Southern Illinois University Carbondale
OpenSIUC

Honors Theses | University Honors Program

5-1988

Analysis of Sorting Algorithms

Terry D. Fryar

Follow this and additional works at: http://opensiuc.lib.siu.edu/uhp_theses

Recommended Citation
Fryar, Terry D., "Analysis of Sorting Algorithms" (1988). *Honors Theses.* Paper 230.

This Dissertation/Thesis is brought to you for free and open access by the University Honors Program at OpenSIUC. It has been accepted for inclusion in Honors Theses by an authorized administrator of OpenSIUC. For more information, please contact opensiuc@lib.siu.edu.

CHAPTER 1 - INTRODUCTION

The concept of order is very important to mankind. Order allows man to understand and better live in the world around him. Imagine trying to find a name in a phone book if the names were not in alphabetical order. Computer science is one area where order is especially important. Much of computer science depends on the concept of order. Many commercial and non-commercial programming applications involve sorting items into ascending or descending order. It is precisely the algorithms that perform the task of sorting that we will concentrate on in this report.

All of the algorithms sort items in ascending order. There are nine algorithms covered. Each algorithm was chosen based on simplicity of design, speed, behavior, and interesting properties brought out by analysis. The research concerning these algorithms was broken up into four areas. The first area was the learning of the programming language C.

C is a very good language for this type of research because it offers high-level programming structure with low-level features. The second area involved the coding of each of the algorithms in C. We coded each algorithm to see how difficult it would be to convert the algorithm from a theoretical description to actual working computer code. The naturalness of these algorithms for computer applications is evident. Each of the algorithms was coded as a C function and could be called from the body of the main program. This brings us to the third area, the timing experiment.

The afforementioned algorithmic functions were used to perform the task of sorting integer numbers. The algorithms were timed individually over a wide range and amount of input data in order to obtain an initial insight into the behavior of each. The entire timing program was executed nonstop over a period of five days. All of the output was routed to a disk file and subsequently printed. This timing data was analyzed and was helpful in providing a basis for further understanding of the nature of the algorithms.

The fourth and final phase involved a closer look at the algorithms through the use of mathematical tools of analysis. Each algorithm was mathematically analyzed and the results were compared to the earlier insight provided by the timing data. Conclusions and criticisms are then formulated and presented. This type of research is vital in maintaining a good understanding of how algorithms work and how they may be improved.

CHAPTER 2 - THE ALGORITHMS

Now that the method and reasoning behind the research is clear, let's start with an introduction and explanation of each of the nine sorting algorithms. The first sorting algorithm is affectionately named the "bubble" sort. This sort is perhaps one of the simplest sorts in terms of complexity. It makes use of a sorting method known as the exchange method. This algorithm compares pairs of adjacent elements and makes exchanges if necessary. The name comes from the fact that each element "bubbles" up to it's own proper position. Here is how bubble sort would sort the integer array 4 3 1 2:

pass 1	1 4 3 2
pass 2	1 2 4 3
pass 3	1 2 3 4

The code for bubble sort is on page 8 of the program listing. The outer loop is performed n-1 times (n=number of elements to be sorted) to ensure that, in the worst case, every element is in it's proper position when the loop terminates. If no exchanges take place after a pass, the algorithm terminates since the elements must be in order. Previous research has labeled the bubble sort as the worst sort ever. We shall test this statment later.

The second sorting algorithm is a modified version of the bubble sort known as the shaker sort or cocktail shaker sort. This algorithm is designed so that subsequent passes over the array reverse direction. This way, greatly out of place elements will travel more quickly to their correct position. Notice that this algorithm is essentially the same as the bubble sort except for the reverse direction passes. Any out of place element is exchanged during a pass. The number of passes is the same as for bubble sort. The code for this algorithm is located on page 8 of the program listing.

The third algorithm uses a different method of sorting known as sorting by selection. This selection sort algorithm picks the smallest element from the array and switches it with the first element. It then picks the smallest element from the rest of the array and switches it with the second element. This process is repeated up to the last pair of elements. Here is how it would sort 2 4 1 3:

```
pass 1        1 4 2 3
pass 2        1 2 4 3
pass 3        1 2 3 4
```

The code for this sort is on page 9 of the program listing.

The fourth algorithm uses the insertion method of sorting. This algorithm first sorts the first two elements of the array. It then inserts the third element in it's proper place in relation to the first 2 sorted elements. This process continues until all of the remaining elements are inserted in their proper position. Here is how it would sort 4 3 1 2:

pass 1	3 4 1 2
pass 2	1 3 4 2
pass 3	1 2 3 4

The code for this insertion sort algorithm is located on page 9 of the program listing.

The fifth algorithm, dubbed the shell sort after its' inventor D.L. Shell, is derived from the insertion sort. The shell sort is based on the idea of diminishing increments. Suppose the array to be sorted was 6 4 1 3 2 5. Here is how the shell sort would work with increments 3 2 1:

pass 1	6 4 1 3 2 5
pass 2	3 2 1 6 4 5
pass 3	1 2 3 4 5 6

Notice that in pass 1, elements 3 positions apart are sorted. Then all elements 2 positions apart are sorted. Finally, all adjacent elements are sorted. The increments can be changed, but the last increment must be one. The choice of the set of increments that make the algorithm most efficient had posed very difficult mathematical problems that were solved only recently. The increments 9 5 3 1 seem to work well, so I used them in the experiment.

It has been suggested that the next algorithm is the best sorting algorithm available today. It is named quick sort due to it's speedy sort time. Quick sort is based on the exchange method of sorting, as is bubble sort, but is also uses the idea of partitioning. Quick sort chooses a median value from the array and uses it to partition the array into two subarrays. The left subarray contains all of the elements that are less than the median value and the right subarray contains all of the elements that are greater than the median value. This process is recursively repeated for each subarray until the array is sorted. The median value can be chosen randomly, but I have coded the algorithm to choose the element that is physically in the middle of the array.

This brings up one nasty aspect of quick sort. If the median value chosen is always the smallest or largest element, the algorithm slows down drastically. This usually will not happen however, since most input data is in a random order and the chance of always picking an extreme value is small. Note that this algorithm is naturally recursive and I have coded it as such. Here is how quick sort would sort 6 5 4 1 3 2:

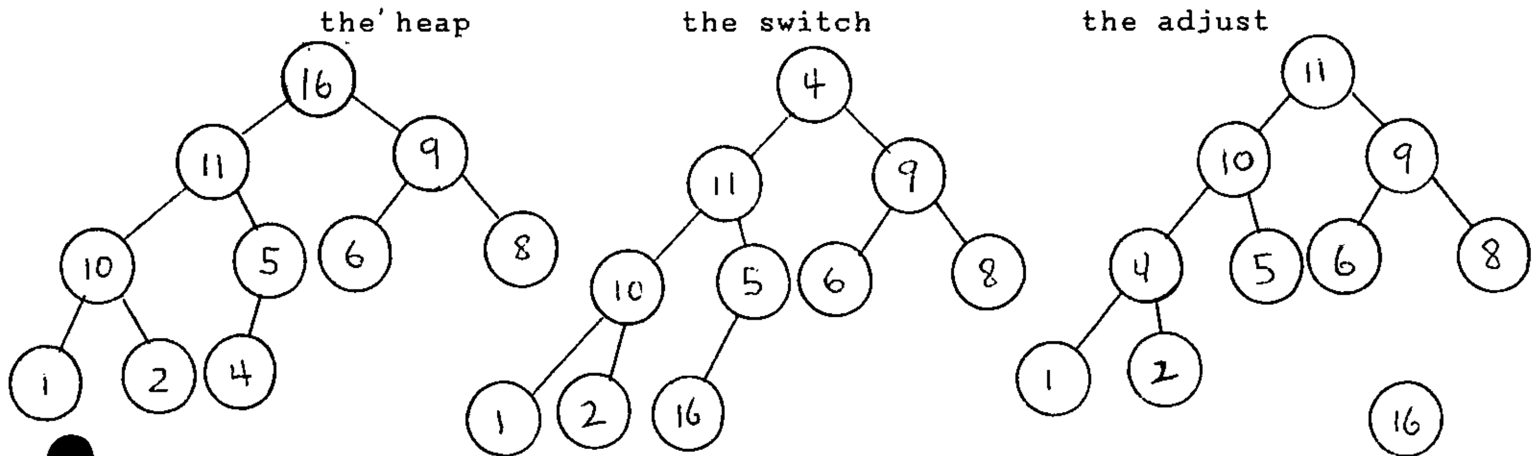
```
pass 1        2 3 1 4 5 6
pass 2        1 2 3  4 5 6
```

Note that after pass 1, the array is partitioned into 2 3 1 and 4 5 6. The process is then repeated for each of these. The code for the quick sort is located on page 10 of the program listing.

The seventh algorithm is known as heap sort. It makes use of a data structure known as a heap. A heap is a complete binary tree organized such that the value of the parent nodes are greater than their children's. With this type of structure, the largest element happens to be the root node. This property of a heap makes it ideal for a sorting algorithm.

The heap sort algorithm first builds a heap with all of the elements. After heap creation, the largest element is located at the root. This root element is switched with the last element at the end of the array. Since the root, which is the largest element, is placed at the end of the array, this largest elememt is now in it's correct position. This position in the array is now off-limits to the algorithm and the rest of the heap is adjusted, starting at the new root, to ensure that all of the parent's values are greater that the values of their children.

This switching and readjusting is performed repeatedly until all of the elements are in their proper position. Here is one pass of heap sort on 16 11 9 10 5 6 8 1 2 4:



The code for heap sort is on page 11 of the program listing.

The next algorithm is an example of divide and conquer. It is called merge sort. Merge sort splits the array into two subarrays, each of almost equal size, and recursively sorts each. The two sorted subarrays are then merged together. The recursive version is very simple and takes full advantage of the power of recursion. Here is an example of how merge sort would sort 5 2 3 1 7: (the [] bars indicate a subarray)

```
[5 2 3] [1 7]
[5 2] [3]
[5] [2]
[2 5]
[2 3 5]
        [1] [7]
        [1 7]
[1 2 3 5 7]
```

Notice how the subarrays are broken down until there is only one element left. Then, two subarrays with only one element each are merged. The merging continues until all left side sorted arrays are merged together. The right side is then broken down and merged. This is a very good example of how recursion can be used to simplify programs. The code for this sort is located on page 12 of the program listing.

