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

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, discussed here, and ProxmapSearch, to be discussed in Part II.

Keywords

CS2, ProxmapSearch, ProxmapSort, searching, sorting

Introduction

When teaching CS2 students, it is sometimes challenging to stimulate interest in algorithms and the proofs of their performance. We've noted that the students who "tune out" when we attempt to teach them run-of-the-mill algorithms "tune in" when we teach them "cool" algorithms — those that are easy to grasp, *very* fast, stingy with memory, and have short proofs or convincing proof sketches that students can readily follow.

We have been presenting the ProxmapSort sorting algorithm, described in [1], [2], and [3], to CS2 students for several years. Students are amazed to learn that, if keys are "well distributed," this algorithm sorts in time O(*n*), much faster than the comparison-based sorting techniques that they have just learned can do no better than O(*n* log *n*). Because the proof of ProxmapSort's performance we've known until recently was too advanced for our CS2 classes, we had to resort to a hand wave. We've since discovered a new proof, presented below, that CS2 students can grasp.

We have also begun discussing the ProxmapSearch searching algorithm, which uses the proxmap generated during a ProxmapSort of the original array. Students are astonished to learn that ProxmapSearch finds a key in an average of 1.5 key comparisons. The ProxmapSearch algorithm, its analysis, and an application showing that it "scales up," are presented in Part II of this paper.

ProxmapSort

We introduce students to ProxmapSort by example and then discuss the algorithm more generally. Then we analyze the algorithm's performance.

Example. Consider a full array A[0..*n* –1] of *n* keys, with the keys drawn randomly and uniformly from the possible key values, and let *i* in [0..*n* –1] be an index of A. We want to sort A's keys into array A2. (See Fig. 1.)

Keys to sort and their corresponding indices – array A

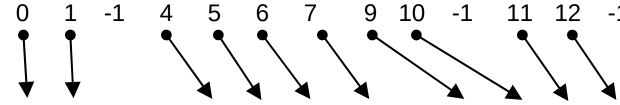
6.7	5.9	8.4	1.2	7.3	3.7	11.5	1.1	4.8	0.4	10.5	6.1	1.8
0	1	2	3	4	5	6	7	8	9	10	11	12

Hit Counts – array H

1	3	0	1	1	1	2	1	1	0	1	1	0

Proxmap – array P

0	1	-1	4	5	6	7	9	10	-1	11	12	-1



[0]	[1	2	3]	[4]	[5]	[6]	[7	8]	[9]	[10]	[11]	[12]

Insertion Locations – array L

7	6	10	1	9	4	12	1	5	0	11	7	1

Moving keys into subarrays of the sorted array A2

[0]	[1	2	3]	[4]	[5]	[6]	[7	8]	[9]	[10]	[11]	[12]
-.-	-.-	-.-	-.-	-.-	-.-	-.-	-.-	-.-	-.-	-.-	-.-	-.-
-.-	-.-	-.-	-.-	-.-	-.-	-.-	6.7	-.-	-.-	-.-	-.-	-.-
-.-	-.-	-.-	-.-	-.-	-.-	5.9	6.7	-.-	-.-	-.-	-.-	-.-
-.-	-.-	-.-	-.-	-.-	-.-	5.9	6.7	-.-	-.-	8.4	-.-	-.-
-.-	1.2	-.-	-.-	-.-	-.-	5.9	6.7	-.-	-.-	8.4	-.-	-.-
-.-	1.2	-.-	-.-	-.-	-.-	5.9	6.7	-.-	7.3	8.4	-.-	-.-
-.-	1.2	-.-	-.-	3.7	-.-	5.9	6.7	-.-	7.3	8.4	-.-	-.-
-.-	1.2	-.-	-.-	3.7	-.-	5.9	6.7	-.-	7.3	8.4	-.-	11.5
-.-	1.1	1.2	-.-	3.7	-.-	5.9	6.7	-.-	7.3	8.4	-.-	11.5
-.-	1.1	1.2	-.-	3.7	4.8	5.9	6.7	-.-	7.3	8.4	-.-	11.5
0.4	1.1	1.2	-.-	3.7	4.8	5.9	6.7	-.-	7.3	8.4	-.-	11.5
0.4	1.1	1.2	-.-	3.7	4.8	5.9	6.7	-.-	7.3	8.4	10.5	11.5
0.4	1.1	1.2	-.-	3.7	4.8	5.9	6.1	6.7	7.3	8.4	10.5	11.5
0.4	1.1	1.2	1.8	3.7	4.8	5.9	6.1	6.7	7.3	8.4	10.5	11.5

