Analytic Derivation of Comparisons in Binary Search

Timothy J. Rolfe, Ph.D.
Dakota State University
Madison SD 57042-1799
rolfe@alpha.dsu.edu
http://www.dsu.edu/~rolfe/

Numerous authors of programming texts and algorithms / data structures texts develop code for the binary search. The common implementation is one in which the loop contains two comparisons for three outcomes — the key is to the left of the midpoint, to the right of the midpoint, or exactly at the midpoint — Implementation A below.

To exit immediately upon finding the item seems to be the obvious choice for fastest execution. But that is not the case.

One can also implement the algorithm with only a single comparison — and no early exit — as in Implementation B below.

Implementation A

```
template <class TP>
int BinSrch (int N, TP X[], TP Key)
{
   int Lo = 0, Hi = N-1, Mid;
   while (Lo <= Hi)
   {
      Mid = (Lo + Hi) / 2;
      if (Key > X[Mid)
         Lo = Mid + 1;
      else if (X[Mid] > Key)
         Hi = Mid - 1;
      else
         break;
}
   return Mid;
// I.e., where the item would BELONG
}
```

Implementation B

```
template <class TP>
int BinSrch (int N, TP X[], TP Key)
{
   int Lo = 0, Hi = N-1, Mid;
   while (Lo < Hi)
   {
      Mid = (Lo + Hi) / 2;
      if (Key > X[Mid)
         Lo = Mid + 1;
      else
         Hi = Mid
   }
   return Lo;
// I.e., where the item is or would belong
}
```

Without the early exit, Implementation B will always go the full $\lg(n)$ iterations (where lg denotes $\log_2$), but it will only perform one comparison on each iteration. Implementation A, with the early exit, will, on average, perform 1.5 comparisons on each iteration, but will sometimes avoid unnecessary iterations.

Implementation B is obviously preferable on failure to find: since failure has no early exit, Implementation A on average requires half again as many comparisons as Implementation B.

The significant question is this: on average which implementation is preferable to *find* an item.

To ease the analysis, let us consider dealing with n items such that $(n+1) = 2^k$ (that is, we have a **full** binary tree for the comparison tree). Implementation B is known to iterate k times. So we need to determine the

average number of iterations for Implementation A to compare the two.

For our purposes here, it will be convenient to number the root of the binary search tree as level 1. Then the number of iterations necessary to find an element at a level *j* will simply be the level of the tree.

To *average* the number of comparisons, we simply need to total up the comparisons required to reach each element in the binary search tree. At a given level *j*, there are 2^{j-1} tree nodes (due to our numbering the root as level 1). In terms of the level, we can write the number of iterations required to find all of the items at that level as $\frac{1}{2} j 2^j$ — writing the 2^{-1} as a constant and simplifying the *j* dependence. The average will then be the total of these divided by the number of nodes in the tree.

$$Average = \left(\frac{1}{2} \sum_{j=1}^{k} j2^j \Big/ n \right) \tag{1}$$

Brian Carlson of Southern Illinois University at Edwardsville has discovered a closed-form solution for the summation:

$$\frac{1}{2} \sum_{j=1}^{k} j2^j = (k-1)2^k + 1 \quad * \tag{2}$$

Given our definition that $n + 1 = 2^k$, we can write *n* as $(2^k - 1)$. Combining equations (1) and (2):

$$Average = \left((k-1)2^k + 1 \right) \Big/ \left(2^k - 1 \right) \tag{3}$$

As k grows, this simplifies to

$$Average \approx (k-1) \tag{4}$$

In other words, at the expense of doing on average half again as much work in each iteration, we have only eliminated *one* of the iterations!

Between sizes $(2^k - 1)$ and $(2^{k+1} - 1)$, of course, one adds leaf nodes in the decision tree requiring $k+1$ comparisons until all leaf nodes are at that level. With the increasing power of personal computers, one can fairly easily obtain exact averages by explicit computations, accumulating statistics on iterations required and comparisons required to find all of the nodes in an array of particular size and then calculating

* $S(k) = (k-1) \times 2^k + 1$ *(Inductive hypothesis)*

$S(1) = (1/2)(1 \times 2^1) = (1-1) \times 2^1 + 1 = 1$ *(Base case confirmed)*

$S(k+1) = S(k) + (1/2)(k+1) \times 2^{k+1}$ *(Definition of recurrence)*

$= (k-1) \times 2^k + 1 + (k+1) \times 2^k$ *(Apply inductive hypo.)*

$= (2k) \times 2^k + 1$ *(Algebraic rearrangement)*

$= (k) \times 2^{k+1} + 1$ *QED*

the exact average. The following figure shows the average number of iterations required and of comparisons required to find array elements as the size of the array increases. Implementation A generates two lines in the chart, while Implementation B (for which there is only one comparison per iteration) generates a single line — the one lying between the two Implementation A lines. Note that all figures use a logarithmic scale for the horizontal axis. Data points were calculated to space evenly along a logarithmic X-axis (i.e., array sizes increase by a multiplicative factor of the eighth root of 2 to generate eight data points for each doubling of the size).

