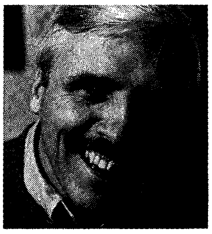
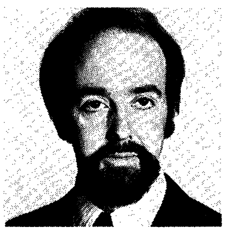
Fibonacci Numbers and Computer Algorithms

John Atkins
Robert Geist



John Atkins is an Associate Professor of Computer Science at West Virginia University. He received an M.A. in Mathematics and an M.S. in Computer Science from West Virginia University, and a Ph.D. in Mathematics from the University of Pittsburgh. Dr. Atkins also has an Oak Ridge Associated Universities appointment with the DOE's Morgantown Energy and Technology Center. He has published papers in Modeling and Simulation, Database Design, and Topology.



Robert Geist is an Associate Professor of Computer Science at Clemson University. He received an M.A. in computer science from Duke University, and a Ph.D. in mathematics from the University of Notre Dame. Professor Geist taught mathematics at Pembroke State University and computer science at Duke University prior to his appointment at Clemson. He has published in both computer science (reliability and performance evaluation) and mathematics (algebraic topology).

Leonard Fibonacci (a.k.a. Leonard Pisano, son of Bonaccio) would be surprised to learn that his famous sequence

$$(F_n = F_{n-1} + F_{n-2}, \text{ where } F_1 = F_2 = 1),$$

defined in the thirteenth century, has distinctly twentieth-century applications. For it is in the twentieth century, with the advent of digital computers, that the ancient study of algorithms has assumed important new significance. Computer scientists have discovered and used many algorithms which can be classified as applications of Fibonacci's sequence. In this article, we consider several of these applications.

Finding Extrema

A function f is *unimodal* in an interval $[a, b]$ if f has a unique maximum x_0 in the interval, and f is strictly increasing to the left of x_0 and strictly decreasing to the right of x_0. Consider now the problem of locating this unique extremum. Standard iterative algorithms attempt to reduce the interval under consideration by choosing two arbitrary intermediate points $x_1, x_2 (a < x_1 < x_2 < b)$, examining the value of the function at x_1 and x_2, and then discarding one of the subintervals, $[a, x_1)$ or $(x_2, b]$. For example, if $f(x_1) < f(x_2)$, the interval $[a, x_1)$ can be excluded (see Figure 1) and all our attention can be focused on the reduced subinterval $[x_1, b]$, which still brackets x_0.

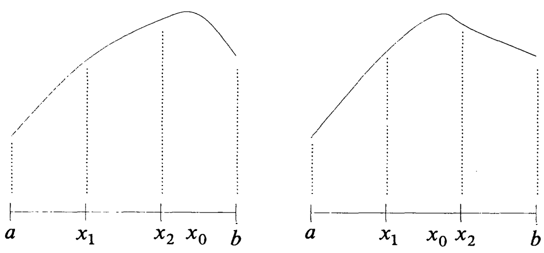


Figure 1.

The functions to which we apply such algorithms typically have rather complex definitions (if not, analytic techniques for locating extrema should be used), and so function evaluations are often computationally expensive. Thus, in selecting the two points from the reduced subinterval $[x_1, b]$ at which we will evaluate f, it is natural to hope that one of them could be the previously chosen x_2 at which f has already been evaluated. The problem, of course, is that an unfortunate original choice for x_2 could lead to an insignificant reduction at this stage in the size of our interval bracketing x_0. The real issue then is this:

Given that we want our final bracketing interval to be no larger than ϵ, how should we choose, at each stage, the intermediate points at which f is to be evaluated so that the total number of function evaluations is minimized?

