ELSEVIER

Theoretical Computer Science 285 (2002) 25–42

Theoretical Computer Science

www.elsevier.com/locate/tcs

QuickHeapsort, an efficient mix of classical sorting algorithms

D. Cantone *, G. Cincotti

Dipartimento di Matematica e Informatica, Università di Catania, Viale A. Doria 6, I-95125 Catania, Italy

Accepted May 2001

Abstract

We present an efficient and practical algorithm for the internal sorting problem. Our algorithm works *in-place* and, on the average, has a running-time of $\mathcal{O}(n \log n)$ in the size n of the input. More specifically, the algorithm performs $n \log n + 2.996n + o(n)$ comparisons and $n \log n + 2.645n + o(n)$ element moves on the average. An experimental comparison of our proposed algorithm with the most efficient variants of Quicksort and Heapsort is carried out and its results are discussed.

Keywords: In-place sorting; Heapsort; Quicksort; Analysis of algorithms

1. Introduction

The problem of sorting an initially unordered collection of elements is one of the most classical and investigated problems in computer science. Many different sorting algorithms exist in literature. Among the comparison based sorting methods, Quicksort [7, 15, 16] and Heapsort [4, 19] turn out, in most cases, to be the most efficient general-purpose sorting algorithms.

A good measure of the running-time of a sorting algorithm is given by the total number of key comparisons and the total number of element moves performed by it. In our presentation, we mainly focus our attention on the number of comparisons, since this often represents the dominant cost in any reasonable implementation. Accordingly, to sort n elements the classical Quicksort algorithm performs on the average $1.386n \log n - 2.846n + 1.386 \log n$ key comparisons and $\mathcal{O}(n^2)$ key comparisons in the worst-case, whereas Floyd's version [4] of the Heapsort algorithm performs $2n \log n + \Theta(n)$ key comparisons in the worst-case.

* Corresponding author.

E-mail addresses: cantone@dmi.unict.it (D. Cantone), cincotti@dmi.unict.it (G. Cincotti).

0304-3975/02/$ - see front matter
PII: S0304-3975(01)00288-2

Several variants of Heapsort have been reported in literature. One of the most efficient is the Bottom-Up-Heapsort algorithm discussed by Wegener in [17], which performs on the average $n\log n + f(n)n$ key comparisons, where $f(n) \in [0.34 \ldots 0.39]$, and no more than $1.5n\log n + \mathcal{O}(n)$ key comparisons in the worst-case. In [8, 9], Katajainen uses a median-finding procedure to reduce the number of comparisons required by Bottom-Up-Heapsort in the worst-case, completely eliminating the sift-up phase. This idea has been further refined by Rosaz in [14]. It is to be noted, though, that the algorithms described in [8, 9, 14], work "in-place" and perform no more than $n\log n + \mathcal{O}(n)$ key comparisons and $n\log n + \mathcal{O}(n)$ element moves in the worst case, but they are mostly of theoretical interest only, due to the overhead introduced by the median-finding procedure.

This paper tries to build a bridge between theory and practice. More specifically, our goal is to produce an efficient and *practical* sorting algorithm which couples some of the theoretical ideas introduced in the algorithms cited above with the efficient strategy used by Quicksort.

Compared to Quicksort, our proposed algorithm, called QuickHeapsort, works "in-place", i.e. no stack is needed for recursion. Moreover, its average number of comparisons and of element moves are shown to be $n\log n + 2.996n + o(n)$ and $n\log n + 2.645n + o(n)$, respectively. Its behavior is also analyzed from an experimental point of view by comparing it to that of Heapsort, Bottom-Up-Heapsort, and some variants of Quicksort. The results show that QuickHeapsort has a very good practical behavior especially when key comparison operations are computationally expensive.

The paper is organized as follows. In Section 2 we introduce a variant of the Heapsort algorithm which does not work in-place, just for the purpose of presenting the main idea upon which QuickHeapsort is based. The QuickHeapsort algorithm is fully described and analyzed in Section 3. An experimental session with some empirical results aiming at evaluating and comparing its efficiency in practice is discussed in Section 4. Section 5 concludes the paper with some final remarks.

2. A not in-place variant of the Heapsort algorithm

In this section we illustrate a variant of the Heapsort algorithm, called External-Heapsort, which uses an external array to store its output. For such a reason, External-Heapsort is mainly of theoretical interest only and it is presented just for the purpose of introducing the main idea upon which our QuickHeapsort algorithm is based. External-Heapsort sorts n elements in $\Theta(n\log n)$ time by performing at most $n\lfloor\log n\rfloor + 2n$ key comparisons and at most $n\lfloor\log n\rfloor + 4n$ element moves in the worst-case.

We begin by recalling some basic concepts relative to the classical binary-heap data structure. A *max-heap* is a binary tree with the following properties:

(1) it is *heap-shaped*: every level is complete, with the possible exception of the last one; moreover the leaves in the last level occupy the left-most positions;

(2) it is *max-ordered*: the key value associated with each non-root node is not larger than that of its parent.

A *min-heap* can be defined by substituting the max-ordering property with the dual min-ordering one. The root of a max-heap (resp. min-heap) always contains the largest (resp. smallest) element of the heap. We refer to the number of elements in a heap as its *size*; the *height* of a heap is the height of its associated binary tree.

A heap data structure of size n can be implicitly stored in an array $A[1 \dots n]$ with n elements, without using any additional pointer, as follows. The root of the heap is the element $A[1]$. Left and right children (if they exist) of the node stored into $A[i]$ are, respectively, $A[2i]$ and $A[2i+1]$, and the parent of the node stored into $A[i]$ (with $i>1$) is $A[\lfloor i/2 \rfloor]$.

In all Heapsort algorithms, the input array is sorted in ascending order, by first building a max-heap and then performing n extractions of the root element. After each extraction, the element in the last leaf of the heap, i.e. the right-most leaf at maximum depth, is moved into the root and the heap size is decreased by one unit. Subsequently, the root is moved down along a suitable path until the max-ordering property is restored (cf. [4]).

Bottom-Up-Heapsort works much like the classical Heapsort algorithm. The only difference lies in the rearrangement strategy used after each extraction of the largest element. As above, after each extraction, the element in the last leaf is moved into the root and the heap size is decreased. Subsequently, starting from the root and iteratively moving down to the child containing the largest key (*bottom phase*), when a leaf is reached it climbs up until it finds a node x with a key larger than the root key (*sift-up phase*). Finally, all elements in the path from x to the root are shifted one position up and the old root is moved into x (cf. [17]).

