ACM Inroads, Vol 3, No. 3, (September 2012), pp. 40-45.

Dynamic Programming the Towers[1]

Timothy Rolfe

Abstract

After a brief overview of the dynamic programming optimization for solving recurrence problems, this paper shows its application to one of the most famous recurrences, that of the Towers of Hanoi. Because of code complexity, the topdown implementation is initially slower than the recursive, but is faster for sizes above 7. The dynamic programming implementations have initially linear behavior, moving eventually to exponential. The top-down implementation at one point requires only 1% of the time for the recursive implementation.

Brief Review of Dynamic Programming

Given a "divide-and-conquer" problem formulated in terms of combining slightly smaller subproblems for the final solution, one might be tempted to implement directly the recursive definition. Fibonacci numbers provide an example:

$F_0 = 0$; $F_1 = 1$; for all $n > 1$, $F_n = F_{n-1} + F_{n-2}$

This is spectacularly inefficient. [3] One can prove that the number of method calls required for this calculation is $2 \cdot F_{n+1} - 1$.[2]

Similarly one can look at binomial coefficients, $C(n,k)$. The formulation in terms of Pascal's Triangle has this recursive definition:

For $n \geq 0$ and $0 \leq k \leq n$: $C(n,0) = C(n,n) = 1$
For $0 < k < n$, $C(n,k) = C(n-1,k) + C(n-1,k-1)$

Again, this is spectacularly inefficient. [1] One can prove that the number of method calls required for this calculation is $2 \cdot C(n, k) - 1$.[3]

Since the cause of the inefficiency is multiple calculations of the same quantities, the solution is to insure that each quantity is only calculated once using something called "dynamic programming". Top-down dynamic programming (also called "memoization") allows retention of the recursive structure while computing each quantity only once.[3] This can be

---

[2] http://penguin.ewu.edu/cscd320/Topic/Recursion/FibonacciRecurrence.html

[3]

```
Calls(0,0)  = 2*C(0, 0) - 1                          Base case
            = 2*1 - 1 = 1
Calls(n,n)  = 2*C(n, n) - 1                          Second base case
            = 2*1 - 1 = 1
Calls(n,k)  = Calls(n-1,k) + Calls(n-1, k-1) + 1     Recurrence for 0 < k < n
            = 2*C(n-1,k)-1 +  2*C(n-1, k-1)-1 + 1    Substitute inductive hypothesis
            = 2*(C(n-1, k) + C(n-1, k-1)) - 2 + 1
            = 2*C(n, k) - 1                          Binomial coefficient recurrence
```

accomplished by having memory set aside to hold values for previous calculations. The cells initially contain a value that flags a particular cell as not having been computed (0 in the `F(int i)` method below). In C one can use static arrays which retain their contents between function calls, but in Java one must use a class array outside of the Java method to get this behavior. For Fibonacci numbers, this can easily be handled by having a vector at the class level. Since a zero value flags an uncomputed cell, Fib(0) must be handled explicitly, as shown in the following code fragment.

```
static final int maxN = 92;
static long knownF[] = new long[maxN+1];
static long F(int i)
{  if (i < 2) return i; // So knownF[k]==0 can flag an uncomputed cell
   if (knownF[i] == 0)
      knownF[i] = F(i-1) + F(i-2);
   return knownF[i];
}
```
[4]

This trades space for time. In return for spending linear space to hold the vector of `knownF`, one computes Fibonacci numbers not in exponential time but in linear time — or faster, since once the vector is populated Fibonacci numbers are available in constant time.

A similar approach can be used for the binomial coefficient problem, but will not be given here.[5] This approach has the desirable characteristic of "on-demand" computation: a particular subproblem is computed only if it is needed for a larger problem. Thus the table of computed binomial coefficients may have numerous cells that are never filled because they are never needed.