The answer is due to Kiefer [11], who provided an optimal procedure based upon the Fibonacci sequence. First find the smallest Fibonacci number F_n such that $(b-a)/F_n < \epsilon$. We can then regard the interval $[a, b]$ as being divided into F_n subintervals, each of length $s = (b-a)/F_n$. Now let $x_1 = b - sF_{n-1}$ and $x_2 = a + sF_{n-1}$. Note (Figure 2) that $b - x_2 = x_1 - a = sF_{n-2}$.

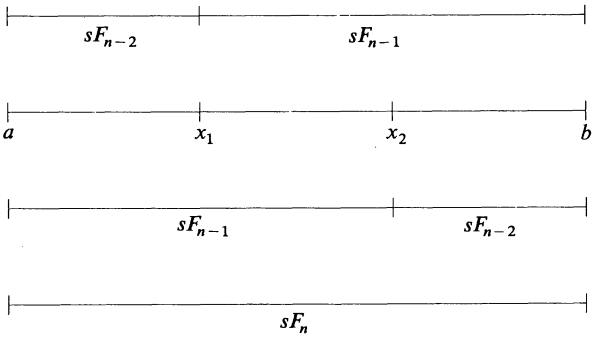


Figure 2.

Returning to the example in which $f(x_1) < f(x_2)$ and we discard $[a, x_1)$, we see that the remaining subinterval $[x_1, b]$ consists of F_{n-1} equal subintervals of length s. We then redefine $a = x_1$ and choose two new intermediate points $x_1' = x_2 = b - sF_{n-2}$ and $x_2' = a + sF_{n-2}$. The other case, $f(x_1) > f(x_2)$, is handled in an analogous fashion. Note that each iteration needs only one evaluation of f, which in fact keeps the number of evaluations to a minimum.

The following procedure in pseudocode provides an implementation of this algorithm.

```
Procedure Fibonacci_maximum;

        $s = (b-a)/F_n$;
        $x_2 = a + s*F_{n-1}$;
        $x_1 = b - s*F_{n-1}$;
        $f_1 = f(x_1)$;
        $f_2 = f(x_2)$;

        do while ( $|x_2 - x_1| > s$ );
            if $f_1 < f_2$ then do;
                $a = x_1$;
                $x_1 = x_2$;
                $f_1 = f_2$;
                $x_2 = b - (x_1 - a)$;
                $f_2 = f(x_2)$;
            end if;
            else do;
                $b = x_2$;
                $x_2 = x_1$;
                $f_2 = f_1$;
                $x_1 = a + (b - x_2)$;
                $f_1 = f(x_1)$;
            end else;
        end while;
end proc:
```

File Merging

One of the most common applications of a computer is sorting an unordered list. Many very efficient sorting algorithms are known [13], but most apply only when the entire list is contained in the computer's main memory. The problem of interest here is sorting lists which are too large to reside in the main memory all at once, and so reside in a file on some external storage device. A *file* is a sequential list of items to which we have limited access. (It is useful to think of a magnetic recording tape, although the physical device may be something else.) In particular, there is a file "window" that allows us to view only one item in the file at a time. We can position this window to the first item in the file (rewind the tape), and we can advance the window to the next item in the file whenever we choose.

The fundamental operation used in sorting such files is the *two-way merge*, a procedure in which we take two sorted (say, largest to smallest) files or subfiles as input and produce as output a single sorted file containing all items from the two original input files or subfiles. We carry this out as follows. Position both input file windows at their beginnings. Look at the two items in the windows and find the larger, say on file i. Copy this item to the output file and advance the window on file i to the next item. If a window reaches the end of its file, that input file need no longer be considered. Repeat until both windows have reached the ends of their respective files.

The most obvious way to use this merge operation in sorting files is the so-called *natural-merge sort*. Break the original file T_0 of length n into two files, T_1 and T_2. Each of these files may be regarded as a collection of $n/2$ sorted subfiles of length one $(n+1)/2$ and $(n-1)/2$, if n is odd. Now perform $n/2$ merges, each time merging a subfile from T_1 with a subfile from T_2 and writing the result back onto T_0.