The last sort is somewhat different from the other sorts. This sort operates using the internal structure of the elements. The elements used in the experiment were two-byte integers. This sort depends on the binary form of these integers. It is for this reason that this algorithm is called radix sorting, since it uses radix 2 representations of the numbers.

The algorithm starts at the most significant bit. It partitions the elements such that all elements with a 0 bit come before those elements with a 1 bit. The algorithm then shifts to the second bit and the process is repeated. This sort is similar to quick sort in that it uses partitioning and exchange methods. Note that the elements are partitioned into 2 subarrays for each pass. The algorithm then recursively processes each subarray. The code for this sort is on page 13 of the program listing.

CHAPTER 3 - THE EXPERIMENT

Now that the algorithms have been discussed, let's look at the experiment. This experiment has been designed to generate timing data that will provide an insight as to the behavior of each of the algorithms. Each algorithm was executed and timed on 3 different types and 6 different sizes of integer arrays. The value of the integer numbers ranged from 0 to 32767. The first type of array contained numbers already in order. The second type contained numbers in reverse order. The third contained numbers in random order as generated by a random number generator. The seed for the random number generator was changed constantly in order to ensure that the numbers were as random as possible.

Each algorithm was executed and timed on varying sizes of arrays. Note that each size of reverse order and inorder (numbers already in order) arrays contained the same sequence (i.e. 1 2 3 4 5) of numbers for each execution. Since this method prevented any timing discrepancies from being introduced into the experiment, I decided to execute and time the algorithms only once for each size of inorder and reverse order array. In order to obtain a fair representative time for the random order arrays, it was necessary to execute and time each algorithm more than once for each size of the random arrays since the sequences of numbers would change. Figure 5 is a table that sums up the exact parameters of the experiment.

Before beginning to analyze the data, we would like to describe a few of the criterion used for judging an algorithm. The behavior of an algorithm is one important criterion. Behavior refers to how hard an algorithm works depending on how ordered the array is initially. An algorithm exhibits natural behavior when it works least on an array that is already sorted and hardest on an array that is in inverse order. An algorithm exhibits unnatural behavior when it works more on a list that is already in order and less on a list that is in inverse order. Depending on the application, natural behavior may or may not be better than unnatural behavior. Natural behavior is usually preferred, however.

Perhaps the most important aspect of a sorting algorithm is how fast it can sort an random case array. Pure speed is sometimes the only factor in choosing a sorting algorithm. Since processing time is an expensive and sometimes limited resource, being quick is a very important characteristic of a sorting algorithm. We shall examine the random case timing data and attempt to determine which sort is best. To aid us in our examination of the data, we have included three graphs which illustrate the relationships of the algorithms in terms of performance. Figure 1 is a graph of the random timing data for the slower sorts. Figure 2 is a graph of the random timing data for the faster sorts. Figure 3 is a graph of all of the sorts.

We would like to note that all of the timing data is summarized in a table in Figure 4. The timing data as generated by the program is included at the end of the program listing.

Let us begin our analysis with perhaps the slowest sort ever conceived by man. The bubble sort is relatively very slow when compared to some of the faster sorts. It took 2394 seconds to sort the reverse array of size 10000 and 1646.5 seconds to sort the random array. It ranks last among all of the other eight sorts in terms of pure speed. Note that the bubble sort seems to exhibit natural behavior since it took one second or less in sorting the inorder case of size 10000 and 2394 seconds to sort the reverse order array. It works least when the list is ordered and most when it is in reverse order. This would indicate that bubble sort would be used where the list to be sorted is almost in order.

An interesting property of the timing data for bubble sort is the manner in which the timings grow in size. On graph 1 I have included a scaled representation of an n squared curve where the time rises exponentially as n increases. Looking at the increase in size of the times and at the shape of the bubble sort curve, it would appear that this sort runs in Order(n**2) or O(n**2) time. Since the times rise quickly, this sort would be very inappropriate for large amounts of data.

Let's move on to the shaker sort. It only beats bubble sort by 4 seconds in the reverse case of size 10000. It also takes less than 1 second to sort the inorder case, indicating that it is also exhibiting natural behavior. The time for the random case of size 10000 only took 1646.5 seconds, 747.5 seconds faster than bubble sort.

This would make it better than bubble sort for larger amounts of data. The shape of the curve for the shaker sort is also very close to the n squared curve. The timings also rise very quickly. This suggests that the shaker sort is O(n**2) also. Even though it is slightly faster than bubble sort, the shaker sort is still too slow to be used to sort a large amount of data.

The time for select sort on a reverse order array of size 10000 is 984 seconds while the time for an inorder case of the same size is 757 seconds. The time for a random case of size 10000 is 758 seconds. Since the algorithm worked only 1 second more for an random case than for the reverse order case, the select sort algorithm is almost exhibiting unnatural behavior. Since there is a 603.5 second difference between the shaker sort and the select sort, the faster select sort algorithm would be ranked ahead of the shaker sort and the bubble sort for large amounts of data. Since the shape of the select sort curve is slightly flatter than the n**2 curve, it is difficult to say for certain that the select sort algorithm is running in O(n**2) time. However, the abrupt rise of the timing data seems to confirm this.

The final slow sort that we will examine is the insert sort. The time for the reverse order case of size 10000 is 1339 seconds, which is 581 seconds slower than select sort. Insert sort does, however, take under 1 second for the inorder case. Since the random case time is well below the reverse case, this algorithm is exhibiting natural behavior. The time for the random case of size 10000 is 672.5, faster than any of the sorts discussed so far.

The insert sort's low random case time and high reverse case time suggests that it might be good for a large number of elements that are almost already in order.

We now move to the fast sorts. These sorts are so named because they are extremely fast in comparison with the sorts examined so far. In fact, the fastest of the fast sorts sorted an random case array of 20000 elements in 8.5 seconds, 79 times faster than the quickest of the slow sorts on an array half the size (10000).

Let's begin with the shell sort. This sort required 677 seconds to sort a reverse order array of size 20000. It only took 5 seconds to sort the reverse order case of size 20000. Since the time for the random case is right in between the reverse order and inorder times, it is exhibiting very natural behavior. Note how the timings increase in Figure 4.

It certainly is not rising in a linear fashion. It does not appear to be O(n**2) either, leaving us to conclude that the order of magnitude is in between these two. The shell curve in Figure 2 rises abruptly, interrupting the smooth flow of the curve. Perhaps a larger amount of timing data would complete the curve better. Nevertheless, the curve is sharper than the n log n curve that is plotted alongside. This affirms our suspicions of the upper and lower bounds. It is operation somewhere between n log n and n**2.

We now come to the fastest sort of the bunch. This speedy algorithm, known as quick sort, is the fastest of all nine sorts in all three cases. It only took 6 seconds to sort the reverse case, 5 seconds for the inorder case, and 8.5 for the random case, all of size 20000.

While this sort may not seem to exhibit natural behavior, it is certainly recommended when the number of elements to sort is large. Since it is also quick when n (number of elements) is small, it can also be used for small sorting jobs but some overhead is created by the recursive calls. Note that it closely resembles the n log n curve in Figure 2. Also note that it's curve is the flattest and lowest of any of the sorts.

The next sort is the heap sort. This was the second slowest of the fast sorts for the random case. At 20 seconds, it was 11.5 seconds slower than quick sort. The times for reverse and inorder cases of size 20000 were 18 and 35 seconds, respectively. This sort exhibits natural behavior. Note that the heap sort curve is also very similar to the n log n curve.

The next sort generated an interesting set of timings. The merge sort took 15 seconds to do each of the three 20000 element cases. This is not natural behavior since the algorithm works the same no matter what order the elements are in. The merge curve is also similar to the n log n curve, but is flatter than the heap sort curve.

The last sort is the radix sort. It is second only to quick sort in all of the three cases of size 20000. It took 9, 8, and 10 seconds for the reverse order, inorder, and random cases, respectively. It does not exhibit natural behavior in this experiment since it took longer for the random case than for the reverse order case. It is, however, a very fast sort. It's curve is close to the quick sort curve in terms of shape and position.

The best sort in terms of the experiment seems to be quick sort, but we will further investigate in the next chapter.

CHAPTER 4 - MATHEMATICAL ANALYSIS

Now that we have a small insight into the performance and behavior of the algorithms, let's try to gain a more complete understanding by using mathematics. Please note that some of the algorithms present difficult mathematical problems and will be difficult to analyze. We will not go into a great deal of detail if this is the case. We will start with the bubble sort algorithm. To mathematically analyze this and the other algorithms, we will count the number of comparisons required by each algorithm. Although there is a count of the number of exchanges, we will gain sufficient insight with just the use of the comparison count. The magnitude of this count will give us a clear picture of why the algorithms behave like they do.

Now, let's look at the bubble sort algorithm. The inner loop of bubble sort will execute n/2 times (n=number of elements to sort) since every pass will bubble up an element into it's proper position and out of place elements are exchanged. Only n/2 comparisons are needed because the list will be ordered when at most half of the elements have been exchanged (worst case). In the random case, we assume that the outer loop will execute approximately n-1 times. This ensures that all elements will be sorted. We then have n/2(n-1) or 1/2(n**2-n) comparisons. The number of comparisons for the best case is n-1, since the algorithm terminates when no exchanges have been made. The order of magnitude of the number of comparisons is n**2, so the algorithm is said to operate in O(n**2) time. This algorithm is slow when n is large, even when ignoring the number of exchanges.

The shaker sort is simply a modified version of bubble sort where the order of magnitude of the number of comparisons is still O(n**2) and the number of exchanges is reduced only by a small amount. It is almost as slow as bubble sort, therefore, and is not recommended for large n.

The selection sort also has an outer loop that executes n-1 times and an inner loop that executes n/2 times. Again we have 1/2(n**2-n) comparisons which makes selection sort an O(n**2) sort. This sort also is slow for large n.

The number of comparisons in the insertion sort algorithm depends on how the list is ordered before it is sorted. If the list is in reverse order, we have 1/2(n**2-n)+1 comparisions since the outer loop executes n-1 times and the inner loop n/2+1 times. The number of comparisons for the random case is better, but even though the number may be small, the number of moves can be a problem since the array is constantly being shifted.