Connecting to the Towers of Hanoi

Briefly, the Towers of Hanoi begins with three towers/pegs and *n* disks of varying diameter arranged so that they are initially all stacked on one tower, in decreasing size from bottom to top. They are to be transferred to another tower subject to two constraints: the disks are moved one at a time, and no disk is ever on top of a smaller one. [2]

With a small shift to self documenting variable names to correspond with later code segments, Sahni's implementation of the Towers of Hanoi becomes the following code:

```
public static void towersOfHanoi(int n, int src, int dst, int tmp)
{// Move the top n disks from tower src to tower dst.
 // Use tower tmp for intermediate storage.
    if (n > 0)
    {  towersOfHanoi(n-1, src, tmp, dst);
       System.out.println("Move top disk from tower " + src +
                          " to top of tower " + dst);
```

[4] [3] Program 5.11 revised based on the definition of Fibonacci numbers used here and 64bit integers. See also http://penguin.ewu.edu/cscd320/Topic/Strategies/DynamicPgming/Fibonacci.java

[5] See http://penguin.ewu.edu/cscd320/Topic/Strategies/DynamicPgming/Binom_Memo.java

```
            towersOfHanoi(n-1, tmp, dst, src);
        }
    }
```

This is a classical exponential problem requiring $2^n - 1$ disk movements to transfer all disks from *src* to *dst* by way of *tmp*. So the answer is inescapably exponential to get some representation of the series of moves. With dynamic programming, however, one *can* trade exponential space for less than exponential time, at least for a region in which dealing with exponential space does not require exponential time. To solve the *n* disk problem, one can have a memoization matrix of *n*+1 rows and 6 columns[6] to contain subproblem solutions. The problem solutions will be represented by character strings.

Solution Representation

Since the storage of earlier results is critical in minimizing the space required, each move is represented by a single character. Every disk movement operation is treated as an ordered pair of tower designations, giving the source and destination towers, ranging from 01 up through 21. If these are considered to be base-3 numbers, the range of magnitudes runs from 1 through 7, and can be stored as a single octal digit. The solution then becomes a string of octal digits representing all moves to accomplish the transfer of all disks from the initial source tower to the final destination tower. Thus for the transfer of *n* disks from *src* to *dst*, using *tmp* for intermediate storage, the solution will be the transfer of (*n*–1) disks from *src* to *tmp*, using *dst* for intermediate storage, followed by the transfer of one disk from *src* to *dst*, and ending with the transfer of (*n*–1) disks from *tmp* to *dst*, using *src* for intermediate storage. This represents the solution for (*n*–1) from *src* to *tmp* using *dst*, concatenated with the single digit representing the move from *src* to *dst*, and then concatenated with the solution for (*n*–1) from *tmp* to *dst* using *src*.

For instance, the solution for moving four disks from tower zero to tower two by way of tower one is represented thus, the middle move in bold and italic: "1251671***2***5365125". Shown as digit pairs, this is "01,02,12,01,20,21,01,***02***,12,10,20,12,01,02,12". Seven moves uncover the bottom disk, which is moved from zero to two, and then seven moves cover it over.

Figure 1 shows all these steps.[7] The first move solves the *n*=1 problem from 0 to 1, the first three, the *n*=2 problem from 0 to 2, and the first seven, the *n*=3 problem from 0 to 1.

---

[6] Given movement of disks among towers 0, 1, and 2, there are six permutations of those digits.

[7] These were generated by the program at http://penguin.ewu.edu/~trolfe/DynamicHanoi/Four_Disk_Hanoi.java. http://en.wikipedia.org/wiki/File:Tower_of_Hanoi_4.gif is a very nice animated gif that André Karwath added to the Wikipedia page on the Towers of Hanoi 22 March 2005. http://penguin.ewu.edu/cscd320/Topic/Recursion/HanoiBAS.exe provides a very primitive (character graphics) animation of the problem. It is specific to Microsoft Windows and is a 32-bit executable, not runnable in a 64bit environment. Source code is in http://penguin.ewu.edu/cscd320/Topic/Recursion/HanoiBAS.bas http://www.qb64.net/ provides a way to exercise BASIC code in a 64-bit Windows environment.