(If n is odd, we perform $(n+1)/2$ merges, the last being trivial.) At the end, T_0 contains $n/2$ (or $(n+1)/2$) sorted subfiles, each of length 2 (except possibly the last). Break this collection of sorted subfiles on T_0 into two equal (or nearly so) subcollections and write them back onto T_1 and T_2. Now perform the merges again, each time merging a subfile from T_1 with a subfile from T_2 and writing the result (a sorted subfile of length 4, except possibly the last) back onto T_0. Continue breaking and merging in this fashion until T_0 contains 1 sorted subfile of length n, namely the original file, now sorted!

The main criticism of this routine is that repeatedly breaking the collection of sorted subfiles on T_0 in two subcollections and writing them back onto T_1 and T_2 involves considerable computer time for transfer and yet contributes nothing to the sort itself. This can be fixed by using an additional output file, call it T_3, to receive the results of the merges of sorted subfiles from T_1 and T_2. Instead of writing the results of each merge onto T_0, we alternately write the results onto T_0 and T_3. After merging the subfiles from T_1 with those from T_2, we find that the resulting collection of sorted subfiles (formerly held on T_0 alone) is now broken into two pieces, ready for merging! So, we just reverse roles, using T_0 and T_3 as input files and T_1 and T_2 as output files, and continue in this fashion. This extension of the natural-merge sort is called a *balanced two-way merge*, and is illustrated in Table 1.

Table 1. Balanced 2-Way Merge.

T_0:	5,11,8,13,9,7,10,12,4,2,1,6,3,14
T_3:	
T_1:	
T_2:	
T_0:	
T_3:	
T_1:	12,4,2,1,6,3,14
T_2:	5,11,8,13,9,7,10
PASS 1	
T_0:	(12,5),(8,2),(9,6),(14,10)
T_3:	(11,4),(13,1),(7,3)
T_1:	
T_2:	
PASS 2	
T_0:	
T_3:	
T_1:	(12,11,5,4),(9,7,6,3)
T_2:	(13,8,2,1),(14,10)
PASS 3	
T_0:	(13,12,11,8,5,4,2,1)
T_3:	(14,10,9,7,6,3)
T_1:	
T_2:	
PASS 4 (FINAL MERGE)	
T_0:	
T_3:	
T_1:	(14,13,12,11,10,9,8,7,6,5,4,3,2,1)
T_2:	

The only obvious criticism of the balanced two-way merge is that the use of a fourth file (magnetic tape drive) seems artificial and somewhat wasteful. After all, at any moment in the merge phase we are really combining two input streams into a single output stream. Isn't there a way to merge sort using just three files and still avoid the time-consuming transfer operations found in the natural-merge sort? An affirmative answer (illustrated in Table 2) is provided by R. L. Gilstad's Fibonacci (polyphase) merge sort [5].

Start with a file T_0 of size F_n (pad with dummy items, as necessary). Break this into files T_1 and T_2 of sizes F_{n-1} and F_{n-2}, respectively. Now regard T_1 as a collection of F_{n-1} sorted subfiles of length 1, and T_2 as a collection of F_{n-2} sorted subfiles of length 1. Using T_1 and T_2 as input and T_0 as output, perform F_{n-2} merges. At the end, T_0 contains F_{n-2} sorted subfiles of length 2, T_1 contains $F_{n-1} - F_{n-2} = F_{n-3}$ sorted subfiles of length 1, and T_2 is, for our purposes, empty. Now use T_0 and T_1 as input and T_2 as output, and perform F_{n-3} merges, each a merge of a sorted subfile from T_0 (length 2) with a sorted subfile from T_1 (length 1). At the end, T_2 contains F_{n-3} sorted subfiles of length 3, T_0 contains $F_{n-2} - F_{n-3} = F_{n-4}$ sorted subfiles of length 2, and T_1 is empty. Continue in this fashion until one file contains a single sorted subfile of length F_n.

