Seeking for the best priority queue: Lessons learnt[1]

Jyrki Katajainen

Department of Computer Science, University of Copenhagen
Universitetsparken 5, 2100 Copenhagen East, Denmark

Introduction

For several years, my research collaborators (Jesper Bojesen, Asger Bruun, Jingsen Chen, Stefan Edelkamp, Amr Elmasry, Ramzi Fadel, Kim Vagn Jakobsen, Claus Jensen, Jens Rasmussen, Maz Spork, Jukka Teuhola, and Fabio Vitale) and I have been studying the performance of different priority queues for different sets of operations in a variety of computational environments. At the theoretical level, we have measured the goodness in terms of the comparison complexity of different operations. As the other optimization criteria, we have considered how to reduce the number of element moves, instructions, branch mispredictions, and cache misses. At the practical level, we have used the actual running time as the key performance indicator. We have done most of our experiments on synthetic request sequences, but we have also done some application engineering.

I assume that the reader is familiar with priority queues studied in most textbooks on algorithms and data structures (see, for example, [10, Section 6.5]). A brief market analysis is given in Table 1. Priority queues can be classified into three categories depending on which operations are supported:

Elementary. *minimum*, *insert*, and *extract-min* are supported (as for a binary heap [38]);

Addressable. *minimum*, *insert*, *extract-min*, *delete*, and *decrease* are supported (the term *addressable* is taken from [32, Section 6.2]);

Mergeable. *minimum*, *insert*, *extract-min*, *delete*, *decrease*, and *union* are supported (the term *mergeable* is taken from [1, Section 4.11]).

Table 1. Asymptotic performance of the most popular priority queues in the word-RAM model. Here N and M denote the number of elements in the priority queues just prior to the operation.

Efficiency / *Operation*	binary heap [33, 38] worst case	Fibonacci heap [23] amortized	run-relaxed heap [11] worst case
minimum	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
insert	$\Theta(\lg N)$	$\Theta(1)$	$\Theta(1)$
decrease	$\Theta(\lg N)$	$\Theta(1)$	$\Theta(1)$
extract-min	$\Theta(\lg N)$	$\Theta(\lg N)$	$\Theta(\lg N)$
delete	$\Theta(\lg N)$	$\Theta(\lg N)$	$\Theta(\lg N)$
union	$\Theta(\lg N \times \lg M)$	$\Theta(1)$	$\Theta(\min\{\lg N, \lg M\})$

[1] Presented at the Dagstuhl Seminar 13391 "Algorithm Engineering" in September 2013

Table 2. Performance of some elementary priority queues in terms of the number of element comparisons performed. Here N denotes the number of elements currently stored. All the data structures mentioned support *minimum* in $O(1)$ worst-case time.

Data structure	*insert*	*extract-min*
binary heaps [38]	$\lg N + O(1)$	$2\lg N + O(1)$
heaps on heaps [24]	$\lg\lg N \pm O(1)^{a)}$	$\lg N + \log^* N \pm O(1)^{b)}$
optimal in-place heaps [15]	$O(1)$	$\lg N + O(1)$
lower bounds	$\Omega(1)$	$\lg N - O(1)$

[a)] Binary search on the *siftup* path (also in [7])
[b)] $\lg N - \lg\lg N$ levels down along the *siftdown* path, *siftup* in a binary-search manner, or recur further down

In my talk I surveyed the theoretical results obtained and I discussed the methodological issues encountered when performing practical experiments. The talk was structured in the question-and-answer form. Below I summarize the questions (**Q**) posed and give links to the papers where (partial) answers (**A**) to these questions are provided.

I Elementary priority queues

Q. What is the best elementary priority queue when handling a request sequence consisting of N *insert* and N *extract-min* operations? Here *best* means in terms of the number of element comparisons performed and in terms of the actual running time. As a binary heap [38], the data structure should be *in-place*, i.e. in addition to an array of elements it should only use $O(1)$ words of additional memory.