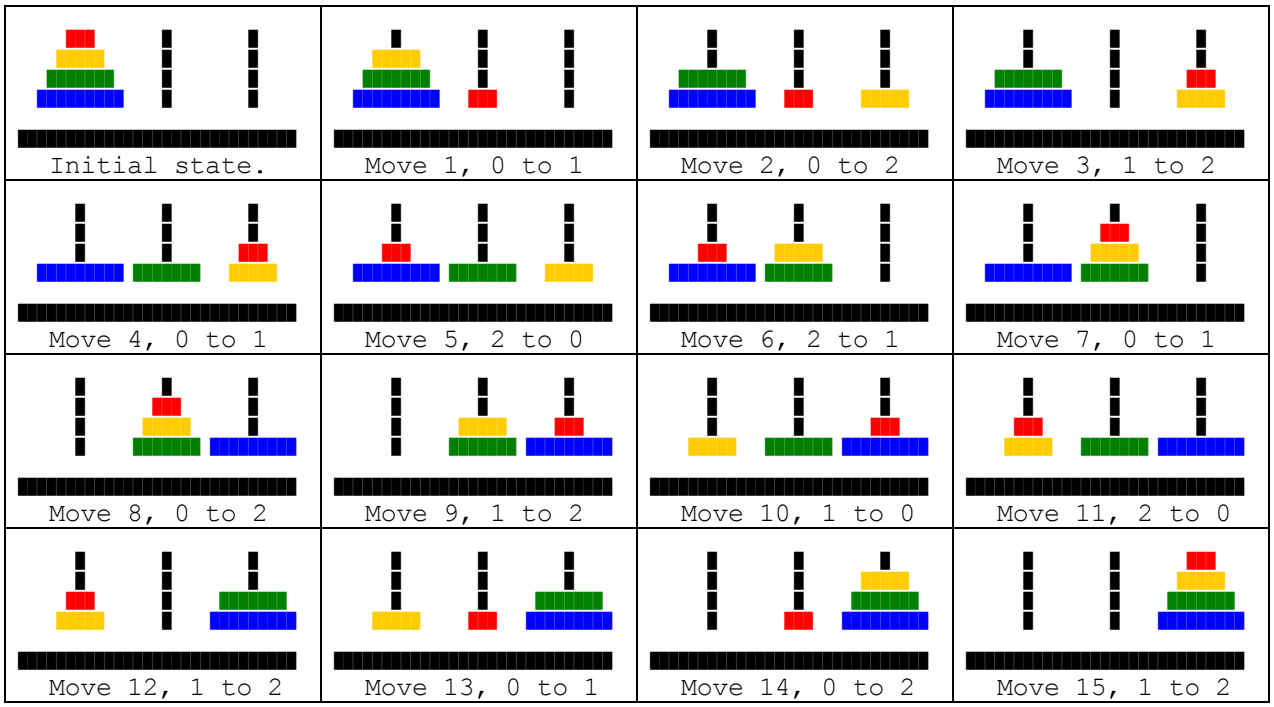

Figure 1: States in solving Hanoi(4, 0, 2,. 1)

The memoization matrix then becomes a six-column matrix of strings. Each column represents a particular permutation of the available source, destination, and temp towers: "012", "021" up through "201" and "210". Each row represents the number of disks being moved. Thus row zero contains the base case: no disks being moved, which for computational convenience is represented by the empty string. Rows of higher index contain six cells for the six possible problems. If the cell contains a null reference, that problem has not yet been solved. For a given triple (*a*,*b*,*c*) as (*src*, *dst*, *tmp*), the solution is the (*n*–1) solution (*a*,*c*,*b*) to clear those disks from *a* onto *c*, followed by the single move from *a* to *b*, ending with the (*n*–1) solution (*c*,*b*,*a*) to move the cleared disks on top of the single disk that was just transferred. The Java code segment that follows shows this:

```
// The class static matrix memo[nRow>n][6] contains solutions
   static String hanoi(int n, int src, int dst, int tmp)
   {  // Initialization code omitted . . .
      if (memo[n][index] == null)
      {  // Two-digit data movement treated as base-3:  01 through 21
         // Get the magnitude of the base-3 number, store as octal
         String addend = String.format("%1o", 3*src + dst);
         String result = hanoi(n-1, src, tmp, dst) + addend +
                         hanoi(n-1, tmp, dst, src);
         memo[n][index] = result;
      }
```

```
    return memo[n][index];
}
```
[8]

If one allows for a trailing '\0' character to terminate a string, the solution for moving n disks requires 2^n characters. Generating the solution, however, requires combining three strings, two of them of size 2^{n-1}. Java obscures the expense of combining strings that is much clearer in C: the space must be allocated from the heap (`malloc`/`alloc`/`calloc`), and then the contents must be copied into it (`strcpy`/`strcat`). For short strings, the expense of memory allocation dominates, giving behavior linear in the number of disks. Though memory copying is fast, for large enough strings that expense, linear in the size of the string, will dominate, giving exponential time for exponential space.

