Theory Comput Syst (2016) 59:209–230
DOI 10.1007/s00224-015-9656-y

QuickHeapsort: Modifications and Improved Analysis

Volker Diekert[1] · **Armin Weiß**[1]

Published online: 15 September 2015

Abstract QuickHeapsort is a combination of Quicksort and Heapsort. We show that the expected number of comparisons for QuickHeapsort is always better than for Quicksort if a usual median-of-constant strategy is used for choosing pivot elements. In order to obtain the result we present a new analysis for QuickHeapsort splitting it into the analysis of the partition-phases and the analysis of the heap-phases. This enables us to consider samples of non-constant size for the pivot selection and leads to better theoretical bounds for the algorithm. Furthermore, we introduce some modifications of QuickHeapsort. We show that for every input the expected number of comparisons is at most $n \log_2 n - 0.03n + o(n)$ for the in-place variant. If we allow n extra bits, then we can lower the bound to $n \log_2 n - 0.997n + o(n)$. Thus, spending n extra bits we can save more that $0.96n$ comparisons if n is large enough. Both estimates improve the previously known results. Moreover, our non-in-place variant does essentially use the same number of comparisons as index based Heapsort variants and Relaxed-Weak-Heapsort which use $n \log_2 n - 0.9n + o(n)$ comparisons in the worst case. However, index based Heapsort variants and Relaxed-Weak-Heapsort require $\Theta(n \log n)$ extra bits whereas we need n bits only. Our theoretical results are upper bounds and valid for every input. Our computer experiments show that the gap between our bounds and the actual values on random inputs is small. Moreover, the computer experiments establish QuickHeapsort as competitive with Quicksort in terms of running time.

Keywords In-place sorting · Heapsort · Quicksort · Analysis of algorithms

✉ Armin Weiß
armin.weiss@fmi.uni-stuttgart.de

Volker Diekert
diekert@fmi.uni-stuttgart.de

[1] FMI, Universität Stuttgart, Universitätsstr. 38, D-70569 Stuttgart, Germany

1 Introduction

QuickHeapsort is a combination of Quicksort and Heapsort which was first described by Cantone and Cincotti [2]. It is based on Katajainen's idea for Ultimate Heapsort [11]. Ultimate Heapsort needs $n \log_2 n + \mathcal{O}(n)$ comparisons in the worst case, but, due to exact median computations, the hidden constant in the linear term is very high. In contrast to Ultimate Heapsort, the worst case behavior of QuickHeapsort is quadratic, but it performs well in practice: just as Quicksort does and for the very same reason. QuickHeapsort always beats Quicksort with respect to the number of comparisons if the same strategy for pivot selection is applied. For example, consider the median-of-three method for choosing pivot elements. This pivot selection is also called *Clever Quicksort*, see e.g. [17]. Clever Quicksort needs $1.18n \log_2 n$ comparisons on average – whereas the expected number of comparisons is only $n \log_2 n + 1.92n$ for QuickHeapsort. The strength of QuickHeapsort lies in the optimal $n \log_2 n$ term and the small constant in the linear term. Moreover, we show that the linear term can be made smaller by applying some improved variants of QuickHeapsort. Using n extra bits and choosing the pivot element as the median out of $\sqrt{n}$ random elements, we obtain a variant where the expected number of comparisons is less than $n \log_2 n - 0.997n$, see Theorem 4.

Quicksort and QuickHeapsort have in common that the array is partitioned into two parts by some pivot element. Quicksort is called recursively for both parts whereas QuickHeapsort sorts the smaller part differently and calls itself recursively for the larger part, only. In the smaller part a heap is constructed and the elements are successively extracted. The crucial observation is that for the sift-down only one comparison per level is needed. This is the point where QuickHeapsort is superior to standard Heapsort which uses two comparisons per level during the sift-down phase, see e.g. the textbook [6, 12]. This is one of the reasons why standard Heapsort cannot compete with Quicksort in practice (of course there are also other reasons like cache behavior). Over the time a lot of solutions to this problem appeared like Bottom-Up-Heapsort [18] or MDR-Heapsort [14, 17], which both perform the sift-down by first going down to some leaf and then searching upward for the correct position.

In [2] the basic version of QuickHeapsort with a fixed position in the array as pivot is analyzed, but no other method for pivot selection is considered. However, the authors compare an implementation of the median-of-three version with other Quick- and Heapsort variants, too. In [8] Edelkamp and Stiegeler compare these variants with so-called Weak-Heapsort [7] and some modifications of it like Relaxed-Weak-Heapsort [8]. Weak-Heapsort needs $n \log_2 n - 0.42n$ comparisons in the average case and $n \log_2 n + 0.1n$ in the worst case; hence, it beats basic QuickHeapsort with respect to the number of comparisons ($n \log_2 n + 0.72n + o(n)$ expected number of comparisons when implemented with $\sqrt{n}$ elements for pivot selection). However, Weak-Heapsort needs $\mathcal{O}(n)$ bits extra-space. Wang and Wu introduced Rank-Heapsort [16]. Rank-Heapsort as well as Relaxed-Weak-Heapsort needs $n \log_2 n - 0.91n$ comparisons in the worst case, but $\mathcal{O}(n \log n)$ bits extra-space. The in-place algorithm Bottom-Up-Heapsort is conjectured to use at most

$n \log_2 n + 0.4n$ on average [18]. One should also note that Ultimate Heapsort can be viewed as a special case of QuickHeapsort where the sample to choose the median is the whole array, thus Ultimate Heapsort determines the exact median position. This bounds the worst-case, but it is too expensive in practice where the average case becomes more interesting. There is also a complicated, iterated in-place MergeInsertion due to Reinhardt [15] which uses only $n \log_2 n - 1.3n + \mathcal{O}(\log n)$ comparisons, but the algorithm is not suitable in practice.

The organization of the present paper is as follows. QuickHeapsort together with our first improvement is described in Section 2. In Section 4 and Section 5, we analyze the expected number of comparisons of QuickHeapsort. Then we introduce some improvements in Section 6 allowing n additional bits. Finally, in Section 7, we present our experimental results comparing the different versions of QuickHeapsort with other Quicksort and Heapsort variants. In our experiments we focus on the number of comparisons but our tables also include some running times in order to show the practical performances. Our paper yields the following results.

1. We give a simplified analysis for the expected number of comparisons which gives better bounds than previously known for QuickHeapsort. This is done by splitting the analysis into three parts: the partitioning phases, the heap construction and the heap extraction. Our approach yields the first precise analysis of QuickHeapsort when the pivot element is taken from a larger sample. In particular, we analyze the median-of-three situation or the even more interesting situation where the sample has size $\sqrt{n}$.
2. We give a simple in-place modification of QuickHeapsort which saves $0.75n$ comparisons.
3. We give a modification of QuickHeapsort using n extra bits only and we can bound the expected number of comparisons to $n \log_2 n - 0.997n + o(n)$. This bound is better than all previously known bounds for the worst case of Heapsort variants using $\mathcal{O}(n \log n)$ extra bits.
4. We have implemented QuickHeapsort and we have performed various computer experiments counting comparisons and measuring running times. The experiments confirm that our theoretical upper bounds on the number match realistic values. For the running time we measured the cost of a comparison using two different functions simulating cheap and expensive comparisons. The experiments show that using extra bits is the better method when comparisons are expensive.

2 QuickHeapsort

A *two-layer-min-heap* is an array $A[1..n]$ of n elements together with a partition (G, R) of $\{1, \ldots, n\}$ into *green* and *red* elements such that for all $g \in G, r \in R$ we have $A[g] \leq A[r]$. Furthermore, the green elements g satisfy the heap condition $A[g] \leq \min\{A[2g], A[2g+1]\}$, and if g is red, then $2g$ and $2g+1$ are red, too.

