BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT beating, on an average, QUICKSORT (if $n$ is not very small)

Ingo Wegener*

FB Informatik, LS II, Universität Dortmund, Postfach 500500, 4600 Dortmund 50, Germany

Abstract

Wegener, I., BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT beating, on an average, QUICKSORT (if $n$ is not very small), Theoretical Computer Science 118 (1993) 81-98.

A variant of HEAPSORT, called BOTTOM-UP-HEAPSORT, is presented. It is based on a new reheap procedure. This sequential sorting algorithm is easy to implement and beats, on an average, QUICKSORT if $n \geq 400$ and a clever version of QUICKSORT (where the split object is the median of 3 randomly chosen objects) if $n \geq 16000$. The worst-case number of comparisons is bounded by $1.5n \log n + O(n)$. Moreover, the new reheap procedure improves the delete procedure for the heap data structure for all $n$.

1. Introduction

Sorting is one of the most fundamental problems in computer science. In this paper, only general (excluding BUCKETSORT) sequential sorting algorithms are studied.

In Section 2 a short review of the long history of efficient sequential sorting algorithms is given. The QUICKSORT variant where the split object is the median of three random objects is, on an average, the most efficient known algorithm for internal sorting. The new algorithm will be compared with this strong competitor, called CLEVER QUICKSORT. In Section 3 the new algorithm, called
BOTTOM-UP-HEAPSORT, is presented and its implementation is discussed. It will turn out that the algorithm can be implemented easily and that it is practically and theoretically efficient.

In Section 4 the average-case behavior of BOTTOM-UP-HEAPSORT is analyzed. For the heap creation phase exact results are obtained. For the selection phase, we run into the same problems as those encountered in the analysis of the average-case behavior of HEAPSORT. Deleting the root of a random heap and applying the reheap procedure does not lead to a random heap. The analysis is possible only under some realistic assumptions. The results are justified by experiments. The results of the experiments are quite convincing, since the variance of the number of comparisons of BOTTOM-UP-HEAPSORT is very small. The average number of comparisons grows as $n \log n + d(n)n$, where log is always log₂ and $d(n) \in [0.34, 0.39]$ depends on the binary representation of $n$. BOTTOM-UP-HEAPSORT beats, on an average, QUICKSORT if $n \geq 400$ and CLEVER QUICKSORT if $n \geq 16000$.

In Section 5 an analysis of the worst-case behavior of BOTTOM-UP-HEAPSORT is presented. The number of comparisons is bounded by $1.5n \log n + O(n)$. The paper is concluded with some remarks concerning the application of the new reheap procedure for the heap data structure and implications of this paper for teaching sorting algorithms.

Reference [15] is a preliminary version of this paper.

2. A short history of sequential sorting algorithms

General sorting algorithms are based on comparisons. We count only the essential comparisons, i.e. comparisons between objects, and not comparisons between indices. It is well known that general sorting algorithms need in the worst case at least

$$\lceil \log(n!) \rceil = n \log n - n \log e + \Theta(\log n) \approx n \log n - 1.4427n$$

comparisons, and, on an average, at least

$$\lceil \log(n!) \rceil - \frac{1}{n!} 2^{\log(n!)} + 1 \geq \lceil \log(n!) \rceil - 1$$

comparisons.

Mergesort needs, if $n = 2^k$, only $n \log n - n + 1$ comparisons, but it needs an array of length $2n$. Since $n$ can be rather large, one is interested in in-place sorting algorithms, i.e. algorithms using an array of length $n$, only a few more variables and perhaps a recursion stack of length $O(\log n)$ like QUICKSORT. Hence, MERGESORT is useful for external sorting only.

Insertionsort [13] needs less than $\log(n!) + n - 1$ comparisons, but the number of interchanges is, on an average even $\Theta(n^2)$. One is interested in sorting algorithms where the number of interchanges and other nonessential operations is bounded, on an average, by $O(n \log n)$. 
QUICKSORT [7] has many nice properties. The original variant selects a random array object \( x = a(k) \) and compares this object with all other ones. At the end of the first round, the object \( x \) is at its right place, say,

\[
x = a(i), a(1), \ldots, a(i - 1) \geq x \quad \text{and} \quad a(i + 1), \ldots, a(n) \leq x.
\]

Afterwards, only the subarrays have to be sorted. For the first round, one needs \( n - 1 \) comparisons and not more interchanges. Hence, the worst-case number of comparisons is \( \Theta(n^2) \). Let \( Q(n) \) be the average number of comparisons. Then \( Q(0) = 0, Q(1) = 0, Q(2) = 1 \) and, for \( n \geq 3 \),

\[
Q(n) = n - 1 + \frac{1}{n} \sum_{1 \leq i \leq n} (Q(i - 1) + Q(n - i)).
\]

It is well known that

\[
Q(n) = 2(n + 1)H_n - 4n,
\]

where

\[
H_n = 1 + \frac{1}{2} + \cdots + \frac{1}{n}
\]

is the \( n \)-th harmonic number.

\[
\lim_{n \to \infty} (H_n - \ln n) = E,
\]

where \( E \approx 0.5772156649 \) is the Eulerian constant. Hence,

\[
Q(n) \approx (2 \ln 2)n \log n - (4 - 2E)n + (2 \ln 2) \log n + 2E
\]

\[
\approx 1.386n \log n - 2.846n + 1.386 \log n + 1.154
\]

CLEVER QUICKSORT chooses 3 random array objects and uses the median of these 3 objects as split object. In order to compute the median of \( x, y \) and \( z \), one can compare w.l.o.g. \( x \) and \( y \) and, afterwards, \( y \) and \( z \). If \( y \) is the median (probability \( 1/3 \)), we are done. Otherwise, one has to compare \( x \) and \( z \). Afterwards, the median has to be compared only with \( n - 3 \) other objects. Hence, we need for the first round, on an average, \( n - 1/3 \) comparisons, but the split object tends to be more in the middle of all the objects. The probability that its rank equals \( i \) is \( (i-1)(n-i)/(3) \). The expected number of comparisons \( C(n) \) fulfills \( C(0) = 0, C(1) = 0, C(2) = 1 \) and, for \( n \geq 3 \),

\[
C(n) = n - \frac{1}{3} + \binom{n}{3}^{-1} \sum_{1 \leq i \leq n} (i - 1)(n - i)(C(i - 1) + C(n - i)).
\]

