FINDING THE MEDIAN IN LINEAR WORST-CASE TIME

By

DAVID B. CROW

Bachelor of Science

East Central University

Ada, Oklahoma

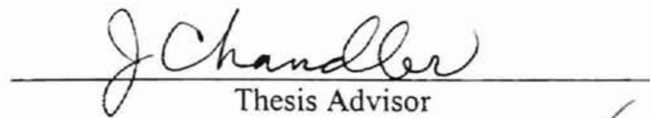
1990

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 2001

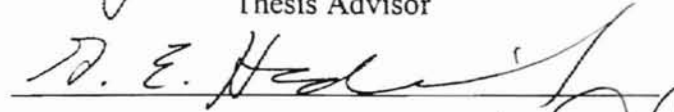
FINDING THE MEDIAN IN LINEAR

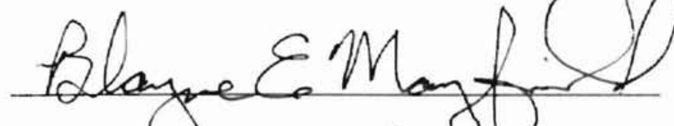
WORST-CASE TIME

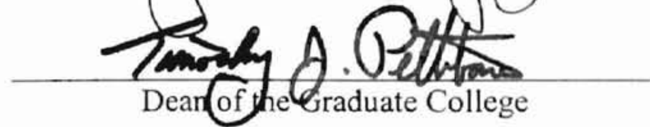
Thesis Approved:



Thesis Advisor







Dean of the Graduate College

PREFACE

The selection problem is that of finding the Kth smallest element of an array of N distinct elements. A number of algorithms have been created to address the selection problem, each with its own strengths and weaknesses.

One obvious method for addressing the selection problem would be to sort the whole input array in $\Theta(N \log(N))$ time using merge sort or heapsort [18] and then simply request the Kth element from the input array. While acceptable for small values of N, sorting quickly becomes inefficient for larger values of N.

A special case of the selection problem arises when the median element is sought. This occurs when $K = \lceil N/2 \rceil$. This is important because the median is a robust estimator of position whereas the mean, or average, is not; the mean is greatly affected by *outliers*, while the median is not. An outlier is an element in a set that is unnaturally far away from the population. For example, the outlier of the set {14, 10, 21, 11, 19, 45, 17} is 45. The average of this set is 19.5 while the median is 17. If 45 were changed to 76, the average would become 24 while the median would remain the same. This is what is meant by a robust estimator: an estimator that is influenced only weakly if at all by the value of an outlier.

TABLE OF CONTENTS

Chapter Page

1. INTRODUCTION 1
2. THE ALGORITHMS 3
   - Heapsort 3
     - Example 6
   - FIND 8
     - Analysis 10
     - Partitioning 12
     - Handling Duplicate Elements 13
     - Example 15
   - SELECT 19
     - Sampling 19
     - Analysis 20
     - Example 23
   - PICK 27
     - Analysis 27
     - Selecting the Median of 5 elements 29
     - Discarding Elements 30
     - Why Groups of 5? 33
     - Example 35
     - Improvements 41
3. METHODS OF TESTING AND COMPARISONS 43
4. RESULTS 46
5. SUMMARY, CONCLUSIONS AND FUTURE WORK 67
   - C source code for Treesort 3 69

C source code for FIND 70
C source Code for Dromey's Improvement to FIND 72
C source code for SELECT 72
C source code for PICK 75
C source code for driver 77
Formula for FIND's average case 78
Formula for FIND's worst case 78
Formula for the minimum elements discarded in PICK 80
Formulas for the cost of the PICK algorithm 81

LIST OF TABLES

Table Page

1 – Cost of duplicate elements 14

2 – Comparison of at-least and at-most formulas 33

3 – Number of comparisons for average case 46

4 - Number of comparisons for worst known case 47

5 – Slope of algorithms in average and worst known cases 48

6 – Number of comparisons using ascending input 49

7 – Number of comparisons using descending input 50

8 – Number of comparisons for PICK with different values of c 50

9 – Dromey's improvement to FIND 64

LIST OF FIGURES

Figure Page

1 – Binary tree representation of a heap-ified array 7

2 - Binary tree representation of a sorted array 8

3 – Cost of PICK for different values of c 35

4 – Average-case behavior for y: No. Comparisons 51

5 – Worst-known-case behavior for y: No. Comparisons 52

6 – Average-case behavior for y: No. Comparisons/N 53

7 – Worst-known-case behavior for y: No. Comparisons/N 54

8 – Worst-known-case behavior for y: No. Comparisons/N (w/o FIND) 55

9 – Average-case behavior for y: No. Comparisons/N log(N) 56

10 – Worst-known-case behavior for y: No. Comparisons/N log(N) 57

11 – Worst-known-case behavior for y: No. Comparisons/N log(N) (w/o FIND) 58

12 – Worst-known-case behavior for y: No. Comparisons/N^2 59

13 – Three different cases for PICK 60

14 – Three different cases for SELECT 61

15 – Three different cases for heapsort 62

16 – Three different cases for FIND 63

17 - Average case for FIND and Dromey's improvement/N 65

18 - Worst case for FIND and worst known case using Dromey's improvement/N^2 66

NOMENCLATURE

A .... The array of elements.

c .... In the context of PICK, the size of each group.

E( ) .... In the context of SELECT, it denotes the expected value of its argument.

F .... In the context of FIND, the pseudorandom number used as the index into A. The resulting element at that location is partitioned around. In the context of PICK, it denotes the pivot element's new position.

K .... Order statistic of the sought element.

ln(N) Natural logarithm.

log(N) Base-2- logarithm.

median .... The middle value in a distribution, above and below which lie an *equal* number of values.

N .... The number of elements in A.

o .... Little-oh. $T(N) = o(p(N))$ says that the growth rate of $T(N)$ is less than the growth of $p(N)$.

Θ .... Theta. Asymptotically tight lower and upper bounds. $T(N) = \Theta(h(N))$ says that the growth rate of $T(N)$ is equal to the growth rate of $h(N)$.

O .... Big-oh. $T(N) = O(f(N))$ says that the growth rate of $T(N)$ is less than or equal to the growth of $f(N)$.

P( ) .... In the context of SELECT, it denotes the probability of an event.

$i \,\theta\, A$ .... The ith smallest element of A, for $0 \le i < |A|$.

$x \,\rho\, A$ .... The rank of x in A, so that $x \,\rho\, A \,\theta\, A = x$.

__int64 .... In the source code for the algorithms, there are a few 64-bit variables of this type. It is Microsoft specific and has no ANSI equivalent.

Chapter 1. Introduction

The selection problem is that of finding the Kth smallest element of an array of N distinct elements. It is one of the most fundamental problems of computer science and it has been studied extensively. Selection is used as a building block in the solution of other fundamental problems such as sorting. A number of algorithms have been created to address the selection problem, each with its own strengths and weaknesses. Those of particular interest are Hoare's FIND algorithm [14], having an average-case running time of $\Theta(N)$ and a worst-case running time of $\Theta(N^2)$; Floyd and Rivest's SELECT algorithm [9], having an average-case running time of $\Theta(N)$ and a worst-case running time of $\Theta(N^2)$; Blum, Floyd, Pratt, Rivest and Tarjan's PICK algorithm [1], having both an average-case and worst-case running time of $\Theta(N)$. For comparative purposes, Floyd's Treesort 3 algorithm [8], having both an average-case and worst-case running time of $\Theta(N \log(N))$, will also be looked at.

Given an input array A of N distinct elements (we will deal later with duplicate elements) and a positive integer K, $0 <= K < N$, we want to determine the Kth smallest element of A and rearrange the input array such that this element is placed in A[K] and all elements with subscripts less than K have smaller values and all elements with subscripts greater than K have larger values. On completion, the following relationship is true:

$$A[0], \ldots, A[K-1] < A[K] < A[K+1], \ldots, A[N-1]$$

By definition, the Kth smallest element of N elements is an element which is less than N - K elements but greater than K - 1 other elements. In corollary, an element can't be the Kth smallest if it is greater than K elements or less than N – K + 1 elements.

One obvious method for addressing the selection problem would be to sort the whole input array in $\Theta(N \log(N))$ time using merge sort or heapsort [18] and then simply request the Kth element from the input array. While acceptable for small values of N, sorting quickly becomes inefficient for larger values of N.

A special case of the selection problem arises when the median element is sought. This occurs when $K = \lceil N/2 \rceil$. This is important because the median is a robust estimator of position whereas the mean, or average, is not; this means that the mean is greatly affected by *outliers*, while the median is not.

Chapter 2. The Algorithms

Heapsort

In 1962, Floyd created the initial version of "heapsort" under the name Treesort [7]. In 1964, Williams created an enhanced version under the now commonly used name heapsort [18]. Floyd created the final version in late 1964 under the name Treesort 3 [8]. All versions have an average-case and a worst-case running time of $\Theta(N \log(N))$.

Although Mergesort was created almost two decades earlier in 1945 by John von Neumann and also has a running time of $\Theta(N \log(N))$, we'll be using the Treesort 3 algorithm mainly because it is an *optimized* version of heapsort, has less memory requirements than Mergesort, which *can* require up to N additional units of storage depending on how it is implemented.

For comparison purposes, using Knuth's MIX machine, quicksort's running time is $8.08N \log_2(N)$ and heapsort's running time is $13N \log_2(N)$ in the average case for large values of N [16]. This means that heapsort takes 60.8% more time than quicksort and quicksort takes 37.8% less time than heapsort, in the average case for large values of N.

The heapsort algorithm is an in-place algorithm where only a constant number of elements are stored outside the input array at any given time.

A heap is a balanced binary tree with the property that the value of the root node is *less* than the value of its children. By reversing the sense of comparisons, an arbitrary input

array can be converted into a heap, with the value at the root of any subtree *greater* than its children. The array can then be sorted by successively deleting the first element in the heap (which will be the largest element in the array) and placing it in its correct position. The last element A[0] need not be deleted. Since all the elements except for one are in their correct positions, the last element must also be in its correct position. The heap can be placed in the same input array, because as the heap shrinks, the sorted part of the input array will grow.

A complete binary tree with N nodes (where N is 1 less than a power of 2) has $\log_2(N + 1)$ levels. Thus if each element of the input array were a leaf, requiring it to be filtered through the entire tree both while creating and adjusting the heap, the sort would still be $\Theta(N \log(N))$. Heapsort is far superior to quicksort in the worst case. In fact, it remains $\Theta(N \log(N))$.

Two phases are required to sort an input array using the heapsort algorithm. The building of the heap phase in $\Theta(N)$ time and the adjusting phase (i.e., moving the largest element to the end) in $\Theta(\log(N))$ time if duplicate elements are not allowed. This results in a total time of $\Theta(N \log(N))$.

Per Weiss [17], in order to remove elements from the heap, you would normally compare node X's children to X and swap the larger of the two children with X. This requires two comparisons of elements per level. An alternate way, suggested by Floyd [8], is to copy X to a temporary location and copy the larger child of X to the position of X, effectively

percolating a "hole" down to the bottom level. This requires only one comparison of elements per level. This would place the new element in a leaf and perhaps violate the heap property temporarily. Therefore, percolate the new element up in the normal fashion. Since the new element was previously a leaf, it probably will not percolate up even one level, and rarely more than one. Per Knuth [16], the probability that the new element will not move up at all, move up one level or move up two levels is 0.848, 0.135 and 0.016 respectively. Pseudo-code for the SiftUp routine, which is used in both phases, looks like the following:

```
SiftUp(A, i, N)
BEGIN
    TEMP ← A[i]
LOOP:
    j ← 2 * i + 1
    // IF THERE IS A LEFT CHILD
    IF j ≤ N THEN
    BEGIN
        // IF THERE IS A RIGHT CHILD
        IF j < N THEN
        BEGIN
            // NOTE THE GREATER OF THE TWO CHILDREN
            IF A[j + 1] > A[j] THEN
            BEGIN
                j ← j + 1
            END
        END
        // IF THE LARGER CHILD IS GREATER THAN NODE 'X'
        IF A[j] > TEMP THEN
        BEGIN
            // PROMOTE THE LARGER CHILD
            A[i] ← A[j]
            // START AGAIN WITH THE LARGER CHILD AS NODE 'X'
            i ← j
            GOTO LOOP
        END
    END
    A[i] ← TEMP
END
```

Example

Given the following input array of 10 elements, the following shows the progress of converting the array into a heap and then sorting the array. Recall that the first phase is to build the heap. This requires $\lfloor N / 2 \rfloor$ - 1 calls to the SiftUp() routine (see Appendix A). The initial input array looks like:

4	9	16	7	1	10	3	8	2	14

After calling SiftUp(A, 4, 10), the 4th and 9th elements are in their correct spots. Had there been 11 elements in the input array, the 4th element would have also been the parent of the 10th element.

4	9	16	7	**14**	10	3	8	2	**1**

After calling SiftUp(A, 3, 10), the 3rd, 7th and 8th elements are in their correct spots.

4	9	16	**8**	14	10	3	**7**	2	1

After calling SiftUp(A, 2, 10), no changes were made because the 2nd, 5th and 6th elements were already in their correct spots.

4	9	16	8	14	10	3	7	2	1

After calling SiftUp(A, 1, 10), the 1st, 3rd and 4th elements are in their correct spots.