Figure 1. ProxmapSort Example

Choose 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. [1], [2] and [3] give strategies for determining a suitable map key function for a variety of situations. For this example, given that the possible key values are in the range ($0.0 \le K < 13.0$), we choose MapKey(K) = floor(K) and show students how that choice meets the above criteria.

For each array index i that is a map key value, we compute a "hit count" of the number of keys that map to i. We use the hit count array H[0..*n* –1] to hold these counts, where H[i] = the number of occurrences of keys K in A such that MapKey(K) = i. To compute H, we initialize H to contain all zeros and then scan sequentially through the keys K in A, incrementing H[MapKey(K)] for each key K.

Next, we convert the hit counts to a *proxmap*. The term *proxmap* is short for *proximity map* because it maps each key onto a location in A2 that is usually in close proximity to its final resting place in sorted order.

Each group of keys mapping to the same i will eventually be placed in the same *reserved subarray* (*subarray* for short). The value of H[i] gives the exact size of this subarray. When all the subarrays are placed next to one another in ascending order in A2, their beginning locations in A2 define a *proxmap* that specifies an approximate mapping of each key to its final place in the sorted array. In Fig. 1 the proxmap values are stored in the array P. Each P[i] points to the starting location of its respective reserved subarray, unless H[i] = 0, in which case P[i] = –1 to denote an empty subarray. From the proxmap definition formula

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

we see that each non-empty subarray starts at a location P[i] that is just the sum of the subarray sizes to its left.

We next compute an array of insertion locations L[0..n – 1]. L[i] stores the location of the beginning of the subarray in A2 where key A[i] is to be inserted. So, for each key A[i], L[i] = proxmap(MapKey(A[i])). We compute this by setting L[i] = P[MapKey(A[i])].

Now we do the actual sorting. For each key A[i] (for i = 0, 1,…, n – 1),we insertion-sort A[i] into its reserved subarray in A2 starting at location L[i]. Thus, if position L[i] is empty, we place K there. If not, we insert K into the sequence of keys starting at L[i] so that ascending order is preserved, moving all keys larger than K (if any) to the right to make a place to insert K into its correct location. Since each subarray is perfectly sized to hold its keys, inserting elements into A2 will never cause a key to collide with the keys in its neighboring subarray, nor will "holes" remain in the array where no key is placed. Since the keys in each subarray are guaranteed to be larger than the keys in the subarray to its left, inserting keys in order into each subarray results in A2 being sorted.

At this point, we show students step-by-step how the example in Fig. 1 works.

Efficiencies. We next tell students about some storage efficiencies that can be obtained. After P[i] has been computed and H[i] has been added to a running total, H[i] is no longer needed. Thus, the hit counts and proxmap can share the same array, saving us n memory slots.

Note that we are computing map keys both to determine the H values and again to determine the L values. If it is faster to look up previously computed map key values than it is to compute them again, we can save time by computing the map key values just once and storing them in L. These map key values can share the L array with the insertion locations since, once a location is computed, the map key value for that location will no longer be needed.

If the original array of keys is not required after the algorithm completes, the keys can be sorted directly in A, eliminating the need for A2. To accomplish this *in situ* sorting, we take a "musical chairs" approach.

We start with all keys having status NOT_YET_MOVED. We begin with A[0], storing this key in the keyToInsert variable. A[0] is now marked EMPTY. We head to L[0], the start of the keyToInsert's subarray. The key there has not yet been moved, so we swap it with the keyToInsert to place the key into its appropriate subarray. Once inserted, this key is marked as MOVED. We now have a new keyToInsert. We go to the start of its subarray, and if the item at this location is NOT_YET_MOVED, we swap it as before. If it is EMPTY, then we just place the key into this empty spot and go looking for a new key to insert, which is just the next key marked as NOT_YET_MOVED that we encounter when scanning A in left-to-right order.