Each one of the above sorts is basically too slow to use since the execution time is directly affected by the number of elements. The faster sorts are used more often since they are not O(n**2) (very slow when n becomes large).

The shell sort is very difficult to analyze and we shall suffice by noting that it has been shown mathematically that the execution time is proportional to n**1.2. This affirms our earlier suspicions about the upper and lower bounds on the performance.

We now come to the fastest sort, quick sort. In the worst case when each of the median values is at an extreme, quick sort is slowed to an O(n**2) sort. Each level of recursion will require that the partioning loops make O(k) comparisons where k is the total number of elements recursively partitioned i.e. at level one, r=n and at level two, r=n-1 etc. Therefore, the number of comparisons is a sum on k where k varies from n or O(n**2). The random case, however, is of O(n log n).

It can be shown [1] that if Ca(n) denotes the average number of comparisons to sort n elements, then the following recurrence relation holds:

$$C_A(n) = n + 1 + \frac{1}{n}\sum_{1\le k\le n}\left(C_A(k-1) + C_A(n-k)\right)$$

n+1 is the number of comparisons needed for the partitioning loops on the first partition. This recurrence can be solved as follows: $nC_A(n) = n(n+1) + 2\left(C_A(0) + C_A(1) + \ldots + C_A(n-1)\right)$

$$(n-1)C_A(n-1) = n(n-1) + 2\left(C_A(0) + \ldots + C_A(n-2)\right)$$

$$nC_A(n) - (n-1)C_A(n-1) = 2n + 2C_A(n-1)$$

$$C_A(n)/(n+1) = C_A(n-1)/n + 2/(n+1)$$

$$\begin{aligned} C_A(n)/(n+1) &= C_A(n-2)/(n-1) + 2/n + 2/(n+1) \\ &= C_A(n-3)/(n-2) + 2/(n-1) + 2/n + 2/(n+1) \\ &\vdots \\ &= C_A(1)/2 + 2\sum_{3\le k\le n+1} 1/k \\ &= 2\sum_{3\le k\le n+1} 1/k \end{aligned}$$

Since $\sum_{3\le k\le n+1} 1/k \le \int_3^{n+2} 1/x \, dx < \log_e(n+2)$

We have $C_A(n) < 2(n+1)\log_e(n+2) = O(n \log n)$

We now will look at the heap sort algorithm. In order to analyze heap sort, we will break the analysis into 2 parts. First we will look at the creation of the heap. Then we will examine the switching and adjusting of the heap. In the worst case, each element inserted becomes the root. Since the heap is a complete binary tree, there are 2**i-1 maximum nodes on any level i where $1 <= i <= \lceil \log_2(n+1) \rceil$. The distance to the root for a node on level i is i-1 so we have the worst case time:

$$\sum_{1 \le i \le \lceil \log_2(n+1) \rceil} (i-1)2^{i-1} < \lceil \log_2(n+1) \rceil 2^{\lceil \log_2(n+1) \rceil} = O(n \log n)$$

What is amazing about the random case heap creation is that the time needed is O(n). Since the proof of this is quite complicated, we shall not try it here. Now let's look at the switching and adjusting loop. The loop that switches elements and calls adjust must perform O(n) operations. Since adjust possibly requires O(log n) operations, we have a worst case time of O(n log n).

Now let's look at merge sort. It can be proved [1] that if T(n) is the execution time needed to sort n elements, the following recurrence relation holds:

$$T(n) = a \quad n = 1 \quad a \text{ is a constant}$$
$$= 2T(n/2) + cn \quad c \text{ is a constant}$$

when $n = 2^k$ we have

$$\begin{aligned} T(n) &= 2(2T(n/4) + cn/2) + cn \\ &= 4T(n/4) + 2cn \\ &= 4(2T(n/8) + cn/4) + 2cn \\ &= \dots 2^k T(1) + kcn \\ &= an + cn \log n \end{aligned}$$

if $2^k < n \le 2^{k+1}$ then $T(n) \le T(2^{k+1})$

therefore $T(n) = O(n \log n)$

CHAPTER 5 - CONCLUSION

Remember that any conclusions stated in this report are based entirely on the research and analysis that was performed. It is possible that some margin of error may be present. One conclusion that can be reached by examining all of this research is that quick sort is the best all-around sorting algorithm in use today. One look at Figure 3, which is a graphing of all of the sorting algorithms in terms of performance, suggests that there are four sorts that are fast enough to be practical: heap sort, merge sort, radix sort, and quick sort.

If one wishes to sort a large amount of data that is in relatively random order, use quick sort. It is simply the fastest general sorting algorithm available. If a small amount of data is to be sorted, perhaps heap sort would be best since it creates no overhead (no recursion used) that would slow it down.

The bubble, shaker, select, and insert sorts may be simple and easy to understand and implement, but they are simply too slow for real-life practical applications. All of these O(n**2) algorithms take too much time for large amounts of data. The shell sort algorithm does present some very interesting mathematical problems but when compared to the O(n log n) sorts, it is also too slow. Remember that quick sort does slow down to O(n**2) time for the worst case. Perhaps merge or heap sort would be better for this type of data since both merge and heap sort remain O(n log n) for all types of data (worst, random, best cases).

Still, the slowing of quick sort to O(n**2) happens rarely and it is the fastest of all of the sorts for random case data. Radix sort is almost as fast as quick sort for some data, but it's execution time depends greatly on the size of the number. Since radix sort uses the internal structure of the element being sorted, it is not good for sorting general elements. Quick sort can be used to sort a variety of things, regardless of element structure, and thus is more versatile than radix sort.

Therefore, according to the experimental data and mathematical analysis, quick sort is the best sort. Perhaps advancements in the field of computer science will produce an algorithm that is better than quick sort. Until then, however, quick sort is the winner!

BIBLIOGRAPHY

[1] Aho, Alfred V., John E. Hopcroft, and Jeffrey D. Ullman. The Design and Analysis of Computer Algoritms. Reading: Addison-Wesley, 1974.

[2] Knuth, Donald E. The Art of Computer Programming. Reading: Addison-Wesley, 1973.

[3] Horowitz, Ellis, and Sartaj Sahni. Fundamentals of Computer Algorithms. Potomac: Computer Science Press Inc., 1978.

[4] Schildt, Herbert. Using Turbo C. Berkeley: Osborne McGraw-Hill, 1988.

