# Using O(*n*) ProxmapSort and O(1) ProxmapSearch to Motivate CS2 Students, Part II

Thomas A. Standish  Norman Jacobson
Donald Bren School of Information and Computer Sciences
University of California, Irvine
Irvine, California 92697-3425
{standish, jacobson}@ics.uci.edu

### Abstract

Presenting "cool" algorithms to CS2 students helps convince them that the study of data structures and algorithms is worthwhile. An algorithm is perceived as cool if it is easy to understand, *very* fast on large data sets, uses memory judiciously and has a straightforward, short proof — or at least a convincing proof sketch — using accessible mathematics. To illustrate, we discuss two related and relatively unknown algorithms: ProxmapSort, previously discussed in Part I of this paper, and ProxmapSearch, discussed here.

### Keywords

CS2, ProxmapSearch, ProxmapSort, searching, sorting

### Introduction

In Part I of this paper, we presented the ProxmapSort sorting algorithm (also described in [1], [2], and [3]) and we showed that, if keys are "well distributed," this algorithm sorts in time O(*n*) — faster than key-comparison sorting techniques, which can do no better than O(*n* log *n*).

In our CS2 classes, we have also been discussing the ProxmapSearch searching algorithm, which was discovered when preparing the instructor's manual for [1]. It can be presented quickly once ProxmapSort has been covered. Students already know that binary search in ordered arrays is considered fast at O(log *n*) time and that searching based on open addressing hashing algorithms is O(1) if the array is relatively empty but tends to O(*n*) as the array becomes saturated. So they are astonished to learn that ProxmapSearch finds a key in an average of 1.5 key comparisons, using information generated during a ProxmapSort of the original array, and that the result holds even when the array is full.

### ProxmapSort Prepares for ProxmapSearch

In Part I of this paper we gave an example to introduce students to the main ideas in ProxmapSort. We include part of that example here (Fig. 1) to illustrate how ProxmapSearch uses the *proxmap* generated by ProxmapSort.

**Example.** Consider a full array A[0..*n* –1] of *n* keys, with the keys drawn randomly and uniformly from the possible key values *K* in the range ($0.0 \le K < 13.0$), and let *i* in [0..*n* –1] be an index of that array. Assume that we have already applied ProxmapSort to sort A's keys, using the hit count array H, the proxmap array P, and the insertion location array L as intermediaries. It is the proxmap array P[0..*n*–1] that must be retained after ProxmapSort is completed in order for ProxmapSearch to work properly.

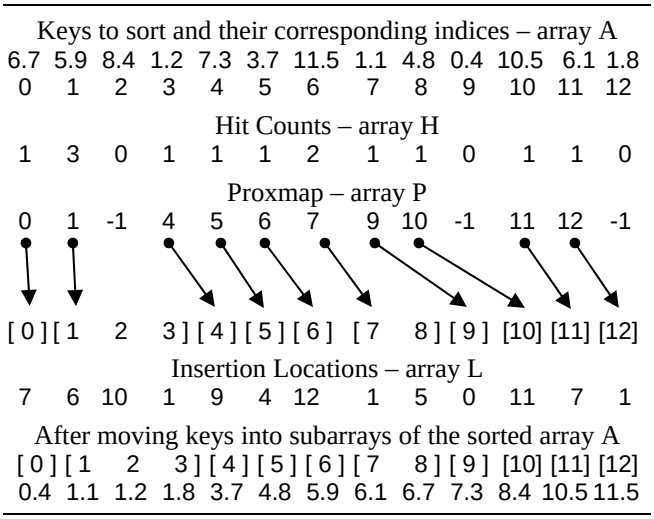


**Figure 1. Part of ProxmapSort example from Part I containing data used by ProxmapSearch**

**Choosing a MapKey function.** Recall (from Part I) that we chose a *map key function* MapKey(*K*) = *i* such that (1) *i* is an array index ($0 \le i < n$), (2) $K_1 < K_2$ whenever MapKey($K_1$) < MapKey($K_2$), (3) for all *i*, the number of keys that map to *i* is nearly identical, and (4) MapKey is fast to compute. In Fig. 1, we used MapKey(*K*) = floor(*K*), where ($0.0 \le K < 13.0$), and showed students how that choice met the above criteria.

In general, the domain of the MapKey function is the space of all possible keys **K** and the range is the set of indices $\{ i \mid 0 \le i < n \}$ of the array A[0..*n* – 1]. Thus, MapKey: $\mathbf{K} \rightarrow$ [0..*n* – 1]. In practice, it is convenient to separate the preparation of the MapKey function into two stages. The first stage involves choosing a function, UnitIntervalMap: $\mathbf{K} \rightarrow [0, 1)$, that maps keys $K \in \mathbf{K}$ uniformly and evenly into floating point numbers in the half-open unit interval. Thus, for all $K \in \mathbf{K}$, UnitIntervalMap(*K*) = *r*, where ($0.0 \le r < 1.0$). Then, in the second stage, given an array A[0..*n* – 1] containing *n* keys (and using Java notation), we set

MapKey(*K*) = (int) Math.floor(*n**UnitIntervalMap(*K*)).