Kemp [8] has solved this recursion:

\[
C(3) = \frac{8}{3}, \quad C(4) = \frac{14}{3}, \quad C(5) = \frac{106}{15}
\]
and, for $n \geqslant 6$,
\[
C(n) = \frac{12}{7} (n+1)H_{n-1} - \frac{477}{147} n + \frac{223}{147} + \frac{252}{147n} \\
\approx \left( \frac{12}{7} \ln 2 \right) n \log(n-1) - \left( \frac{477}{147} - \frac{12}{7} E \right) n + \left( \frac{12}{7} \ln 2 \right) \log(n-1) \\
+ \frac{12}{7} E + \frac{223}{147} + \frac{252}{147n} \\
\approx 1.18825231n \log(n-1) - 2.255384816n + 1.1882531 \log(n-1) \\
+ 2.5065.
\]

In general, one can consider QUICKSORT variants where the split object is the median of $2k+1$ randomly chosen objects. These variants have been analyzed by van Emden [14] and Sedgewick [12]. The average number of comparisons equals
\[
(n+1)H_{n+1}/(H_{2k+2} - H_{k+1}) + O(n)
\]
if $k$ is a constant. The linear term is not known explicitly. The sequence
\[
H_{2k+2} - H_{k+1} = \frac{1}{k+2} + \cdots + \frac{1}{2k+2}
\]
is increasing and is converging to $\ln 2$. Hence, the leading term of the average-case complexity is $\alpha_k n \log n$, where $\alpha_k > 1$ and $\alpha_k \to 1$ as $k \to \infty$. It may be expected that the multiplier of the linear term is increasing for increasing $k$. Variants where $k = k(n)$ is increasing are not analyzed. For large $k$, in particular $2k+1 = n$, we cannot get efficient QUICKSORT variants, since there do not exist median algorithms of sufficient efficiency. One also can improve QUICKSORT by applying INSERTIONSORT or MERGESORT for subarrays of length smaller than $l$ (see [12]). In order to avoid the disadvantages of INSERTIONSORT or MERGESORT, $l$ is always chosen as an appropriate constant. These improvements change only the linear term of the number of comparisons. Altogether, no QUICKSORT variant whose average-case complexity is $n \log n + o(n \log n)$ is known. From the practical point of view, it seems to be fair to compare BOTTOM-UP-HEAPSORT with CLEVER QUICKSORT.

HEAPSORT [17, 6] works in-place and the number of interchanges is at most half the number of comparisons. The worst-case number of comparisons is $2n \log n + \Theta(n)$. It is the only in-place sorting algorithm with $O(n \log n)$ comparisons and interchanges. Therefore, HEAPSORT is taught in all elementary courses on efficient algorithms and data structures. But, on an average, and even in almost all cases, HEAPSORT is worse than QUICKSORT or CLEVER QUICKSORT.

The basic idea of BOTTOM-UP-HEAPSORT goes back to Floyd (see [9]) and has been rediscovered quite often. There are some variants which guarantee quite a good behavior in the worst case. The algorithm of Carlsson [1] needs $n \log n + \Theta(n \log \log n)$ comparisons not only in the worst case but also in the average case. Hence, CLEVER
QUICKSORT is better if $n \leq 10^{16}$. The algorithm of Xunrang and Yuzhang [18] guarantees a worst-case behavior of $(4/3)n \log n$. Our considerations will show that their variant is by almost $n$ comparisons worse than BOTTOM-UP-HEAPSORT in the average case. Our aim is to discuss the HEAPSORT variant with the best average-case behavior and not to discuss changes which make the analysis simpler but the average-case behavior worse.

3. BOTTOM-UP-HEAPSORT

We recall the original version of HEAPSORT. We consider an array $a(1), \ldots, a(n)$, with objects from an ordered set $S$. The heap property is fulfilled for position $i$ if $a(i) \leq a(2i)$ or $i > \lfloor n/2 \rfloor$ and $a(i) \leq a(2i + 1)$ or $i \geq \lceil n/2 \rceil$. The array is called a heap if the heap property is fulfilled for all positions.

The sons of position $i$ are the positions $2i$, if $2i \leq n$, and $2i + 1$, if $2i + 1 \leq n$, and its father is position $\lfloor i/2 \rfloor$, if $i \geq 2$. In this way, the array is considered as a binary tree which can be implemented without pointers. The procedure $\text{reheap}(m, i)$ considers only the array positions $1, \ldots, m$ and looks at the subtree with root $i$. If the heap property is fulfilled for all positions of the subtree besides (perhaps) the root, $\text{reheap}(m, i)$ transforms this subtree into a heap. This leads to the well-known HEAPSORT algorithm.

**HEAPSORT**

1. For $i = \lfloor n/2 \rfloor, \ldots, 1$: $\text{reheap}(n, i)$ (heap creation phase).
2. For $m = n, \ldots, 2$:
   - interchange $a(1)$ and $a(m)$;
   - if $m \neq 2$ then $\text{reheap}(m - 1, 1)$ (selection phase).

We give an informal description of the classical $\text{reheap}$ procedure.

**Procedure $\text{reheap}(m, i)$**

1. If $i > m/2$, STOP.
2. If $i < m/2$, compute with 2 comparisons $MIN$, the minimum of $a(i)$, $a(2i)$ and $a(2i + 1)$. If $i = m/2$, $MIN = \min\{a(i), a(2i)\}$.
3. (a) $MIN = a(i)$. STOP.
   (b) Not (a) and $MIN = a(2i)$. Interchange $a(i)$ and $a(2i)$, $\text{reheap}(m, 2i)$.
   (c) Not (a) or (b). Interchange $a(i)$ and $a(2i + 1)$, $\text{reheap}(m, 2i + 1)$.

If the considered subtree has depth $d$, $\text{reheap}(m, i)$ needs at most $2d$ comparisons and $d$ interchanges. In a heap, large objects tend to be situated near the leaves. Furthermore, in a binary tree, most of the nodes are near the leaves. During the heap creation phase, the root of the considered subtree contains a random object. By the above considerations, there is high probability that $\text{reheap}$ has to go down the tree to a position near the leaves. During the selection phase, the root of the tree contains
a former leaf object. Hence, the probability that we have to go down to a position near
the leaves is even larger than for the heap creation phase.

HEAPSORT uses in the worst case, and on an average, \(2n \log n + \Theta(n)\) comparisons. The reason for the factor 2 is that we need, in step 2 of reheap 2, comparisons for
the computation of the minimum of 3 objects. We try to save one of these two comparisons and search directly for that leaf of the considered subtree which we reach
by going always to the son with the smaller object. This leaf is called special leaf.