4	**14**	16	8	**9**	10	3	7	2	1
0	1	2	3	4	5	6	7	8	9

With the exception of the root node, the input array is now heap-ified. The reason for the root node being in violation of the heap property is explained below. As mentioned

before, this phase has a running time of $\Theta(N)$. The heap can now be viewed as a binary tree:

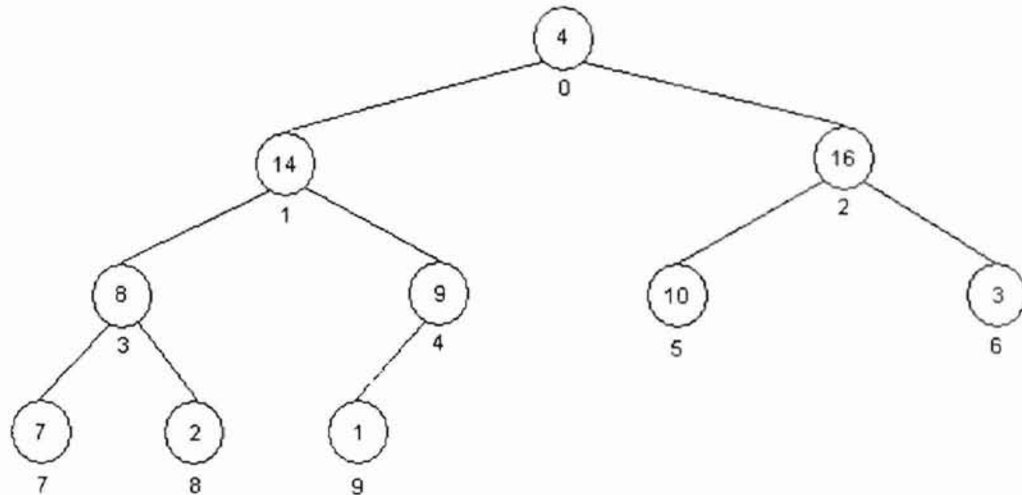


Figure 1 – Binary tree representation of a heap-ified array

With the exception of the root node, every other node is greater in value than each "child."

The second phase of this algorithm entails continually heap-ifying the root node (the 0th element) and exchanging its successor (the 1st or 2nd element) with the last element in the input array. With each successive call to SiftUp(), the input array shrinks from the right-end by decreasing the value of N by 1 since elements to the right of N are in their final spot. We will make N – 1 calls to the SiftUp() routine.

This phase has a best and worst-case running time of $\Theta(N \log(N))$ with Floyd's improvement [8]. Otherwise, the best-case behavior is $\Theta(N)$ if distinct elements are not specified.

The 9 calls to the SiftUp(A, 0, N) routine, with N ranging in value from 10 down to 1 have the following result:

1	14	10	8	9	4	3	7	2	16
2	9	10	8	1	4	3	7	14	16
7	9	4	8	1	2	3	10	14	16
3	8	4	7	1	2	9	10	14	16
2	7	4	3	1	8	9	10	14	16
1	3	4	2	7	8	9	10	14	16
2	3	1	4	7	8	9	10	14	16
1	2	3	4	7	8	9	10	14	16
1	2	3	4	7	8	9	10	14	16

The input array is now sorted with the binary tree representation looking like:

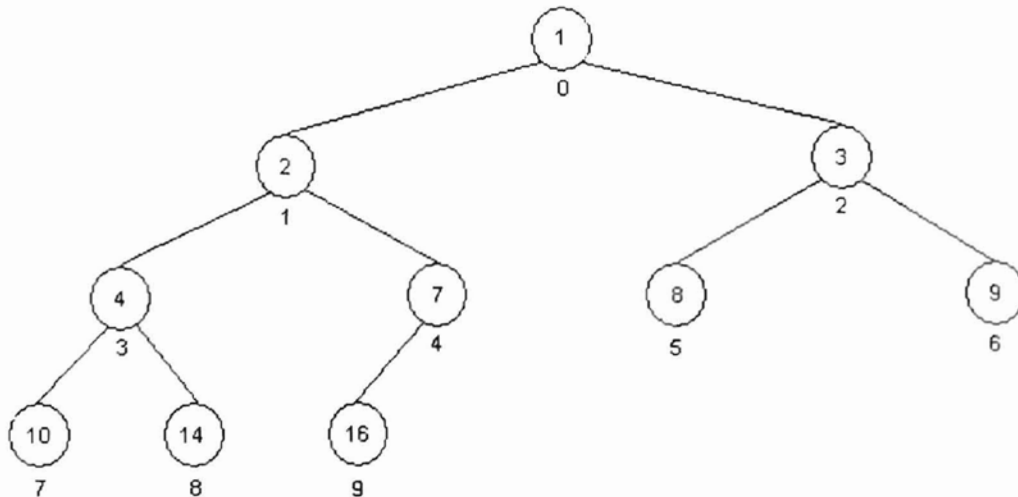


Figure 2 - Binary tree representation of a sorted array

FIND

In 1961, Hoare created the FIND algorithm [14] for finding the Kth smallest element, which has an average-case running time of $\Theta(N)$ and a worst-case running time of $\Theta(N^2)$. It bears resemblance to his quicksort algorithm [13] in that with each iteration of the

algorithm, the input array is partitioned around a pivot element but the FIND algorithm need only operate on the left *or* right sub-array, thus providing an improvement over quicksort's average-case running time of $\Theta(N \log(N))$. FIND saves time by not requiring a *full* sort.

Hoare implemented FIND using recursive calls, although this is not necessary, as will be discussed below.

A pivot element is *selected* and used to split the input array into two smaller sub-arrays and, recursively, the algorithm then operates on the sub-array containing the sought element. With each recursion, another element will be in its correct location with respect to the elements at its left and right. The algorithm makes no assumptions about the initial state of the input array. Pseudo-code is given below.

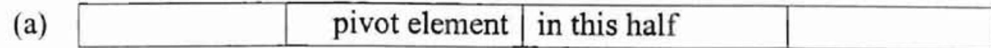
The worst-case running time of this algorithm for finding the median element occurs when the pivot element is the largest element in the input array (this is explained in more detail shortly), or the smallest if the comparisons are reversed. There will be no splitting and the time will become:

$$N + (N - 1) + (N - 2) + \ldots + \lceil N / 2 \rceil = \Theta(N^2)$$

For *odd* values of N, the maximum number of comparisons is $\frac{1}{2}N^2 + 2N - 2.5$. For *even* values of N, the maximum number of comparisons is $\frac{1}{2}N^2 + 2N - 3$.

The average-case running time depends on two possible cases:

(a) The pivot element is in the middle half of the input array. We would then recurse on at most 3N/4 elements because one element from either end will be discarded. This looks like:



(b) The pivot element is *not* in the middle so we would pessimistically recurse on at most N – 1 elements. This looks like:

Since each case occurs with 50% probability, the time will become (see Appendix B):

$$T(N) \le \frac{1}{2}T(N) + \frac{1}{2}T\left(3N/4\right) + N$$
$$T(N) \le T\left(3N/4\right) + 2N$$
$$T(N) = \Theta(N)$$

A more detailed analysis of the average case is provided by Knuth [16].

Analysis

At the start of the algorithm, we have an input array A[0:N - 1], index i is set to the value of the leftmost element and index j is set to the value of the rightmost element. We want to assign to A[K] the value it would have if A[0:N - 1] were sorted. The input array is then partitioned by scanning from the left to find an element A[i] > *pivot*, scanning from the right to find an element A[j] < *pivot*, exchanging them and continuing the process

until the indexes i and j either meet or cross. At the point that the indexes either meet or cross, one of three conditions must exist.

(a) Index j < index i <= K: elements A[0], …, A[j] are less than N – K + 1 other elements, so we need to find the $(K - i + 1)^{th}$ smallest element in the right-hand end A[j], …, A[N - 1] of the input array.

(b) K <= index j < index i: elements A[i], …, A[N - 1] are greater than K other elements, so we need to find the Kth smallest element in the left-hand end A[0], …, A[j] of the input array.

(c) Index j < K < index i: the Kth smallest element is in its final resting place and the algorithm is done.

Hoare describes the scanning process as moving

> "…lower valued elements of the array to the end – the *left-hand* end – and higher valued elements of the array to the other end – the *right-hand* end. This suggests that the array be scanned, starting at the left-hand end and moving rightward. Any element encountered that is small will remain where it is, but any element that is large should be moved up to the right-hand end of the array, in exchange for a small one. In order to find such a small element, a separate scan is made, starting at the right-hand end and moving leftward. In this scan, any large element encountered remains where it is; the first small element encountered is moved down to the left-hand end in exchange for the large element already encountered in the rightward scan.

Then both scans can be resumed until the next exchange is necessary. The process is repeated until the scans meet somewhere in the middle of the array. It is then known that all elements to the left of this meeting point will be small, and all elements to the right will be large. When this condition holds, we will say that they array is *split* at the given point into two parts."

So that we know what constitutes a lower-valued element or a higher-valued element, we need to select some value to compare with. One method is to select a pseudo-random number between index i and index j. Any element having a lower value is considered small; any element having a higher value is considered large.

Dromey [5] makes an observation regarding the way that FIND terminates, resulting in a marked improvement (i.e., reduced number) of comparisons in the average case and the worst case. He noted that if index j passes K on its way to meet up with index i, the current choice of A[K] was too small. Similarly, if index i passes K on its way to meet up with index j, the current choice of A[K] was too large. Therefore, termination can be applied when either j or i pass K rather than when i and j meet, thus avoiding unnecessary steps with a choice of A[K] that is known to be inappropriate. C source code is available in Appendix A.

Partitioning

The FIND algorithm is actually comprised of FIND itself and of another algorithm, PARTITION [12], which is where the scanning and exchanging take place. FIND makes

an initial call to PARTITION and, upon returning from that, calls itself recursively on either the left or right sub-array. Because of this recursion, stack overflow might prevent the algorithm from finishing if a near-worst-case input were encountered. A non-recursive solution is shown in chapter 4. The quicksort algorithm [13] also uses PARTITION. Pseudo-code for it looks like the following (see also Appendix A):

```
PARTITION(A, L, R)
BEGIN
    F ← a pseudorandom number between L and R
    X ← A[F]
    WHILE (L < R)
    BEGIN
        WHILE (A[L] <= X)
        BEGIN
            L ← L + 1
        END
        WHILE (A[R] >= X)
        BEGIN
            R ← R - 1
        END
        IF (L < R) THEN
        BEGIN
            SWAP A[L] WITH A[R]
            L ← L + 1
            R ← R - 1
        END
    END
    IF (L < F) THEN
    BEGIN
        SWAP A[L] WITH A[F]
        L ← L + 1
    END
    ELSE IF (F < R) THEN
    BEGIN
        SWAP A[F] WITH A[R]
        R ← R - 1
    END
END
```

Handling Duplicate Elements

Per Weiss [17], what if there is more than one element in the input array that is equal to the pivot element? Should index i stop when it encounters an element that is equal to the

pivot element, or should index j stop when it encounters an element that is equal to the pivot element, or both? They should do the same thing. Otherwise all of the equal elements end up in one sub-array rather than split evenly across both sub-arrays.

Consider the worst case where the input array contains all equal elements and index i and index j stop scanning, there will be a large number of swaps performed as the indexes make their way towards one another. However, they will eventually cross and the pivot element will be roughly in the middle. This produces a running time of $\Theta(N \log(N))$.

However, if neither index i or index j stop scanning (i.e., they are allowed to progress to the opposite end of the array without running off of the end) then no swaps are performed. This minimizes swaps but the pivot element will be set to the last element that index i referenced, creating a very uneven sub-array. This produces a running time of $\Theta(N^2)$. So, it is cheaper to perform the unnecessary swaps yielding even sub-arrays rather than risk creating very uneven sub-arrays. As is evident from Table 1 below, it quickly becomes an issue if equal elements are not handled efficiently.

N	N $\log_2(N)$	N^2	Ratio
10	33.21	100	2.01
100	664.38	10000	14.05
1000	9965.78	1000000	99.34
10000	132877.12	100000000	751.58
100000	1660964.00	10000000000	6019.60

Table 1 – Cost of duplicate elements

We can demonstrate the worst case for FIND by doing two things. The first is to pre-sort the input array in descending order. This has the effect of starting the larger elements to the left of the soon-to-be median and the smaller elements to the right of the soon-to-be median. Upon completion, this relation will be reversed. The second is to change the PARTITION routine such that the variable F is assigned the value of M rather than a pseudo-random number between M and N. Each time the PARTITION routine is called, index M is one greater than the previous call; thus the elements to the left of index M are in their correct location.

Example

In the following diagrams, we will track the progress of an input array A of numbers through the FIND algorithm. Index i will start at 0 and move rightward until it crosses index j; similarly, index j will start at N - 1and move leftward until it crosses index i. When the algorithm is complete, A[K] will contain the element it would have if A[0:N - 1] were sorted. The initial input array looks like:

40	29	19	27	35	11	26	32	8	3	39	44	2	15	38	31	45	62	90	76
i										K									j

The location F chosen for the pivot element is 1. During this first set of scans, we want to move index i rightward until we find an element that is greater than the pivot element X. After that, we want to move index j leftward until we find an element that is less than X. Once the qualifying elements are located, if the indexes i and j have not met, their respective elements are swapped and the indexes are moved one more time accordingly.

The value at index i was 40 and the value at index j was 15. Since index i was less than index j, this yields:

15	29	19	27	35	11	26	32	8	3	39	44	2	40	38	31	45	62	90	76
	F,i									K		j							

After the next *two* sets of scans, indexes i and j are brought closer together where they eventually meet. In the process, the out-of-order elements are swapped and the indexes moved accordingly. At this point, elements 32 and 3 are swapped and, after adjusting, indexes i and j have met.

