A minor improvement of Heapsort.

Heapsort is an efficient algorithm for sorting in situ the elements of a linear array $m(i: 0 \leq i < N)$. When sorting the elements in ascending order, the algorithm maintains $H_2$, defined by

$$H_2: (\forall i, j: p \leq i < j < q \land C_2(i, j): m(i) \geq m(j))$$

where $C_2$ is given by

$$C_2(i, j): 2 \cdot i < j \leq 2 \cdot (i + 1)$$

Note that in terms of $CC_2$, i.e. the transitive closure of $C_2$:

$$CC_2(i, j): C_2(i, j) \lor (\exists k: C_2(i, k) \land CC_2(k, j))$$

we could have formulated also

$$H_2: (\forall i, j: p \leq i < j < q \land CC_2(i, j): m(i) \geq m(j))$$

Relation $H_2$ enjoys the useful property

$$(H_2 \land p = 0) \Rightarrow (\forall j: 0 \leq j < q: m(0) \geq m(j))$$

Algorithm Heapsort has the following form:
\[ p, q := N \div 2, N ; \{ H_2 \land q = N \} \]
\[ \text{do } p \neq 0 \rightarrow p := p - 1 ; \]
\[ \{ H_2(p := p + 1) \} \text{ sift } \{ H_2 \land q = N \} \]
\[ \text{od } ; \{ H_2 \land p = 0 \land q = N \} \]
\[ \text{do } q > 1 \rightarrow \{ H_2 \land p = 0 \} ; q := q - 1 ; m := \text{swap}(0, q) ; \]
\[ \{ H_2(p := p + 1) \} \text{ sift } \{ H_2 \land p = 0 \} \]
\[ \text{od } . \]

Here "H2(p := p + 1)" stands for the predicate that is derived from H2 by replacing in it all (free) occurrences of p by p+1. Since q = N is a precondition of the second repetition and the latter maintains p = 0, property (0) ensures that the sorted sequence is built up "from right to left".

By rearranging elements of array m, routine sift satisfies

\[ \{ H_2(p := p + 1) \} \text{ sift } \{ H_2 \} ; \]

it does so by establishing — by \( w := p - \) and maintaining \( SH_2 \), defined by

\[ SH_2: (\forall i, j: p \leq i < j < q \land CC_2(i, j): m(i) > m(j) \lor i = w) , \]

which enjoys the useful property

\[ (SH_2 \land 2 \cdot w + 1 > q) \Rightarrow H_2 . \]
Routine sift can repeatedly perform under invariance of $S_{H2}$ either $W := 2 \cdot W + 1$ or $W := 2 \cdot W + 2$; sift compares each time $m(w)$ with the maximum of $m(2 \cdot W + 1)$ and $m(2 \cdot W + 2)$. If $m(w)$ is large enough, $H2$ holds and sift terminates; otherwise $w$ can be "doubled" at the price of 2 comparisons and 1 swap in array $m$. For further details we refer the reader to [0].

We can do better by replacing $C2$ by $C3$, defined by

$$C3(i,j) : 3 \cdot i < j \leq 3 \cdot (i+1)$$

(and similarly, $C_{C2}$, $H2$, and $S_{H2}$ by $C_{C3}$, $H3$, and $S_{H3}$ respectively). Firstly, we can then start with a smaller $p$, viz. $(N+1) \div 3$; secondly, sift can then "triple" $w$ at the cost of 3 comparisons and 1 swap in array $m$. Thus 6 comparisons and 2 swaps multiply $w$ by 9, whereas originally 6 comparisons and 3 swaps were needed for a factor of 8. (With the analogous $C4$, etc., the gain in comparisons is lost again: $2^3 < 3^2$, but $2^4 = 4^2$. Since $2^5 > 5^2$, $C5$ etc. is expected to lead to more comparisons in sift.)

A worst-case sift is one that terminates with $2 \cdot W + 1 \geq q$ (or $3 \cdot W + 1 \geq q$ respectively). A sort in which all sifts are worst-case sifts would clearly
be a worst-case sort. Since such sorts can occur —see below— and our modification improves worst-case sorts, the worst-case behaviour of Heapsort has, indeed, been improved.

The crucial observation is that, when upon completion of a call of `sift` the final value of `w` is not destroyed, the effect of that call can be undone: `sift` itself has a unique inverse `sift⁻¹` (ending with `w=p`). Starting with an increasing array `m`, we can play Heapsort backwards, supplying each time `sift⁻¹` with a "proper" initial value for `w` such that `2·w+1 ≥ q` (or `3·w+1 ≥ q` respectively) —for a detailed discussion of the notion "proper", see below—. Our backwards game ends with an `m` that would lead to a sort with worst-case shifts only.

Now a detailing of the notion "proper." Our backwards game starts increasing `q` repeatedly by

\[
\{ H2 \land p=0 \} \xrightarrow{sift⁻¹} \{ H2(p:=p+1) \}; \\
m: \text{swap}(0,q); q:=q+1 \{ H2 \land p=0 \} . \tag{1}
\]

Independently of our choice of `w`, `H2` holds after the swap because the new `m(0)` satisfies \(\forall j: 0 \leq j < q: m(0) \geq m(j)\). But does `H2` hold after `q:=q+1`? It does if `m(q-1)` is then small enough. We can achieve this, for instance, by choosing for `sift⁻¹` initially `w=q-1`; program section (1) then
maintains
\[(\forall i \leq q : m(i) \geq m(q-1)).\]

Our backwards game continues increasing \(p\) repeatedly by
\[
\begin{align*}
\{ H2 \land q = N \} & \quad \text{sift}^{-1} \{ H2 \langle p := p+1 \rangle \}; \\
p & := p+1
\end{align*}
\]

Since \(p = 0\) is now not an invariant, we must take precautions to ensure that \text{sift}^{-1} can end with \(w = p\); here "proper" means that for \text{sift}^{-1} we choose initially a \(w\) satisfying \(CC2(p,w)\), i.e. such that\(\) do \(w \neq p \rightarrow w := (w-1) \mod 2\) terminates. For \(C3,\) etc., the same argument applies.

Compared to the above worst-case analysis, the analysis of the average case seems too difficult and insufficiently rewarding.

Acknowledgements. I am indebted to Ross A. Honsberger, who sent me a collection of combinatorial problems, one of which was solved by observing \(2^3 \times 8^2\). The problem was how to partition a given positive integer into positive integer parts such that the product of the parts is maximal. The solution is to take as many parts \(=3\) as is possible without introducing a remaining part \(=1\). The preponderance of 3's is not amazing: 3 is the nearest integer
approximation of \( e \). I am indebted to R. W. Butlerman, who spotted an error in my original form of H3 which failed to satisfy the analogue of (0); in the literature, Heap sort traditionally sorts \( (c, t \in c \in N) \) and unthinkingly I had adopted that unfortunate convention, which induced my error.

I have gratefully adopted Gary Marc Levin's suggestion to indicate subscript ranges uniformly by a predicate. Finally I am indebted to Eric C. R. Hehner and the members of the Tuesday Afternoon Club, who helped me with the worst-case analysis, in which we clearly benefited from our earlier work on program inversion (see [1]).


P.S. Today, 26 May 1981, I learned that Les Goldschlager of Wollongong University, Australia, came to the same conclusion while working at Toronto, Canada. (End of P.S.)

Plataanstraat 5
5671 AL NUENEN
The Netherlands

24 May 1981
prof. dr. Edsger W. Dijkstra
Burroughs Research Fellow