Dynamic Programming

This is the first in a series of columns on advanced programming techniques and algorithms. This issue's column discusses dynamic programming, a powerful algorithmic scheme for solving discrete optimization problems. We illustrate the concepts with the generation of Fibonacci numbers, and then present two nontrivial examples, optimal matrix-chain multiplication and multiple-class mean value analysis of queueing networks. This last example applies the technique to a queueing-theory problem that was solved in a very different way by A.O. Allen and G. Hynes in volume 1, issue 3 of this journal.

David B. Wagner

Certain problems have recursive solutions that are very natural but also are very inefficient. The reason for this inefficiency is that many identical recursive calls are made during any given computation. This property, called *overlapping subproblems*, is characteristic of many important problems. The most well-known examples are discrete optimization problems such as the 0–1 knapsack problem, optimal matrix-chain multiplication, finding the longest common subsequence of two sequences, and the Floyd-Warshall algorithm for finding shortest paths in a graph [Cormen et al. 1990]. Other, non-optimization examples include convolution and multiple-class mean value analysis of queueing networks, the latter being the topic of the second part of this article. (A good source for a discussion of both of these topics is [Jain 1991].)

Problems having the overlapping subproblems property are almost always solved using *dynamic programming*, a catch-all term for any algorithm in which the definition of a function is extended as the computation proceeds [Cormen et al. 1990]. This is generally accomplished by constructing a solution "bottom up" (e.g., progressing from simpler to more complex cases), the goal being to solve each subproblem before it is needed by any other subproblem.

The main disadvantage of dynamic programming is that it is often nontrivial to write code that evaluates the subproblems in the most efficient order. However, there is an elegant dynamic programming technique that does not require the programmer to establish the evaluation order: recursion with *result caching* (sometimes called *memoization* in the computer science literature). By caching the results of all recursive calls, the second and subsequent evaluations of any subproblem become constant-time operations, reducing the overall running time considerably. The ability to add rules to a function as the function executes makes result caching very easy to implement in *Mathematica*.

In this article we will solve problems having the overlapping subproblems property. The first of these, the computation of Fibonacci numbers, is a "toy" problem that we use to illustrate the concepts. The second problem we will consider is that of finding the optimal multiplication order for a chain of matrix multiplications, a problem having considerable practical significance. These two examples are excerpted from the author's forthcoming book, *Power Programming with Mathematica*, which will be published by McGraw-Hill. The third problem is that of solving a queueing network using the multiple-class mean value analysis technique [Reiser and Lavenberg 1980], which has been considered previously in this journal by Arnold Allen and Gary Hynes [Allen and Hynes 1991]. We will find that our solution is both easier to understand and more efficient than the solution presented by Allen and Hynes.

Fibonacci Numbers

Consider the following recursive function that generates the well-known Fibonacci numbers:

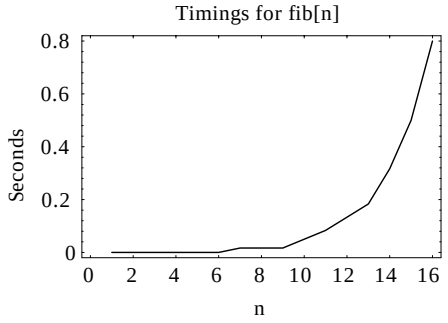
```
In[1]:= fib[n_] := fib[n-1] + fib[n-2]
        fib[0] = fib[1] = 1;

In[2]:= Array[fib, 8]
Out[2]= {1, 2, 3, 5, 8, 13, 21, 34}
```

Unfortunately, the time required to calculate `fib[n]` explodes exponentially as `n` increases.

```
In[3]:= t = Table[Timing[fib[n]][[1,1]], {n, 1, 16}];
```

David B. Wagner is the president and founder of Principia Consulting, a training and consulting firm specializing in Mathematica and other technical software packages. He earned his Ph.D. in computer science at the University of Washington, after which he became an Assistant Professor of Computer Science at the University of Colorado, Boulder. During his academic career he published various papers on concurrency control in distributed database systems, object-oriented parallel programming, parallel simulation, and queueing theory. He now divides his energies between finishing his book on Mathematica programming and teaching Mathematica workshops.

```
In[4]:= ListPlot[t, PlotJoined -> True,
          PlotLabel -> "Timings for fib[n]",
          Frame -> True, FrameLabel -> {"n", "Seconds"},
          FrameTicks -> {Range[0, 16, 2], Automatic}]
```



A look at the execution trace of **fib** reveals the source of the inefficiency:

```
In[5]:= Trace[fib[4], fib[_]]
Out[5]= {fib[4], {fib[3], {fib[2], {fib[1]}, {fib[0]}},
           {fib[1]}}, {fib[2], {fib[1]}, {fib[0]}}}
```

(The second argument to **Trace** causes it to print only those intermediate expressions that match the pattern **fib[_]**). In this small example, **fib[3]** is evaluated once, **fib[2]** is evaluated twice (once during the call to **fib[4]** and once during the call to **fib[3]**), **fib[1]** is called three times, and **fib[0]** is called twice. In fact, the number of times **fib[1]** is called during the evaluation of **fib[n]** is equal to **fib[n-1]**.

```
In[6]:= Table[Count[Flatten[Trace[fib[i], fib[_]]],
                    HoldForm[fib[1]]],
                {i, 1, 12}]
Out[6]= {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144}
```

This is a classic example of the overlapping subproblems problem.