Other Requirements for the Top-Down Solution

Examination of the final contents of the memoization matrix shows that slightly less than half of it is used. Table 1 shows the cells of the matrix that are used when solving the problem of moving twenty disks from tower 0 to tower 2. Row 0 contains the base-case solutions, empty strings. The top-most row contains only the cell explicitly computed (021), which then generates two cells in the row below that (012 and 120). After that only three of the six cells are computed.

Since the solution for k disks requires 2^k characters, one can explicitly compute the space required. The top-most row contains one 2^n string. Below that are two entries with 2^{n-1} strings, representing another 2^n characters. Below that is simply three times the summation of powers of two.

$$3\sum_{k=0}^{n-2} 2^k = 3 \bullet (2^{n-1} - 1) = \frac{3}{2} \bullet 2^n - 3$$

The total space required is thus about $3½ \cdot 2^n$.

The function calls are linear, thanks to memoization. By direct measurement, Calls(1) = 3, Calls(2) = 7, and then for $n>2$, Calls(n) = $6n - 3$.

[8] The full program is available in http://penguin.ewu.edu/~trolfe/DynamicHanoi/

Table of results actually computed						
Row	012	021	102	120	201	210
20		X				
19	X			X		
18		X	X			X
17	X			X	X	
16		X	X			X
15	X			X	X	
14		X	X			X
13	X			X	X	
12		X	X			X
11	X			X	X	
10		X	X			X
9	X			X	X	
8		X	X			X
7	X			X	X	
6		X	X			X
5	X			X	X	
4		X	X			X
3	X			X	X	
2		X	X			X
1	X			X	X	
0	X	X	X	X	X	X

Table 1: Memoization Matrix Cells Used

Bottom-Up Dynamic Programming Solution

The "bottom-up" dynamic programming approach is to build successively higher levels of solutions starting with the level of the base case or the base cases. In the process, there is no need to retain lower levels once the higher levels are computed that rely on them. Sedgewick notes "In the case of Fibonacci numbers, we can even dispense with the array and keep track of just the previous two values (see Exercise 5.37)." [3, p. 219] Here is a specimen implementation.

```
static long fib(int n)
{  long v[] = { 0L, 1L, 1L };

   // Compute until fib(n) is in v[0]
   while (n-- > 0)   // Count down
   {  v[2] = v[0] + v[1];
      v[0] = v[1];   // Shift downward
      v[1] = v[2];
   }
   return v[0];
}
```

Similarly, for the Towers of Hanoi one can retain just two rows from the memoization matrix, the previous case computed (initialized to empty strings) and the current case being computed. These then are swapped in working to the next higher number. An implementation follows:

```
   static String botHanoi(int n, int s, int d, int t)
   {  // 6 empty strings for 01 through 21; null for 00 and 11
      String[] prev = { null, "", "", "", null, "", "", ""  },
               curr = new String[8],   // compute into this
               temp;                   // interchange prev and curr
      int i, // Outer loop on level of solution
          j, // Inner loop for the six problems at level i
          offset = 3*s + d; // Position of final answer

      // Final result will be in prev
      for (i = 1; i <= n; i++)
      {  // Populate curr from prev
         for (j = 1; j < curr.length; j++)   // Omit 00
         {  // 11 is not allowed
            int src = j/3, dst = j%3,
                tmp = 3 - src - dst;

            if (src == dst) continue;  // I.e., 11
            //h(i-1,src,tmp,dst) + src to dst + h(i-1,tmp,dst,src)
            curr[j] = prev[src*3+tmp] + (src*3+dst) +
prev[tmp*3+dst];
         }
         // Swap curr and prev.
         temp = curr;  curr = prev;  prev = temp;
      }
      return prev[offset];
   }
```
[9]

Other Requirements for the Bottom-Up Solution

The garbage collector is quite busy in this implementation: all solutions are discarded except for the final two sets, six for the n–1 problem and six for the n problem. The six for the n–1 problem require $6 \cdot 2^{n-1}$ characters, or $3 \cdot 2^n$, while the six for the n problem require $6 \cdot 2^n$ more, for a total of $9 \cdot 2^n$ characters. The total space required for the top-down approach, discussed above, is about $3½ \cdot 2^n$.characters. There is only one function call, since the solution is iterative.

