The Role of Dynamic Programming & Control Structures in Performance*

Robert Bernecky
Snake Island Research Inc
18 Fifth Street, Ward's Island
Toronto, Ontario M5J 2B9
Canada
+1 416 203 0854
bernecky@acm.org

Abstract

Dynamic programming is used infrequently by APL programmers, in spite of its ability to reduce the computational effort required to solve many problems. This paper uses the *String Shuffle* problem as the basis for a comparison of brute force, recursive, and dynamic programming algorithms. Simple algorithms for each of these approaches are designed and evaluated, then individually optimized and re-evaluated, to show the benefits of dynamic programming.

The dynamic programming algorithm is then recast to use flow control structures recently introduced into ISI J and APL*PLUS III. Use of control structures in conjunction with dynamic programming results in orders of magnitude performance improvement over brute force and naive recursive algorithms. APL*PLUS III control structures are shown to provide a performance improvement of up to 30% over GOTO-based loops.

*This paper originally appeared in the ACM SIGAPL APL95 Conference Proceedings. [Ber95]

Keywords

dynamic programming, flow control structures, control structures, performance, optimization, APL, J, puzzles, interpreter overhead.

1 Introduction

The *String Shuffle* problem is a simple puzzle that will serve to demonstrate why brute force solutions, although often elegant and simple to express in APL, may be impractical for solving many problems because of the large computational resources they require. The use of recursion, dynamic programming, and flow control structures permits solutions to the same problems in far shorter times.

The string shuffle problem may be stated as: Given a string `s` and two substrings `si` and `sj`, such that `(⍴s)=⍴si,sj`, can `s` be constructed by alternately picking zero or more leading characters from the remaining parts of `si` and `sj`? If so, then `s` is said to *shuffle* the substrings.

For example, the string `'NUTS!'` shuffles substrings `'NT'` and `'US!'`, but not `'NT'` and `'U!S'`, because of the restriction on picking characters from the front of each substring. A variant on the problem asks *In how many ways can the string be constructed?*, but we shall concentrate on the simpler decision problem.

We will first consider three basic APL algorithms for solving the string shuffle problem. The algorithms will then be tuned to improve their performance. Finally, the dynamic programming algorithm, recast to use the flow control structures recently made available in Iverson Software's ISI J and Manugistics' APL*PLUS III, will be compared to the GOTO-based algorithm. The approaches to be examined are:

- Brute force
- Naive and intelligent recursion
- Dynamic programming

2 The Brute Force Solution

One brute force solution is to interleave the substrings in all possible ways, then determine if any of them match the string. The hard part is to figure out how to generate all the possible combinations of the substrings.

We gain insight into the problem by viewing it backwards: How can the string be broken up into two substrings of the proper length, while preserving the required character ordering? Compression, of course! A suitable compression vector, `cv`, with as many 1s in it as there are elements in `si`, can be used to extract a substring from `s` of the same length as `si`. Similarly, the logical negation of `cv` can be used to extract a substring of the same length as `sj`. If any pairs of these extracted substrings match `si` and `sj`, then we have found a solution to the string shuffle problem. For example:

```
  1 0 1 0 0/'Nuts!' ⍝ Generate si
Nt
  0 1 0 1 1/'Nuts!' ⍝ Generate sj
us!
```

The compression technique reduces the string shuffle problem to that of generating the complete set of possible compression vectors. Since we know the compression vector must have `⍴si` 1s in it, the problem is easy. First, generate all possible compression vectors for `s`. This can be done in APL and J as follows: [1]

```
APL:  i←⍉((⍴s)⍴2)⊤⍳2*⍴s
J:    i=.#: i. 2∧#x.
```

Then discard those that do not have `⍴si` 1s in them:

```
APL:  j←((⍴si)=+/i)⌿i
J:    j=.((#si)=+/"1 i)#i
```

From here on, things get easy: Use `j` to generate all potential `si` lists from `s`, use `~j` to generate all potential `sj` lists, then match each pair of those against `si,sj`. In the defined verb presented as APL in Figure 1 and as J in Figure 2, this is done by turning the two generated sets of lists into tables, then catenating the two tables. An alternative approach would be to use indexing to generate potential `s` lists, then match them against `s`. This approach is slower and more space-hungry, as it requires generation and use of integer index sets instead of Boolean compression vectors.

We define two utility functions to simplify readability and porting across APL systems: `D` is disclose or open; `C` is *catenate with enclose*, which encloses or boxes each of its arguments, then catenates those results to form a two-element list:

[1]All code in this paper uses index origin zero. The J code contains extra white space to make reading easier for those who are not familiar with J.

