# ON SOME ADVANTAGES OF ASYMMETRY IN ALGORITHM DESIGN

**Douglas Maurer**

Department of Electrical Engineering and Computer Science,
The George Washington University Washington, DC 20052

Certain situations in algorithm design immediately suggest a >symmetrical treat ment. Upon closer examination, however, an asymmetrical treatment can often improve the performance of an algorithm, in ways not immediately apparent. Consider, for example, the pointers to the first and last elements of a queue, implemented as a circular array. It is not too difficult to see that these could be implemented as pointers to the positions one before the first element, and one after the last element.Better performance, however, arises from an asymmetrical treatment.One pointer points to the first element; the other one points to the position one beyond the last element. Or consider the pointers to the first and last elements of a subarray during a binary search. This subarray is repeatedly divided in two during the search, and one of the two halves is discarded. Again we need pointers to the first and last elements of the subarray, and again an asymmetrical treatment gives better performance; one pointer points to the end of the subarray, while the other pointer points one before the start of it. Still further examples arise from the treatment of parameters, in modern programming languages, in such a way that they are processed faster when it is known that they will not change. This does not affect output of data X, but it does affect input of X, since this process changes X. The result is that the symmetrical treatment of READ(X) and WRITE(X) in older languages has given way to an asymmetrical treatment of c= getc () and putc(c) in newer languages. Balanced trees also give poorer performance in many situations than do trees which are slightly (although not too much) out of balance. Algorithm design is replete with such examples, as we shall illustrate in the full paper.