(The conditions are required to hold, only if the indices involved are in the range of 1 to n.) The green elements are called "green" because they can be extracted out of the heap without caution, whereas the "red" elements are blocked. *Two-layer-max-heaps* are defined analogously. We can think of a two-layer-heap as rooted binary tree such that each node is either green or red. Green nodes satisfy the standard heap-condition, children of red nodes are red. Two-layer-heaps were defined in [11]. In [2] for the same concept a different language is used (they describe the algorithm in terms of External Heapsort). Now, we are ready to describe the QuickHeapsort algorithm as it has been proposed in [2]. Most of it can be found in pseudocode in Section 3.

QuickHeapsort sorts an array $A[1..n]$ as follows. A pivot element $p \in \{A[1], \ldots, A[n]\}$ is selected as median of some random sample. This is the randomized part of the algorithm. Once p is chosen, the array is rearranged according to p as in Quicksort. That means, using $n-1$ comparisons the partitioning function returns an index k and rearranges the array A so that $A[i] \geq A[k]$ for $i < k$, $A[k] = p$, and $A[k] \geq A[j]$ for $k < j$. After the partitioning a two-layer-heap is built out of the elements of the smaller part of the array, either the part left of the pivot or right of the pivot. We call this smaller part *heap-area* and the larger part *work-area*. More precisely, if $k-1 < n-k$, then $\{1, \ldots, k-1\}$ is the heap-area and $\{k+1, \ldots, n\}$ is the work-area. If $k-1 \geq n-k$, then $\{1, \ldots, k-1\}$ is the work-area and $\{k+1, \ldots, n\}$ is the heap-area. Note that we know the final position of the pivot element without any further comparison. Therefore, we do not count it to the heap-area nor to the work-area. If the heap-area is the part of the array left of the pivot, a two-layer-max-heap is built, otherwise a two-layer-min-heap is built.

At the beginning, the heap-area is an ordinary heap; hence, it is a two-layer-heap consisting of green elements, only. Now, the heap extraction phase starts. We assume that we are in the case of a max-heap. The other case is symmetric. Let m denote the size of the heap-area. The m elements of the heap-area are moved to the work-area. The extraction of one element works as follows: the root of the heap is placed at the current position of the work-area (which at the beginning is its last position). Then, starting from the root the resulting "hole" is trickled down: always the larger child is moved up into the vacant position and then this child is treated recursively. This stops as soon as a leaf is reached. We call this the SpecialLeaf procedure (Algorithm 2) according to [2]. Now, the element which previously was at the current position in the work-area is placed as red element in this hole at the leaf in the heap-area. Finally, the current position in the work-area is moved by one and the next element can be extracted.

After the partitioning it is guaranteed that no red element is greater than any green element because we assumed to be in the situation of a max-heap. Moreover, when we start heap extraction, we know in addition that $A[g] \geq \max\{A[2g], A[2g+1]\}$ whenever g is green. Hence, the procedure sorts correctly. Furthermore, there is enough space in the work-area to place all green elements of the heap since the heap is always the smaller part of the array. After extracting all green elements the pivot element

it placed at its final position and the remaining elements are sorted recursively. The pseudo code of the algorithm is in Section 3.

Actually we can improve the procedure, thereby saving $3n/4$ comparisons by a simple trick. Before the heap extraction phase starts in the heap-area with m elements, we perform at most $\frac{m+2}{4}$ additional comparisons in order to arrange all pairs of leaves which share a parent such that the left child is not smaller than its right sibling. Now, in every call of SpecialLeaf, we can save exactly one comparison since we do not need to compare two leaves. For a max-heap we only need to move up the left child and put the right one at the place of the former left one. Summing up over all heaps during an execution of standard QuickHeapsort, we invest $\frac{n+2t}{4}$ comparisons in order to save n comparisons, where t is the number of recursive calls. The expected number of t is in $\mathcal{O}(\log n)$. Hence, we can expect to save $\frac{3n}{4} + \mathcal{O}(\log n)$ comparisons. We call this version the *improved* variant of QuickHeapsort.

3 Pseudocode

Algorithm 1 is the main procedure. It uses the following subroutines.

- ChoosePivot: It returns an element p of the array.
- PartitionReverse: It returns an index k and rearranges the array A so that $p = A[k]$, $A[i] \geq A[k]$ for $i < k$ and $A[i] \leq A[k]$ for $i > k$ using $n-1$ comparisons.
- ConstructMaxHeap: Constructs a max-heap on the input array.

Algorithm 1

```
procedure QuickHeapsort(A[1..n])
begin
  if n > 1 then
    p := ChoosePivot;
    k := PartitionReverse(A[1..n], p);
    if k ≤ n/2 then
      TwoLayerMaxHeap(A[1..n], k − 1);          (* heap-area: {1..k − 1} *)
      swap(A[k], A[n − k + 1]);
      QuickHeapsort(A[1..n − k]);
    else
      TwoLayerMinHeap(A[1..n], n − k);          (* heap-area: {k + 1..n} *)
      swap(A[k], A[n − k + 1]);
      QuickHeapsort(A[(n − k + 2)..n]);
    endif
  endif
endprocedure
```

Algorithm 2

```
function SpecialLeaf(A[1..m]):
begin
  i := 1;
  while 2i ≤ m do                                   (* while i is not a leaf *)
    if 2i + 1 ≤ m and A[2i + 1] > A[2i] then
      A[i] := A[2i + 1];
      i := 2i + 1;
    else
      A[i] := A[2i];
      i := 2i;
    endif
  endwhile
  return i;
endfunction
```

Algorithm 3

```
procedure TwoLayerMaxHeap(A[1..n], m)
begin
  ConstructMaxHeap(A[1..m]);
  for i := 1 to m do
    temp := A[n − i + 1];
    A[n − i + 1] := A[1];
    j :=SpecialLeaf(A[1..m]);
    A[j] := temp;
  endfor
endprocedure
```

The procedure TwoLayerMinHeap is symmetric to TwoLayerMaxHeap.

4 Analysis of QuickHeapsort

This section contains the main contribution of the paper. We analyze the number of comparisons. By n we denote the number of elements of an array to be sorted. We use standard $\mathcal{O}$-notation where $\mathcal{O}(g)$, $\Theta(g)$, $o(g)$, and $\omega(g)$ denote classes of functions. Throughout, the logarithm is to base 2 and it is denoted by lg, thus, $\lg n = \log_2 n$. In our analysis we do not assume any probability distribution of the input, i.e., our bound for the expected number of comparisons is valid for every permutation of the input array. Randomization is used however for pivot selection.

4.1 Comparison Between Quicksort and QuickHeapsort

Let us start with an informal discussion which explains why QuickHeapsort is able to beat Quicksort with respect to the number of comparisons. We assume a median-

of-constant pivot selection. In Quicksort every position in the array is chosen exactly once as pivot. Thus, the random process yields a permutation of the set $\{1, \ldots, n\}$. More precisely, it yields a binary search tree where the root r is the position of the first pivot, the root of its left subtree is the pivot chosen in the array $A[1..r-1]$, etc. QuickHeapsort deviates from Quicksort in such a way that the recursion is only used for the larger subtree. This means that we expect only $\mathcal{O}(\log n)$ pivot elements to be chosen rather then n for Quicksort. Therefore, costs for finding the pivot elements sum up to $o(n)$ in QuickHeapsort, only (also when the pivot is selected from a sample of growing size); and it has no effect on the linear term. This is true, as long as pivot elements are chosen from a sample of size $o(n)$, e.g. $\sqrt{n}$ elements.

