A minor improvement of Heapsort.

Heapsort is an efficient algorithm for sorting in situ the elements of a linear array $m(i: 0 \le i < N)$. When sorting the elements in ascending order, the algorithm maintains H2 , defined by

$$H2: \quad (\underline{A}\, i,j: p \le i < j < q \wedge C2(i,j): m(i) \ge m(j))$$

where C2 is given by

$$C2(i,j): \qquad 2 \cdot i < j \le 2 \cdot (i+1) \quad .$$

Note that in terms of CC2 , i.e. the transitive closure of C2 :

$$CC2(i,j): \qquad C2(i,j) \vee (\underline{E}\, k : C2(i,k): CC2(k,j))$$

we could have formulated also

$$H2: \quad (\underline{A}\, i,j: p \le i < j < q \wedge CC2(i,j): m(i) \ge m(j)) \quad .$$

Relation H2 enjoys the useful property

$$(H2 \wedge p=0) \Rightarrow (\underline{A}\, j: 0 \le j < q: m(0) \ge m(j)) \quad . \quad (0)$$

Algorithm Heapsort has the following form:

$p, q := N \underline{\text{div}}\, 2, N ; \{H2 \wedge q = N\}$
$\underline{\text{do}}\ p \neq 0 \rightarrow p := p-1;$
$\qquad \{H2(p := p+1)\}$ sift $\{H2 \wedge q = N\}$
$\underline{\text{od}}; \{H2 \wedge p = 0 \wedge q = N\}$
$\underline{\text{do}}\ q > 1 \rightarrow \{H2 \wedge p = 0\}\ q := q-1;\ m{:}swap(0, q);$
$\qquad \{H2(p := p+1)\}$ sift $\{H2 \wedge p = 0\}$
$\underline{\text{od}}$.

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

$$\{H2(p := p+1)\}\ \text{sift}\ \{H2\} \qquad ;$$

it does so by establishing — by $w := p$ — and maintaining $SH2$, defined by

$SH2$: $(\underline{A}\, i, j : p \leq i < j < q \wedge CC2(i, j) : m(i) \geq m(j) \vee i = w)$,

which enjoys the useful property

$(SH2 \wedge 2 \cdot w + 1) \geq q) \Rightarrow H2$.

Routine sift can repeatedly perform under invariance of SH2 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

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

(and, similarly, CC2, H2, and SH2 by CC3, H3, and SH3 respectively). Firstly, we can then start with a smaller p , viz. $(N+1)$ <u>div</u> 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 sifts, 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 <u>not</u> 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 $2 \cdot w + 1 \geq q$ (or $3 \cdot w + 1 \geq q$ 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 sifts only.

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

$$\{H2 \wedge p=0\}\ sift^{-1}\ \{H2(p := p+1)\};$$
$$m{:}\,swap(0,q);\ q := q+1\ \{H2 \wedge p=0\} \qquad . (1)$$

Independently of our choice of w, H2 holds after the swap because the new m(0) satisfies $(\underline{A} j : 0 \leq j < q : m(0) \geq m(j))$. 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

$$(\underline{A}\, i : 0 \le i < q : m(i) \ge m(q-1)) \quad .$$

Our backwards game continues increasing p repeatedly by

$$\{H2 \wedge q=N\}\ sift^{-1}\ \{H2(p:=p+1)\};$$
$$p:=p+1 \quad .$$

Since $p=0$ is now not an invariant, we must take precautions to ensure that $sift^{-1}$ can end with $w=p$; here "proper" means that for $sift^{-1}$ we choose initially a w satisfying $CC2(p,w)$, i.e. such that $\underline{do}\ w \ne p \rightarrow w := (w-1)\ \underline{div}\ 2\ \underline{od}$ 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 $2^3 < 3^2$. (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. Bulterman, who spotted an error in my original form of H3 which failed to satisfy the analogue of (0); in the literature, Heapsort traditionally sorts $m(i: 1 \le i \le N)$ 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 benefitted from our earlier work on program inversion (see [1]).

[0] Wirth, Niklaus, Algorithms + Data Structures = Programs, Englewood Cliffs, NJ. USA, Prentice-Hall Inc., 1976, pp. 72-76

[1] Bauer, F.L. and Broy, M. (Ed.), Program Construction, Lecture Notes in Computer Science 69, Berlin Heidelberg New York, Springer Verlag, 1979, pp. 54-57

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