FIG 1

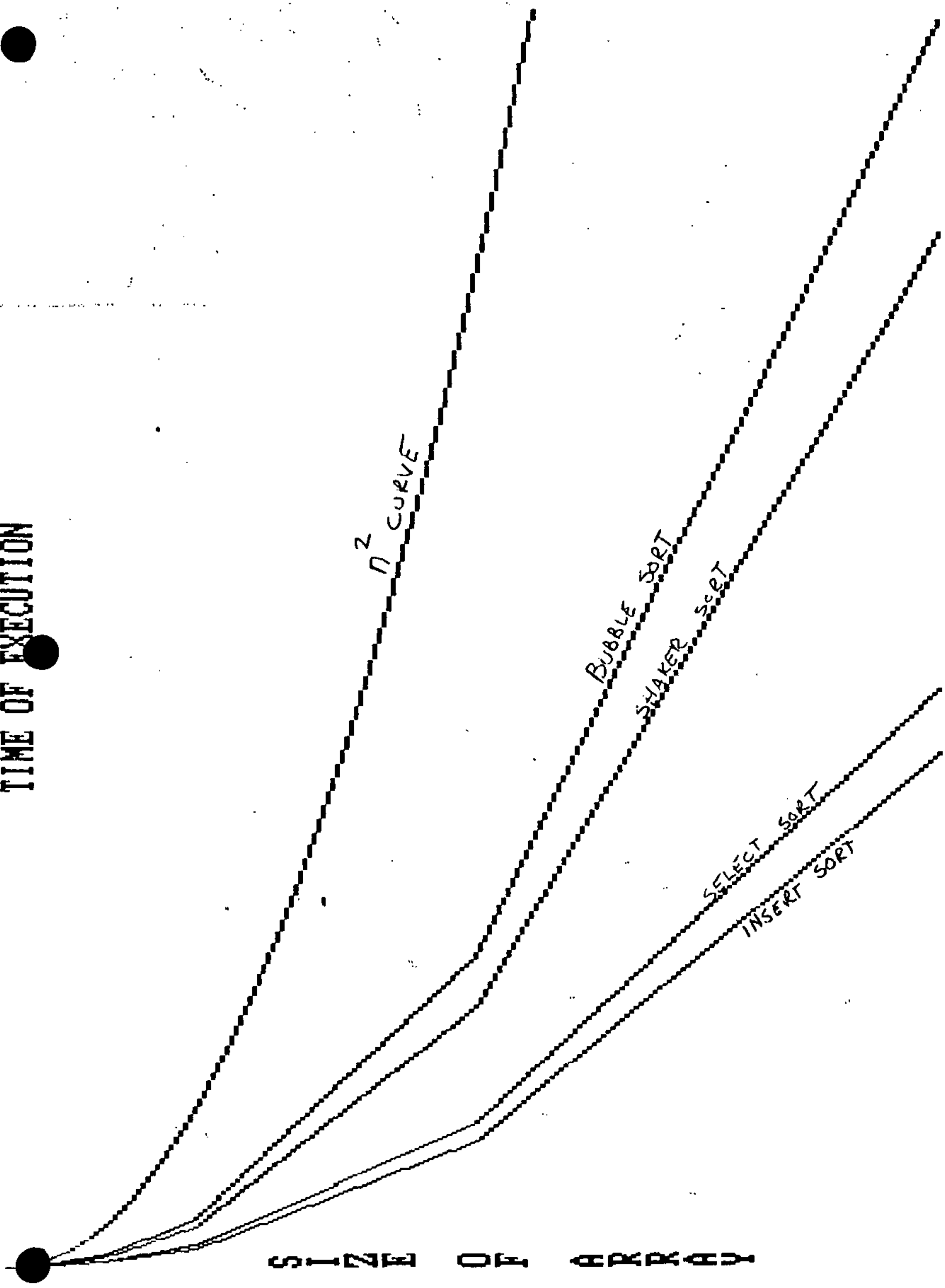


FIG 2

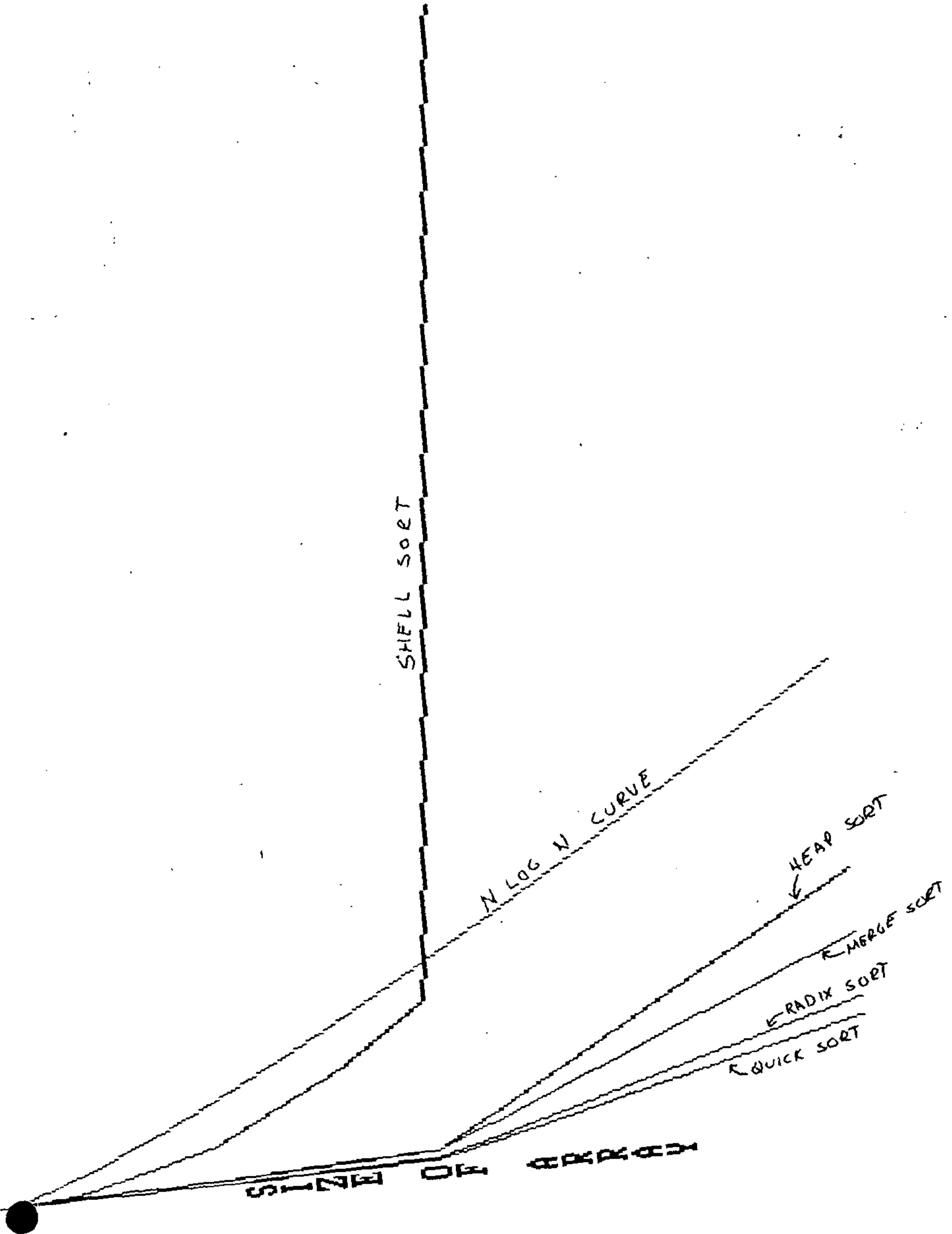


FIG 3

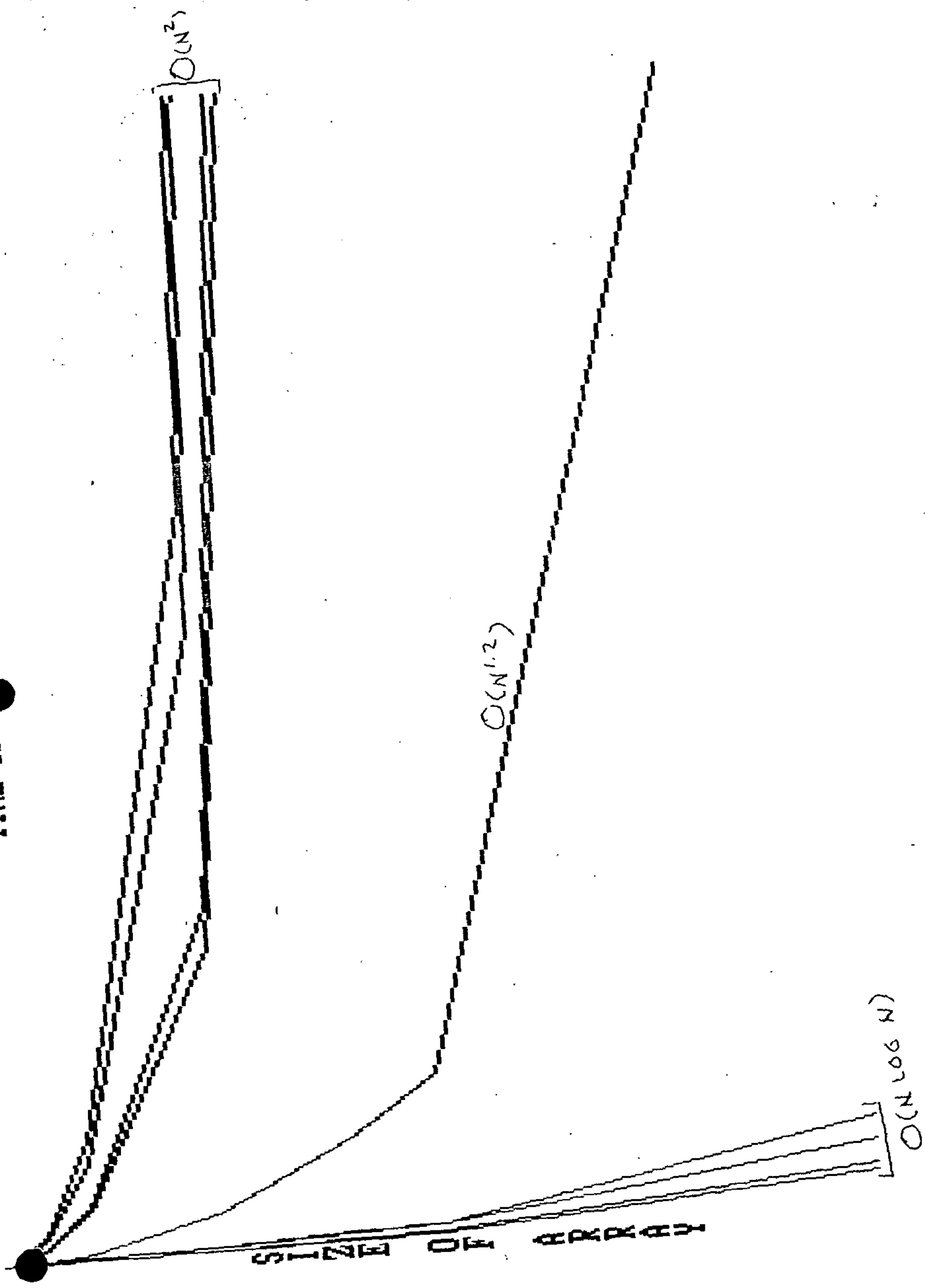


FIG 4

TIMING DATA

Sort	Array Size	Case	Time(in secs)
Bubble	100	reverse	1.0
	500	"	6.0
	1000	"	24.0
	2000	"	95.0
	5000	"	598.0
	10000	"	2394.0
	100	inorder	0.0
	500	"	0.0
	1000	"	0.0
	2000	"	0.0
	5000	"	0.0
	10000	"	1.0
	100	random	0.142857
	500	"	4.0
	1000	"	16.4
	2000	"	65.75
	5000	"	409.3
	10000	"	1646.5
Shaker	100	reverse	1.0
	500	"	6.0
	1000	"	24.0
	2000	"	96.0
	5000	"	597.0
	10000	"	2390.0
	100	inorder	0.0
	500	"	0.0
	1000	"	0.0
	2000	"	0.0
	5000	"	0.0
	10000	"	0.0
	100	random	0.142857
	500	"	3.3
	1000	"	13.4
	2000	"	54.25
	5000	"	340.0
	10000	"	1361.5

TIMING DATA(cont'd)

Sort	Array Size	Case	Time(in secs)
Select	100	reverse	0.0
	500	"	3.0
	1000	"	10.0
	2000	"	39.0
	5000	"	246.0
	10000	"	984.0
	100	inorder	0.0
	500	"	3.0
	1000	"	10.0
	2000	"	39.0
	5000	"	246.0
	10000	"	984.0
	100	random	0.0
	500	"	2.0
	1000	"	7.6
	2000	"	30.25
	5000	"	189.3
	10000	"	758.0
Insert	100	reverse	0.0
	500	"	4.0
	1000	"	13.0
	2000	"	54.0
	5000	"	335.0
	10000	"	1339.0
	100	inorder	0.0
	500	"	0.0
	1000	"	0.0
	2000	"	0.0
	5000	"	0.0
	10000	"	0.0
	100	random	0.142857
	500	"	1.3
	1000	"	6.8
	2000	"	26.75
	5000	"	166.67
	10000	"	672.5
Shell	1000	reverse	2.0
	2000	"	8.0
	5000	"	43.0
	8000	"	109.0
	10000	"	170.0
	20000	"	677.0

TIMING DATA(cont'd)