One way to solve this problem efficiently, which is quite straightforward in this simple case, is to perform the computation "bottom up," that is, use results for smaller arguments to calculate results for larger arguments in a monotonically increasing fashion:

```
In[7]:= bufib[n_] := Nest[{#[[2]], Plus @@ #}&, {0, 1}, n][[2]]

In[8]:= Table[bufib[n], {n, 0, 11}]
Out[8]= {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144}
```

A more general and (in the author's opinion) more elegant solution, which is the topic of the present article, is to cache the results of earlier computations. In *Mathematica*, result caching is accomplished by a modest change to the definition of a function:

```
In[9]:= Clear[fib]

In[10]:= fib[n_] := fib[n] = fib[n-1] + fib[n-2]
         fib[0] = fib[1] = 1;
```

Comparing this definition of **fib** to the first one given, we see that the only difference is the prepending of **fib[n]=** to the right-hand side of the definition. Before we explain how this modification works, we give an example of its consequences. Here is the rule set for **fib** before any evaluations are done.

```
In[11]:= ?fib
         Global`fib

         fib[0] = 1
         fib[1] = 1
         fib[n_] := fib[n] = fib[n - 1] + fib[n - 2]
```

After evaluating **fib[3]**…

```
In[12]:= fib[3]
Out[12]= 3
```

…there are more rules for **fib** than before!

```
In[13]:= ?fib
         Global`fib

         fib[0] = 1
         fib[1] = 1
         fib[2] = 2
         fib[3] = 3
         fib[n_] := fib[n] = fib[n - 1] + fib[n - 2]
```

What is going on here? When **fib[n_]** is matched with a particular value for the pattern variable **n**, say n_0, the kernel evaluates the right-hand side of the definition. But the right-hand side is itself a call to **Set**, which results in the assignment of a value to the expression **fib[**n_0**]**. From this time forward, whenever the value of **fib[**n_0**]** is required, no recursive call is made.

The new **fib** function is significantly faster.

```
In[14]:= Timing[fib[16]]
Out[14]= {0.0333333 Second, 1597}
```

Using the old definition of **fib**, the following computation would take approximately 10^8 years:

```
In[15]:= Timing[fib[100]]
Out[15]= {0.133333 Second, 573147844013817084101}
```

Furthermore, because the value of **fib[100]** has been cached, the next call takes no time at all!

```
In[16]:= Timing[fib[100]]
Out[16]= {0. Second, 573147844013817084101}
```

Of course, result caching is a trade-off of memory for time – there are now 102 rules defined for the symbol **fib**. If you evaluate a cached function for extremely large values of its arguments, you may run out of memory. And since the basis of the technique is recursion, it is not difficult to exceed **$RecursionLimit** on large enough input values.

A more subtle difficulty, which is likely to be encountered when solving optimization problems, is that the cached values created for one set of inputs are probably not correct for a different set of inputs (the other problems we consider are examples of this). Thus, the *Mathematica* programmer must provide an easy way for the user to re-initialize the cached functions. Finally, it must be pointed out that during the course of debugging, care must be taken always to **Clear** the cached functions and re-initialize them whenever *any* changes are made to them.

Matrix-Chain Multiplication

The *matrix-chain multiplication* problem can be stated as follows: Given a chain (a sequence) of matrices whose dot product we wish to compute, parenthesize the chain to force the dot products to occur in an order that minimizes the number of scalar multiplications performed. Our presentation of this problem is modeled after [Cormen et al. 1990, sec. 16.1], and we quote results freely from that source. Readers interested in the details should consult this reference.

The total number of scalar multiplications necessary to carry out a matrix-chain product can vary dramatically based on the parenthesization of the chain. Here's an example that shows how important the choice of parenthesization can be. Suppose we wish to compute the matrix-chain product of the following matrices:

```
In[17]:= b1 = Table[Random[], {300}, {10}];
         b2 = Table[Random[], {10}, {300}];
         b3 = Table[Random[], {300}, {10}];
```

There are two mathematically equivalent ways to compute **b1.b2.b3**: as **(b1.b2).b3** and as **b1.(b2.b3)**. In terms of computational effort, however, they are anything but equivalent. Note that the number of scalar multiplications required to compute the dot product of a $p \times q$ matrix with a $q \times r$ matrix is pqr If the matrix product in this example were computed as **(b1.b2).b3**, the total number of multiplications would be:

```
In[18]:= 300*10*300 + 300*300*10
Out[18]= 1800000
```

But if the product were computed as **b1.(b2.b3)**, the number of scalar multiplications would be reduced by a factor of 30*:*

```
In[19]:= 10*300*10 + 300*10*10
Out[19]= 60000
```

The built-in function **Dot** is not smart enough to evaluate this product in the optimal order. Here is how long it takes **Dot** to compute the example product:

```
In[20]:= b1 . b2 . b3; // Timing
Out[20]= {2.55 Second, Null}
```

Note the dramatic speed-up if we force multiplication in the optimal order. (We use explicit calls to **Dot**, rather than parentheses, to effect the optimal multiplication order because the parser converts **b1.(b2.b3)** into **Dot[b1, b2, b3]**.)

```
In[21]:= Dot[b1, Dot[b2, b3]]; // Timing
Out[21]= {0.15 Second, Null}
```

This example motivates the need for an algorithm to determine the optimal matrix-chain multiplication order.

Suppose that the matrix chain to be multiplied is $A_1 \mathbin{\square} A_n$, where A_i has dimensions $p_i \times p_{i+1}$. The problem we will solve is to find the cost of multiplying the matrix chain using an optimal parenthesization, and to produce a nested list of the indices $1,\ldots,n$ that indicates an optimal parenthesization, given the list of matrix dimensions $p = \{p_1,\ldots, p_{n+1}\}$. In the example above, the list of dimensions is {300, 10, 300, 10}, the cost of an optimal evaluation is 60000, and an optimal parenthesization (the only one, in this case) is {1, {2, 3}}.