Quicksort and QuickHeapsort both call a recursion for the larger subtree, but for the smaller subtree QuickHeapsort creates a heap and then performs the sorting using heap extraction. Assume that the smaller subtree has m elements. Quicksort with a median-of-constant pivot selection is expected to make at least $c \cdot m \lg m$ comparisons with $c > 1$ and with $c \approx 1.18$ for the median-of-three method. But QuickHeapsort uses here $m \lg m + \mathcal{O}(m)$ comparisons, only. Indeed, the standard procedure for heap construction needs $\mathcal{O}(m)$ comparisons. Now the heap extraction amounts to $m \lg m$ comparisons, because whenever an element is extracted from the heap, a new element is inserted and sifts down to the bottom. The sift down needs one comparison per level because the newly inserted element is always smaller (resp. greater) than any element in the heap.

4.2 The Main Result

With $\Pr[e]$ we denote the probability of some event e. The expected value of a random variable T is denoted by $\mathbb{E}[T]$. Let $T(n)$ denote the number of comparisons during QuickHeapsort on a fixed array of n elements. We do not count assignments or "swaps" since their number is of the same magnitude. We split the analysis of QuickHeapsort into three parts:

1. Partitioning with an expected number of comparisons $\mathbb{E}[T_{\text{part}}(n)]$.
2. Heap construction with at most $T_{\text{cstr}}(n)$ comparisons (worst case).
3. Heap extraction or "sorting phase" with at most $T_{\text{ext}}(n)$ comparisons (worst case).

We analyze the three parts separately and put them together at the end. The partitioning is the only randomized part of our algorithm. The expected number of comparisons depends on the selection method for the pivot. Throughout we use the fact that the median of k elements can be determined with $\mathcal{O}(k)$ comparisons, see for example [1]. The expected number of comparisons by QuickHeapsort on an input array of size n is given by the following inequality

$$\mathbb{E}[T(n)] \leq T_{\text{cstr}}(n) + T_{\text{ext}}(n) + \mathbb{E}[T_{\text{part}}(n)].$$

Note that the Heapsort part of the algorithm is not randomized. Therefore we bound the worst case. An alternative approach would be to average $T_{\text{cstr}}(n)$ and $T_{\text{ext}}(n)$ over all possible input permutations. However, this alternative would not bound the expected number of comparisons for one fixed input.

Theorem 1 *The expected number $\mathbb{E}[T(n)]$ of comparisons by basic (resp. improved) QuickHeapsort with pivot as median of k randomly selected elements on an input array of size n satisfies $\mathbb{E}[T(n)] \leq n \lg n + c_k n + o(n)$ with c_k as follows:*

k	c_k *basic QHS variant*	c_k *improved QHS variant*
1	$+2.72$	$+1.97$
3	$+1.92$	$+1.17$
$f(n)$	$+0.72$	-0.03

The last row is valid for all $f \in \omega(1) \cap o(n)$ with $1 \leq f(n) \leq n$, e.g., $f(n) = \sqrt{n}$.

The proof of Theorem 1 is given in Section 5.

5 Proof of Theorem 1

Note that it is enough to prove the results without the improvement since the difference is always $0.75n$ as explained above. The table shows that the selection method for the pivot is relevant although the bounds for fixed size samples for pivot selection are not tight.

5.1 Heap Construction

The standard heap construction [9] needs at most $2m$ comparisons to construct a heap of size m in the worst case and approximately $1.88m$ in the average case. For the mathematical analysis better theoretical bounds can be used. The best result we are aware of is due to Chen et al. in [5]. According to this result we have $T_{\text{cstr}}(m) \leq 1.625m + o(m)$. Earlier results are of similar magnitude, by [4] it has been known that $T_{\text{cstr}}(m) \leq 1.632m + o(m)$ and by [10] it has been known $T_{\text{cstr}}(m) \leq 1.625m + o(m)$, but Gonnet and Munro used $\mathcal{O}(m)$ extra bits to get this result, whereas the new result of Chen et al. is in-place (by using only $\mathcal{O}(\lg m)$ extra bits). During the execution of QuickHeapsort over n elements, every element is part of a heap only once. Hence, the sizes of all heaps during the entire procedure sum up to n. With the result of [5] the total number of comparisons performed in the construction of all heaps satisfies:

Proposition 1 $T_{\text{cstr}}(n) \leq 1.625n + o(n)$.

5.2 Heap Extraction

For a real number $r \in \mathbb{R}$ with $r > 0$ we define $\{r\}$ by the following condition

$$r = 2^k + \{r\} \text{ with } k \in \mathbb{Z} \text{ and } 0 \leq \{r\} < 2^k.$$

This means that 2^k is largest power of 2 which is less than or equal to r and $\{r\}$ is the difference to that power, i.e., $\{r\} = r - 2^{\lfloor \lg r \rfloor}$. In this section we first analyze the extraction phase of one two-layer-heap of size m. After that, we bound the number of comparisons $T_{\text{ext}}(n)$ performed in the worst case during all heap extraction phases

of one execution of QuickHeapsort on an array of size n. Theorem 2 is our central result about heap extraction.

Theorem 2 $T_{\text{ext}}(n) \leq n \cdot (\lfloor \lg n \rfloor - 3) + 2\{n\} + \mathcal{O}(\lg^2 n)$.

The proof of Theorem 2 covers almost the rest of Section 5.2. In the following, the *height* height(υ) of an element υ in a heap H is the maximal distance from that node to a leaf below it. The *height* of H is the height of its root. The *level* level(υ) of υ is its distance from the root. In this section we want to count the comparisons during SpecialLeaf procedures, only. Recall that a SpecialLeaf procedure is a cyclic shift on a path from the root down to some leaf, and the number of comparisons is exactly the length of this path. Hence, an upper bound is the height of the heap. But there is a better analysis.

Let us consider a heap with m green elements which are all extracted by SpecialLeaf procedures. The picture is as follows: First, we color the green root red. Next, we perform a cyclic shift defined by the SpecialLeaf procedure. In particular, the leaf is now red. Moreover, red positions remain red, but there is exactly one position υ which has changed its color from green to red. This position υ is on the path defined by the SpecialLeaf procedure. Hence, the number of comparisons needed to color the position υ red is bounded by height(υ) + level(υ).

The total number of comparisons $E(m)$ to extract all m elements of a Heap H is therefore bounded by

$$E(m) \leq \sum_{\upsilon \in H} (\text{height}(\upsilon) + \text{level}(\upsilon)).$$

We have height(H) $-$ 1 $\leq$ height(υ) + level(υ) $\leq$ height(H) $= \lfloor \lg m \rfloor$ for all $\upsilon \in H$. We now count the number of elements υ where height(υ)+level(υ) $= \lfloor \lg m \rfloor$ and the number of elements υ where height(υ) + level(υ) $= \lfloor \lg m \rfloor - 1$. Since there are exactly $\{m\}+1$ nodes of level $\lfloor \lg m \rfloor$, there are at most $2\{m\}+1+\lg m$ elements υ with height(υ)+level(υ) $= \lfloor \lg m \rfloor$. All other elements satisfy height(υ)+level(υ) $= \lfloor \lg m \rfloor - 1$. We obtain

$$\begin{aligned} E(m) &\leq 2 \cdot \{m\} \cdot \lfloor \lg m \rfloor + (m - 2 \cdot \{m\})(\lfloor \lg m \rfloor - 1) + \mathcal{O}(\lg m) \\ &= m \cdot (\lfloor \lg m \rfloor - 1) + 2 \cdot \{m\} + \mathcal{O}(\lg m). \end{aligned} \tag{1}$$

Note that this is an estimate of the worst case, however this analysis also shows that the best case only differs by $\mathcal{O}(\lg m)$-terms from the worst case.

Now, we want to estimate the number of comparisons in the worst case performed during all heap extraction phases together. During QuickHeapsort over n elements we create a sequence $H_1, \ldots, H_t$ of heaps of green elements which are extracted using the SpecialLeaf procedure. Let $m_i = |H_i|$ be the size of the i-th Heap. The sequence satisfies $2m_i \leq n - \sum_{j<i} m_j$ because heaps are constructed and extracted on the smaller part of the array.