Sort	Array Size	Case	Time(in secs)
Shell	1000	inorder	1.0
	2000	"	1.0
	5000	"	1.0
	8000	"	2.0
	10000	"	3.0
	20000	"	5.0
	1000	random	1.428571
	2000	"	4.3
	5000	"	23.4
	8000	"	58.5
	10000	"	90.67
	20000	"	353.0
Quick	1000	reverse	0.0
	2000	"	0.0
	5000	"	1.0
	8000	"	2.0
	10000	"	2.0
	20000	"	6.0
	1000	inorder	0.0
	2000	"	1.0
	5000	"	1.0
	8000	"	2.0
	10000	"	2.0
	20000	"	5.0
	1000	random	0.285714
	2000	"	1.0
	5000	"	2.0
	8000	"	3.0
	10000	"	4.0
	20000	"	8.5
Heap	1000	reverse	1.0
	2000	"	1.0
	5000	"	4.0
	8000	"	7.0
	10000	"	8.0
	20000	"	18.0
	1000	inorder	1.0
	2000	"	2.0
	5000	"	8.0
	8000	"	13.0
	10000	"	16.0
	20000	"	35.0

TIMING DATA(cont'd)

Sort	Array Size	Case	Time(in secs)
Heap	1000	random	0.857143
	2000	"	1.5
	5000	"	4.4
	8000	"	7.75
	10000	"	9.0
	20000	"	20.0
Merge	1000	reverse	1.0
	2000	"	1.0
	5000	"	4.0
	8000	"	7.0
	10000	"	8.0
	20000	"	15.0
	1000	inorder	1.0
	2000	"	1.0
	5000	"	3.0
	8000	"	6.0
	10000	"	9.0
	20000	"	15.0
	1000	random	0.857143
	2000	"	1.67
	5000	"	4.2
	8000	"	7.0
	10000	"	9.0
	20000	"	15.0
Radix	1000	reverse	0.0
	2000	"	1.0
	5000	"	2.0
	8000	"	3.0
	10000	"	4.0
	20000	"	9.0
	1000	inorder	0.0
	2000	"	1.0
	5000	"	2.0
	8000	"	3.0
	10000	"	4.0
	20000	"	8.0
	1000	random	0.285714
	2000	"	0.83
	5000	"	2.4
	8000	"	4.0
	10000	"	5.0
	20000	"	10.0

FIG 5

ENVIRONMENT FOR SIMULATION

Bubble, shaker, select, and insert sorts used:

Number of Timing Samples	Size of Array to be Sorted
7	100
6	500
5	1000
4	2000
3	5000
2	10000

Shell, quick, heap, merge, and radix sorts used:

Number of Timing Samples	Size of Array to be Sorted
7	1000
6	2000
5	5000
4	8000
3	10000
2	20000