A brute-force approach to this problem, computing the cost of every possible parenthesization of the matrix chain, would take time that is exponential in n, the length of the chain (more precisely, it is at least as bad as $4^n/n^{3/2}$). This approach is computationally infeasible for all but the smallest values of n.

As a starting point in the search for a better solution, we note that the matrix-chain multiplication problem satisfies the *optimal substructure* property. Optimal substructure means that the optimal solution to the problem is built from the optimal solutions of smaller problems having the same structure as the original. For example, to find the optimal multiplication order for the matrix chain $A_1A_2A_3A_4A_5$ we must consider four alternative ways to split the original problem: $A_1(A_2A_3A_4A_5)$, $(A_1A_2)(A_3A_4A_5)$, $(A_1A_2A_3)(A_4A_5)$, and $(A_1A_2A_3A_4)A_5$. (There are only $n-1$ such splits in a chain of length n since matrix multiplication is noncommutative.) The optimal solution *given a particular split* must consist of optimal solutions to each of the two subproblems. Therefore, the subproblems have the same structure as the original one, but on a smaller scale.

This observation suggests a recursive formulation of the solution. If we denote by $m(i, j)$ the cost of optimally multiplying the matrix chain $A_i \mathbin{\square} A_j$, then a recursive definition for $m(i, j)$ is:

$$m(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i \le k < j} \left[m(i,k) + m(k+1,j) + p_i p_{k+1} p_{j+1} \right] & \text{if } i < j \end{cases} \qquad (1)$$

This equation says that for any choice of where to "split" the chain (given by the index k), the total cost is equal to the cost of the optimal multiplication of all matrices to the left of the split ($m(i, k)$), plus the cost of the optimal multiplication of all matrices to the right ($m(k + 1, j)$), plus the cost of multiplying together these two intermediate results ($p_i\, p_{k+1}\, p_{j+1}$). The optimal cost, then, is the minimum cost over all of the $j \quad i$ possible choice for k.

It is clear as well that the matrix-chain multiplication problem suffers from overlapping subproblems. For example, each of the subproblems $A_1A_2A_3A_4$ and $A_2A_3A_4A_5$ requires

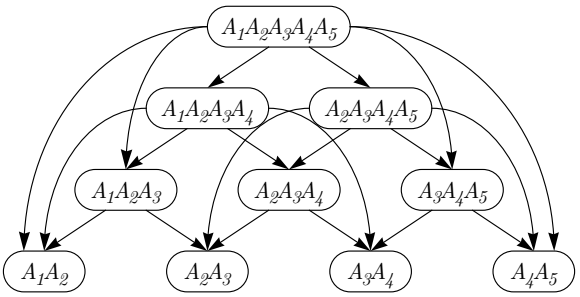


FIGURE 1. Computational structure of the matrix-chain multiplication problem.

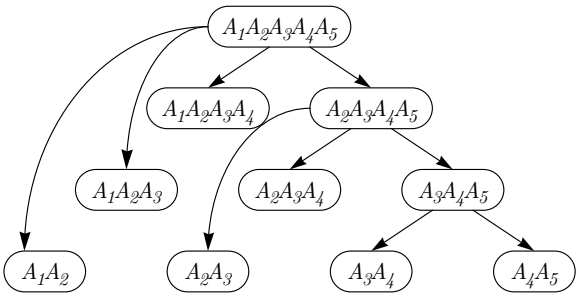


FIGURE 2. Result-cached computation of the matrix-chain multiplication problem.

the solution to the subproblem $A_2A_3A_4$. Likewise, $A_2A_3A_4A_5$ requires the solution to $A_3A_4A_5$, which is already being computed as part of $(A_1A_2)(A_3A_4A_5)$. The entire problem can be viewed as a pyramid-shaped directed graph in which the complete chain is at the apex and the individual pairings are at the base (Figure 1). The number of paths from the apex to any intermediate node in the graph is the number of times the solution to that subproblem will be required. [Cormen et al. 1990] shows that the naïve recursive approach has a computational time complexity that is at least 2^n.

The dynamic programming approach to solving this problem is to compute the bottom row of the pyramid, $m(i, i+1)$ for $i = 1, \ldots, n-1$; then compute the second row from the bottom, $m(i, i+2)$ for $i = 1, \ldots, n-2$; and so on, until finally the apex $m(1, n)$ has been computed. The computational time complexity of this approach is only proportional to n^3.

In contrast to the bottom-up approach to dynamic programming, here is a "top-down" result-caching implementation of the recurrence equation for $m(i, j)$. Note that the form of the solution is a *direct* translation into a *Mathematica* expression of the recursive definition given in equation 1.

```
In[22]:= m[i_, j_] /; i < j := m[i, j] =
           Min[Table[
                 m[i, k] + m[k+1, j] + p[[i]] p[[k+1]] p[[j+1]],
                 {k, i, j-1}] ]
         m[__] = 0
```

(The `m[__]` case is used whenever an expression such as `m[i,i]` is evaluated.)

The result-caching implementation computes subproblems as shown in Figure 2 (the reader should attempt to verify this). For example, when the subproblem $A_2A_3A_4$ needs the solution to the subproblem A_3A_4, no work is done because the latter subproblem has been solved already by $A_3A_4A_5$.

