's Confessions of an Efficient Coder in the same series.

The first edition of *Programming Pearls* earned five stars at Amazon.com. The readers' comments include ``a gem", ``the small book is itself a pearl", ``a manual with hacker spirit!", and ``best book about how to `think' for software engineering".

K. N. King put *Programming Pearls* on his list of recommended books. ``The author's light touch makes the book as enjoyable to read as it is informative."

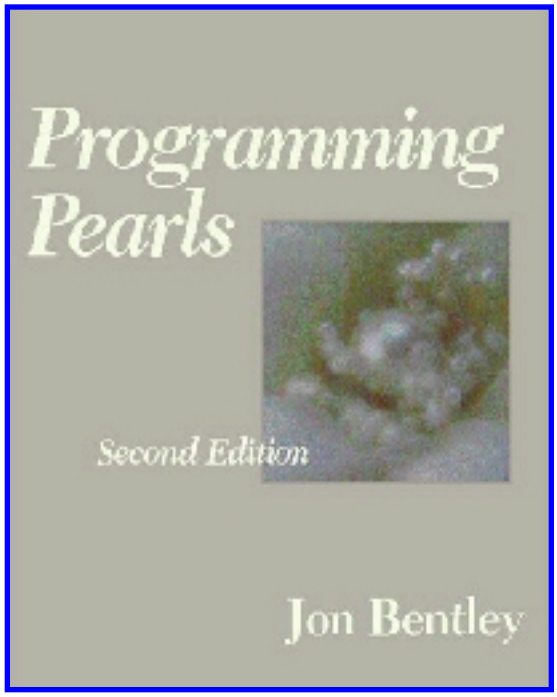
Other comments on *Programming Pearls*: enough fresh insights to make this a real bookshelf treasure [a book] that I think everyone should read [a book] that changed me lots of useful lessons in the craft an important, classic book clever language solutions to common programming problems one of the best books I know on general issues in programming list of recommended programming books very entertaining articles books I can recommend a brilliant book one of the most thoughtful gifts I have ever received

The Secret of Bill's Success?

From *Microsoft Secrets: How the World's Most Powerful Company Creates Technology, Shapes Markets, and Manages People*, by Michael Cusumano and Richard Selby (The Free Press, A Division of Simon & Schuster Inc., NY, NY, 1995), p. 112:

``The mentoring system? Actually, that's not broken.... There are certain things, certain algorithmic techniques and things. But give the guy a copy of *Programming Pearls*, make sure he's read through the thing and understands everything in there. Show him a piece of code. Ask him some questions about it...."

Performance (Part II of Programming Pearls)



A simple, powerful program that delights its users and does not vex its builders -- that is the programmer's ultimate goal and the emphasis of the five previous columns.

We'll turn our attention now to one specific aspect of delightful programs: efficiency. Inefficient programs sadden their users with long waits and missed opportunities. These columns therefore describe several paths to performance.

Column 6 surveys the approaches and how they interact. The three subsequent columns discuss three methods for improving run time, in the order in which they are usually applied:

- Column 7 shows how ``back-of-the-envelope'' calculations used early in the design process can ensure that the basic system structure is efficient enough.
- Column 8 is about algorithm design techniques that sometimes dramatically reduce the run time of a module.
- Column 9 discusses code tuning, which is usually done late in the implementation of a system.

To wrap up Part II, Column 10 turns to another important aspect of performance: space efficiency.

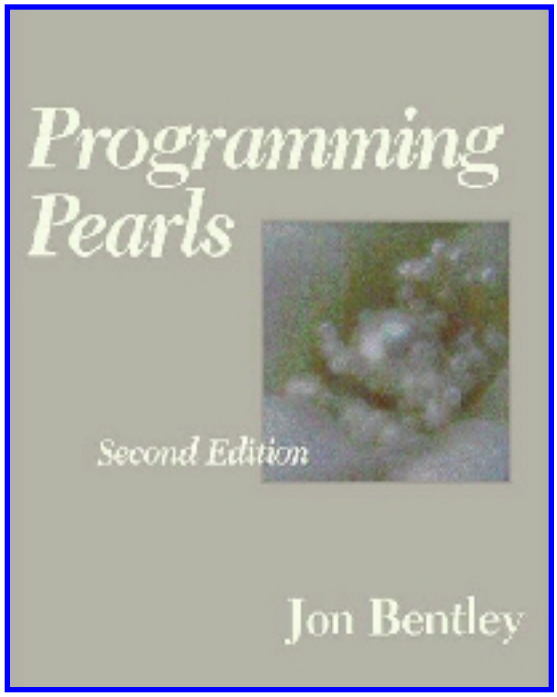
There are three good reasons for studying efficiency. The first is its intrinsic importance in many applications. I'm willing to bet that every reader of this book has at some time stared in frustration at a monitor, wishing fervently that the program were faster. A software manager I know estimates that half her development budget goes to performance improvement. Many programs have stringent time requirements, including real-time programs, huge database systems and interactive software.

The second reason for studying performance is educational. Apart from practical benefits, efficiency is a fine training ground. These columns cover ideas ranging from the theory of algorithms to common-sense techniques like ``back-of-the-envelope'' calculations. The major theme is fluidity of thinking; Column 6, especially, encourages us to look at a problem from many different viewpoints.