PROGRAM LISTING

```
/****************************************************************************/
/* AUTHOR :  Terry David Fryar                                             */
/* SSN    :  344-62-3964                                                   */
/* TITLE  :  EXPERIMENTAL ANALYSIS OF VARIOUS SORTING ALGORITHMS.          */
/* DESCRIPTION:  This program performs the task of producing experimental*/
/*               data on the performance and behavior of sorting           */
/*               algorithms.  The algorithms tested are:  Bubblesort,      */
/*               Shakersort, Selectsort, Insertsort, Shell sort,           */
/*               Quicksort, Heapsort, Merge sort, and Radix sort.  These */
/*               are coded in C and are executed on various sizes of       */
/*               arrays of numbers in order, in reverse order, and in      */
/*               random order.  The timing data produced is sent to a      */
/*               file on disk.                                             */
/****************************************************************************/

#include "time.h"
#include "stddef.h"
#include "stdio.h"
#include "dos.h"
#include "stdlib.h"

main()
{

   int sortarray[30000];         /* the array to be sorted */
   int countarray[6];            /* holds the number of elements to be sorted */
   int numpasses[6];             /* holds how many times the sort is to be
                                    performed on one size of sortarray to obtain
                                    an average of timings */
   double timearray[8];          /* holds the timing results */
   time_t start,end;             /* used for timings */
   FILE *outf;                   /* output file */
   int pnum;                     /* loop control var for number of passes */
   double totaltime;             /* all times added together (for average) */
   int index;                    /* this is an index used to correspond the
                                    following experiment parameters:

                                    Bubble, shaker, select, insert sorts use:

                                 INDEX    NUM OF PASSES        SIZE OF SORTARRAY
                                 ------------------------------------------------
                                   0           7                    100
                                   1           6                    500
                                   2           5                    1000
                                   3           4                    2000
                                   4           3                    5000
                                   5           2                    10000

                                    Shell, quick, heap, merge, radix sorts use:

                                   0           7                    1000
                                   1           6                    2000
                                   2           5                    5000
                                   3           4                    8000
                                   4           3                    10000
                                   5           2                    20000
                                                                              */
```

```
   countarray[0]=100;  /* initialize count array with counts for */
   countarray[1]=500;  /* slow sorts */
   countarray[2]=1000;
   countarray[3]=2000;
   countarray[4]=5000;
   countarray[5]=10000;

   numpasses[0]=7; /* initialize pass array with number of passes for the */
   numpasses[1]=6; /* slow sorts */
   numpasses[2]=5;
   numpasses[3]=4;
   numpasses[4]=3;
   numpasses[5]=2;

   outf=fopen("b:proj.out","w");    /* open output file */

/**************************************************************************/
/* Here is the experimentation on the slower sorts (O(n**2)): bubble,     */
/* shaker(an improved bubble sort), select, insert.                       */
/**************************************************************************/

/**************************************************************************/
/*                              BUBBLE SORT                               */
/**************************************************************************/

   for (index=0; index<=5; ++index) {
     revorder(sortarray,countarray[index]);
     start=time(0);
     bubble(sortarray,countarray[index]);
     end=time(0);
     fprintf(outf,"Bubble sort on revorder case of size(%d): %f \n",
             countarray[index], difftime(end,start));
   }
   for (index=0; index<=5; ++index) {
     inorder(sortarray,countarray[index]);
     start=time(0);
     bubble(sortarray,countarray[index]);
     end=time(0);
     fprintf(outf,"Bubble sort on inorder case of size(%d): %f \n",
             countarray[index], difftime(end,start));
   }
   for (index=0; index<=5; ++index) {
     totaltime=0;
     for (pnum=0; pnum<numpasses[index]; ++pnum) {
       average(sortarray,countarray[index]);
       start=time(0);
       bubble(sortarray,countarray[index]);
       end=time(0);
       timearray[pnum]=difftime(end,start);
       totaltime=totaltime+timearray[pnum];
     }
     fprintf(outf,"Bubble sort on average cases of size(%d): \n",
             countarray[index]);
     filetimes(timearray,numpasses[index],outf);
     fprintf(outf,"Average time: %f \n\n",totaltime/numpasses[index]);
   }

/**************************************************************************/
/*                              SHAKER SORT                               */
/**************************************************************************/

   for (index=0; index<=5; ++index) {
     revorder(sortarray,countarray[index]);
     start=time(0);
     shaker(sortarray,countarray[index]);
     end=time(0);
```

```
    fprintf(outf,"Shaker sort on revorder case of size(%d): %f \n",
            countarray[index], difftime(end,start));
  }
  for (index=0; index<=5; ++index) {
    inorder(sortarray,countarray[index]);
    start=time(0);
    shaker(sortarray,countarray[index]);
    end=time(0);
    fprintf(outf,"Shaker sort on inorder case of size(%d): %f \n",
            countarray[index], difftime(end,start));
  }
  for (index=0; index<=5; ++index) {
    totaltime=0;
    for (pnum=0; pnum<numpasses[index]; ++pnum) {
      average(sortarray,countarray[index]);
      start=time(0);
      shaker(sortarray,countarray[index]);
      end=time(0);
      timearray[pnum]=difftime(end,start);
      totaltime=totaltime+timearray[pnum];
    }
    fprintf(outf,"Shaker sort on average cases of size(%d): \n",
            countarray[index]);
    filetimes(timearray,numpasses[index],outf);
    fprintf(outf,"Average time: %f \n\n",totaltime/numpasses[index]);
  }

/****************************************************************************/
/*                              SELECT SORT                                 */
/****************************************************************************/

  for (index=0; index<=5; ++index) {
    revorder(sortarray,countarray[index]);
    start=time(0);
    select(sortarray,countarray[index]);
    end=time(0);
    fprintf(outf,"Select sort on revorder case of size(%d): %f \n",
            countarray[index], difftime(end,start));
  }
  for (index=0; index<=5; ++index) {
    inorder(sortarray,countarray[index]);
    start=time(0);
    select(sortarray,countarray[index]);
    end=time(0);
    fprintf(outf,"Select sort on inorder case of size(%d): %f \n",
            countarray[index], difftime(end,start));
  }
  for (index=0; index<=5; ++index) {
    totaltime=0;
    for (pnum=0; pnum<numpasses[index]; ++pnum) {
      average(sortarray,countarray[index]);
      start=time(0);
      select(sortarray,countarray[index]);
      end=time(0);
      timearray[pnum]=difftime(end,start);
      totaltime=totaltime+timearray[pnum];
    }
    fprintf(outf,"Select sort on average cases of size(%d): \n",
            countarray[index]);
    filetimes(timearray,numpasses[index],outf);
    fprintf(outf,"Average time: %f \n\n",totaltime/numpasses[index]);
  }

/****************************************************************************/
/*                              INSERT SORT                                 */
/****************************************************************************/
```

```
    for (index=0; index<=5; ++index) {
      revorder(sortarray,countarray[index]);
      start=time(0);
      insert(sortarray,countarray[index]);
      end=time(0);
      fprintf(outf,"Insert sort on revorder case of size(%d): %f \n",
              countarray[index], difftime(end,start));
    }
    for (index=0; index<=5; ++index) {
      inorder(sortarray,countarray[index]);
      start=time(0);
      insert(sortarray,countarray[index]);
      end=time(0);
      fprintf(outf,"Insert sort on inorder case of size(%d): %f \n",
              countarray[index], difftime(end,start));
    }
    for (index=0; index<=5; ++index) {
      totaltime=0;
      for (pnum=0; pnum<numpasses[index]; ++pnum) {
        average(sortarray,countarray[index]);
        start=time(0);
        insert(sortarray,countarray[index]);
        end=time(0);
        timearray[pnum]=difftime(end,start);
        totaltime=totaltime+timearray[pnum];
      }
      fprintf(outf,"Insert sort on average cases of size(%d): \n",
              countarray[index]);
      filetimes(timearray,numpasses[index],outf);
      fprintf(outf,"Average time: %f \n\n",totaltime/numpasses[index]);
    }

    /*********************************************************************/
    /* Initialize experiment parameters for the faster sorts.           */
    /*********************************************************************/

    countarray[0]=1000;  /* initialize count array with counts for */
    countarray[1]=2000;  /* fast sorts */
    countarray[2]=5000;
    countarray[3]=8000;
    countarray[4]=10000;
    countarray[5]=20000;

    numpasses[0]=7; /* initialize pass array with number of passes for the */
    numpasses[1]=6; /* fast sorts */
    numpasses[2]=5;
    numpasses[3]=4;
    numpasses[4]=3;
    numpasses[5]=2;

/**************************************************************************/
/*                              SHELL SORT                                */
/**************************************************************************/

    for (index=0; index<=5; ++index) {
      revorder(sortarray,countarray[index]);
      start=time(0);
      shell(sortarray,countarray[index]);
      end=time(0);
      fprintf(outf,"Shell sort on revorder case of size(%d): %f \n",
              countarray[index], difftime(end,start));
    }
    for (index=0; index<=5; ++index) {
      inorder(sortarray,countarray[index]);
      start=time(0);
```

```
    shell(sortarray,countarray[index]);
    end=time(0);
    fprintf(outf,"Shell sort on inorder case of size(%d): %f \n",
            countarray[index], difftime(end,start));
  }
  for (index=0; index<=5; ++index) {
    totaltime=0;
    for (pnum=0; pnum<numpasses[index]; ++pnum) {
      average(sortarray,countarray[index]);
      start=time(0);
      shell(sortarray,countarray[index]);
      end=time(0);
      timearray[pnum]=difftime(end,start);
      totaltime=totaltime+timearray[pnum];
    }
    fprintf(outf,"Shell sort on average cases of size(%d): \n",
            countarray[index]);
    filetimes(timearray,numpasses[index],outf);
    fprintf(outf,"Average time: %f \n\n",totaltime/numpasses[index]);
  }

/****************************************************************************/
/*                              QUICKSORT                                   */
/****************************************************************************/

  for (index=0; index<=5; ++index) {
    revorder(sortarray,countarray[index]);
    start=time(0);
    quick(sortarray,countarray[index]);
    end=time(0);
    fprintf(outf,"Quick sort on revorder case of size(%d): %f \n",
            countarray[index], difftime(end,start));
  }
  for (index=0; index<=5; ++index) {
    inorder(sortarray,countarray[index]);
    start=time(0);
    quick(sortarray,countarray[index]);
    end=time(0);
    fprintf(outf,"Quick sort on inorder case of size(%d): %f \n",
            countarray[index], difftime(end,start));
  }
  for (index=0; index<=5; ++index) {
    totaltime=0;
    for (pnum=0; pnum<numpasses[index]; ++pnum) {
      average(sortarray,countarray[index]);
      start=time(0);
      quick(sortarray,countarray[index]);
      end=time(0);
      timearray[pnum]=difftime(end,start);
      totaltime=totaltime+timearray[pnum];
    }
    fprintf(outf,"Quick sort on average cases of size(%d): \n",
            countarray[index]);
    filetimes(timearray,numpasses[index],outf);
    fprintf(outf,"Average time: %f \n\n",totaltime/numpasses[index]);
  }

/****************************************************************************/
/*                              HEAPSORT                                    */
/****************************************************************************/

  for (index=0; index<=5; ++index) {
    revorder(sortarray,countarray[index]);
    start=time(0);
    heapsort(sortarray,countarray[index]);
    end=time(0);
```

```
    fprintf(outf,"Heap sort on revorder case of size(%d): %f \n",
            countarray[index], difftime(end,start));
  }
  for (index=0; index<=5; ++index) {
    inorder(sortarray,countarray[index]);
    start=time(0);
    heapsort(sortarray,countarray[index]);
    end=time(0);
    fprintf(outf,"Heap sort on inorder case of size(%d): %f \n",
            countarray[index], difftime(end,start));
  }
  for (index=0; index<=5; ++index) {
    totaltime=0;
    for (pnum=0; pnum<numpasses[index]; ++pnum) {
      average(sortarray,countarray[index]);
      start=time(0);
      heapsort(sortarray,countarray[index]);
      end=time(0);
      timearray[pnum]=difftime(end,start);
      totaltime=totaltime+timearray[pnum];
    }
    fprintf(outf,"Heap sort on average cases of size(%d): \n",
            countarray[index]);
    filetimes(timearray,numpasses[index],outf);
    fprintf(outf,"Average time: %f \n\n",totaltime/numpasses[index]);
  }

/****************************************************************************/
/*                              MERGE SORT                                  */