15	29	19	27	2	11	26	**3**	8	**32**	39	44	35	40	38	31	45	62	90	76
	F							i,j		K									

After the next set of scans, index i and index j have crossed. We then swap elements at index F and index j and then decrement index j. This swap is done so that the pivot element (29) satisfies the condition of having only smaller elements to its left and larger elements to its right.

15	**8**	19	27	2	11	26	3	**29**	32	39	44	35	40	38	31	45	62	90	76
	F						j		i	K									

Since index i was less than K (a), FIND starts the scanning process again on the right-hand end of the input array of size N – i + 1. At this point we have eliminated 9 elements from any further scanning. Our input array now looks like:

15	8	19	27	2	11	26	3	29	32	39	44	35	40	38	31	45	62	90	76
									i	K									j

The value chosen for F is 18. During this first set of scans, index i and index j have met. We then swap elements at index F and index j and then decrement index j. This swap is done so that the pivot element (90) satisfies the condition of having only smaller elements to its left and larger elements to its right.

15	8	19	27	2	11	26	3	29	32	39	44	35	40	38	31	45	62	**76**	**90**
										K								F,j	i

Since index j was greater than K (b), FIND starts the scanning process again on the left-hand end of the input array of size j – K + 1. At this point we have eliminated 1 element from any further scanning. Our input array now looks like:

15	8	19	27	2	11	26	3	29	32	39	44	35	40	38	31	45	62	76	90
									i	K								j	

The value chosen for F is 13. During the first set of scans, FIND swaps the pair of elements that are out of order and moves the indexes accordingly. This yields:

15	8	19	27	2	11	26	3	29	32	39	**31**	35	40	38	**44**	45	62	76	90
										K		i	F	j					

After the next set of scans, index i and index j have crossed. We then swap elements at index F and index j and then decrement index j. This swap is done so that the pivot

element (40) satisfies the condition of having only smaller elements to its left and larger elements to its right.

15	8	19	27	2	11	26	3	29	32	39	31	35	**38**	**40**	44	45	62	76	90
										K			F,j		i				

Since index j was greater than K (b), FIND starts the process again on the left-hand end of the input array of size j – K + 1. At this point we have eliminated 5 elements from any further scanning. Our input array now looks like:

15	8	19	27	2	11	26	3	29	32	39	31	35	38	40	44	45	62	76	90
									i	K			j		i				

The value chosen for F is 9. During the first set of scans, FIND swaps the pair of elements that are out of order and moves the indexes accordingly. This yields:

15	8	19	27	2	11	26	3	29	32	**31**	**39**	35	38	40	44	45	62	76	90
									F	K,j	i				i				

After the next set of scans, index i and index j have crossed. We then swap elements at index F and index j and then decrement index j. This swap is done so that the pivot element (32) satisfies the condition of having only smaller elements to its left and larger elements to its right.

15	8	19	27	2	11	26	3	29	**31**	**32**	39	35	38	40	44	45	62	76	90
									F,j	K	i				i				

Since neither K is less than or equal to index j nor is index i less than or equal to K (c), FIND is complete and A[K] now contains the element it would have if the input array were sorted. The final input array now looks like:

15	8	19	27	2	11	26	3	29	31	**32**	39	35	38	40	44	45	62	76	90
										K									

In order to determine how many comparisons were required to get A[K] in its rightful place, we will consider a comparison to be when any element A[i] or A[j] is compared to the pivot element. This input array generated 53 comparisons. C source code for this algorithm is located in Appendix A.

SELECT

In 1975, Floyd and Rivest created the SELECT algorithm [9], which, although the average-case running time is also O(N) and the worst-case running time is still O(N^2), is "...very efficient on the average, both theoretically and practically." It is functionally equivalent to Hoare's FIND algorithm [15] but significantly faster on the average due to the effective use of sampling to determine the element K about which to partition A. The partitioning loop is also slightly faster (on most machines) since subscript range checking is eliminated.

Sampling

The biggest difference between this algorithm and the FIND algorithm [14] is the use of selective sampling. Sampling is a process of selectively partitioning on a *subset* of the input array and then partitioning on the input array itself (i.e., getting a cheap estimate of the final value which subsequently makes the main algorithm faster).

Their experimentation found that sampling should only be done when N > 600 due to the expense of computing square roots, logarithms, etc., which cost more than they are worth for small N. Pseudo-code for the sampling portion of SELECT looks like the following (see also Appendix A):

```
SELECT(A, L, R, K)
BEGIN
    IF (|A| > 600) THEN
    BEGIN
        N ← |A|
        I ← K - L + 1
        Z ← ln(N)
        S ← 0.5 * exp(2 * Z / 3)
        SD ← 0.5 * sqrt(Z * S * (N-S) / N) * sign(I - N/2)
        LL ← max(L, K - I * S / N + SD)
        RR ← min(R, K + (N-I) * S/N + SD)
        SELECT(A, LL, RR, K)
    END
    ...
END
```

where N is the number of elements in the array A; I is the number of elements between the leftmost element L and the sought element K; S is the size of the sample; LL is the leftmost boundary of the sample S; RR is the rightmost boundary of the sample S.

Analysis

It is desired that three conditions be met when choosing the left and right boundaries of the sample A. Two elements, u and v ($u < v$), are selected from A using the algorithm recursively, such that the set $\{x \in A \mid u <= x <= v\}$ is expected to be size $o(n)$ and yet expected to contain $i \theta A$, that is, the ith smallest element in A. Selecting u and v partitions A into those elements less than u (set A_1), those elements between u and v (set B_1) and those elements greater than v (set C_1).

The three conditions are:

(a) expected value $E(u \rho A) <= i <= E(v \rho A)$, that is, u's rank in A is less than or equal to i is less than or equal to v's rank in A.

(b) $E(|B|) = E(v \rho A) - E(u \rho A)$ is $o(n)$

(c) probability $P(i < u \rho A \text{ or } i > v \rho A) = o(n^{-1})$ (i.e., extremely unlikely as N approaches ∞). Using $(\ln(N))^{1/2}$, or $\sqrt{\ln(N)}$, will ensure this.

The algorithm is laid out as follows:

1. Draw a random sample S_1 of size s_1 from A, and select u_1 and v_1 using this algorithm recursively.
2. Determine the sets A_2, B_2, and C_2, a partition of S_2, by comparing each element in $S_2 - S_1$ to u_1 and v_1.
3. Next, determine u_2 and v_2 by applying this algorithm recursively to B_2 (in the most likely case; else A_2 or C_2).
4. Extend the partition of S_2 determined by u_2 and v_2 into a partition A_3, B_3, C_3 of S_3 by comparing each element of $S_3 - S_2$ to u_2 and v_2.

5. Continue in this fashion until partition A_k, B_k, C_k of the set $S_k = A$ has been created.
6. Then use the algorithm recursively once more to extract $i \theta A$ from B_k (or A_k or C_k, if necessary).

Floyd and Rivest [9] state that

> "Any reduction in the complexity of partitioning will show up as a significant increase in the efficiency of the whole algorithm. The basic algorithm, however, requires partitioning X about both u and v simultaneously into three sets A, B, and C, an inherently inefficient operation. On the other hand, partitioning X completely about one of u or v before beginning the partition about the other can be done very fast. We therefore use an improved version of Hoare's PARTITION algorithm [12] to do the basic partitioning. A further (minor) difference is that after partitioning has been completed about one element another sample is drawn to determine the next element about which to partition. This permits a very compact control structure at little extra cost." (Floyd and Rivest use X to denote the array that we call A).

The algorithm uses two loops: an outer loop that controls the left L and right R boundaries and an inner loop that controls the elements A[i] and A[j] which surround the elements that have yet to be partitioned around the pivot element. Prior to the inner, partitioning loop, the pivot element is swapped with element A[L]. At that point, one of the following relations will hold true:

(a) A[K] is *greater than or equal to* A[R]

(b) A[K] is *less than* A[R]

If (a), then the pivot element will remain at A[L]. If (b), the pivot element will be moved to A[R]. At this point the relation A[L] > A[R] is true. This has to be the case because inside the partitioning loop, these two elements are swapped such that the smaller element is in A[L].

After the partitioning loop, the pivot element will either be in A[L] or A[R]. If the former, the pivot element is swapped with element A[j], which is less than or equal to the pivot element. Otherwise, the pivot element is swapped with element A[j + 1], which is greater than or equal to the pivot element. At that point, one or both of the following relations will hold true:

(a) Index j is *less than or equal to* K

(b) K is *less than or equal to* index j

If (a), then the sought item lies in the right side so the left boundary L is adjusted. If (b), then the sought item lies in the left side so the right boundary R is adjusted.

Example

Let's track the progress of the same set of 20 numbers through SELECT. When the algorithm is complete, A[K] will contain the element it would have if A[0:N - 1] were sorted. The input array initially looks like:

40	29	19	27	35	11	26	32	8	3	39	44	2	15	38	31	45	62	90	76
L										K									R

Prior to the partitioning loop, it was found that the pivot element (39) was less than A[L] and A[R], so it is moved to A[R].

76	29	19	27	35	11	26	32	8	3	40	44	2	15	38	31	45	62	90	39
L,i										K									R,j

The first pass through the partitioning loop will move the pivot element to A[L] and partition the remaining elements around that value. At this point, elements A[L], …, A[j] are less than the pivot element (39) and elements A[i], …, A[R] are greater than the pivot element.

39	29	19	27	35	11	26	32	8	3	31	38	2	15	44	40	45	62	90	76
L										K			j	i					R

Since the pivot element remained at A[L] during the partitioning loop, it must be moved to its correct location such that the relation

$A[L], \ldots, A[j-1] < A[j] < A[j+1], \ldots, A[R]$ is true. This yields:

15	29	19	27	35	11	26	32	8	3	31	38	2	39	44	40	45	62	90	76
L										K			j	i					R

After the partitioning loop, K was found to be less than or equal to index j so the sought element is in the left side, thus index R needs adjusting. At this point we have eliminated 7 elements from any further processing.

The new pivot element (31) is greater than elements A[L] and A[R], so it is moved to A[L] prior to the partitioning loop.

31	29	19	27	35	11	26	32	8	3	15	38	2	39	44	40	45	62	90	76
L,i										K		R,j							

The first pass through the partitioning loop will move the pivot element to A[R] and partition the remaining elements around that value. At this point, elements A[L], …, A[j] are less than the pivot element (31) and elements A[i], …, A[R] are greater than the pivot element.

2	29	19	27	15	11	26	3	8	32	35	38	31	39	44	40	45	62	90	76
L								j	i	K		R							

Since the pivot element remained at A[R] during the partitioning loop, it must be moved to its correct location such that the relation

$A[L], \ldots, A[j-1] < A[j] < A[j+1], \ldots, A[R]$ is true. This yields:

2	29	19	27	15	11	26	3	8	31	35	38	32	39	44	40	45	62	90	76
L									j,i	K		R							

After the partitioning loop, index j was found to be less than K so the sought element is in the right side, thus index L needs adjusting. We have eliminated 10 elements from any further processing.

The new pivot element (35) is greater than elements A[L] and A[R] so it is moved to A[L] prior to the partitioning loop.

2	29	19	27	15	11	26	3	8	31	35	38	32	39	44	40	45	62	90	76
										L,K,i		R,j							

The first pass through the partitioning loop will move the pivot element to A[R] and partition the remaining elements around that value. At this point, elements A[L], …, A[j] < the pivot element (35) and elements A[i], …, A[R] > the pivot element.

2	29	19	27	15	11	26	3	8	31	32	38	35	39	44	40	45	62	90	76
										L,K	j,i	R							

Since the pivot element remained at A[R] during the partitioning loop, it must be moved to its correct location such that the relation A[L], …, A[j – 1] < A[j] < A[j + 1], …, A[R] is true. This yields:

2	29	19	27	15	11	26	3	8	31	32	35	38	39	44	40	45	62	90	76
										R,K,L	j,i								

Since index L is no longer less than index R, SELECT is complete and A[K] now contains the element it would have if the input array were fully sorted.

This example generated 17 comparisons. C source code for this algorithm is located in Appendix A.

PICK

In 1973, Blum et al. created the PICK algorithm [1], in which the average-case running time and the worst-case running time are both $\Theta(N)$. This is made possible by eliminating roughly 0.3N elements for the final recursive call to the algorithm. It operates by recursively discarding elements of A which are known to be too small or too large to be the median, until only the median remains. PICK is often called a "median of medians" algorithm.

Analysis

We want to select the Kth element of A where $N = |A|$ and $0 <= K < N$. Here is the algorithm as outlined by its authors [1]:

1. (Select an element $m \in A$):
   a. Arrange A into $\frac{N}{c}$ groups of length c, and sort each group.
   b. Select $m = b \ \theta \ T$, where $T =_{def}$ the set of $\frac{N}{c}$ elements which are the kth smallest element from each group. Use PICK recursively if $\frac{N}{c} > 1$.

2. (Compute $m \rho A$): Compare m to every other element x in A for which it is not yet known whether $m < x$ or $m > x$.
3. (Discard or halt): If $m \rho A = i$, halt (since $m = i \theta A$), otherwise if $m \rho A > i$, discard $D = \{x \mid x >= m\}$ and set $N \leftarrow N - |D|$, otherwise discard $D = \{x \mid x <= m\}$ and set $N \leftarrow N - |D|$, $i \leftarrow i - |D|$.

Return to step #1.

In addition, here is a further explanation of each of the above steps with c = 5:

1. If $N \leq 5$, sort the elements and return the middle, or median, element.

   In addition to the $\left\lfloor \frac{N}{5} \right\rfloor$ groups, there may be at most one group of the remaining N mod 5 elements.

2. Select the median element from each group into a temporary array T. This will require $\left\lceil \frac{N}{5} \right\rceil$ extra units of memory. Select the median-of-medians element m by calling PICK recursively using T.