Similar lessons come from many other topics. These columns might have been built around user interfaces, system robustness or security. Efficiency has the advantage that it can be measured: we can all agree that one program is 2.5 times faster than another, while discussions on user interfaces, for instance, often get bogged down in personal tastes.

The most important reason for studying performance is described best in the immortal words of the 1986 film *Top Gun*: ``I feel the need ... the need for speed!"

Writing Correct Programs (A Sketch of Column 4 of Programming Pearls)



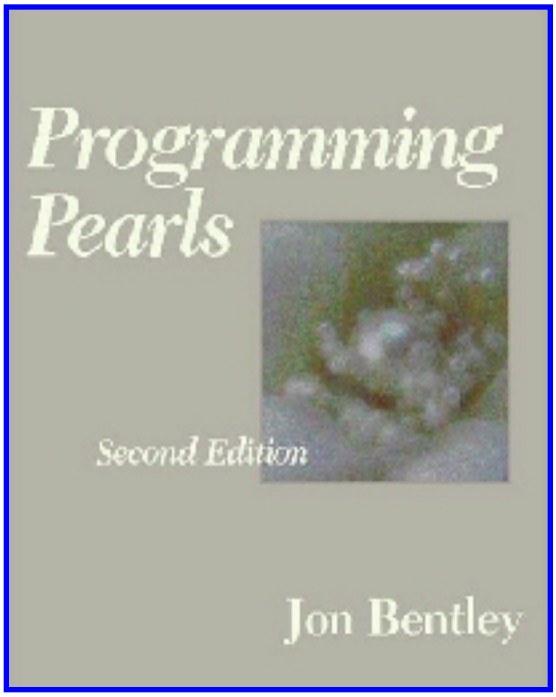
In the late 1960's people were talking about the promise of programs that verify the correctness of other programs. Unfortunately, in the intervening decades, with precious few exceptions, there is still little more than talk about automated verification systems. In spite of unrealized expectations, though, research on program verification has given us something far more valuable than a black box that gobbles programs and flashes ``good'' or ``bad'' -- we now have a fundamental understanding of computer programming.

The purpose of this column is to show how that fundamental understanding can help real programmers write correct programs. One reader characterized the approach that most programmers grow up with as ``write your code, throw it over the wall, and have Quality Assurance or Testing deal with the bugs.'' This column describes an alternative. Before we get to the subject itself, we must keep it in perspective. Coding skill is just one small part of writing correct programs. The majority of the task is the subject of the three previous columns: problem definition, algorithm design, and data structure selection. If you perform those tasks well, writing correct code is usually easy.

The Rest of the Column

The teaching material contains overhead transparencies based on this column; the slides are available in both Postscript and Acrobat.

Algorithm Design Techniques (A Sketch of Column 8 of Programming Pearls)



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.

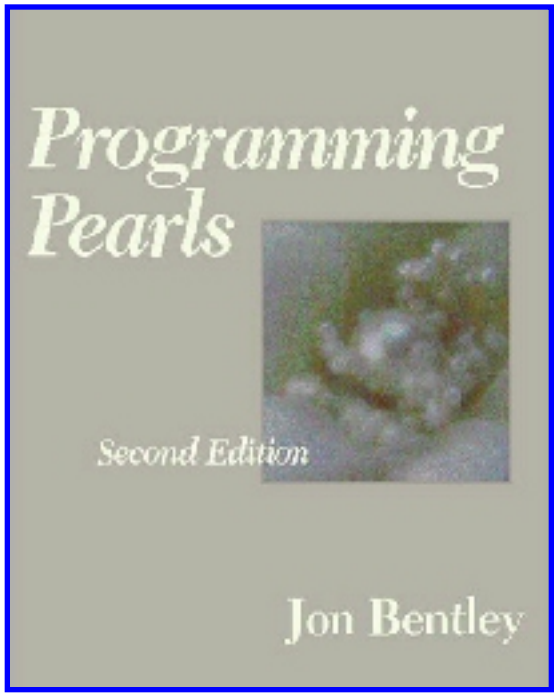
This column studies four different algorithms for one small problem, with an emphasis on the techniques used to design them. Some of the algorithms are a little complicated, but with justification. While the first program we'll study takes fifteen days to solve a problem of size 100,000, the final program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

The Rest of the Column

The teaching material contains overhead transparencies based on this column; the slides are available in both Postscript and Acrobat.

The code for Column 8 contains a program for testing and timing all the algorithms.

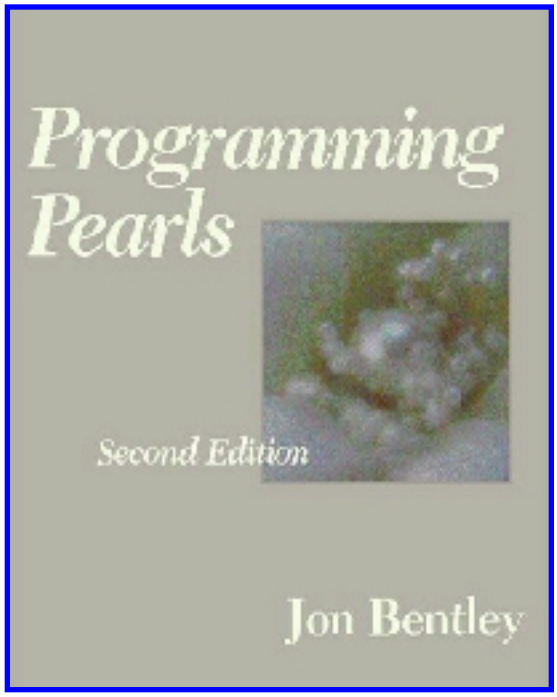
Programming Pearls Table of Contents



For Column 1 For Column 5 For Column 7 For Column 15

- Index

The Preface to Programming Pearls



Computer programming has many faces. Fred Brooks paints the big picture in *The Mythical Man Month*; his essays underscore the crucial role of management in large software projects. At a finer grain, Steve McConnell teaches good programming style in *Code Complete*. The topics in those books are the key to good software and the hallmark of the professional programmer. Unfortunately, though, the workmanlike application of those sound engineering principles isn't always thrilling -- until the software is completed on time and works without surprise.

