Algorithms

L.D. Fosdick and
A.K. Cline, Editors

Some Performance Tests of "quicksort" and Descendants

Rudolf Loeser
Smithsonian Astrophysical Observatory

Detailed performance evaluations are presented for six ACM algorithms: *quicksort* (No. 64), *Shellsort* (No. 201), *stringsort* (No. 207), *"TREESORT3"* (No. 245), *quickersort* (No. 271), and *qsort* (No. 402). Algorithms 271 and 402 are refinements of algorithm 64, and all three are discussed in some detail. The evidence given here demonstrates that *qsort* (No. 402) requires many more comparisons than its author claims. Of all these algorithms, *quickersort* requires the fewest comparisons to sort random arrays.

Key Words and Phrases: sorting, in-place sorting, sorting efficiency, sorting performance tests, quicksort, quickersort, qsort, Shellsort, stringsort, TREESORT3, utility sort algorithm, general-purpose sort algorithm, sorting algorithm documentation

CR Categories: 4.49, 5.31

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

Smithsonian Institution, Astrophysical Observatory, Cambridge, MA 02138.

1. Introduction

Recently, I had to obtain an in-place sort routine requiring the fewest possible comparisons. Thinking that the latest sort procedure among the ACM algorithms would likely be best, I prepared a Fortran version of van Emden's *qsort*, which appeared satisfactory. But thorough testing showed that it does not perform as claimed. Its grandfather, Hoare's *quicksort*, and especially its father, Scowen's *quickersort*, required fewer comparisons. One purpose of this report is to compare the performance of these three algorithms in some detail.

A close look at the published data pertaining to performance tests of these and other ACM algorithms reveals that these data are incomplete. The actual numbers of comparisons required to sort particular arrays are never stated; sometimes the routine actually tested is only vaguely described, and its listing not given. Consequently, reliable conclusions about the relative merits of the algorithms cannot be reached. At least a listing of the actual code tested should be provided, the test arrays should be precisely specified, and the number of comparisons should be explicitly stated; such information is more useful than the timing figures usually reported. The other purpose of this report is to provide this kind of information for the following six ACM sort algorithms:

No. 64	*quicksort*	by C.A.R. Hoare	[1]
No. 201	*Shellsort*	by J. Boothroyd	[2]
No. 207	*stringsort*	by J. Boothroyd	[3]
No. 245	*TREESORT3*	by R.W. Floyd	[4]
No. 271	*quickersort*	by R.S. Scowen	[5]
No. 402	*qsort*	by M.H. van Emden	[6]

The report is arranged in the following sections: Section 2 briefly describes *quicksort*, *quickersort*, and *qsort*; Section 3 outlines the performance tests; and Section 4 discusses the results.

2. *quicksort, quickersort,* and *qsort*

Hoare [7] has provided an excellent account of how *quicksort* works. A brief summary may suffice as background for subsequent discussions. Given an array $a(j)$, $i \leq j \leq k$, to be sorted, *quicksort* first chooses an element Y of this array at random. It then rearranges the elements until the array has been partitioned into three parts: (a) a middle subarray, consisting of Y; (b) a low subarray, none of whose elements is larger than Y; and (c) a high subarray, none of whose elements is less than Y. While Y is now in its proper sorted position and does not require further processing, the low and the high subarrays are not necessarily yet in proper order. *quicksort* continues to partition the subarrays until low or high subarrays of length one are obtained; such subarrays need not be sorted further. Though seemingly complex, this procedure can be easily coded and is very efficient. Its correctness has been proved by Foley and

Hoare [8]. The number of comparisons required to sort an array of length n is proportional to $n \log_2 (n)$.

After an array has been partitioned and either the low or the high subarray chosen for immediate further processing, the index pair (i,k), specifying the subarray whose processing is deferred, is saved in a pushdown stack. The process can be coded recursively to use a stack administered by a compiler/operating system, as Hoare did in *quicksort*, or a stack can be simulated explicitly, as Scowen did in *quickersort* and as was done in the versions shown here.

Hillmore [9] improved on Hoare's published version of *quicksort* by pointing out that it is better not to split subarrays of length two, but just to sort them instead. Sorting such arrays requires only one comparison, whereas splitting may require more.

In *quickersort* [5], Scowen improved on *quicksort* by selecting the middle element of the array as Y, rather than choosing an element at random. He pointed out [5, p. 670] that "the best possible value of Y would be one which splits the segment into two halves of equal size, thus if the array (segment) is roughly sorted, the middle element is an excellent choice. If the array is completely random, the middle element is as good as any other. If however, the array $a[1:j]$ is such that the parts $a[1:j \div 2]$ and $a[j \div 2 + 1:j]$ are both sorted the middle element could be very bad." In such circumstances, Y should be chosen at random, as in *quicksort*.

In addition to choosing Y differently and using Hillmore's modification, Scowen's *quickersort* contains a third improvement over *quicksort*, concerning the length of the stack. The number of index pairs to be saved is minimal ($\leq \log_2 (n)$) if those that define the larger of the two subarrays are stored, while immediate processing continues with the smaller.

While picking the middle element is better than choosing Y at random when the array is already almost in sort order (a not unusual situation), *quicksort* and *quickersort* are expected to perform identically with random arrays. Hoare [7] predicted that the expected average number of comparisons required for *quicksort* to sort n unequal randomly ordered items is $\sim 2n \log_e (n) = 1.386\ n \log_2 (n)$, and stated that the theoretical minimum number of comparisons in this situation, which will be achieved if the splitting always yields subarrays of equal length, is $\log_2 (n!) \sim n \log_2 (n)$. This is discussed further in the Appendix.

After noting that the predicted average expected number of comparisons exceeds the predicted theoretical minimum by a factor of ~ 1.4, Hoare suggested that the number of comparisons could be reduced by choosing Y on the basis of a random sample of the array to be partitioned. Van Emden [10] shows theoretically how the average number of comparisons for the entire array decreases as the size of the random sample increases. This approach has also been investigated by Frazer and McKellar [11].

In *qsort* [6], however, van Emden introduces another method for choosing Y. Briefly, instead of a single element, he chooses two, $X = a(i)$ and $Z = a(j)$, $X \leq Z$, defining a bounding interval. The current values of X and Z are continually updated to allow for a proper partition, which is complete when the elements have been rearranged such that $X = a(i)$ and $Z = a(i + 1)$. Now there are a low subarray, none of whose elements is greater than X, a high subarray, none of whose elements is less than Z, and a middle subarray, which now contains two elements. The low and high segments obtained by *qsort* are expected to contain more nearly the same number of elements than they would with *quicksort* or *quickersort*. The average number of comparisons for random arrays is therefore expected to be closer to the theoretical minimum. Van Emden predicts that this number, for large n, is $1.140\ n \log_2 (n)$, less than the predicted value for *quicksort*, and reports timing data intended to verify this prediction.

3. Performance Tests

Sorting algorithms were tested with five different types of array: (a) sorted arrays: $a(i) = i$, $1 \leq i \leq n$; (b) arrays sorted in reverse order: $a(i) = n + 1 - i$, $1 \leq i \leq n$; and (c) random arrays: $a(i) = r(i)$, $1 \leq i \leq n$, where each $r(i)$ is one of a sequence of normally distributed random numbers in the interval (0, 1), generated by a procedure written by Murphy [12]; (d) arrays almost in sort, generated as follows: first set $a_m(i) = i$, $1 \leq i \leq n$, then pick m elements at random and set each equal to a different random number, i.e. $a_m[n \cdot r(k)] = n \cdot r'(k)$, $1 \leq k \leq m$, $m < n$, $n \cdot r(k) \geq 1$, where $r(k)$ and $r'(k)$ are random numbers as in (c) above; (e) arrays of equal-length sorted blocks, generated as follows: first set $a_m(i) = r(i)$, $1 \leq i \leq n$, where $r(i)$ are random numbers as in (c) above, then sort all m subarrays of length n/m of $a_m(i)$ in place, such that $a_m(i)$ contains m adjacent sorted sequences.

After every test, the array was checked to verify that it had been sorted properly. All tests involving random numbers were repeated R times, the measurements were averaged, and the standard deviations of the averages were computed.

The following parameters were measured: (a) N_c, the number of comparisons; (b) N_f, the number of fetches from the array; (c) N_s, the number of stores into the array; and (d) N_r, the number of partitions required (or the number of segments of length greater than one) (for *quicksort*, *quickersort*, and *qsort* only). The results are shown in Tables I–VII and are discussed briefly in Section 4. All counts in Tables I–VI have been divided by $n \log_2 (n)$ (i.e. a reported count K_r was computed from the observed count K_0 according to $K_r = (0.69315\ K_0)/[n \log_e (n)]$. A value followed by a second one in parentheses is the average of the values from R repeat runs (each repeat uses different random numbers); the value in parentheses is the standard deviation of the average (expressed in percent, truncated).

Table I.

QUICKSORT : SORTING PERFORMANCE TEST RESULTS

LINE	N	M	R	COMPARES	FETCHES	STORES	PARTITIONS
SORTED ARRAYS							
1	32		96	1.231(9)	1.363(8)	0.000(0)	.131(4)
2	128		48	1.249(8)	1.344(7)	0.000(0)	.095(2)
3	512		24	1.267(5)	1.341(4)	0.000(0)	.074(0)
4	2048		12	1.277(4)	1.338(4)	0.000(0)	.061(0)
5	8192		6	1.311(5)	1.362(5)	0.000(0)	.051(0)
6	32768		3	1.321(2)	1.366(2)	0.000(0)	.044(0)
ARRAYS SORTED IN REVERSE ORDER							
7	32		96	1.194(7)	1.525(6)	.200(0)	.131(4)
8	128		48	1.239(6)	1.478(5)	.143(0)	.095(2)
9	512		24	1.281(6)	1.467(5)	.111(0)	.074(1)
10	2048		12	1.307(3)	1.459(2)	.091(0)	.060(0)
11	8192		6	1.282(3)	1.410(3)	.077(0)	.051(0)
12	32768		3	1.273(0)	1.384(0)	.067(0)	.044(0)
RANDOM ARRAYS							
13	32		96	1.150(9)	1.719(6)	.431(7)	.131(4)
14	128		48	1.204(7)	1.739(4)	.440(3)	.094(2)
15	512		24	1.222(5)	1.741(3)	.444(1)	.074(1)
16	2048		12	1.229(2)	1.739(1)	.449(0)	.060(0)
17	8192		6	1.283(3)	1.783(2)	.449(0)	.051(0)
18	32768		3	1.282(1)	1.779(0)	.452(0)	.044(0)
ARRAYS ALMOST IN SORT							
19	32	4	96	1.175(10)	1.538(8)	.225(36)	.131(4)
20	512	64	24	1.227(5)	1.557(4)	.256(8)	.074(1)
21	512	8	24	1.242(6)	1.484(5)	.168(23)	.074(1)
22	512	1	24	1.254(5)	1.394(5)	.066(78)	.074(1)
23	8192	1024	6	1.319(4)	1.614(4)	.243(8)	.051(0)
24	8192	128	6	1.298(3)	1.532(2)	.183(8)	.051(0)
25	8192	16	6	1.266(2)	1.449(1)	.131(23)	.051(0)
26	8192	2	6	1.272(2)	1.393(1)	.070(48)	.051(0)
ARRAYS OF EQUAL-LENGTH SORTED BLOCKS							
27	32	2	96	1.150(8)	1.694(6)	.413(10)	.131(4)
28	32	4	96	1.138(7)	1.706(5)	.438(8)	.125(5)
29	32	16	96	1.144(8)	1.719(5)	.444(7)	.131(4)
30	512	2	24	1.253(6)	1.771(4)	.444(2)	.074(1)
31	512	4	24	1.240(7)	1.756(4)	.442(2)	.074(1)
32	512	16	24	1.215(5)	1.734(3)	.446(2)	.073(1)
33	512	64	24	1.223(6)	1.740(3)	.443(2)	.074(1)
34	8192	2	6	1.274(4)	1.771(3)	.446(0)	.051(0)
35	8192	4	6	1.295(3)	1.795(2)	.449(1)	.051(0)
36	8192	16	6	1.283(3)	1.780(2)	.446(0)	.051(0)
37	8192	64	6	1.308(4)	1.805(3)	.446(1)	.051(0)

ALL COUNTS HAVE BEEN DIVIDED BY
N*LOG2(N)

Table II.

QUICKERSORT : SORTING PERFORMANCE TEST RESULTS

LINE	N	M	R	COMPARES	FETCHES	STORES	PARTITIONS
SORTED ARRAYS							
1	32			.738	1.119	.281	.100
2	128			.795	1.077	.211	.071
3	512			.836	1.057	.166	.056
4	2048			.864	1.046	.136	.045
5	8192			.885	1.039	.115	.038
6	32768			.900	1.033	.100	.033
ARRAYS SORTED IN REVERSE ORDER							
7	32			.813	1.513	.563	.138
8	128			.845	1.358	.413	.100
9	512			.875	1.274	.322	.078
10	2048			.896	1.223	.264	.064
11	8192			.912	1.189	.223	.054
12	32768			.923	1.163	.193	.047
RANDOM ARRAYS							
13	32		96	.925(11)	1.631(6)	.569(6)	.131(4)
14	128		48	1.052(9)	1.675(5)	.528(3)	.094(2)
15	512		24	1.099(6)	1.690(3)	.518(1)	.074(1)
16	2048		12	1.169(5)	1.736(3)	.506(0)	.060(0)
17	8192		6	1.178(3)	1.730(2)	.500(0)	.051(0)
18	32768		3	1.241(3)	1.780(2)	.495(0)	.044(0)
ARRAYS ALMOST IN SORT							
19	32	4	96	.775(5)	1.294(5)	.388(9)	.125(5)
20	512	64	24	.929(3)	1.311(3)	.308(7)	.074(1)
21	512	8	24	.870(2)	1.182(3)	.238(10)	.073(2)
22	512	1	24	.843(1)	1.091(2)	.186(9)	.062(6)
23	8192	1024	6	1.034(2)	1.392(3)	.306(6)	.051(0)
24	8192	128	6	.948(2)	1.212(3)	.214(9)	.051(1)
25	8192	16	6	.909(1)	1.129(3)	.170(16)	.050(2)
26	8192	2	6	.896(0)	1.074(2)	.133(7)	.045(9)
ARRAYS OF EQUAL-LENGTH SORTED BLOCKS							
27	32	2	96	1.425(16)	2.100(11)	.538(9)	.131(4)
28	32	4	96	1.063(15)	1.763(9)	.563(5)	.131(4)
29	32	16	96	.925(13)	1.631(7)	.569(5)	.125(5)
30	512	2	24	3.373(24)	3.919(20)	.471(5)	.074(1)
31	512	4	24	1.602(12)	2.184(9)	.508(1)	.074(1)
32	512	16	24	1.209(8)	1.798(5)	.514(1)	.074(1)
33	512	64	24	1.113(5)	1.704(3)	.516(1)	.074(0)
34	8192	2	6	7.851(36)	8.343(34)	.441(3)	.051(0)
35	8192	4	6	1.880(8)	2.422(6)	.490(1)	.051(0)
36	8192	16	6	1.332(5)	1.880(3)	.497(0)	.051(0)
37	8192	64	6	1.234(3)	1.785(2)	.500(0)	.051(0)

ALL COUNTS HAVE BEEN DIVIDED BY
N*LOG2(N)

Table III.