2.1. External-Heapsort

The algorithm External-Heapsort, whose pseudo-code is shown in Fig. 1, takes the elements of the input array $A[1 \dots n]$ and rearranges them in ascending sorted order into the output array $Ext[1 \dots n]$.

As other Heapsort algorithms, after constructing an initial max-heap, External-Heapsort performs a sequence of n extractions of the largest element. However, extracted elements are stored into the output array in reverse order, rather than moved at the end of the heap. After each extraction, the heap properties are restored by a procedure similar to the bottom-phase of Bottom-Up-Heapsort. Specifically, starting at the root of the heap, the child with the largest key is chosen and it is moved one level up. The same step is iteratively repeated until a leaf, called *special leaf*, is reached. At this point the value $-\infty$ is stored into the special leaf key field.[1] The path from the root to the special leaf is called *special path*.

[1] We assume that a key value $-\infty$, smaller than all keys occurring in $A[1 \dots n]$, is available.

External-Heapsort

```
procedure  External-Heapsort (Input A : Array; Input n : Integer;
                              Output Ext : Array)
  var  j, l : Integer;
    begin
        Build-Heap (A, n);
        for j := n downto 1 do
            Ext[j] := A[1];
            l := Special-Leaf(A, n);
            A[l] := −∞;
        end for;
    end;

function  Special-Leaf (Input-Output A : Array , n : Integer) : Integer
  var  i : Integer;
    begin
        i := 2;
        while (i < n) do
            if (A[i] < A[i + 1]) then  i := i + 1;  end if;
            A[⌊i/2⌋] := A[i];
            i := 2i;
        end while;
        if (i = n) then
            A[i/2] := A[n];
            i := 2i;
        end if;
        return i/2;
    end;
```

Fig. 1. Pseudo-code of External-Heapsort algorithm.

Unlike Bottom-Up-Heapsort, External-Heapsort does not require the sift-up phase; on the other hand, it is to be noticed that the length of special paths does not decrease during the execution of the algorithm.

External-Heapsort makes use of the procedure Build-Heap and the function Special-Leaf. Build-Heap(A,n) rearranges the input array $A[1\ldots n]$ into a classical max-heap, e.g., by using the standard heap-construction algorithm by Floyd [4]. Starting at the root of the heap contained in $A[1\ldots n]$, the function Special-Leaf(A,n) returns the index of the special leaf found; such function, whose pseudo-code is also shown in Fig. 1, assumes that the value contained in the root of the heap has already been disposed of.

Correctness of the algorithm follows by observing that if a node x contains the key $-\infty$, then each node in the sub-tree rooted at x contains $-\infty$ as well. It is easy to check that the max-ordering property is fulfilled after each extraction.

We proceed now to estimate the number of key comparisons and elements moves performed by External-Heapsort both in the worst and in the average case.[2]

Many variants for building heaps have been proposed in the literature [1, 6, 12, 17, 18], requiring quite involved implementations. Since our goal is to give a practical and efficient general-purpose sorting algorithm, we simply use the classical heap-construction procedure due to Floyd [4, 10], which has the lowest overhead.

The following results will be used in the analysis of QuickHeapsort.

Lemma 1 (Floyd [4], Knuth [10]). *In the worst-case, the classical heap-construction algorithm builds a heap with n elements by performing at most* $2n$ *key comparisons and* $2n$ *element moves.*

Lemma 2 (Doberkat [2], Knuth [10], Wegener [17]). *On the average, the classical heap-construction algorithm builds a heap with n elements by performing about* $1.881n$ *key comparisons and* $1.531n$ *element moves.*

The number of key comparisons and element moves performed by the Special-Leaf function obviously depends only on the size of the input array, so that worst- and average-case values coincides for it.

Lemma 3. *Given a heap of size n, the Special-Leaf function performs exactly* $\lfloor \log n \rfloor$ *or* $\lfloor \log n \rfloor - 1$ *key comparisons and the same number of element moves.*

Proof. It is enough to observe that the length of any special path is either $\lfloor \log n \rfloor$ or $\lfloor \log n \rfloor - 1$. □

The preceding lemmas yield immediately the following result.

Theorem 4. *External-Heapsort sorts n elements in* $\Theta(n \log n)$ *worst-case time by performing fewer than* $n\lfloor \log n \rfloor + 2n$ *key comparisons and* $n\lfloor \log n \rfloor + 4n$ *element moves. Moreover, on the average,* $H_{\mathrm{avg}}^{[c]}(n)$ *key comparisons and* $H_{\mathrm{avg}}^{[m]}(n)$ *element moves are performed, with*

$$H_{\mathrm{avg}}^{[c]}(n) \approx n \log n + 1.881n + \mathrm{o}(n),$$

$$H_{\mathrm{avg}}^{[m]}(n) \approx n \log n + 3.531n + \mathrm{o}(n).$$

Proof. The total number of key comparisons is obtained by counting the comparisons required both to build an heap of size n and to execute n times the Special-Leaf function.

[2] In the average-case analysis we make use of the assumption that all input permutations are equally likely.

Up to an additional term $2n$, which counts the number of assignments[3] inside the for-loop occurring in the External-Heapsort procedure, the total number of element moves can be computed much in the same way. □

In the following, we will also use a min-heap variant of the External-Heapsort algorithm. In particular, special paths in min-heaps are obtained by following the children with smallest key. In addition, the min-heap variant of the External-Heapsort algorithm uses the value $+\infty$ in place of $-\infty$. Obviously, the same complexity analysis can be carried out for the min-heap variant.

3. QuickHeapsort

In this section, a practical and efficient in-place sorting algorithm, called QuickHeapsort, is presented. It is obtained by a mix of two classical algorithms: Quicksort and Heapsort. More specifically, QuickHeapsort combines the Quicksort partition step with two *adapted* min-heap and max-heap variants of the External-Heapsort algorithm presented in the previous section, where in place of the infinity keys $\pm\infty$, only occurrences of keys in the input array are used.

As we will see, QuickHeapsort works in place and is completely iterative, so that no additional space is required.