Requirements for the Recursive Solution

By the nature of the problem, the recursive stack only goes as deep as the number of disks. The space requirement is simply 2^n for the length of the string representing the moves, but the

[9] The full program is available in http://penguin.ewu.edu/~trolfe/DynamicHanoi/

garbage collector is busy recycling intermediate solutions. The number of function calls, of course, is exponential. Calls(0) = 1, Calls(n>0) = 2 Calls(n–1) + 1 = $2^{n+1} - 1$. [10]

Experimental Results

How, then, do these three implementations stack up against each other? Three separate Java programs were developed to capture time statistics for the three different implementations discussed above (top-down, bottom-up, and recursive).[11] Each size was forced to execute for at least two seconds to average over multiple runs for fast-executing sizes. To allow for extremely long strings, the Java Virtual Machine was initialized with both initial and maximum heap size as 1 GByte:

```
java -Xms1024m -Xmx1024m <program>
```

The results are interesting. The dynamic programming implementations begin with a linear region before turning exponential. The programs were run on an unloaded computer in the Computer Science Department at Eastern Washington University under Linux.[12] Table 2 shows the results[13] of one set of executions for problems from size 2 through size 25. It is useful to have the explicit numbers, since the logarithmic scale in the graph exaggerates small numbers and minimizes large numbers. The time reported is in elapsed milliseconds, available through the Java method System.nanoTime().

---

[10] Calls(0) = $2^1 - 1 = 1$ *Base case*
Calls(n) = 2 Calls(n-1) + 1 *Recurrence*
= $2\ (2^n - 1) + 1$ *Substitute inductive hypothesis*
= $2^{n+1} - 2 + 1$
= $2^{n+1} - 1$ *QED*

[11] Available in http://penguin.ewu.edu/~trolfe/DynamicHanoi/ as TopHanoi.java, BotHanoi.java, and RecHanoi.java.

[12] Processor (from `/proc/cpuinfo`):
Intel(R) Xeon(R) CPU 5160 @ 3.00GHz — two processors reported
Operating System (from `/proc/version`):
Linux version 2.6.32-24-server (buildd@yellow) (gcc version 4.4.3 (Ubuntu 4.4.3-4ubuntu5)) #43-Ubuntu SMP Thu Sep 16 16:05:42 UTC 2010
Java version (from `java -version`):
Java(TM) SE Runtime Environment (build 1.6.0_26-b03)
Java HotSpot(TM) 64-Bit Server VM (build 20.1-b02, mixed mode)

[13] The Excel workbook: http://penguin.ewu.edu/~trolfe/DynamicHanoi/25Avg.xls

size	Top-Down	Bottom-up	Recursive
2	0.021	0.001	0.003
3	0.038	0.001	0.007
4	0.048	0.002	0.013
5	0.064	0.003	0.027
6	0.079	0.004	0.055
7	0.094	0.006	0.108
8	0.110	0.010	0.215
9	0.127	0.019	0.447
10	0.147	0.036	0.873
11	0.171	0.068	1.738
12	0.206	0.130	3.697
13	0.259	0.253	7.061
14	0.352	0.497	14.061
15	0.517	0.987	29.181
16	0.837	1.983	57.461
17	1.472	4.098	115.054
18	2.826	9.270	254.244
19	5.889	21.505	480.490
20	13.866	50.037	1029.500
21	31.728	120.463	2193.638
22	74.903	280.264	4114.886
23	168.563	832.855	8389.754
24	360.890	1807.628	17008.208
25	1254.996	4476.174	33808.957

Table 2: Runs from 6 June 2012

The top-down appears to be showing approximately linear behavior up through size 10, and then begins increasing, reaching purely exponential at size 18. The bottom-up has a linear region through size 7, and is purely exponential above size 13. The recursive implementation, as expected, is exponential throughout. Figure 2 shows these results graphically. The “Ratio” plot is the “Top-Down” time divided by the “Recursive” time, showing the speed-up with the dynamic programming implementation that performed best at larger sizes.