QSORT SORTING PERFORMANCE TEST RESULTS

LINE	N	M	R	COMPARES	FETCHES	STORES	PARTITIONS
SORTED ARRAYS							
1	32			1.338	.700	0.000	.094
2	128			1.560	.741	0.000	.070
3	512			1.737	.785	0.000	.055
4	2048			1.867	.820	0.000	.045
5	8192			1.963	.847	0.000	.038
6	32768			2.034	.867	0.000	.033
ARRAYS SORTED IN REVERSE ORDER							
7	32			1.338	.700	.200	.094
8	128			1.560	.741	.143	.070
9	512			1.737	.785	.111	.055
10	2048			1.867	.820	.091	.045
11	8192			1.963	.847	.077	.038
12	32768			2.034	.867	.067	.033
RANDOM ARRAYS							
13	32		96	1.156(3)	.738(4)	.463(8)	.081(0)
14	128		48	1.302(1)	.815(3)	.467(2)	.059(1)
15	512		24	1.421(1)	.883(1)	.469(1)	.047(0)
16	2048		12	1.508(1)	.940(2)	.467(0)	.038(0)
17	8192		6	1.576(0)	.989(1)	.467(0)	.033(0)
18	32768		3	1.617(0)	1.011(1)	.470(0)	.028(0)
ARRAYS ALMOST IN SORT							
19	32	4	96	1.225(7)	.838(15)	.250(32)	.081(7)
20	512	64	24	2.074(12)	1.795(13)	.264(13)	.047(1)
21	512	8	24	2.305(24)	1.902(31)	.194(19)	.048(4)
22	512	1	24	1.904(13)	1.132(32)	.065(72)	.051(4)
23	8192	1024	6	2.647(16)	2.346(18)	.280(5)	.033(0)
24	8192	128	6	9.219(19)	8.986(20)	.196(14)	.033(1)
25	8192	16	6	25.940(40)	25.625(41)	.139(17)	.033(4)
26	8192	2	6	5.902(50)	5.214(60)	.073(60)	.033(4)
ARRAYS OF EQUAL-LENGTH SORTED BLOCKS							
27	32	2	96	1.306(1)	.769(2)	.463(9)	.075(8)
28	32	4	96	1.206(3)	.725(4)	.463(9)	.081(0)
29	32	16	96	1.163(3)	.725(5)	.475(7)	.081(7)
30	512	2	24	1.989(3)	1.379(4)	.473(1)	.047(1)
31	512	4	24	1.639(2)	1.071(4)	.461(2)	.047(0)
32	512	16	24	1.466(2)	.923(4)	.465(1)	.047(1)
33	512	64	24	1.411(1)	.871(2)	.465(1)	.047(1)
34	8192	2	6	2.649(2)	1.999(3)	.479(1)	.033(0)
35	8192	4	6	2.035(3)	1.419(4)	.466(1)	.033(0)
36	8192	16	6	1.702(1)	1.104(2)	.466(1)	.032(0)
37	8192	64	6	1.616(2)	1.026(5)	.467(0)	.032(0)

ALL COUNTS HAVE BEEN DIVIDED BY
N*LOG2(N)

Table IV.

SHELLSORT : SORTING PERFORMANCE TEST RESULTS

LINE	N	M	R	COMPARES	FETCHES	STORES
SORTED ARRAYS						
1	32			.644	1.288	0.000
2	128			.724	1.449	0.000
3	512			.780	1.560	0.000
4	2048			.819	1.638	0.000
5	8192			.846	1.693	0.000
6	32768			.867	1.733	0.000
ARRAYS SORTED IN REVERSE ORDER						
7	32			.938	2.750	.875
8	128			1.051	3.009	.906
9	512			1.137	3.200	.926
10	2048			1.199	3.337	.939
11	8192			1.244	3.437	.949
12	32768			1.278	3.511	.956
RANDOM ARRAYS						
13	32		96	1.000(4)	2.900(6)	.900(11)
14	128		48	1.200(2)	3.506(4)	1.104(6)
15	512		24	1.379(2)	4.083(2)	1.325(4)
16	2048		12	1.567(3)	4.733(4)	1.599(6)
17	8192		6	1.779(3)	5.510(4)	1.952(6)
18	32768		3	2.143(3)	6.914(4)	2.628(6)
ARRAYS ALMOST IN SORT						
19	32	4	96	.800(6)	1.963(10)	.350(30)
20	512	64	24	1.166(2)	3.125(3)	.792(7)
21	512	8	24	.945(2)	2.224(4)	.333(14)
22	512	1	24	.813(2)	1.690(5)	.065(72)
23	8192	1024	6	1.514(1)	4.379(2)	1.350(3)
24	8192	128	6	1.225(0)	3.211(1)	.760(2)
25	8192	16	6	1.013(2)	2.361(3)	.334(13)
26	8192	2	6	.900(2)	1.907(5)	.107(48)
ARRAYS OF EQUAL-LENGTH SORTED BLOCKS						
27	32	2	96	.869(2)	2.331(5)	.594(12)
28	32	4	96	1.013(4)	2.975(6)	.950(9)
29	32	16	96	.975(3)	2.806(5)	.850(10)
30	512	2	24	1.079(3)	2.840(5)	.681(10)
31	512	4	24	1.508(3)	4.605(5)	1.589(7)
32	512	16	24	2.492(2)	8.634(3)	3.650(4)
33	512	64	24	1.740(2)	5.579(3)	2.098(3)
34	8192	2	6	1.139(1)	2.911(2)	.633(5)
35	8192	4	6	1.565(2)	4.640(4)	1.510(7)
36	8192	16	6	4.205(1)	15.264(1)	6.855(1)
37	8192	64	6	7.362(0)	28.005(0)	13.281(0)

ALL COUNTS HAVE BEEN DIVIDED BY
N*LOG2(N)

Table V.

STRINGSORT ; SORTING PERFORMANCE TEST RESULTS

LINE	N	M	R	COMPARES	FETCHES	STORES
				SORTED ARRAYS		
1	32			1.175	2.750	.400
2	128			.853	1.991	.286
3	512			.666	1.554	.222
4	2048			.545	1.272	.182
5	8192			.462	1.077	.154
6	32768					
				ARRAYS SORTED IN REVERSE ORDER		
7	32			1.175	2.750	.400
8	128			.853	1.991	.286
9	512			.666	1.554	.222
10	2048			.545	1.272	.182
11	8192			.462	1.077	.154
12	32768					
				RANDOM ARRAYS		
13	32		96	2.819(19)	6.675(19)	1.025(18)
14	128		48	2.868(14)	6.756(14)	1.018(14)
15	512		24	2.776(11)	6.524(11)	.972(11)
16	2048		12	2.659(5)	6.242(5)	.924(5)
17	8192		6	2.672(0)	6.268(0)	.923(0)
18	32768					
				ARRAYS ALMOST IN SORT		
19	32	4	96	2.281(4)	5.356(5)	.794(4)
20	512	64	24	2.605(0)	6.100(0)	.889(0)
21	512	8	24	1.797(17)	4.196(17)	.602(17)
22	512	1	24	.665(0)	1.553(0)	.222(0)
23	8192	1024	6	2.724(0)	6.371(0)	.923(0)
24	8192	128	6	1.839(0)	4.293(0)	.615(0)
25	8192	16	6	1.384(0)	3.229(0)	.462(0)
26	8192	2	6	.923(0)	2.154(0)	.308(0)
				ARRAYS OF EQUAL-LENGTH SORTED BLOCKS		
27	32	2	96	1.156(0)	2.719(0)	.400(0)
28	32	4	96	2.263(1)	5.331(1)	.800(0)
29	32	16	96	2.788(20)	6.588(20)	1.013(19)
30	512	2	24	.665(0)	1.553(0)	.222(0)
31	512	4	24	1.329(0)	3.102(0)	.444(0)
32	512	16	24	1.941(0)	4.549(0)	.667(0)
33	512	64	24	2.546(0)	5.980(0)	.889(0)
34	8192	2	6	.461(0)	1.077(0)	.154(0)
35	8192	4	6	.923(0)	2.153(0)	.308(0)
36	8192	16	6	1.382(0)	3.226(0)	.462(0)
37	8192	64	6	1.808(0)	4.232(0)	.615(0)

ALL COUNTS HAVE BEEN DIVIDED BY N*LOG2(N)

Table VI.

TREESORT3 SORTING PERFORMANCE TEST RESULTS

LINE	N	M	R	COMPARES	FETCHES	STORES
				SORTED ARRAYS		
1	32			1.444	3.544	1.394
2	128			1.628	3.738	1.302
3	512			1.727	3.832	1.246
4	2048			1.785	3.874	1.198
5	8192			1.825	3.917	1.181
6	32768			1.849	3.924	1.151
				ARRAYS SORTED IN REVERSE ORDER		
7	32			1.263	3.056	1.181
8	128			1.444	3.298	1.137
9	512			1.563	3.439	1.096
10	2048			1.641	3.543	1.082
11	8192			1.697	3.613	1.067
12	32768			1.739	3.671	1.062
				RANDOM ARRAYS		
13	32		96	1.388(1)	3.381(1)	1.313(1)
14	128		48	1.556(0)	3.558(0)	1.228(0)
15	512		24	1.656(0)	3.662(0)	1.178(0)
16	2048		12	1.720(0)	3.726(0)	1.146(0)
17	8192		6	1.762(0)	3.767(0)	1.123(0)
18	32768		3	1.794(0)	3.798(0)	1.107(0)
				ARRAYS ALMOST IN SORT		
19	32	4	96	1.425(1)	3.500(1)	1.381(0)
20	512	64	24	1.695(1)	3.763(1)	1.224(1)
21	512	8	24	1.722(0)	3.823(0)	1.243(0)
22	512	1	24	1.728(0)	3.835(0)	1.246(0)
23	8192	1024	6	1.777(2)	3.813(2)	1.148(2)
24	8192	128	6	1.804(1)	3.871(1)	1.166(1)
25	8192	16	6	1.825(0)	3.913(0)	1.177(0)
26	8192	2	6	1.824(0)	3.913(0)	1.178(0)
				ARRAYS OF EQUAL-LENGTH SORTED BLOCKS		
27	32	2	96	1.381(1)	3.363(1)	1.319(1)
28	32	4	96	1.388(1)	3.388(1)	1.325(1)
29	32	16	96	1.400(1)	3.413(1)	1.325(1)
30	512	2	24	1.635(0)	3.603(0)	1.152(0)
31	512	4	24	1.619(0)	3.581(0)	1.154(0)
32	512	16	24	1.641(0)	3.628(0)	1.168(0)
33	512	64	24	1.655(0)	3.661(0)	1.179(0)
34	8192	2	6	1.744(0)	3.717(0)	1.102(0)
35	8192	4	6	1.731(0)	3.697(0)	1.101(0)
36	8192	16	6	1.737(0)	3.710(0)	1.105(0)
37	8192	64	6	1.745(0)	3.728(0)	1.110(0)

ALL COUNTS HAVE BEEN DIVIDED BY N*LOG2(N)

The Fortran listings are exactly those of the routines tested. Since Fortran does not allow the recursive calls of the Algol versions of *quicksort* and *qsort*, I provided a driver that saves and restores index pairs explicitly (Listing 1). This routine drives not only *quicksort* and *qsort*; but *quickersort* as well (even though Scowen did not use recursive calls). Thus:

quicksort	Listing 1 and Listing 2,
quickersort	Listing 1 and Listing 3,
qsort	Listing 1 and Listing 4,
Shellsort	Listing 5,
stringsort	Listing 6,
TREESORT 3	Listing 7.

Tests of all routines except *quicksort* with sorted arrays, reverse-sorted arrays, and random arrays, for $n = 128$ and 2048, were repeated in Algol with a different random-number generator [13]. (The compiler used [14] would not compile *quicksort*.) The Algol versions were exactly as published (with small exceptions in *qsort* and *TREESORT*3, specified later). They were modified for determining N_c, N_s, and N_r. The observed values of N_c, N_s, and N_r from the Algol tests were identical to those in Tables I–VI for sorted and reverse-sorted arrays; for random arrays, the Algol values were well within the standard deviations of the Fortran results.

All the tests reported here involved arrays whose length n is even and equal to an integral power of 2.