The computational complexity analysis of the proposed algorithm reveals that the number of key comparisons performed on the average is $n\log n + 2.996n + o(n)$, with n the size of the input, whereas the worst-case analysis remains identical to that of the classical Quicksort algorithm. From an implementation point of view, QuickHeapsort preserves Quicksort efficiency, and in some cases it has better running times than Quicksort, as the experimental Section 4 illustrates.

Analogously to Quicksort, the first step of QuickHeapsort consists in choosing a *pivot*, which is used to partition the array. We refer to the sub-array with smaller size as *heap area* and to the one with larger size as *work area*. Depending on which of the two sub-arrays is taken as heap-area, the adapted max-heap or min-heap variant of External-Heapsort is applied and the work area is used as an external array. At the end of each stage, the elements moved in the work area are in correct sorted order and the remaining unsorted part of the array can be processed iteratively in the same way.

A more detailed description of the QuickHeapsort algorithm follows:

(1) Let $A[1\ldots n]$ be the input array of n elements to be sorted in ascending order. A pivot M, of index m, is chosen in the set $\{A[1], A[2], \ldots, A[n]\}$. As in Quicksort, the choice of the pivot can be done in a deterministic way (with or without sampling) or randomly. The computational complexity analysis of the algorithm is obviously influenced by the choice adopted.

[3] As will be clear in the next section, it is convenient to count the assignment of $-\infty$ to a node as an element move.

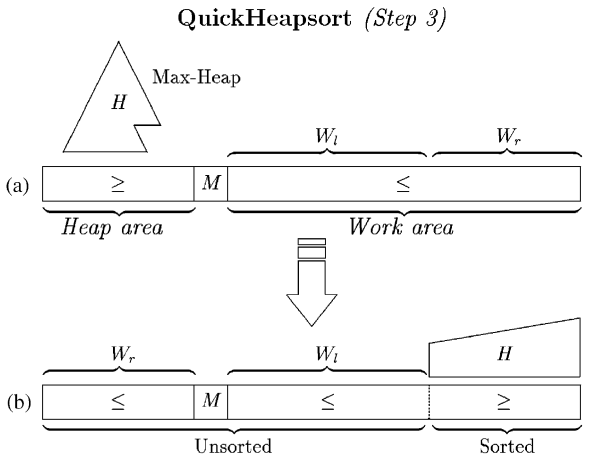


Fig. 2. Rearrangement due to step (3) in the case in which the *heap area* is on the left of the *work area*. (a) shows the input array before step (3): all the elements in the heap area, whose collection is denoted by H, are greater than or equal to the pivot M and the elements in the work area are less than or equal to M; moreover, the elements in the work area are conceptually partitioned in two sets, W_l and W_r, in such a way that $|W_r| = |H|$. (b) shows the input array after step (3): the elements in W_r have been moved in the old heap area, the elements of H have been moved in ascending sorted order in the right-most part of the input array, and all the elements in W_l remain untouched.

(2) The array $A[1 \ldots n]$ is partitioned into two sub-arrays, $A[1 \ldots Pivot-1]$ and $A[Pivot + 1 \ldots n]$, such that $A[Pivot] = M$, the keys in $A[1 \ldots Pivot - 1]$ are larger than or equal to M, and the keys in $A[Pivot + 1 \ldots n]$ are smaller than or equal to M.[4] The sub-array with the smaller size is assumed to be the *heap area*, whereas the larger one is treated as the *work area*.[5]

(3) Depending on which sub-array is taken as heap area, the adapted max-heap or min-heap variant of External-Heapsort is called, with the work area used as external array. Moreover occurrences of keys contained in the work area are used in place of the infinity values $\pm\infty$.

More precisely, if $A[1 \ldots Pivot - 1]$ is the heap area, then the max-heap version of External-Heapsort is applied to it using the right-most region of the work area as external array. In this case, at the end of the stage, the right-most region of the work area will contain the elements formerly in $A[1 \ldots Pivot - 1]$ in ascending sorted order, and the elements previously contained in the right-most region of the work area have been moved in the heap area. Such a situation is depicted in Fig. 2.

Similarly, if $A[Pivot + 1 \ldots n]$ is the heap area, then the min-heap version of External-Heapsort is applied to it using the left-most region of the work area as external array. In this case, at the end of the stage, the left-most region of the

[4] Observe that Quicksort partitions the array in the reverse way.

[5] If the two sub-arrays have the same size, a choice can be made non-deterministically.

work area will contain the elements formerly in $A[Pivot+1\ldots n]$ in ascending sorted order.

(4) The element $A[Pivot]$ is moved in the correct place and the remaining part of $A[1\ldots n]$, i.e. the heap area together with the unused part of the work area, is sorted iteratively in same fashion.

The pseudo-code of the algorithm is given in Fig. 3. QuickHeapsort makes use of some sub-routines, described below. The function Choose-Pivot(A,l,r) returns the index m of the pivot chosen in $A[l\ldots r]$ according to some criterion. Given the sub-array $A[l\ldots r]$ and the array entry $M=A[m]$, the function Reverse-Partition(A,l,r,m) moves the elements as explained in step (2) of the above description and returns the index *Pivot* where the chosen pivot M is now stored. The procedure Build-MaxHeap(A,l,r) (resp. Build-MinHeap) rearranges the sub-array $A[l\ldots r]$ into a classical max-heap (resp. min-heap) [1, 4, 6, 12, 17]; the root of the heap is always contained in $A[l]$. The function Special-MaxLeaf(A,l,r) (resp. Special-MinLeaf) works as described in Section 2, restricted to the heap contained in the sub-array $A[l\ldots r]$.

Correctness of QuickHeapsort follows from that of the max-heap and min-heap variants of External-Heapsort, by observing that assigning to a special leaf a key value taken in the work area is completely equivalent to assigning the key value $-\infty$, in the case of the max-heap variant, or the key value $+\infty$, in the case of the min-heap variant.

3.1. Average-case analysis of QuickHeapsort

For simplicity, we will carry out our analysis only in the case in which pivots are chosen deterministically and without sampling, e.g. always the first element of the array is chosen. We first prove a technical lemma.