ISIAPL	APL*PLUS III	J
D:>y	D:⊃y	D=.>
C:x,⍤<y	C:(⊂x),⊂y	C=.,&<

```
r←s SBF y;si;sj;i
⍝ Brute force string shuffle
⍝ s is string to generate
⍝ Substrings
si←D y[0] ⋄ sj←D y[1]
⍝ # of matches
r←+/(s bf⍴si)∧.=si,sj
r←s bf len;si;sj;i;j;k;m
⍝ Generate all shuffles.
i←⍉((⍴s)⍴2)⊤⍳2*⍴s
⍝ Mask on char distn.
j←(len=+/i)≠i
k←1↑⍴j ⋄ j←,j ⋄ m←(⍴j)⍴s
si←(k,len)⍴j/m
sj←(k,(⍴s)-len)⍴(~j)/m
r←si,sj
```

Figure 1: Brute force APL string shuffle algorithm

In computer science, simple algorithms are either the best or the worst thing going. The brute force string shuffle algorithm represents the latter case. As Figure 3 shows, both the APL and J versions of the algorithm fail with string lengths less than 20, but they nonetheless consume an inordinate amount of computer time and storage while doing so. [2] It is clear that better approaches are called for, and recursion might be one such approach.

3 Recursion

The use of recursion seems a natural for this sort of application, but it must be used with care, as we shall see. An empty string `s` will shuffle two empty substrings `si` and `sj`. A non-empty string `s` will shuffle `si` and `sj` if its first character matches the first character of `si` and `1↓s` shuffles `1↓si` and `sj`. `s` will also shuffle `si` and `sj` if its first character matches the first character of `sj` and `1↓s` shuffles `1↓sj` and `si`. Such a recursive definition is shown as APL in Figure 5 and as J in Figure 6. Like `SBF`, the recursive implementation gives the number of possible shuffles.

[2]It is interesting to observe that the execution times are about the same for APL and J. This is in sharp contrast to what we shall observe later.

```
SBF =. 3 : 0
: NB. x. is string to match
 si =. >0{ y.  NB. Substrings
 sj =. >1{ y.
 NB. All subsets
 i =. #: i. 2 ^ # x.
 j =. ((#si)=+/"1 i)#i NB. mask
 ss =. (j# x.),"1 (-.j)#x.
 NB. Build & catenate substrings
+/ ss -:"1 si,sj NB. # of matches
)
```

Figure 2: Brute force J string shuffle algorithm

These recursive functions avoid creating the huge intermediate arrays that the brute force approach uses, and thereby avoid the storage limitations that kill `SBF`. However, the curves in Figure 4 labeled *SR* and *SR-J*, representing the recursive APL and J functions, respectively, show us that the performance of these functions is also dismal – they are doing an exponential amount of work, just as the brute force algorithm does.

Later on, we will see how to reduce the workload to a reasonable size. Perhaps there is a way to break the problem into sub-problems that are more tractable in terms of required computation. Dynamic Programming is one approach we might take to achieve that end.

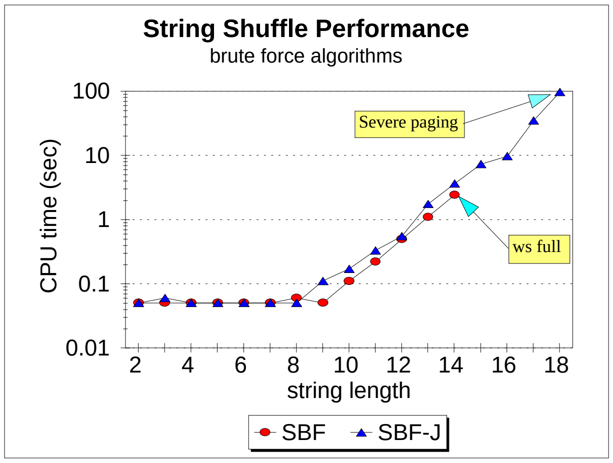

Figure 3: Performance of brute force algorithms

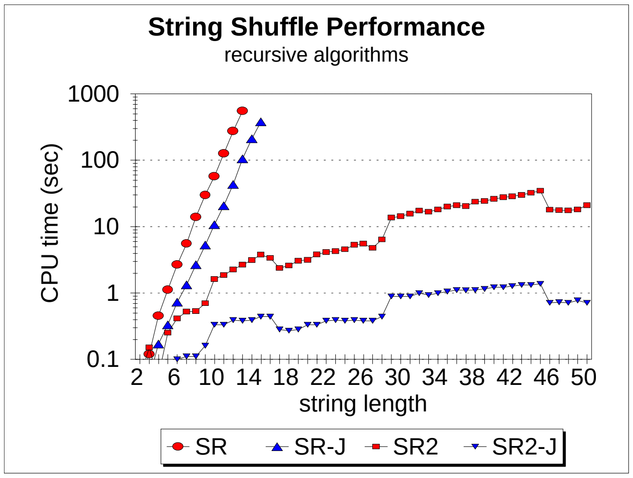

Figure 4: Performance of recursive algorithms

4 Dynamic Programming

Dynamic Programming is used infrequently by APL programmers, perhaps because they are not familiar with it, or perhaps because explicit looping does not fit in with what some describe as "good APL coding style." Certainly, it *is* rare that looping APL code will outperform straight-line code. Nonetheless, as we shall see, dynamic programming is a very effective tool for the programmer, with performance benefits which are difficult or impossible to achieve in non-iterative forms.