Here comes a subtle observation: Assume that $m_1 + m_2 \leq n/2$. If we replace the first two heaps with one heap H' of size $|H|' = m_1 + m_2$, then the analysis using the sequence $H', H_3, \ldots, H_t$ cannot lead to a better bound. Continuing this way, we

may assume that we have $t \in \mathcal{O}(\lg n)$ and therefore $\sum_{1\le i\le t} \mathcal{O}(\lg m_i) \subseteq \mathcal{O}(\lg^2 n)$. With (1) we obtain the bound

$$T_{\text{ext}}(n) \le \sum_{i=1}^{t} E(m_i) = \left(\sum_{i=1}^{t} (m_i \cdot \lfloor \lg m_i \rfloor + 2\{m_i\})\right) - n + \mathcal{O}(\lg^2 n). \quad (2)$$

Later we will replace the m_i by other positive real numbers. Therefore, we define the following notion. Let $1 \le \nu \in \mathbb{R}$. We say a sequence $x_1, x_2, \ldots, x_t$ with $x_i \in \mathbb{R}^{>0}$ is *valid* w.r.t. ν if for all $1 \le i \le t$ we have $2x_i \le \nu - \sum_{j<i} x_j$.

As just mentioned the initial sequence $m_1, m_2 \ldots, m_t$ is valid w.r.t. n. Let us define a continuous function $F : \mathbb{R}^{>0} \to \mathbb{R}$ by $F(x) = x \cdot \lfloor \lg x \rfloor + 2\{x\}$. It is continuous since for $x = 2^k, k \in \mathbb{Z}$ we have $F(x) = xk = \lim_{\varepsilon\to 0}(x-\varepsilon)(k-1)+2\{x-\varepsilon\}$. It is piecewise differentiable with right derivative $\lfloor \lg x \rfloor + 2$. Therefore:

Lemma 1 *Let $x \ge y > \delta \ge 0$. Then we have the inequalities:*

$$F(x) + F(y) \le F(x+\delta) + F(y-\delta) \text{ and } F(x) + F(y) \le F(x+y).$$

Proof Since the right derivative is monotonically increasing we have:

$$F(x+\delta) - F(x) = \int_x^{x+\delta} F'(t)\,\mathrm{d}t \ge F'(x) \cdot \delta = (\lfloor \lg x \rfloor + 2)\delta$$

and

$$F(y) - F(y-\delta) = \int_{y-\delta}^{y} F'(t)\,\mathrm{d}t \le F'(y) \cdot \delta = (\lfloor \lg y \rfloor + 2)\delta.$$

This yields:

$$F(y) - F(y-\delta) \le (\lfloor \lg y \rfloor + 2)\delta \le (\lfloor \lg x \rfloor + 2)\delta \le F(x+\delta) - F(x).$$

By adding $F(x) + F(y-\delta)$ on both sides we obtain the first claim of Lemma 1. Note that $\lim_{\varepsilon\to 0} F(\varepsilon) = 0$. Hence, the second claim follows from the first by considering the limit $\delta \to y$. □

Lemma 2 *Let $1 \le \nu \in \mathbb{R}$. For all sequences $x_1, x_2, \ldots, x_t$ with $x_i \in \mathbb{R}^{>0}$, which are valid w.r.t. ν, we have*

$$\sum_{i=1}^{t} F(x_i) \le \sum_{i=1}^{\lfloor \lg \nu \rfloor} F\left(\frac{\nu}{2^i}\right).$$

Proof The result is true for $\nu \le 2$ because then $F(x_i) \le F(\nu/2) \le F(1) = 0$ for all i. Thus, we may assume $\nu \ge 2$. We perform induction on t. For $t = 1$ the statement is clear since $\lg \nu \ge 1$ and $x_1 \le \nu/2$. Now, let $t > 1$. By Lemma 1, we have $F(x_1) + F(x_2) < F(x_1 + x_2)$. Now, if $x_1 + x_2 \le \frac{\nu}{2}$, then the sequence $x_1+x_2, x_3, \ldots, x_t$ is valid, too; and we are done by induction. Hence, we may assume $x_1 + x_2 > \frac{\nu}{2}$. If $x_1 \le x_2$, then

$$2x_1 = 2x_2 + 2(x_1 - x_2) \le \nu - x_1 + 2(x_1 - x_2) = \nu - x_2 + x_1 - x_2 \le \nu - x_2.$$

Thus, if $x_1 \le x_2$, then the sequence $x_2, x_1, x_3, \ldots, x_t$ is valid, too. Thus, it is enough to consider $x_1 \ge x_2$ with $x_1 + x_2 > \frac{\nu}{2}$.

We have $\frac{\nu}{2} \ge 1$ and the sequence $x_2', x_3, \ldots x_t$ with $x_2' = x_1 + x_2 - \frac{\nu}{2}$ is valid w.r.t. $\nu/2$ because

$$x_2' = x_1 + x_2 - \frac{\nu}{2} \le x_1 + \frac{\nu - x_1}{2} - \frac{\nu}{2} = \frac{x_1}{2} \le \frac{\nu}{4}.$$

Therefore, by induction on t and Lemma 1 we obtain the claim:

$$\begin{aligned}\sum_{i=1}^{t} F(x_i) &\le F(\nu/2) + F(x_2') + \sum_{i=3}^{t} F(x_i)\\ &\le F(\nu/2) + \sum_{i=2}^{\lfloor \lg \nu \rfloor} F\left(\frac{\nu}{2^i}\right) \le \sum_{i=1}^{\lfloor \lg \nu \rfloor} F\left(\frac{\nu}{2^i}\right).\end{aligned}$$

□

Lemma 3 $\sum_{i=1}^{\lfloor \lg n \rfloor} F\left(\frac{n}{2^i}\right) \le F(n) - 2n + \mathcal{O}(\lg n)$.

Proof

$$\begin{aligned}\sum_{i=1}^{\lfloor \lg n \rfloor} F\left(\frac{n}{2^i}\right) &= n\lfloor \lg n \rfloor \cdot \sum_{i=1}^{\lfloor \lg n \rfloor} \frac{1}{2^i} - n \cdot \sum_{i=1}^{\lfloor \lg n \rfloor} \frac{i}{2^i} + 2\{n\} \cdot \sum_{i=1}^{\lfloor \lg n \rfloor} \frac{1}{2^i}\\ &\le n\lfloor \lg n \rfloor \cdot \sum_{i \ge 1} \frac{1}{2^i} - n \cdot \sum_{i \ge 1} \frac{i}{2^i} + 2\{n\} \cdot \sum_{i \ge 1} \frac{1}{2^i} + \frac{n}{2^{\lfloor \lg n \rfloor}} \cdot \sum_{i > 0} \frac{i + \lfloor \lg n \rfloor}{2^i}\\ &= n\lfloor \lg n \rfloor - 2n + 2\{n\} + \mathcal{O}(\lg n).\end{aligned}$$

□

Applying these lemmata to (2) yields the proof of Theorem 2.

Corollary 1 *We have* $T_{\text{ext}}(n) \le n \lg n - 2.9139n + \mathcal{O}(\lg^2 n)$.