About the Book

The columns in this book are about a more glamorous aspect of the profession: programming pearls whose origins lie beyond solid engineering, in the realm of insight and creativity. Just as natural pearls grow from grains of sand that have irritated oysters, these programming pearls have grown from real problems that have irritated real programmers. The programs are fun, and they teach important programming techniques and fundamental design principles.

Most of these essays originally appeared in my ``Programming Pearls'' column in *Communications of the Association for Computing Machinery*. They were collected, revised and published as the first edition of this book in 1986. Twelve of the thirteen pieces in the first edition have been edited substantially for this edition, and three new columns have been added.

The only background the book assumes is programming experience in a high-level language. Advanced techniques (such as templates in C++) show up now and then, but the reader unfamiliar with such topics will be able to skip to the next section with impunity.

Although each column may be read by itself, there is a logical grouping to the complete set. Columns 1 through 5 form Part I of the book. They review programming fundamentals: problem definition, algorithms, data structures and program verification and testing. Part II is built around the theme of efficiency, which is sometimes important in itself and is always a fine springboard into interesting programming problems. Part III applies those techniques to several substantial problems in sorting, searching and strings.

One hint about reading the essays: don't go too fast. Read them carefully, one per sitting. Try the problems as they are posed -- some of them look easy until you've butted your head against them for an hour or two. Afterwards, work hard on the problems at the end of each column: most of what you learn from this book will come out the end of your pencil as you scribble down your solutions. If possible, discuss your ideas with friends and colleagues before peeking at the hints and solutions in the back of the book. The further reading at the end of each chapter isn't intended as a scholarly reference list; I've recommended some good books that are an important part of my personal library.

This book is written for programmers. I hope that the problems, hints, solutions, and further reading make it useful

for individuals. The book has been used in classes including Algorithms, Program Verification and Software Engineering. The catalog of algorithms in Appendix 1 is a reference for practicing programmers, and also shows how the book can be integrated into classes on algorithms and data structures.

The Code

The pseudocode programs in the first edition of the book were all implemented, but I was the only person to see the real code. For this edition, I have rewritten all the old programs and written about the same amount of new code. The programs are available at this book's web site. The code includes much of the scaffolding for testing, debugging and timing the functions. The site also contains other relevant material. Because so much software is now available online, a new theme in this edition is how to evaluate and use software components.

The programs use a terse coding style: short variable names, few blank lines, and little or no error checking. This is inappropriate in large software projects, but it is useful to convey the key ideas of algorithms. Solution 5.1 gives more background on this style.

The text includes a few real C and C++ programs, but most functions are expressed in a pseudocode that takes less space and avoids inelegant syntax. The notation *for i = [0, n)* iterates *i* from 0 through *n-1*. In these *for* loops, left and right parentheses denote open ranges (which do not include the end values), and left and right square brackets denote closed ranges (which do include the end values). The phrase *function(i, j)* still calls a function with parameters *i* and *j*, and *array[i, j]* still accesses an array element.

This edition reports the run times of many programs on ``my computer'', a 400MHz Pentium II with 128 megabytes of RAM running Windows NT 4.0. I timed the programs on several other machines, and the book reports the few substantial differences that I observed. All experiments used the highest available level of compiler optimization. I encourage you to time the programs on your machine; I bet that you'll find similar ratios of run times.

To Readers of the First Edition

I hope that your first response as you thumb through this edition of the book is, ``This sure looks familiar.'' A few minutes later, I hope that you'll observe, ``I've never seen that before.''

This version has the same focus as the first edition, but is set in a larger context. Computing has grown substantially in important areas such as databases, networking and user interfaces. Most programmers should be familiar users of such technologies. At the center of each of those areas, though, is a hard core of programming problems. Those programs remain the theme of this book. This edition of the book is a slightly larger fish in a much larger pond.

One section from old Column 4 on implementing binary search grew into new Column 5 on testing, debugging and timing. Old Column 11 grew and split into new Columns 12 (on the original problem) and 13 (on set representations). Old Column 13 described a spelling checker that ran in a 64-kilobyte address space; it has been deleted, but its heart lives on in Section 13.8. New Column 15 is about string problems. Many sections have been inserted into the old columns, and other sections were deleted along the way. With new problems, new solutions, and four new appendices, this edition of the book is 25 percent longer.

Many of the old case studies in this edition are unchanged, for their historical interest. A few old stories have been

recast in modern terms.

Acknowledgments for the First Edition

I am grateful for much support from many people. The idea for a *Communications of the ACM* column was originally conceived by Peter Denning and Stuart Lynn. Peter worked diligently within ACM to make the column possible and recruited me for the job. ACM Headquarters staff, particularly Roz Steier and Nancy Adriance, have been very supportive as these columns were published in their original form. I am especially indebted to the ACM for encouraging publication of the columns in their present form, and to the many *CACM* readers who made this expanded version necessary and possible by their comments on the original columns.