**Procedure leaf-search\((m, i)\)**

\((0)\) \(j := i.\)

\((1)\) while \(2j < m\) do begin
  
  if \(a(2j) < a(2j + 1)\) then \(j := 2j\)
  
  else \(j := 2j + 1\) end.

\((2)\) if \(2j = m\) then \(j := m.\)

It is obvious that \(\text{leaf-search}(m, i)\) computes the position \(j\) of the special leaf. The
path which is used is called special path. If the considered subtree has depth \(d\), \(\text{leaf-}
search\) needs either \(d - 1\) comparisons (the special path is called short) or \(d\) compar-
isons (the special path is called long) and no interchanges.

Let \(b(1), \ldots, b(r)\) be the objects on the special path excluding the root, \(b(0) := -\infty, \)
\(b(r + 1) := \infty\) and \(x\) the object at the root. Since the heap property is fulfilled for all
positions but the root, \(b(0) \leq b(1) \leq \cdots \leq b(r) \leq b(r + 1)\). Procedure reheap finds the
smallest \(l\) such that \(b(l) < x \leq b(l + 1)\). At the end of reheap, \(x\) is placed at the position of
\(b(l), \) and \(b(1), \ldots, b(l)\) are placed at the positions of their fathers. We look for the
largest \(l\) such that \(b(l) \leq x < b(l + 1)\). If all objects are different, this \(l\) is the same as that
computed by reheap. We expect that \(l\) is large, most probably \(l = r\). Hence, it is better
to search bottom-up.

**Procedure bottom-up-search\((i, j)\)**

\((* j\) is the output of \(\text{leaf-search}(m, i)\) \(*\))

\((1)\) while \(a(i) < a(j)\) do \(j := \lfloor j/2 \rfloor.\)

We can search the path bottom-up, although we have not stored the path. The
father of each position can always be computed directly. Obviously, \(\text{bottom-up-search}\)
"finds" the largest \(l\) such that \(b(l) \leq x < b(l + 1)\). We can easily prevent that \(a(i)\) is
compared with itself by checking whether \(i = j\). In Section 4 we show that, on an
average, the number of comparisons during calls of \(\text{bottom-up-search}\) is only by
a small constant factor larger than the number of these calls. Finally, we have to
rearrange the objects. The root object should take the position of \(b(l)\) and \(b(1), \ldots, b(l)\)
should take the positions of their fathers. The actual positions of \(b(1), \ldots, b(l)\) are
\(\lfloor j/2^{l-1} \rfloor, \ldots, j\), respectively, \(l\) is the depth of position \(j\) with respect to the root \(i;\) hence,
\(l = \lfloor \log(j/i) \rfloor.\) The following procedure does the job.
**BOTTOM-UP-HEAPSORT**, a new variant

**Procedure interchange-1**(i, j)

(*) j is the output of bottom-up-search(i, j) *)

1. \( l:=\lfloor \log(j/i) \rfloor, x:=a(i) \).
2. for \( k=l-1, \ldots, 0 \): \( a(\lfloor j/2^k \rfloor) := a(\lfloor j/2^k \rfloor) \).
3. \( a(j) := x \).

This is a very efficient implementation if \( \lfloor j/2^k \rfloor \) is computed by a simple shift operation erasing the last \( k \) bits of \( j \) and not by a floating-point division. The numbering of the heap positions has the property that all nodes of depth \( d \) have \( d+1 \) significant bits. Let \( \text{bin}(i) \) be the number of significant bits of \( i \). Then \( l:=\lfloor \log(j/i) \rfloor \) can be computed efficiently, i.e. without floating-point division and log-operation, as \( \text{bin}(j) - \text{bin}(i) \). The number of essential assignments is only \( l+2 \). If one likes to compare BOTTOM-UP-HEAPSORT with the variants of QUICKSORT, one has to remember that, for QUICKSORT, interchanges are counted and 3 assignments are necessary for an interchange of 2 objects. If the shift operation is not available, one can use the following procedure.

**Procedure interchange-2**(i, j)

1. \( l:=\lfloor \log(j/i) \rfloor, x:=a(j), a(j) := a(i) \).
2. while \( j > i \) do begin
   interchange \( a(\lfloor j/2 \rfloor) \) and \( x \),
   \( j:=\lfloor j/2 \rfloor \) end.

BOTTOM-UP-HEAPSORT works like HEAPSORT but reheap is replaced by bottom-up-reheap.

**Procedure bottom-up-reheap**(m, i)

\( \text{leaf-search}(m, i); \)
\( \text{bottom-up-search}(i, j); \)
\( \text{interchange-1}(i, j). \)

The following statement on bottom-up-reheap is obvious. If in the array, with positions 1, \ldots, m, the heap property is fulfilled for all nodes of the subtree rooted at \( i \) besides the root, then bottom-up-reheap(m, i) returns a heap on this subtree. If all objects are different, it returns the same heap as reheap(m, i).

We may save some further assignments. During the selection phase, the interchange of \( a(1) \) and \( a(m) \) is followed by reheap(\( m-1, 1 \)). The interchange is done by three assignments: \( x:=a(m), a(m):=a(1), a(1):=x \). The next assignment is done by procedure interchange-1 which starts with \( x:=a(1) \). These 4 assignments can be replaced by the 2 assignments \( x:=a(m), a(m):=a(1) \) if bottom-up-search refers to \( x \) instead of \( a(i) = a(1) \).

People who like to compare the CPU time of BOTTOM-UP-HEAPSORT with the CPU time of CLEVER QUICKSORT should note that the fastest QUICKSORT implementations need some more comparisons than stated in Section 2, and that they
I. Wegener

use a stack of length \( \Omega(\log n) \). In such a competition, BOTTOM-UP-HEAPSORT is able to store the special path and its length. This saves all divisions or shift operations.

### 4. The average-case analysis

The average-case analysis assumes that \( a(1), \ldots, a(n) \) are random real numbers drawn independently according to the uniform distribution on the interval \([0, 1]\). Doberkat [3, 4] has shown that this assumption is equivalent to the assumption that \((a(1), \ldots, a(n))\) is a random permutation of \((1, \ldots, n)\). In particular, the probability that \(a(i) = a(j)\) for some \(i \neq j\) equals 0.

We start our analysis with the heap creation phase. This phase is well understood, since HEAPSORT and, therefore, also BOTTOM-UP-HEAPSORT creates random heaps (see [4]). First we investigate the number of comparisons during the calls of leaf-search.

**Lemma 4.1.** The number of comparisons during the calls of leaf-search and the heap creation phase is at least \( n - \lceil \log(n + 1) \rceil - \lceil \log n \rceil + 1 \) and at most \( n - 2 \).