Thus, a suitable UnitIntervalMap can be chosen in advance of knowing the size *n* of the array A to be sorted, and, once

*n* is known, its unit interval range can be scaled by *n* to yield a suitable MapKey function.

**Critical proxmap properties.** Recall from Part I that each hit count H[*i*] gives the size of the reserved subarray S that contains all keys *K* that map to location *i* and that the proxmap values stored in P[*i*] were computed during ProxmapSort according to the formula

$$P[i] = -1 \text{ if } H[i] = 0, \text{ otherwise } P[i] = \sum_{(0 \le j < i)} H[j].$$

This implies: (*i*) that each nonempty reserved subarray S starts at a location $p$ = proxmap[MapKey(*K*)] that is the sum of the sizes of the reserved subarrays to its left (all of which contain keys smaller than those in S by MapKey property (2) above), and (*ii*) that the proxmap value P[*i*] is –1 for any empty subarray S (i.e., one for which S's size, H[*i*], was 0). These two facts are crucial to understanding how ProxmapSearch works.

## ProxmapSearch

**Overview.** Consider the array A just sorted with ProxmapSort. For any search key *K*, we know that MapKey(*K*) = *i* is an index of A[0..*n* – 1], so $0 \le i < n$. We now assume that the array A is extended by one item A[*n*] and that we store *K* in A[*n*] before starting to search for *K*.

To search for key *K*, $p$ = proxmap[MapKey(*K*)] either has the value $p = -1$, in which case *K* is not in A because its corresponding reserved subarray is empty, or else, if $p \ge 0$, then $p$ gives the start of the subarray in which the key *K* must reside, if *K* is in A. So we search upwards in A, moving past all locations A[$p$], A[$p$+1], … containing keys smaller than *K*, and we stop at the first location A[$q$] such that A[$q$] ≥ *K*. If A[$q$] > *K* or if $q = n$, *K* was not in A, so we return –1. Otherwise, by elimination, *K* = A[$q$], so we return the position $q$ where we found it.

Running through a few searches, using the ProxmapSort example data in Fig. 1 and using search keys *K* chosen from the interval ($0.0 \le K < 13.0$), quickly convinces students that ProxmapSearch works. ProxmapSearch's code is given in Fig. 2.

```
int proxmapSearch(KeyType K, KeyType[ ] A, int numberOfKeys)
{
  // get first location to search using the proxmap
  int currentPosition = proxmap[MapKey(K)];

  if (currentPosition == -1)    // subarray empty
    return -1;                  //  key not in A

  A[numberOfKeys] = K; // save K in extension of A[0..n-1] at A[n]

  // the subarray is nonempty; begin search at its proxmap location.
  // see footnote 1 below for another version of this search loop.
  int comparisonResult;
  while (true) {
    comparisonResult = A[currentPosition].compareTo(K);
    if (comparisonResult >= 0) // exit search loop
        break;                 // if A[currentPosition] >= K
    currentPosition++;    // keep looking if A[currentPosition] < K
  }
  if ((comparisonResult > 0) || (currentPosition == numberOfKeys))
      return -1;           // key K not contained in A
    else
      return currentPosition;    // K found in A; return its position
}

/*  footnote 1: Many experts consider the use of a "break" statement
 *  to exit from the middle of a "while loop" to be poor programming
 *  practice. We employed a break because we were optimizing
 *  ProxmapSearch's speed. The break can be avoided at the
 *  expense of evaluating the expression "(comparisonResult < 0)"
 *  twice. One approach is:
 *
 *  int comparisonResult;
 *  do {
 *    comparisonResult = A[currentPosition].compareTo(K);
 *    if (comparisonResult < 0)
 *      currentPosition++;    // keep looking if A[currentPosition] < K
 *  } while (comparisonResult < 0); // leave if A[currentPosition] >= K
 */
```

**Figure 2. The ProxmapSearch Algorithm**

**Distribution of reserved subarray sizes**. Our claim in Part I that the proxmap sends each key *K* to an insertion location that is usually in close proximity to its final position in sorted order, and the reason why ProxmapSearch starts searching for a key *K* at a location that is usually close to the place where *K* can be found in A, are based on the fact that most reserved subarrays are small. We can understand just how small they are on average by studying the distribution of their sizes. Our assumption of randomly and uniformly drawn keys produces subarrays whose sizes form a binomial distribution. The Poisson approximation to the binomial distribution (see [4], p. 143) closely estimates the fraction of the reserved subarrays of size *k* as $1/(k!\, e)$. Table 1 shows the percentages of reserved subarrays of various sizes according to this approximation.

