INSTITUTO DE COMPUTAÇÃO
UNIVERSIDADE ESTADUAL DE CAMPINAS

Heapsort beats Quicksort
DRAFT - September 10, 2004

Jorge Stolfi

Technical Report - IC-04-010 - Relatório Técnico

September - 2004 - Setembro

Heapsort beats Quicksort

Jorge Stolfi
Institute of Computing
University of Campinas
13084-971 Campinas, SP - Brazil.

DRAFT - September 10, 2004

Abstract

A simple modification to the textbook implementation of the heap *delete_min* algorithm cuts the number of comparisons by almost 50%. The change makes heapsort about as fast as quicksort, and even faster when the number of data elements exceeds 1000 or so.

Heapsort is widely believed to be much slower than quicksort. This statement is found in many popular introductory computer science textbooks, which generally claim that, on random (or randomized) inputs, heapsort makes about twice as many comparisons as its competitor [1, 3, 2, 4].

Here we observe that the reputed slowness of heapsort is in fact due to a seemingly innocent code simplification trick used in the heap's *delete_min* procedure. By omitting this trick (which is found in most published versions of the algorithm), the number of comparisons is reduced to about one half of its "usual" value, and heapsort becomes slightly faster than quicksort.

Standard heap delete-min. Let the heap stored in a vector $h[0..\,m-1]$, as usual. In the textbook implementation of *delete_min* [1, 3], the vacancy at $h[0]$ created by removing the root is filled with the last element $h[m-1]$, which then must be "bubbled down" to its proper place. Namely, while the relocated element $h[i]$ is smaller than its largest child $h[j]$, we swap the two elements and set $i \leftarrow j$.

The average depth of a random element in the heap is $\log_2 m + O(1)$, so the relocated element $h[m-1]$ is expected to sink again by that many levels — especially considering that it was taken from the lowest tier. Moreover, at each step of its descent two comparisons are needed: one to identify its largest child, and another one to decide whether a swap is needed. So the expected number of comparisons in the standard *delete_min* is $2\log_2 m + O(1)$.

Improved delete-min. This analysis justifies the following change in the heap removal algorithm. Instead of filling $h[0]$ with the last element right away, we first propagate the vacancy down the heap until it reaches a leaf node. That is, we repeatedly find the greatest child $h[j]$ of the vacancy $h[i]$, and set $h[i] \leftarrow h[j]$, $i \leftarrow j$, until $h[i]$ has no more children. Only then we fill the vacancy $h[i]$ with the last element $h[m-1]$. Finally, we "bubble up" the new $h[i]$ to its proper place. That is, while the new element $h[i]$ is greater than its parent $h[j]$, we exchange the two elements and set $i \leftarrow j$. See figure 1

```
int delete_min(int *h, int *n, int cmp(int a, int b), int sgn)
  /* Deletes the root element from the heap h[0..*n-1], returns it. */
  { if ((*n) <= 0)
      { error("empty heap"); }
    else
      { int m = (*n);
        /* Save current root: */
        int w = h[0];
        int i = 0;  /* h[i] is a vacant slot. */
        /* Promote child into vacancy h[i] until it reaches the base: */
        int ja = 1; /* h[ja] is the first child of h[i]. */
        while (ja < m)
          { /* Find largest child h[j] of h[i]: */
            int jb = ja + 1; /* h[jb] is the second child of h[i]. */
            int j = ((jb < m) && (sgn*cmp(h[ja], h[jb]) > 0) ? jb : ja);
            /* Promote largest child into hole: */
            h[i] = h[j]; i = j; ja = 2*i + 1;
          }
        /* One less element in heap: */
        m--;
        if (i < m)
          { /* Fill h[i] with h[m], bubble it up: */
            int v = h[m], j;
            while ((i > 0) && (sgn*cmp(v, h[j=(i-1)/2]) < 0))
              { h[i] = h[j]; i = j; }
            h[i] = v;
          }
        *n = m;
        return w;
      }
  }
```

Figure 1: The modified *delete_min* procedure.