**Proof.** Procedure leaf-search is called for \((n, \lfloor n/2 \rfloor), \ldots, (n, 1)\). The whole array is considered, and each inner node of the tree is once the root of leaf-search. One walks from this root to some leaf.

First we prove that the number of comparisons equals \( n - \lceil \log(n + 1) \rceil \) for \( n = 2^k - 1 \). Then \( 2^{k-1} - (i + 1) \) inner nodes, where \( 1 \leq l \leq k - 1 \), cause \( l \) comparisons each. The whole number of comparisons equals \( \sum_{1 \leq l \leq k-1} 2^{k-1} - 1 = 2^k - 1 - k = n - \lceil \log(n + 1) \rceil \).

Now we fill the next level of the tree from left to right with nodes, i.e. we consider the cases \( n = 2^k, \ldots, 2^k + 1 - 1 \). Let \( i = n - (2^k - 1) \) be the number of nodes on the \( k \)-th level. Remember that no new comparison is necessary for \( i = 1 \). There are \( \lceil (i - 1)/2 \rceil \) new inner nodes on depth \( k - 1 \), each causing a new comparison, and \( \lceil (i - 1)/2 \rceil \) inner nodes on depth \( k - r \), \( 2 \leq r \leq k \), each causing in the worst case one more comparison. Hence, the worst-case number of additional comparisons equals \( \sum_{1 \leq r \leq k} \lceil (i - 1)/2^r \rceil \leq i - 2 + k \).

The worst-case number of comparisons is bounded above by \( 2^k - 1 - k + i - 2 + k = n - 2 \). It is also easy to see that the worst-case number of comparisons is not smaller than \( n - \lceil \log(n + 1) \rceil \). Only if the root of leaf-search is on the path from 1 to \( n \), we may save in the best case one comparison. This path does not contain, if \( n \neq 2^k \), more than \( \lceil \log n \rceil - 1 \) inner nodes and only inner nodes are roots of leaf-search. \( \Box \)

For the calls of leaf-search during the heap creation phase, the best-case and worst-case number of comparisons are almost identical. Hence, we do not investigate the average case in more detail.

Next we discuss the calls of bottom-up-search during the heap creation phase. Let \( b(1), \ldots, b(d) \) be the objects on the special path excluding the root, \( b(0) = -\infty \),
BOTTOM-UP-HEAPSORT, a new variant

$b(d+1):=\infty$, and $x$ the root object. If $b(j)<x<b(j+1)$, HEAPSORT uses $2(j+1)$ comparisons if $j<d$, and $2d$ comparisons if $j=d$. If $b(j)<x<b(j+1)$, bottom-up-search uses $d-j+1$ comparisons if $j>0$, and $d$ comparisons if $j=0$. Let us denote by $l_{HS}$ half the number of comparisons of HEAPSORT and by $l_{BUS}$ the number of comparisons used by bottom-up-search. We have shown that

$$l_{HS} + l_{BUS} = d + 2 \quad \text{if } 0 < j < d$$
and

$$l_{HS} + l_{BUS} = d + 1 \quad \text{if } j = 0 \text{ or } j = d.$$

In order to estimate the mean value of $l_{BUS}$ (or the sum of all $l_{BUS}$), we use the following theorem of Doberkat [4].

**Theorem 4.2.** Let $\alpha_i = \sum_{1 \leq j < \infty} (2^j - 1)^{-1}$. The heap creation phase of HEAPSORT uses, on an average, $(\alpha_1 + 2\alpha_2 - 2)n + \Theta(\log n)$ comparisons and $(\alpha_1 + \alpha_2 - 2)n + \Theta(\log n)$ interchanges. $\alpha_1 = 1.6066951 \ldots$, $\alpha_2 = 1.1373387 \ldots$

We have to perform $\lfloor n/2 \rfloor$ calls of bottom-up-search. By $L_{BUS}$, $L_{HS}$ and $D$, we denote the random variables which are the sums of all $l_{BUS}$, $l_{HS}$ and $d$, respectively. We are interested in the mean value $E(L_{BUS})$. We know from Lemma 4.1 and Theorem 4.2 that

$$E(D) = n + \Theta(\log n)$$
and

$$E(L_{HS}) = (\alpha_1/2 + \alpha_2 - 1)n + \Theta(\log n).$$

Let $T$ be the random variable counting the calls of bottom-up-search where $l_{HS} + l_{BUS} = d + 1$. Then

$$E(L_{BUS}) = E(D) + 2\lfloor n/2 \rfloor - E(T) - E(L_{HS})$$
$$= (3 - \alpha_1/2 - \alpha_2)n - E(T) + \Theta(\log n).$$

The event $j=0$ happens if the root object is the smallest object in the considered subtree. If the subtree contains $r$ objects, the probability that $j=0$ equals $1/r$, since the root object has not been investigated before and the other objects of the subtree are only permuted. Since we allow an error of $\Theta(\log n)$, we may assume that $n/4$ subtrees have 3 objects, $n/8$ subtrees have 7 objects; in general, $n/2^h$ subtrees have $2^h - 1$ objects. Hence, the expected number of situations where $j=0$ equals $\beta n + \Theta(\log n)$, where

$$\beta := \sum_{2 \leq k < \infty} [2^h(2^h - 1)]^{-1} = 0.1066952 \ldots$$

Next we consider the event $j=d$. The object $b(j)$ is called special object. Then the event $j=d$ is equal to the event that the special object has no son. Let $c$ be the number of comparisons used by the old reheap procedure, $i$ the number of interchanges and $s$ the number of sons of the special object. Then

$$c = 2i + s.$$
Again we denote by $C$, $I$ and $S$ the random variables which are the sums of all $c$, $i$ and $s$, respectively. By Theorem 4.2,

$$E(S) = E(C) - 2E(I) = (2 - \alpha \gamma) n + \Theta(\log n).$$

Only for $\lfloor \log n \rfloor$ calls of \texttt{reheap} it might happen that $s = 1$. Otherwise, $s \in \{0, 2\}$. The probability that $s = 2$ can be computed by

$$(E(S)/\lfloor n/2 \rfloor)/2 = 2 - \alpha \gamma + \Theta(n^{-1} \log n) = 0.3933049 \ldots$$

Therefore, the probability that $s = 0$ equals

$$\alpha \gamma - 1 + \Theta(n^{-1} \log n) = 0.6066951 \ldots$$

Hence, the expected number of situations where $j = d$ equals

$$(\alpha \gamma - 1 + \Theta(n^{-1} \log n)) \lfloor n/2 \rfloor = (\alpha \gamma / 2 - 1/2)n + \Theta(\log n).$$