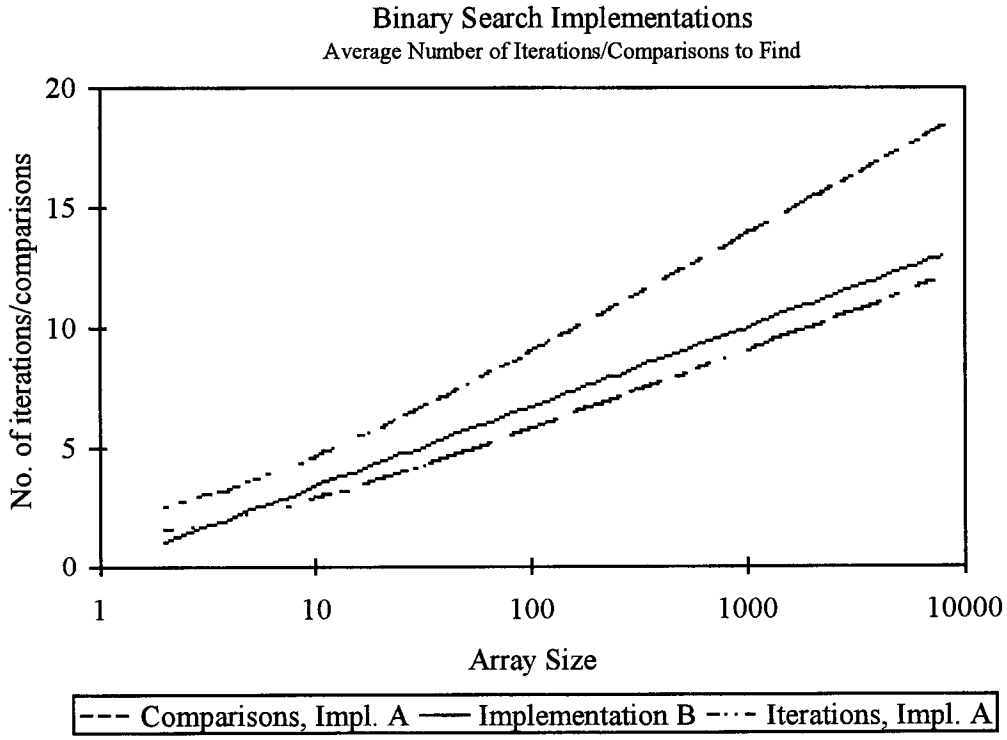

Figure 1

One interesting fact arises from these statistics: the ratio of comparisons to iterations for Implementation A is slightly *greater* than 1.5: while it takes on average 1.5 comparisons to progress beyond an interior node in the search, it always takes two comparisons to find an item.

The figure shows that Implementation A on average requires more comparisons, but economizes somewhat on the number of iterations, something we already know from the analysis above. It will depend on the details of particular hardware implementations as to whether Implementation A is always worse than Implementation B, or whether there is a range of sizes that favor Implementation A.

The following two figures show explicit timing results of searching a 16-bit integer array in two different environments: Figure 2 shows results on a 20 MHz 80386 processor under the FreeBSD operating system, with the code compiled by g++. Figure 3 shows results on a Pentium computer, specifically a P5-90 running in MS-DOS mode under Windows-95 (i.e., in single-process mode to avoid the Windows-95 overhead), the code having been compiled under version 3.1 of Borland C++. In both figures, Implementation A is the solid line (concave upward) and Implementation B is the dotted line (approximately linear).