Problems that are good candidates for dynamic programming are those that exhibit *optimal substructure*. Optimal structure means that an optimal solution to the problem contains sub-problems whose solutions are also optimal.

As an example, consider P, the optimal (shortest) path through some graph, that happens to pass through two nodes x and y. In that case, the path taken from x to y is also optimal – if it were not optimal, then P could be shortened by taking the new shorter path from x to y. This contradicts the assumption of optimality of P, hence the path from x to y must also be optimal. This insight suggests that P can be constructed by assembling it from smaller paths, until the final solution is reached. Bellman investigated this approach in 1957 [Bel57].

A key to understanding dynamic programming is to recognize when redundant work on sub-problems is being done, and then stop doing that redundant work. If the same sub-problem is being solved more than once, the problem is very likely a good candidate for a dynamic programming solution.

Good introductions to dynamic programming and other effective algorithms can be found in Baase [Baa88] and Cormen, Leiserson, and Rivest [CLR90]. Some articles published in recent APL conferences discussing dynamic programming include Lin, et al. [LB90] and Kimbrough [Kim95].

In string shuffle, both the recursive and brute force algorithms examine the tail end of the string repeatedly. A dynamic programming algorithm avoids this, looking at each character a minimal number of times. An example will show how it works.

We build `m`, a Boolean *dynamic programming matrix*, of shape `((1+⍴si),1+⍴sj)`. If `1=m[i;j]`,

```
 r←s SR y;si;sj;i;j;k
 ⍝ Recursive string shuffler.
 si←D y[0] ⋄ sj←D y[1]
 →(1≠⍴s)⍴R ⋄ r←+/s=si,sj ⋄ →0
R:→(0≠⍴si)⍴R2 ⋄ i←0 ⋄ →R3
R2:
i←(s[0]=si[0])×(1↓s)SR (1↓si)C sj
R3:→(0≠⍴sj)⍴R4 ⋄ j←0 ⋄ →R5
R4:
j←(s[0]=sj[0])×(1↓s)SR si C 1↓sj
R5:r←i+j
```

Figure 5: Recursive APL string shuffle algorithm

then `(i+j)↑s` shuffles `i↑si` and `j↑sj`. The first row and column, which correspond to taking zero characters from one of the substrings, are initialized by comparison against the other substring. This can be performed in APL as follows:

```
m[;0]←1,∧\si=(⍴si)↑s
m[0;]←1,∧\sj=(⍴sj)↑s
```

The definition of `m[i;j]` given in the previous paragraph is implemented by noting that a shuffle will exist if either of the following conditions exist:

- `cc`, the current character of `s` matches `si[i-1]`, the current character of `si` and a shuffle exists for the previous prefix of `si` and the current prefix of `sj`:

  ```
  ti←m[i-1;j]∧cc=si[i-1]
  ```

- Or, `cc`, the current character of `s` matches `sj[j- 1]`, the current character of `sj` and a shuffle exists for the previous prefix of `sj` and the current prefix of `si`:

  ```
  tj←m[i;j-1]∧cc=sj[j-1]
  ```

Two loops take us across both substrings, entering elements of `m`, until both substrings have been

```
SR =. 3 : 0
: NB. x. is string to match
 NB. s1;s2 substrings
 si =. >0{ y.
 sj =. >1{ y.
 if. 1 >: $x. do.
   x. = si,sj
  else.  NB. Recurse on si
   i=. 0
   if. 0 ~: $si do.
    i =. (}. x.) SR (}.si);sj
    i =. i * x. =&{. si
   end.
   j=. 0 NB. Recurse on sj
   if. 0 ~: $sj do.
    j =. (}. x.) SR si;}.sj
    j =. j * x. =&{. sj
   end.
  i+j
 end.
)
```

Figure 6: Recursive J string shuffle algorithm

exhausted. Thus, the complete computation requires `(⍴si)×⍴sj` iterations. When both loops have been exhausted, at least one shuffle exists if `1=¯1 ¯1↑m`. For the two `"NUTS!"` examples given earlier, the final dynamic programming matrices look like this, when annotated with their respective arguments:

```
   " U S !
 " 1 0 0 0
 N 1 1 0 0
 T 0 1 1 1

   " U ! S
 " 1 0 0 0
 N 1 1 0 0
 T 0 1 0 0
```

Although `SDyn`, shown as APL in Figure 7, does not directly provide the number of possible shuffles, the dynamic programming matrix `m` permits backtracking to determine that information.

The performance of `SDyn` on the string shuffle problem is shown as the curve labeled *SDyn* in Figure 12. Ignoring the other curves on the graph for the moment, note that execution time remains under one second for a string length of 24, long after the competition (brute force and simple recursion) have failed.