Combining our results, we obtain

$$E(L_{\text{BUS}}) = (7/2 - \alpha \gamma - \alpha \beta - \beta)n + \Theta(\log n).$$

\textbf{Theorem 4.3.} \textsc{Bottom-up-Heapsort} uses, on an average, for the heap creation phase

$$(9/2 - \alpha \gamma - \alpha \beta)n + \Theta(\log n) \approx 1.649271n$$

comparisons. The average number of assignments equals

$$(\alpha \gamma + \alpha \beta - 1 - 2\beta)n + \Theta(\log n) \approx 1.5306434n.$$  

\textbf{Proof.} The result on the number of comparisons follows by adding the number of comparisons for \texttt{leaf-search} and \texttt{bottom-up-search}. Procedure \texttt{bottom-up-reheap} produces the same heap as \texttt{reheap}. If \texttt{reheap} uses $i > 0$ interchanges, \texttt{interchanges-1} gets by with $i + 2$ assignments. If $i = 0$, also \texttt{interchange-1} gets by without assignments. The result follows from Theorem 4.2, since $\lfloor n/2 \rfloor$ reheapings are performed. \hfill \Box

One may ask whether it is possible to improve \texttt{bottom-up-search} by climbing up the special path in a more clever way. We can explicitly compute the distribution for the number of comparisons of \texttt{bottom-up-search}. For this distribution, it is easy to prove that no search procedure can surpass \texttt{bottom-up-search}.

The investigation of the selection phase is more difficult, since (see \cite{3,4}) we do not obtain a random heap by deleting the root of a random heap and applying \texttt{reheap} or \texttt{bottom-up-reheap}.\footnote{Recently, Schaffer and Sedgewick \cite{11} have used the method of backwards analysis for \textsc{Heapsort}. From their results it is easy to conclude that the average-case number of comparisons of \textsc{Bottom-up-Heapsort} is bounded by $n \log n + O(n \log \log n)$. This result is not precise enough to decide for which $n$ \textsc{Bottom-up-Heapsort} beats \textsc{Clever Quicksort}. We prefer to carry out a more precise analysis under some realistic assumptions.}
Lemma 4.4. The worst-case number of comparisons for the calls of leaf-search during the selection phase equals, for \( n \geq 3 \),

\[
n\lfloor \log(n-2) \rfloor - 2 \cdot 2^\lfloor \log(n-2) \rfloor - \lfloor \log(n-2) \rfloor + 2.
\]

Proof. Procedure \texttt{leaf-search} is called for \((n-1,1), \ldots , (2,1)\). The worst-case number of comparisons for \texttt{leaf-search}(i,1) equals \( \lfloor \log(i-1) \rfloor \). The worst-case number for all calls of \texttt{leaf-search} equals

\[
\sum_{2 \leq i \leq n-1} \lfloor \log(i-1) \rfloor = \sum_{1 \leq i \leq n-2} \lfloor \log i \rfloor
\]

\[
= \sum_{1 \leq i \leq \lfloor \log(n-2) \rfloor} i 2^i + \lfloor \log(n-2) \rfloor \lfloor n-2^\lfloor \log(n-2) \rfloor \rfloor - 1
\]

\[
= (\lfloor \log(n-2) \rfloor - 2) 2^\lfloor \log(n-2) \rfloor + 2
\]

\[
+ \lfloor \log(n-2) \rfloor \lfloor n-2^\lfloor \log(n-2) \rfloor \rfloor - 1
\]

\[
= n \lfloor \log(n-2) \rfloor - 2 \cdot 2^\lfloor \log(n-2) \rfloor - \lfloor \log(n-2) \rfloor + 2. \quad \Box
\]

We like to express \( n \lfloor \log(n-2) \rfloor - 2 \cdot 2^\lfloor \log(n-2) \rfloor \) as \( n \log n - c(n)n \). Hence,

\[
c(n) = (\log n - \lfloor \log(n-2) \rfloor) + 2 \cdot 2^\lfloor \log(n-2) \rfloor / n.
\]

We consider the interval \( I = [2^k + 2, 2^{k+1} + 1] \), such that \( \lfloor \log(x-2) \rfloor = k \) for \( x \in I \). Let

\[
f(x) = \log x - k + 2^{k+1} x^{-1}.
\]

\[
f'(x) = (\ln^{-1} 2) x^{-1} - 2^{k+1} x^{-2} = 0 \iff x = 2^{k+1} \ln 2 = (2 \ln 2) 2^k.
\]

We see that \( c(n) \) takes its maximal value, approximately 2, if \( n \) is approximately some power of 2, and \( c(n) \) takes its minimal value if \( n \approx (2 \ln 2) 2^k \approx 1.386 \cdot 2^k \). In this case

\[
c(n) \approx k + \log(2 \ln 2) - k + 2 \cdot 2^k / ((2 \ln 2) 2^k)
\]

\[
= \log(2 \ln 2) + \ln^{-1} 2 \approx 1.91393.
\]

For most of the sorting algorithms like CLEVER QUICKSORT, the expected number of comparisons can be approximated quite well by \( cn \log n + c'n \) for some constants \( c \) and \( c' \). For BOTTOM-UP-HEAPSORT, it will turn out that \( c = 1 \) and, by the above considerations, \( c' \) has to depend on \( n \).

During the selection phase, we have good chances that special paths are short paths, i.e. \texttt{leaf-search}(i,1) performs \( \lfloor \log(i-1) \rfloor - 1 \) instead of \( \lfloor \log(i-1) \rfloor \) comparisons. The probability for a short path is 0 for \( i = 2^k \) and is almost 1 for \( i = 2^k + 1 \). What happens on an average? We have already argued why we cannot answer this question exactly.

Carlsson [2] has considered random heaps. They have the following property. If the left subtree at node \( x \) has \( i \) nodes and the right subtree has \( j \) nodes, the probability that the special path follows the left son equals \( i/(i+j) \). In the following, all results on random heaps are due to Carlsson [2]. Recently, some bugs in Carlsson’s proofs have
been found (and communicated to Carlsson). At the moment nobody knows a correct and precise analysis even for random heaps. Nevertheless, since we believe that the results of Carlsson are (at least almost) correct, we continue to compare our results with those of Carlsson.

For random heaps, we can expect only $0.279n$ short paths for $n = 2^k - 1$. This value is far from the results of experiments.