Table 2.

```
T0:  5,11,8,13,9,7,10,12,4,2,1,6,3
T1:
T2:

T0:
T1:  7,10,12,4,2,1,6,3
T2:  5,11,8,13,9

PASS 1
T0:  (7,5),(11,10),(12,8),(13,4),(9,2)
T1:  1,6,3
T2:

PASS 2
T0:  (13,4),(9,2)
T1:
T2:  (7,5,1),(11,10,6),(12,8,3)

PASS 3
T0:
T1:  (13,7,5,4,1),(11,10,9,6,2)
T2:  (12,8,3)

PASS 4
T0:  (13,12,8,7,5,4,3,1)
T1:  (11,10,9,6,2)
T2:

PASS 5 (FINAL MERGE)
T0:
T1:
T2:  (13,12,11,10,9,8,7,6,5,4,3,2,1)
```

Fibonacci Search of Ordered Arrays

Our next algorithm locates a key value in an ordered array of numbers where the number of elements in the array is a Fibonacci number minus one. If the array does not have exactly $F_n - 1$ entries for some Fibonacci number F_n, we pad the array with dummy elements. A simple example will demonstrate the efficiency of this algorithm.

Determine if 15 is an element of the array $(0,1,2,3,5,6,9,11,15,18,20,23)$. Note that in this example there are $F_7 - 1 = 12$ entries.

We begin by comparing the value 15 to entry $F_{7-1} = 8$, which has the value 11. Since $11 < 15$, we may eliminate all entries to the left of and including entry 8. Thus, the array is now reduced to $(15,18,20,23)$, which contains $F_5 - 1$ entries. (Had the value of entry 8 been > 15 we would have eliminated entries to the right of and including entry 8, leaving $F_6 - 1$ entries.)

We now compare entry $F_{5-1} = 3$ in the sub-array, which has the value 20, to the value 15. Since $15 < 20$ we eliminate all entries to the right of and including entry 3. The remaining sub-array is $(15,18)$, which contains $F_4 - 1$ entries.

Using the same approach, we compare entry $F_3 = 2$, having value 18, to 15 and discard entry 2. We finally compare the value 15 to entry $F_2 = 1$ and have a match! We would thus report yes!

We include an implementation of this algorithm.

```
Procedure Fibonacci_search;
/* array A() contains $F_n$ -1 entries, and key is the value we seek

    left = 1;        /* index of left end of sub array still under consideration
    right = $F_n$ -1; /* index of right end of sub array still under consideration
    s = $F_{n-1}$; /* successive Fibonacci numbers
    t = $F_{n-2}$;

    do while ( left<right and A(left-1+s) ≠ key );
        if key < A(left-1+s) then do;
            right = right - t;
            t = s - t;
            s = s - t;
        end_if;
        else do;
            left = left + s;
            s = s - t;
            t = t - s;
        end_else;
    end_while;
    if A(left-1+s)=key then print ("yes");
    else print ("no");
end_proc.
```

Fibonacci Search Trees

Fast storage and retrieval of information has always been an issue of primary concern to users of computer systems. Our discussion of this problem requires some vocabulary.

A *binary tree* is a finite nonempty set of nodes in which one node is designated the root and the remaining nodes are partitioned into at most two disjoint subsets, each of which is a binary tree. The *level* of a node in a tree is as follows: The level of the root is 1. If the root of a tree (subtree) has level i, the roots of its two subtrees have level $i + 1$. The maximum level found in a tree is called the *depth* or *height* of the tree. A node with no subtrees is called a *leaf node*. A binary tree of height h is *complete* (or has minimal height) if each leaf node has level h or $h - 1$.

Binary trees are easy to represent in computer memory. We store with each node the addresses (in memory) of the roots of its left and right subtrees.