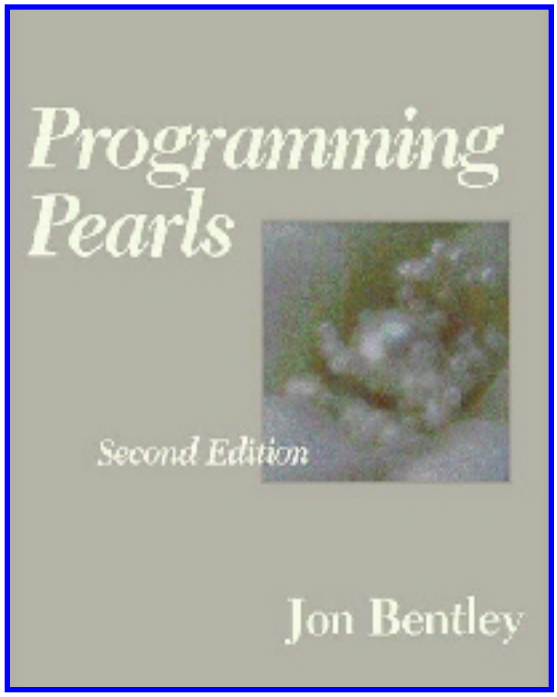
Al Aho, Peter Denning, Mike Garey, David Johnson, Brian Kernighan, John Linderman, Doug McIlroy and Don Stanat have all read each column with great care, often under extreme time pressure. I am also grateful for the particularly helpful comments of Henry Baird, Bill Cleveland, David Gries, Eric Grosse, Lynn Jelinski, Steve Johnson, Bob Melville, Bob Martin, Arno Penzias, Marilyn Roper, Chris Van Wyk, Vic Vyssotsky and Pamela Zave. Al Aho, Andrew Hume, Brian Kernighan, Ravi Sethi, Laura Skinger and Bjarne Stroustrup provided invaluable help in bookmaking, and West Point cadets in EF 485 field tested the penultimate draft of the manuscript. Thanks, all.

Acknowledgments for the Second Edition

Dan Bentley, Russ Cox, Brian Kernighan, Mark Kernighan, John Linderman, Steve McConnell, Doug McIlroy, Rob Pike, Howard Trickey and Chris Van Wyk have all read this edition with great care. I am also grateful for the particularly helpful comments of Paul Abrahams, Glenda Childress, Eric Grosse, Ann Martin, Peter McIlroy, Peter Memishian, Sundar Narasimhan, Lisa Ricker, Dennis Ritchie, Ravi Sethi, Carol Smith, Tom Szymanski and Kentaro Toyama. I thank Peter Gordon and his colleagues at Addison-Wesley for their help in preparing this edition.

J. B.
Murray Hill, New Jersey
December, 1985
August, 1999

Preliminaries (Part I of Programming Pearls)

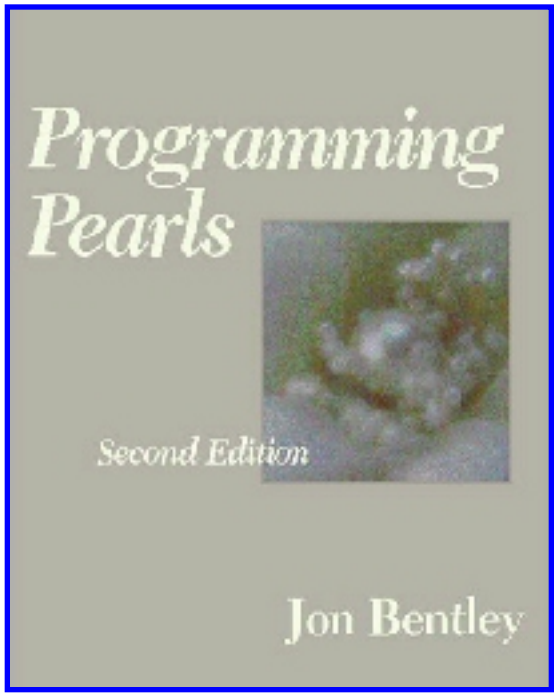


These five columns review the basics of programming. Column 1 is the history of a single problem. A combination of careful problem definition and straightforward programming techniques led to an elegant solution. The case illustrates the central theme of this book: thinking hard about a real case study can be fun and can also lead to practical benefits.

Column 2 examines three problems, with an emphasis on how algorithmic insights can yield simple and effective code. Column 3 surveys the crucial role that the structure of data can play in software design.

Column 4 introduces program verification as a tool for writing correct code. Verification techniques are used extensively as we derive subtle (and fast) functions in Columns 9, 11 and 14. Column 5 shows how we implement those abstract programs in real code, using scaffolding to probe a function, to bombard it with test cases, and to measure its performance.

Aha! Algorithms (A Sketch of Column 2 of Programming Pearls)



The study of algorithms offers much to the practicing programmer. A course on the subject equips students with functions for important tasks and techniques for attacking new problems. We'll see in later columns how advanced algorithmic tools sometimes have a substantial influence on software systems, both in reduced development time and in faster execution speed.