We assume w.l.o.g. that the heap contains the objects $1, \ldots, n$. Let $n = 2^k - 1 + 2^{k-1}$. The levels $0, \ldots, k - 1$ and the leftmost $2^{k-1}$ positions of level $k$ are filled. The left subtree is twice as large as the right subtree, i.e. the special path chooses the left subtree with probability $2/3$. Each call of `bottom-up-reheap` eliminates one node, at first only in the left subtree. The smallest objects are leaving the tree and former leaf objects; hence, large objects, are sinking into the heap. If object $i \leq n/3$ is not a leaf object of the left subtree, the $i$th special path chooses that subtree where object $i$ has been situated at the beginning of the selection phase. The probability that this is the left subtree equals $2/3$. But the left subtree contains at this time only $2(n-1)/3-i$ objects. If the heap is assumed to be random, the probability that the special path chooses the left subtree equals

$$\frac{2(n-1)/3-i}{n-1-i} \cdot e^{\left[ \frac{1}{2}, \frac{2}{3} \right]}.$$

These considerations lead to the following simple model. The probability that a special path chooses the left son at node $x$ does not change during the selection phase.

Let $n = 2^k - 1 + \alpha 2^k$ and $0 \leq \alpha < 1$. On level $l < k$, the leftmost $\lfloor \alpha 2^l \rfloor$ nodes form the bad region, since all paths through these nodes are long ones. Then there may be one node leading to short and long paths and the other rightmost nodes form the good region leading only to short paths. A complete binary tree of depth $D$ is approximately twice as large as a complete binary tree of depth $D - 1$. At the beginning of the selection phase, we have a random heap. Hence, the probability of reaching the bad region is $2\alpha/(1+\alpha)$, the probability for the good region is $(1-\alpha)/(1+\alpha)$.

We consider the first $\alpha 2^k$ calls of `bottom-up-reheap`. If we reach the good region, the special path is short. If we reach the bad region, the special path is not necessarily long. Perhaps, we have already eliminated enough leaves such that the path is short. On an average, half the directions in the bad region lead to short paths. The expected number of short paths equals

$$\left( \frac{1-\alpha}{1+\alpha} + \frac{2\alpha}{1+\alpha} \cdot \frac{1}{2} \right) \alpha 2^k = \frac{\alpha}{1+\alpha} 2^k.$$

For the other $2^k - 1 \approx 2^k$ deletions, the bad region is reached, on an average, by $\lfloor 2\alpha/(1+\alpha) \rfloor 2^k$ objects. If the tree contains on the last level (of depth $d$) more than $\alpha 2^d$ nodes, the special path is long. In half of the other cases, the special path is short. This
BOTTOM-UP-HEAPSORT, a new variant

leads to approximately

$$\frac{2\alpha}{1 + \alpha} \frac{1}{2} 2^k = \frac{\alpha^2}{1 + \alpha} 2^k$$

short paths. The other $$[(1 - \alpha)/(1 + \alpha)] 2^k$$ objects reach the good region. If the tree contains on the last level less than $$\alpha 2^d$$ nodes, the special path is short. In half of the other cases, the special path is short. This leads to approximately

$$\left(\frac{1 - \alpha}{1 + \alpha} 2^k\right) \left(\frac{1}{2} (1 - \alpha) + \alpha\right) = \frac{1 - \alpha^2}{1 + \alpha} \frac{1}{2} 2^k$$

short paths. Altogether, the number of short paths can be estimated by

$$\frac{1}{1 + \alpha} 2^k (\alpha + \alpha^2 + 1/2 - \alpha^2/2) = 2^{k-1} \frac{1}{1 + \alpha} (\alpha^2 + 2\alpha + 1) = (1 + \alpha) 2^{k-1}.$$  

By this simple model, we expect $$n/2$$ short paths. But we should take into account two further aspects to develop the simple model into a realistic one.

By the first aspect, we are overestimating the number of short paths. Let, e.g., $$n = 2^k$$. Typical leaves contain large objects. But some leaves contain small objects $$i < n/2$$. Such leaf objects have a good chance to climb up the heap during a call of bottom-up-reheap before they are interchanged with the root object. This probability is larger for the leftmost leaf objects, because they wait a longer time before they are interchanged with the root. If such a leaf object is interchanged with the root object, it has (by our model) a probability of $$1/2$$ of sinking into the other subtree. This leads to a weak tendency of special paths to the left.

By the second aspect, we are underestimating the number of short paths, in particular, if $$n$$ is far away from powers of 2. The simple model states that the probability for the bad region is $$2\alpha/(1 + \alpha)$$ for the whole time. By the arguments used above, large leaf objects have some chance to change their positions from right to left or vice versa before they leave the heap. If $$2^{k-1} - 1$$ objects are left in the heap, many of them have been interchanged with the root object and have sunk again into the heap. The heap looks more like a random heap rather than the one assumed by our simple model. We overestimate this aspect and assume that the heap is a random one if $$2^{k-1}$$ objects are left. Then we can expect a little less than $$2^{k-2}$$ short paths instead of the

$$\frac{1}{1 + \alpha} 2^{k-1} (\alpha^2 + 1/2 - \alpha^2/2) = \frac{1 + \alpha^2}{1 + \alpha} 2^{k-2}$$

short paths announced by the simple model. We work with $$2^{k-2}$$ short paths. The difference between the number of short paths expected by the models is

$$\left(1 - \frac{1 + \alpha^2}{1 + \alpha}\right) 2^{k-2} = \frac{\alpha - \alpha^2}{4(1 + \alpha)^2} n.$$  

The factor $$(\alpha - \alpha^2)/(4(1 + \alpha)^2)$$ takes its maximal value, namely $$1/32 \approx 0.03$$, for $$\alpha = 1/3$$. 


If we use our simple model and take into account the two aspects discussed above, we get a realistic model describing the behavior of BOTTOM-UP-HEAPSORT as it is observed in experiments quite well. The results of the experiments are very stable. The number of short paths is, for \( n \approx 2^k \), \( k \in \mathbb{N} \) and \( n \geq 2000 \), between 0.469\( n \) and 0.471\( n \). The factor of the linear term takes its maximal value, approximately 0.519, if \( n \approx 1.4 \cdot 2^k \).

We still have to investigate the calls of bottom-up-search during the selection phase. From the results of Doberkat [3] it can easily be concluded that, for \( n = 2^k - 1 \) and a random heap, the probability that bottom-up-search needs at most 2 comparisons tends to 1 as \( n \to \infty \). For random heaps the number of comparisons during all calls of bottom-up-search of the selection phase equals 1.299\( n \).