/****************************************************************************/

  for (index=0; index<=5; ++index) {
    revorder(sortarray,countarray[index]);
    start=time(0);
    mergesort(sortarray,countarray[index]);
    end=time(0);
    fprintf(outf,"Merge sort on revorder case of size(%d): %f \n",
            countarray[index], difftime(end,start));
  }
  for (index=0; index<=5; ++index) {
    inorder(sortarray,countarray[index]);
    start=time(0);
    mergesort(sortarray,countarray[index]);
    end=time(0);
    fprintf(outf,"Merge sort on inorder case of size(%d): %f \n",
            countarray[index], difftime(end,start));
  }
  for (index=0; index<=5; ++index) {
    totaltime=0;
    for (pnum=0; pnum<numpasses[index]; ++pnum) {
      average(sortarray,countarray[index]);
      start=time(0);
      mergesort(sortarray,countarray[index]);
      end=time(0);
      timearray[pnum]=difftime(end,start);
      totaltime=totaltime+timearray[pnum];
    }
    fprintf(outf,"Merge sort on average cases of size(%d): \n",
            countarray[index]);
    filetimes(timearray,numpasses[index],outf);
    fprintf(outf,"Average time: %f \n\n",totaltime/numpasses[index]);
  }

/****************************************************************************/
/*                              RADIX SORT                                  */
/****************************************************************************/
```

```
    for (index=0; index<=5; ++index) {
      revorder(sortarray,countarray[index]);
      start=time(0);
      radsort(sortarray,countarray[index]);
      end=time(0);
      fprintf(outf,"Radix sort on revorder case of size(%d): %f \n",
              countarray[index], difftime(end,start));
    }
    for (index=0; index<=5; ++index) {
      inorder(sortarray,countarray[index]);
      start=time(0);
      radsort(sortarray,countarray[index]);
      end=time(0);
      fprintf(outf,"Radix sort on inorder case of size(%d): %f \n",
              countarray[index], difftime(end,start));
    }
    for (index=0; index<=5; ++index) {
      totaltime=0;
      for (pnum=0; pnum<numpasses[index]; ++pnum) {
        average(sortarray,countarray[index]);
        start=time(0);
        radsort(sortarray,countarray[index]);
        end=time(0);
        timearray[pnum]=difftime(end,start);
        totaltime=totaltime+timearray[pnum];
      }
      fprintf(outf,"Radix sort on average cases of size(%d): \n",
              countarray[index]);
      filetimes(timearray,numpasses[index],outf);
      fprintf(outf,"Average time: %f \n\n",totaltime/numpasses[index]);
    }

    fclose(outf);   /* close output file and flush stream */
    soundalarm();   /* sound off when experiment completed */
}

/***************************************************************************/
/* function soundalarm:  This sounds off when the program is done.        */
/***************************************************************************/

soundalarm()

{
  char ch;

  while (!(kbhit())) printf("\a");
  ch=getch();
}

/***************************************************************************/
/* function kbhit:  Returns 0 if no key hit, true otherwise.              */
/***************************************************************************/

kbhit()
{
   return((char) bdos(0xB,0,0));
}

/***************************************************************************/
/* function bubble:  This is the bubble sort function.                    */
/***************************************************************************/

bubble(sarray,count)
int *sarray;
int count;
```

```
{
   register int a,b;
   register int temp;
   int exch;

   a=1;
   exch=1;
   while (exch && (a<count)) {
     exch=0;
     for(b=count-1; b>=a; --b) {
       if (sarray[b-1]>sarray[b]) {
          temp=sarray[b-1];
          sarray[b-1]=sarray[b];
          sarray[b]=temp;
          exch=1;
       }
     }
     a++;
   }
}

/***************************************************************************/
/* function shaker: This is an improved version of the bubble sort.        */
/***************************************************************************/

shaker(sarray,count)
int *sarray;
int count;

{
   register int a, b, c, d;
   int temp;

   c=1;
   b=count-1;
   d=count-1;
   do {
     for (a=d; a>=c; --a) {
       if (sarray[a-1] > sarray[a]) {
         temp = sarray[a-1];
         sarray[a-1]=sarray[a];
         sarray[a]=temp;
         b=a;
       }
     }
     c=b+1;
     for (a=c; a<d+1; ++a) {
       if (sarray[a-1]>sarray[a]) {
         temp=sarray[a-1];
         sarray[a-1]=sarray[a];
         sarray[a]=temp;
         b=a;
       }
     }
     d=b-1;
   } while (c <= d);
}

/***************************************************************************/
/* function select:  This is the selection sort algorithm.                 */
/***************************************************************************/

select(sarray,count)
int *sarray;
int count;
```

```
{
   register int a, b, c;
   int temp;

   for (a=0; a<count-1; ++a) {
     c=a;
     temp=sarray[a];
     for (b=a+1; b<count; ++b) {
       if (sarray[b]<temp) {
         c=b;
         temp=sarray[b];
       }
     }
     sarray[c]=sarray[a];
     sarray[a]=temp;
   }
}

/****************************************************************************/
/* function insert:  This is the insert sort algorithm.                     */
/****************************************************************************/

insert(sarray,count)
int *sarray;
int count;

{
   register int a, b;
   int temp;

   for (a=1; a<count; ++a) {
     temp=sarray[a];
     b=a-1;
     while (b>=0 && temp<sarray[b]) {
       sarray[b+1]=sarray[b];
       b=b-1;
     }
     sarray[b+1]=temp;
   }
}

/****************************************************************************/
/* function shell:  This function is the shell sort.                        */
/****************************************************************************/

shell(sarray,count)
int *sarray;
int count;

{
   register int i, j, k, s, w;
   int x, a[5];

   a[0]=9;
   a[1]=5;
   a[2]=3;
   a[3]=2;
   a[4]=1;
   for (w=0; w<5; w++) {
     k=a[w];
     s=-k;
     for (i=k; i<count; ++i) {
       x = sarray[i];
       j=i-k;
```

```
          s=-k;
          s=s+1;
          sarray[s]=x;
        }
        while (x<sarray[j] && j>=0 && j<=count) {
          sarray[j+k]=sarray[j];
          j=j-k;
        }
        sarray[j+k]=x;
      }
    }
}

/****************************************************************************/
/* function quick:  This is the quicksort algorithm.                        */
/****************************************************************************/

quick(sarray,count)
int *sarray;
int count;

{
   qs(sarray,0,count-1);
}

qs(sarray,left,right)
int *sarray;
int left,right;

{
   register int i, j;
   int x, y;

   i=left;
   j=right;
   x=sarray[(left+right)/2];
   do {
     while(sarray[i]<x && i<right) i++;
     while(x<sarray[j] && j>left) j--;
     if (i<=j) {
       y=sarray[i];
       sarray[i]=sarray[j];
       sarray[j]=y;
       i=i+1;
       j=j-1;
     }
   } while (i<=j);
   if (left<j) qs(sarray,left,j);
   if (i<right) qs(sarray,i,right);
}

/****************************************************************************/
/* function heapsort:  This is the heapsort algorithm.                      */
/****************************************************************************/

heapsort(sarray,count)
int *sarray;
int count;

{
   int temp;
   register int nextposition;

   create_heap(sarray,count);
   for (nextposition=count-1; nextposition>=2; nextposition--) {
     temp=sarray[nextposition];
```

```
     sarray[nextposition]=sarray[1];
     sarray[1]=temp;
     adjust(sarray,nextposition-1);
   }
}

/*************************************************************************/
/* function adjust:  This function, when necessary, switches parents and */
/*                   children to assure that the heap is correct.        */
/*************************************************************************/

adjust(sarray,k)
int *sarray;
int k;

{
   int parent, child, temp;

   parent=1;
   child=2;
   if ((k>=3) && (sarray[3]>sarray[2])) child=3;
   while ((child<=k) && (sarray[child]>sarray[parent])) {
     temp=sarray[child];
     sarray[child]=sarray[parent];
     sarray[parent]=temp;
     parent=child;
     child=2*parent;
     if (child+1<=k) if (sarray[child+1]>sarray[child]) child++;
   }
}

/*************************************************************************/
/* function create_heap:  This function creates a heap within an array.  */
/*************************************************************************/

create_heap(sarray,count)
int *sarray;
int count;

{
   int node, parent,temp;
   register int nextnode;

   for (nextnode=2; nextnode<count; nextnode++) {
     node=nextnode;
     parent=node/2;
     while ((node != 1) && (sarray[parent]<sarray[node])) {
       temp=sarray[parent];
       sarray[parent]=sarray[node];
       sarray[node]=temp;
       node=parent;
       parent=node/2;
     }
   }
}
/*************************************************************************/
/* function mergesort:  This function is the merge sort algorithm.       */
/*************************************************************************/

mergesort(sarray,count)
int *sarray;
int count;

{
   int low=1;
   int high;
```

```
   high=count-1;
   msort(sarray,low,high);
}

msort(sarray,low,high)
int *sarray;
int low;
int high;

{
   int mid;

   if (low < high) {
     mid=((low+high)/2);
     msort(sarray,low,mid);
     msort(sarray,mid+1,high);
     merge(sarray,low,mid,high);
   }
}

/*************************************************************************/
/* function merge:  This function merges the two subarrays together.     */
/*************************************************************************/

merge(sarray,low,mid,high)
int *sarray;
int low;
int mid;
int high;

{
  register int h,i,j,k;

/* HERE IS THE TEMPORARY ARRAY */

  int temparray[30000];

  h=low;
  i=low;
  j=mid+1;
  while ((h<=mid) && (j<=high)) {
    if (sarray[h]<=sarray[j]) {
      temparray[i]=sarray[h];
      h++;
    }
    else {
      temparray[i]=sarray[j];
      j++;
    }
    i++;
  }
  if (h>mid) for (k=j; k<=high; k++) {
    temparray[i]=sarray[k];
    i++;
  }
  else for (k=h; k<=mid; k++) {
    temparray[i]=sarray[k];
    i++;
  }
  for (k=low; k<=high; k++) sarray[k]=temparray[k];
}

/*************************************************************************/
/* function radsort:  This is the radix exchange method of sorting.      */
/*************************************************************************/
```

```
radsort(sarray,count)
int *sarray;
int count;

{
   int bitnum;
   int bitarray[16];
   bitarray[0]=0x1;
   bitarray[1]=0x2;
   bitarray[2]=0x4;
   bitarray[3]=0x8;
   bitarray[4]=0x10;
   bitarray[5]=0x20;
   bitarray[6]=0x40;
   bitarray[7]=0x80;
   bitarray[8]=0x100;
   bitarray[9]=0x200;
   bitarray[10]=0x400;
   bitarray[11]=0x800;
   bitarray[12]=0x1000;
   bitarray[13]=0x2000;
   bitarray[14]=0x4000;
   bitarray[15]=0x8000;
   bitnum=15;

   rs(sarray,0,count-1,bitnum,bitarray);
}

/***************************************************************************/
/* function rs:  This is the recursive algorithm.                          */
/***************************************************************************/

rs(sarray,left,right,bitnum,bitarray)
int *sarray;
int left;
int right;
int bitnum;
int *bitarray;

{
   register int i,j;
   int temp;

   if (bitnum>=0){
     i=left;
     j=right;
     do {
       while(!(sarray[i] & bitarray[bitnum]) && i<right) i++;
       while((sarray[j] & bitarray[bitnum]) && j>left) j--;
       if (i<=j) {
         temp=sarray[i];
         sarray[i]=sarray[j];
         sarray[j]=temp;
         i=i+1;
         j=j-1;
       }
     } while (i<=j);
     bitnum--;
     if ((j<left) || (i>right)) rs(sarray,left,right,bitnum,bitarray);
     else {
       if (left<j) rs(sarray,left,j,bitnum,bitarray);
       if (i<right) rs(sarray,i,right,bitnum,bitarray);
     }
   }
}
```

```
/**************************************************************************/
/* function inorder:   This function generates the inorder case array.      *
/
/**************************************************************************/

inorder(rndarray,count)
int *rndarray;
int count;
{
   int indx;
   for (indx=0;indx<count;indx++) rndarray[indx]=indx;
}

/**************************************************************************/
/* function revorder:  This function generates the revorder case array.     *
/
/**************************************************************************/

revorder(rndarray,count)
int *rndarray;
int count;
{
   int indx;
   int val;

   val = count-1;
   for (indx=0;indx<count;indx++)
       {
            rndarray[indx]=val;
            val = val-1;
       }
}

/**************************************************************************/
/* function average:  This function generates the average case array.    */
/**************************************************************************/

average(rndarray,count)
int *rndarray;
int count;
{
   int indx;
   int stime;
   long ltime;

   ltime=time(NULL);
   stime=(unsigned int) ltime/2;
   srand(stime);
   for (indx=0;indx<count;indx++) rndarray[indx]=rand();
}

/**************************************************************************/
/* function displayarray:This function displays the array.                */
/**************************************************************************/

displayarray(sarray,count)
int *sarray;
int count;

{
   int indx;
   int lcount=0;

   for(indx=0; indx<count; indx++) {
```

```
      printf("%d ",sarray[indx]);
      lcount=lcount+1;
      if (lcount>5) {
         lcount=0;
         printf("\n");
      }
   }
}

/***************************************************************************/
/* function filetimes:  This function moves the timing results to disk.   */
/***************************************************************************/

filetimes(results,high,outf)
double *results;
int high;
FILE *outf;

{
   int index;

   fprintf(outf,"\n");
   for (index=0; index<high; ++index) fprintf(outf," %f ",results[index]);
   fprintf(outf,"\n");
}
```