Proof By [17, Thm. 1] we have $F(n) - 2n \le n \lg n - 1.9139n$. Hence, Corollary 1 follows directly from Theorem 2. □

5.3 Partitioning

In the following $T_{\text{pivot}}(n)$ denotes the number of comparisons required to choose the pivot element in the worst case; and, as before, $\mathbb{E}[T_{\text{part}}(n)]$ denotes the expected

number of comparisons performed during partitioning. We have the following recurrence:

$$\mathbb{E}[T_{\text{part}}(n)] \leq n - 1 + T_{\text{pivot}}(n) + \sum_{k=1}^{n} \Pr[\text{pivot} = k] \cdot \mathbb{E}[T_{\text{part}}(\max\{k-1, n-k\})]. \quad (3)$$

If we choose the pivot at random, then we obtain by standard methods, see for example [6, Section 9.2]:

$$\mathbb{E}[T_{\text{part}}(n)] = n - 1 + \frac{1}{n} \cdot \sum_{k=1}^{n} \mathbb{E}[T_{\text{part}}(\max\{k-1, n-k\})] \leq 4n. \quad (4)$$

Similarly, if we choose the pivot with the median-of-three method, we obtain:

$$\mathbb{E}[T_{\text{part}}(n)] = n - 1 + \frac{1}{\binom{n}{3}} \cdot \sum_{k=1}^{n} (k-1)(n-k)\mathbb{E}[T_{\text{part}}(\max\{k-1, n-k\})]. \quad (5)$$

Equation 5 holds since we assume that all $\binom{n}{3}$ three-element-sets are chosen equally likely as sample for pivot selection. Moreover, for a fixed $k \in \{1, \ldots, n\}$ there are exactly $(k-1)(n-k)$ three-element-sets where k is the median. By induction on n, (5) yields:

$$\mathbb{E}[T_{\text{part}}(n)] \leq 3.2n + \mathcal{O}(\log n). \quad (6)$$

The proof of the first part of Theorem 1 (when the pivot is chosen randomly or as median of three random elements) follows from (4) and (6), Theorem 2, and Proposition 1. The more interesting case is when the sample size grows with n. Here, we obtain significantly better bounds. The key step is Lemma 4. It is a Chernoff type result and actually it can be derived using classical Chernoff bounds. However, we prefer a direct proof because it is straightforward and does not require any additional results.

Lemma 4 *Let* $0 < \delta < \frac{1}{2}$ *and* $\alpha = 4\left(\frac{1}{4} - \delta^2\right) < 1$. *If we choose the pivot as median of* $2c+1$ *elements such that* $2c + 1 \leq \frac{n}{2}$ *then we have*

$$\Pr[\text{pivot} \leq \frac{n}{2} - \delta n] < (2c+1)\alpha^c.$$

Proof First note that the probability for choosing the k-th element as pivot satisfies

$$\binom{n}{2c+1} \cdot \Pr[\text{pivot} = k] = \binom{k-1}{c}\binom{n-k}{c}.$$

We use the notation of *falling factorial* $x^{\underline{\ell}} = x \cdots (x - \ell + 1)$. Thus, $\binom{x}{\ell} = \frac{x^{\underline{\ell}}}{\ell!}$.

$$\begin{aligned}\Pr[\text{pivot } = k] &= \frac{(2c+1)! \cdot (k-1)^{\underline{c}} \cdot (n-k)^{\underline{c}}}{(c!)^2 \cdot n^{\underline{2c+1}}} \\ &= \binom{2c}{c}(2c+1)\frac{1}{(n-2c)}\prod_{i=0}^{c-1}\frac{(k-1-i)(n-k-i)}{(n-2i-1)(n-2i)}.\end{aligned}$$

For $k \le c$ we have $\Pr[\text{pivot } = k] = 0$. So, let $c < k \le \frac{n}{2} - \delta n$ and let us consider an index i in the product with $0 \le i < c$:

$$\begin{aligned}\frac{(k-1-i)(n-k-i)}{(n-2i-1)(n-2i)} &\le \frac{(k-i)(n-k-i)}{(n-2i)(n-2i)} \\ &= \frac{\left(\left(\frac{n}{2}-i\right)-\left(\frac{n}{2}-k\right)\right)\cdot\left(\left(\frac{n}{2}-i\right)+\left(\frac{n}{2}-k\right)\right)}{(n-2i)^2} \\ &= \frac{\left(\frac{n}{2}-i\right)^2-\left(\frac{n}{2}-k\right)^2}{(n-2i)^2} \\ &\le \frac{1}{4} - \frac{\left(\frac{n}{2}-\left(\frac{n}{2}-\delta n\right)\right)^2}{n^2} = \frac{1}{4}-\delta^2.\end{aligned}$$

We have $\binom{2c}{c} \le 4^c$. Since $2c+1 \le \frac{n}{2}$, we obtain:

$$\Pr[\text{pivot } = k] \le 4^c(2c+1)\frac{1}{(n-2c)}\left(\frac{1}{4}-\delta^2\right)^c < (2c+1)\frac{2}{n}\alpha^c.$$

Now, we obtain the desired result:

$$\Pr\left[\text{pivot} \le \frac{n}{2}-\delta n\right] < \sum_{k=0}^{\lfloor \frac{n}{2}-\delta n\rfloor}(2c+1)\frac{2}{n}\alpha^c \le (2c+1)\alpha^c.$$

□

Recall that the only part of Theorem 1 which remains to be shown concerns strategy where the pivot is the median of $f(n)$ randomly selected elements with $f \in \omega(1) \cap o(n)$ with $1 \le f(n) \le n$. As usual, we assume that the median of k elements can be chosen with $\mathcal{O}(k)$ comparisons (and in linear time), e.g. with the algorithm of [1]. Theorem 1 now follows from Theorem 2, Proposition 1, and Theorem 3 which is stated and shown next.

Theorem 3 *Let $f \in \omega(1) \cap o(n)$ with $1 \le f(n) \le n$ and let the pivot be chosen as median of $f(n)$ randomly selected elements. Then the expected number of comparisons used in all recursive calls of partitioning satisfies*

$$\mathbb{E}[T_{\text{part}}(n)] \le 2n + o(n).$$

Theorem 3 is close to a well-known result in [13, Thm. 5] due to Martínez and Roura on Quickselect, see Corollary 2 for the precise statement. We cannot use it directly because we deal with QuickHeapsort, where after partitioning the recursive call is always on the larger part. Our result has an elementary proof which is simpler than that in [13].

Proof of Theorem 3 As an abbreviation, we let $\mathbb{E}(n) = \mathbb{E}[T_{\text{part}}(n)]$ be the expected number of comparisons performed during partitioning. We are going to show that for all $\epsilon > 0$ there is some $D \in \mathbb{R}$ such that

$$E(n) < (2+\epsilon)n + D. \tag{7}$$

We fix some $1 \geq \epsilon > 0$ and choose $\delta > 0$ such that $(2+\epsilon)\delta < \frac{\epsilon}{4}$. Moreover, for this proof let $\mu = \frac{n+1}{2}$. Positions of possible pivots k with $\mu - \delta n \leq k \leq \mu + \delta n$ form a small fraction of all positions, and they are located around the median. Nevertheless, applying Lemma 4 with $c = (f(n)-1)/2 \in \omega(1) \cap o(n)$ yields for all n, which are large enough:

$$\Pr[\text{pivot} < \mu - \delta n] \leq f(n) \cdot \left(1 - 4\delta^2\right)^{(f(n)-1)/2} \leq \frac{1}{48}\epsilon. \tag{8}$$

The analogous inequality holds for $\Pr[\text{pivot} > \mu + \delta n]$. Because $T_{\text{pivot}}(n) \in o(n)$, we have

$$T_{\text{pivot}}(n) \leq \frac{1}{8}\epsilon n \tag{9}$$

for n large enough. Now, we choose n_0 such that (8) and (9) hold for $n \geq n_0$ and such that we have $(2+\epsilon)\delta + \frac{2}{n_0} < \frac{\epsilon}{4}$. We set $D = E(n_0) + 1$. Hence, for $n < n_0$ the desired result (7) holds. Now, let $n \geq n_0$. From (3) we deduce by symmetry that

$$\begin{aligned} E(n) \leq n - 1 + T_{\text{pivot}}(n) &+ \sum_{k=\lceil \mu - \delta n \rceil}^{\lfloor \mu + \delta n \rfloor} \Pr[\text{pivot} = k] \cdot E(k-1) \\ &+ 2 \sum_{k=\lfloor \mu + \delta n \rfloor + 1}^{n} \Pr[\text{pivot} = k] \cdot E(k-1). \end{aligned}$$