Our experiments have shown that BOTTOM-UP-HEAPSORT creates heaps that perform better than random heaps. We have sorted 20 random sequences of length \( n \in \{1000, 2000, \ldots, 50000\} \). For \( n \leq 10000 \), the probability that bottom-up-search needs only one comparison lies in the interval \([0.8469, 0.8506]\) and for \( n > 10000 \) in the interval \([0.8468, 0.8489]\). There is no tendency as to for which \( x \), where \( n = 2^k - 1 + a2^x \), this probability is larger or smaller. The probability that at most 2 comparisons are performed lies in the interval \([0.9823, 0.9832]\). Hence, the average number of comparisons is approximately between 1.169\( n \) and 1.171\( n \).

We try to explain these results. Procedure bottom-up-heapsort interchanges the rightmost leaf object \( x \) on the last level with the root object. Object \( x \) is a typical leaf object and, hence, quite large. Let \( a(0), \ldots, a(d) = x \) be the objects on the path from the root to this selected leaf. Let \( b(0) = a(0), b(1), \ldots, b(d') \), where \( d' \in \{d - 1, d\} \), be the objects on the special path. The special path chooses always the smaller son, while the other path looks for a selected direction. Hence, the paths separate, on an average, after only 2 steps. Let \( b(i) = a(i) \) and \( b(i + 1) \neq a(i + 1) \). By definition of the special path, \( b(i + 1) < a(i + 1) \). Since the direction of the \( a \)-path is independent of the size of the objects, it is very likely that \( b(d') < a(d) \) and that one comparison is sufficient for bottom-up-search. These arguments hold also for random heaps.

Why is the expected number of comparisons for BOTTOM-UP-HEAPSORT even smaller than for random heaps? Again, our model yields a good explanation of this effect. Let us consider first the case \( n = 2^k - 1 \). The objects on the last but one level are, on an average, much smaller than the objects on the last level. After some calls of bottom-up-search, the rightmost positions of the last but one level have become leaf positions. In the random heap model these positions are filled by objects of typical size for leaf objects. But in reality they are much smaller. If the special path reaches such a position, it is much more likely than in random heaps that one comparison is sufficient for bottom-up-search. Moreover, this event happens more often than in the random heap model, since the number of short paths is underestimated in the random heap model. After \( 2^{k - 1} \) calls of bottom-up-search the last but one level has become the last level. In our model it is more likely than in the random heap model that the rightmost positions have been replaced by large objects. Hence, our arguments hold again also for all other levels.
Also for the general case \( n = 2^k - 1 + \alpha 2^k \), the distribution of the directions of special paths is in our model closer to the uniform distribution than in the model of random heaps. Our arguments explain also why the probability that one comparison is sufficient for bottom-up-search does not depend essentially on \( \alpha \).

We can explain why the calls of bottom-up-search during the selection phase need much less comparisons than it could be expected by the model of random heaps. But we cannot explain why the average number of comparisons is almost exactly 1.17n. Since one comparison is necessary for each call of bottom-up-search, the decrease from 1.30n (for the model of random heaps) to 1.17n is drastic, approximately 43% of the nonnecessary comparisons are saved.

Combining our results (partly based only on realistic models), we come to the following conjecture.

**Conjecture.** Let \( d(n) \) be that number such that \( n \log n + d(n) n \) is the expected number of comparisons for BOTTOM-UP-HEAPSORT. Then \( d(n) \in [0.34, 0.39] \). Furthermore, \( d(n) \) is small if \( n \approx 2^k \), and large if \( n \approx 1.4 \cdot 2^k \).

This conjecture is well established by experiments:

<table>
<thead>
<tr>
<th>( n )</th>
<th>1000</th>
<th>2000</th>
<th>3000</th>
<th>4000</th>
<th>5000</th>
<th>6000</th>
<th>7000</th>
<th>8000</th>
<th>9000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d_{\text{exp}}(n) )</td>
<td>0.345</td>
<td>0.349</td>
<td>0.383</td>
<td>0.358</td>
<td>0.375</td>
<td>0.383</td>
<td>0.376</td>
<td>0.357</td>
<td>0.367</td>
<td>0.375</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( n )</th>
<th>11000</th>
<th>12000</th>
<th>13000</th>
<th>14000</th>
<th>15000</th>
<th>16000</th>
<th>17000</th>
<th>18000</th>
<th>19000</th>
<th>20000</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d_{\text{exp}}(n) )</td>
<td>0.383</td>
<td>0.386</td>
<td>0.383</td>
<td>0.378</td>
<td>0.371</td>
<td>0.359</td>
<td>0.360</td>
<td>0.366</td>
<td>0.372</td>
<td>0.378</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( n )</th>
<th>21000</th>
<th>22000</th>
<th>23000</th>
<th>24000</th>
<th>25000</th>
<th>26000</th>
<th>27000</th>
<th>28000</th>
<th>29000</th>
<th>30000</th>
</tr>
</thead>
<tbody>
<tr>
<td>( d_{\text{exp}}(n) )</td>
<td>0.382</td>
<td>0.386</td>
<td>0.385</td>
<td>0.386</td>
<td>0.385</td>
<td>0.384</td>
<td>0.381</td>
<td>0.378</td>
<td>0.373</td>
<td>0.371</td>
</tr>
</tbody>
</table>

Also the number of assignments is rather small. Remember the close relation between the number of comparisons and assignments. If the special path has length \( d \) and bottom-up-search performs \( j \) comparisons, the number of assignments performed by the subsequent call of interchange-1 equals \( d - j + 3 \). If the conjecture on the number of comparisons holds, the average number of assignments is bounded by \( n \log n + 2.618 n \).

We compare BOTTOM-UP-HEAPSORT (BUH) with CLEVER QUICKSORT (CQS). Both sorting algorithms work in-place and both algorithms are easy to implement. BUH is much better with respect to the worst case. It is well known that the variance of the number of comparisons of CQS is quite small. But BUH is even more stable. This follows from the arguments used in this section and from the experiments. During 20 runs, for \( n = 30000 \), the best-case number of comparisons of BUH was 457189 and the worst-case number 457394. The difference for CQS was by a factor of more than 100 larger.