As crucial as those sophisticated ideas are, algorithms have a more important effect at a more common level of programming. In his book *Aha! Insight* (from which I stole my title), Martin Gardner describes the contribution I have in mind: ``A problem that seems difficult may have a simple, unexpected solution.'' Unlike the advanced methods, the *aha!* insights of algorithms don't come only after extensive study; they're available to any programmer willing to think seriously before, during and after coding.

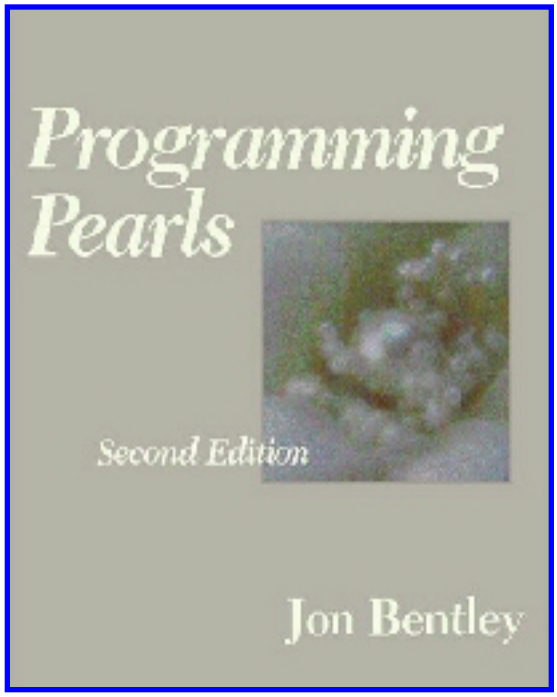
The Rest of the Column

2.1 Three Problems
2.2 Ubiquitous Binary Search
2.3 The Power of Primitives
2.4 Getting It Together: Sorting
2.5 Principles
2.6 Problems
2.7 Further Reading
2.8 Implementing an Anagram Program

The teaching material contains overhead transparencies describing the vector rotation algorithm in Section 2.3; the slides are available in both Postscript and Acrobat. It also contains material on the anagram algorithm in Sections 2.4 and 2.8; the slides are available in both Postscript and Acrobat.

The code for Column 2 contains the programs for generating anagrams and a program for timing several vector rotation algorithms.

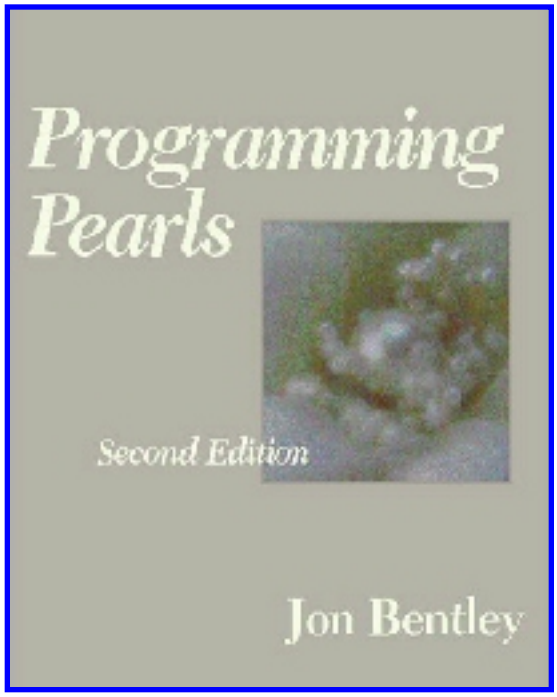
The Product (Part III of Programming Pearls)



Now comes the fun. Parts I and II laid a foundation; the next five columns use that material to build interesting programs. The problems are important in themselves, and they provide focal points where the techniques of previous columns converge in real applications.

Column 11 describes several general-purpose sorting algorithms. Column 12 describes a particular problem from a real application (generating a random sample of integers), and shows how it can be attacked in a variety of ways. One approach is to view it as a problem in set representation, which is the subject of Column 13. Column 14 introduces the heap data structure, and shows how it yields efficient algorithms for sorting and for priority queues. Column 15 tackles several problems that involve searching for words or phrases in very long text strings.

Heaps
(A Sketch of Column 14 of Programming Pearls)



This column is about ``heaps'', a data structure that we'll use to solve two important problems.

> *Sorting*. Heapsort never takes more than O(*n* log *n*) time to sort an *n*-element array, and uses just a few words of extra space.
> *Priority Queues*. Heaps maintain a set of elements under the operations of inserting new elements and extracting the smallest element in the set; each operation requires O(log *n*) time.

For both problems, heaps are simple to code and computationally efficient.

This column has a bottom-up organization: we start at the details and work up to the big picture. The next two sections describe the heap data structure and two functions to operate on it. The two subsequent sections use those tools to solve the problems mentioned above.