If the key we encounter in the subarray was MOVED there, then either we swap that key with the keyToInsert or leave that key alone, whichever leaves the smaller of the two keys at the start of the subarray (as we want the keys in order). We then move to the next subarray item and check again. If the next key location is marked EMPTY, we place the keyToInsert in this empty location and scan to find a new key to insert. But if the next key location is marked NOT_YET_MOVED, we swap it with the keyToInsert as before. Finally, if the next key location was marked MOVED, we again leave the smaller of the keyToInsert or the current key and move right to check the next subarray item. If no more NOT_YET_MOVED keys are encountered when scanning left-to-right, the sorting process is complete.

As we will see in Part II of this paper, ProxmapSearch needs to use the proxmap values stored in P[i] that were computed during ProxmapSort (using the formula given above). If ProxmapSearch is not going to be performed later, then further space savings can be obtained by storing the status flags in the proxmap array, since at this point in ProxmapSort, the proxmap values are no longer needed.

ProxmapSort algorithm. We present the ProxmapSort algorithm to students in the form of a Java method (Fig. 2) that reflects the approach just explained. We further note how this method could be used in the larger context of an object-oriented Java implementation (because we use an object-oriented approach and Java 5.0 for our laboratory exercises). Implementations of ProxmapSort in Pascal, C and Java 1.2 can be found in [1], [2] and [3], respectively.

Analysis of Running Time. It's easy to show that the worst case running time of ProxmapSort is $O(n^2)$. Consider a data distribution so skewed, or a MapKey function so poorly chosen, that all keys map to one location. Then all keys will be insertion-sorted into the same subarray, and insertion sort is $O(n^2)$.

```
proxmapSort(KeyType[] A, int numberOfKeys)
{
  final int EMPTY = 0;
  final int NOT_YET_MOVED = 1;
  final int MOVED = 2;

  int[] proxmap = new int[numberOfKeys];
  int[] locations = new int[numberOfKeys];
  int[] status = new int[numberOfKeys];

  // compute hit counts; they share storage with the proxmap.
  // map keys and locations also share the same storage.
  // MapKey() is the map key function
  for (int i = 0; i < numberOfKeys; i++) // no hits yet
    proxmap [i] = 0;
  for (int i = 0; i < numberOfKeys; i++) {
    int hitLocation = MapKey(A[i]);
    locations[i] = hitLocation;
    proxmap [hitLocation]++;
  }

  // convert hit counts to a proxmap
  int nextStart = 0;
  for (int i = 0; i < numberOfKeys; i++) {
    if (proxmap[i] > 0) {
      int thisSubarraySize = proxmap[i];
      proxmap [i] = nextStart;
      nextStart += thisSubarraySize;
    }
    else
      proxmap[i] = -1;  // indicates empty subarray
  }

  // compute the insertion locations
  for (int i = 0; i < numberOfKeys; i++)
    locations[i] = proxmap [locations[i]];

  // rearrange A[i] in situ into ascending sorted order.
  // status flags can use proxmap's memory
  // if proxmap not needed for later ProxmapSearch
  for (int i = 0; i < numberOfKeys; i++)
    status[i] = NOT_YET_MOVED;

  for (int i = 0; i < numberOfKeys; i++) {
    // next key NotYetMoved is next key to insert
    if (status[i] == NOT_YET_MOVED) {
      int targetLocation = locations[i];
      KeyType keyToInsert = A[i];
      status[i] = EMPTY;
      boolean notInserted = true;

      while (notInserted) {
        KeyType tempKey;      // key being processed

        // if target position has key that has not been
        // moved, swap it with key stored there; note
        // key ís moved; key swapped out is next to move
        if (status[targetLocation] == NOT_YET_MOVED) {
          tempKey = A[targetLocation];
          A[targetLocation] = keyToInsert;
          keyToInsert = tempKey;
          status[targetLocation] = MOVED;
          targetLocation = locations[targetLocation];
        }
        // target MOVED, key belongs in this location; swap it in
        else if (status[targetLocation] == MOVED) {
          if (keyToInsert.compareTo(A[targetLocation]) < 0) {
            tempKey = A[targetLocation];
            A[targetLocation] = keyToInsert;
            keyToInsert = tempKey;
          }
          // prepare to check next subarray location
          targetLocation++;
        }
        else {
          // the target is empty; insert the key and mark
          // as MOVED; we're done with this cycle of key moves
          A[targetLocation] = keyToInsert;
          status[targetLocation] = MOVED;
          notInserted = false;
        }
      }
    }
  }
}
```