A. As to the number of element comparisons performed, the most significant results are summarized in Table 2. Half a century it was open whether there exists an in-place data structure that guarantees $O(1)$ worst-case time for *minimum* and *insert*, and $O(\lg N)$ worst-case time for *extract-min* such that *extract-min* performs at most $\lg N + O(1)$ element comparisons. In view of the lower bounds proved in [24], it was not entirely clear if such a structure exists. We settled this long-standing open problem in [15]. Unfortunately, the devised data structure is *galactic* [30] in a sense that—in spite of its optimal asymptotic behaviour—only a masochist would implement it and one cannot expect any positive effect in performance in this galaxy.

Although impractical, the ideas behind optimal in-place heaps are interesting. To bypass the lower bound for *extract-min*, we reinforce a stronger requirement at the bottom levels that the element at any left child is not larger than the element at its right sibling. To bypass the lower bound for *insert*, we allow $O(\lg^2 N)$ nodes not to obey the binary-heap order in relation to their parents. Time will show if optimal in-place heaps, or some of their substructures, can be converted into practical data structures.

Q. What is the importance of other complexity measures?

A. It is widely known that variants of binary heaps guaranteeing good worst-case comparison complexity are slow in practice (see, e.g. [8, 25]). Therefore, several investigators have tried to improve the performance of binary heaps by considering other complexity measures (the number of element moves, instructions, branch mispredictions, and cache misses). The most significant competitors to Williams' original version [38] are:

Bottom-up version. The *siftup* function is implemented as in the original, but the *siftdown* function is modified such that it first traverses down from the root to a leaf along the path of minimum children and then up along the same path until the correct position of the element under processing is found [27, Section 5.2.3, Exercise 18] (see also [36]).

Multi-ary version. By increasing the number of children per node from 2 to $d > 2$ [26], *insert* will become faster since the height of the heap decreases, but *extract-min* may become slower since the determination of the minimum child at each level is more involved. In practice, values $d \in \{2, 3, 4, 8\}$ are relevant.

Instruction-optimized version. In [3], the *siftdown* and *siftup* functions of the original version were converted into pure C, which is a glorified assembly language, and then the code was rewritten to avoid all extraneous instructions. In particular, move instructions `i = j;` were avoided, array indexing was replaced with pointer arithmetic, and all multiplications were avoided when manipulating pointers.

Branch-optimized version. Assume that `less` is the function used in element comparisons. The idea behind branch optimization proposed in [19] is simple: Remove the **if** statement

```
if (less(a[j], a[j + 1])) {
  j += 1;
}
```

in the *siftdown* function, since the outcome of this branch is hard to predict, and replace it with

```
j += less(a[j], a[j + 1]);
```

Cache-optimized version. External heaps described in [37] are binary heaps where each node is a sorted array of elements. The data structure could be made in-place, but in a practical implementation it is convenient to use two buffers, one reserved for *insert* and another for *extract-min*. Only occasionally, when the insertion buffer gets full or the deletion buffer becomes empty, it is necessary update the heap itself. More advanced external heaps are described in [22, 34].

The theoretical performance of the proposed variants is summarized in Tables 3 a)–d). Observe that the cache performance of external heaps is actually better than that claimed in the original paper [37]. Bojesen [2] observed that the bounds can be improved by increasing the size of the

Table 3. The theoretical performance of different versions of binary heaps. N denotes the number of elements stored, M the number of elements that fit in the cache, and B the number of elements that fit in a cache line. For a function f, tilde notation means that $\sim f(N)$ approaches $f(N)$ as N grows.

a) **# element moves**

Data structure	*insert*	*extract-min*
original version [38]	$\sim\lg N$	$\sim\lg N$
4-ary version [29]	$\sim(1/2)\lg N$	$\sim(1/2)\lg N$

b) **# pure-C instructions**

Data structure	*insert*	*extract-min*
original version [38]	$\sim 9\lg N$	$\sim 12\lg N$
code-tuned version [3]	$\sim 5\lg N$	$\sim 9\lg N$

c) **# branch mispredictions**

Data structure	*insert*	*extract-min*
original version [38]	$O(1)$	$\sim(1/2)\lg N$
branch-optimized version [19]	$O(1)$	$O(1)$

d) **# cache misses**

Data structure	*insert*	*extract-min*
original version [38]	$\sim\lg(N/M)$	$\sim\lg(N/M)$
cache-optimized version [37]	$\sim(1/B)\lg(N/M)^{*)}$	$\sim(2/B)\lg(N/M)^{*)}$

*) amortized

nodes from $\Theta(B)$ to $\Theta(M)$ ($(1/4)M$ in our implementation). Here B and M denote the size of cache lines and the size of the cache (L1 cache in our case), respectively; both measured in elements.