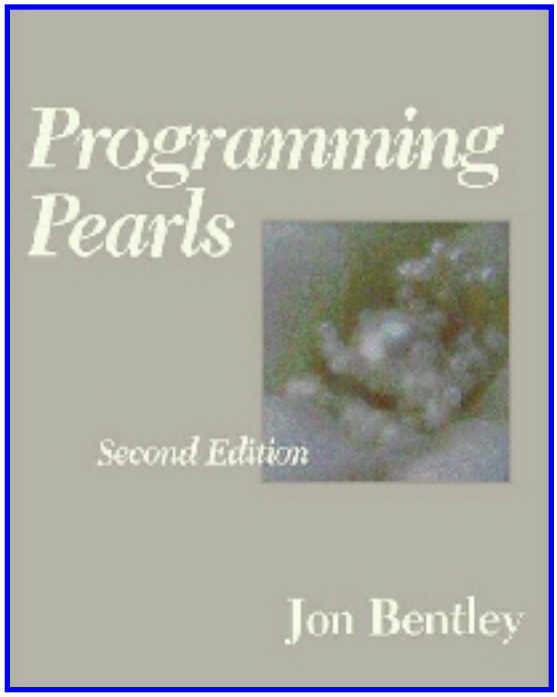
The Rest of the Column

The teaching material contains overhead transparencies based on this column; the slides are available in both Postscript and Acrobat.

The code for Column 14 contains the code for priority queues, and the code for Column 11 contains the code for Heapsort.

The animations of sorting algorithms show Heapsort at work. (But heed the warning on that page that the sleazy implementation leaves the first element of the array dangling in mid-air.)

Solutions (To Column 5 of Programming Pearls)



1. When I write large programs, I use long names (ten or twenty characters) for my global variables. This column uses short variable names such as *x*, *n* and *t*. In most software projects, the shortest plausible names might be more like *elem*, *nelems* and *target*. I find the short names convenient for building scaffolding and essential for mathematical proofs like that in Section 4.3. Similar rules apply to mathematics: the unfamiliar may want to hear that ``the square of the hypotenuse of a right triangle is equal to the sum of the squares of the two adjacent sides", but people working on the problem usually say ``$a^2 + b^2 = c^2$".

I've tried to stick close to Kernighan and Ritchie's C coding style, but I put the first line of code with the opening curly brace of a function and delete other blank lines to save space (a substantial percentage, for the little functions in this book).

The binary search in Section 5.1 returns an integer that is -1 if the value is not present, and points to the value if it is present. Steve McConnell suggested that the search should properly return two values: a *boolean* telling whether it is present, and an index that is used only if the *boolean* is true:

```
boolean BinarySearch(DataType TargetValue, int *TargetIndex)
 /* precondition: Element[0] <= Element[1] <=
        ... <= Element[NumElements-1]
    postcondition:
        result == false =>
            TargetValue not in Element[0..NumElements-1]
        result == true  =>
            Element[*TargetIndex] == TargetValue
  */
```

Listing 18.3 on page 402 of McConnell's *Code Complete* is a Pascal Insertion Sort that occupies one (large) page; the code and comments are together 41 lines. That style is appropriate for large software projects. Section 11.1 of this book presents the same algorithm in just five lines of code.

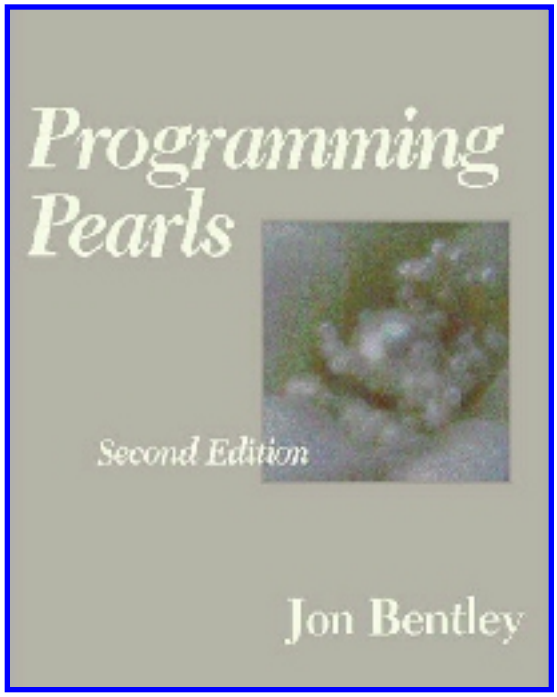
Few of the programs have error checking. Some functions read data from files into arrays of size *MAX*, and *scanf* calls can easily overflow their buffers. Array arguments that should be parameters are instead global variables.

Throughout this book, I've used shortcuts that are appropriate for textbooks and scaffolding, but not for large software projects. As Kernighan and Pike observe in Section 1.1 of their *The Practice of Programming*, ``clarity is often achieved through brevity". Even so, most of my code avoids the incredibly dense style illustrated in the C++

code in Section 14.3.

7. For n=1000, searching through the array in sorted order required 351 nanoseconds per search, while searching it in random order raised the average cost to 418 nanoseconds (about a twenty percent slowdown). For $n=10^6$, the experiment overflows even the L2 cache and the slowdown is a factor of 2.7. For the highly-tuned binary search in Section 8.3, though, the ordered searches zipped right through an n=1000-element table in 125 nanoseconds each, while the random searches required 266 nanoseconds, a slowdown of over a factor of two.

Sorting Algorithm Animations from Programming Pearls



Column 11 describes Insertion Sort and two Quicksorts (a simple version and the typical version using 2-way partitioning). Column 14 describes Heapsort. This page animates those algorithms and two additional simple sorting algorithms from the Solutions to Column 11: Selection Sort and Shell Sort.