Listing 1.

```
      FUNCTION SHORT (A,N,NC,NF,NS,NR,PUSH,LPUSH)
C  RUDOLF LOESER, 1972 OCT 10.
C--THIS IS A SORTING ROUTINE. SPECIFICALLY, IT IS A DRIVER
C  FOR ARRAY-SPLITTING SORTING ALGORITHMS - FUNCTION KSORT.
C--THE CONTENTS OF THE ARRAY A, COMPRISING N ELEMENTS IN
C  N SUCCESSIVE WORDS, WILL BE SORTED INTO NON-DESCENDING
C  ORDER.
C  NORMAL RETURN IS WITH SHORT>0.
C--UPON RETURN, NC = NUMBER OF COMPARISONS; NF = NUMBER OF
C  FETCHES FROM A; NS = NUMBER OF STORES IN TO A.
C--ALSO UPON RETURN, NR = NUMBER OF SUBARRAYS OF LENGTH >1
C  (I.E. THE NUMBER OF CALLS TO KSORT).
C--PUSH, OF LENGTH LPUSH, IS AN ARRAY OF WORKING STORAGE,
C  FOR SIMULATING A PUSHDOWN STACK.
C  LPUSH=2*LOG2(N) SHOULD BE AMPLE.
C  IF, AS SORTING PROCEEDS, MORE STORAGE IS REQUIRED THAN
C  WAS SUPPLIED, SHORT WILL STOP SORTING AND IMMEDIATELY
C  RETURN WITH SHORT=0.
      INTEGER PUSH(1),U,U1,SHORT
      NC=0
      NF=0
      NS=0
      NR=0
      SHORT=1
      IF (N-1) 109,109,101
101   J=LPUSH-2
      M=0
      L1=1
      U1=N
102   IF (U1-L1) 107,107,103
103   K=KSORT(A,L1,U1,L,U,MC,MF,MS)
      NR=NR+1
      NC=NC+MC
      NF=NF+MF
      NS=NS+MS
      IF (K) 107,107,104
104   IF (M-J) 106,106,105
105   SHORT=0
      GO TO 109
106   M=M+2
      PUSH(M-1)=L
      PUSH(M)=U
      GO TO 102
107   IF (M) 109,109,108
108   L1=PUSH(M-1)
      U1=PUSH(M)
      M=M-2
      GO TO 102
109   RETURN
      END
```

Listing 2.

```
      FUNCTION KSORT (A,L1,U1,L,U,NC,NF,NS)
C--THIS IS A SPLITTING ROUTINE FOR SHORT.
C  IT IS AN ADAPTATION OF
C  -QUICKSORT- (ACM ALGORITHMS 63-64) BY C.A.R. HOARE.
C  (TRANSLATED FROM ALGOL,
C  RUDOLF LOESER, 1971 SEP 10).
C--UPON ENTRY, (L1,U1) ARE THE INCLUSIVE LIMITING INDICES
C  OF THE ARRAY TO BE SPLIT. IF U1-L1<1: CASE A:
C  IF U1-L1=1: CASE B: IF U1-L1>1: CASE C.
C--UPON RETURN,
C  IF CASE A: KSORT=0 AND NOTHING WAS DONE:
C  IF CASE B: KSORT=0 AND THE ARRAY WAS SORTED:
C  IF CASE C: KSORT=1 AND THE ARRAY WAS SPLIT, SUCH THAT
C  (L1,U1) ARE THE LIMITING INDICES OF THE SMALLER SEGMENT,
C  AND (L,U) ARE THE LIMITING INDICES OF THE LARGER ONE.
      DIMENSION A(1)
      INTEGER U1,U,P
      NC=0
      NF=0
      NS=0
      IF (U1-L1-1) 100,102,104
100   KSORT=0
101   CONTINUE
      RETURN
102   NC=NC+1
      NF=NF+2
      IF (A(L1)-A(U1)) 100,100,103
103   NF=NF+2
      NS=NS+2
      X=A(L1)
      A(L1)=A(U1)
      A(U1)=X
      GO TO 100
104   KSORT=1
      D=U1-L1+1
105   P=D*RANF(0)
      P=P+L1
      IF (P-L1) 105,106,106
106   IF (P-U1) 107,107,105
107   NF=NF+1
      T=A(P)
      I=L1
      J=U1
108   IF (I-U1) 110,110,109
109   I=U1
      GO TO 112
110   NC=NC+1
      NF=NF+1
      IF (A(I)-T) 111,111,112
111   I=I+1
      GO TO 108
112   IF (J-L1) 113,114,114
113   J=L1
      GO TO 116
114   NC=NC+1
      NF=NF+1
      IF (A(J)-T) 116,115,115
115   J=J-1
      GO TO 112
116   IF (I-J) 117,119,119
117   NF=NF+2
      NS=NS+2
      X=A(I)
      A(I)=A(J)
      A(J)=X
      I=I+1
      J=J-1
      GO TO 108
119   IF (I-P) 120,122,122
120   NF=NF+2
      NS=NS+2
      X=A(I)
      A(I)=A(P)
      A(P)=X
      I=I+1
      GO TO 125
122   IF (P-J) 123,125,125
123   NF=NF+2
      NS=NS+2
      X=A(P)
      A(P)=A(J)
      A(J)=X
      J=J-1
125   IF ((J-L1)-(U1-I)) 127,126,126
126   L=L1
      U=J
      L1=I
      GO TO 101
127   L=I
      U=U1
      U1=J
      GO TO 101
      END
```

Listing 3.

```
      FUNCTION KSORT (A,L1,U1,L,U,NC,NF,NS)
C--THIS IS A SPLITTING ROUTINE FOR SHORT.
C  IT IS AN ADAPTATION OF
C  -QUICKERSORT- (ACM ALGORITHM 271) BY R.A. SCOWEN.
C  (TRANSLATED FROM ALGOL,
C  RUDOLF LOESER, 1971 SEP 10).
C--UPON ENTRY, (L1,U1) ARE THE INCLUSIVE LIMITING INDICES
C  OF THE ARRAY TO BE SPLIT. IF U1-L1<1: CASE A:
C  IF U1-L1=1: CASE B: IF U1-L1>1: CASE C.
C--UPON RETURN,
C  IF CASE A: KSORT=0 AND NOTHING WAS DONE:
C  IF CASE B: KSORT=0 AND THE ARRAY WAS SORTED:
C  IF CASE C: KSORT=1 AND THE ARRAY WAS SPLIT, SUCH THAT
C  (L1,U1) ARE THE LIMITING INDICES OF THE SMALLER SEGMENT,
C  AND (L,U) ARE THE LIMITING INDICES OF THE LARGER ONE.
      DIMENSION A(1)
      INTEGER U1,U,P,Q
      NC=0
      NF=0
      NS=0
      IF (U1-L1-1) 100,102,104
100   KSORT=0
101   CONTINUE
      RETURN
102   NC=NC+1
      NF=NF+2
      IF (A(L1)-A(U1)) 100,100,103
103   NF=NF+2
      NS=NS+2
      X=A(L1)
      A(L1)=A(U1)
      A(U1)=X
      GO TO 100
104   KSORT=1
      P=(L1+U1)/2
      NF=NF+1
      T=A(P)
      NF=NF+1
      NS=NS+1
      A(P)=A(L1)
      Q=U1
      K=L1
106   K=K+1
      IF (K-Q) 107,107,113
107   NC=NC+1
      NF=NF+1
      IF (A(K)-T) 106,106,108
108   IF (Q-K) 113,109,109
109   NC=NC+1
      NF=NF+1
      IF (A(Q)-T) 111,110,110
110   Q=Q-1
      GO TO 108
111   NF=NF+2
      NS=NS+2
      X=A(K)
      A(K)=A(Q)
      A(Q)=X
      Q=Q-1
      GO TO 106
113   NF=NF+2
      NS=NS+2
      A(L1)=A(Q)
      A(Q)=T
      IF ((Q+Q)-(L1+U1)) 116,116,115
115   L=L1
      U=Q-1
      L1=Q+1
      GO TO 101
116   L=Q+1
      U=U1
      U1=Q-1
      GO TO 101
      END
```

They were all repeated with arrays of odd length $n - 1$. (Arrays of length n were generated just as before, but the value of the array length transmitted as a calling argument was $n - 1$.) All the measured quantities were comparable; neither odd nor even array lengths gave anomalous results.

4. Results

The tests with sorted arrays, reverse-sorted arrays, and arrays of equal-length sorted blocks were intended to show how these routines perform under extreme conditions. They do not represent practical applications of sorting. Tests with random arrays and with arrays

Listing 4.

```
      FUNCTION KSORT (A,L1,U1,L,U,NC,NF,NS)
C--THIS IS A SPLITTING ROUTINE FOR SHORT.
C  IT IS AN ADAPTATION OF
C  -QSORT- (ACM ALGORITHM 402) BY M.H. VAN EMDEN.
C  (TRANSLATED FROM ALGOL.
C  RUDOLF LOESER, 1971 JAN 13).
C--UPON ENTRY, (L1,U1) ARE THE INCLUSIVE LIMITING INDICES
C  OF THE ARRAY TO BE SPLIT. IF U1-L1<1: CASE A;
C  IF U1-L1=1: CASE B; IF U1-L1>1: CASE C.
C--UPON RETURN,
C  IF CASE A; KSORT=0 AND NOTHING WAS DONE;
C  IF CASE B; KSORT=0 AND THE ARRAY WAS SORTED;
C  IF CASE C; KSORT=1 AND THE ARRAY WAS SPLIT, SUCH THAT
C  (L1,U1) ARE THE LIMITING INDICES OF THE SMALLER SEGMENT,
C  AND (L,U) ARE THE LIMITING INDICES OF THE LARGER ONE.
      DIMENSION A(1), INDEX(1)
      INTEGER U,U1,P,Q
      KSORT=1
      NC=0
      NS=0
      L=L1
      U=U1
      P=L
      Q=U
      NF=2
      X=A(P)
      Z=A(Q)
      NC=NC+1
      IF (X-Z) 102,102,101
101   NS=NS+2
      Y=X
      X=Z
      A(P)=Z
      Z=Y
      A(Q)=Y
102   IF (U-L-1) 103,103,105
103   KSORT=0
104   RETURN
105   XX=X
      IX=P
      ZZ=Z
      IZ=Q
106   JP=P+1
      KP=Q-1
      IF (KP-JP) 110,107,107
107   DO 109 P=JP,KP
      NF=NF+1
      X=A(P)
      NC=NC+1
      IF (X-XX) 109,111,111
109   CONTINUE
110   P=KP
      GO TO 122
111   JP=P+1
      KP=Q-1
      IF (KP-JP) 115,112,112
112   DO 114 J=JP,KP
      Q=Q-1
      NF=NF+1
      Z=A(Q)
      NC=NC+1
      IF (Z-ZZ) 116,116,114
114   CONTINUE
115   Q=P
      P=P-1
      Z=X
      NF=NF+1
      X=A(P)
116   NC=NC+1
      IF (X-Z) 118,118,117
117   NS=NS+2
      Y=X
      X=Z
      A(P)=Z
      Z=Y
      A(Q)=Y
118   NC=NC+1
      IF (X-XX) 120,120,119
119   XX=X
      IX=P
120   NC=NC+1
      IF (Z-ZZ) 121,106,106
121   ZZ=Z
      IZ=Q
      GO TO 106
122   IF (P-IX) 123,125,123
123   NC=NC+1
      IF (X-XX) 124,125,125
124   NS=NS+2
      A(P)=XX
      A(IX)=X
125   IF (Q-IZ) 126,128,126
126   NC=NC+1
      IF (Z-ZZ) 127,128,127
127   NS=NS+2
      A(Q)=ZZ
      A(IZ)=Z
128   IF ((U-Q)-(P-L)) 130,130,129
129   L1=L
      U1=P-1
      L=Q+1
      GO TO 104
130   U1=U
      L1=Q+1
      U=P-1
      GO TO 104
      END
```

Listing 5.

```
      SUBROUTINE SHELL (A,N,NC,NF,NS)
C--THIS IS A SORTING ROUTINE.
C  IT IS AN ADAPTATION OF
C  -SHELLSORT- (ACM ALGORITHM 201) BY J. BOOTHROYD.
C  (TRANSLATED FROM ALGOL.
C  RUDOLF LOESER, 1971 OCT 11).
C--THE CONTENTS OF THE ARRAY A, COMPRISING N ELEMENTS IN
C  N SUCCESSIVE WORDS, WILL BE SORTED INTO NON-DESCENDING
C  ORDER.
C--UPON RETURN, NC = NUMBER OF COMPARISONS; NF = NUMBER OF
C  FETCHES FROM A; NS = NUMBER OF STORES IN TO A.
      DIMENSION A(1)
      NC=0
      NF=0
      NS=0
      I=1
101   IF (I-N) 102,102,103
102   I=I+I
      GO TO 101
103   M=I-1
104   M=M/2
      IF (M) 110,110,105
105   K=N-M
      DO 109 J=1,K
      I=J+M
106   I=I-M
      IF (I) 109,109,107
107   L=I+M
      NC=NC+1
      NF=NF+2
      IF (A(L)-A(I)) 108,109,109
108   NF=NF+2
      NS=NS+2
      W=A(I)
      A(I)=A(L)
      A(L)=W
      GO TO 106
109   CONTINUE
      GO TO 104
110   RETURN
      END
```

almost in sort, however, have practical relevance and may be useful to someone attempting to choose an algorithm for a particular sorting application.

Results for *quicksort* and *quickersort* appear in Tables I and II, respectively. Both the Fortran and the Algol versions of *quicksort* incorporate the improvement suggested by Hillmore [9]. For random arrays (lines 13–18), the two routines were expected to perform nearly alike. Indeed, their values of N_r are the same, which indicates that they split the arrays in the same way. The differences in the other measurements simply reflect their different scan-termination and array-element permutation techniques.

Overall, *quicksort* is a remarkably stable algorithm. Values of N_c and N_r seem to depend mainly on n and are essentially the same for the different types of arrays. Perhaps this stability results from choosing Y at random, no matter what the situation. In no case does N_c attain Hoare's theoretical minimum $N_c/[n \log_2(n)] = 1$. N_c always remains well below the expected average value $N_c/[n \log_2(n)] = 1.386$.

Algorithm *quickersort* does respond to the type of array. It uses fewer comparisons than *quicksort* when its choice of Y yields values nearer the median than *quicksort*'s random choice. It performs badly on arrays of equal-length sorted blocks, where it picks an inappropriate value for Y. Both these effects were predicted. In many cases, N_c is considerably below $n \log_2(n)$ and even approaches the actual minimum value (cf. Appendix).