This change, which increases the code by only a couple of lines, is advantageous because propgating the vacancy requires only one element comparison per level, rather than two. As in the creation phase, the relocated element $h[m-1]$ is expected to rise $O(1)$ levels on the average. Therefore the overall number of comparisons of *delete_min* is expected to be cut in half — a conclusion that is well supported by experiments.

The speedup may seem paradoxal, since the modified version allows the vacancy to propagate all the way to the heap's base, whereas the original version begins with the same sequence of swaps but stops earlier, at the level where $h[m-1]$ should be inserted. However, as discussed above, the correct level is likely to be very close to the base anyway; so all the extra comparisons needed to identify that level on the way down are essentially wasted.

Heapsort. This improvement in *delete_min* has a significant impact in the cost (number of comparisons) of heapsort. Recall that heapsort first inserts the n input elements, one by one, into an inverted heap (with the largest element at the root). Then the largest element from the heap is repeatedly removed and placed into the output array, from last to first. The heap can be stored in the first m elements of the same array $h[0..n-1]$ that holds the input and output data, so that only a constant amount of extra storage is needed — a convenient feature that has kept heapsort popular in spite of its perceived slowness compared to quicksort.

In the heap creation phase, the next input element $h[m]$ is inserted at very end of the heap, which is position $h[m]$ itself (so this step reduces to $m \leftarrow m+1$), and then it is "bubbled up" to its proper level. By the above reasoning, the new element is expected to rise only $O(1)$ levels. Therefore the cost of heapsort is dominated by that of *delete_min*, and the improvement above is expected to reduce that cost by 50%, in the limit of large n.

Tests. Table 1 shows the observed average and standard deviation of the comparison counts for the two versions of heapsort, standard and modified, as well as for an implementation of quicksort. For each algorithm and input data size n, the three algorithms were executed on the same set of 50 data vectors. The ith data vector was an array of pseudo-random integers generated by the `random` function from the Linux C library, with seed i. The quicksort implementation switches to insertion sort when $n \leq 5$, the threshold which appears to minimize the comparison count.

A Linux C implementation of heap insertion/deletion procedures and heapsort, incorporating this change, and the testing program above can be found in `http://www.ic.unicamp.br/~stolfi/EXPORT/heapsort.tgz`.

References

[1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[2] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, Reading, Mass., 1973.

[3] Robert Sedgewick. *Algorithms in Modula-3*. Addison-Wesley, Reading, Mass., 1993.

[4] Nivio Ziviani. *Projeto de Algoritmos*. Pioneira Thomson Learning, So Paulo, 2nd edition, 2004.

Algorithm	n	min	max	avg	sdv
standard	4	6	7	6.6	0.5
modified	4	5	8	6.6	0.8
quicksort	4	3	6	5.0	0.9
standard	16	73	91	83.5	3.6
modified	16	57	75	67.5	4.0
quicksort	16	43	69	53.1	6.7
standard	64	568	611	589.1	11.6
modified	64	404	451	425.2	11.7
quicksort	64	307	477	363.8	37.4
standard	256	3346	3442	3392.6	21.8
modified	256	2211	2307	2265.0	22.0
quicksort	256	1874	2629	2152.0	179.3
standard	1024	17575	17823	17707.7	51.7
modified	1024	11085	11309	11201.5	49.7
quicksort	1024	10437	13702	11431.4	634.2
standard	4096	87057	87462	87299.3	113.2
modified	4096	52919	53298	53141.1	95.7
quicksort	4096	51579	63633	57035.1	2712.8
standard	16384	414045	415111	414678.2	228.5
modified	16384	244846	245871	245424.0	231.5
quicksort	16384	258139	291150	270903.5	7958.1
standard	65536	1919497	1922237	1920931.2	472.9
modified	65536	1111554	1113934	1112846.9	461.2
quicksort	65536	1210014	1431337	1286681.8	48848.0

Table 1: Comparison counts for standard heapsort, modified heapsort, and quicksort on 50 arrays of n pseudo-random integers.