It is important to understand some of the factors contributing to these times before seeking other methods of performance improvement. We shall discuss the impact of interpreter overhead briefly in the following section before resuming our efforts to improve the performance of the string shuffle algorithms.

```
  r←s SDyn y;m;i;j;si;sj;ti;tj;cc
  ⍝ Dynamic Prog. string shuffle
  si←D y[0] ⋄ sj←D y[1]
  m←(1+(⍴si),⍴sj)⍴0 ⍝ DP matrix.
  m[;0]←1,∧\si=(⍴si)↑s ⍝ Prime
  m[0;]←1,∧\sj=(⍴sj)↑s ⍝ pump
  j←1
 lpj:→(j>⍴sj)⍴lpjz ⍝ FOR loop
  i←1
 lpi:→(i>⍴si)⍴lpiz ⍝ FOR loop
  cc←s[i+j-1] ⍝ Current char in s
  ⍝ Match on si, sj
  ti←m[i-1;j]∧cc=si[i-1]
  tj←m[i;j-1]∧cc=sj[j-1]
  m[i;j]←ti∨tj
  i←i+1 ⋄ →lpi
 lpiz:j←j+1 ⋄ →lpj
 lpjz:r← ¯1 ¯1 ↑m
```

Figure 7: Dynamic programming APL string shuffle algorithm

5 Performance and Execution Overhead

Performance of the various string shuffle algorithms is strongly algorithm- and data-sensitive, as can be seen from Figure 8, which shows string shuffle times for all algorithms on strings of varying lengths. [3] Since the y-axis of the plot is a log scale, the increase in execution time required is much greater than the curve suggests. Since these differences are so dramatic, it may be instructive to take a brief look at the origins of APL interpreter execution overheads.

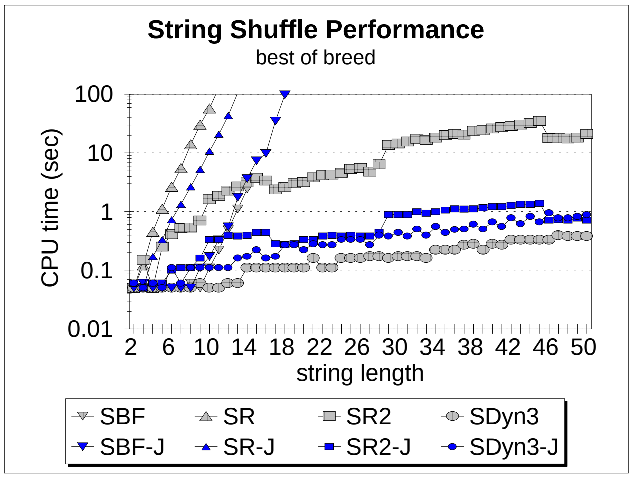

Figure 8: Comparative performance of APL and J algorithms

5.1 APL Execution Overheads

All successful APL systems to date have been interactive and interpreter-based. Unlike compiled lan-

[3]Timings were performed in APL*PLUS III and ISI J Version 2.04beta, running under Windows 3.1 on a 16 megabyte IBM-compatible 486/33 PC. The discontinuities in the plot are due to grossly inadequate timer resolution on PCs. Users of profiling tools should be aware that results of timing can be skewed by coarse timing facilities.

guages, the interactive nature of APL lets a user dynamically alter any aspect of the environment. The type, rank, and shape of arrays can change, functions can change their valence (e.g., from monadic to dyadic), and the syntax class of named objects can change underfoot, as when someone expunges a variable and then defines a function with the same name. These factors restrict the amount of interpreter analysis that can usefully be done on an APL expression to improve its performance. In particular, such factors require that the interpreter validate arguments to verbs and adverbs at each primitive execution, determine how the primitive is to be executed (e.g., "`Boolean+real`" has to be done using real arithmetic), perform conformability checks to ensure that arrays are of appropriate shapes, and perform storage management operations to create results and discard arguments. The net result is that there is substantial overhead associated with the execution of each APL primitive, independent of its element count. In computations on arrays, this startup overhead is amortized over the array elements. The overhead for such operations on arrays of a dozen elements or more becomes acceptably low, as shown in Figure 9. [4]

The graph also shows that, unfortunately, operations on scalars and arrays of a few elements can take 25 times as long per element as operations on large arrays: overhead dominates the execution time for scalar and small array operations. Execution profiles taken at the system level confirm that this is the case. The execution time for such operations can, therefore, be estimated quite closely by counting the number of primitives executed. The best way to improve the performance of such applications is to reduce the number of primitives executed. This technique will be used in the next section to reduce the execution time of the algorithms presented thus far.

[4]This plot is taken from a mainframe APL system, as PC clock resolution was inadequate to make these measurements.