Figure 2. **The ProxmapSort Algorithm**

However, given an array A of *n* keys drawn from a uniform random distribution, ProxmapSort takes an average of 1.5 n – 0.5 unit operations in its key insertion phase, and O(n) average time to sort all the keys.

Proof: Because the preliminary passes used by ProxmapSort to compute hit counts, the proxmap, insertion locations and initial values of the flags take a fixed number of unit operations per key, it takes O(n) time to prepare for the key insertion phase. For ProxmapSort to be O(n), it remains to show that the key insertion phase is also O(n).

During the insertion phase we are essentially starting with an empty destination array of n cells and inserting new keys one-by-one. Consider the situation when we are about to insert the i^{th} key ($1 \le i \le n$). At this moment, $i - 1$ keys have already been inserted in subarrays that have been uniformly and randomly chosen. So the average length of the sequence of keys in a subarray just before we insert the i^{th} key is $(i - 1)/n$. Therefore to insert the i^{th} key costs $1 + (i - 1)/n$ basic unit operations (comparing keys, swapping keys, moving to the right one slot, and/or dropping a key into a slot). Thus, to insert all n keys into the array requires

$$\sum_{i=1}^{n}\left(1+\frac{i-1}{n}\right) = \sum_{i=1}^{n} 1 + \frac{1}{n}\sum_{i=1}^{n}(i-1) = n + \frac{1}{n}\left(\frac{n*(n-1)}{2}\right)$$
$$= n + (n-1)/2 = n + n/2 - 1/2 = 1.5n - 0.5 .$$

To show students that the results obtained theoretically hold in practice, we present a table comparing predicted results with the number of unit operations used in actual executions of ProxmapSort (Table 1). The predictive power of theory becomes quite apparent.

Data for *ProxmapSort* Insertion Phase 100 Trials		
Array size n	*av. observed #. of operations*	*predicted #. of ops*
64	1.504*n	1.492*n
128	1.498*n	1.496*n
256	1.504*n	1.498*n
512	1.502*n	1.499*n
1024	1.499*n	1.500*n

Table 1. Observed vs. Predicted Data for ProxmapSort

To drive home just how fast ProxmapSort is, we compare its actual running times to the running times of other sorting methods students have studied (Table 2).

The numbers in Table 2 are running times measured in milliticks (60,000^{ths} of a second). The results are averaged over 100 trials using randomly-chosen single-precision floating point keys. Students can see that ProxmapSort significantly outperforms the others if its keys are uniformly distributed.

array size =	64	128	256	512	1024
QuickSort	0.40	0.98	2.22	4.94	10.86
HeapSort	0.61	1.43	3.28	7.43	16.57
ProxmapSort	0.38	0.75	1.51	3.00	5.99
ShellSort	0.42	1.04	2.37	5.44	11.97
BubbleSort	2.76	11.36	46.42	189.35	766.22
InsertionSort	1.12	4.47	17.58	69.89	280.27
SelectionSort	1.40	5.56	22.18	88.66	354.48
MergeSort	0.99	2.28	5.13	11.45	25.11

Table 2. Comparing Different Sorting Methods

We also note that ProxmapSort takes about 2*n* extra space, which is more than many other sorts. We thus have a nice illustration of the classic issue of space/time trade-off.

ProxmapSearch

In Part II of this paper, we discuss ProxmapSearch, which uses the *proxmap* generated by ProxmapSort to search for keys in an array A[0..n – 1]. We show that ProxmapSearch uses only 1.5 key comparisons on average. We also discuss an “inverted” phone book of 1,000,000 entries, showing that ProxmapSearch “scales up,” i.e., continues to perform well as the search array gets very large.

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.