When testing the practical behaviour[2] of the proposed variants, we considered a request sequence consisting of N *insert* operations followed by N *extract-min* operations. The elements were random integers of type `int`. To make sure that clock imprecision did not have any negative effect on the measurement results, we repeated each test $2^{26}/N$ times. Tables 4 a)–f) report the obtained results for different performance indicators.

[2] All the experiments, that should be taken as sanity checks, were carried out in the following environment:

Processor. Intel® Core™ i5-2520M CPU @ 2.50GHz × 4

Memory system. 8-way-associative L1 cache: 32 KB; 12-way-associative L3 cache: 3 MB; cache lines: 64 B; main memory: 3.8 GB

Operating system. Ubuntu 13.04 (Linux kernel 3.5.0-37-generic)

Compiler. `g++` compiler (`gcc` version 4.7.3) with optimization `-O3`

Profiler. `valgrind` simulators (version 3.8.1)

Table 4. The practical performance of different versions of binary heaps for a request sequence consisting of N *insert* operations followed by N *extract-min* operations. M denotes the number of elements that fit in the cache and B the number of elements that fit in a cache line. Each test was repeated $r = 2^{26}/N$ times and the reported values are the grand totals divided by $r \times N \lg N$ or $r \times (N/B) \lg (\max\{2, N/M\})$. CPU times are given in nanoseconds.

a) **# element comparisons**

Data structure \ N	2^{10}	2^{15}	2^{20}	2^{25}
original version [38]	1.73	1.82	1.87	1.89
bottom-up version [36]	1.09	1.07	1.05	1.04
comparison-optimized version [7]	1.49	1.37	1.37	1.3

b) **# element moves**

Data structure \ N	2^{10}	2^{15}	2^{20}	2^{25}
original version [38]	1.96	1.64	1.48	1.39
4-ary version [29]	1.41	1.11	0.95	0.86

c) **# instructions**

Data structure \ N	2^{10}	2^{15}	2^{20}	2^{25}
original version [38]	15.1	14.9	14.8	14.7
code-tuned version [3]	15.5	14.8	14.5	14.3

d) **# branch mispredictions**

Data structure \ N	2^{10}	2^{15}	2^{20}	2^{25}
original version [38]	0.59	0.56	0.54	0.53
branch-optimized version [19]	0.20	0.14	0.10	0.08

e) **# L1 cache misses**

Data structure \ N	2^{10}	2^{15}	2^{20}	2^{25}
original version [38]	1.00	13.1	19.1	19.7
cache-optimized version [37]	1.06	6.83	4.26	3.63

f) **CPU times**

Data structure \ N	2^{10}	2^{15}	2^{20}	2^{25}
`std`-library version [`g++`]	6.41	6.09	6.96	12.6
original version [38]	5.59	5.45	6.47	12.2
comparison-optimized version [7]	9.69	8.34	9.02	14.5
code-tuned version [3]	5.44	5.41	6.31	11.8
branch-optimized version [19]	3.92	4.16	7.2	25.8
bottom-up version [36]	6.77	6.63	7.34	13.1
4-ary version [29]	5.47	5.49	6.3	10.4
cache-optimized version [37]	5.23	5.27	5.69	6.04

Based on the results of these experiments, we conclude the following:

1. In spite of the many papers written on binary heaps, very little progress has been made since the seminal work of Williams [38]. His original programs are fast and the variants proposed seem to make the programs slower (for integer data).
2. In 2000 [3], with code tuning one could improve the running time of the programs by about 20%. On contemporary computers with today's compiler technology, time spent on code tuning seems wasted.
3. Branch optimization pays off for small problem instances, but for large problem instances something strange happens. We do not know the reason for the extreme slowdown when N is large.
4. External heaps work amazingly well. We used the code from Bojesen's heaplab [2] as the starting point for our development, so much of the credit belongs to him. Naturally, we used branch optimization when manipulating the buffers. When implemented this way, an external heap seems to be the only real competitor to an ordinary binary heap. However, the running times are amortized, not worst-case, as for other versions.