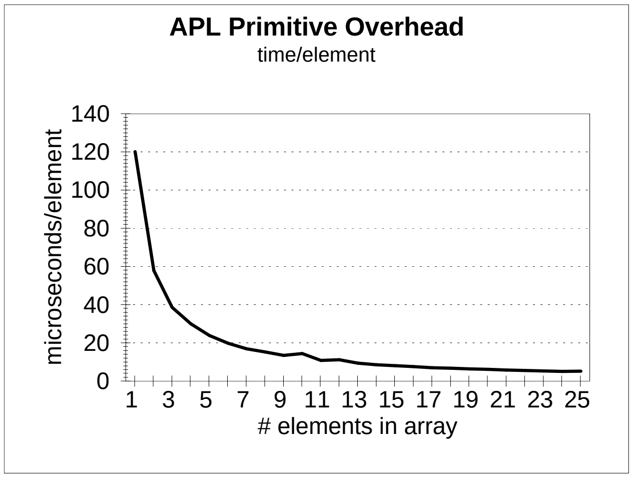

Figure 9: APL primitive startup overhead

An additional overhead in APL execution is that associated with a defined function call. This overhead is largely imposed by APL's dynamic scoping rules. As with per-primitive overhead, its impact on execution time depends heavily on the amount of computation performed within the defined function. If the amount of computation is small, then the time taken by the function call dominates.

In both cases, a compiled version should perform significantly better than an interpreted version. A compiler has the luxury of being able to spend time examining the application to deduce array properties [Ber93] and to analyze the names used in the computation. These steps facilitate faster function calls, removal of certain run-time conformability checks, better storage management, and so on. In addition, tail recursions can be turned into iterations, thereby allowing further optimizations to take place in `SR`.

5.2 The Impact of Interpreter Overhead

With the above knowledge of interpreter behavior in hand, a look at the string shuffle algorithms begins to reveal why they perform as they do. Figure 8 shows most of the algorithms discussed in this article, and offers a quick way to compare their performance. Consider the brute force (SBF) approach first: For very small arrays, brute force beats the recursive (SR) and dynamic programming (SDyn) approaches, because it has the fewest primitives executed and array sizes are small enough that per-primitive overhead dominates the computation on all algorithms. Simple recursion (SR) in APL and J is always a loser for this problem, taking 10 – 100 times as long as the competition on an 8-character argument. This is because it executes many primitives per call, all operating on very small arrays. Furthermore, lines R2 (and R4) of SR fail to avoid the recursive call when the result is already known to be zero because s[0]≠si[0]. Hence, SR does an immense amount of unnecessary work.

As argument sizes grow, the nugatory work performed by SBF and SR begin to take its toll, and SDyn starts to shine. With arguments of 11 characters, it breaks even with SBF, and by 14 characters, dynamic programming is running 7 times faster than the brute force approach. Naive recursion looks even worse than SBF, taking two orders of magnitude longer to execute. Can the performance of these techniques be improved using knowledge of interpreter characteristics? If so, what effect, if any, will that have on their performance?

6 Optimizing APL Applications

The performance of real-life APL applications can usually be best improved by use of profiling tools such as ⎕fm in SHARP APL and ISIAPL [BB93] or ⎕mf in APL*PLUS [Ber89]. However, for this simple exercise, we restrict our attention to reducing the number of calls to defined and primitive functions and to reducing the amount of work they do. We will also appeal to some basic computational complexity arguments to decide where such efforts would be wasted.

6.1 Optimizing SBF

Taking a brief look at SBF, we note that an s-element string generates arrays with 2*⍴s elements, thereby placing it in the class of exponentially expensive functions whose execution time doubles for each element added to the argument. This computational complexity makes such functions impractical to use for any but the smallest arguments.

As Figure 3 shows, exponential growth in cpu time is a bad sign – the execution time of SBF grows so rapidly that any efforts to improve its performance are lost when the string grows a few characters longer. SBF's exponential growth in space is manifested by the fact that it fails with a workspace full at 15 characters in a 3.7 megabyte workspace. Therefore, we shall ignore the brute force technique, as it is unacceptably expensive in practice.

6.2 Optimizing SR

Several optimizations of SR are possible, but the one with the biggest payoff is elimination of the spurious recursive calls to SR when we do not care about the result, because we know in advance of the call that no match can possibly exist. In lines R2 and R4 of SR in Figure 5, recursive calls are made in which the result of the calls is multiplied by a Boolean. Rewriting the code to avoid the call when the Boolean is zero, shown as SR2 as APL in Figure 10 and as J in Figure 11, pays off handsomely. The performance of this better recursive solution is shown as curves *SR2* and *SR2-J* in Figure 4.