A *binary search tree* is a binary tree in which the information in the nodes has been ordered in a special way. All nodes in any left subtree have value less than that in the root (of the subtree), and all nodes in any right subtree have value greater than that in the root. A typical binary search tree is shown in Figure 3.

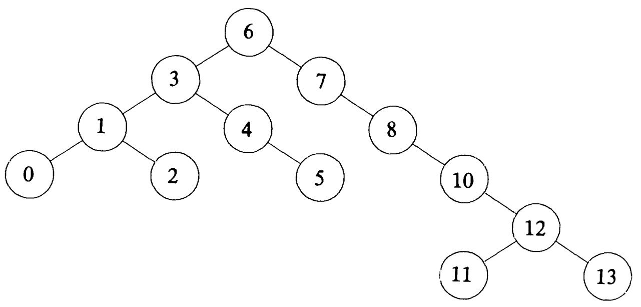


Figure 3.

The algorithm used to search such a tree for the node which contains a specific value is elementary:

> Look at the value in the root. If that is the desired value, we are done! If the desired value is less than the root value, go to the left subtree. If the desired value is greater than the root value, go to the right subtree. Repeat until the desired value is found or a leaf node is encountered.

If we use this procedure on a complete binary search tree with n nodes and height h, then $2^{h-1} \leq n \leq 2^h - 1$, so $\log_2(n + 1) \leq h \leq (\log_2 n) + 1$, and we will need at most $h = \lceil \log_2(n+1) \rceil$ "looks" to find a value or to determine that it is not there.

The difficulty in using a complete binary search tree for storage and retrieval is that insertions and deletions can destroy completeness, and with it the advantage (in "looks") of such a storage arrangement over a simple linear arrangement. No algorithms are known which will maintain completeness of a binary search tree when insertions and deletions are dynamically taking place.

Adel'son-Vel'skii and Landis investigated the problem of efficient searches on trees where insertions and deletions occur. Their approach was to consider a generalization of the complete tree. An *AVL* (*balanced*) *tree* is a tree in which the heights of subtrees at any given node differ by no more than 1. Adel'son-Vel'skii and Landis were able to develop algorithms which insert and delete nodes from such trees and still keep the trees balanced [13]. The natural question is how much higher an AVL tree of n nodes is than a complete tree of n nodes, or equivalently, how many more looks will we need to find the information we want?

Let us determine the fewest nodes in an AVL tree of height h. From the assumed minimality of the tree, we can assert that if the tree has height h, one subtree has height $h-1$ and the other has height $h-2$. Thus, to construct an AVL tree of height h with the fewest nodes, we need only recursively find an AVL tree of height $h-1$ with the fewest nodes and, similarly, one of height $h-2$ with the fewest nodes, and attach them to a root node. Figure 4 shows an example construction.

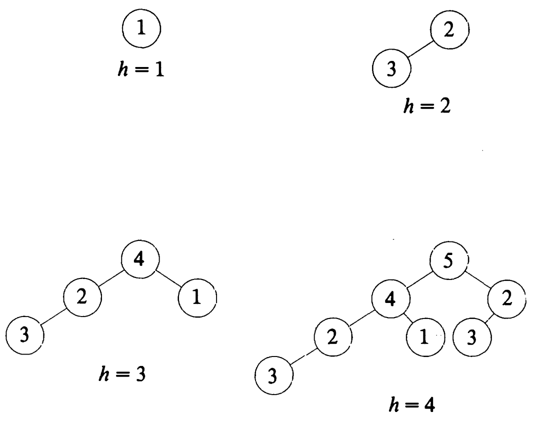


Figure 4.

We now note that if $n(h)$ is the minimal number of nodes in the AVL tree of height h, then

$$n(1)=1, \quad n(2)=2, \quad \text{and} \quad n(3)=n(1)+n(2)+1.$$

It follows from the previous discussion that

$$n(h)=n(h-1)+n(h-2)+1.$$

But $n(1)=F_3-1$ and $n(2)=F_4-1$, so

$$n(h)=(F_{h+1}-1)+(F_h-1)+1=F_{h+2}-1.$$

The minimal number of nodes in an AVL tree of height h is thus $F_{h+2}-1$. An AVL tree with a minimum number of nodes is called a *Fibonacci tree*.