| *Subarrays of Size k as a Percentage of all Subarrays* | | |
|---|---|---|
| *subarray size k* | *fraction of total* | *percentage of total* |
| 0 | 0.36788 | 36.788% |
| 1 | 0.36788 | 36.788% |
| 2 | 0.18394 | 18.394% |
| 3 | 0.06131 | 6.131% |
| 4 | 0.01533 | 1.533% |
| 5 | 0.00307 | 0.307% |
| 6 | 0.00051 | 0.051% |
| 7 | 0.00007 | 0.007% |
| ≥ 8 | 0.00001 | 0.001% |
| total = | 1.00000 | 100.000% |

**Table 1. Percent of Subarrays of Given Size *k***

Thus, Table I implies that, in a ProxmapSorted array, fewer than 0.4% of the subarrays will contain more than four keys.

**Analysis of Running Time**. As discussed above, if the keys in A were uniformly and randomly chosen, and MapKey(*K*) maps all possible search keys *K* uniformly and evenly onto the array indices of A, most of the subarrays will be small. ProxmapSearch will check at most the keys

in one subarray S and the first key past the end of S, so it ought to be fast. In the case of successful search, the proof of ProxmapSearch's performance is easy for CS2 students to follow, but the proof for unsuccessful search uses, in a simple way, probabilities resulting from Bernoulli trials, an approach that is not always familiar to CS2 students. Still, both proofs can be sketched quickly and convincingly.

**Successful ProxmapSearch.** Successful search for a key *K* in an array of length *n* is breathtakingly fast, taking on average C = 1.5 – 1/(2*n*) key comparisons.

*Proof*: The start of the search is at location *p* = proxmap[MapKey(*K*)], where *p* gives the start of a subarray S containing *j* keys that contains *K*.

Thus, after uniform and random insertion of *i* keys into A, the average size of *j* is 1 + (*i* – 1)/*n*, and after inserting all *n* keys into A the average size of *j* is *j* = 1 + (*n* – 1)/*n*. When we search for *K* in S, it could be in any of these *j* possible positions with equal probability, so the average number of key comparisons needed to find it successfully, C, is just

$$(1 + 2 + ... + j)/j = j * (j + 1)/2 * (1/j) = (j + 1)/2.$$

Substituting $j = 1 + (n – 1)/n$ in this expression gives

$$C = (1 + (n – 1)/ n + 1)/2 = 1.5 – 1/(2n).$$

**Unsuccessful ProxmapSearch.** The average number of key comparisons C´ for an unsuccessful search is $C' = 1.5 – (1 – 1/n)^n$, and for large *n*, C´ ≅ 1.5 – 1/*e* — even faster than successful search!

*Proof*: When searching for a key *K* that is not in A, *p* = proxmap[MapKey(*K*)] could lead to an empty subarray (indicated by *p* = –1). If so, no key comparisons are required to determine that *K* is not in A. The proxmap could instead lead to a non-empty subarray S. If so, we must search in S, and possibly one key position past the end of S, to determine that *K* is not in A. We need to know the expected size *j* of S to determine the average number of key comparisons needed to know that *K* is not in A.

The probability that a subarray of A will be empty is the probability that none of the *n* keys in A maps to a given location in A under MapKey(*K*). This is given by having *k* = 0 successes in *n* Bernoulli trials *b*(*k*, *n*, *p*) with probability *p* = 1/*n* for success and *q* = (*n* – 1)/*n* for failure (see [4], p. 137). Thus,

$$b(k, n, p) = \binom{n}{k} p^k q^{n-k} = \binom{n}{k} (1/n)^k ((n-1)/n)^{n-k}.$$

By setting *k* = 0 (for 0 successes) and recalling that 0! = 1, this simplifies to $\left(1-\frac{1}{n}\right)^n$, a quantity that eventually approaches the limit 1/*e* = 0.36788 as *n* gets larger (cf. [4], p. 142). Recall that roughly 36.8% of the subarrays in a proxmap-sorted array are empty. Now, let $f = 1 – (1 – 1/n)^n$ be the fraction of subarrays in A that are non-empty. If all *n* keys in A are stored in the *n***f* non-empty subarrays of A, then the average size *j* of a non-empty subarray is *j* = 1/*f*.

