

# A Synopsis of ProxmapSort & ProxmapSearch

5/20/06

revised 6/15/09

updated, including to discuss ProxmapSearch, 3/28/2010

minor bug & typos fixed 5/19/2010

a few small changes to improve clarity 5/19/2011

- invented late 1980's by Thomas A. Standish, Prof. Emeritus, Bren School of ICS
- for in-depth discussion see "Using O(n) ProxmapSort and O(1) ProxmapSearch to motivate CS2 students (Part 1)" <http://portal.acm.org/citation.cfm?id=1113847.1113874> and "... (Part II)" <http://portal.acm.org/citation.cfm?id=1138403.1138427>

## The Proxmap

- Basic strategy  
Given an array:
  - map a key to a part—a subarray of—the destination array A2 by applying a "mapkey" function to each array item
  - figure out how many keys will map to the same bucket this is an array of "hit counts," H
  - figure out where each subarray will begin so that each bucket is exactly the right size to hold all the keys that will map to it this is an array of "proxmaps," P
  - for each key, compute into which bucket it will map this is an array of "locations," L
  - For each key, look up its location place it in that cell of A2 if it "collides" with an key already in that position, insert the key into the subarray at position that maintains order of the keys, moving keys > this key to the right one cell in order to free up a space for this key: Since the subarray is large enough to hold all keys that map to it, such movement will never cause the keys to move into the following subarray.
- Example and associated pseudocode: cell indices begin at 0

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

```
lastUsed = A.length - 1; // last used position of array of keys to sort
MapKey(K) = floor(K) // MapKey(K) will range from 0 to 11, so define H to be of size 12
for i = 0 to 11 // compute hit counts
    H[i] = 0;
for i = 0 to 11
{
    pos = MapKey(A[i]);
    H[pos] = H[pos] + 1;
}
```

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

```
runningTotal = 0; // compute proxmap - location of start of each subarray
for i = 0 to 11
    if H[i] = 0
        P[i] = -9; //skip this cell, since no key maps to it
    else
        P[i] = runningTotal;
        runningTotal = runningTotal + H[i];
```

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  
H 1 3 0 1 1 1 2 1 1 0 1 1  
P 0 1 -9 4 5 6 7 9 10 -9 11 12

```

for i = 0 to lastUsed // compute location – subarray – in A2 into which each item in A is to be placed
  L[i] = P[MapKey(A[i])];

```

|   |     |     |     |     |     |     |      |     |     |     |      |     |     |
|---|-----|-----|-----|-----|-----|-----|------|-----|-----|-----|------|-----|-----|
| 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 |
| H | 1   | 3   | 0   | 1   | 1   | 1   | 2    | 1   | 1   | 0   | 1    | 1   |     |
| P | 0   | 1   | -9  | 4   | 5   | 6   | 7    | 9   | 10  | -9  | 11   | 12  |     |
| L | 7   | 6   | 10  | 1   | 9   | 4   | 12   | 1   | 5   | 0   | 11   | 7   | 1   |

```

for i = 0 to lastUsed; // sort items
  A2[i] = <empty>;
for i = 0 to 12 // insert each item into subarray beginning at start, preserving order
{
  start = L[i]; // subarray for this item begins at this location
  insertion made = false;
  for j = start to <(an empty cell) or (the end of A2 is found and insertion not made)>
  {
    if A2[j] == <empty> // if subarray empty, just put item in first position of subarray
      A2[j] = A[i];
      insertion made = true;
    else if key < A2[j] // key belongs at A2[j]
      int end = j + 1; // Find end of used part of bucket – where first <empty> is
      while (A2[end] != <empty>)
        end++;
      for k = end - 1 to j // Move larger keys to the right 1 cell
        A2[k+1] = A2[k];
      A2[j] = A[i];
      insertion made = true; // Add in new key
    }
  }
}

```

|    |     |     |     |     |     |     |      |     |     |     |      |      |      |
|----|-----|-----|-----|-----|-----|-----|------|-----|-----|-----|------|------|------|
| 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  |
| H  | 1   | 3   | 0   | 1   | 1   | 1   | 2    | 1   | 1   | 0   | 1    | 1    |      |
| P  | 0   | 1   | -9  | 4   | 5   | 6   | 7    | 9   | 10  | -9  | 11   | 12   |      |
| L  | 7   | 6   | 10  | 1   | 9   | 4   | 12   | 1   | 5   | 0   | 11   | 7    | 1    |
| A2 | 0.4 | 1.1 | 1.2 | 1.8 | 3.7 | 4.8 | 5.9  | 6.1 | 6.7 | 7.3 | 8.4  | 10.5 | 11.1 |

- Rough analysis

- computing H, P and L all take  $O(n)$  time  
each is computed with one pass through an array,  
with constant time spent at each array location

— worst case:

- MapKey places all items into 1 subarray  
so, get standard insertion sort, and time of  $O(n^2)$

— best case:

- MapKey delivers the same small number of items to each part of the array  
in an order where the best case of insertion sort occurs  
each insertion sort is  $O(c)$ ,  $c$  the size of the parts; there are  $p$  parts  
 $p * c = n$ , so insertion sorts take  $O(n)$ ; thus, building proxmap is  $O(n)$

— average case:

say size of each subarray is at most  $c$ , a constant; insertion sort is then  $O(c^2)$  at worst – a constant!  
(actually much better, since we don't sort  $c$  items until the last item is placed in the bucket)  
total time is number of buckets,  $(n/c)$ , times  $O(c^2) =$  roughly  $n/c * c^2 = n * c$  so time is  $O(n)$

- Having good MapKey is imperative for avoiding worst case  
We must know something about the distribution of the data to come up with a good key
- Save time: save MapKey(i) values so don't have to re-compute them (notice they're recomputed in code above)
- Save space: if clever, can reuse array H (if made sufficiently large) to store L values, and can store values back into A (so you don't need array A2)

### ProxmapSort

- Array A2 is in sorted order, so just read out its items in order to obtain a sorted list
- Time to read out list is  $O(n)$ , so ProxmapSort is  $O(n)$  average case,  $O(n^2)$  worst case

### ProxmapSearch

- Build the proxmap structure, keeping MapKey routine, P and A2
- To search for a key, go to  $P[\text{MapKey}(k)]$ , the start of the subarray that contains the key, if it is in the data set
- Sequentially search the subarray; if find the key, return it (and associated information) if find a value  $>$  key, the key is not in the data set
- Computing  $P[\text{MapKey}(k)]$  takes  $O(1)$

If a mapkey that gives a good distribution of keys was used, each subarray is bounded above by a constant  $c$ , so at most  $c$  comparisons are needed to find the key or know it is not present; therefore ProxmapSearch is  $O(1)$ , once the proxmap has been built

If the worst map key was used, all keys are in the same subarray, so ProxmapSearch, in this worst case, will require  $O(n)$  comparisons, once the proxmap has been built