3. Partition the input array around the median element found in step #2 such that the following relation is true:

   $A[0], \ldots, A[K-1] < A[K] < A[K+1], \ldots, A[N-1]$

4. If the sought element is in the left side of the input array, return to step #1 using the left side. If the sought element is in the right side of the input array, return to step #1 using the right side. Otherwise halt, since the sought element is now in A[K].

Selecting the Median of 5 elements

In the preceding steps, it says to sort each of the $\left\lceil \frac{N}{5} \right\rceil$ groups, with a group containing no more than 5 elements, and then select the middle element of those 5 as the median. If N is an even number, the larger of the two medians is selected. To fully sort the group is not necessary, since it would require at most 7 comparisons, when all we want is to find the median of the group. This can be done with at most 6 comparisons. For example, given a group containing 5 elements with the elements labeled A, B, C, D and E, median selection is as follows:

1. Compare A with B, swap if necessary.
2. Compare C with D, swap if necessary.
3. Compare A with C, swap A with C and B with D if necessary.
4. At this point, $A > B$ and $A > C > D$. Because A is greater than at least 3 elements, it cannot be the median.
5. Compare B with E, swap if necessary.
6. Compare B with C, swap B with C and D with E if necessary.
7. At this point, $B > E$ and $B > C > D$. Because B is greater than at least 3 elements, it cannot be the median. Similarly, because D is less than at least 3 elements, it cannot be the median.
8. Now just compare C with E and select the lesser of the two as the median of the group.

Discarding Elements

How many elements are discarded each time? After the input array has been partitioned around the median m, it has several deterministic properties. For example, if N = 35, the input array could be represented as [17]:

T	T	T	T	.	.	.	
T	T	T	T	.	.	.	
S	S	S	m	L	L	L	← medians
.	.	.	H	H	H	H	
.	.	.	H	H	H	H	

The medians that are smaller than m are denoted by S and the medians that are larger than m are denoted by L. Since each group has 5 elements, there are two elements that are smaller than a small median S and two elements that are larger than a large median L. These elements are denoted by T(iny) and H(uge) respectively. There are also 2 elements *larger* than m and 2 elements *smaller* than m.

Let K denote the medians of type S and L. For each K of type S, there are 2K elements of type T and for each K of type L, there are 2K elements of type H.
The group that m itself is in also has 2 elements of type T and 2 elements of type H.

So, there are 2K + 2 elements of type T and 2K + 2 elements of type H. Thus, there are 3K + 2 elements that are guaranteed to be smaller than m and 3K + 2 elements that are guaranteed to be larger than m. In our example, K = 3, so there are 8 elements of type T

and 8 elements of type H, for a total of 11 elements guaranteed to be smaller than m and 11 elements guaranteed to be larger than m.

That accounts for 22 of the 35 elements, so what is known of the remaining 13? One of those is m itself, so the remaining 4K elements are unknown as to their relationship to m; they could be smaller or larger and thus will be included in the final recursive call to the algorithm.

Therefore N is in the form 10K + 5 since

$$2(3K + 2) + 2(2K) + 1 =$$

$$(6K + 4) + 4K + 1 =$$

$$10K + 5$$

One of the 3K + 2 groups will be discarded since it is known to be either too small or too large, so the final recursive call to the algorithm will be on at most $7K + 2 < 0.7N$ elements. Note that it is not 7K + 3 because the median m is not included in subsequent processing.

If N is not evenly divisible by 5 (i.e., having one incomplete group), we can calculate the minimum number of elements that are discarded using the formula from [3]. Of the $\left\lceil \frac{N}{5} \right\rceil$ groups, at least half of those have medians that are greater than m. Discounting the group containing m and the one group that has less than 5 elements if N is not evenly divisible by 5, the remaining groups contribute 3 elements that are greater than or equal to m.

Therefore, the total number of elements greater than m is at least $\frac{3N}{10} - 6$. Also, the total number of elements less than m is at least $\frac{3N}{10} - 6$. So, in the worst case, the algorithm is called recursively on at most $\frac{7N}{10} + 6$ elements since $\left(\left\lfloor \frac{3N}{10} \right\rfloor - 6\right) + \left(\left\lfloor \frac{7N}{10} \right\rfloor + 6\right) + 1 = N$.

For example, if N = 35, we can visualize the discarded elements using:

T	T	T	T	.	.	.
T	T	T	T	.	.	.
S	S	S	m	L	L	L
.	.	.	H	H	H	H
.	.	.	H	H	H	H

with the $\frac{3N}{10} - 6$ elements shaded in gray.

To show how the *at-most* and *at-least* formulas compare to each other, the following table is helpful:

N	$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{N}{5} \right\rceil \right\rceil - 2\right)$ (at least)	$\frac{3N}{10} - 6$ (at least)	$K = \frac{1}{2}\left\lceil \frac{N}{5} \right\rceil$ $3K + 2$ (at most)
36	6	4.8	14
37	6	5.1	14
38	6	5.4	14
39	6	5.7	14
40	6	6	14
41	7.5	6.3	14
42	7.5	6.6	14
43	7.5	6.9	14
44	7.5	7.2	14
45	7.5	7.5	14
46	9	7.8	17

Table 2 – Comparison of at-least and at-most formulas

Of the two recursive calls made to the algorithm, the first is made after the medians of the $\left\lceil \frac{N}{5} \right\rceil$ groups are selected. This recursive call selects the median of those medians. The size of this call is 0.2N or $\left(\frac{N/5}{N}\right)N = \frac{N}{5}$.

Why Groups of 5?

The analysis thus far has used a group size of c=5, but that's not to imply that other values of c cannot be used. In fact, this value was chosen because it is the smallest value that can be used that produces Θ(N) worst-case behavior. Why can't 3 be used? Looking at the formula in Appendix B, if c = 3 and d = 2, we would have

$P(n) \leq \left(\frac{2 \times (h(c) + c)}{d - 2} \right) \times n$ which would yield a divide-by-zero error. We can also view this from a slightly different angle. During the simplifying of

$P(n) \leq \frac{3n}{3} + P\left(\frac{n}{3}\right) + n + P\left(\frac{3n}{6}\right)$, we eventually get to $\frac{6}{6}P(n) - \frac{6}{6}P(n) \leq \frac{6}{3}n$ which is an invalid equation since 0 would end up by itself on the left side. Therefore, 3 cannot be used as the group size.

Is there an optimal group size? Using the formula from Appendix B with different values for c, Figure 3 shows the cost relationship with the optimal value for c being 11, which differs from the authors' choice of c = 21. The odd numbers 5 through 23 were chosen for comparison purposes.

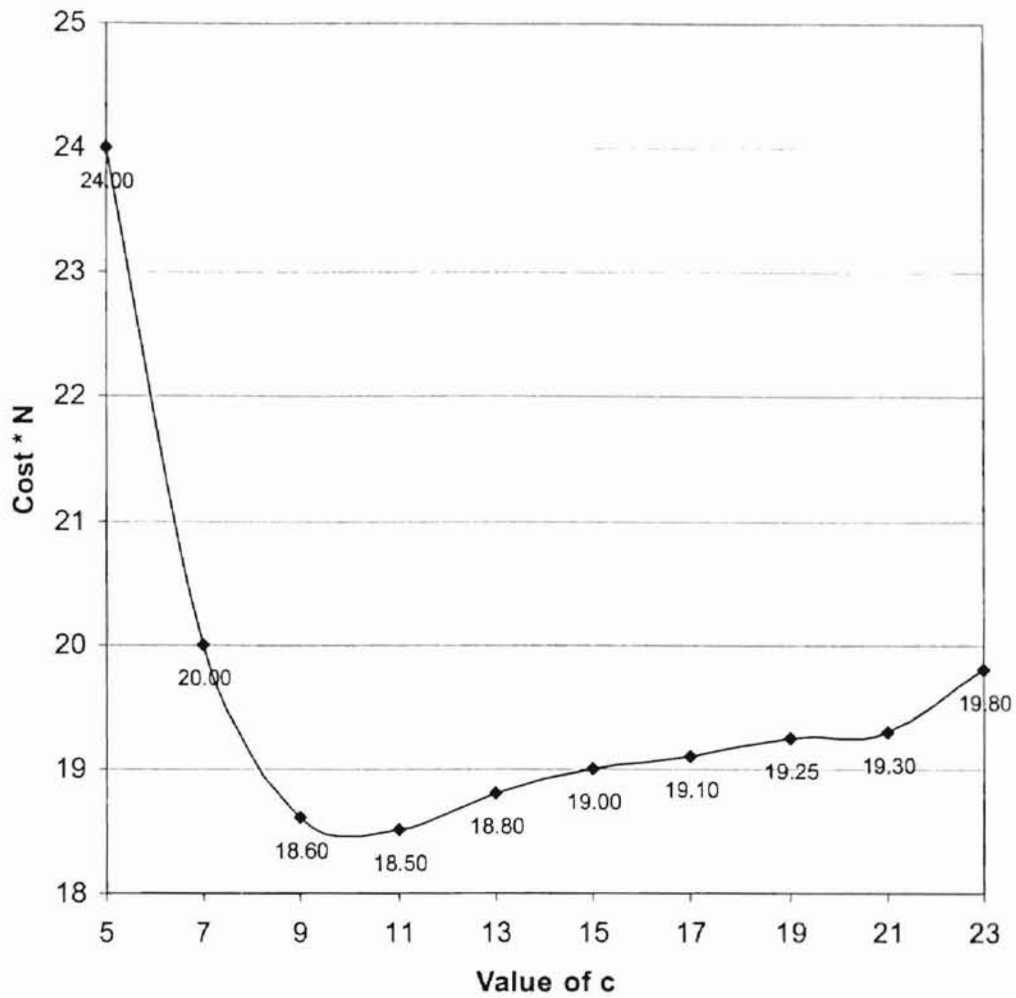


Figure 3 – Cost of PICK for different values of c

Example

Following the same set of 20 numbers through PICK yields the following. The initial input array looks like:

40	29	19	27	35	11	26	32	8	3	39	44	2	15	38	31	45	62	90	76

Since N = 20, the input array is broken up into $\left\lceil \frac{N}{5} \right\rceil$ groups of 5 elements each. The medians of each of those groups are selected and the median-of-medians is selected via a recursive call to PICK. The input array now looks like the following, with the medians shaded and the median-of-medians denoted by m:

19	27	**29**	40	35	8	3	**11**	26	32	2	15	39	44	**38**	31	45	**62**	90	76
														m					

Using 38 as the pivot element, we will now partition around that element, similar in fashion to both FIND and SELECT. The indexes i and j start at positions one past the left-most element and one past the right-most element respectively.

	19	27	29	40	35	8	3	11	26	32	2	15	39	44	38	31	45	62	90	76	
i											K				m						j

During the first set of scans, the out-of-order elements are swapped, yielding:

19	27	29	**31**	35	8	3	11	26	32	2	15	39	44	38	**40**	45	62	90	76
			i							K				m	j				

After the next set of scans, the out-of-order elements are swapped. Because one of the elements to be swapped is also the pivot element, we note its *new* position using F.

19	27	29	31	35	8	3	11	26	32	2	15	**38**	44	**39**	40	45	62	90	76
										K		F,i		m,j					

After the next set of scans, index i and index j have crossed. Because the pivot element was moved from its original location, we must ensure that the condition of only having smaller elements to its left and larger elements to its right exists. It does, *but* index j must be moved to the right to reflect how many elements are less than the pivot element.

19	27	29	31	35	8	3	11	26	32	2	15	38	44	39	40	45	62	90	76
										K		F	j,i	m					

The input array has now been partitioned around the pivot element 38. Since K was less than index j, we will recursively call PICK on the left-hand end of the input array. At this point we have eliminated N – j, or 7, elements from any further processing. The input array now looks like:

19	27	29	31	35	8	3	11	26	32	2	15	38	44	39	40	45	62	90	76

Since N = 13, the input array is broken up into $\left\lceil \frac{N}{5} \right\rceil$ groups of 5 elements each. The medians of those groups are selected and the median-of-medians is selected via a recursive call to PICK. The input array now looks like the following, with the medians shaded and the median-of-medians denoted by m:

19	27	**29**	31	35	3	8	**11**	26	32	2	**15**	38	44	39	40	45	62	90	76
										K	m								

Using 15 as the pivot element, we will now partition around that element. The indexes i and j start at positions one past the left-most element and one past the right-most element respectively.

	19	27	29	31	35	3	8	11	26	32	2	15	38	44	39	40	45	62	90	76
i											K	m		j						

During the first set of scans, the out-of-order elements are swapped. Because one of the elements to be swapped is also the pivot element, we note its *new* position using F. This yields:

15	27	29	31	35	3	8	11	26	32	2	**19**	38	44	39	40	45	62	90	76
F,i										K	m,j								

After the next *four* sets of scans, the out-of-order elements are swapped. The input array now looks like:

15	2	11	8	**3**	**35**	31	29	26	32	27	19	38	44	39	40	45	62	90	76
F				i	j					K	m								

After the next set of scans, index i and index j have crossed. Because the pivot element was moved from its original location, we must ensure that the condition of only having smaller elements to its left and larger elements to its right exists. The elements at positions A[F] and A[j] are swapped. Elements up to A[j] are less than the pivot element.