Below, we solve an example problem from [Cormen et al. 1990]. Here are the dimensions of the matrices in the chain.

```
In[23]:= p = {30, 35, 15, 5, 10, 20, 25};
```

A call to `m[1,6]` returns the minimum number of scalar multiplications required for this chain.

```
In[24]:= m[1,6]
Out[24]= 15125
```

Here are all the intermediate results.

```
In[25]:= TableForm[
             Array[m, {5, 6}],
             TableHeadings -> Automatic,
             TableAlignments -> {Center, Right}]
Out[25]//TableForm=
```

	1	2	3	4	5	6
1	0	15750	7875	9375	11875	15125
2	0	0	2625	4375	7125	10500
3	0	0	0	750	2500	5375
4	0	0	0	0	1000	3500
5	0	0	0	0	0	5000

The given algorithm tells us the cost of the optimal multiplication order, but not what that order actually is. To construct the full solution, the `m` function needs to leave a "trail of bread crumbs" as it works. The modified version of `m` shown below stores the table of alternative costs in a local variable called `choices`. Then the index of the minimum cost (the optimal split) is stored in a global variable `s[i, j]`. Although purists may recoil at this use of side-effects, it is defensible in this case for two reasons. First, the most obvious alternatives (having `m` return a list consisting of `{cost, position}`, or passing `s` as a *by-reference* parameter to `m`) make the code more complicated and less efficient. Second, when this code is encapsulated in a package, `s` will be hidden inside a private context, so these side-effects will not be visible to the user.

```
In[26]:= Clear[m]

In[27]:= m[i_, j_] /; i < j := m[i, j] =
           Module[{choices, best},
               choices =
                   Table[m[i, k] + m[k+1, j] +
                           p[[i]] p[[k+1]] p[[j+1]],
                       {k, i, j-1}];
               best = Min[choices];
               s[i, j] = Position[choices, best][[1,1]] + i - 1;
               best ]
         m[__] := 0

In[28]:= m[1,6]
Out[28]= 15125
```

Here is the entire **s** table. (The rule for **s[__]** has been added for convenience.)

```
In[29]:= s[__] := 0

In[30]:= TableForm[
           Array[s, {5, 6}],
           TableHeadings -> Automatic,
           TableAlignments -> {Center, Right}]
```

Out[30]//TableForm=

	1	2	3	4	5	6
1	0	1	1	3	3	3
2	0	0	2	3	3	3
3	0	0	0	3	3	3
4	0	0	0	0	4	5
5	0	0	0	0	0	5

The **s** table shows that the optimal split for the main problem is between the 3rd and 4th matrix (**s[1,6]=3**). The optimal split for the subproblem $A_1A_2A_3$ is between matrices 1 and 2 (**s[1,3]=1**), and the optimal split for the subproblem $A_4A_5A_6$ is between matrices 5 and 6 (**s[4,6]=5**). We can use this information to generate a nested list of indices indicating the optimal parenthesization:

```
In[31]:= {1, 6} //. {i_Integer, j_Integer} /; i < j :>
                    {{i, s[i,j]}, {s[i,j] + 1, j}}
Out[31]= {{{1, 1}, {{2, 2}, {3, 3}}}, {{{4, 4}, {5, 5}}, {6, 6}}}

In[32]:= multorder = % /. {i_, i_} -> i
Out[32]= {{1, {2, 3}}, {{4, 5}, 6}}
```

Now that we have this list, how do we use it? First, suppose we have a list of matrices of the given sizes:

```
In[33]:= A = Table[Random[], {i, 6}, {p[[i]]}, {p[[i+1]]}];
```

We need to turn the parenthesized list of indices, **multorder**, into an expression of the following form:

```
Dot[Dot[A[[1]], Dot[A[[2]], A[[3]]]],
    Dot[Dot[A[[4]], A[[5]]], A[[6]]]]
```

Note that we could not simply have had **m** compute the matrix product as it went along, because that would have wasted a lot of memory and time as non-optimal multiplications (such as **A[[1]].A[[2]]**) were computed along the way.

This transformation ought to be easy, but it turns out to be slightly tricky. The obvious thing to do first is to change all of the heads in **multorder** from **List** into **Dot**. Unfortunately, that destroys the nested structure (which we worked so hard to concoct in the first place!) because **Dot** is **Flat**:

```
In[34]:= multorder /. List -> Dot // FullForm
Out[34]= Dot[1, 2, 3, 4, 5, 6]
```

We can get around this problem by applying **Hold** to the expression before making the substitution. The construct **Hold @@ {multorder}** allows **multorder** to evaluate before the **Hold** head is wrapped around it. (The author prefers this construct to the equivalent **Hold[Evaluate[multorder]]**.)

```
In[35]:= Hold @@ {multorder} /. List -> Dot // FullForm
Out[35]= Hold[Dot[Dot[1, Dot[2, 3]], Dot[Dot[4, 5], 6]]]
```

Next we turn each index i into **A[[**i**]]** using a straightforward delayed-rule substitution:

```
In[36]:= % /. i_Integer :> A[[i]];
```

Now simply apply **ReleaseHold** to the "parenthesized" chain to evaluate all of the **Dot** operators. Since **Dot** evaluates its arguments, more deeply-nested dot products will be evaluated before less deeply-nested ones, thus preserving the "parenthesization." Here is the time required for the optimally-parenthesized matrix-chain product.

```
In[37]:= foo = ReleaseHold[%]; // Timing
Out[37]= {0.05 Second, Null}
```

For this example, it is only slightly faster than passing the entire chain to **Dot**.

```
In[38]:= Timing[bar = Dot @@ A;]
Out[38]= {0.0833333 Second, Null}

In[39]:= foo == bar
Out[39]= True
```

A very clever alternative to the **ReleaseHold[**f**[Hold[**$expr$**]]]** paradigm was suggested by Allan Hayes. The construct **Block[{**$head$**},** f**[**$expr$**]]** effectively gives the kernel a case of temporary amnesia, preventing it from applying any rules associated with *head* until after f**[**$expr$**]** has been constructed and returned from the **Block**. In the present context, here is how it could be used:

```
In[40]:= Block[{Dot},
           multorder /. {List -> Dot, i_Integer :> A[[i]]}];

In[41]:= % == bar
Out[41]= True
```

Within the **Block**, **Dot** behaves like a symbol with no values. When **Block** returns, all of the **Dot** operations inside the returned expression evaluate.