II Addressable priority queues

Q. What is the best addressable priority queue when handling a request sequence consisting of N *insert*, N *extract-min*, and M *decrease* operations? We use the number of element comparisons and the actual running time as the performance indicators.

A. Interestingly, in [13] we could prove that Fibonacci heaps [23] are not optimal with respect to the number of element comparisons performed, although they were designed having this request sequence in mind. A rank-relaxed weak heap [13] can process the request sequence with at most $2M + 1.5N \lg N$ element comparisons, whereas the best bound known for a Fibonacci heap is $2M + 2.89N \lg N$ element comparisons. On the other hand, more complicated data-structural transformations are needed in the implementation of a rank-relaxed weak heap; the program code turned out to be about a factor of three longer compared to that of a streamlined implementation of a Fibonacci heap (for details, see [13]).

Q. Does a factor of two matter?

A. Yes, when we talk about the amount of code needed in the implementation of the data structures. In many of our experimental studies [6, 13, 21] we have relied on policy-based design when implementing a set of related data structures. By parametrizing the implementations with policies a significant reuse of code is possible. Also, policy-based design will lead to fairer benchmarking. When only one or two policies are changed, keeping the other parts of the data structure unchanged, it is clear that any differences in performance are due to these changes. That is, we will be less vulnerable for the abilities of an individual programmer.

As to the running time, a single factor-of-two improvement may not be that significant, but in [13] we showed that, when implementing Dijkstra's algorithm for computing the shortest-path distances in a directed graph with non-negative edge lengths, it was possible to make several factor-of-two improvements to the basic algorithm given in [31]. It turned out that the priority queue was not the computational bottleneck in this algorithm. We had to improve the algorithm, simplify the graph representation, reduce the number of cache misses, and tune the underlying priority-queue implementations. First after these changes one could see which priority queue performed best in this application.

In theory rank-relaxed weak heaps were the best, but in this application binary heaps where the fastest and weak heaps [12] (see also [14]) performed the fewest number of element comparisons. In synthetic experiments the lazy variant of Fibonacci heaps was almost unbeatable; for *extract-min*, only weak queues (binary variants of binomial queues) [35] were faster and weak heaps [12] were better with respect to the number of element comparisons performed. For more details on these experiments, consult [13].

With hindsight, it is clear that we made several mistakes when studying the performance of addressable priority queues.

1. In our analysis the emphasis was on the worst-case complexity, whereas in our experiments we considered randomly generated data. We, as others, failed to provide a typical-case analysis simply because this is difficult to grasp.
2. Our focus has been too much on numeric data. For different kind of data the experimental results would have been different (see, e.g. [25]).
3. Many of our experiments indicated that the correlation between the number of element comparisons performed and running time can be poor. Seems that caching effects should be taken more seriously.
4. For dense graphs, Dijkstra's algorithm was not a good benchmark for priority queues because, compared to the number of edges, *decrease* operation was only called few times. We should have considered worst-case input instances as well.
5. We tried hard to program the best theoretical designs. Sometimes we should have taken shortcuts and we could have used more heuristics.

III Mergeable priority queues

Q. What is the best mergeable heap with respect to the number of element comparisons performed and with respect to the running time?

A. Fibonacci heaps [23] support *minimum*, *insert*, *decrease*, and *union* in $O(1)$ amortized time; and *extract-min* and *delete* in $O(\lg N)$ amortized time. Three data structures are known to match these bounds in the worst case. Their performance with respect to the number of element comparisons is summarized in Table 5. The large constant in the leading term in the complexity of *extract-min* and *delete* makes these data structures galactic.

Table 5. Performance of some mergeable heaps in terms of the number of element comparisons performed. N denotes the number of elements currently stored.