The results for *qsort* appear in Table III. The Algol version tested corresponds to the published one, with

Listing 6.

```
      SUBROUTINE STRINGS (A,N,NC,NF,NS)
C--THIS IS A SORTING ROUTINE.
C  IT IS AN ADAPTATION OF
C  -STRINGSORT- (ACM ALGORITHM 207) BY J. BOOTHROYD.
C  (TRANSLATED FROM ALGOL.
C  RUDOLF LOESER, 1971 OCT 11).
C--THE CONTENTS OF THE ARRAY A, COMPRISING N ELEMENTS IN
C  N SUCCESSIVE WORDS, WILL BE SORTED INTO NON-DESCENDING
C  ORDER.
C--UPON RETURN, NC = NUMBER OF COMPARISONS; NF = NUMBER OF
C  FETCHES FROM A; NS = NUMBER OF STORES IN TO A.
C>>USES A(N+1) THROUGH A(2*N) FOR SCRATCH STORAGE.<<
      DIMENSION A(1)
      INTEGER D,U,V,Z,C(3)
      NC=0
      NF=0
      NS=0
100   I=1
      J=N
      C(1)=N+1
      C(3)=N+N
101   D=1
      GO TO 114
102   NC=NC+1
      NF=NF+2
      IF (A(I)-A(Z)) 111,103,103
103   GO TO (104,108), V
104   NC=NC+1
      NF=NF+2
      IF (A(J)-A(Z)) 110,105,105
105   NC=NC+1
      NF=NF+2
      IF (A(I)-A(J)) 108,106,106
106   NF=NF+1
      NS=NS+1
      A(M)=A(J)
      J=J-1
      GO TO 115
108   NF=NF+1
      NS=NS+1
      A(M)=A(I)
      I=I+1
      GO TO 115
110   V=2
      GO TO 108
111   U=2
112   NC=NC+1
      NF=NF+2
      IF (A(J)-A(Z)) 113,106,106
113   D=-D
      C(D+2)=M
114   MD=-D
      M=C(MD+2)
      V=1
      U=1
      GO TO 105
115   Z=M
      M=M+D
      IF (J-I) 117,116,116
116   GO TO (102,112), U
117   IF (M-(N+1)) 100,119,118
118   I=N+1
      J=N+N
      C(1)=1
      C(3)=N
      GO TO 101
119   RETURN
      END
```

Listing 7.

```
      SUBROUTINE TREES (A,N,NC,NF,NS)
C--THIS IS A SORTING ROUTINE.
C  IT IS AN ADAPTATION OF
C  -TREESORT3- (ACM ALGORITHM 245) BY R.W. FLOYD.
C  (TRANSLATED FROM ALGOL.
C  RUDOLF LOESER, 1971 OCT 11).
C--THE CONTENTS OF THE ARRAY A, COMPRISING N ELEMENTS IN
C  N SUCCESSIVE WORDS, WILL BE SORTED INTO NON-DESCENDING
C  ORDER.
C--UPON RETURN, NC = NUMBER OF COMPARISONS; NF = NUMBER OF
C  FETCHES FROM A; NS = NUMBER OF STORES IN TO A.
      DIMENSION A(1)
      NC=0
      NF=0
      NS=0
      I=N/2+1
101   I=I-1
      IF (I-1) 103,103,102
102   CALL SIFTUP (A,I,N,NC,NF,NS)
      GO TO 101
103   I=N+1
104   I=I-1
      IF (I-1) 106,106,105
105   CALL SIFTUP (A,1,I,NC,NF,NS)
      NF=NF+2
      NS=NS+2
      X=A(1)
      A(1)=A(I)
      A(I)=X
      GO TO 104
106   RETURN
      END


      SUBROUTINE SIFTUP (A,L,N,NC,NF,NS)
C--A SUBROUTINE FOR TREES.
      DIMENSION A(1)
      I=L
      NF=NF+1
      COPY=A(I)
100   J=I+I
      IF (J-N) 101,101,107
101   IF (J-N) 102,104,104
102   NC=NC+1
      NF=NF+2
      IF (A(J+1)-A(J)) 104,104,103
103   J=J+1
104   NC=NC+1
      NF=NF+1
      IF (A(J)-COPY) 107,107,105
105   NF=NF+1
      NS=NS+1
      A(I)=A(J)
      I=J
      GO TO 100
107   NS=NS+1
      A(I)=COPY
      RETURN
      END
```

two small modifications: the **if** statement labeled *out* and the next **if** statement as well were recoded as two **if** statements each, so that unnecessary comparisons could be avoided. The modified code reads:

```
out:  if $p \neq ix$ then
      begin if $x \neq xx$ then
          begin $a[p] := xx$; $a[ix] := x$ end
      end;
      if $q \neq iz$ then
      begin if $z \neq zz$ then
          begin $a[q] := zz$; $a[iz] := z$ end
      end;
mark: if $u - q > p - l$ then        etc.
```

The Fortran version is equivalent to this modified Algol version.

Consider first the results for random arrays. The observed value of N_c is greater than *quicksort*'s and *quickersort*'s; in fact, it is almost 50 percent greater than the value van Emden predicted for large n. Nevertheless, there are two indications that *qsort* partitions arrays more effectively than *quickersort*: (1) *qsort*'s standard deviation of N_c is consistently less than half that for *quickersort*; and (2) *qsort*'s N_r is significantly smaller than *quickersort*'s. But only part of this decrease is due to better partitions; some of it comes about because *qsort* makes middle subarrays containing two, rather than just one, elements. To test how *qsort* would perform if it gave middle subarrays containing just one array element (like *quickersort*), I modified it: just before the **if** statement labeled *mark*, I randomly set either $p = q$ or $q = p$. The results, labeled *qsortx*, are shown in Table VIII. About 15 percent of the decrease of N_r is due to the larger middle subarrays; the remaining 85 percent must be due to better partitions.

In a second test, to determine the reason for the large values of N_c, the comparisons occurring in the bounding-interval adjustment section of the algorithm (beginning with the statement labeled *dist* and ending just before the statement labeled *out*) were counted separately. The results appear in Figure 1. When the comparisons in question are excluded from N_c, the remaining number of comparisons does not exceed the predicted limit $N_c/[n \log_2(n)] = 1.140$.

It is not clear why van Emden found that *qsort* ran faster than his version of *quickersort*. The effect of the greater N_c must have been more than compensated by some other effect; most likely, the smaller N_r caused the smaller running time.

I tested the routines *Shellsort*, *stringsort*, and *TREESORT*3 because Blair [15] reported on them in his certification of *quickersort*. He gives only timing data, which show *quickersort* running the fastest. I wanted to see whether the machine-independent measurements of my tests corroborate his results; they do. The results appear in Tables IV-VI. The Algol versions tested were exactly as published, except that in *TREESORT*3, following Abrams' [16] suggestion, *exchange* was not coded as a separate procedure.

Boothroyd's *Shellsort* has the advantage of being a very compact algorithm. His *stringsort*, however, is quite complex and, more important, requires auxiliary storage as large as the array to be sorted. Floyd's *TREESORT*3, whose correctness has been proved by London [17], is much like *quicksort* in its stability, but not in its performance.

All the test results are summarized in Table VII, where, at the position for each measurement, the name of the algorithm producing the smallest value of that measurement is given. (Of course, the column labeled "Partitions" applies only for *quicksort*, *quickersort*, and *qsort*.)

In order to choose an in-place sorting algorithm for a particular application in a particular computing environment, one should consider not only the values of N_c, N_f, N_s, and N_r reported here, but also the times t_c, t_f, t_s, and t_r for comparing, fetching, storing, and initiating a splitting pass, respectively. If relative values of these times are known, then data such as those re-

Fig. 1. Number of comparisons vs array size (random arrays) for *qsort*.

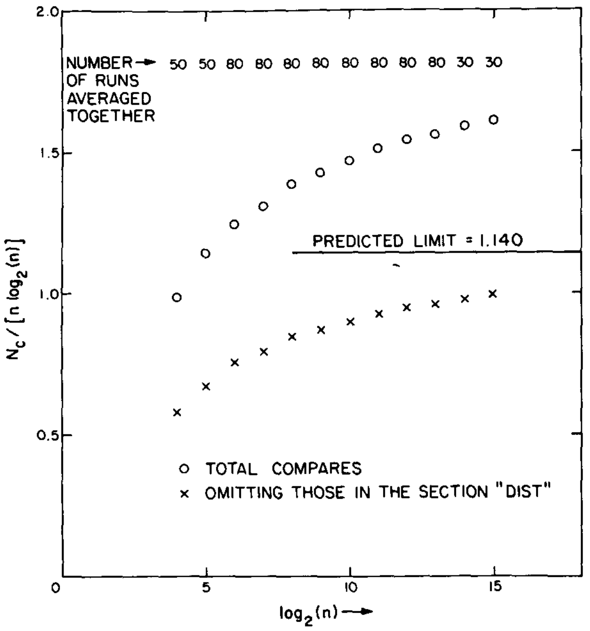


Table VII.

SORTING PERFORMANCE TESTS : SUMMARY
-ROUTINES HAVING SMALLEST VALUES OF THE MEASURED COUNTS.

LINE	N	M	R	COMPARES	FETCHES	STORES	PARTITIONS
				SORTED ARRAYS			
1	32			SHELLSORT	QSORT	QUICKSORT	QSORT
2	128			SHELLSORT	QSORT	QUICKSORT	QSORT
3	512			STRINGSORT	QSORT	QUICKSORT	QSORT
4	2048			STRINGSORT	QSORT	QUICKSORT	QUICKERSORT
5	8192			STRINGSORT	QSORT	QUICKSORT	QUICKERSORT
6	32768			SHELLSORT	QSORT	QUICKSORT	QUICKERSORT
				ARRAYS SORTED IN REVERSE ORDER			
7	32			QUICKERSORT	QSORT	QUICKSORT	QSORT
8	128			QUICKERSORT	QSORT	QUICKSORT	QSORT
9	512			STRINGSORT	QSORT	QUICKSORT	QSORT
10	2048			STRINGSORT	QSORT	QUICKSORT	QSORT
11	8192			STRINGSORT	QSORT	QUICKSORT	QSORT
12	32768			QUICKERSORT	QSORT	QUICKSORT	QSORT
				RANDOM ARRAYS			
13	32			QUICKERSORT	QSORT	QUICKSORT	QSORT
14	128			QUICKERSORT	QSORT	QUICKSORT	QSORT
15	512			QUICKERSORT	QSORT	QUICKSORT	QSORT
16	2048			QUICKERSORT	QSORT	QUICKSORT	QSORT
17	8192			QUICKERSORT	QSORT	QUICKSORT	QSORT
18	32768			QUICKERSORT	QSORT	QUICKSORT	QSORT
				ARRAYS ALMOST IN SORT			
19	32	4		QUICKERSORT	QSORT	QUICKSORT	QSORT
20	512	64		QUICKERSORT	QUICKERSORT	QUICKSORT	QSORT
21	512	8		QUICKERSORT	QUICKERSORT	QUICKSORT	QSORT
22	512	1		STRINGSORT	QUICKERSORT	QSORT	QSORT
23	8192	1024		QUICKERSORT	QUICKERSORT	QUICKSORT	QSORT
24	8192	128		QUICKERSORT	QUICKERSORT	QUICKSORT	QSORT
25	8192	16		QUICKERSORT	QUICKERSORT	QUICKSORT	QSORT
26	8192	2		QUICKERSORT	QUICKERSORT	QUICKSORT	QSORT
				ARRAYS OF EQUAL-LENGTH SORTED BLOCKS			
27	32	2		SHELLSORT	QSORT	STRINGSORT	QSORT
28	32	4		SHELLSORT	QSORT	QUICKSORT	QSORT
29	32	16		QUICKERSORT	QSORT	QUICKSORT	QSORT
30	512	2		STRINGSORT	QSORT	STRINGSORT	QSORT
31	512	4		QUICKSORT	QSORT	QUICKSORT	QSORT
32	512	16		QUICKERSORT	QSORT	QUICKSORT	QSORT
33	512	64		QUICKERSORT	QSORT	QUICKSORT	QSORT
34	8192	2		STRINGSORT	STRINGSORT	STRINGSORT	QSORT
35	8192	4		STRINGSORT	QSORT	STRINGSORT	QSORT
36	8192	16		QUICKSORT	QSORT	QUICKSORT	QSORT
37	8192	64		QUICKERSORT	QSORT	QUICKSORT	QSORT

Table VIII.

Performance on Random Arrays (average of 10 runs)

	n	"quickersort"	"qsortx"	"qsort"
N_r	256	169	160	108
	1024	680	640	432
	4096	2722	2569	1726
N_c	256	2232	3226	2799
	1024	11676	17368	14966
	4096	59070	84702	75545

n = array size

N_r = number of calls to partitioning routine

N_c = number of comparisons

ported here can be used to select the most appropriate algorithm. If the choice is not clear, then the algorithms' performance, in the computing environment in question, should be tested explicitly. In any case, values of n, N_c, N_f, N_s, and N_r should be observed and monitored on an ongoing basis.

Acknowledgment. It is my pleasure to thank Dr. Fred Young for helping with this work, the Computer Center of the Smithsonian Astrophysical Observatory for providing a large amount of running time on its Control Data 6400 system, and the referees for helpful comments.

Appendix

Hoare states the following derivation of a theoretical minimum number of comparisons [7, p. 12]: "The theoretical minimum average number of comparisons required to sort n unequal randomly-ordered items may be estimated on information-theoretic considerations. As a result of a single binary comparison, the maximum entropy which may be destroyed is $-\log(2)$, while the original entropy of the randomly-ordered data is $-\log(n!)$; the final entropy of the sorted data is zero. The minimum number of comparisons required to achieve this reduction in entropy is $-\log(n!)/-\log(2) = \log_2(n!)$." Since values of this expression are not readily obtainable for large n, Hoare also gives the approximation $\log_2(n!) \approx n \log_2(n)$.