Lemma 5. *Let* $H(n)=n\log n+\alpha n+\mathrm{o}(n)$ *and let* $f_1(n), f_2(n)$ *be functions of type* $\beta n+\mathcal{O}(1)$, *with* $\alpha,\beta\in\mathbb{R}$, *for* $n\in\mathbb{N}$. *The solution to the following recurrence equations, with initial conditions* $C(0)=C(1)=0$ *and* $C(2)=1$:

$$C(2n)=\frac{1}{2n}[(2n+1)C(2n-1)-C(n-1)+H(n-1)+f_1(n)], \tag{1}$$

$$C(2n+1)=\frac{1}{2n+1}[2(n+1)C(2n)-C(n)+H(n)+f_2(n)], \tag{2}$$

for $n\geqslant 1$, *is*:

$$C(n)\approx n\log n+(\alpha+\beta-2.8854)n+\mathrm{o}(n).$$

Proof. Among the reasonable solutions, we posit the trial solution

$$C(n)=an\log n+bn+g(n) \tag{3}$$

QuickHeapsort

```
procedure QuickHeapsort (Input-Output A : Array; Input n : Integer)
  var j, l, m, Left, Right, Pivot : Integer;
      PivotEntry : Array entry;

begin
   Left := 1;
   Right := n;
   while (Left < Right) do
      m := Choose-Pivot (A, Left, Right);
      Pivot := Reverse-Partition (A, Left, Right, m);
      PivotEntry := A[Pivot];
      if (Right + Left > 2 · Pivot)
         then
            Build-MaxHeap (A, Left, Pivot − 1);
            A[Pivot] := A[Right];
            for j := 0 to (Pivot − Left − 1) do
               A[Right − j] := A[Left];
               l := Special-MaxLeaf (A, Left, Pivot−1);
               A[l] := A[Right − j − 1];
            end for;
            Right := Right − (Pivot − Left) − 1;
            A[Right + 1] := PivotEntry;
         else
            Build-MinHeap (A, Pivot + 1, Right);
            A[Pivot] := A[Left];
            for j := 0 to (Right − Pivot − 1) do
               A[Left + j] := A[Pivot + 1];
               l := Special-MinLeaf (A, Pivot+1, Right);
               A[l] := A[Left + j + 1];
            end for;
            Left := Left + (Right − Pivot) + 1;
            A[Left − 1] := PivotEntry;
         end if;
      end while;
end;
```

Fig. 3. Pseudo-code for the QuickHeapsort algorithm.

with $a, b \in \mathbb{R}$ and where $g(n)$ is any function such that

- $g(n) = o(n)$, and
- $g(2n) - g(2n-1) = o(1)$.

Notice that all linear combinations of functions of the type $n^{\varepsilon}\log^{k} n$, with $\varepsilon<1$ and $k\in\mathbb{R}$ enjoy the above two properties.

In some of the calculations below we need to manipulate expressions of the form $\log(m\pm 1)$ with $m\in\mathbb{N},\ m>1$. By expanding the natural logarithm $\ln(1+x)$, for small $x\in\mathbb{R}$, we obtain $\ln(1+x)=x+\Theta(x^2)$, so that $\ln(m\pm 1)=\ln(m(1\pm 1/m))=\ln m\pm 1/m+\Theta(1/m^2)$. Hence, we get

$$\log(m\pm 1)=\log m\pm\frac{s}{m}+\Theta\left(\frac{1}{m^2}\right),$$

where $s=1/(\ln 2)\approx 1.4427$.

Using the above expression in the definition of $H(n)$ and in (3), we get

$$H(n-1)=n\log n+\alpha n+\mathrm{o}(n), \tag{4}$$

$$C(n-1)=an\log n+bn-a\log n-as-b+g(n-1)+\Theta\left(\frac{1}{n}\right), \tag{5}$$

$$C(2n-1)=2an\log n+2(a+b)n-a\log n - a-as-b+g(2n-1)+\Theta\left(\frac{1}{n}\right). \tag{6}$$

By substituting (4)–(6) into Eq. (1) and observing that

$$C(2n)=2an\log n+2(a+b)n+g(2n),$$

we obtain

$$\left(\frac{a-1}{2}\right)\log n+(b-\alpha-\beta+2as)+g(2n)-g(2n-1)=\mathrm{o}(1),$$

which, recalling that $g(2n)-g(2n-1)=\mathrm{o}(1)$, yields the system of asymptotic *consistency requirements*

$$a-1=0,$$

$$b-\alpha-\beta+2as=0.$$

Thus, we have[6]

$$a=1,\qquad b=\alpha+\beta-2s.$$

Hence

$$C(n)\approx n\log n+(\alpha+\beta-2.8854)n+\mathrm{o}(n).$$

Numerical computations on an extended range of n have confirmed that the function $n\log n+(\alpha+\beta-2.8854)n$ approximates with high precision a solution to (1) and (2). □

We are now ready to show our main complexity results.

[6] Such constraints can also be obtained by expanding Eq. (2).

Theorem 6. *QuickHeapsort sorts n elements in-place in $\mathcal{O}(n\log n)$ average-case time. More specifically, it performs on the average $n\log n+2.996n+\mathrm{o}(n)$ key comparisons and $n\log n+2.645n+\mathrm{o}(n)$ element moves.*

Proof. First, we estimate the average number of key comparisons.

Let $H_{\mathrm{avg}}(n)$ (resp. $H'_{\mathrm{avg}}(n)$) be the average number of key comparisons to sort n elements with the adapted max-heap (resp. min-heap) version of External-Heapsort (see the beginning of Section 3). Plainly, we have $H_{\mathrm{avg}}(i)=H'_{\mathrm{avg}}(i)=0$, for $i=0,1$.