3	2	11	8	**15**	35	31	29	26	32	27	19	38	44	39	40	45	62	90	76
F				j	i					K	m								

The input array has now been partitioned around the pivot element 15. Since K was greater than index j, we will recursively call PICK on the right-hand end of the input array. At this point we have eliminated j + 1, or 5, elements from any further processing. The input array now looks like:

3	2	11	8	15	35	31	29	26	32	27	19	38	44	39	40	45	62	90	76

Since N = 8, the input array is broken up into $\left\lceil \frac{N}{5} \right\rceil$ groups of 5 elements each. The medians of each of those groups are selected and the median-of-medians is selected via a recursive call to PICK. The input array now looks like the following, with the medians shaded and the median-of-medians denoted by m:

3	2	11	8	15	26	29	**31**	35	32	19	**27**	38	44	39	40	45	62	90	76
							m			K									

Using 31 as the pivot element, we will now partition around that element. The indexes i and j start at positions one past the left-most element and one past the right-most element respectively.

3	2	11	8	15	26	29	31	35	32	19	27	38	44	39	40	45	62	90	76
				i			m			K			j						

During the first set of scans, the out-of-order elements are swapped. Because one of the elements to be swapped is also the pivot element, we note its *new* position using F. This yields:

3	2	11	8	15	26	29	**27**	35	32	19	**31**	38	44	39	40	45	62	90	76
							m,i			K	F,j								

After the next set of scans, the out-of-order elements are swapped.

3	2	11	8	15	26	29	27	**19**	32	**35**	31	38	44	39	40	45	62	90	76
							m	i		K,j	F								

After the next set of scans, index i and index j have crossed. Because the pivot element was moved from its original location, we must ensure that the condition of only having smaller elements to its left and larger elements to its right exists. The elements at positions A[i] and A[F] are swapped and index j moved to the right to reflect how many elements are less than the pivot element.

3	2	11	8	15	26	29	27	19	**31**	35	**32**	38	44	39	40	45	62	90	76
							m		j,i	K	F								

The input array has now been partitioned around the pivot element 31. Since K was greater than index j, we will recursively call PICK on the right-hand end of the input array. At this point we have eliminated j + 1, or 5, elements from any further processing. The input array now looks like:

3	2	11	8	15	26	29	27	19	31	35	32	38	44	39	40	45	62	90	76

Since N = 3, we simply need to sort those elements, thus placing the median of the input array at position A[K], which is where it would be if the input array were fully sorted. The algorithm is done, with the final input array looking like:

3	2	11	8	15	26	29	27	19	31	32	35	38	44	39	40	45	62	90	76
										K									

Improvements

While we don't go into the details here, it is worth mentioning that Dor and Zwick [4] improved these results and presented a selection algorithm that uses at most 2.95N comparisons. Their work slightly narrows the gap between the best known lower and upper bounds on the comparison complexity of finding the median.

In order to obtain their results for finding the median, many new ideas were required, with the central idea being *green factories* and an *amortized* analysis of their production costs. Factories are used for the mass production of certain partial orders at a much-reduced cost.

According to the authors,

> "The performance of a green factory is mainly characterized by two parameters A_0 and A_1 (the *upper* and *lower* element costs). Using a green factory with parameters A_0 and A_1 we obtain an algorithm for the selection of the αn-th

> element using at most $(A_0\alpha + A_1(1 - \alpha)) * n + o(n)$ comparisons. To select the median, we use a factory with $A_0, A_1 \approx 2.95$. Actually, there is a tradeoff between the lower and upper costs of a factory. For every $0 < \alpha \leq ½$ we may choose a factory that minimizes $A_0\alpha + A_1(1 - \alpha)$. We can select the n/4-th element, for example, using at most 2.69n comparisons, by using a factory with $A_0 \approx 4$ and $A_1 \approx 2.25$. In this paper, we concentrate on factories for median selection."

They go on to say that it is easy to verify that their algorithm as described above, as the median finding algorithm of Blum et al. [1], can be implemented in linear time in the RAM model.

Chapter 3. Methods of Testing and Comparisons

In order to minimize the possibility of the input array having one or more elements that match the pivot element, the following C code will be employed to generate pseudo-random elements.

```
#define A 48271L        // multiplier
#define M 2147483647    // modulus
#define Q (M / A)
#define R (M % A)

static int nSeed = 1;

int Random( void )
{
    int nTempSeed;

    nTempSeed = A * (nSeed % Q) - R * (nSeed / Q);
    if (nTempSeed >= 0)
        nSeed = nTempSeed;
    else
        nSeed = nTempSeed + M;

    nTempSeed = ((double) nSeed / M) * 1000000000;
    return (nTempSeed);
}
```

In order to achieve the average case for heapsort, FIND, SELECT or PICK, each algorithm is applied to an input array of N "fresh" elements. This is done nSets times (nSets=100 was used) for each value of N. Then the average of those were taken. In addition, when a *worst-case* array is found, it is saved for later use. Pseudo-code for this looks like the following (see also Appendix A):

```
N_START ← 10
N_END ← 40960
NSETS ← 100
FOR N = N_START TO N_END STEP ×2
    FOR C = 1 TO NSETS STEP 1
        GENERATE INPUT ARRAY A1 AND SAVE IT FOR LATER USE
        CALL PICK
        ACCUMULATE COMPARISON COUNTS
```

```
        IF COMP COUNT > MAX COMP COUNT THEN
            MAX COMP COUNT ← COMP COUNT
            A2 ← A1 (WORST-CASE INPUT ARRAY SO FAR)
        END
    NEXT C
    REPORT AVERAGE COMPARISON COUNTS FOR CURRENT N
NEXT N
FIND WORST-CASE USING A2 AS STARTING POINT
```

In order to achieve the worst-known-case scenarios for the algorithms, three different methods were employed. This was done for SELECT and PICK only, because their worst cases are complicated. The worst case for FIND is known and simple.

The first method uses the worst of the average-case input arrays as a starting point. This guarantees that the worst-known-case input array will perform at least as many comparisons as the worst average-case input array. Using the worst of the average-case input arrays, we pseudo-randomly swapped elements nSwaps times (typically nSwaps=100,000), trying to find a new worst-known-case input array that generated more comparisons. If a new worst-known-case input array is found, a copy is made for later use. Each iteration through the loop will be using the last-known worst-known-case array as a starting point before swapping two elements and calling the algorithm. That way we never lose the worst case found so far. This is referred to as the Monte Carlo method. Pseudo-code for this looks like the following:

```
N_SWAPS ← 100000
MAXIMUM ← 0
FOR X = 1 TO N_SWAPS
    # OF COMPARISONS ← 0
    A2 ← "BEFORE" COPY OF THE INPUT ARRAY FROM A
    CALL PICK
    IF # OF COMPARISONS IS GREATER THAN MAXIMUM
        MAXIMUM ← # OF COMPARISONS
        A3 ← COPY OF NEW WORST-CASE INPUT ARRAY FROM A2
```

```
        END
        A ← COPY OF WORST-CASE INPUT ARRAY FROM A3
        R1 ← RANDOM NUMBER MOD N
        R2 ← RANDOM NUMBER MOD N
        SWAP ELEMENTS AT POSITION R1 AND R2
    NEXT X
```

Some sorting and selection algorithms perform well with random data but poorly with pre-ordered data. So, the second and third methods simply involve generating an input array with N pseudo-random elements and sorting it once in ascending order and taking the results, then sorting it again in descending order and taking the results. In the case of FIND, a change is made to PARTITION such that F ← M instead of F ← *random number between M and N*. Whichever of these three methods caused the most comparisons was the one used for the worst known case.

In order to achieve the worst case behavior for FIND, the pivot element will always be the left-most (i.e., smallest) element in the input array instead of a pseudo-random number between the left and right ends. So, each time PARTITION is called from FIND, the pivot element would be the value of A[0]. This achieves the desired result because N – 1 elements are being partitioned around the smallest or largest element at that point.

Chapter 4. Results

In Table 3 below, the comparisons for the average-case behavior are listed for each algorithm. For all values of N, SELECT is the best performing algorithm while, as expected, heapsort is the worst of the four algorithms, since it is $\Theta(N \log(N))$ in the average case.

Average Case: Number of Comparisons				
N	**Heapsort**	**FIND**	**SELECT**	**PICK (c=5)**
10	21	30	23	39
20	81	93	70	128
40	237	220	180	344
80	626	472	388	827
160	1561	999	833	1857
320	3743	2034	1715	4040
640	8742	4178	3796	8634
1280	20016	8659	7040	18230
2560	45126	17435	13351	37974
5120	100459	34168	25514	77750
10240	221355	68427	49368	158792
20480	483605	139356	97051	324161
40960	1049045	275876	183502	655212

Table 3 – Number of comparisons for average case

In Table 4 below, the comparisons for the worst known case are listed for each algorithm. PICK performs better than FIND for all values of N, better than heapsort when $N \geq 160$ and better than SELECT when $N \geq 5120$.

Worst Known Case: Number of Comparisons				
N	**Heapsort**	**FIND**	**SELECT**	**PICK (c=5)**
10	23	67	20	60
20	86	237	60	140
40	250	877	140	317
80	661	3357	300	683
160	1645	13117	620	1483
320	3941	51837	1260	2978
640	9205	206077	2793	5893
1280	21025	821757	7664	11469
2560	47278	3281917	18690	22404
5120	104920	13117437	44399	44238
10240	230535	52449277	101401	88454
20480	502218	209756157	250942	175461
40960	1086528	838942717	552109	348281

Table 4 - Number of comparisons for worst known case

A special note about FIND's worst-case running time: it is very recursion-intensive. Since index i or j is only moving once per call to PARTITION, it can be seen that N – 1 recursive calls are necessary. Some environments can handle this depth of recursion while others cannot and thus this can cause a stack overflow. To work around the problem, recursion can be removed by simply adjusting M and N inside of FIND. Pseudo-code for this looks like the following:

```
WHILE (M < N)
BEGIN
    CALL PARTITION
    IF (K <= J) THEN
        N ← J
    ELSE IF (I <= K) THEN
        M ← I
    ELSE
        M ← N
    END
END
```

Since we now have the worst-case running times of each of the algorithms, it is easy to verify those using the following formula:

$$\frac{\log(Comp_2) - \log(Comp_1)}{\log(N_2) - \log(N_1)}$$

If an algorithm is truly O(N), it will have a slope on a log-log graph very near 1. If an algorithm is $O(N^2)$, such as FIND, it will have a slope very near 2. Using 10 and 40960 for N_1 and N_2, we find the slope of each of the algorithms in Table 5 below.

	Heapsort **A: Θ(N log(N))** **W: Θ(N log(N))**	**FIND** **A:O(N)** **W: Θ(N²)**	**SELECT** **A:O(N)** **W:O(N²)**	**PICK** **A:O(N)** **W:O(N)**
Average Case	1.23	1.10	0.94	1.17
Worst Known Case	1.29	1.96	1.23	1.17

Table 5 – Slope of algorithms in average and worst known cases

The fact that the slope for SELECT is less than 1 indicates that it is sub-linear, which would be nice but is obviously impossible, since sub-linear complexity implies that for sufficiently large values of N, the median could be found without examining some elements at all. These results do, however, indicate the very high efficiency of SELECT.

In Table 6 below, we see that the best worst-known-case behavior for PICK is achieved with ascending-order input. The best worst-known-case behavior for SELECT is also achieved with ascending-order input. Heapsort is not affected by the initial state of the input array as random, ascending-order and descending-order input all produce the same-order behavior. FIND achieves its worst-case running time of $\Theta(N^2)$ with descending-order input. Ascending-order input is only slightly better. The maximum number of

comparisons for FIND with ascending input for odd values of N is $\frac{3}{8}N^2 + 1.5N + 1.125$.

For even values of N, the maximum number of comparisons is $\frac{3}{8}N^2 + 1.75N + 2$.

Ascending: Number of Comparisons				
N	**Heapsort**	**FIND**	**SELECT**	**PICK (c=5)**
10	23	57	10	46
20	86	187	30	113
40	250	672	70	280
80	661	2542	150	662
160	1645	9882	310	1422
320	3941	38962	630	2903
640	9205	154722	1325	6121
1280	21025	616642	2692	12885
2560	47278	2462082	5391	26850
5120	104920	9839362	10733	52562
10240	230535	39339522	21325	107903
20480	502218	157322242	42366	221586
40960	1086528	629217282	84276	454509

Table 6 – Number of comparisons using ascending input

In Table 7 below, we see that PICK performs no worse than average when presented with descending-order input. When N ≥ 1280, SELECT's worst-known-case behavior is achieved with descending-order input. For large values of N, PICK is worse than SELECT when using descending-order input.

Descending: Number of Comparisons				
N	**Heapsort**	**FIND**	**SELECT**	**PICK (c=5)**
10	19	67	20	57
20	73	237	60	174
40	216	877	140	248
80	575	3357	300	517
160	1451	13117	620	1155
320	3505	51837	1260	2392
640	8228	206077	2793	4628
1280	18926	821757	7664	8969
2560	42942	3281917	18690	19906
5120	96033	13117437	44399	39664
10240	212531	52449277	101401	77510
20480	465908	209756157	250942	146991
40960	1014140	838942717	552109	322830

Table 7 – Number of comparisons using descending input

In Table 8 below, empirical testing results show the comparison of the three values of c mentioned on page 33. Clearly, when c = 5, fewer comparisons are made than when c = 11 or c = 21. This is an indication as to the complexity of the PICK algorithm and how difficult a worst-case input is to achieve. It also contradicts [1] where c = 21 was claimed to be optimal.

N	c = 5	c = 11	c = 21
10	61	45	45
20	142	191	190
40	322	473	664
80	702	960	1593
160	1507	2197	3327
320	3109	4485	73779
640	6005	8666	14047
1280	11641	15989	26769
2560	22358	29922	48614
5120	44502	57892	90769
10240	88455	114326	173565
20480	175921	224222	337182
40960	349890	440457	660072

Table 8 – Number of comparisons for PICK with different values of c

In Figure 4 below, we plot average-case performance on a log-log graph. SELECT is clearly the best performing algorithm for small and large values of N, in the average case. The point at which it starts to perform sampling (N = 600) is evident.