Note that no evaluations except those having head *head* are affected by **Block[{**$head$**}, ...]**, which makes the use of this technique exceptionally straightforward. In fact, we could have created and evaluated the parenthesized chain directly from the table of **s**-values, without going through the nested-list intermediate form:

```
In[42]:= Block[{Dot},
           Dot[1, 6] //.
             Dot[i_Integer, j_Integer] /; i < j :>
               Dot[Dot[i, s[i,j]], Dot[s[i,j] + 1, j]] /.
             Dot[i_, i_] :> A[[i]] ];

In[43]:= % == bar
Out[43]= True
```

Avoiding the intermediate form could not have been accomplished using the `Hold-ReleaseHold` technique because `s[i,j]` would not have evaluated inside the `Hold`.

"Solving a Queueing Model" Revisited

A notorious example of the difficulty of programming the correct "bottom-up" order to the sub-solutions of a dynamic programming problem is multiple-class mean value analysis (MVA) of queueing networks [Reiser and Lavenberg 1980]. In this algorithm, one of the inputs to the problem controls the *dimensionality* of the main loop. To appreciate the difficulty involved, see "Solving a queueing model with *Mathematica*" by Arnold Allen and Gary Hynes [1991]. Here we shall examine the MVA algorithm, but not the types of models that MVA is used to solve. The latter are discussed briefly by Allen and Hynes; for a book-length exposition the interested reader is encouraged to refer to [Lazowska et al. 1984].

We'll begin with single-class MVA, as it is much simpler than multiple-class MVA and can be extended (conceptually, at least) in a straightforward manner. For each of the algorithms, we first describe the solution given by Allen and Hynes, and then show how much easier it is to solve the problem using recursion with result caching.

Consider a queueing network containing a fixed population of N indistinguishable *customers* that circulate among K *servers*. Under the appropriate assumptions about service disciplines and customer routing between the servers [Baskett et al. 1975], it can be shown that the average *waiting time* for a customer at the k-th server is given by

$$W_k(N) = D_k \cdot \left[A_k(N) + 1\right] \tag{2}$$

where D_k is the *service demand* (the expected amount of service time required by a customer) at the k-th server and $A_k(N)$ is the *arrival-instant queue-length* (the expected number of customers that an arriving customer finds) at the k-th server. $A_k(N)$ is a conditional expectation. It was shown independently by [Lavenberg and Reiser 1980] and [Sevcik and Mitrani 1981] that $A_k(N)$ is equal to $L_k(N-1)$, the *unconditional* expected number of customers at the k-th server in a network having one fewer customers. This remarkable result, which is called the *arrival-instant queue-length theorem*, enables us to rewrite the equation for waiting time as:

$$W_k(N) = D_k \cdot \left[L_k(N-1) + 1\right] \tag{3}$$

Given the waiting times at each server, the *throughput* of customers in the network is

$$\lambda(N) = \frac{N}{\sum_{k=1}^{K} W_k(N)} \tag{4}$$

and the average queue length at the k-th server is given by

$$L_k(N) = \lambda(N) W_k(N) \tag{5}$$

(These last two results are consequences of Little's Law [Little 1961].)

Thus, given the set of queue lengths for a network with a particular population, it is possible to use equations 3–5 to compute waiting times, throughputs, and queue lengths for a network with an additional customer. Starting from an empty network (in which the queue lengths are zero), this procedure can be used to determine performance measures for a network with any population, given only the per-server service demands; this entire process is called mean-value analysis [Reiser and Lavenberg 1980]. MVA is usually implemented "bottom up," for reasons that will be explained shortly. The implementation below differs in a few details but captures the essence of the one developed by Allen and Hynes (which they called `CentralServer` for reasons that we won't go into here):

```
In[44]:= CentralServer[nmax_Integer, d_?VectorQ] :=
           Module[{n, L = Map[0&, d], W, lambda},
             Do[ W = d (L+1);
                 lambda = n / Plus @@ W;
                 L = lambda W,
               {n, nmax}
             ];
             {W, lambda, L} ]
```

Note that `d`, `L`, and `W` are vectors; the fact that the arithmetic operators are all `Listable` eliminates the need for the nested loops that are ubiquitous in MVA implementations in other programming languages.

A "top down" implementation of single-class MVA is shown below:

```
In[45]:= W[n_] := d (1 + L[n-1])
         lambda[n_] := n / Plus @@ W[n]
         L[0] := Map[0&, d]
         L[n_] := lambda[n] W[n]
```

This implementation has the virtue of mirroring equations 3–5, and it also eliminates the need for any flow-control code (such as the `Do` loop). Unfortunately this implementation has a time complexity that is exponential in `n`. The reason is that there are two recursive calls to `W[n]` for each `n`: one in the evaluation of `lambda[n]`, and one in the evaluation of `L[n]`.