Let $C(n)$ be the average number of key comparisons to sort n elements with Quick-Heapsort. Notice that we have $C(0)=C(1)=0$. To compute the total average number of key comparisons we add the number of comparisons $p(m)=m+1$ needed to partition an array of size m to the average number of comparisons needed to sort the two sub-arrays of the partition. The index j denotes all the possible choices (uniformly distributed) for the pivot. Thus we have, for $n\geqslant 1$

$$C(2n) = p(2n)+\frac{1}{2n}\left[\sum_{j=1}^{n}[H_{\mathrm{avg}}(j-1)+C(2n-j)] + \sum_{j=n+1}^{2n}[H'_{\mathrm{avg}}(2n-j)+C(j-1)]\right],$$

$$C(2n+1) = p(2n+1)+\frac{1}{2n+1}\left[\sum_{j=1}^{n}[H_{\mathrm{avg}}(j-1)+C(2n+1-j)] +[H_{\mathrm{avg}}(n)+C(n)]+\sum_{j=n+2}^{2n+1}[H'_{\mathrm{avg}}(2n+1-j)+C(j-1)]\right].$$

Obviously $H'_{\mathrm{avg}}(n)=H_{\mathrm{avg}}(n)$, so by simple indices manipulation, we obtain the following two recurrence equations:

$$2nC(2n)=2n\,p(2n)+2\sum_{j=1}^{n}[H_{\mathrm{avg}}(j-1)+C(2n-j)], \tag{7}$$

$$(2n+1)C(2n+1)=(2n+1)\,p(2n+1)+[H_{\mathrm{avg}}(n)+C(n)] +2\sum_{j=1}^{n}[H_{\mathrm{avg}}(j-1)+C(2n+1-j)]. \tag{8}$$

Such recurrences depend on the previous history, but they can easily be reduced to semi-first order recurrences as follows. Let $(8)_{n-1}$ be the equation obtained from (8) by substituting the index n with $n-1$. By subtracting Eq. (7) from $(8)_{n-1}$ and from (8) we obtain the following two equations:

$$C(2n)=\frac{1}{2n}[(2n+1)C(2n-1)-C(n-1)+H_{\mathrm{avg}}(n-1)+f_1(n)],$$

$$C(2n+1)=\frac{1}{2n+1}[(2n+2)C(2n)-C(n)+H_{\mathrm{avg}}(n)+f_2(n)],$$

where

$$f_1(n) = 2n\,p(2n) - (2n-1)\,p(2n-1) = 4n \text{ and}$$

$$f_2(n) = (2n+1)\,p(2n+1) - 2n\,p(2n) = 4n+2.$$

By Lemma 5 and Theorem 4, it follows immediately that

$$C(n) \approx n\log n + 2.996n + o(n).$$

Analogous recurrence equations can be written to get the average number of element moves. In such a case, we assume that the function $H_{\text{avg}}(n)$ (resp. $H'_{\text{avg}}(n)$) denotes the average number of element moves to sort n elements with the adapted max-heap (resp. min-heap) version of External-Heapsort; moreover, we let $p(m)$ denote three times the average number $q(m)$ of exchanges used during the partitioning stage of an array of size m.

If the chosen pivot $A[1]$ is the kth smallest element in the array of size $m, q(m)$ is the number of keys among $A[2],\ldots,A[k]$ which are smaller than the pivot. There are exactly t such keys with probability

$$p_{k,t}^{(m)} = \frac{\binom{m-k}{t}\binom{k-1}{k-1-t}}{\binom{m-1}{k-1}}.$$

Averaging on t and k, we obtain

$$q(m) = \frac{1}{m}\sum_{k=1}^{m}\sum_{t=0}^{k-1}[t\,p_{k,t}^{(m)}]$$

$$= \frac{1}{m}\sum_{k=1}^{m}\left[\frac{m-k}{\binom{m-1}{k-1}}\sum_{t=0}^{k-1}\binom{m-k-1}{t-1}\binom{k-1}{k-1-t}\right] = \frac{1}{6}(m-2),$$

where the last equality follows by two applications of Vandermonde's convolution.

From $p(m) = \frac{1}{2}(m-2)$, we get $f_1(n) = 2n - \frac{3}{2}$ and $f_2(n) = 2n - \frac{1}{2}$. Thus, Lemma 5 and Theorem 4 yield immediately that the average number of element moves is $n\log n + 2.645n + o(n)$. □

4. Experimental results

In this section we present some empirical results relative to the performance of our proposed algorithm. Specifically, we compare the number of basic operations and the timing results of both QuickHeapsort (QH) and its variant clever-QuickHeapsort (c-QH) (which implements the median of three elements strategy) with those of the following comparison-based sorting algorithms:

- the classical Heapsort algorithm (H), implemented with a trick which saves some element moves;

- the Bottom-Up-Heapsort algorithm (BU), implemented with bit shift operations, as suggested in [17];
- the iterative version of Quicksort (i-Q), implemented as described in [3];
- the Quicksort algorithm (Q), implemented with bounded stack usage, as suggested in [5];
- the very efficient LEDA [11] version of clever-Quicksort (c-Q), where the median of three elements is used as pivot.

Our implementations have been developed in standard C (GNU C compiler ver. 2.7) and all experiments have been carried out on a PC Pentium (133 MHz) 32MB RAM with the Linux 2.0.36 operating system.

The choice to use C, rather than C++ extended with the LEDA library, is motivated by precise technical reasons. In order to get running-times independent of the implementation of the data type ⟨array⟩ provided by the LEDA library, we preferred to implement all algorithms by simply using C *arrays*, and accordingly by suitably rewriting the source code supplied by LEDA for Quicksort.

Observe that all implementation tricks and strategies as well as the various policies to choose the pivot used for Quicksort can be applied to QuickHeapsort too.

For each size $n = 10^i$, with $i = 1, \ldots, 6$, a fixed sample of 100 arrays has been given as input to each sorting algorithm; each array in such a sample is a randomly generated permutation of the keys $1, \ldots, n$. For each algorithm, the average number of key comparisons executed, $E[C_n]$, is reported together with its relative standard deviation, $\tilde{\sigma}[C_n] = \sigma[C_n]/n$, normalized with respect to n. Analogously, $E[A_n]$ and $\tilde{\sigma}[A_n] = \sigma[A_n]/n$ refer to the number of element moves. Experimental results are collected in Fig. 4. Furthermore, in Fig. 5 we show two diagrams reporting, respectively, the normalized quantities $\tilde{E}[C_n] = E[C_n]/n$ and $\tilde{E}[A_n] = E[A_n]/n$. They confirm pretty well the theoretical results hinted at in the previous section. Notice that most of the numbers quoted in Fig. 4 about Heapsort and Quicksort are in perfect agreement with the detailed experimental study of Moret and Shapiro [13].