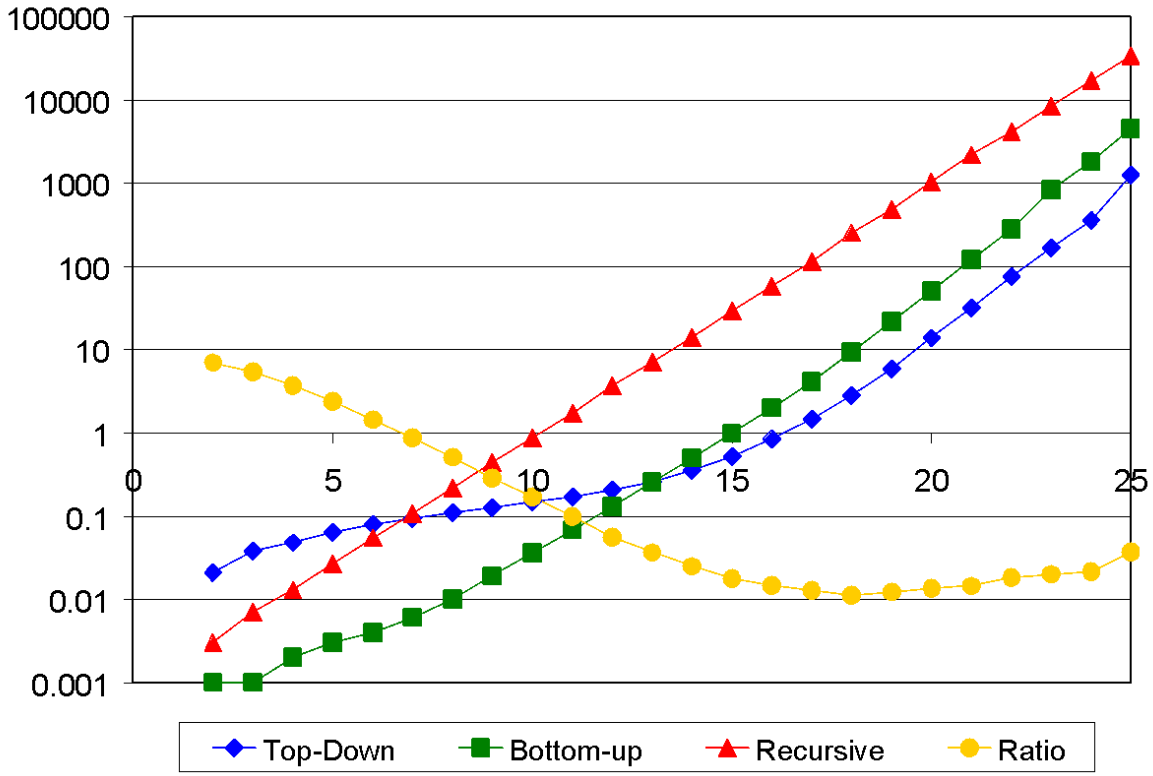

Figure 2: Measured execution times in milliseconds for the three Java implementations

Performance of C Implementations

These same algorithms can be implemented in the C language.[14] In that environment, however, the programmer is responsible for memory management, while the Java language provides a garbage collector. On the other hand, as a compiled rather than interpreted language, C tends to generate faster-executing code. Figure 3 shows the times captured from the C programs equivalent to the Java programs that generated Figure 2.[15] Compared with the Java implementations, the C implementations are blindingly fast.

[14] The C environment on the computer used is gcc. `gcc --version` reports the following.
gcc (Ubuntu 4.4.3-4ubuntu5.1) 4.4.3 Copyright (C) 2009 Free Software Foundation, Inc.

[15] http://penguin.ewu.edu/~trolfe/DynamicHanoi/ provides access to the Excel workbook, 25Avg_C.xls, and to the three C programs: TopHanoi.c, BotHanoi.c, and RecHanoi.c. The data in 25Avg_C.xls show that compiling using "`gcc -O3`" only improves the recursive implementation's performance.