In general, we compute proxmap[MapKey(*K*)] = *p*, and if *p* ≥ 0, we start comparing *K* to the keys A[*p*], A[*p*+1], ... , A[*p*+*j*]. As soon as we find the first key in A that is greater than *K* or we find *K* in A[*n*], we can conclude that *K* is not in A[0..*n* – 1]. Because there is an equal chance of finding that *K* is not in the subarray after looking at any key in it or at the key right after its last key, the average number of key comparisons needed to find that *K* is not in the subarray is (1 + 2 + ... + (*j*+1)) /(*j*+1) = (*j*+2)/2 = 1+*j*/2. But since this search applies only to the fraction $f = 1 – (1 – 1/n)^n$ of subarrays in A that are non-empty, the average number of key comparisons needed to determine that *K* is not in A is

$$f * (1+j/2) = f * (1 + (1/f)/2) = f + 1/2$$
$$= 1 – (1 – 1/n)^n + 0.5 = 1.5 – (1 – 1/n)^n.$$

Because $(1 – 1/n)^n$ tends to 1/*e* as *n* increases, for large *n* we can say that C´ is about 1.5 – 1/*e*.

**Comparing Actual and Predicted Results.** As with ProxmapSort, we show students data to demonstrate how well the algorithm performs in practice and how well theory agrees with observed results.

Table 2 shows predicted results for successful and unsuccessful ProxmapSearch for various array sizes. The last row shows the limits that are approached for infinitely large *n*. Even for small *n*, the results are reasonably close to the theoretical limits.

*ProxmapSearch*'s
Predicted Average Number of Keys Inspected
in Successful and Unsuccessful Searches

| *array size n* | *av. keys in successful search* | *av. keys in unsuccessful search* | *av. zero length subarray hits* | *sum of last two columns* |
|---|---|---|---|---|
| 64 | 1.49219 | 1.13501 | 0.36499 | 1.50000 |
| 128 | 1.49609 | 1.13356 | 0.36644 | 1.50000 |
| 256 | 1.49805 | 1.13284 | 0.36716 | 1.50000 |
| 512 | 1.49902 | 1.13248 | 0.36752 | 1.50000 |
| 1024 | 1.49951 | 1.13230 | 0.36770 | 1.50000 |
| ∞ | 1.50000 | 1.13212 | 0.36788 | 1.50000 |

**Table 2. Predicted Data for ProxmapSearch**

Table 3 shows the observed average number of key comparisons used in successful and unsuccessful proxmap searches for arrays of various sizes, using single-precision floating point numbers as keys. It's apparent how well theoretical and observed results agree.

*ProxmapSearch*
Average Number of Keys Inspected in
Successful and Unsuccessful Searches: 10000 Trials

| *array size* | *av. keys in successful search* | *av. keys in unsuccessful search* | *av. zero length subarray hits* | *sum of last two columns* |
|---|---|---|---|---|
| 64 | 1.49177 | 1.13538 | 0.36491 | 1.50029 |
| 128 | 1.49640 | 1.13506 | 0.36633 | 1.50139 |
| 256 | 1.49868 | 1.13075 | 0.36741 | 1.49816 |
| 512 | 1.49870 | 1.13211 | 0.36734 | 1.49945 |
| 1024 | 1.49919 | 1.13209 | 0.36814 | 1.50023 |

**Table 3. Experimental Data for ProxmapSearch**

## Learning about algorithms that scale up

The ProxmapSearch algorithm "scales up"— it continues to work well as the search array gets really big. Students readily understand this concept, as we just showed them that ProxmapSearch takes 1.5 comparisons on average to find keys, regardless of the array's size.

An impressive illustration is a "reverse phone book" of 1,000,000 phone numbers. First, choose a good hash function $h(n)$ (see [5]) that spreads out clusters of phone numbers $n$ with the same area codes and prefixes so that the $h(n)$ are distributed uniformly — which is needed for ProxmapSort and ProxmapSearch to work well. Second, proxmap-sort the hash codes $h(n)$ of all phone numbers in the reverse phone book. To find the owner $N$ of the phone number $n$, we proxmap-search for the key $h(n)$ to find the record $(h(n), n, N)$ containing $N$.

## Conclusions

Our experience presenting many algorithms to CS2 students has shown us that students quickly develop a real appreciation for theoretical computer science when they see how its practice produces algorithms such as ProxmapSort and ProxmapSearch. Cool algorithms really do show that theory is cool.

## References


[1] Standish, T. A., *Data Structures, Algorithms, and Software Principles*, Addison-Wesley, Reading, MA, 1994.

[2] Standish, T.A., *Data Structures, Algorithms, and Software Principles in C*, Addison-Wesley, Reading, MA, 1995.

[3] Standish, T.A., *Data Structures in Java*, Addison-Wesley, Reading, MA, 1998.

[4] Feller, W., *An Introduction to Probability Theory and Its Applications, Vol. I*, Wiley, New York, 1957.

[5] Knott, G.D., Hashing Functions, *Computer Journal*, 18:3, pp. 265-278, Aug. 1975.