Since E is monotone, $E(k)$ can be bounded by the highest value in the respective interval. Hence, using (9) we obtain

$$\begin{aligned} E(n) &\leq n + \frac{1}{8}\epsilon n + \Pr[\mu - \delta n \leq \text{pivot} \leq \mu + \delta n] \cdot E\left(\lfloor \mu + \delta n \rfloor\right) \\ &\quad + 2\Pr[\text{pivot} > \mu + \delta n] \cdot E(n-1) \\ &\leq n + \frac{1}{8}\epsilon n + \left(1 - \frac{1}{24}\epsilon\right) \cdot E\left(\lfloor \mu + \delta n \rfloor\right) + 2\frac{1}{48}\epsilon \cdot E(n-1). \end{aligned}$$

By induction we assume $E(k) \leq (2+\epsilon)k + D$ for $k < n$. Hence,

$$\begin{aligned} E(n) &\leq n + \frac{1}{8}\epsilon n + \left(1 - \frac{1}{24}\epsilon\right)((2+\epsilon)(\mu + \delta n) + D) + \frac{1}{24}\epsilon((2+\epsilon)n + D) \\ &\leq n + (2+\epsilon)\left(\frac{n+1}{2} + \delta n\right) + \frac{1}{8}\epsilon n + \frac{1}{24}\epsilon(2+\epsilon)n + D \\ &\leq 2n + 1 + \frac{\epsilon}{2} + (2+\epsilon)\delta n + \frac{3}{4}\epsilon n + D \\ &< (2+\epsilon)n + D. \end{aligned}$$

□

Corollary 2 ([13]) *Let $f \in \omega(1) \cap o(n)$ with $1 \leq f(n) \leq n$. When implementing Quickselect with the median of $f(n)$ randomly selected elements as pivot, the expected number of comparisons is $2n + o(n)$.*

Proof In QuickHeapsort the recursion is always on the larger part of the array. Hence, the number of comparisons in partitioning for QuickHeapsort is an upper bound on the number of comparisons in Quickselect. □

In [13] it is also proved that choosing the pivot as median of $\mathcal{O}(\sqrt{n})$ elements is optimal for Quicksort as well as for Quickselect. This suggests that we choose the same value in QuickHeapsort; what is backed by our experiments.

6 Modifications of QuickHeapsort Using Extra-space

In this section we want to describe some modification of QuickHeapsort using n bits of extra storage. We introduce two bit arrays. In one of them (the CompareArray) we store the comparisons already done. Since comparisons always take place between the two children of some node in the heap, we can assign to every inner node one of the three values right, left, unknown – according to the current knowledge about the larger child. Hence, for CompareArray we are willing to spend two bits per inner node. In the other array (the RedGreenArray) we store which element is red and which is green.

Since the heaps have maximum size $n/2$, the RedGreenArray only requires $n/2$ bits. The CompareArray is only needed for the inner nodes of the heaps, i.e., length $n/4$ is sufficient. Totally this sums up to n extra bits.

For the heap construction we do not use the algorithms described in Section 5.1. With the CompareArray we can do better by using the algorithm of McDiarmid and Reed [14]. The heap construction works similarly to Bottom-Up-Heapsort, i.e., the array is traversed backward calling for all inner positions i the Reheap procedure on i. The Reheap procedure takes the subheap with root i and restores the heap condition if it is violated at the position i. First, the Reheap procedure determines a *special leaf* using the SpecialLeaf procedure as described in Section 2, but without moving the elements. Then, the final position of the former root is determined going upward

from the special leaf (bottom-up-phase). In the end, the elements above this final position are moved up towards the root by one position. That means that all but one element which are compared during the bottom-up-phase, stay in their places. Since in the SpecialLeaf procedure these elements have been compared with their siblings, these comparisons can be stored in the CompareArray and can be used later.

With another improvement concerning the construction of heaps with seven elements as in [3] the benefits of this array can be exploited even more.

The RedGreenArray is used during the sorting phase, only. Its functionality is straightforward: Every time a red element is inserted into the heap, the corresponding bit is set to red. The SpecialLeaf procedure can stop as soon as it reaches an element without green children. Whenever a red and a green element have to be compared, the comparison can be skipped.

Theorem 4 *Let $f \in \omega(1) \cap o(n)$ with $1 \leq f(n) \leq n$, e.g., $f(n) = \sqrt{n}$, and let $\mathbb{E}[T(n)]$ be the expected number of comparisons by QuickHeapsort using the CompareArray with the improvement of* [3] *and the RedGreenArray on a fixed input array of size n. Choosing the pivot as median of $f(n)$ randomly selected elements in time $\mathcal{O}(f(n))$, we have*

$$\mathbb{E}[T(n)] \leq n \lg n - 0.997n + o(n).$$

Proof We can analyze the savings by the two arrays separately because the CompareArray only affects comparisons between two green elements, while the RedGreenArray only affects comparisons involving at least one red element.

First, we consider the heap construction using the CompareArray. With this array we obtain the same worst case bound as for the standard heap construction method. However, the CompareArray has the advantage that at the end of the heap construction many comparisons are stored in the array and can be reused for the extraction phase. More precisely: For every comparison except the first one made when going upward from the special leaf, one comparison is stored in the CompareArray since for every additional comparison one element on the path defined by SpecialLeaf stays at its place. Because every pair of siblings has to be compared at one point during the heap construction or extraction, all these stored comparisons can be reused. Hence, we only have to count the comparisons in the SpecialLeaf procedure during the construction plus $\frac{n}{2}$ for the first comparison when going upward. Thus, we get an amortized bound for the comparisons during construction of $\frac{3n}{2}$.

In [3] the notion of *Fine-Heaps* is introduced. A Fine Heap is a heap with the additional CompareArray such that for every node the larger child is stored in the array. Such a Fine-Heap of size m can be constructed using the above method with $2m$ comparisons. In [3] Carlsson, Chen and Mattsson showed that a Fine-Heap of size m actually can be constructed with only $\frac{23}{12}m + \mathcal{O}(\lg^2 m)$ comparisons. Thus, by using the algorithm of [3] as a black-box, we have to invest $\frac{23}{12}m + \mathcal{O}(\lg^2 m)$ for the heap construction and at the end there are $\frac{m}{2}$ comparisons stored in the array. All these comparisons stored in the array are used later. Summing up over all heaps during an execution of QuickHeapsort, we can save another $\frac{1}{12}n$ comparisons additionally to the comparisons saved by the CompareArray with the result of [3]. Hence,

for the amortized cost of the heap construction $T_{\text{cstr}}^{\text{amort}}$ (i.e., the number of comparisons needed to build the heap minus the number of comparisons stored in the CompareArray after the construction which all can be reused later) we have obtained

$$T_{\text{cstr}}^{\text{amort}}(n) \leq \frac{17}{12}n + o(n).$$

This bound is slightly better than the average case for the heap construction with the algorithm of [14] which is $1.52n$.

Now, we want to count the number of comparisons we save using the RedGreenArray. We distinguish the two cases that two red elements are compared and that a red and a green element are compared. Every position in the heap has to turn red at one point. At that time, all nodes below this position are already red. Hence, for that element we save as many comparisons as the element is above the bottom level. Summing over all levels of a heap of size m the saving results in $\approx \frac{m}{4}\cdot 1 + \frac{m}{8}\cdot 2 + \cdots = m \cdot \sum_{i\geq 1} i2^{-i-1} = m$. This estimate is exact up to $\mathcal{O}(\lg m)$-terms. Since the expected number of heaps is $\mathcal{O}(\lg n)$, we obtain for the overall saving the value $T_{\text{saveRR}}(n) = n + \mathcal{O}(\lg^2 n)$.