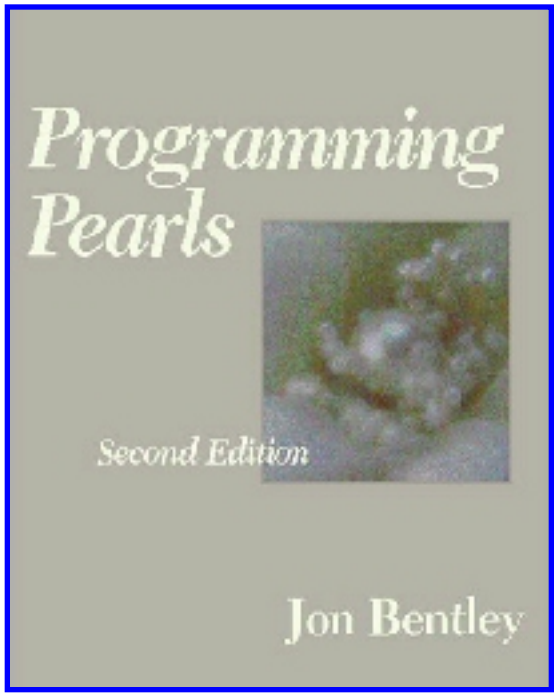
To view an animation, hit the run button below. Each dot represents an element in the array; the x-value is the element's index and the y-value is the element's value. Thus a random array is a uniform smear of points, and a sorted array is a trail of points wandering from the bottom left to the top right.

To experiment with the algorithms, select a value of n, an input distribution, and a sorting algorithm before hitting run. For each algorithm, be sure to start with a small value of n, and work up to larger values. WARNING! There is no way to stop an animation in progress (this is a simple Java program) other than by terminating your browser. Enjoy!

This animation shows the essential algorithms, but has few bells and whistles. Most of the algorithms sort the array a[0..n-1]. **BEWARE!** The Heapsort takes a sleazy shortcut around the problem of zero-indexed arrays: it sorts the array a[1..n-1], and leaves a[0] dangling in mid-air.

The code for this animation is SortAnim.java, available with the rest of the source code.

Why A Second Edition of Programming Pearls?



In a draft Preface for the second edition, I explained why (and how) I undertook writing this version. The reviewers were unanimous about that Preface: a subtle reviewer said that it was ``more effusive'' than the rest of the book, while a more forthright reviewer said only, ``so little space, so many cliches''. They were right; it is gone from the book, replaced by a straightforward description of the changes. But most of the draft preface now lives on in this tiny corner of the web, complete with effusive cliches.

A long time ago, in a Preface far, far away ...

I started writing this second edition as a duty, admittedly a labor of love, but still recalling full well the tribulations of producing a book. It was indeed all that, but more. It became a joy. It became a visit to old friends, seeing that while a few had fallen with time, most had grown into a proud maturity. And, oh, the delight of the next generation! The new sections and the new problems have the strong family resemblance, but they have the vitality of youth, and a freshness of their own.

But I'm getting ahead of myself; let me return to the beginning. I wrote my first program in high school, in 1969. The first magazine I saw about computing was *Communications of the ACM*. In 1983, I started to write a *CACM* column called ``Programming Pearls''. It was about about programs whose origins lie beyond solid engineering, in the realm of insight and creativity. A few years later, I collected the most fundamental columns into the first edition of this book.

It is hard for any programming book to survive the tremendous changes in our field, and I watched other books I had written enjoy their years in the sun and then go gently into that good night. This book was different: the examples were showing their age, but the principles seemed to stand the test of time. What to do?

I hemmed and hawed, consistently finding reasons not to undertake the hard work. Things finally came together in early 1999: I came across reviews with words like ``classic but dated'', respected colleagues asked about updating the book, and I had a chunk of free time in my work schedule. It was time to revisit the salt mines.

Following my preferred style of software development, I decided to prototype a single column. I chose Column 8 on algorithm design techniques. The techniques were still relevant, but the implementations were painfully outdated (the run times were reported for the ancient VAX-11/750, and the code was not fit for public consumption). I had great fun recoding the algorithms in a modern style, complete with scaffolding for testing and timing. The tests reported a bug; were my old experiments flawed? No, but Problem 8.7 now describes an interesting aspect of floating-point computation. One tune-up I tried reduced the time of Algorithm 4 by 40 percent, but increased the time of Algorithm 3 by a factor of 100. Fascinating problems started trickling in at a steady rate.

My first real shock came when I prepared a table of the run times of four algorithms. The times were *almost* unchanged across fifteen years:

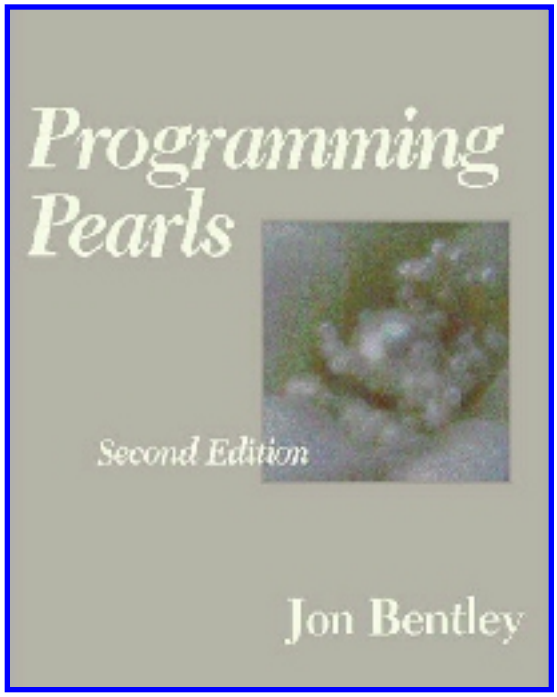
ALGORITHM	**1**	**2**	**3**	**4**
First edition	$3.4n^3$	$13n^2$	$46n \log_2 n$	$33n$
Second edition	$1.3n^3$	$10n^2$	$47n \log_2 n$	$48n$

I was astonished that the coefficients were so close. The only difference is that the top row reports the run time in *microseconds*, while the bottom row reports the time in *nanoseconds*! For these functions, my 400MHz Pentium II is almost exactly one thousand times faster than the venerable VAX. The more things change....