The J version of SR2 is an order of magnitude faster than the APL version. A cursory benchmark to determine if the cause was APL's *shadowing* of local variables at function call time was not informative.

```
 r←str SR2 s;si;sj;i;j;k
 ⍝ Recursive string
 si←D s[0] ◇ sj←D s[1]
 r←str≡si,sj ◇ →(1=⍴str)⍴0
 j←0 ◇ →(0=⍴si)⍴rec3
 →(str[0]≠si[0])⍴rec3
 j←(1↓str) SR(1↓si) C sj
rec3:k←0 ◇ →(0=⍴sj)⍴rec5
 →(str[0]≠sj[0])⍴rec5
 k←(1↓str) SR si C 1↓sj
rec5:r←j+k
```

Figure 10: Optimized APL recursive string shuffle

```
SR2 =. 3 : 0
: NB. x. is string to match
 si =. >0{ y. NB. s1;s1 substring
 sj =. >1{ y.
 if. 1 >: $x. do.     x. = si,sj
  else.
   i=. 0 NB. Recurse on si
   if. 0 ~: $si do.
     if. (0{x.) = 0{si do.
      i =. (}. x.) SR (}.si);sj
     end.
   end.
   j=. 0 NB. Recurse on sj
   if. 0 ~: $sj do.
     if. (0{x.) = 0{sj do.
      j =. (}. x.) SR si;}.sj
     end.
   end.
  i+j
 end.
)
```

Figure 11: Optimized J recursive string shuffle

6.3 Optimizing SDyn

Dynamic programming solutions usually perform simple computations on array elements, iterating over potentially large arrays without exploiting APL's array processing capabilities. Hence, dynamic programming algorithms in APL place us at the high end of the overhead curve in Figure 9, where computation time is dominated by the number of primitives executed. To improve performance here, the best method is to move computations out of loops when possible, and to simplify them otherwise. We will perform these types of optimizations on `SDyn`:

- code hoisting
- strength reduction
- common subexpression elimination (CSE)
- other improvements

6.3.1 Code Hoisting

The inner loop of `SDyn`, starting at label `lpi` as shown in Figure 7 contains considerable computation on scalars, mostly involving the current character in the argument string `s`. The computation and use of `cc` is static in the sense that it depends only on the values of the string and substrings, and does not require examining any elements of the dynamic programming matrix `m` as it evolves. Thus, removing that code from the inner loop, and precomputing an array corresponding to the values of `cc`, then indexing elements from that array as execution proceeds, has the potential for replacing ten computations with one on each iteration. This precomputation technique is known to compiler writers as *code hoisting*, where it is used to remove loop-invariant code from loops by moving it before the loop.

With regard to performance of this optimization, we recognize that the entire dynamic programming inner loop is scalar computations. Hence, we expect that their removal should have a significant impact on loop execution time. It does, removing nearly half of the primitives in that loop. Figure 14 shows the function after application of this and the two other optimizations discussed below.

It is important to note that the precomputation of `m` is linear in the product of the sizes of the two substrings, unlike the brute force method, which is exponential in the size of the argument string. Thus, the time required for this precomputation is low.

6.3.2 Strength Reduction

Strength reduction is another frequently encountered tool in the compiler writer's toolkit. Strength reduction traditionally replaces one operation by another that is simpler or faster to compute. A traditional example is replacement of `x×2` by `x+x`, although RISC technology has reduced the impact of that particular optimization. In APL, some primitives are faster and/or simpler than others. In the example used here, indexing into the dynamic programming matrix (`m[i;j]`) is replaced by the less complex indexing into a vector (`m[i]`). The performance improvement gained by doing this particular optimization is probably considerably less substantial than that gained by code hoisting. Nonetheless, it can pay off when, for example, a complex indexing expression can be replaced by a simple Boolean computation.

6.3.3 Common Subexpression Elimination

Common subexpression elimination (CSE) is a compiler optimization technique that replaces two or more occurrences of the same expression in a program with one. For example, the code fragment `a[d[i]]+b[d[i]]` would be turned into `temp1←d[i] ◇ a[temp1]+b[temp1]`. The latter fragment runs faster because it does not have to re-evaluate the inner index expression. CSE is a standard feature of every compiler written today, even though it has a fairly small direct impact on user-written code, because most programmers avoid writing such expressions. CSE has a substantial impact on compiled code performance because compilers generate common subexpressions as part of an intermediate step of the compilation process, then use CSE to remove them. For example, the expression `x[i]+j[i]` would generate two sets of code to turn the array index `i` into an address offset into the arrays in storage. CSE would then remove one of these, producing a temporary value that would then be used to reference both arrays.

In `SDyn2`, the two references to `b[i]` are replaced by an assignment to a temporary noun `j←b[i]`, followed by two references to the temporary `j`. As with strength reduction, the performance improvement gained is probably small compared to that of code hoisting. Without code hoisting, the improvement of CSE by itself is likely to be negligible. However, as more and more code is removed from the inner loop, the observed benefit of such techniques increases.