Data structure	*extract-min/delete*	*Other operations*
meldable priority queue [4]	$\beta \lg N^{a)}$	$O(1)$
optimal priority queue [20]	$\approx 70 \lg N$	$O(\kappa)^{b)}$
strict Fibonacci heap [5]	$\tau \lg N^{c)}$	$O(1)$
lower bound	$\lg N - O(1)$	$\Omega(1)$

[a)] Brodal's constant β is high
[b)] Katajainen's constant κ is high
[c)] Tarjan's constant τ is unknown

If the complexity of *union* is allowed to be logarithmic, the number of element comparisons involved in *extract-min* and *delete* gets as low as $\lg N + O(\lg \lg N)$ [17]. If the complexity of *decrease* is allowed to be logarithmic, the number of element comparisons involved in *extract-min* and *delete* is bounded above by $2 \lg N + O(1)$ [18]. Even though worst-case-efficient mergeable heaps are seldom needed in practice, it is fascinating how big the jump is in the comparison complexity of *extract-min* and *delete* if all the other operations are required to take $O(1)$ worst-case time.

Concluding remarks

Q. What are the open problems?

A. The driving force in our research has been an intellectual curiosity to determine the comparison complexity of priority-queue operations when different sets of operations are supported. It is open whether the best bounds proved can be improved or not.

Katajainen's 1st conjecture. When *minimum*, *insert*, and *union* (but not *decrease*) are to be supported in $O(1)$ worst-case time, *extract-min* and *delete* can be supported in $O(\lg N)$ worst-case time including at most $\lg N + O(\lg \lg N)$ element comparisons.

Katajainen's 2nd conjecture. When *minimum*, *insert*, *decrease*, and *union* are to be supported in $O(1)$ worst-case time, *extract-min* and *delete* can be supported in $O(\lg N)$ worst-case time including at most $20 \lg N$ element comparisons.

Katajainen's 3rd conjecture. A request sequence consisting of N *insert*, N *extract-min*, and M *decrease* operations can be processed in $O(M + N \lg N)$ worst-case time by performing at most $2M + N \lg N + o(N \lg N)$ element comparisons.

Q. What to do next?

A. In my opinion the following questions would merit further investigation.

1. Try to make the galactic data structures mentioned practical.
2. In the theory of in-place data structures it is normal to assume that an infinite array of elements is available. In our implementations we assumed that N is known beforehand, so we pre-allocated the space needed. If it is necessary to rely on dynamic memory allocation, which effect does memory management have on the performance of the data structures?
3. An external heap [37] is one of the few data structures that is also efficient in main memory. Are there other data structures that have this property? Also, analyse the constant factors in the complexity bounds of external-memory algorithms whose performance is analysed asymptotically.
4. In an early study [25], where we used policy-based design, the policies were quite small. Sometimes such policies introduced an extra overhead because the compiler was not able to optimize highly parametrized programs properly. It would important to understand the reason for this abstraction overhead.

Acknowledgements

As in the past, I enjoyed the stay at Schloss Dagstuhl and I thank the organizers for inviting me to this meeting.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974

[2] J. Bojesen. Heap implementations and variations. 1998. Available online at `http://www.diku.dk/forskning/performance-engineering/Jesper/heaplab/heapsurvey_html/Welcome.html`

[3] J. Bojesen, J. Katajainen, and M. Spork. Performance engineering case study: Heap construction. *ACM J. Exp. Algorithmics* **5**:Article 15, 2000

[4] G. S. Brodal. Worst-case efficient priority queues. *SODA 1996*, pp. 52–58. ACM/SIAM, 1996

[5] G. S. Brodal, G. Lagogiannis, and R. E. Tarjan. Strict Fibonacci heaps. *STOC 2012*, pp. 1177–1184. ACM, 2012

[6] A. Bruun, S. Edelkamp, J. Katajainen, and J. Rasmussen. Policy-based benchmarking of weak heaps and their relatives. *SEA 2010*, LNCS **6049**, pp. 459–435. Springer, 2010

[7] S. Carlsson. A variant of Heapsort with almost optimal number of comparisons. *Inform. Process. Lett. 24*(4):247–250, 1987

[8] S. Carlsson. A note on Heapsort. *Comput. J.* **35**(4):410–411, 1992

[9] J. Chen, S. Edelkamp, A. Elmasry, and J. Katajainen. In-place heap construction with optimized comparisons, moves, and cache misses. *MFCS 2012*, LNCS **7464**, pp. 259–270, Springer, 2012

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. 3nd edition, The MIT Press, 2009