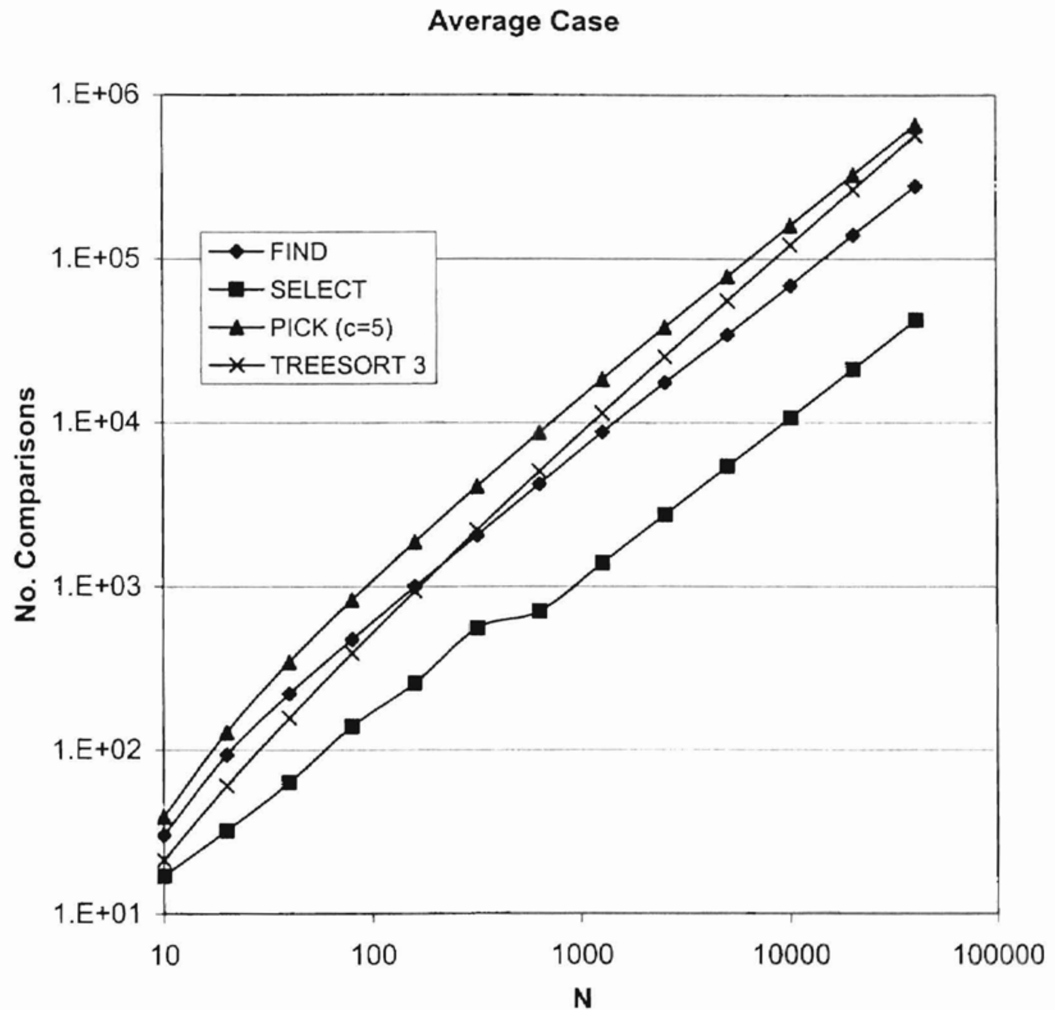


Figure 4 – Average-case behavior for y: No. Comparisons

In Figure 5 below, we plot the worst-case performance on a log-log graph. The slope of FIND is approximately 2, which confirms that its worst-case running time is $\Theta(N^2)$. PICK is less efficient than SELECT and heapsort for small values of N, but for values of $N \geq 5120$, PICK is the better-performing algorithm, in the worst known case.

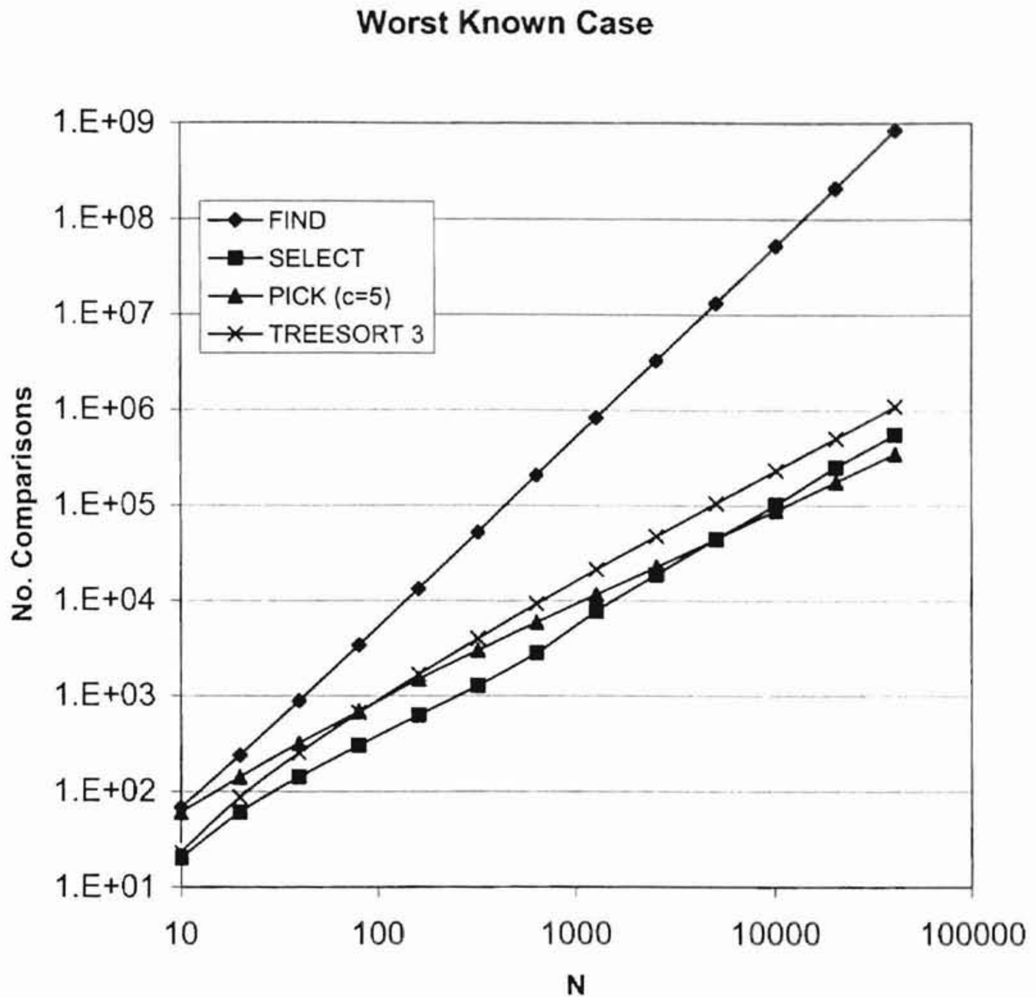


Figure 5 – Worst-known-case behavior for y: No. Comparisons

It is difficult to see the exact dependence of a given curve on the above graphs. We can better see the dependences by dividing the ordinate values by some hypothetical dependence such as $\Theta(N)$, $\Theta(N \log(N))$ or $\Theta(N^2)$. An algorithm that follows the hypothetical dependence will produce a curve that approaches a constant for large values of N. A dependence that is larger (slower) than hypothesized will produce a curve that rises without bound as the value of N becomes large; a dependence that is smaller than hypothesized will produce a curve that approaches zero as the value of N becomes large.

In Figure 6 below, we divide the average-case behavior by a hypothetical dependence of N. PICK approaches a constant for large values of N while FIND and SELECT approach a constant for small values of N. Heapsort is clearly the dominant algorithm for large values of N since its average-case running time of Θ(N log(N)) is still Θ(log(N)) once N is factored out. The other O(N) algorithms are left with small coefficients once N is factored out.

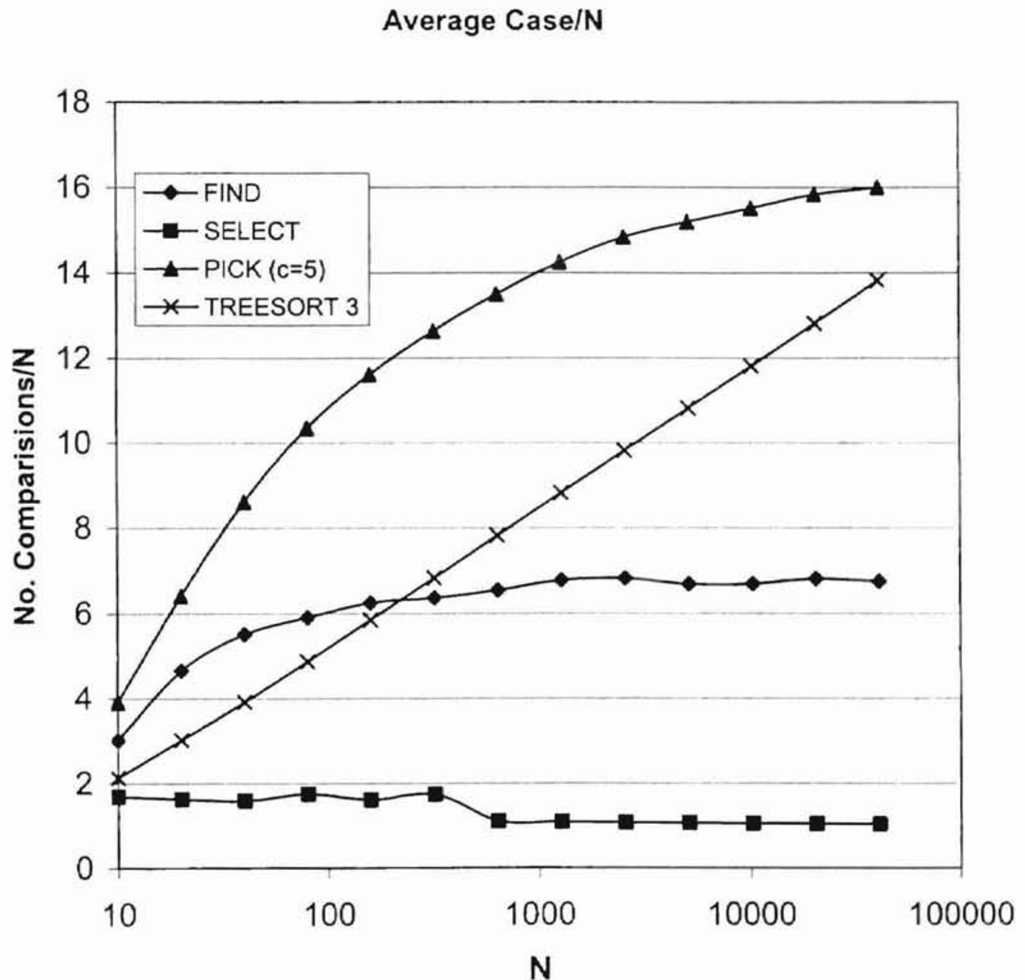


Figure 6 – Average-case behavior for y: No. Comparisons/N

In Figure 7, we divide the worst-known-case behavior by N. We don't see much, other than FIND is much less efficient in the worst known case than the other algorithms. In Figure 8, FIND has been removed so that the other algorithms can be seen in a bit more detail. Heapsort is the dominant algorithm since its worst-case running time is still $\Theta(\log(N))$ after N is factored out. PICK surpasses SELECT once $N \geq 5120$.