In our experiments, we were mainly interested in the number of key comparisons, since these represent the dominant cost, in terms of running-times, in any reasonable implementation. Observe that, in agreement with intuition, the improvement of c-Q relative to Q (in terms of number of key comparisons) is more sensible than that of c-QH relative to QH. With the exception of BU, when n is large enough, c-QH executes the smallest number of key comparisons, on the average; moreover, in agreement with the theoretical results, QH always beats both Q and i-Q. It is also interesting to note that H and BU are very stable, in the sense that they present a small variance of the number of key comparisons; this fact is well illustrated in Fig. 6, where we reported the variance ratio $\sigma[C_n]/E[C_n]$.

In Fig. 7, we report the average running times required by each algorithm to sort a fixed sample of 10 randomly chosen arrays of size $n = 10^i$, with $i \in \{4, 5, 6\}$. Such results depend on the data type of the keys to be ordered (integer or double) and the type of comparison operation used (either built-in or via a user-defined function *cmp*). In particular, six different cases were considered. In the first two cases, the

Statistics of empirical results

	$n = 10$				$n = 10^2$			
	$E[C_n]$	$\tilde{\sigma}[C_n]$	$E[A_n]$	$\tilde{\sigma}[A_n]$	$E[C_n]$	$\tilde{\sigma}[C_n]$	$E[A_n]$	$\tilde{\sigma}[A_n]$
H	39	(.21)	73	(.17)	1,030	(.08)	1,078	(.07)
BU	35	(.21)	73	(.17)	709	(.08)	1,078	(.07)
i-Q	63	(.96)	43	(.64)	990	(.61)	685	(.26)
Q	41	(.63)	27	(.32)	868	(.64)	500	(.19)
c-Q	28	(.19)	37	(.53)	638	(.29)	617	(.20)
QH	39	(.48)	54	(.39)	806	(.58)	847	(.23)
c-QH	29	(.20)	60	(.51)	714	(.22)	870	(.22)

	$n = 10^3$				$n = 10^4$			
	$E[C_n]$	$\tilde{\sigma}[C_n]$	$E[A_n]$	$\tilde{\sigma}[A_n]$	$E[C_n]$	$\tilde{\sigma}[C_n]$	$E[A_n]$	$\tilde{\sigma}[A_n]$
H	16,848	(.031)	14,074	(.024)	235,370	(.010)	174,198	(.007)
BU	10,422	(.021)	14,074	(.024)	137,724	(.006)	174,198	(.007)
i-Q	14,471	(.605)	9,146	(.106)	194,279	(.878)	114,419	(.092)
Q	13,297	(.609)	7,285	(.095)	179,948	(.654)	95,807	(.072)
c-Q	10,299	(.355)	8,543	(.102)	142,443	(.401)	109,141	(.065)
QH	11,881	(.630)	11,838	(.202)	152,789	(.664)	152,155	(.201)
c-QH	11,135	(.333)	11,959	(.182)	146,643	(.323)	152,909	(.121)

	$n = 10^5$				$n = 10^6$			
	$E[C_n]$	$\tilde{\sigma}[C_n]$	$E[A_n]$	$\tilde{\sigma}[A_n]$	$E[C_n]$	$\tilde{\sigma}[C_n]$	$E[A_n]$	$\tilde{\sigma}[A_n]$
H	3,019,638	(.0031)	2,074,976	(.0025)	36,793,760	(.0010)	24,048,296	(.0008)
BU	1,710,259	(.0024)	2,074,976	(.0025)	20,401,466	(.0007)	24,048,296	(.0008)
i-Q	2,421,867	(.7037)	1,374,534	(.0689)	28,840,152	(.6192)	16,068,733	(.0649)
Q	2,249,273	(.6828)	1,189,502	(.0726)	27,003,832	(.5389)	14,212,076	(.0635)
c-Q	1,816,706	(.3367)	1,328,265	(.0546)	22,113,966	(.2962)	15,649,667	(.0497)
QH	1,869,769	(.6497)	1,854,265	(.2003)	21,891,874	(.6473)	21,901,092	(.1853)
c-QH	1,799,240	(.3254)	1,866,359	(.1675)	21,355,988	(.3282)	21,951,600	(.1678)

Fig. 4. Average number of key comparisons and element moves over a sample of 100 random arrays of size n.

comparison operation used was the built-in one. In the third and fourth case, a simple comparison function *cmp* was used. Finally, in the last two cases, two computationally more expensive comparison functions cmp_1 and cmp_2 were used (but only with keys of type integer), to simulate situations in which the cost of a comparison operation is much higher than that of an element move.[7] The function cmp_1 (resp. cmp_2) was obtained from a simple function *cmp* by adding one call (resp. two calls) to the function "log" of the C standard mathematical library. For each case, we also report an approximation of the average time required by a single key comparison t_c and by a single element move t_m.

[7] For instance, such situations arise when the array to sort contains pointers to the actual records. A move is then just a pointer assignment, but a comparison involves at least one level of indirection, so that comparisons become the dominant factor.