[11] J. R. Driscoll, H. N. Gabow, R. Shrairman, and R. E. Tarjan. Relaxed heaps: An alternative to Fibonacci heaps with applications to parallel computation. *Commun. ACM* **31**(11):1343–1354, 1988

[12] R. D. Dutton. Weak-heap sort. *BIT* **33**(3):372–381, 1993
[13] S. Edelkamp, A. Elmasry, and J. Katajainen. The weak-heap data structure: Variants and applications. *J. Discrete Algorithms* **16**:187–205, 2012
[14] S. Edelkamp, A. Elmasry, and J. Katajainen. Weak heaps engineered. *Journal of Discrete Algorithms*, to appear
[15] S. Edelkamp, A. Elmasry, and J. Katajainen. Optimal in-place heaps. 2013
[16] A. Elmasry, C. Jensen, and J. Katajainen. Multipartite priority queues. *ACM Trans. Algorithms* **5**(1):Article 14, 2008
[17] A. Elmasry, C. Jensen, and J. Katajainen. Two-tier relaxed heaps. *Acta Inform.* **45**(3):193–210, 2008
[18] A. Elmasry, C. Jensen, and J. Katajainen. Strictly-regular number system and data structures. SWAT 2010, LNCS **6139**, pp. 26–37, Springer, 2010
[19] A. Elmasry and J. Katajainen. Lean programs, branch mispredictions, and sorting. *FUN 2012*, LNCS **7288**, pp. 119–130, Springer, 2012
[20] A. Elmasry and J. Katajainen. Worst-case optimal priority queues via extended regular counters. *CSR 2012*, LNCS **7353**, pp. 130–142, Springer, 2012
[21] A. Elmasry and J. Katajainen. Fat heaps without regular counters. *Discrete Mathematics, Algorithms and Applications* **5**(2):Article 1360006, 21 pp., 2013
[22] R. Fadel, K. V. Jakobsen, J. Katajainen, and J. Teuhola. Heaps and heapsort on secondary storage. *Theoret. Comput. Sci.* **220**(2):345–362, 1999
[23] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *J. ACM* **34**(3):596–615, 1987
[24] G. H. Gonnet and J. I. Munro. Heaps on heaps. *SIAM J. Comput.* **15**(4):964–971, 1986
[25] C. Jensen, J. Katajainen, and F. Vitale. An extended truth about heaps. CPH STL Report **2003-5**, Department of Computer Science, University of Copenhagen, 16 pp., 2003
[26] D. B. Johnson. Priority queues with update and finding minimum spanning trees. *Inform. Process. Lett. 4*(3):53–57, 1975
[27] D. E. Knuth. *The Art of Computer Programming*, Vol. 3: *Sorting and Searching*. Addison-Wesley Publishing Company, 1973
[28] D. E. Knuth. *The Stanford GraphBase: A Platform for Combinatorial Computing*. ACM Press, 1994
[29] A. LaMarca and R. Ladner . The influence of caches on the performance of heaps. *ACM J. Exp. Algorithmics* **1**:Article 4, 1996
[30] R. J. Lipton. Galactic algorithms. 2010. Available online at `http://rjlipton.wordpress.com/2010/10/23/galactic-algorithms/`
[31] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999
[32] K. Mehlhorn and P. Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer-Verlag, 2008
[33] J.-R. Sack and T. Strothotte. An algorithm for merging heaps. *Acta Inform.* **22**(2):171–186, 1985
[34] P. Sanders. Fast priority queues for cached memory. *ACM J. Exp. Algorithmics* **5**:Article 7, 2000
[35] J. Vuillemin. A data structure for manipulating priority queues. *Commun. ACM* **21**(4):309–315, 1978
[36] I. Wegener. Bottom-Up-Heapsort, a new variant of Heapsort beating, on an average, Quicksort (if n is not very small). *Theoret. Comput. Sci.* **118**(1):81–98, 1993
[37] L. M. Wegner and J. I. Teuhola. The external heapsort. *IEEE Trans. Softw. Eng.* **15**(7):917–925, 1989
[38] J. W. J. Williams. Algorithm 232: Heapsort. *Commun. ACM* **7**(6):347–348, 1964