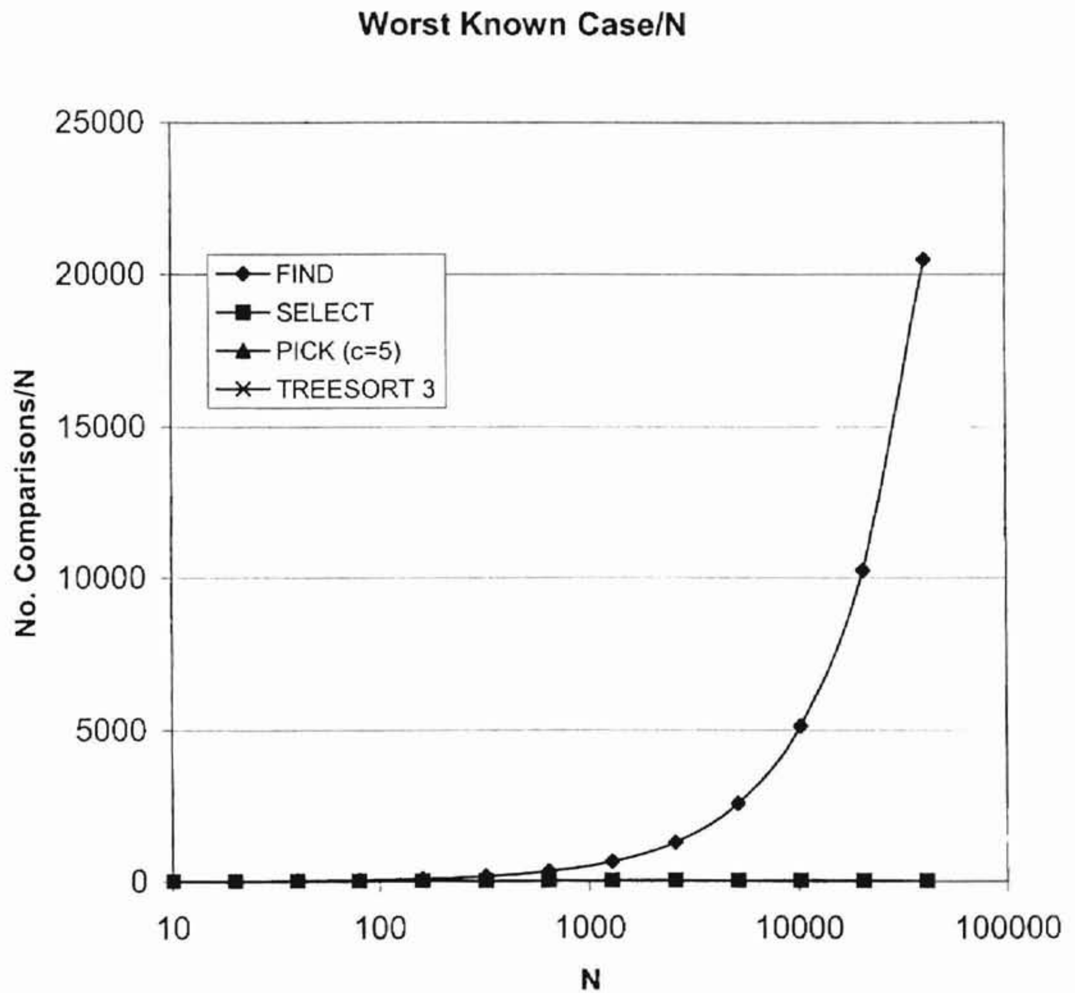


Figure 7 – Worst-known-case behavior for y: No. Comparisons/N

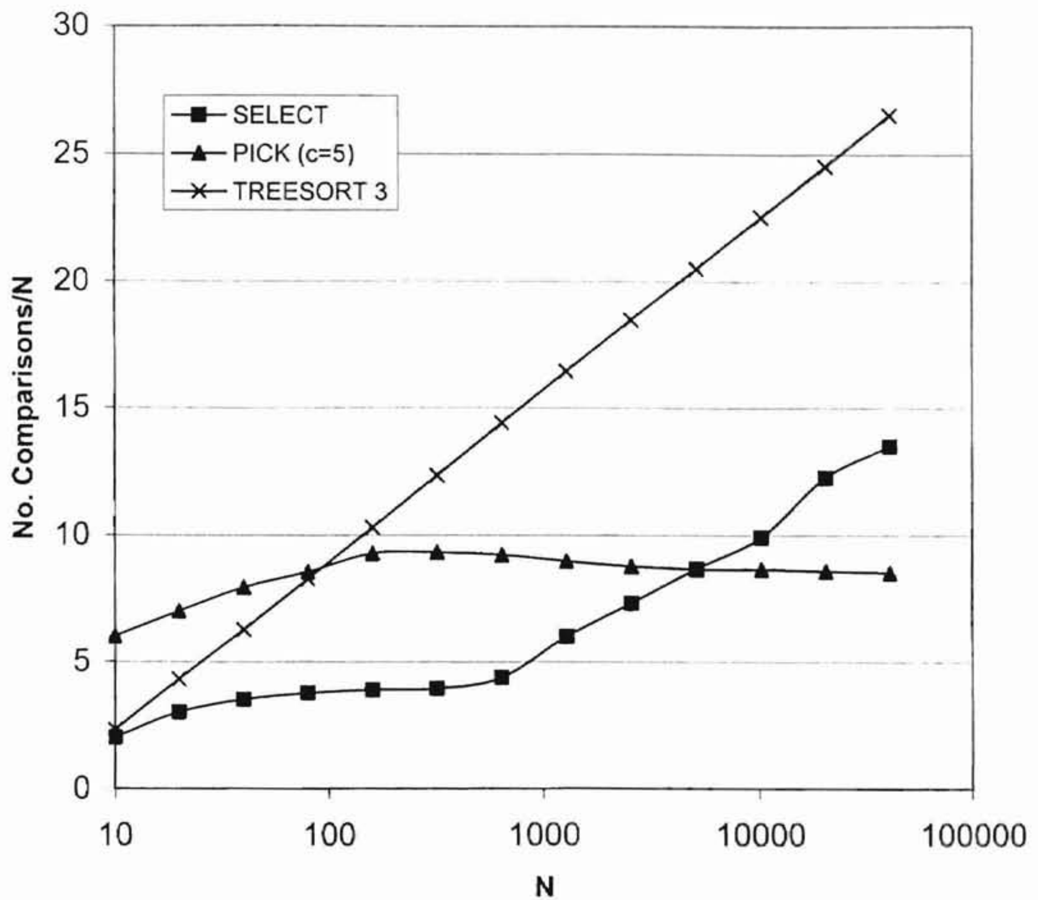


Figure 8 – Worst-known-case behavior for y: No. Comparisons/N (w/o FIND)

In Figure 9, we plot the average-case behavior divided by N log(N). We see that heapsort is slowly approaching a constant of 1.00 while FIND and SELECT fall below that since they are more efficient than $\Theta(N \log(N))$ once $N \geq 320$. Although not visible, PICK, too, surpasses Treesort3 for large values of N. SELECT is clearly the best performing algorithm for all values of N in the average case.

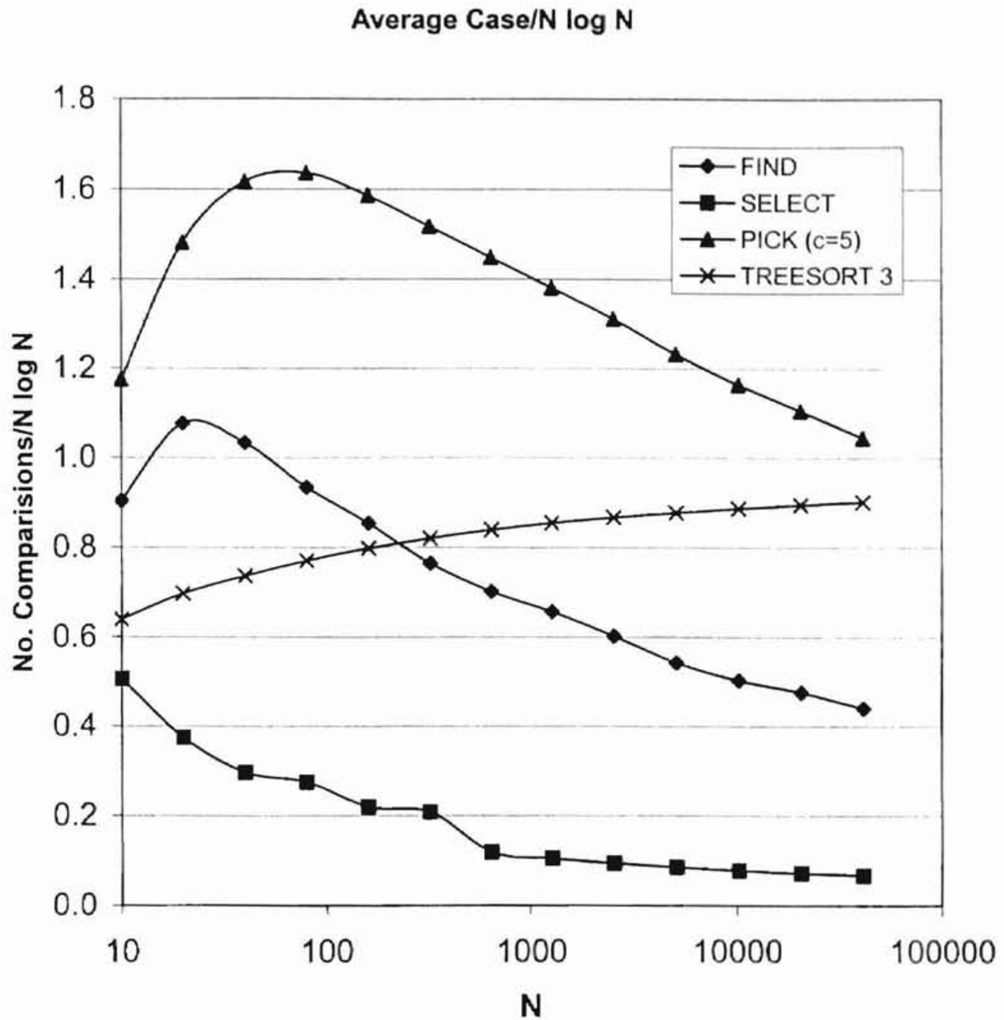


Figure 9 – Average-case behavior for y: No. Comparisons/N log(N)

In Figure 10, we divide the worst-known-case behavior by N log(N). We don't see much other than FIND is much less efficient than the other algorithms in the worst known case. In Figure 11, FIND has been removed so that the other algorithms can be seen in a bit more detail. Heapsort is slowly tapering towards 2.0 while PICK surpasses SELECT when $N \geq 5120$.

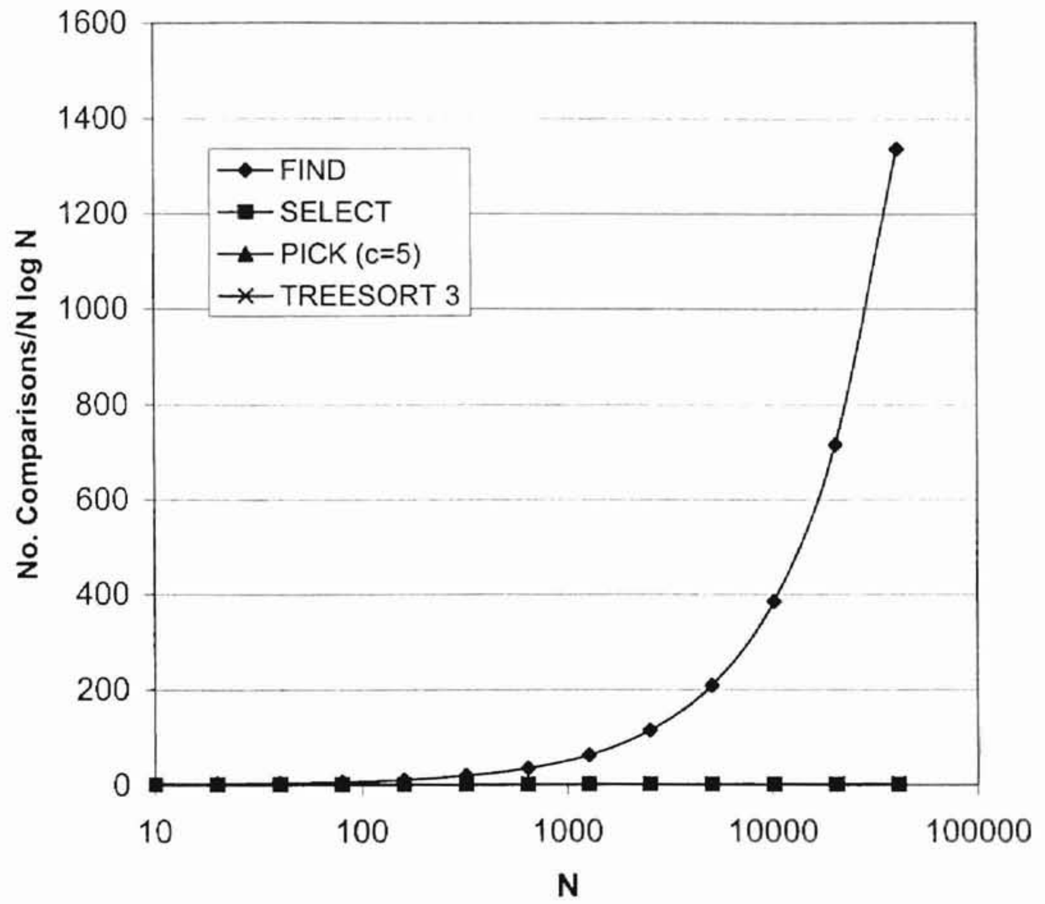


Figure 10 – Worst-known-case behavior for y: No. Comparisons/N log(N)