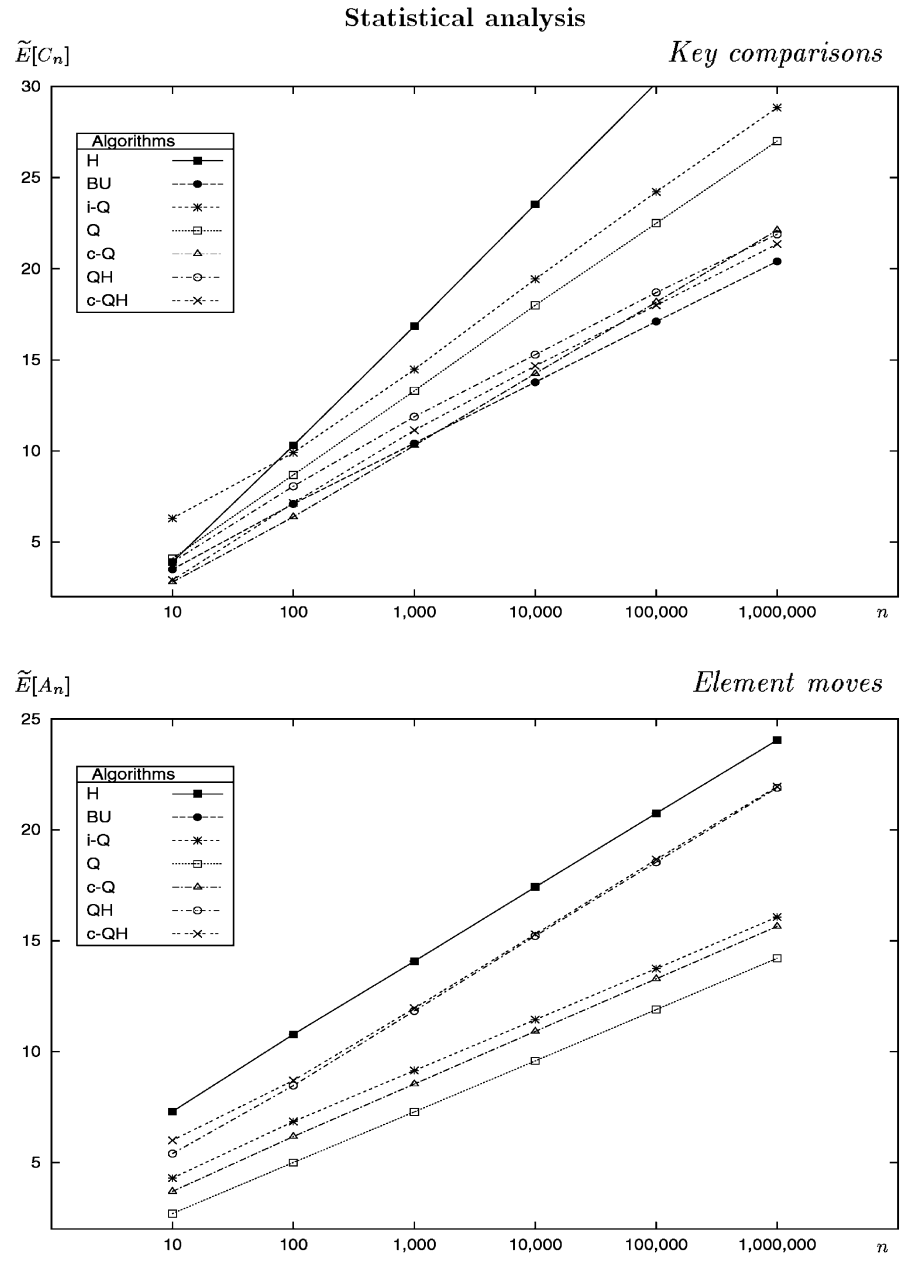
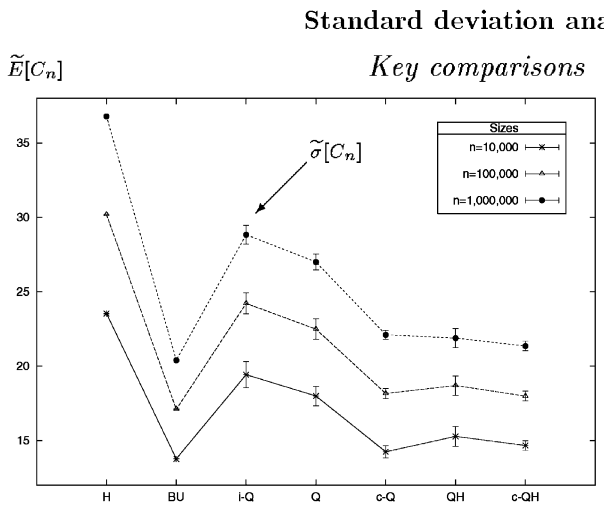


Fig. 5. Plot of the empirical data reported in Fig. 4.

Notice that in all trials, the best running times (represented in Fig. 7 as underlined boldface values) are always achieved by a "clever" algorithm, namely either c-Q or c-QH.



	$\sigma[C_n]/E[C_n]$		
$n =$	10^4	10^5	10^6
H	.0004	.0001	.0000
BU	.0004	.0001	.0000
i-Q	.0452	.0290	.0215
Q	.0363	.0303	.0199
c-Q	.0281	.0185	.0134
QH	.0434	.0347	.0296
c-QH	.0220	.0181	.0154

Fig. 6. Standard deviation analysis of the empirical data reported in Fig. 4.

Fig. 7 confirms the good behavior of all Quicksort variants i-Q, Q, c-Q, due to their very low overhead; moreover, it can be observed that though BU is characterized by the lowest number of key comparisons, it suffers from high overhead, because of its complex internal bookkeeping. On the other hand, QH and c-QH couple a quite low complexity (in terms of key comparisons) together with an overhead lower than that of BU. In fact, in most cases the running-times of QH and c-QH are between those of the variants H and BU of Heapsort and those of the variants Q, i-Q, and c-Q of Quicksort. In particular, when the cost of each key comparison is neither too small nor too high, c-QH turns out to be the best algorithm, on the average, in terms of running time.

Cache performance has considerably less influence on the behavior of sorting algorithms than does paging performance (cf. [13, Chapter 8]); for such reason, we believe that we can ignore completely possible negative effects due to caching.

Concerning virtual memory problems, i.e. paging demand, all Quicksort algorithms show good locality of reference, whereas Heapsort algorithms, and also QuickHeapsort algorithms, tend to use pages that contain the top of the heap heavily, and to use in a random manner pages that contain the bottom of the heap (cf. [13]). Such observation allows us to conclude that an execution of c-Q cannot be more penalized than an execution of c-QH by delays due to paging problems. Hence, we can reasonably conclude that the success of c-QH is not due to paging performance.

5. Conclusions

We presented QuickHeapsort, a new practical "in-place" sorting algorithm obtained by merging some characteristics of Bottom-Up-Heapsort and Quicksort. Both theoretical analysis and experimental tests confirm the merits of QuickHeapsort.