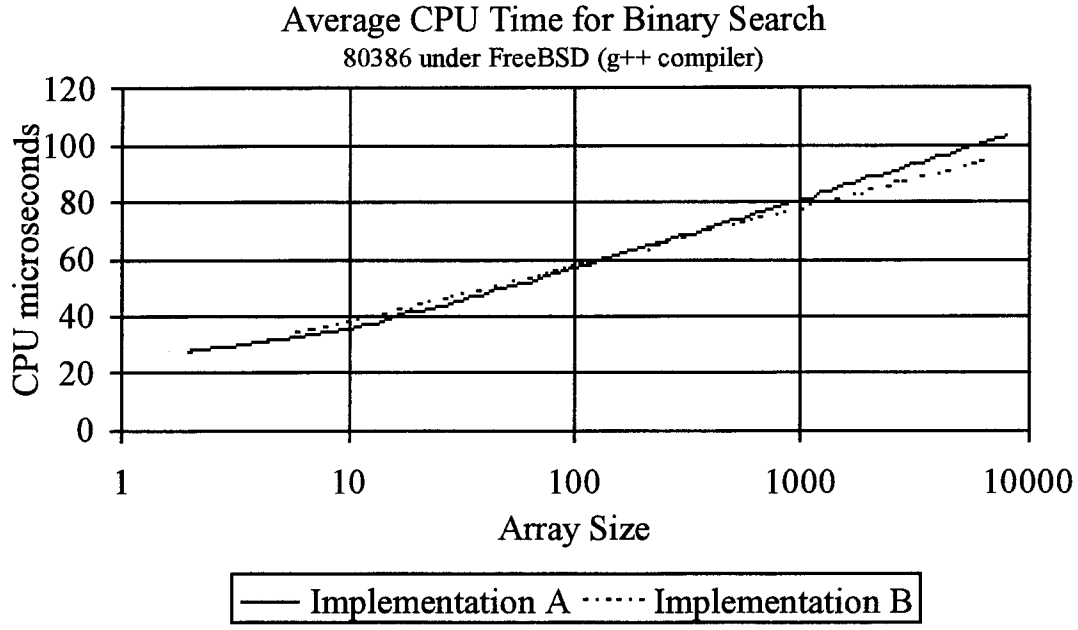

Figure 2

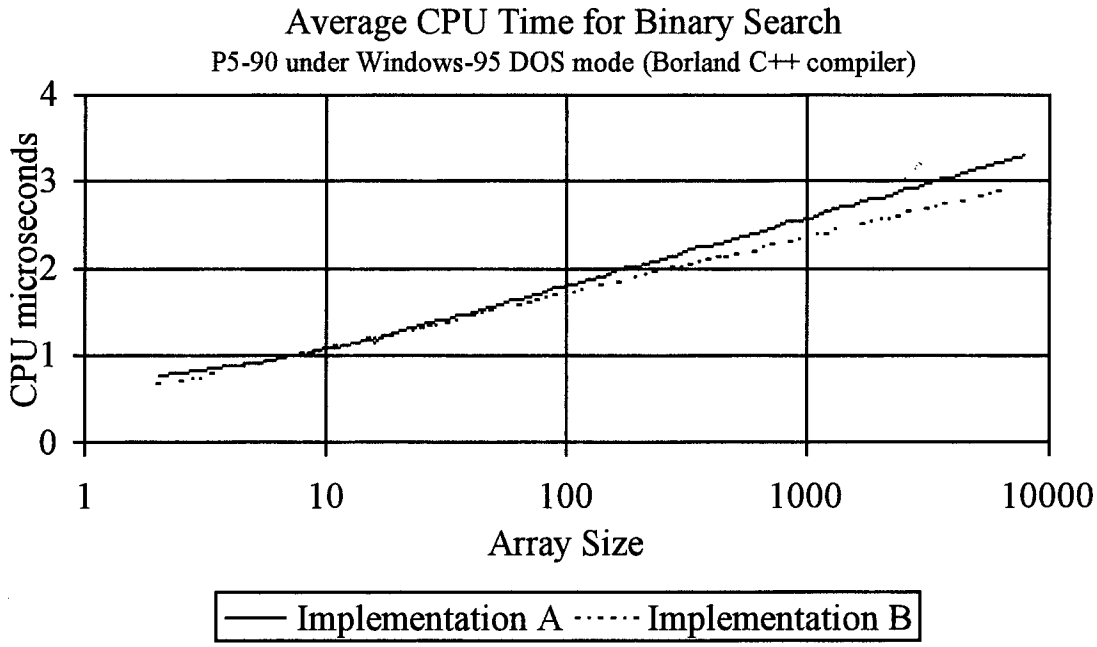

Figure 3

Implementation B is clearly preferable: while there *is* curve crossing in the 80386 runs, for large values of n it proves to be the faster of the two. Implementation A pays in comparisons for the small benefit provided by the early exit.

Acknowledgements

The computations reported were performed on equipment owned by the state of South Dakota and located at Dakota State University. (A similar set was performed on a Digital Equipment Corporation alpha computer, alpha.dsu.edu. That computer, however, is the Dakota State University web server and the variability of the time-sharing environment perturbed the timings attempted.)

I wish to thank Dr. Brian Carlson of South Illinois University at Edwardsville (formerly of Dakota State University) for the analytic formula reported as Equation (2) above, and more generally for helpful discussions and critiques on this problem during his time at Dakota State University.