This derivation has no particular regard for the detailed behavior of the published version of *quicksort*. The minimum possible number of comparisons for *quicksort* is greater than $\log_2(n!)$, because *quicksort* has been coded to require as many as $n + 2$ comparisons to split an array of length n. On the other hand, *quickersort* has been coded to make n comparisons to split an array of length n. Indeed, it may be possible to devise a scan requiring only $n - 1$ comparisons, since Y is one of the elements of the array to be partitioned and need not be compared with itself.

Optimum partitions, using the minimum number of comparisons, occur when segments are continually split into low and high subarrays of equal length. Consider the case $n = 11$ with a partitioning routine that requires $k = n + 2$ comparisons per scan and splits (rather than sorts) subarrays of length two (e.g., like *quicksort*). The first partition would require 13 comparisons; the result would be two segments of length five (and a middle part of length one, which needs no further consideration). Each segment of length five would require seven comparisons to be split into two segments, each of length two (and again with middle parts of length one). Arrays of length two are the smallest to be split, requiring four comparisons each. The total number of comparisons clearly is $13 + 7 + 7 + 4 + 4 + 4 + 4 = 43$. For $n = 11 + 11 + 1 = 23$, the first partition requires 25 comparisons and can produce two segments each of length 11. The minimum number of comparisons evidently is $43 + 43 + 25 = 111$. Similarly, with $n = 23 + 23 + 1 = 47$, the minimum number of comparisons is $111 + 111 + 49 = 271$.

If subarrays of length two are sorted rather than split (i.e

Table A-1.
Minimum number of comparisons, for sorting arrays of length n with splitting routines requiring k comparisons per splitting scan, producing middle parts of length 1, and either splitting segments of length 2 (case a) or sorting segments of length 2 (case b). All values were divided by $n \log_2(n)$.

n	$\log_2(n!)$	k = n−1		k = n		k = n+1		k = n+2	
		a	b	a	b	a	b	a	b
11	0.664	0.578	0.578	0.762	0.657	0.946	0.736	1.130	0.815
23	0.716	0.634	0.634	0.779	0.702	0.923	0.769	1.067	0.836
47	0.756	0.682	0.682	0.801	0.739	0.919	0.797	1.038	0.854
95	0.789	0.721	0.721	0.822	0.771	0.923	0.820	1.024	0.870
191	0.813	0.753	0.753	0.841	0.797	0.929	0.840	1.016	0.884
383	0.834	0.780	0.780	0.857	0.818	0.935	0.857	1.012	0.895
767	0.850	0.801	0.801	0.871	0.836	0.940	0.871	1.010	0.905
1535	0.864	0.820	0.820	0.882	0.851	0.945	0.882	1.008	0.914
3071	0.876	0.835	0.835	0.892	0.864	0.950	0.892	1.007	0.921
6143	0.886	0.848	0.848	0.901	0.874	0.954	0.901	1.007	0.927
12287	0.894	0.859	0.859	0.908	0.883	0.957	0.908	1.006	0.932
24575	0.901	0.869	0.869	0.914	0.891	0.960	0.914	1.006	0.937
49151	0.907	0.877	0.877	0.919	0.898	0.962	0.920	1.005	0.941
98303	0.913	0.884	0.884	0.925	0.904	0.965	0.925	1.005	0.945
196607	0.918	0.891	0.891	0.929	0.910	0.967	0.929	1.005	0.948
393215	0.922	0.897	0.897	0.933	0.915	0.969	0.933	1.004	0.951
786431	0.926	0.902	0.902	0.936	0.919	0.970	0.936	1.004	0.953

Hillmore's modification), it would take 13 + 7 + 7 + 1 + 1 + 1 + 1 = 31 comparisons to sort an array of length eleven. If, in addition, the splitting routine uses only $k = n$ comparisons per scan, then only 11 + 5 + 5 + 1 + 1 + 1 + 1 = 25 comparisons would be needed to sort that same array.

The foregoing illustrates a recursive procedure that generates a certain sequence of array sizes n_i and the corresponding minimum number of comparisons c_i. The c_i depend on the number of comparisons per scan and on the handling of subarrays of length two; the n_i depend on the number of elements in the middle subarrays. (The values of c_i obtained in this way are probably local minima of the variation of c with n, since the n_i are deliberately chosen to make equal-length subarrays possible—no other values of n permit this.) Table A-1 shows values of c_i for middle subarrays of length one. There are two sets of values for each value of k: (1) for the case that subarrays of length two are partitioned, and (2) for the case that subarrays of length two are sorted. Values of $\log_2 (n_i!)$ are also shown. (These were obtained by summation, according to $\log_2 n! = \log_2 n + \log_2 [(n - 1)!]$.) All numbers in Table A-1 have been divided by $n \log_2 (n)$, as described in Section 3.

Note first that for the array sizes shown, values of the approximation $n \log_2 (n)$ are considerably larger than $\log_2 (n!)$. Nevertheless, it happens that $n \log_2 (n)$ predicts the minimum number of comparisons quite well for a routine like *quicksort* ($k = n + 2$, case 1). However, its values, and those of $\log_2 (n!)$ as well, are too large for a routine like *quickersort* ($k = n$, case 2); and even more so for a routine with $k = n - 1$.

Received February 1972; revised December 1972 and April 1973

References

1. Hoare, C.A.R. Algorithm 64, *quicksort*. *Comm. ACM 4*, 7 (July 1961), p. 321.
2. Boothroyd J. Algorithm 201, *Shellsort*. *Comm. ACM 6*, 8 (Aug. 1963), p. 445.
3. Boothroyd, J. Algorithm 207, *stringsort*. *Comm. ACM 6*, 10 (Oct. 1963), p. 615.
4. Floyd, R.W. Algorithm 245, *TREESORT*3. *Comm. ACM 7*, 12 (Dec. 1964), p. 701.
5. Scowen, R.S. Algorithm 271, *quickersort*. *Comm. ACM 8*, 11 (Nov. 1965), 669–670.
6. van Emden, M.H. Algorithm 402, *qsort*. *Comm. ACM 13*, 11 (Nov. 1970), 693–694.
7. Hoare, C.A.R. *Quicksort*. *Comp. J. 5* (1962), 10–15.
8. Foley, M., and Hoare, C.A.R. Proof of a recursive program: Quicksort. *Comp. J. 14*, 4 (Nov. 1971), 391–395.
9. Hillmore, J.S. Certification of Algorithms 63, 64, 65. *Comm. ACM 5*, 8 (Aug. 1962), p. 439.
10. van Emden, M.H. Increasing the efficiency of quicksort. *Comm. ACM 13*, 9 (Sept. 1970), 563–567.
11. Frazer, W.D., and McKellar, A.C. Samplesort: A sampling approach to minimal storage tree sorting. *J. ACM 17*, 3 (July 1970), 496–507.
12. Murphy, H.M., Jr. RANF (random number generator for Control Data 6000 computers, written 22 Nov. 1968 at Air Force Weapons Laboratory, N.M.).
13. Pike, M.C., and Hill, I.D. Algorithm 266, *random*. *Comm. ACM 8*, 10 (Oct. 1965), 605–606. (Used here with $y = 62104023$.)
14. Control Data Corporation 6000 Series Algol, Version 2 (Scope Level 3.3).
15. Blair, C.R. Certification of Algorithm 271. *Comm. ACM 9*, 5 (May 1966), p. 354.
16. Abrams, P.S. Certification of Algorithm 245. *Comm. ACM 8*, 7 (July 1965), p. 445.
17. London, R.L. Certification of Algorithm 245. *Comm. ACM 13*, 6 (June 1970), 371–373.

Submittal of an algorithm for consideration for publication in Communications of the ACM implies unrestricted use of the algorithm within a computer is permissible.

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

Algorithm 475

Visible Surface Plotting Program [J6]

Thomas Wright [Recd. 18 Apr. 1972, 13 Oct. 1972]
Computing Facility, National Center for Atmospheric Research, Boulder, CO 80302

National Center for Atmospheric Research is sponsored by the National Science Foundation

Key Words and Phrases: hidden line problem, computer graphics, contour surface
CR Categories: 3.65, 4.41, 8.2
Language: Fortran

[This program is not in ANSI Fortran. Nonstandard features are noted in the text. A demonstration driver is included to illustrate use of the subroutines. I/O unit 9 is used by this driver.—LDF.]

Description

This package of three routines produces a perspective picture of an arbitrary object or group of objects with the hidden parts not drawn. The objects are assumed to be stored in the format described below, a format which was chosen to facilitate the display of functions of three variables (Figure 1) or output from three-dimensional computer simulations (Figure 2). The basic method is to contour cuts through the array, starting with a cut nearest the observer. The algorithm leaves out the hidden parts of the contours by suppressing lines enclosed within lines produced while processing preceding cuts. The technique is described in detail in [2].

The object is defined in a three-dimensional array by setting words to one where the object is, and to zero where it is not. That is, the position in the array corresponds to a position in three-space, and the value of the array tells whether any object is present at that position or not. Because a large array is needed to define objects with good resolution, only a part of the array is passed to the package with each call.

There are three subroutines in the package. *INIT*3*D* is called

Fig. 1. Four contour surfaces of the wave function of a 3-P electron in a one electron atom: 50 × 50 × 50 object cube, 100 × 100 screen model.

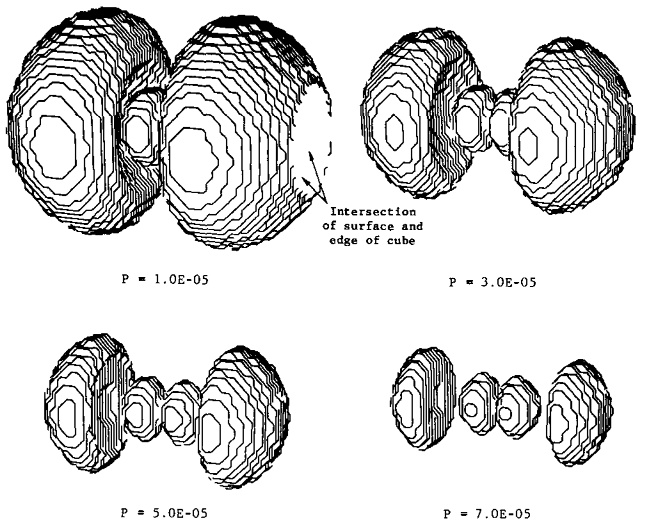


Fig. 2. Output from a three-dimensional cloud model: 100 × 100 × 60 object cube, 200 × 200 screen model.

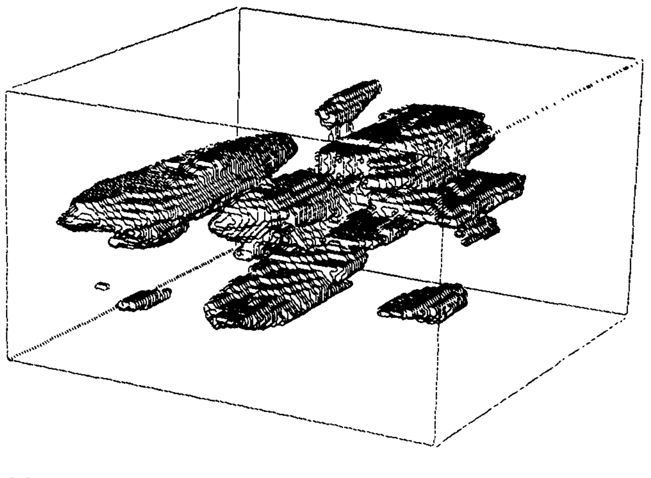
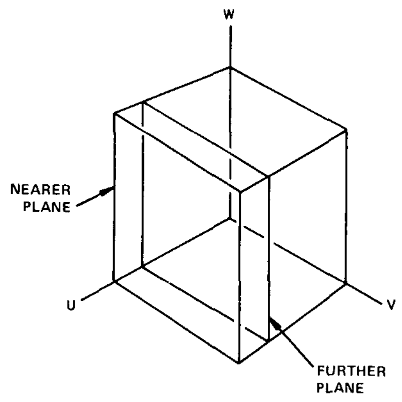


Fig. 3. Processing different parts of a three-dimensional array.



at the beginning of a picture. This call can be skipped sometimes if certain criteria are met and certain precautions are taken. See the comment lines for details. *SETORG* (which has an entry point *PERSPC*) does three-space to two-space perspective transformations. It is called by *INIT3D* and need not be called by the user. The mathematical method for the three-space to two-space transformation is due to Kubert, Szabo, and Giulieri [1]. *DANDR* (*d*raw *and r*emember) is called successively to process different parts of the three-dimensional array. For example, in Figure 3, the nearer plane would be processed in the first call to *DANDR*, while the further plane would be processed in a subsequent call. A sample program is provided with the algorithm to illustrate this point.

Although this package was developed using NCAR's CDC machines with locally written systems and compilers, implementation on different machines or systems should not be too difficult regardless of the plotter. The algorithm has been tested on the Minnesota Fortran compiler (MNF), and when the following items are taken care of, should be portable.

There is a *PROGRAM* card in the demonstration program There is an *ENTRY* statement in *SETORG*. *ENTRY* statements are nonstandard, but are generally portable. It could be eliminated, but the package would run longer. There are two machine-dependent variables used and described in *DANDR*. There is one system routine, *LINE*, called once and described in *DANDR*, which must be implemented or simulated to use this package. In three statements (which are marked) in *DANDR*, *.OR.* and *.AND.* are used for masking operations with integer variables. Some compilers may not produce the desired code, so references to machine language functions may have to be substituted. There is a nonstandard but common form of the *DATA* statement in *DANDR*. Functions which are assumed available are *SQRT*, *ACOS*, and *SIN*.

Figures 4 and 5 are referred to in the listing as the first picture and the second picture.

References

1. Kubert, B., Szabo, J., and Giulieri, S. The perspective representation of functions of two variables. *J. ACM 15*, 2 (Apr. 1968), 193–204.
2. Wright, T. A one-pass hidden-line remover for computer drawn three-space objects. Proc. 1972 Summer Comput. Simulation Conf., pp. 261–267.