Another place where we save comparisons with the RedGreenArray is when a red element is compared with a green element. It occurs at least one time – when the node looses its last green child – for every inner node that we compare a red child with a green child. Hence, we save at least as many comparisons as there are inner nodes with two children, i.e., at least $\frac{m}{2} - 1$. Since every element – except the expected $\mathcal{O}(\lg n)$ pivot elements – is part of a heap exactly once, we save at least $T_{\text{saveRG}}(n) \geq \frac{n}{2} + \mathcal{O}(\lg n)$ comparisons when comparing green with red elements. In the average case the saving might be even slightly higher since comparisons can also be saved when a node does not loose its last green child.

Summing up all our savings and using the median of $f(n) \in \omega(1) \cap o(n)$ as pivot we obtain the proof of Theorem 4:

$$\begin{aligned}
\mathbb{E}[T(n)] &\leq T_{\text{cstr}}^{\text{amort}}(n) + T_{\text{ext}}(n) + \mathbb{E}[T_{\text{part}}(n)] - T_{\text{saveRR}}(n) - T_{\text{saveRG}}(n) \\
&\leq \frac{17}{12}n + n \cdot (\lfloor \lg n \rfloor - 3) + 2\{n\} + 2n - \frac{3n}{2} + o(n) \\
&\leq n \lg n - 0.997n + o(n).
\end{aligned}$$

□

7 Experimental Results

We implemented the above described algorithms and others and ran experiments comparing the running times and number of comparisons. We sorted arrays of different sizes; the data elements were randomly chosen 32-bit integers. All our experiments were run on one core of an Intel Core i7-3770 CPU (3.40GHz, 8MB Cache) with 32GB RAM (Operating system: Ubuntu Linux 64bit; Compiler: GNU's `g++` (version 4.6.3) optimized with flag `-O3`).

7.1 Number of comparisons

In Fig. 1 we present the outcome of our experiments for the number of comparisons for different values of n. We implemented different versions of QuickHeapsort – the basic version, the improved variant of Section 2, and the version using bit arrays – and compare these variants with Quicksort, Ultimate Heapsort, Bottom-Up-Heapsort and MDR-Heapsort. All algorithms which require choosing pivot elements are implemented with median of $\sqrt{n}$ elements as pivot – for Quicksort we additionally show the results for the standard version with median of three.

For the QuickHeapsort variants without extra bits we implemented the normal algorithm for heap construction due to Floyd [9] (using at most $2m$ comparisons to

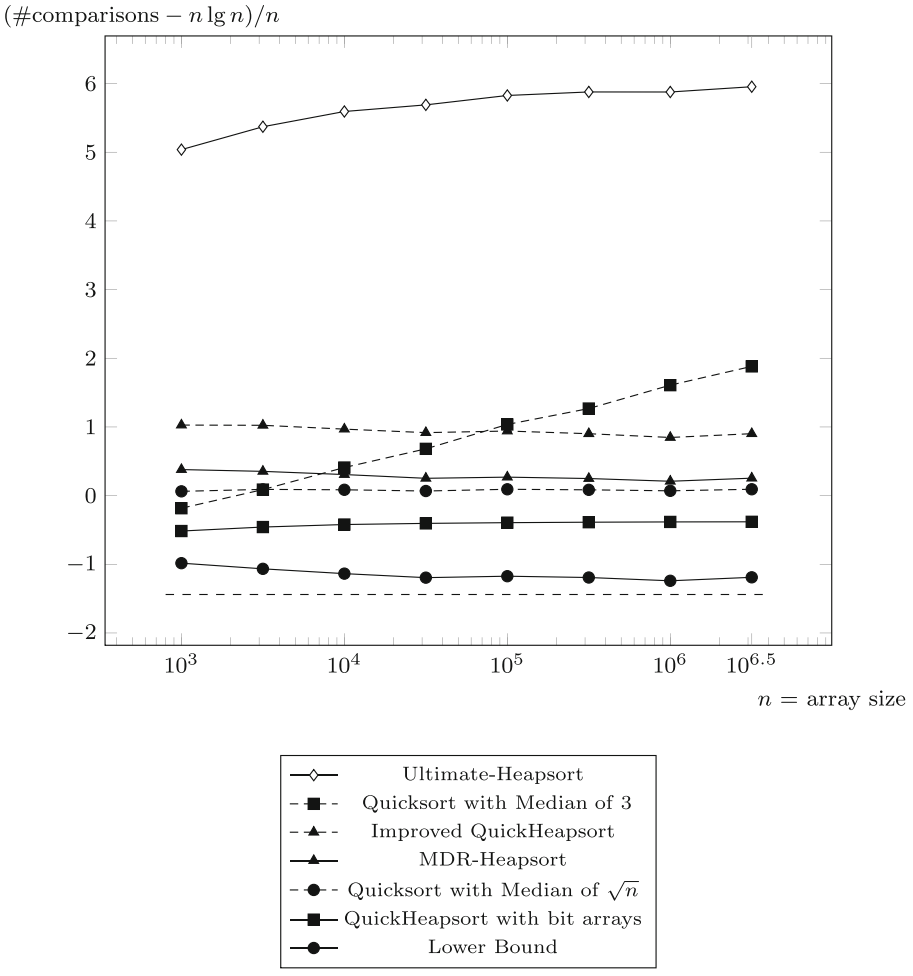

Fig. 1 QuickHeapsort with bit arrays and a median-of-$\sqrt{n}$ pivot selection beats other algorithms with respect to the number of comparisons

Table 1 Number of comparisons of QuickHeapsort and other algorithms run on 10^6 elements (the data for Relaxed-Weak-Heapsort is taken from [8])

Sorting algorithm	Average number of comparisons for $n = 10^6$	
Basic QuickHeapsort with median of 3	21327478	
Basic QuickHeapsort with median of $\sqrt{n}$	20783631	
Improved QuickHeapsort, median of 3	20639046	
QuickHeapsort with bit arrays, median of 3	19207289	
Improved QuickHeapsort, median of $\sqrt{n}$	20135688	
QuickHeapsort with bit arrays, median of $\sqrt{n}$	18690841	∗Best result∗
Quicksort with median of 3	21491310	
Quicksort with median of $\sqrt{n}$	19548149	
Bottom-Up-Heapsort	20294866	
MDR-Heapsort	20001084	
Relaxed-Weak-Heapsort	18951425	
Lower Bound: $\lg n!$	18488884 $\approx \lg(10^6!)$	

construct a heap of m elements). For the variant with extra bits, we implemented the algorithm for heap construction as described in Section 6 (which uses the extra bit array and which is the same as in MDR-Heapsort) – without the additional modification by [3] for the Fine-Heap construction. Fig. 1 gives the the fraction $(\#\text{comparisons} - n \lg n)/n$ for different values of n: this yields approximately the constant of the linear term of the number of comparisons. In Table 1 we display the actual numbers of comparisons for $n = 10^6$. We also added the values for Relaxed-Weak-Heapsort which were presented in [8].