Timing data actually generated by program

```
Bubble sort on revorder case of size(100): 1.000000
Bubble sort on revorder case of size(500): 6.000000
Bubble sort on revorder case of size(1000): 24.000000
Bubble sort on revorder case of size(2000): 95.000000
Bubble sort on revorder case of size(5000): 598.000000
Bubble sort on revorder case of size(10000): 2394.000000
Bubble sort on inorder case of size(100): 0.000000
Bubble sort on inorder case of size(500): 0.000000
Bubble sort on inorder case of size(1000): 0.000000
Bubble sort on inorder case of size(2000): 0.000000
Bubble sort on inorder case of size(5000): 0.000000
Bubble sort on inorder case of size(10000): 1.000000
Bubble sort on average cases of size(100):

 0.000000  0.000000  0.000000  0.000000  0.000000  1.000000  0.000000
Average time: 0.142857

Bubble sort on average cases of size(500):

 4.000000  4.000000  4.000000  4.000000  4.000000  4.000000
Average time: 4.000000

Bubble sort on average cases of size(1000):

 16.000000  17.000000  16.000000  16.000000  17.000000
Average time: 16.400000

Bubble sort on average cases of size(2000):

 66.000000  66.000000  65.000000  66.000000
Average time: 65.750000

Bubble sort on average cases of size(5000):

 410.000000  408.000000  410.000000
Average time: 409.333333

Bubble sort on average cases of size(10000):

 1655.000000  1638.000000
Average time: 1646.500000

Shaker sort on revorder case of size(100): 1.000000
Shaker sort on revorder case of size(500): 6.000000
Shaker sort on revorder case of size(1000): 24.000000
Shaker sort on revorder case of size(2000): 96.000000
Shaker sort on revorder case of size(5000): 597.000000
Shaker sort on revorder case of size(10000): 2390.000000
Shaker sort on inorder case of size(100): 0.000000
Shaker sort on inorder case of size(500): 0.000000
Shaker sort on inorder case of size(1000): 0.000000
Shaker sort on inorder case of size(2000): 0.000000
Shaker sort on inorder case of size(5000): 0.000000
Shaker sort on inorder case of size(10000): 0.000000
Shaker sort on average cases of size(100):

 0.000000  1.000000  0.000000  0.000000  0.000000  0.000000  0.000000
Average time: 0.142857

Shaker sort on average cases of size(500):
```

```
 3.000000  3.000000  4.000000  3.000000  4.000000  3.000000
Average time: 3.333333

Shaker sort on average cases of size(1000):

 13.000000  13.000000  14.000000  13.000000  14.000000
Average time: 13.400000

Shaker sort on average cases of size(2000):

 54.000000  55.000000  55.000000  53.000000
Average time: 54.250000

Shaker sort on average cases of size(5000):

 345.000000  336.000000  339.000000
Average time: 340.000000

Shaker sort on average cases of size(10000):

 1353.000000  1370.000000
Average time: 1361.500000

Select sort on revorder case of size(100): 0.000000
Select sort on revorder case of size(500): 3.000000
Select sort on revorder case of size(1000): 10.000000
Select sort on revorder case of size(2000): 39.000000
Select sort on revorder case of size(5000): 246.000000
Select sort on revorder case of size(10000): 984.000000
Select sort on inorder case of size(100): 0.000000
Select sort on inorder case of size(500): 2.000000
Select sort on inorder case of size(1000): 8.000000
Select sort on inorder case of size(2000): 30.000000
Select sort on inorder case of size(5000): 190.000000
Select sort on inorder case of size(10000): 757.000000
Select sort on average cases of size(100):

 0.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
Average time: 0.000000

Select sort on average cases of size(500):

 2.000000  2.000000  2.000000  2.000000  2.000000  2.000000
Average time: 2.000000

Select sort on average cases of size(1000):

 7.000000  8.000000  8.000000  8.000000  7.000000
Average time: 7.600000

Select sort on average cases of size(2000):

 30.000000  31.000000  30.000000  30.000000
Average time: 30.250000

Select sort on average cases of size(5000):

 189.000000  189.000000  190.000000
Average time: 189.333333

Select sort on average cases of size(10000):

 758.000000  758.000000
Average time: 758.000000
```

```
Insert sort on revorder case of size(100): 0.000000
Insert sort on revorder case of size(500): 4.000000
Insert sort on revorder case of size(1000): 13.000000
Insert sort on revorder case of size(2000): 54.000000
Insert sort on revorder case of size(5000): 335.000000
Insert sort on revorder case of size(10000): 1339.000000
Insert sort on inorder case of size(100): 0.000000
Insert sort on inorder case of size(500): 0.000000
Insert sort on inorder case of size(1000): 0.000000
Insert sort on inorder case of size(2000): 0.000000
Insert sort on inorder case of size(5000): 0.000000
Insert sort on inorder case of size(10000): 0.000000
Insert sort on average cases of size(100):

 1.000000  0.000000  0.000000  0.000000  0.000000  0.000000  0.000000
Average time: 0.142857

Insert sort on average cases of size(500):

 2.000000  1.000000  1.000000  2.000000  1.000000  1.000000
Average time: 1.333333

Insert sort on average cases of size(1000):

 7.000000  7.000000  7.000000  7.000000  6.000000
Average time: 6.800000

Insert sort on average cases of size(2000):

 27.000000  27.000000  27.000000  26.000000
Average time: 26.750000

Insert sort on average cases of size(5000):

 168.000000  164.000000  168.000000
Average time: 166.666667

Insert sort on average cases of size(10000):

 673.000000  672.000000
Average time: 672.500000

Shell sort on revorder case of size(1000): 2.000000
Shell sort on revorder case of size(2000): 8.000000
Shell sort on revorder case of size(5000): 43.000000
Shell sort on revorder case of size(8000): 109.000000
Shell sort on revorder case of size(10000): 170.000000
Shell sort on revorder case of size(20000): 677.000000
Shell sort on inorder case of size(1000): 1.000000
Shell sort on inorder case of size(2000): 1.000000
Shell sort on inorder case of size(5000): 1.000000
Shell sort on inorder case of size(8000): 2.000000
Shell sort on inorder case of size(10000): 3.000000
Shell sort on inorder case of size(20000): 5.000000
Shell sort on average cases of size(1000):

 2.000000  1.000000  1.000000  2.000000  1.000000  1.000000  2.000000
Average time: 1.428571

Shell sort on average cases of size(2000):

 4.000000  4.000000  5.000000  4.000000  5.000000  4.000000
Average time: 4.333333

Shell sort on average cases of size(5000):
```

```
 24.000000  24.000000  23.000000  23.000000  23.000000
Average time: 23.400000

Shell sort on average cases of size(8000):

 59.000000  58.000000  59.000000  58.000000
Average time: 58.500000

Shell sort on average cases of size(10000):

 90.000000  91.000000  91.000000
Average time: 90.666666

Shell sort on average cases of size(20000):

 356.000000  350.000000
Average time: 353.000000

Quick sort on revorder case of size(1000): 0.000000
Quick sort on revorder case of size(2000): 0.000000
Quick sort on revorder case of size(5000): 1.000000
Quick sort on revorder case of size(8000): 2.000000
Quick sort on revorder case of size(10000): 2.000000
Quick sort on revorder case of size(20000): 6.000000
Quick sort on inorder case of size(1000): 0.000000
Quick sort on inorder case of size(2000): 1.000000
Quick sort on inorder case of size(5000): 1.000000
Quick sort on inorder case of size(8000): 2.000000
Quick sort on inorder case of size(10000): 2.000000
Quick sort on inorder case of size(20000): 5.000000
Quick sort on average cases of size(1000):

 1.000000  0.000000  0.000000  0.000000  0.000000  1.000000  0.000000
Average time: 0.285714

Quick sort on average cases of size(2000):

 1.000000  1.000000  1.000000  1.000000  1.000000  1.000000
Average time: 1.000000

Quick sort on average cases of size(5000):

 2.000000  2.000000  2.000000  2.000000  2.000000
Average time: 2.000000

Quick sort on average cases of size(8000):

 3.000000  3.000000  3.000000  3.000000
Average time: 3.000000

Quick sort on average cases of size(10000):

 4.000000  4.000000  4.000000
Average time: 4.000000

Quick sort on average cases of size(20000):

 8.000000  9.000000
Average time: 8.500000

Heap sort on revorder case of size(1000): 1.000000
Heap sort on revorder case of size(2000): 1.000000
Heap sort on revorder case of size(5000): 4.000000
Heap sort on revorder case of size(8000): 7.000000
Heap sort on revorder case of size(10000): 8.000000
Heap sort on revorder case of size(20000): 18.000000
```

```
Heap sort on inorder case of size(1000): 1.000000
Heap sort on inorder case of size(2000): 2.000000
Heap sort on inorder case of size(5000): 8.000000
Heap sort on inorder case of size(8000): 13.000000
Heap sort on inorder case of size(10000): 16.000000
Heap sort on inorder case of size(20000): 35.000000
Heap sort on average cases of size(1000):

 1.000000  1.000000  1.000000  0.000000  1.000000  1.000000  1.000000
Average time: 0.857143

Heap sort on average cases of size(2000):

 1.000000  1.000000  2.000000  2.000000  1.000000  2.000000
Average time: 1.500000

Heap sort on average cases of size(5000):

 4.000000  5.000000  4.000000  4.000000  5.000000
Average time: 4.400000

Heap sort on average cases of size(8000):

 7.000000  8.000000  8.000000  8.000000
Average time: 7.750000

Heap sort on average cases of size(10000):

 9.000000  9.000000  9.000000
Average time: 9.000000

Heap sort on average cases of size(20000):

 20.000000  20.000000
Average time: 20.000000

Merge sort on revorder case of size(1000): 1.000000
Merge sort on revorder case of size(2000): 1.000000
Merge sort on revorder case of size(5000): 4.000000
Merge sort on revorder case of size(8000): 7.000000
Merge sort on revorder case of size(10000): 8.000000
Merge sort on revorder case of size(20000): 15.000000
Merge sort on inorder case of size(1000): 1.000000
Merge sort on inorder case of size(2000): 1.000000
Merge sort on inorder case of size(5000): 3.000000
Merge sort on inorder case of size(8000): 6.000000
Merge sort on inorder case of size(10000): 9.000000
Merge sort on inorder case of size(20000): 15.000000
Merge sort on average cases of size(1000):

 0.000000  1.000000  1.000000  1.000000  1.000000  1.000000  1.000000
Average time: 0.857143

Merge sort on average cases of size(2000):

 2.000000  1.000000  1.000000  2.000000  2.000000  2.000000
Average time: 1.666667

Merge sort on average cases of size(5000):

 5.000000  4.000000  4.000000  4.000000  4.000000
Average time: 4.200000

Merge sort on average cases of size(8000):

 7.000000  7.000000  7.000000  7.000000
```

```
Average time: 7.000000

Merge sort on average cases of size(10000):

 9.000000  9.000000  9.000000
Average time: 9.000000

Merge sort on average cases of size(20000):

 15.000000  15.000000
Average time: 15.000000

Radix sort on revorder case of size(1000): 0.000000
Radix sort on revorder case of size(2000): 1.000000
Radix sort on revorder case of size(5000): 2.000000
Radix sort on revorder case of size(8000): 3.000000
Radix sort on revorder case of size(10000): 4.000000
Radix sort on revorder case of size(20000): 9.000000
Radix sort on inorder case of size(1000): 0.000000
Radix sort on inorder case of size(2000): 1.000000
Radix sort on inorder case of size(5000): 2.000000
Radix sort on inorder case of size(8000): 3.000000
Radix sort on inorder case of size(10000): 4.000000
Radix sort on inorder case of size(20000): 8.000000
Radix sort on average cases of size(1000):

 0.000000  0.000000  0.000000  1.000000  0.000000  1.000000  0.000000
Average time: 0.285714

Radix sort on average cases of size(2000):

 1.000000  1.000000  0.000000  1.000000  1.000000  1.000000
Average time: 0.833333

Radix sort on average cases of size(5000):

 3.000000  3.000000  2.000000  2.000000  2.000000
Average time: 2.400000

Radix sort on average cases of size(8000):

 4.000000  4.000000  4.000000  4.000000
Average time: 4.000000

Radix sort on average cases of size(10000):

 5.000000  5.000000  5.000000
Average time: 5.000000

Radix sort on average cases of size(20000):

 10.000000  10.000000
Average time: 10.000000
```