Algorithm

```
      PRØGRAM ACMTEST
C DEMØNSTRATIØN PRØGRAM
      DIMENSIØN EYE(3), S(4), ST1(80,80,2), IS2(3,160)
      DIMENSIØN IOBJ(80,80)
C USE WHØLE FRAME
      S(1) = 0.
      S(2) = 1.
      S(3) = 0.
      S(4) = 1.
C SET EYE PØSITIØN
      EYE(1) = 250.
      EYE(2) = 150.
      EYE(3) = 100.
C INITIALIZE PACKAGE
      CALL INIT3D(EYE, 80, 80, 80, ST1, 3, 160, IS2, 9, S)
C CREATE AND PLØT TEST ØBJECT
      DO 50 I=1,80
        A = (I-50)**2
        DØ 40 J=1,80
          C = (J-25)**2
          D = IABS(J-63) + IABS(I-25)
          DØ 30 K=1,80
C FLØØR
            IF (K.EQ.1) GØ TØ 10
C BALL
            IF (SQRT(A+C+(FLØAT(K)-25.)**2).LE.25.) GØ TØ 10
C PØINT
            IF (D.GT.FLØAT(80-K)*.1875) GØ TØ 20
   10       IØBJ(J,K) = 1
            GØ TØ 30
   20       IØBJ(J,K) = 0
   30     CØNTINUE
   40   CØNTINUE
        CALL DANDR(80, 80, ST1, 3, 160, 160, IS2, 9, S, IØBJ,
     *  80)
   50 CONTINUE
C ADVANCE TØ THE NEXT FRAME.
      CALL FRAME
C A SECØND PICTURE WILL NOW BE CALLED USING THE SAME SIZE
C ARRAYS AND EYE PØSITIØN.  THIS MEANS THE CALL TO INIT3D,
C THE BIGGEST TIME CØNSUMER, CAN BE SKIPPED IF THE FØLLØWING
C FØUR LINES ARE INCLUDED.
      REWIND 9
      DØ 70 I=1,3
        DO 60 J=1,160
          IS2(I,J) = 0
   60   CØNTINUE
   70 CØNTINUE
C THIS PICTURE WILL BE THE T=4 CONTOUR SURFACE OF
C T=1/SQRT(U*U+V*V+W*W)+(.5-V)**2/SQRT(U*U+V*V).
      DØ 120 I=1,80
        U = (40.5-FLØAT(I))/79.
        UU = U*U
        DØ 110 J=1,80
          V = (FLØAT(J)-40.5)/79.
          VV = V*V
          A = 1./SQRT(UU+VV)
          DO 100 K=1,80
```

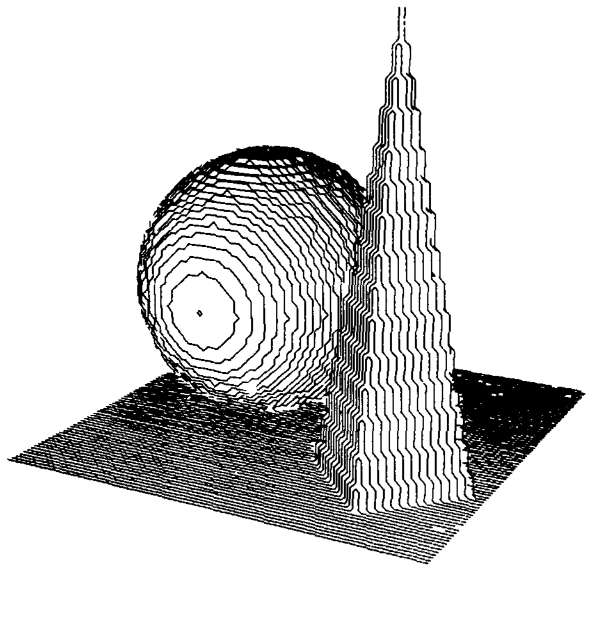


Fig. 4. The first picture produced by the test program.