Once again the problem of overlapping subproblems has reared its ugly head, and once again we beat it back down with result-caching. Although we could cache results for all of the rules, it turns out that the best performance is obtained by caching only `W[n]`. Caching `lambda[n]` and `L[n]` is a waste of time because for any `n`, each is used only once during the computation.

```
In[46]:= W[n_] := W[n] = d (1 + L[n-1])
```

Suppose that the service demands are

```
In[47]:= d = {2., 1., 3., 4};
```

Then the performance measures with 50 customers in the network are

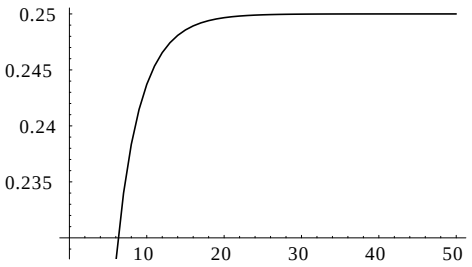
```
In[48]:= $RecursionLimit = 400;

In[49]:= {W[50], lambda[50], L[50]} // Timing
Out[49]= {0.25 Second, {{4., 1.33333, 11.9999, 182.667}, 0.25,
          {0.999999, 0.333333, 2.99997, 45.6667}}}
```

It turns out that the recursive, result-caching algorithm is nearly as fast as the "bottom up" algorithm:

```
In[50]:= Timing[CentralServer[50, d]]
Out[50]= {0.2 Second, {{4., 1.33333, 11.9999, 182.667}, 0.25,
          {0.999999, 0.333333, 2.99997, 45.6667}}}
```

However, this is not a fair comparison: If the bottom-up algorithm is modified to keep track of the intermediate results, it actually turns out to be slightly slower than the result-caching algorithm. Though this example may seem like a "straw man," it is not. Saving all of the intermediate results in MVA usually is considered desirable, as it allows the performance analyst to explore "what if" scenarios about queueing network performance. For example, by plotting throughput as a function of population one can identify a point of diminishing returns:

```
In[51]:= ListPlot[Array[lambda, 50], PlotJoined -> True]
```



For this network, it's clear that allowing more than about 15 customers inside the network at once is not productive.

Multi-Class MVA

MVA can be modified to accommodate queueing networks in which there are several *classes* of customers, each having their own set of service demands. Suppose that there are C different classes of customers. The parameterization of the problem now consists of a $C \times K$ *matrix* of service demands and a vector of population sizes $\mathbf{N} = (N_1, \ldots, N_C)$. The outputs are also either vectors (throughputs) or matrices (waiting times and queue lengths). Equations 3–5 become:

$$W_{c,k}(\mathbf{N}) = D_{c,k} \cdot \left[1 + \sum_{j=1}^{C} L_{j,k}(\mathbf{N} - \mathbf{1}_c)\right] \qquad (6)$$

$$\text{for } c = 1, \ldots, C \text{ and } k = 1, \ldots, K$$

$$\lambda_c(\mathbf{N}) = \frac{N_c}{\sum_{k=1}^{K} W_{c,k}(\mathbf{N})} \qquad \text{for } c = 1, \ldots, C \qquad (7)$$

$$L_{c,k}(\mathbf{N}) = \lambda_c(\mathbf{N}) W_{c,k}(\mathbf{N}) \quad \text{for } c = 1, \ldots, C \text{ and } k = 1, \ldots, K \qquad (8)$$

The notation $\mathbf{1}_c$ in equation 6 indicates a unit vector in dimension c, so the expression $\mathbf{N} - 1_c$ is the given customer population minus one customer of class c. Also, the summation in that equation represents the total number of customers (of all classes) that are encountered by a customer of class c arriving at server k.

To make these equations more concrete, Figure 3 shows the computational structure of this problem when there are two customer classes having three and two customers, respectively. Each subproblem in the graph is labeled "n_1, n_2", where n_i is the number of customers of class i. The solution for population (3, 2) depends upon the solutions for populations (2, 2) and (3, 1), and so on (because of equation 6). In the general case of C customer classes, the computation graph is a C-dimensional lattice. It should be obvious from the structure of the computation that this problem needs to be solved by a dynamic programming algorithm.

In principle, the computation of the subproblems could be done in any order that does not violate any dependencies. In practice, one of two methods is used: Either the lattice is traversed one dimension at a time (along the diagonals in the figure) or it is traversed in order of increasing total customer population (along the horizontal levels in the figure). The former method typically is used when writing a program for exactly C customer classes; then the flow-control can be accomplished simply by using C nested loops. (Note, however, that the resulting code works only for the given number of customer classes). The latter method typically is used when the intent is to create a program that works for *any* number of customer classes; this is the approach taken by Allen and Hynes. The "catch" is that it is not easy to enumerate the populations in the proper order for an arbi-

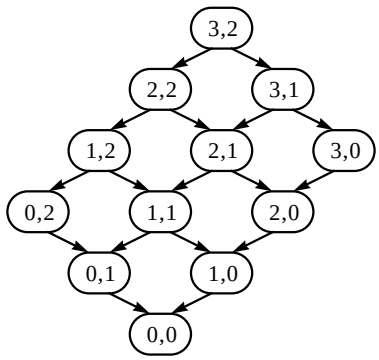


FIGURE 3. Computation graph for 2-class MVA with population **N** = (3,2).

trary number of customer classes. Allen and Hynes' solution hinges upon the following function, which generates all sub-populations of size `n` (the first parameter) drawn from the total population vector `max` (the second parameter):

```
In[52]:= Partitions[n_Integer, _] := {} /; n < 0
         Partitions[0, max_] := {Map[0&, max]}
         Partitions[n_Integer, {max1_, rest___}] :=
           If[max1 < 0 || Plus[max1, rest] < n,
             {},
             Join[
               Prepend[#, max1]& /@ Partitions[n-max1, {rest}],
               Partitions[n, {max1-1, rest}]]]

In[53]:= Partitions[3, {3, 2, 1}]
Out[53]= {{3, 0, 0}, {2, 1, 0}, {2, 0, 1}, {1, 2, 0},
          {1, 1, 1}, {0, 2, 1}}
```

This function reduces the problem of finding all sub-populations to two simpler problems: finding all sub-populations having no customers of class one, and finding all sub-populations having at least one customer of class one. (Allen and Hynes did not explain the insight that led to this non-obvious approach, but the reasoning is similar to Buzen's convolution recurrence for computing the *normalization constant* of multiple-class queueing networks [Buzen 1973].) Given the `Partitions` function, the rest of the code for multi-class MVA is not much more complicated than for the single-class case; interested readers should refer to the function `MultiCentralServer` in [Allen and Hynes 1991].