6.3.4 Other Improvements

Another improvement arises from simple code engineering practice. The dynamic programming matrix `m` is created initially as a Boolean array. The dynamic programming iteration through this matrix has the sole purpose of setting more of its elements to zero. Elements which are already zero need not be examined during the iterative stage, so a reduction in work can be achieved by building a worklist `b`, the index vector of non-zero elements in `m`. Since `m` will tend to be largely zero for typical data, this improvement reduces the amount of work that has to be performed.

7 Optimized Performance

After optimization, the APL performance of the old and new recursive (`SR`, `SR2`) and dynamic programming (`SDyn`, `SDyn2`) algorithms have improved significantly, although the relative performance levels are roughly the same, as shown in Figure 12 and Figure 8.

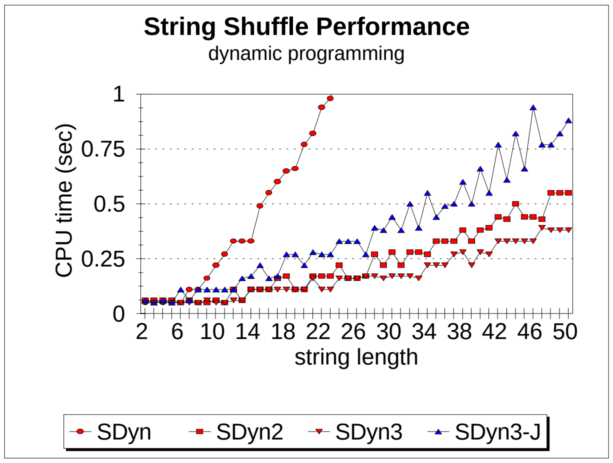

Figure 12: Performance of dynamic programming algorithms

8 Control Structures

We now have a fairly well optimized version of a dynamic programming algorithm for the string shuffle problem. This algorithm is highly iterative, so it is time to turn to the introduction of flow control structures. Flow control structures are popular in most compiled languages because they make code more readable than GOTO-ridden code and because they allow compilers to produce more efficient code. Since two vendors of APL products [5] have recently released support for flow control structures, it is appropriate to examine the performance and readability of structured APL code compared to traditional methods.

The function `SDyn3`, shown in Figure 15, is the algorithm of `SDyn2` trivially revised to use the `:for` flow control structure of APL*PLUS III. The list of values taken on by the induction variable `i` is specified after the `:in` keyword.[6] The loop then operates sequentially, with `i` taking on the next element of the list on each iteration.

The structured code is shorter and easier to read than the GOTO-based code in Figure 14, potentially improving its reliability and maintainability. It also runs faster than its predecessor, as seen in Figure 13: For large arguments, the structured code uses less than 70% of the time required by GOTO-based code. This performance improvement arises from the removal of the induction variable update code. In `SDyn2`, seven primitives are executed to handle the increment and testing of `i` and loop closure. Most of the processing time associated with this maintenance is overhead removed by use of the `:for` loop.

It is also interesting to observe that the performance of APL*PLUS III on `SDyn2` and `SDyn3` is nearly an order of magnitude faster than that of *SDyn3-J* on ISI J Release 2.04. This is exactly the opposite result from that observed with the recursive functions, where J was substantially faster than APL*PLUS III. It is this sort of disparity in performance ratios that makes one eschew benchmarking as a black art. It also lets marketeers on all sides claim, with as much honesty as they can muster, that *Our product is an order of magnitude faster than theirs!*

[5]Iverson Software Inc. and Manugistics, Inc.

[6]An *induction variable* is one whose value in a loop changes in a well-defined and predictable manner. In Figure 14, `i` is an induction variable.

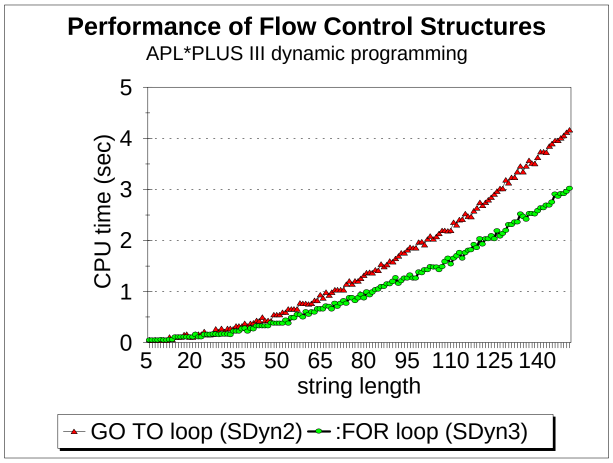

Figure 13: Performance of APL control structures vs GOTO

9 Summary

This paper has taken a simple puzzle and examined it from several algorithmic viewpoints. The lessons we may learn from this analysis are:

- Recursion usually does not pay off if the amount of computation performed at each level is small. Naive recursion is worse than brute force, in most cases.
- Brute force algorithms are often attractive for very small arguments, but their computational complexity often makes them impractical as argument sizes increase.
- Iteration in APL is not necessarily An Evil Thing To Do.
- Dynamic programming can win big over other approaches. It has wide application in areas such as computations on graphs, approximate string matching, binary search trees, text formatting, etc.
- Structured programming facilities improve program efficiency and maintainability in APL, just as they do in other languages.
- APL is faster than J.
- J is faster than APL.
- J and APL run at the same speed.