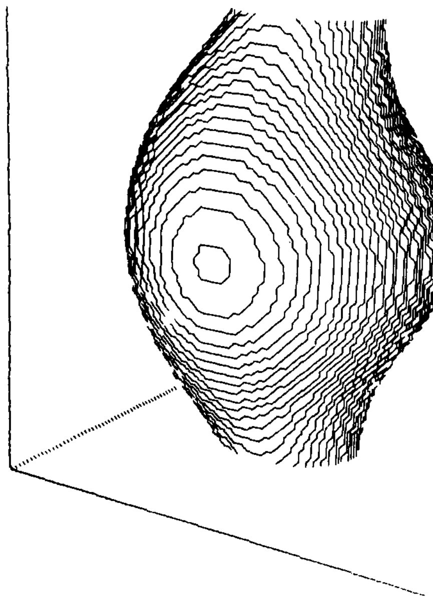


Fig. 5. The second picture produced by the test program.

```
C THE FØLLØWING CARD ADDS AXES.
            IF (I*J.EQ.1 .ØR. I*K.EQ.1 .ØR. J*K.EQ.1) GØ TØ 80
            W = (FLØAT(K)-40.5)/79.
            IF (1./SQRT(UU+VV+W*W)+(.5-V)**2*A.LE.4.) GØ TØ 90
  80        IØBJ(J,K) = 1
            GØ TØ 100
  90        IØBJ(J,K) = 0
 100      CØNTINUE
 110    CØNTINUE
        CALL DANDR(80, 80, ST1, 3, 160, 160, IS2, 9, S, IØBJ,
     *  80)
 120 CØNTINUE
C FLUSH PLØT BUFFER
      CALL FRAME
      STØP
      END

      SUBRØUTINE INIT3D(EYE, NU, NV, NW, ST1, LX, NY, IS2, IU,
     * S)
      DIMENSIØN EYE(3), ST1(NV,NW,2), IS2(LX,NY), S(4)
C BY THØMAS WRIGHT
C CØMPUTING FACILITY
C THE NATIØNAL CENTER FØR ATMØSPHERIC RESEARCH
C BØULDER, CØLØRADØ 80302
C NCAR IS SPØNSØRED BY THE NATIØNAL SCIENCE FØUNDATIØN.
C THE METHØD IS DESCRIBED IN DETAIL IN - A ØNE-PASS HIDDEN-
C LINE REMØVER FØR CØMPUTER DRAWN THREE-SPACE ØBJECTS. PRØC
C 1972 SUMMER CØMPUTER SIMULATIØN CØNFERENCE, 261-267, 1972.
C THIS VERSIØN IS FØR USE ØN CDC 6000 ØR 7000 CØMPUTERS.
C THIS PACKAGE ØF RØUTINES PLØTS 3-DIMENSIØNAL ØBJECTS WITH
C HIDDEN PARTS NØT SHØWN.  ØBJECTS ARE STØRED IN AN ARRAY,
C WITH THE PØSITIØN IN THE ARRAY CØRRESPØNDING TØ A LØCATIØN
C IN 3-SPACE AND THE VALUE ØF THE ARRAY ELEMENT TELLING IF
C ANY ØBJECT IS PRESENT AT THE LØCATIØN.
C INIT3D IS AN INITIALIZATIØN RØUTINE FØR THIS PACKAGE.  IT
C IS CALLED, THEN A SEQUENCE ØF CALLS ARE MADE TØ DANDR TØ
C PRØDUCE A PICTURE.
C EYE    AN ARRAY 3 LØNG CØNTAINING THE U, V, AND W CØØRDI-
C        NATES ØF THE EYE PØSITIØN.  ØBJECTS ARE CØNSIDERED
C        TØ BE IN A BØX WITH 2 EXTREME CØRNERS AT (1,1,1) AND
C        (NU,NV,NW).  THE EYE PØSITIØN MUST HAVE PØSITIVE
C        CØØRDINATES AWAY FRØM THE CØØRDINATE PLANES U=0,
C        V=0, AND W=0.  WHILE GAINING EXPERIENCE WITH THE
C        PACKAGE, USE EYE(1)=5*NU, EYE(2)=4*NV, EYE(3)=3*NW.
C NU     U DIRECTIØN LENGTH ØF THE BØX CØNTAINING THE ØBJECTS
C NV     V DIRECTIØN LENGTH ØF THE BØX CØNTAINING THE ØBJECTS
C NW     W DIRECTIØN LENGTH ØF THE BØX CØNTAINING THE ØBJECTS
C ST1    A SCRATCH ARRAY AT LEAST NV*NW*2 WØRDS LØNG.
C LX     FIRST DIMENSIØN ØF A SCRATCH ARRAY, IS2, USED BY THE
C        PACKAGE FØR REMEMBERING WHERE IT SHØULD NØT DRAW.
C        LX=1+NX/NBPW.  SEE DANDR CØMMENTS FØR NX AND NBPW.
C NY     SECØND DIMENSIØN ØF IS2.  SEE DANDR CØMMENTS.
C IS2    A SCRATCH ARRAY AT LEAST LX*NY WØRDS LØNG.
C IU     UNIT NUMBER ØF SCRATCH FILE FØR THE PACKAGE.  ST1
C        WILL BE WRITTEN NU TIMES ØN THIS FILE.
C S      AN ARRAY 4 LØNG WHICH CØNTAINS THE CØØRDINATES ØF
C        THE AREA WHERE THE PICTURE IS TØ BE DRAWN.  THAT IS,
C        ALL PLØTTING CØØRDINATES GENERATED WILL BE BØUNDED
C        AS FØLLØWS-- X CØØRDINATES WILL BE BETWEEN S(1) AND
C        S(2), Y CØØRDINATES WILL BE BETWEEN S(3) AND S(4).
C        TØ PREVENT DISTØRTIØN, HAVE S(2)-S(1)=S(4)-S(3).
C IF SEVERAL PICTURES ARE TØ BE DRAWN WITH THE SAME SIZE
C ARRAYS AND EYE PØSITIØN AND THE USER REWINDS IU AND FILLS
C IS2 WITH ZERØES, INIT3D NEED NØT BE CALLED FØR ØTHER THAN
C THE FIRST PICTURE.
C SET UP TRANSFØRMATIØN RØUTINE FØR THIS LINE ØF SIGHT.
      U = NU
      V = NV
      W = NW
      CALL SETØRG(U*.5, V*.5, W*.5, EYE(1), EYE(2), EYE(3))
```

```
C FIND EXTREMES IN TRANSFØRMED SPACE.
      CALL PERSPC(1., 1., W, D, YT, D)
      CALL PERSPC(U, V, 1., D, YB, D)
      CALL PERSPC(U, 1., 1., XL, D, D)
      CALL PERSPC(1., V, 1., XR, D, D)
C ADJUST EXTREMES TØ PREVENT DISTØRTIØN WHEN GØING FRØM
C TRANSFØRMED SPACE TØ PLØTTER SPACE.
      DIF = (XR-XL-YT+YB)*.5
      IF (DIF) 10, 30, 20
   10 XL = XL + DIF
      XR = XR - DIF
      GØ TØ 30
   20 YB = YB - DIF
      YT = YT + DIF
   30 REWIND IU
C FIND THE PLØTTER CØØRDINATES ØF THE 3-SPACE LATTICE PØINTS
      C1 = .9*(S(2)-S(1))/(XR-XL)
      C2 = .05*(S(2)-S(1)) + S(1)
      C3 = .9*(S(4)-S(3))/(YT-YB)
      C4 = .05*(S(4)-S(3)) + S(3)
      DØ 60 I=1,NU
        U = NU + 1 - I
        DØ 50 J=1,NV
          V = J
          DØ 40 K=1,NW
            CALL PERSPC(U, V, FLØAT(K), X, Y, D)
            ST1(J,K,1) = C1*(X-XL) + C2
            ST1(J,K,2) = C3*(Y-YB) + C4
   40     CØNTINUE
   50   CØNTINUE
C WRITE THEM ØN UNIT IU.
        WRITE (IU) ST1
   60 CØNTINUE
      REWIND IU
C ZERØ ØUT ARRAY WHERE VISIBILITY IS REMEMBERED.

      DØ 80 J=1,NY
        DØ 70 I=1,LX
          IS2(I,J) = 0
   70   CØNTINUE
   80 CØNTINUE
      RETURN
      END

      SUBRØUTINE SETØRG(X, Y, Z, XT, YT, ZT)
C THIS RØUTINE IMPLEMENTS THE 3-SPACE TØ 2-SPACE TRANSFØR-
C MATIØN BY KUBER, SZABØ AND GIULIERI, THE PERSPECTIVE
C REPRESENTATIØN ØF FUNCTIØNS ØF TWØ VARIABLES. J. ACM 15,
C 2, 193-204,1968.
C SETØRG ARGUMENTS
C X,Y,Z     ARE THE 3-SPACE CØØRDINATES ØF THE INTERSECTIØN
C           ØF THE LINE ØF SIGHT AND THE IMAGE PLANE.  THIS
C           PØINT CAN BE THØUGHT ØF AS THE PØINT LØØKED AT.
C XT,YT,ZT ARE THE 3-SPACE CØØRDINATES ØF THE EYE PØSITIØN.
C PERSPC ARGUMENTS
C X,Y,Z     ARE THE 3-SPACE CØØRDINATES ØF A PØINT TØ BE
C           TRANSFØRMED.
C XT,YT     THE RESULTS ØF THE 3-SPACE TØ 2-SPACE TRANSFØR-
C           MATIØN.
C ZT        NØT USED.
C STØRE THE PARAMETERS ØF THE SETØRG CALL FØR USE WHEN
C PERSPC IS CALLED.
      AX = X
      AY = Y
      AZ = Z
      EX = XT
      EY = YT
      EZ = ZT
C AS MUCH CØMPUTATIØN AS PØSSIBLE IS DØNE DURING EXECUTIØN
C ØF SETØRG SINCE PERSPC IS CALLED THØUSANDS ØF TIMES FØR
```

```
C EACH CALL TØ SETØRG.
      DX = AX - EX
      DY = AY - EY
      DZ = AZ - EZ
      D = SQRT(DX*DX+DY*DY+DZ*DZ)
      CØSAL = DX/D
      CØSBE = DY/D
      CØSGA = DZ/D
      AL = ACØS(CØSAL)
      BE = ACØS(CØSBE)
      GA = ACØS(CØSGA)
      SINGA = SIN(GA)
C THE 3-SPACE PØINT LØØKED AT IS TRANSFØRMED INTØ (0,0) ØF
C THE 2-SPACE.  THE 3-SPACE Z AXIS IS TRANSFØRMED INTØ THE
C 2-SPACE Y AXIS.  IF THE LINE ØF SIGHT IS CLØSE TØ PARALLEL
C TØ THE 3-SPACE Z AXIS, THE 3-SPACE Y AXIS IS CHØSEN (IN-
C STEAD ØF THE 3-SPACE Z AXIS) TØ BE TRANSFØRMED INTØ THE
C 2-SPACE Y AXIS.
      IF (SINGA.LT.0.0001) GØ TØ 10
      R = 1./SINGA
      ASSIGN 20 TØ JUMP
      RETURN
   10 SINBE = SIN(BE)
      R = 1./SINBE
      ASSIGN 30 TØ JUMP
      RETURN
C ******************** ENTRY PERSPC **********************
      ENTRY PERSPC
      Q = D/((X-EX)*CØSAL+(Y-EY)*CØSBE+(Z-EZ)*CØSGA)
      GØ TØ JUMP, (20,30)
   20 XT = ((EX+Q*(X-EX)-AX)*CØSBE-(EY+Q*(Y-EY)-AY)*CØSAL)*R
      YT = (EZ+Q*(Z-EZ)-AZ)*R
      RETURN
   30 XT = ((EZ+Q*(Z-EZ)-AZ)*CØSAL-(EX+Q*(X-EX)-AX)*CØSGA)*R
      YT = (EY+Q*(Y-EY)-AY)*R
      RETURN
      END


      SUBRØUTINE DANDR(NV, NW, ST1, LX, NX, NY, IS2, IU, S,
     * IØBJS, MV)
      DIMENSIØN ST1(NV,NW,2), IS2(LX,NY), S(4), IØBJS(MV,NW)
C THIS RØUTINE IS CALLED NU TIMES, EACH CALL PRØCESSING THE
C PART ØF THE PICTURE AT U=NU+1-I WHERE I IS THE NUMBER ØF
C THE CALL TØ DANDR.  THAT IS, THE PART ØF THE PICTURE AT
C U=NU IS PRØCESSED DURING THE FIRST CALL, THE PART ØF THE
C PICTURE AT U=NU-1 IS PRØCESSED DURING THE SECØND CALL, AND
C SØ ØN UNTIL THE PART ØF THE PICTURE AT U=1 IS PRØCESSED
C DURING THE LAST CALL.
C NV     SEE INIT3D CØMMENTS.
C NW     SEE INIT3D CØMMENTS.
C ST1    SEE INIT3D CØMMENTS.
C LX     THE NUMBER ØF WØRDS NEEDED TØ HØLD NX BITS.  ALSØ,
C        THE FIRST DIMENSIØN ØF IS2.
C NX     NUMBER ØF CELLS IN THE X DIRECTIØN ØF A MØDEL ØF THE
C        IMAGE PLANE.  A SILHØUETTE ØF THE PARTS ØF THE PIC-
C        TURE PRØCESSED SØ FAR IS STØRED IN THIS MØDEL. LINES
C        TØ BE DRAWN ARE TESTED FØR VISIBILITY BY EXAMINING
C        THE SILHØUETTE.  LINES IN THE SILHØUETTE ARE HIDDEN.
C        LINES ØUT ØF THE SILHØUETTE ARE VISIBLE.  THE SØLU-
C        TIØN IS APPRØXIMATE BECAUSE THE SILHØUETTE IS NØT
C        FØRMED EXACTLY.  SEE IS2 CØMMENT BELØW.
C NY     NUMBER ØF CELLS IN THE Y DIRECTIØN ØF THE MØDEL ØF
C        THE IMAGE PLANE.  ALSØ THE SECØND DIMENSIØN ØF IS2.
C IS2    AN ARRAY TØ HØLD THE IMAGE PLANE MØDEL.  IT IS
C        DIMENSIØNED LX BY NY.  THE MØDEL IS NX BY NY AND
C        PACKED DENSELY.  IF HIDDEN LINES ARE DRAWN, DECREASE
C        NX AND NY (AND LX IF PØSSIBLE).  IF VISIBLE LINES
C        ARE LEFT ØUT ØF THE PICTURE, INCREASE NX AND NY (AND
C        LX IF NEED BE).  AS A GUIDE, SØME EXAMPLES SHØWING
C        SUCCESSFUL CHØICES ARE LISTED
C           GIVEN  NU  NV  NW    RESULTING NX  NY FRØM TESTING
C                 100 100  60              200 200
C                  60  60  60              110 110
C                  40  40  40               75  75
C IU     SEE INIT3D CØMMENTS.
C IØBJS  A NV BY NW ARRAY (WITH ACTUAL FIRST DIMENSIØN MV IN
C        THE CALLING PRØGRAM) DESCRIBING THE ØBJECT.  IF THIS
C        IS CALL NUMBER I TØ DANDR, THE PART ØF THE PICTURE
C        AT U=NU+1-I IS TØ BE PRØCESSED.  IØBJS DEFINES THE
C        ØBJECTS TØ BE DRAWN IN THE FØLLØWING MANNER --
C        IØBJS(J,K)=1 IF ANY ØBJECT CØNTAINS THE PØINT
C        (NU+1-I,J,K) AND IØBJS(J,K)=0 ØTHERWISE.
C MV     ACTUAL FIRST DIMENSIØN ØF IØBJS IN THE CALLING
C        PRØGRAM.
C************** MACHINE DEPENDANT CØNSTANTS ****************
C NBPW NUMBER ØF BITS PER WØRD
C MASK AN ARRAY NBPW LØNG.  MASK(I)=2**(I-1), I=1,2,...,NBPW
C CDC 6000 ØR 7000 VERSIØN
      DIMENSIØN MASK(60)
      DATA NBPW/60/
      DATA MASK/1B,2B,4B,10B,20B,40B,100B,200B,400B,1000B,
     * 2000B,4000B,10000B,20000B,40000B,100000B,200000B,
     * 400000B,1000000B,2000000B,4000000B,10000000B,
     * 20000000B,40000000B,100000000B,200000000B,400000000B,
     * 1000000000B,2000000000B,4000000000B,10000000000B,
     * 20000000000B,40000000000B,100000000000B,
     * 200000000000B,400000000000B,1000000000000B,
     * 2000000000000B,4000000000000B,10000000000000B,
     * 20000000000000B,40000000000000B,100000000000000B,
     * 200000000000000B,400000000000000B,1000000000000000B,
     * 2000000000000000B,4000000000000000B,
     * 10000000000000000B,20000000000000000B,
     * 40000000000000000B,100000000000000000B,
     * 200000000000000000B,400000000000000000B,
     * 1000000000000000000B,2000000000000000000B,
     * 40000000Q0Q00000000B,10000000000000000000B,
     * 20000000000000000000B,40000000000000000000B/
      ASSIGN 120 TØ IRET
C RX AND RY ARE USED TØ MAP PLØTTER CØØRDINATES INTØ THE
C IMAGE PLANE MØDEL.
      RX = (FLØAT(NX)-1.)/(S(2)-S(1))
      RY = (FLØAT(NY)-1.)/(S(4)-S(3))
C READ THE RELATIVE PLØTTER CØØRDINATES ØF THE LATTICE
C PØINTS FRØM UNIT IU.
      READ (IU) ST1
C DX, DY AND DZ ARE USED TØ FIND REQUIRED CØØRDINATES ØF
C NØN-LATTICE PØINTS.
      NVD2 = NV/2
      NWD2 = NW/2
      DX = (ST1(NV,NWD2,1)-ST1(1,NWD2,1))*.5/(FLØAT(NV)-1.)
      DY = (ST1(1,NWD2,2)-ST1(NV,NWD2,2))*.5/(FLØAT(NV)-1.)
      DZ = (ST1(NVD2,NW,2)-ST1(NVD2,1,2))*.5/(FLØAT(NW)-1.)
C SLØPE IS USED TØ DEFØRM THE IMAGE PLANE MØDEL SØ THAT
C LINES ØF CØNSTANT Y ØF THE IMAGE MØDEL HAVE THE SAME
C SLØPE AS LINES ØF CØNSTANT U AND W IN THE PICTURE.  THIS
C IMPRØVES THE PICTURE.
      SLØPE = DY/DX
C THE FØLLØWING LØØPS THRØUGH STATEMENT 130 GENERATE THE .5
C CØNTØUR LINES IN 2-SPACE FØR THE ARRAY IØBJS (WHICH CØN-
C TAINS ØNLY ZERØES AND ØNES), TESTS THE LINES FØR VISIBIL-
C ITY, AND CALLS A RØUTINE TØ PLØT THE VISIBLE LINES.
      DØ 130 I=2,NV
        JUMP = IØBJS(I-1,1)*8 + IØBJS(I,1)*4 + 1
        DØ 120 J=2,NW
          X = ST1(I,J,1)
          Y = ST1(I,J,2)
C DECIDE WHICH ØF THE 16 PØSSIBILITIES THIS IS.
          JUMP = (JUMP-1)/4 + IØBJS(I-1,J)*8 + IØBJS(I,J)*4 + 1
          GØ TØ (120,20,40,50,70,80,30,100,100,10,80,70,50,40,
     *    20,120),JUMP
C GØING TØ 10 MEANS JUMP=10 WHICH MEANS ØNLY THE LØWER-RIGHT
C AND UPPER-LEFT ELEMENTS ØF THIS CELL ARE SET TØ 1.
C TWØ LINES SHØULD BE DRAWN, A DIAGØNAL CØNNECTING THE
C MIDDLE ØF THE BØTTØM TØ THE MIDDLE ØF THE RIGHT SIDE ØF
C THE CELL (LØWER-RIGHT LINE), AND A DIAGØNAL CØNNECTING THE
C MIDDLE ØF THE LEFT SIDE TØ THE MIDDLE ØF THE TØP (UPPER-
C LEFT LINE) ØF THE CELL.
   10     ASSIGN 90 TØ IRET
C LØWER-RIGHT LINE
   20     X1 = X
          Y1 = Y - DZ
          X2 = X + DX
          Y2 = Y - DY
          GØ TØ 110
C LØWER-LEFT AND UPPER-RIGHT
   30     ASSIGN 60 TØ IRET
C LØWER-LEFT
   40     X1 = X
          Y1 = Y - DZ
          X2 = X - DX
          Y2 = Y + DY
          GØ TØ 110
C HØRIZØNTAL
   50     X1 = X + DX
          Y1 = Y - DY
          X2 = X - DX
          Y2 = Y + DY
          GØ TØ 110
C UPPER-LEFT
   60     ASSIGN 120 TØ IRET
   70     X1 = X + DX
          Y1 = Y - DY
          X2 = X
          Y2 = Y + DZ
          GØ TØ 110
C VERTICAL
   80     X1 = X
          Y1 = Y - DZ
          X2 = X
          Y2 = Y + DZ
          GØ TØ 110
   90     ASSIGN 120 TØ IRET
C UPPER-LEFT
  100     X1 = X - DX
          Y1 = Y + DY
          X2 = X
          Y2 = Y + DZ
C TEST VISIBILITY ØF THIS LINE SEGMENT.
  110     IX = (X1-S(1))*RX
          IY=MØD(IFIX((Y1-S(3))*RY-SLØPE*FLØAT(IX))+NY,NY)+1
          IBIT = MØD(IX,NBPW) + 1
          IX = IX/NBPW + 1
C *********** .AND. USED AS A MASKING ØPERATØR *************
          IV=IS2(IX,IY).AND.MASK(IBIT)
C IF EITHER END ØF THE LINE IS AT A MARKED SPØT ØN THE IMAGE
C PLANE MØDEL, THE LINE IS HIDDEN
          IF (IV.NE.0) GØ TØ IRET, (60,90,120)
          IX = (X2-S(1))*RX
          IY=MØD(IFIX((Y2-S(3))*RY-SLØPE*FLØAT(IX))+NY,NY)+1
          IBIT = MØD(IX,NBPW) + 1
          IX = IX/NBPW + 1
C *********** .AND. USED AS A MASKING ØPERATØR *************
          IV=IS2(IX,IY).AND.MASK(IBIT)
          IF (IV.NE.0) GØ TØ IRET, (60,90,120)
C *************** UNDEFINED EXTERNAL REFERENCE *************
C SUBRØUTINE LINE(X1,Y1,X2,Y2) IS ASSUMED TØ DRAW A LINE
C FRØM (X1,Y1) TØ (X2,Y2)
          CALL LINE(X1, Y1, X2, Y2)
          GØ TØ IRET, (60,90,120)
  120   CØNTINUE
  130 CØNTINUE
C CØDE THRØUGH STATEMENT 150 CREATES AN APPRØXIMATIØN ØF
C THE SILHØUETTE ØF THE PART ØF THE PICTURE JUST DRAWN BY
C MARKING THE IMAGE PLANE MØDEL WHERE THE ØBJECT ØCCURS.
      DØ 150 I=1,NV
        DØ 140 J=1,NW
          IF (IØBJS(I,J).EQ.0) GØ TØ 140
          IX = (ST1(I,J,1)-S(1))*RX + 0.5
          TWK = SLØPE*FLØAT(IX) - 0.5
          IY=MØD(IFIX((ST1(I,J,2)-S(3))*RY-TWK)+NY,NY)+1
          IBIT = MØD(IX,NBPW) + 1
          IX = IX/NBPW + 1
C ************ .ØR. USED AS A MASKING ØPERATØR *************
          IS2(IX,IY)=IS2(IX,IY).ØR.MASK(IBIT)
  140   CØNTINUE
  150 CØNTINUE
      RETURN
      END
```

Remark on Algorithm 179 [S14]
Incomplete Beta Ratio
[Oliver G. Ludwig, *Comm. ACM 6* (June 1963), 314]

Nancy E. Bosten and E.L. Battiste [Recd. 1 Sept. 1972 and 15 Mar. 1973]
IMSL, Suite 510/6200 Hillcroft, Houston, TX 77036

Description

Algorithm 179 (modified to include the remark by M.C. Pike and I.D. Hill [1]) computes the Incomplete Beta Ratio using this equation

$$I_x(p,q) = \frac{INFSUM \cdot x^p \cdot \Gamma(PS+p)}{\Gamma(PS)\cdot\Gamma(p+1)} + \frac{x^p\cdot(1-x)^q\cdot\Gamma(p+q)\,FINSUM}{\Gamma(p)\cdot\Gamma(q+1)}$$

INFSUM and *FINSUM* represent two series summations defined as follows:

$$INFSUM = \sum_{i=0}^{\infty} \frac{(1-PS)_i \cdot p}{p+i}\,\frac{x^i}{i!}, \text{ where}$$

$$\begin{aligned}(1-PS)_i &= 1 & [i=0]\\ &= (1-PS)\cdot(2-PS)\cdots(i-PS) = \frac{\Gamma(1+i-PS)}{\Gamma(1-PS)} & [i>0]\end{aligned}$$

$$\text{and } FINSUM = \sum_{i=1}^{[q]} \frac{q\cdot(q-1)\cdots(q-i+1)}{(p+q-1)(p+q-2)\cdots(p+q-i)}\,\frac{1}{(1-x)^i},$$

where $[q]$ is equal to the largest integer less than q. If $[q] = 0$, then $FINSUM = 0$. PS is defined as

$$\begin{aligned}PS &= 1, \text{ if } q \text{ is an integer; otherwise}\\ &= q - [q].\end{aligned}$$

By rearranging Algorithm 179 so that scaling can be introduced, the argument range of p and q can be extended and accuracy can be improved.