We also compare the different pivot selection strategies on the basic QuickHeapsort with no modifications. We test sample of sizes of one, three, approximately $\lg n$,

Table 2 Different strategies for pivot selection for basic QuickHeapsort tested on 10^4 and 10^6 elements

n	10^4		10^6	
Sample size	Average number of comparisons	Standard deviation	Average number of comparisons	Standard deviation
1	152573	4.281	21975912	3.452
3	146485	2.169	21327478	1.494
$\sim \lg n$	143669	0.954	20945889	0.525
$\sim \sqrt[4]{n}$	143620	0.857	20880430	0.352
$\sim \sqrt{n/\lg n}$	142634	0.413	20795986	0.315
$\sim \sqrt{n}$	142642	0.305	20783631	0.281
$\sim n^{\frac{3}{4}}$	147134	0.195	20914822	0.168

The standard deviation of our experiments is given in percent of the average number of comparisons

Table 3 Running times for QuickHeapsort and other algorithms tested on 10^6 elements, average over 10 runs

Sorting algorithm	integer data time [s]	$\log^{(4)}$-test-function time [s]
Basic QuickHeapsort, median of 3	0.1154	4.21
Basic QuickHeapsort, median of $\sqrt{n}$	0.1171	4.109
Improved QuickHeapsort, median of 3	0.1073	4.049
Improved QuickHeapsort, median of $\sqrt{n}$	0.1118	3.911
QuickHeapsort with bit arrays, median of 3	0.1581	3.756
QuickHeapsort with bit arrays, median of $\sqrt{n}$	0.164	3.70
Quicksort with median of 3	0.1181	3.946
Quicksort with median of $\sqrt{n}$	0.1316	3.648
Ultimate Heapsort	0.135	5.109
Bottom-Up-Heapsort	0.1677	4.132
MDR-Heapsort	0.2596	4.129

$\sqrt[4]{n}$, $\sqrt{n/\lg n}$, $\sqrt{n}$, and $n^{\frac{3}{4}}$ for the pivot selection. In Table 2 the average number of comparisons and the standard deviations are listed. We ran the algorithms on arrays of length $n = 10^4$ and $n = 10^6$. The displayed data is the average resp. standard deviation of 100 runs of QuickHeapsort with the respective pivot selection strategy. Clearly, the standard deviation decreases if more samples are used for pivot selection. The average number of comparisons reaches its minimum with a sample size of approximately $\sqrt{n}$ elements. The table also shows that the difference for the average number of comparisons is relatively small as soon as we use a sample size between $\lg n$ and $n^{\frac{3}{4}}$.

7.2 Running time experiments

In Table 3 we present actual running times of the different algorithms for $n = 10^6$. The numbers displayed here are averages over 10 runs. We tested two different comparison functions. "Integer data time" means the usual integer comparison, whereas "$\log^{(4)}$-test-function time" means that four times the logarithm of both operands is computed before comparing them. This test function has been suggested for example in [8] in order to simulate expensive comparisons. Table 3 shows that for normal integer comparisons the best variant is improved QuickHeapsort with median-of-three for pivot selection. For expensive comparisons, Quicksort with median of $\sqrt{n}$ elements for pivot selection is slightly better than QuickHeapsort with bit arrays and median of $\sqrt{n}$ elements for pivot selection.

8 Conclusion

In this paper we have shown that with known techniques QuickHeapsort can be implemented with expected number of comparisons less than $n \lg n - 0.03n + o(n)$

and extra storage $O(1)$. On the other hand, using n extra bits we can improve this to $n \lg n - 0.997n + o(n)$, i.e., we showed that QuickHeapsort can compete with the most advanced Heapsort variants. These theoretical estimates were also confirmed by our experiments. We also considered different pivot selection strategies. For any constant size sample for pivot selection, QuickHeapsort beats Quicksort for large n since Quicksort performs $\approx Cn \lg n$ comparisons on average with $C > 1$. However, when choosing the pivot as median of $\sqrt{n}$ elements (i.e., with the optimal strategy), then our experiments show that Quicksort needs less comparisons than QuickHeapsort. However, using bit arrays QuickHeapsort is the winner, again. In order to make the last statement rigorous, better theoretical bounds for Quicksort with sampling $\sqrt{n}$ elements are needed. For future work it would also be of interest to prove the optimality of $\sqrt{n}$ elements for pivot selection in QuickHeapsort, to estimate the lower order terms of the expected number of comparisons of QuickHeapsort and also to find an exact average case analysis for the saving by the bit arrays.

Acknowledgments We thank Martin Dietzfelbinger, Stefan Edelkamp, Jyrki Katajainen and the anonymous referees for various comments which improved the presentation of the paper. We also thank Simon Paridon for helping us with the implementation of the algorithms.

Conflict of interests The authors declare that they have no conflict of interest.

References

1. Blum, M., Floyd, R.W., Pratt, V., Rivest, R.L., Tarjan, R.E.: Time bounds for selection. J. Comput. Syst. Sci. **7**(4), 448–461 (1973)
2. Cantone, D., Cincotti, G.: QuickHeapsort, an efficient mix of classical sorting algorithms. Theor. Comput. Sci. **285**(1), 25–42 (2002)
3. Carlsson, S., Chen, J., Mattsson, C.: Heaps with Bits. In: D.-Z. Du, X.-S. Zhang (eds.) ISAAC, vol. 834 of *LNCS*, pp. 288–296. Springer (1994)
4. Chen, J.: A Framework for Constructing Heap-like structures in-place. In: K.-W. Ng, et al. (eds.) ISAAC, vol. 762 of LNCS, pp. 118–127. Springer (1993)
5. Chen, J., Edelkamp, S., Elmasry, A., Katajainen, J.: In-place Heap Construction with Optimized Comparisons, Moves, and Cache Misses. In: B. Rovan, V. Sassone, P. Widmayer (eds.) MFCS, vol. 7464 of LNCS, pp. 259–270. Springer (2012)
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms. The MIT Press, 3rd edn. (2009)
7. Dutton, R.D.: Weak-heap sort. BIT **33**(3), 372–381 (1993)
8. Edelkamp, S., Stiegeler, P.: Implementing HEAPSORT with $n \lg n - 0.9n$ and QUICKSORT with $n \lg n + 0.2n$ comparisons. ACM J. of Exp. Alg., 7:5 (2002)
9. Floyd, R.W.: Algorithm 245: Treesort. Commun. ACM **7**(12), 701 (1964)
10. Gonnet, G.H., Munro, J.I.: Heaps on Heaps. SIAM J. Comput. **15**(4), 964–971 (1986)
11. Katajainen, J.: The ultimate heapsort. In: X. Lin (ed.) CATS, vol. 20 of Australian Computer Science Communications, pp. 87–96. Springer-Verlag (1998)
12. Knuth, D.E.: The art of computer programming. vol. 3 Addison-Wesley (1998)
13. Martínez, C., Roura, S.: Optimal sampling strategies in Quicksort and Quickselect. SIAM J. Comput. **31**(3), 683–705 (2001)
14. McDiarmid, C., Reed, B.A.: Building Heaps Fast. J. Alg. **10**(3), 352–365 (1989)
15. Reinhardt, K.: Sorting *in-place* with a *worst case* complexity of $n \log n - 1.3n + O(\log n)$ comparisons and $\epsilon n \log n + O(1)$ transports. In: T. Ibaraki, et al. (eds.) ISAAC, vol. 650 of *LNCS*, pp. 489–498. Springer (1992)

16. Wang, X.-D., Wu, Y.-J.: An improved HEAPSORT Algorithm with $n \lg n - 0.788928n$ comparisons in the Worst Case. J. Comput. Sci. Techn. **22**, 898–903 (2007)
17. Wegener, I.: The worst case complexity of McDiarmid and Reed's variant of Bottom-Up-Heap sort is less than $n \lg n + 1.1n$. In: C. Choffrut, M. Jantzen (eds.) STACS, vol. 480 of LNCS, pp. 137–147. Springer (1991)
18. Wegener, I.: BOTTOM-UP-HEAPSORT, a new variant of HEAPSORT, beating, on an average, QUICKSORT (if n is not very small). Theor. Comp. Sci. **118**(1), 81–98 (1993)