Additional material about dynamic programming is available [SF92, AHU74, HS78]. A good pragmatic introduction to dynamic programming appears in Baase [Baa88].

10 Acknowledgments

I am grateful to Diane Whitehouse and Marc Griffiths for their thoughtful and meticulous review and editing of this article. The presentation of material was also improved substantially as a result of suggestions received from the APL95 referees.

References

[AHU74] Alfred V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[Baa88] Sara Baase. *Computer Algorithms: Introduction to Design and Analysis*. Addison-Wesley Publishing Company, 1988.

[BB93] Robert Bernecky and Paul Berry. *SHARP APL Reference Manual*. Iverson Software Inc., 33 Major St., Toronto, Canada, 2nd edition, 1993.

[Bel57] Richard Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[Ber89] Robert Bernecky. Profiling, performance, and perfection. In *ACM SIGAPL APL89 Session Tutorials*. ACM SIGAPL, August 1989. ISBN 0-89791-331-0.

[Ber93] Robert Bernecky. Array morphology. *ACM SIGAPL Quote Quad*, 24(1):6–16, August 1993.

[Ber95] Robert Bernecky. The role of dynamic programming and control structures in performance. *ACM SIGAPL Quote Quad*, 26(1):11–19, June 1995.

[CLR90] Thomas H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[HS78] Ellis Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, 1978.

[Kim95] Scott Kimbrough. APL, dynamic programming, and the optimal control of electromagnetic brake retarders. *ACM SIGAPL Quote Quad*, 26(1):98–108, June 1995.

[LB90] Edward Y.H. Lin and Dennis L. Bricker. Implementing the recursive APL code for dynamic programming. *ACM SIGAPL Quote Quad*, 20(4):239–250, August 1990.

[SF92] Moshe Sneidovich and Suzanne Findlay. Jogging with APL along the shortest path. *ACM SIGAPL Quote Quad*, 23(1):221–227, July 1992.

```
 r←s SDyn2 y;m;i;si;sj;b;d;j
 ⍝ DP string shuffler (opt)
 si←D y[0] ⋄ sj←D y[1]
 m←(' ',s)[(⍳1+⍴si)∘.+⍳1+⍴sj]
 b←m←(m=(⍴m)⍴' ',sj)∨m=⍉(⌽⍴m)⍴' ',si
 b[0;]←b[;0]←0 ⋄ b←,b ⋄ b←b/⍳⍴b
 m[;0]←∧\m[;0] ⋄ m[0;]←∧\m[0;]
 m←,m
 i←0
 d←1+0,⍴sj ⍝ Dist to next row, col
lp:→(i=⍴b)⍴lpz ⍝ FOR loop
 j←b[i]
 m[j]←∨/m[j-d]
 i←i+1 ⋄ →lp
lpz:r←¯1↑m
```

Figure 14: Optimized APL dynamic programming string shuffle

```
 r←s SDyn3 y;i;m;si;sj;b;d;j
 ⍝ DP string shuffler (opt)
 si←D y[0] ⋄ sj←D y[1]
 m←(' ',s)[(⍳1+⍴si)∘.+⍳1+⍴sj]
 m←(m=(⍴m)⍴' ',sj)∨m=⍉(⌽⍴m)⍴' ',si
 b←m
 b[0;]←b[;0]←0 ⋄ b←,b ⋄ b←b/⍳⍴b
 m[;0]←∧\m[;0] ⋄ m[0;]←∧\m[0;]
 m←,m
 d←1+0,⍴sj ⍝ Dist to next row, col
 :for i :in ⍳⍴b ⍝ FOR loop
   j←b[i]
   m[j]←∨/m[j-d]
   :endfor
 r←¯1↑m
```

Figure 15: DP APL string shuffle with control structures

```
SDyn3 =. 3 : 0
: NB. x. is string to match
 si =. >0{ y. NB. y. is substrings
 sj =. >1{ y.
NB. Build DP matrix
 m =. si +/ & (i. & #) sj
 m =. m{ 1 }. x.
 b =. sj="1 m NB. Build worklist
 b =. m=. b +.|: si="1 |: m
 b =. ,0,0,"1 b
 b =. b#i.#b
NB. Set edge of DP matrix
 m =. (*./\sj=(#sj)$x.),m
 m =. ,(,. *./\1,si=(#si)$x.),"1 m
NB. Distance to next row, column
 d =. 1+0,#sj
 i =. 0 NB. DP loop
 while. i <#b do.
  j=. i{b
  m =. (+./(j-d){m) j}m
  i=. >:i
 end.
 {: m
)
```

Figure 16: DP J string shuffle with control structures