I next dug into Column 2. In 1983, I was able to find all anagrams in a dictionary of 72,000 words in half an hour. Now, a dictionary of 230,000 words was processed in 18 seconds. Problem 2.4 had revealed fascinating paging behavior on the old VAX, and now led to the remarkable graph of Pentium II caching performance in Solution 2.4.

I tried a few more columns, and each one led to surprises. The ancient program in Column 1 for sorting 27,000 political districts in a kilobyte turned into a modern program for sorting ten million toll-free telephone numbers in a megabyte. The old story in Section 3.5 about a new-fangled (for 1983) tool called a database turned into a little essay about hypertext, spreadsheets, databases, and other tools now familiar to high-school students. A section from old Column 4 on implementing binary search grew into a fun program and then into the new Column 5. I was hooked. I called Peter Gordon, my editor at Addison Wesley Longman, and enthusiastically committed to write this second edition.

Epilog to the Second Edition of Programming Pearls



Some traditions are continued for their inherent quality. Others persist anyway.

Q: Welcome back; it's been a long time.

A: Fourteen years.

Q: Let's start where we did before. Why a new edition of the book?

A: I like the book, a lot. It was fun to write, and the readers have been very kind over the years. The principles in the book have stood the test of time, but many examples in the first edition were woefully out of date. Modern readers can't relate to a ``huge" computer that has a half-megabyte of main memory.

Q: So what did you change in this edition?

A: Pretty much what I said I changed in the preface. Don't you guys prepare before these interviews?

Q: Oops -- sorry. I see that you talk there about how the code for this book is available at the web site.

A: Writing that code was the most fun that I had in working on this edition. I implemented most of the programs in the first edition, but mine were the only eyes to see the real code. For this edition I wrote about 2500 lines of C and C++, to be shown to the whole world.

Q: You call that code ready for public view? I've read some of it; what dreadful style! Microscopic variable names, weird function definitions, global variables that should be parameters, and the list goes on. Aren't you embarrassed to let real software engineers see the code?

A: The style I used can indeed prove fatal in large software projects. This book, however, is not a large software project. It's not even a large book. Solution 5.1 describes the terse coding style and why I chose it. Had I wanted to write a thousand-page book, I would have adopted a lengthier coding style.

Q: Speaking of long code, your *sort.cpp* program measures the C Standard Library *qsort*, the C++ Standard Template Library *sort*, and several hand-made Quicksorts. Can't you make up your mind? Should a programmer use library functions or build code from scratch?

A: Tom Duff gave the best answer to that question: ``Whenever possible, steal code." Libraries are great; use them whenever they do the job. Start with your system library, then search other libraries for appropriate functions. In

any engineering activity, though, not all artifacts can be all things to all customers. When the library functions don't measure up, you may have to build your own. I hope that the pseudocode fragments in the book (and the real code on the web site) will prove a useful starting point for programmers who have to write their own functions. I think that the scaffolding and the experimental approach of this book will help those programmers to evaluate a variety of algorithms and choose the best one for their application.

Q: Apart from the public code and updating some stories, what is *really* new in this edition?

A: I've tried to confront code tuning in the presence of caches and instruction-level parallelism. At a larger level, the three new columns reflect three major changes that pervade this edition: Column 5 describes real code and scaffolding, Column 13 gives details on data structures, and Column 15 derives advanced algorithms. Most of the ideas in the book have appeared in print before, but the cost model for space in Appendix 3 and the Markov-text algorithm in Section 15.3 are presented here for the first time. The new Markov-text algorithm compares quite favorably to the classic algorithm described by Kernighan and Pike.

Q: There you go with more Bell Labs people. The last time we talked, you were enthusiastic about the place, but you had only been there a few years. Lots has changed at the Labs in the last 14 years; what do you now think about the place and the changes?

A: When I wrote the first columns in the book, Bell Labs was part of the Bell System. When the first edition was published, we were part of AT&T; now we are part of Lucent Technologies. The companies, the telecommunications industry, and the field of computing have all changed dramatically in that period. Bell Labs has kept up with those changes, and has often led the way. I came to the Labs because I enjoy balancing the theoretical and the applied, because I want to build products and write books. The pendulum has swung back and forth during my years at the Labs, but my management has always encouraged a wide range of activities.

A reviewer of the first edition of this book wrote ``Bentley's everyday working environment is a programming nirvana. He is a Member of Technical Staff at Bell Labs in Murray Hill, New Jersey, has immediate access to cutting-edge hardware and software technology, and stands in line at the cafeteria with some of the most brilliant software developers in the world." Bell Labs is still that kind of place.

Q: Nirvana every single day of the week?

A: Nirvana many days, and pretty darn nice the others.