The most important quantity is the average number of comparisons. The conjecture and the results of Section 2 show that BUH beats QS for \( n \geq 400 \) and CQS for \( n \geq 16000 \). Since we cannot prove all our results on BUH, we do not try to estimate more exactly the critical value \( n_0 \) such that BUH beats CQS iff \( n \geq n_0 \). For \( n = 10^5, 10^6, 10^9, 10^{20} \), we can expect that CQS needs, on an average, 2.9, 5.5, 9.9,
5. The worst-case analysis

The worst-case number of comparisons is obviously bounded by \(2n \log n + O(n)\). We can adopt the idea of Carlsson [1] in order to reduce the worst-case number of comparisons to \(n \log n + O(n \log \log n)\). For this, procedure bottom-up-search is changed in the following way. We climb up the special path only for, say, 10 steps and perform, if the search for the special object is not successful, a binary search on the special path. This change of the algorithm is in conflict with the philosophy of BOTTOM-UP-HEAPSORT, since bottom-up-search is shown to be optimal on an average. Hence, we come back to BOTTOM-UP-HEAPSORT.

**Theorem 5.1.** **BOTTOM-UP-HEAPSORT** needs not more than \(1.5n \log n + O(n)\) comparisons.

**Proof.** By Lemma 4.1, the number of comparisons during the heap creation phase is bounded by \(2n\). By Lemma 4.4, the number of comparisons during the calls of leaf-search of the selection phase is bounded by \(n \log n - c(n)n\). We still have to investigate the calls of bottom-up-search during the selection phase.

Let \(n = 2^k - 1 + a 2^k\) for some \(a \in [0, 1)\). The levels \(0, \ldots, k - 1\) are complete and level \(k\) contains \(a 2^k\) nodes. The number of leaves equals \(\lceil n/2 \rceil\). We assume w.l.o.g. that the heap contains the objects \(1, \ldots, n\). The objects \(1, \ldots, \lceil n/2 \rceil\) are called small, the other objects are called large. We investigate the first \(\lfloor n/2 \rfloor\) calls of bottom-up-search which eliminate the leaf positions. Afterwards, the heap contains exactly all large objects.

The heap properties ensure that all predecessors of small objects are small and all successors of large objects are large. Hence, at most \(\lfloor n/4 \rfloor\) leaf objects are small. This implies also that, for at most \(\lfloor n/4 \rfloor\) calls of bottom-up-search, the root object is a small one. For \(\lceil n/4 \rceil\) calls of bottom-up-search, among them all calls with small root objects, we estimate the number of comparisons for bottom-up-search by the current depth \(d(t)\) of the heap.

Let us now consider the \(m\)th \((1 \leq m \leq \lceil n/2 \rceil - \lfloor n/4 \rfloor)\) of the other calls of bottom-up-search. If the new position of the root object is on level \(d_m\), the number of comparisons equals \(d(t) - d_m + 1\). Since the large objects are climbing up only during the first \(\lceil n/2 \rceil\) calls of bottom-up-search, the sum of all \(d_m\) is not smaller than the minimal sum of the depth of \(\lceil n/2 \rceil - \lfloor n/4 \rfloor\) nodes in a binary tree. In this case the levels \(0, \ldots, k - 3\) are filled and \(\lfloor a 2^{k-2} \rfloor\) positions on level \(k - 2\) are filled. The sum of the depths equals

\[
\sum_{0 \leq i \leq k - 3} i 2^i + \lfloor a 2^{k-2} \rfloor (k - 2) = (k - 4) 2^{k-2} + 2 + \lfloor a 2^{k-2} \rfloor (k - 2) \geq (k - 4)n/4 + 2a 2^{k-2} - k + 4.
\]
The sum of all \([n/2]\) considered values of \(d(t)\) equals
\[a2^k + ([n/2] - a2^k)(k - 1) = [n/2]k - ([n/2] - a2^k).\]

Finally, we get the term \([n/2] - [n/4]\) from the term 1 in \(d(t) - d_m + 1\). The number of comparisons during the \([n/2]\) calls of \textit{bottom-up-search} is bounded by
\[[n/2]k - ([n/2] - a2^k) + [n/2] - [n/4] - (k - 4)n/4 - 2a2^k - 2 + k - 4\]
\[\leq ([n/2] - n/4)k + 3n/4 + a2^k - 1 + k.\]

The same estimations hold for the elimination of the positions which are now leaves and so on. Altogether, the number of comparisons during the calls of \textit{bottom-up-search} of the selection phase is bounded by
\[0.5n\log n - 0.25n + 1.5n + a2^k + k^2\]
\[\leq 0.5n\log n + 2.25n + O(\log^2 n).\]

The worst-case number of comparisons of BOTTOM-UP-HEAPSORT is bounded by
\[1.5n\log n + 2.25n + O(\log^2 n).\]

By more careful estimations, it is possible to decrease the linear term a little bit. But the leading term is more important. Our estimations for the number of comparisons of those calls of \textit{bottom-up-search} which include the small leaf objects seem to be very inexact. But they are at least almost optimal. Fleischer [5] and Schaffer and Sedgewick [11] have, independently, given examples where the number of comparisons of BOTTOM-UP-HEAPSORT equals \(1.5n\log n - o(n\log n)\).

6. Conclusion

BOTTOM-UP-HEAPSORT is the first in-place sequential sorting algorithm which is efficient with respect to the average number of comparisons and other operations and with respect to the worst-case number of comparisons and other operations. BOTTOM-UP-HEAPSORT is easy to implement and practically efficient. Since CLEVER QUICKSORT is indeed clever, the profit gained by BOTTOM-UP-HEAPSORT is not too large.

If we are able (as we usually are) to spend an extra storage for \(n\) bits, we may use a HEAPSORT variant due to McDiarmid and Reed [10]. Wegener [16] has proved that its worst-case number of comparisons is bounded by \(n\log n + n\) (if \(n = 2^k - 1\)) or \(n\log n + (3 - c(n))n\) in the general case. The average-case complexity is approximately by \(0.3n\) comparisons smaller than for BOTTOM-UP-HEAPSORT, but we have to pay for these savings by approximately \(n\log n\) bit tests. Hence, for practical purposes, BOTTOM-UP-HEAPSORT is superior to this variant.

Heaps build also an important dynamic data structure supporting insertions and deletions of the actual minimal object. Procedure \textit{bottom-up-reheap} improves, on an average, procedure \textit{reheap} for all \(n\).
Every computer science student has to study sequential sorting algorithms. Up to now, HEAPSORT has been taught because of its worst-case behavior and CLEVER QUICKSORT is taught because of its average-case behavior and its practical efficiency. BOTTOM-UP-HEAPSORT combines the advantages of both algorithms. In view of the results of this paper, one should teach in future BOTTOM-UP-HEAPSORT instead of HEAPSORT.

References