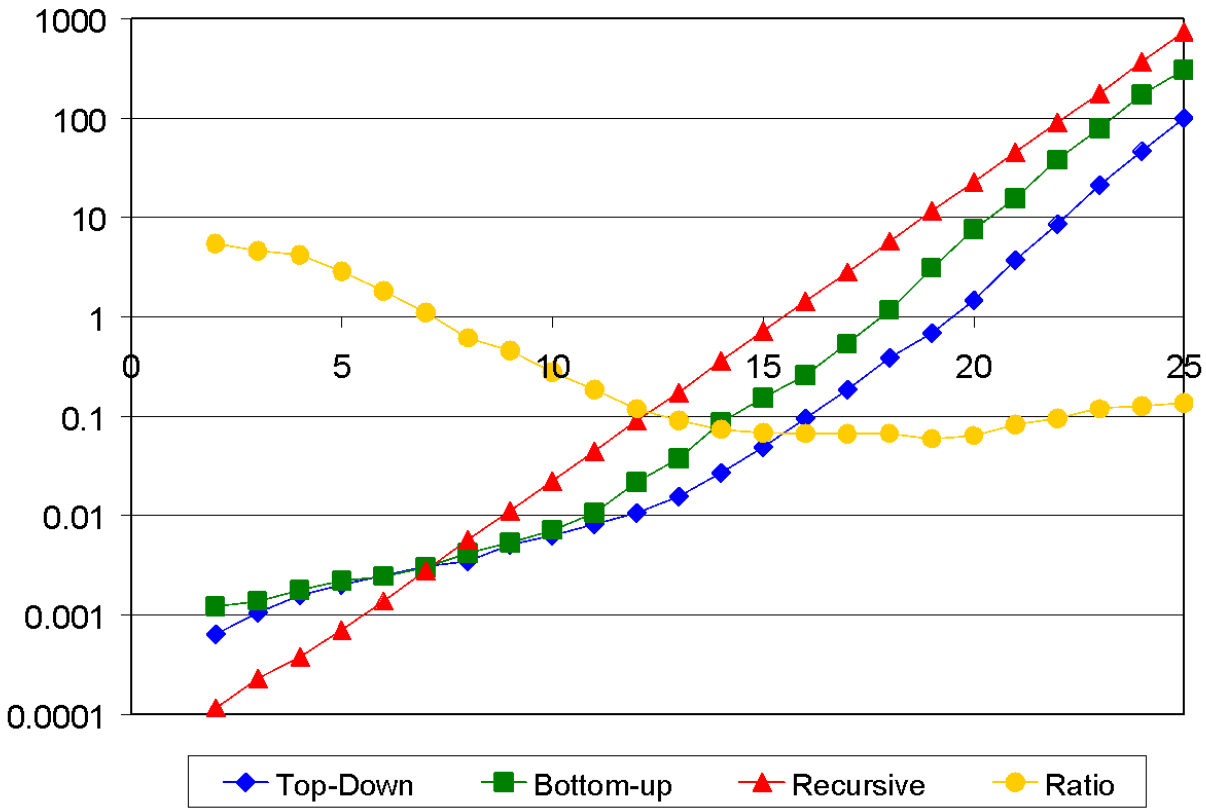

Figure 3: Measured execution times in milliseconds for the three C implementations

Acknowledgements

These computations were performed on otherwise idle computers in the Computer Science Department at Eastern Washington University. I would like to thank Bojian Xu, Ph.D., Assistant Professor of Computer Science at Eastern Washington University, for his helpful comments as I developed this paper. I also wish to thank the anonymous reviewers of this paper for their helpful suggestions, and Prof. John Impagliazzo, the editor of **Inroads**, for his help in wrestling this paper into final form.

References

[1] Rolfe, Timothy J., "Binomial Coefficient Recursion: The Good, and The Bad and Ugly", **inroads,** Vol 33, No. 2 (June 2001), pp. 35-36.

[2] Sahni, Sartaj, **Data Structures, Algorithms, and Applications in Java**, (2nd edition; Silicon Press, 2005), pp.320323.

[3] Sedgewick, Robert, **Algorithms in Java**, (3rd edition; Addison-Wesley, 2003), pp.219-226.

Timothy Rolfe
Professor of Computer Science Emeritus
Eastern Washington University
319F CEB
Cheney, Washington 99004-2493, USA

http://penguin.ewu.edu/~trolfe
mailto:Timothy.Rolfe@mail.ewu.edu

Categories and Subject Descriptors
D.2.8 Metrics — Performance measures
F.2.2 Nonnumerical Algorithms and Problems — Computations on discrete structures

General Terms
Algorithms, Performance

Keywords
Dynamic Programming, Memoization, Towers of Hanoi, Optimization