Multi-class MVA is an excellent example of the trickiness of the bottom-up method of dynamic programming, and it is here that the result-caching technique really shines. Our strategy will be to use almost the exact same rule definitions as for the single-class case, except that the argument to each function will now be a *list* of integers (the population vector **N**). The only tricky part of the algorithm is the recursive calling of `L`: `W[{n1,n2,...}]` must call `L[{n1-1,n2,...}]`, `L[{n1,n2-1,...}]`, and so on. For convenience, we define the following utility function that subtracts one from the `c`-th element of a list:

```
In[54]:= minus1[c_, n_] := MapAt[#-1&, n, {c}]
```

`W[n]` can then simply call `L[minus1[c,n]]` for each possible value of `c`.

Referring again to equation 6 we see that `W[n]` needs to sum `L[minus1[c,n]][[j,k]]` over the index `j` (that is, by columns); and referring to equation 7 we see that `lambda[n]` needs to sum `W[n][[c, k]]` over the index `k` (by rows). The following two utility functions are by no means necessary, but serve to make the code slightly cleaner:

```
In[55]:= sumc[m_] := Plus @@ m

In[56]:= sumk[m_] := Apply[Plus, m, {1}]
```

Now we're ready to write down the main rules:

```
In[57]:= Clear[W, lambda, L];

In[58]:= W[n_] := W[n] =
             Table[d[[c]] (1 + sumc[L[minus1[c, n]]]),
               {c, Length[n]}];

In[59]:= lambda[n_] := n/sumk[W[n]];

In[60]:= L[{(0)..}] := Map[0&, d, {-1}];
         L[n:{__?NonNegative}] := L[n] = lambda[n] W[n];
         L[_] := Map[0&, d, {-1}];
```

There are now three rules for `L` rather than two, since there are now two boundary conditions: all of the sub-populations can be zero or one of the sub-populations can be negative. The latter case, which happens along the borders of the lattice, is handled by the final "catch-all" rule for `L`.

In contrast to the single-class case, we have cached the results of `L` as well as `W`. However, it turns out that caching the boundary conditions for `L` is counterproductive, because any given boundary condition simply isn't evaluated that many times, and the code for the boundary conditions isn't all that complicated. In fact, a simpler and more effective enhancement is to pre-evaluate the right-hand side of each boundary condition. This can be done by initializing `d` before initializing `L`, and changing the `SetDelayed` used in the definition of each boundary condition to `Set`. (This is a sensible thing to do anyway, since if `d` is changed all of the rules have to be re-initialized to eliminate any incorrect cached values.) For convenience we wrap up all of this in a function called `MVAinit`:

```
In[61]:= Clear[MVAinit]

In[62]:= MVAinit[d_?MatrixQ] :=
         (   Clear[W, lambda, L];
             W[n_] := W[n] =
               Table[d[[c]] (1 + sumc[L[minus1[c, n]]]),
                 {c, Length[n]}];
             lambda[n_] := n/sumk[W[n]];
             L[{(0)..}] = Map[0&, d, {-1}];
             L[n:{__?NonNegative}] := L[n] = lambda[n] W[n];
             L[_] = Map[0&, d, {-1}];
         )
```

Here is an example involving three customer classes (with populations 10, 5, and 2) and two queueing centers. Note that `d` is now a matrix.

```
In[63]:= d = {{1., 3.}, {2., 4.}, {0., 5.}};

In[64]:= Timing[MVAinit[d]; W[{10, 5, 2}]]
Out[64]= {2.95 Second, {{1.51276, 49.4617},
            {2.98046, 66.0391}, {0., 82.1711}}}
```

As before, because the function values have been cached the evaluation of other performance measures is nearly instantaneous:

```
In[65]:= Timing[L[{5, 5, 0}]]
Out[65]= {0. Second, {{0.293089, 4.70691},
          {0.412995, 4.58701}, {0, 0}}}
```

Less obvious, but just as important, is that the analysis can be extended to larger populations without having to redo any work:

```
In[66]:= Timing[W[{11, 6, 2}]]
Out[66]= {0.916667 Second, {{1.5333, 55.4001},
          {3.02507, 73.9499}, {0., 92.0933}}}
```

(Building up the answer to large problems on top of smaller ones is also an easy way to avoid the **$RecursionLimit** barrier.)

Discounting the three trivial utility functions, the result-caching code for multi-class MVA consists of five "one-liners" – only one more than for the single-class algorithm. (The reason for the additional rule was explained earlier.) In addition, the code is completely general: It works for any number of customer classes and any number of queueing centers. The code even works for the single-class case, as long as one remembers to initialize **d** as a matrix with one row rather than as a vector:

```
In[67]:= $RecursionLimit = 600;

In[68]:= MVAinit[{{2., 1., 3., 4.}}]; W[{50}]
Out[68]= {{4., 1.33333, 11.9999, 182.667}}
```

Most important, however, is the fact that the code is a straightforward translation of the equations that define the problem, and requires no explicit flow-control. This makes development of a working algorithm much easier than using the bottom-up approach.