To answer the initial question as to how much higher an AVL tree with n nodes is than a complete tree with n nodes, we can replace F_{h+2} by its representation in terms of the golden ratio.

Recall [12] that a closed-form expression for the nth Fibonacci number is

$$F_n = \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right) \Big/ \sqrt{5}.$$

To the nearest integer then, $F_n = ((1+\sqrt{5})/2)^n/\sqrt{5}$. An AVL tree of height h with $n \geq F_{h+2} - 1$ nodes thus has

$$n \geq \left(\left(\frac{1+\sqrt{5}}{2} \right)^{h+2} \Big/ \sqrt{5} \right) - 1$$

so that its height h is bounded above by $1.44 \log_2(n+1) - .328$. If we compare this with our earlier expression for the height of a complete binary tree, $\log_2(n+1)$, we see that the necessary sacrifice in height (number of looks) in the AVL approach is surprisingly little.

Conclusion

The Fibonacci Sequence was originally proposed to describe a vast diversity of phenomena from nature [2], [3], [15]. Evidently the hand of nature reaches far and wide, for we have seen that this same sequence serves as the foundation for a diversity of algorithms used extensively in computer science. The interested reader is encouraged to consult the references for additional examples.

REFERENCES

1. M. Avriel, *Nonlinear Programming*, Prentice Hall, 1976, pp. 225–233.
2. H. S. M. Coxeter, "The Golden Section, Phyllotaxis and Wythoff's Game," Scripta Mathematica 1:1 (1953) pp. 135–143.
3. J. DeVita, "Fibonacci, Insects and Flowers," Fibonacci Quarterly 16 (August, 1978) 315–317.
4. Keenan Forsythe and Stenberg Organick, *Computer Science*, *A First Course*, John Wiley, New York, NY, 1969.
5. R. L. Gilstad, "Polyphase Merge Sorting—An Advanced Technique," Proc. AFIPS Eastern Jt. Comp. Conf. 18 (1960) 143–48.
6. G. H. Gonnet, "Balancing Binary Trees by Internal Path Reduction," CACM 26 (1983), 1074–1081.
7. R. R. Hill and K. L. Goldstein, "A Non-Fibonacci Search Plan with Fibonacci-like Results," Fibonacci Quarterly 19 (April, 1981) 131–136.
8. Y. Horike, "Notes on Fibonacci Trees and Their Optimality," Fibonacci Quarterly 21 (May, 1983) 118–128.
9. E. Horowitz and S. Sahni, *Fundamentals of Data Structures*, Computer Science Press, Potomac, MD, 1976.
10. ______, *Fundamentals of Computer Algorithms*, Computer Science Press, Potomac, MD, 1978.
11. J. Kiefer, "Sequential Minimax Search for a Maximum," PAMS 4 (1953) 502–506.
12. D. E. Knuth, *The Art of Computer Programming*, *Fundamental Algorithms*, Vol. 1, Addison-Wesley, Reading, MA, 1973.
13. ______, *The Art of Computer Programming*, *Sorting and Searching*, Vol. 3, Addison-Wesley, Reading, MA, 1973.
14. J. Nievergelt and E. M. Reingold, "Binary Search Trees of Bounded Balance," SIAM J. Comput. 2:1 (1973) 33–43.
15. P. S. Stevens, *Patterns in Nature*, Atlantic Monthly Press, Little, Brown, and Company, Boston, MA, 1974.
16. N. Wirth, *Algorithms + Data Structures = Programs*, Prentice Hall, 1976, pp. 87–121.
17. C. Witzgall, "Fibonacci Search with Arbitrary First Evaluation," Fibonacci Quarterly 10:2 (1972) 113–135.