Since $I_x(p,q)$ is a probability and, therefore, bounded [0, 1], and *INFSUM* and *FINSUM* are series having only positive terms, we see that $I_x(p,q)$ is a collection of terms all of which are positive and bounded in the range [0, 1] if: (1) each term of *INFSUM* is multiplied by $(x^p \cdot \Gamma(PS+p))/(\Gamma(PS)\cdot\Gamma(p+1))$; and (2) each term of *FINSUM* is multiplied by $(x^p\cdot(1-x)^q\cdot\Gamma(p+q))/(\Gamma(p)\cdot\Gamma(q+1))$.

Knowing this fact, we can apply a scaling procedure to the algorithm. *INFSUM* is a decreasing series. If the product of the first term of *INFSUM* and its multiplicative factor would underflow, then the sum of this series could be set to zero and all calculations involving underflow could be avoided. This is handled in the modification of the algorithm given below. However, since *INFSUM* is a decreasing series, underflows may occur later in the calculations. No attempt has been made to handle them here.

The second summation is more complicated. The series is decreasing if $q/((q+p-1)(1-x))$ is less than 1. If an individual term becomes less than $1.E$-6 times the previous sum, calculation can be legitimately terminated since no additivity is apparent. If a term of the decreasing series is less than an arbitrarily small constant (*EPS*2), calculation is also terminated. This is done to prevent underflows in the later terms.

If the series is increasing, the first terms may underflow. In this case a power of ϵ_1 (machine precision $-$ $1.E$-78 on the IBM 360/370) may be factored from each term in *FINSUM* (times its multiplier). These terms cannot be added to the sum since they are less than machine precision; however, they are useful in retaining the accuracy of the initial terms, which are then used recursively. By the nature of the problem, we know that any term in *FINSUM*, times its multiplier, must be less than or equal to 1, but we have factored out powers of ϵ_1. Therefore, if a term of *FINSUM* becomes greater than 1, we know that rescaling, by multiplying the term by ϵ_1, is in order.

Testing on the IBM 360/195 has shown that, by rearranging the calculations of the original Algorithm 179, and thus including scaling, the input range of the algorithm can be greatly extended with a high degree of accuracy.

MDBETA requires a double precision function *DLGAMA* which computes the log of the gamma function. ACM Algorithm 291 may be used. *MDBETA* was tested against the SSP routine *BDTR* given in the manual *System/360 Scientific Subroutine Package (360A-CM-03X) Version III Programmer's Manual*, H20-0205. *MDBETA* ran 3.5 times faster than *BDTR* with greater accuracy. For example, in the case $x = .5$, $p = 2000$ and $q = 2000$, *MDBETA* gave the correct result, .5, while *BDTR* gave an answer of .497026. The IMSL subroutine, *MDBIN*, was used for an additional comparison when p and q are integers. *MDBIN* maintains IBM 370/360 single precision accuracy (approximately six significant digits). Over the tests performed the maximum difference occurred in the fifth significant digit when p and q were less than 200. Three to four significant digits of accuracy can be expected with p and q as large as 2000.

Acknowledgments. The above ideas are the application of ideas learned from the late Hirondo Kuki. Routine *MDBETA* originated from a code which resides in IMSL Library 1. We thank Wayne Fullerton, from the University of California, Los Alamos Scientific Laboratory, for refereeing the paper.

Algorithm

```
      SUBROUTINE MDBETA(X, P, Q, PROB, IER)
C FUNCTION                  - INCOMPLETE BETA PROBABILITY
C                             DISTRIBUTION FUNCTION
C USAGE        - CALL MDBETA (X,P,Q,PROB,IER)
C PARAMETERS
C   X     - VALUE TO WHICH FUNCTION IS TO BE INTEGRATED. X
C              MUST BE IN THE RANGE (Ø,1) INCLUSIVE.
C   P     - INPUT (1ST) PARAMETER (MUST BE GREATER THAN Ø)
C   Q     - INPUT (2ND) PARAMETER (MUST BE GREATER THAN Ø)
C   PROB  - OUTPUT PROBABILITY THAT A RANDOM VARIABLE FROM A
C              BETA DISTRIBUTION HAVING PARAMETERS P AND Q
C              WILL BE LESS THAN OR EQUAL TO X.
C   IER   - ERROR PARAMETER.
C              IER = Ø INDICATES A NORMAL EXIT
C              IER = 1 INDICATES THAT X IS NOT IN THE RANGE
C                (Ø,1) INCLUSIVE.
C              IER = 2 INDICATES THAT P AND/OR Q IS LESS THAN
C                OR EQUAL TO Ø.
      DOUBLE PRECISION PS, PX, Y, P1, DP, INFSUM, CNT, WH, XB,
     * DQ, C, EPS, EPS1, ALEPS, FINSUM, PQ, D4, EPS2, DLGAMA
C DOUBLE PRECISION FUNCTION DLGAMA
C MACHINE PRECISION
      DATA EPS/1.D-6/
C SMALLEST POSITIVE NUMBER REPRESENTABLE
      DATA EPS1/1.D-78/
C NATURAL LOG OF EPS1
      DATA ALEPS/-179.6Ø16DØ/
C ARBITRARILY SMALL NUMBER
      DATA EPS2/1.D-5Ø/
C CHECK RANGES OF THE ARGUMENTS
      Y = X
      IF ((X.LE.1.Ø) .AND. (X.GE.Ø.Ø)) GO TO 1Ø
      IER = 1
      GO TO 14Ø
   1Ø IF ((P.GT.Ø.Ø) .AND. (Q.GT.Ø.Ø)) GO TO 2Ø
      IER = 2
      GO TO 14Ø
   2Ø IER = Ø
      IF (X.GT.Ø.5) GO TO 3Ø
      INT = Ø
      GO TO 4Ø
C SWITCH ARGUMENTS FOR MORE EFFICIENT USE OF THE POWER
C SERIES
   3Ø INT = 1
      TEMP = P
      P = Q
      Q = TEMP
      Y = 1.DØ - Y
   4Ø IF (X.NE.Ø. .AND. X.NE.1.) GO TO 6Ø
C SPECIAL CASE - X IS Ø. OR 1.
   5Ø PROB = Ø.
      GO TO 13Ø
   6Ø IB = Q
      TEMP = IB
      PS = Q - FLOAT(IB)
      IF (Q.EQ.TEMP) PS = 1.DØ
      DP = P
      DQ = Q
      PX = DP*DLOG(Y)
      PQ = DLGAMA(DP+DQ)
      P1 = DLGAMA(DP)
      C = DLGAMA(DQ)
      D4 = DLOG(DP)
      IF (Y.GT.EPS) GO TO 7Ø
C SPECIAL CASE - X IS CLOSE TO Ø. OR 1.
      XB = PX + PQ - D4 - P1 - C
      IF (XB.LE.ALEPS) GO TO 5Ø
      PROB = DEXP(XB)
      GO TO 13Ø
C DLGAMA IS A FUNCTION WHICH CALCULATES THE DOUBLE
C PRECISION LOG GAMMA FUNCTION
   7Ø XB = PX + DLGAMA(PS+DP) - DLGAMA(PS) - D4 - P1
C SCALING
      IB = XB/ALEPS
      INFSUM = Ø.DØ
```

```
C FIRST TERM OF A DECREASING SERIES WILL UNDERFLOW
      IF (IB.NE.0) GO TO 90
      INFSUM = DEXP(XB)
      CNT = INFSUM*DP
C CNT WILL EQUAL DEXP(TEMP)*(1.D0-PS)I*P*Y**I/FACTORIAL(I)
      WH = 0.0D0
   80 WH = WH + 1.D0
      CNT = CNT*(WH-PS)*Y/WH
      XB = CNT/(DP+WH)
      INFSUM = INFSUM + XB
      IF (XB/EPS.GT.INFSUM) GO TO 80
C DLGAMA IS A FUNCTION WHICH CALCULATES THE DOUBLE
C PRECISION LOG GAMMA FUNCTION
   90 FINSUM = 0.D0
      IF (DQ.LE.1.D0) GO TO 120
      XB = PX + DQ*DLOG(1.D0-Y) + PQ - P1 - DLOG(DQ) - C
C SCALING
      IB = XB/ALEPS
      IF (IB.LT.0) IB = 0
      C = 1.D0/(1.D0-Y)
      CNT = DEXP(XB-FLOAT(IB)*ALEPS)
      PS = DQ
      WH = DQ
      P1 = (PS*C)/(DP+WH-1.D0)
      XB = P1*CNT
      IF (XB.LE.EPS2 .AND. P1.LE.1.D0) GO TO 120
  100 WH = WH - 1.D0
      IF (WH.LE.0.0D0) GO TO 120
      IF (P1.LE.1.D0 .AND. CNT/EPS.LE.FINSUM) GO TO 120
      CNT = (PS*C*CNT)/(DP+WH)
      IF (CNT.LE.1.D0) GO TO 110
C RESCALE
      IB = IB - 1
      CNT = CNT*EPS1
  110 PS = WH
      IF (IB.EQ.0) FINSUM = FINSUM + CNT
      GO TO 100
  120 PROB = FINSUM + INFSUM
  130 IF (INT.EQ.0) GO TO 140
      PROB = 1.0 - PROB
      TEMP = P
      P = Q
      Q = TEMP
  140 RETURN
      END
```

Remark on Algorithm 419 [C2]

Zeros of a Complex Polynomial [M.A. Jenkins and J.F. Traub, *Comm. ACM 15* (Feb. 1972), 97–99]

David H. Withers [Rec. 9 Oct. 1972 and 14 May 1973]
IBM, Essex Junction, VT 04352

The published algorithm has performed satisfactorily for all except one (degenerate) case. When removing zeros at the origin, the algorithm does not stop if all roots have been located. An error will occur if the polynomials, $X^N = 0$ or $a_N = 0$ are given to the algorithm. The difficulty may be avoided by inserting after statement 40 the statement

IF (*NN.EQ.* 1)*RETURN*

The referee pointed out the second type of degenerate case above and two typographical errors:

1. In the initialization of constants section *COSR* should be initialized by *COSR* = −.069756474.
2. In the *FUNCTIONS SCALE* and *CMOD*, the declaration of *DSQRT* as *DOUBLE PRECISION* was accidentally typed as *DSQURT.*

Remark on Algorithm 431 [H]

A Computer Routine for Quadratic and Linear Programming Problems [H] [Arunachalam Ravindran, *Comm. ACM 15* (Sept., 1972), 818]

Arunachalam Ravindran [Recd. 12 Mar. 1973]
School of Industrial Engineering, Purdue University, West Lafayette, IN 47907

A small error has been brought to my notice in this algorithm. The error is in defining the matrix M. It should read as

$$M = \begin{pmatrix} Q+Q' & -A' \\ A & 0 \end{pmatrix}.$$

Graphics and Image Processing — W. Newman, Editor

Scan Conversion Algorithms for a Cell Organized Raster Display

R.C. Barrett
Hughes Aircraft Co.
and
B.W. Jordan Jr.
Northwestern University

Raster scan computer graphics with "real time" character generators have previously been limited to alphanumeric characters. A display has been described which extends the capabilities of this organization to include general graphics.

Two fundamentally different scan conversion algorithms which have been developed to support this display are presented. One is most suitable to noninteractive applications and the other to interactive applications. The algorithms were implemented in Fortran on the CDC6400 computer. Results obtained from the implementations show that the noninteractive algorithms can significantly reduce display file storage requirements at little cost in execution time over that of a conventional raster display. The interactive algorithm can improve response time and reduce storage requirements.

Key Words and Phrases: graphics, scan conversion, raster display , line drawing, discrete image, dot generation, matrix displays

CR Categories: 4.41, 6.35, 8.2

Copyright © 1974, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

This work supported in part by ONR Grant No. N0014-67-A-0356 Mod. AE and by NSF Science Development Grant GU-3851.
Authors' addresses: R.C. Barrett, Computer Applications Department, Hughes Aircraft Company, Culver City, CA 90230; B.W. Jordan Jr., Departments of Computer Sciences and Electrical Engineering, Northwestern University, Evanston, IL 60201.