Perhaps surprisingly, the result-caching implementation is also faster than the algorithm developed by Allen and Hynes. (Their **Queueing.m** package can be found on *MathSource* as item 0205-636.)

```
In[69]:= Remove[CentralServer, Partitions];

In[70]:= << Queueing.m

In[71]:= MultiCentralServer[{10, 5, 2}, d] // Timing
Out[71]= {3.9 Second, {{50.9745, 69.0195, 82.1711},
          {0.196177, 0.0724432, 0.0243395},
          {0.512681, 16.4873}, {0.341063, 1.}}}
```

Note that the first component of the **MultiCentralServer** solution is a list of W_c (the total waiting time for each class) rather than the matrix $W_{c,k}$; we can see that our answer matches theirs by summing **W[{10,5,2}]** over the index k:

```
In[72]:= sumk[W[{10, 5, 2}]]
Out[72]= {50.9745, 69.0195, 82.1711}
```

Strangely, the speed of the Allen and Hynes algorithm is sensitive to the order of the customer classes. Here is an example:

```
In[73]:= pops = NestList[RotateRight, {20, 1, 1}, 2]
Out[73]= {{20, 1, 1}, {1, 20, 1}, {1, 1, 20}}

In[74]:= Timing[MultiCentralServer[#, d]][[1]]& /@ pops
Out[74]= {1.75 Second, 2.11667 Second, 3.11667 Second}
```

This variability in execution times is attributable to the **Partitions** function. The lion's share of the execution time is spent figuring out the order in which to evaluate the subproblems!

```
In[75]:= Timing[Do[Partitions[i, #], {i, 22}]][[1]]& /@ pops
Out[75]= {0.983333 Second, 1.38333 Second, 2.26667 Second}
```

The result-caching code has no such performance quirks.

```
In[76]:= Timing[MVAinit[d]; W[#]][[1]]& /@ pops
Out[76]= {1.18333 Second, 1.15 Second, 1.16667 Second}
```

Summary

Caching the results of function calls is a powerful and intuitive technique. It makes the solution to many dynamic programming problems about as simple as translating the recursive formulation of a problem into *Mathematica* code. Furthermore, the programmer does not need to write any flow-control code, which is a great advantage when the optimum evaluation order is either not obvious or simply hard to program. Finally, function result caching is relatively efficient, thanks to the speediness of *Mathematica*'s built-in pattern-matching engine.

The only drawbacks to function result caching are that it uses memory to store the intermediate results and that it requires a lot of stack space. The former "bug" is sometimes a "feature," as access to intermediate results frequently is necessary in optimization problems (as we saw in the optimal matrix-chain multiplication problem). The second drawback is the more serious one, but generally can be overcome by "bootstrapping" the solution to large problems from smaller ones.

In summary, dynamic programming and function result caching belong in every *Mathematica* programmer's toolbox!

Package Implementation Notes

The electronic supplement contains the package **OptimalDot.m**, which not only implements the optimal matrix-chain multiplication algorithm given in the article, but also overrides the built-in **Dot** function to use this algorithm whenever three or more matrices are passed to **Dot**. An option called **Optimize** (with default value **True**) has also been added to **Dot**. Setting **Optimize** to **False** can be used to prevent **Dot** from calculating the optimal multiplication order, which is a waste of time for small matrices or for chains in which all of the matrices are square. The default value for **Optimize** can be changed using **SetOptions**.

The function **m** and the variables **p** and **s** are encapsulated within a private context and so are not directly accessible to the user. A noteworthy trick is that, after the rules for **m** are

initialized, they are saved in a private variable using the assignment

```
mdv = DownValues[m]
```

Subsequently, each time the algorithm is run, the rule set for `m` is restored to a "virgin" state in one fell swoop using

```
DownValues[m] = mdv
```

The values of `s` do not need to be initialized since they are simply overwritten during each run.

Finally, the code uses Allan Hayes' `Block` technique (described in the article) to perform the actual dot products.

The electronic supplement also contains the package `MeanValueAnalysis.m`, which contains the `MVAinit` function developed in the article. The symbol names `W`, `lambda`, and `L` are declared within the package context, `MeanValueAnalysis``, so that users of the package do not inadvertently wipe out their own definitions for these symbols when `MVAinit` is called.

References

Allen, A.O., and G. Hynes. 1991. Solving a queueing model with *Mathematica*. *The Mathematica Journal* 1(3): 108–112.

Baskett, F., K.M. Chandy, R.R. Muntz, and F.G. Palacios. 1975. Open, closed, and mixed networks of queues with different classes of customers. *Journal of the ACM* 22(2): 248–260.

Buzen, J.P. 1973. Computational algorithms for closed queueing networks with exponential servers. *Communications of the ACM* 16(9): 527–531.

Cormen, T.H., C.E. Leiserson, and R.L. Rivest. 1990. *Introduction to Algorithms*. McGraw-Hill, New York, NY. (Also available from MIT Press, Cambridge, MA.)

Jain, R. 1991. *The Art of Computer Systems Performance Analysis*. Wiley, New York, NY.

Lavenberg, S.S., and M. Reiser. 1980. Stationary state probabilities of arrival instants for closed queueing networks. *Journal of Applied Probability* (December).

Lazowska, E.D., J. Zahorjan, G.S. Graham, and K.C. Sevcik. 1984. *Quantitative System Performance: Queueing System Analysis Using Queueing Network Models*. Prentice-Hall, Englewood Cliffs, NJ.

Little, J.D.C. 1961. A proof for the queueing formula: $L = \lambda W$. *Operations Research* 9(3): 383–387.

Reiser, M., and S.S. Lavenberg. 1980. Mean-value analysis of closed multichain queueing networks. *Journal of the ACM* 25(2): 313–322.

Sevcik, K.C., and I. Mitrani. 1981. The distribution of queueing network states at input and output instants. *Journal of the ACM* 28(2): 358–371.

David B. Wagner
Principia Consulting
3841 Orion Court, Boulder, CO 80304-1024
dbwagner@princon.com

The electronic supplement contains the packages `OptimalDot.m` and `MeanValueAnalysis.m`.