Running times analysis

	Integer			Double			cmp(Double)		
	$t_c = 0.05\mu$sec , $t_m = 0.06\mu$sec			$t_c = 0.05\mu$sec , $t_m = 0.07\mu$sec			$t_c = 0.3\mu$sec , $t_m = 0.07\mu$sec		
n=	10^4	10^5	10^6	10^4	10^5	10^6	10^4	10^5	10^6
H	0.05	0.75	12.37	0.10	1.40	20.35	0.14	1.96	27.28
BU	0.09	1.25	18.80	0.13	1.83	25.88	0.15	2.00	27.62
i-Q	0.04	0.40	4.81	0.07	0.80	9.74	0.10	1.25	15.10
Q	0.03	0.37	4.54	0.06	0.74	8.92	0.09	1.12	13.49
c-Q	**0.03**	**0.33**	**4.08**	**0.05**	**0.69**	**8.34**	**0.08**	**0.99**	**11.99**
QH	0.05	0.64	10.43	0.08	1.10	16.85	0.10	1.42	19.96
c-QH	0.04	0.65	10.48	0.08	1.11	16.92	0.10	1.38	20.05

	cmp(Integer)			cmp_1 (Integer)			cmp_2 (Integer)		
	$t_c = 0.19\mu$sec , $t_m = 0.06\mu$sec			$t_c = 2.9\mu$sec , $t_m = 0.06\mu$sec			$t_c = 4.7\mu$sec , $t_m = 0.06\mu$sec		
n=	10^4	10^5	10^6	10^4	10^5	10^6	10^4	10^5	10^6
H	0.10	1.35	19.16	0.54	7.04	88.29	1.00	12.92	159.66
BU	0.11	1.52	21.23	0.37	4.72	58.94	0.64	8.09	98.42
i-Q	0.07	0.85	10.12	0.42	5.44	64.96	0.79	10.24	120.69
Q	0.07	0.80	9.60	0.40	4.93	59.53	0.74	9.20	110.29
c-Q	**0.05**	**0.68**	**8.30**	0.35	4.42	53.82	0.64	8.22	99.24
QH	0.08	0.99	13.99	0.37	4.57	54.57	0.66	8.17	95.11
c-QH	0.07	0.98	13.98	**0.34**	**4.22**	**52.15**	**0.61**	**7.63**	**91.58**

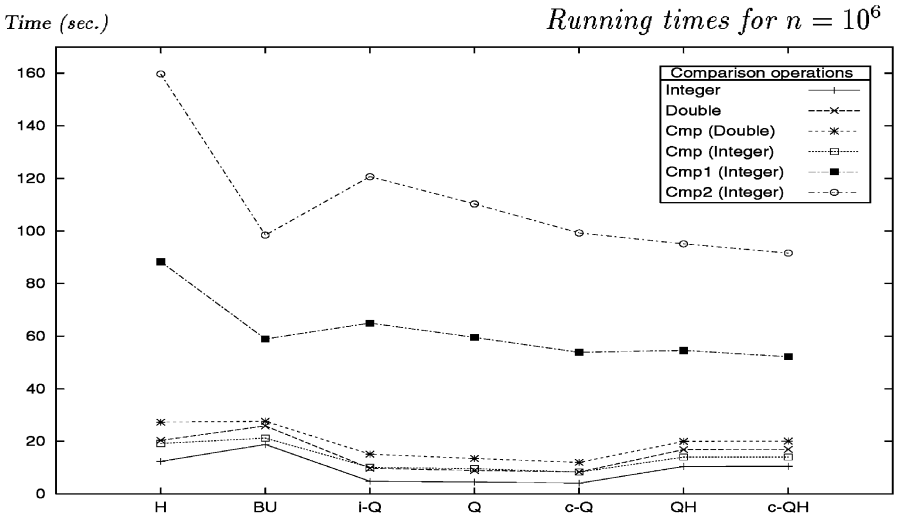


Fig. 7. Average running times in seconds, over a sample of 10 random arrays of size n, using various type of key comparison operations.

The experimental results obtained show that it is convenient to use clever-Quick-Heapsort when the input size n is large enough and the cost of each key comparison is neither too small, nor too high.

Acknowledgements

We thank Micha Hofri for helping us to solve the recurrence equations in Lemma 5.

References

[1] S. Carlsson, A variant of heapsort with almost optimal number of comparisons, Inform. Process. Lett. 24 (1987) 247–250.
[2] E.E. Doberkat, An average analysis of Floyd's algorithm to construct heaps, Inform. Control 61 (1984) 114–131.
[3] B. Durian, Quicksort without a stack, Lecture Notes in Computer Science, Vol. 233, Proc. MFCS, Springer, New York, 1986, pp. 283–289.
[4] R.W. Floyd, Treesort 3 (alg. 245), Comm. ACM 7 (1964) 701.
[5] G. Gonnet, R. Baeza-Yates, Handbook of Algorithms and Data Structures, Addison-Wesley, Reading, MA, 1991.
[6] G. Gonnet, J. Munro, Heaps on heap, Lecture Notes in Computer Science, Vol. 140, Proc. ICALP, Springer, New York, 1982.
[7] C.A.R. Hoare, Algorithm 63(partition) and algorithm 65(find), Comm. ACM 4 (7) (1961) 321–322.
[8] J. Katajainen, The ultimate heapsort, DIKU Report 96/42, Department of Computer Science, University of Copenhagen, 1996.
[9] J. Katajainen, T. Pasanen, J. Tehuola, Top-down not-up heapsort, Proc. The Algorithm Day in Copenhagen, Department of Computer Science, University of Copenhagen, 1997, pp. 7–9.
[10] D.E. Knuth, The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, MA, 1973.
[11] LEDA, Library of Efficient Data structures and Algorithms, http://www.mpi-sb.mpg.de/LEDA/leda.html.
[12] C.J. McDiarmid, B.A. Reed, Building heaps fast, J. Algorithms 10 (1989) 352–365.
[13] B.M.E. Moret, H.D. Shapiro, Algorithms from P to NP, Vol. 1: Design and Efficiency, The Benjamin Cummings Publishing Company, Menlo Park, CA, 1990.
[14] L. Rosaz, Improving Katajainen's ultimate heapsort, Technical Report No. 1115, Laboratoire de Recherche en Informatique, Université de Paris Sud, Orsay, 1997.
[15] R. Sedgewick, Quicksort, Garland Publishing, New York, 1980.
[16] R. Sedgewick, Implementing quicksort programs, Comm. ACM 21 (10) (1978) 847–857.
[17] I. Wegener, Bottom-Up-Heapsort, a new variant of Heapsort beating, on an average, Quicksort (if n is not very small), Theoret. Comput. Sci. 118 (1993) 81–98.
[18] I. Wegener, The worst case complexity of McDiarmid and Reed's variant of Bottom-Up-Heapsort is less than $n\log n + 1.1n$, Inform. Comput. 97 (1992) 86–96.
[19] J.W. Williams, Heapsort (alg.232), Comm. ACM 7 (1964) 347–348.