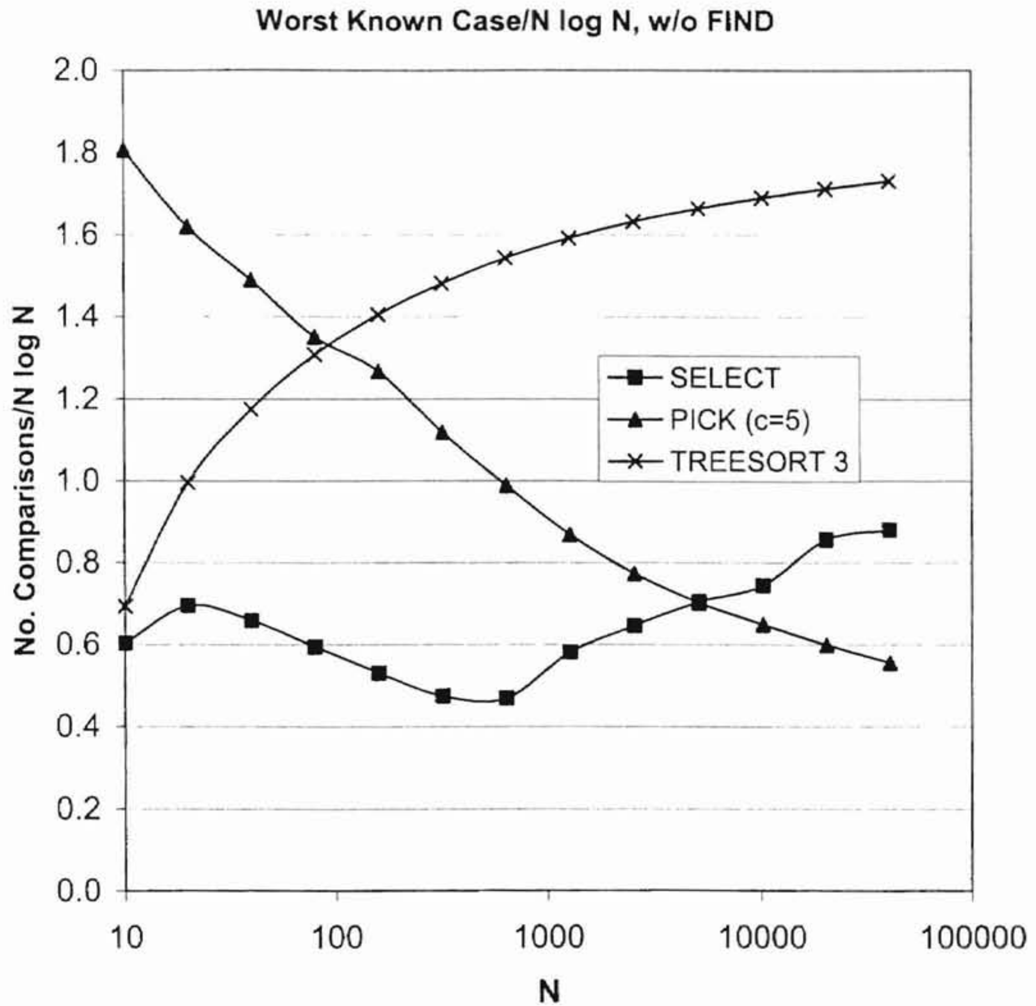


Figure 11 – Worst-known-case behavior for y: No. Comparisons/N log(N) (w/o FIND)

In Figure 12 below, we plot the worst-known-case behavior divided by N^2. We see that FIND, with a worst-case running time of $\Theta(N^2)$, dominates and approaches 0.5, which is the dominant coefficient from $\dfrac{\frac{1}{2}N^2 + 2N - 3}{N^2} = \frac{1}{2} + 2\frac{1}{N} - \frac{3}{N^2}$.

The other algorithms fall far below that since they have worst-case running times of Θ(N log(N)) or $\Theta(N^2)$, leaving a smaller coefficient after N^2 has been factored out of each.

This implies that SELECT is faster than $O(N^2)$ but the worst case could not be obtained empirically.

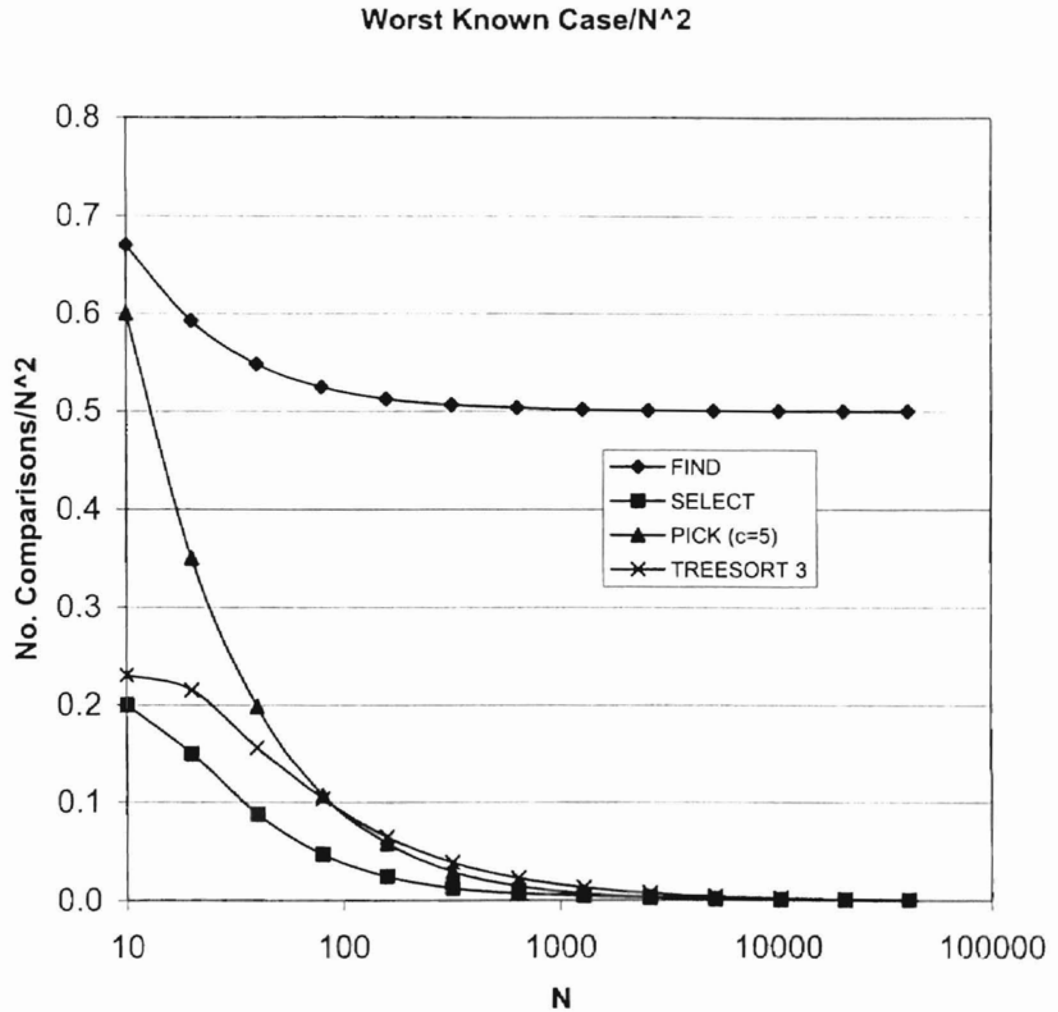


Figure 12 – Worst-known-case behavior for y: No. Comparisons/N^2

In Figure 13 below, we plot the behavior of PICK with random, ascending-order, and descending-order input. There is no significant difference between descending and ascending-order input. PICK performs its best with ascending input.

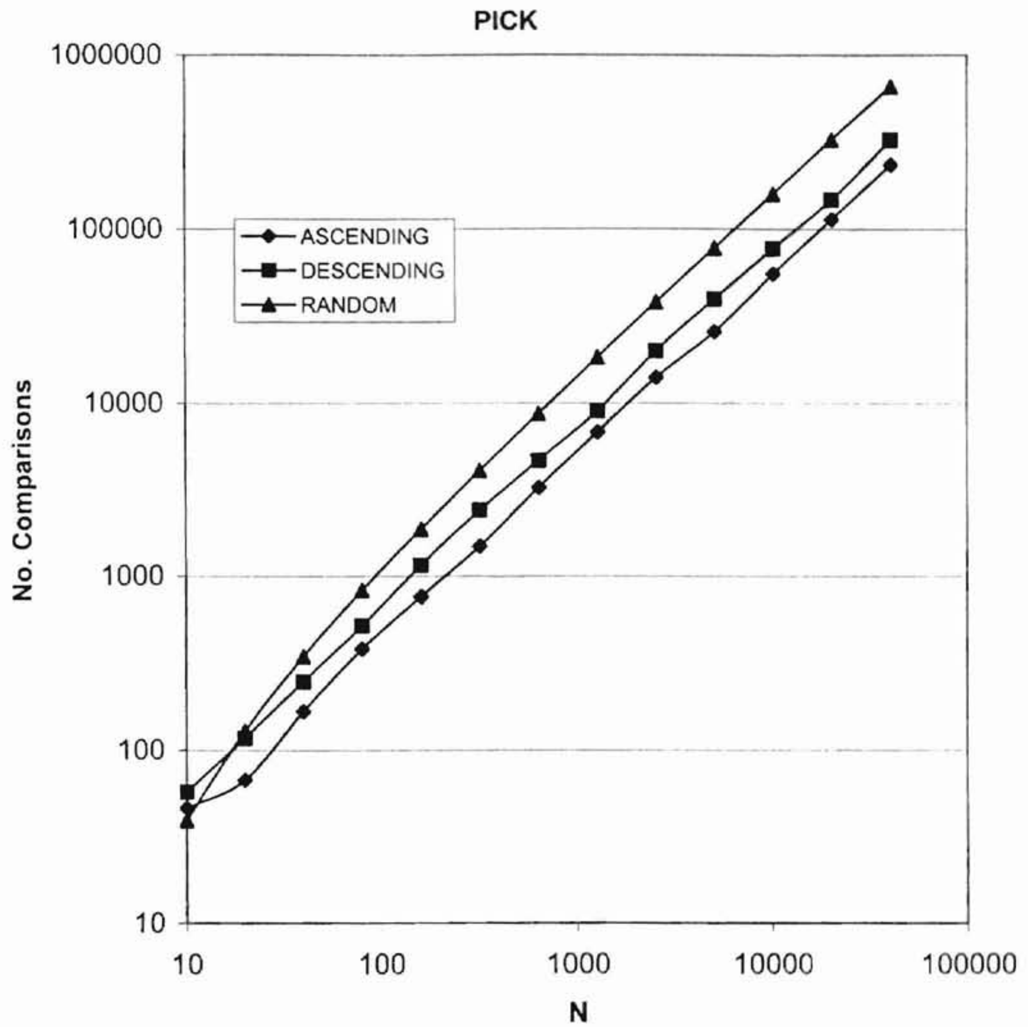


Figure 13 – Three different cases for PICK

In Figure 14 below, we plot the behavior of SELECT with random, ascending-order and descending-order input. Its worst-known-case performance is achieved with descending-order input when $N \geq 1280$. PICK performs its best with ascending input.

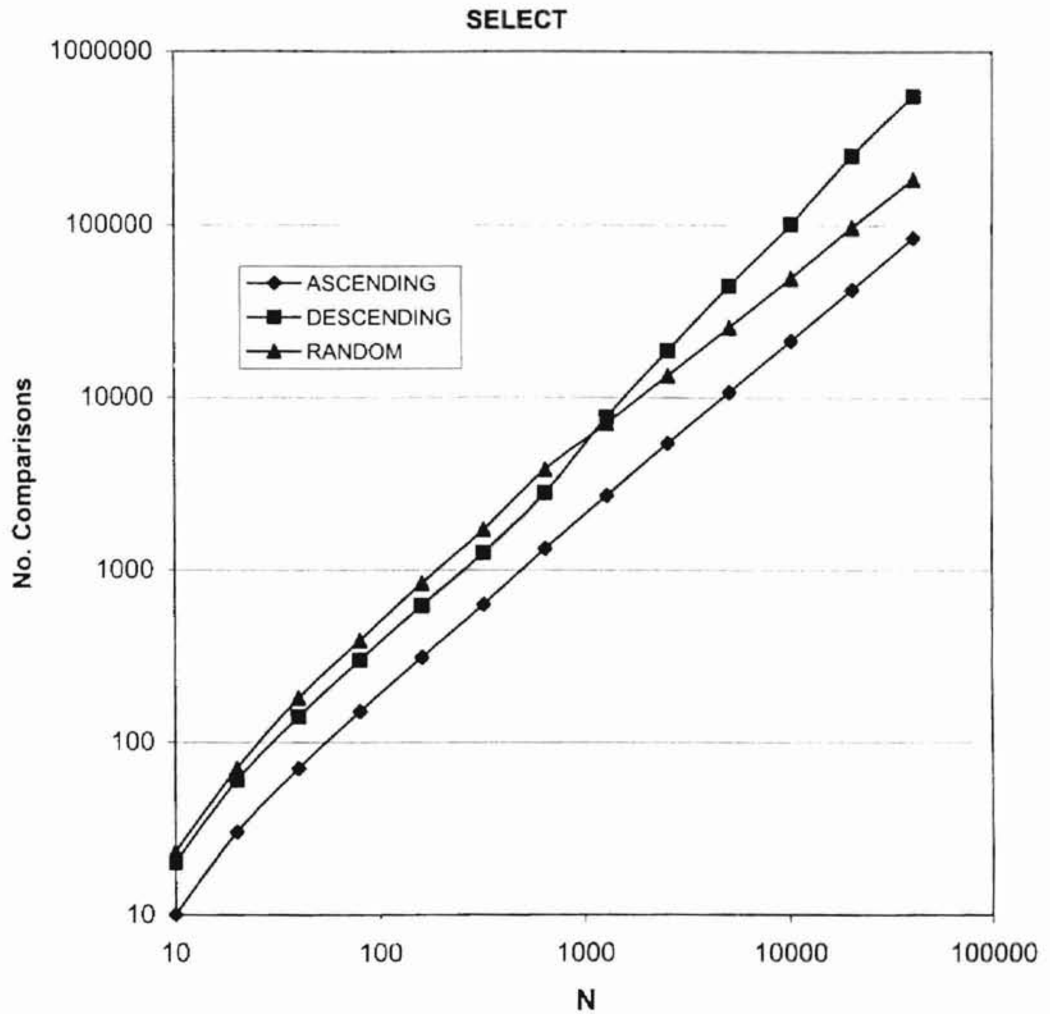


Figure 14 – Three different cases for SELECT

In Figure 15 below, we plot the behavior of heapsort with random, ascending-order and descending-order input. Because its average-case and worst-case running time is $\Theta(N \log(N))$, it is not affected by the initial state of the input, thus the data series for each appears as one.

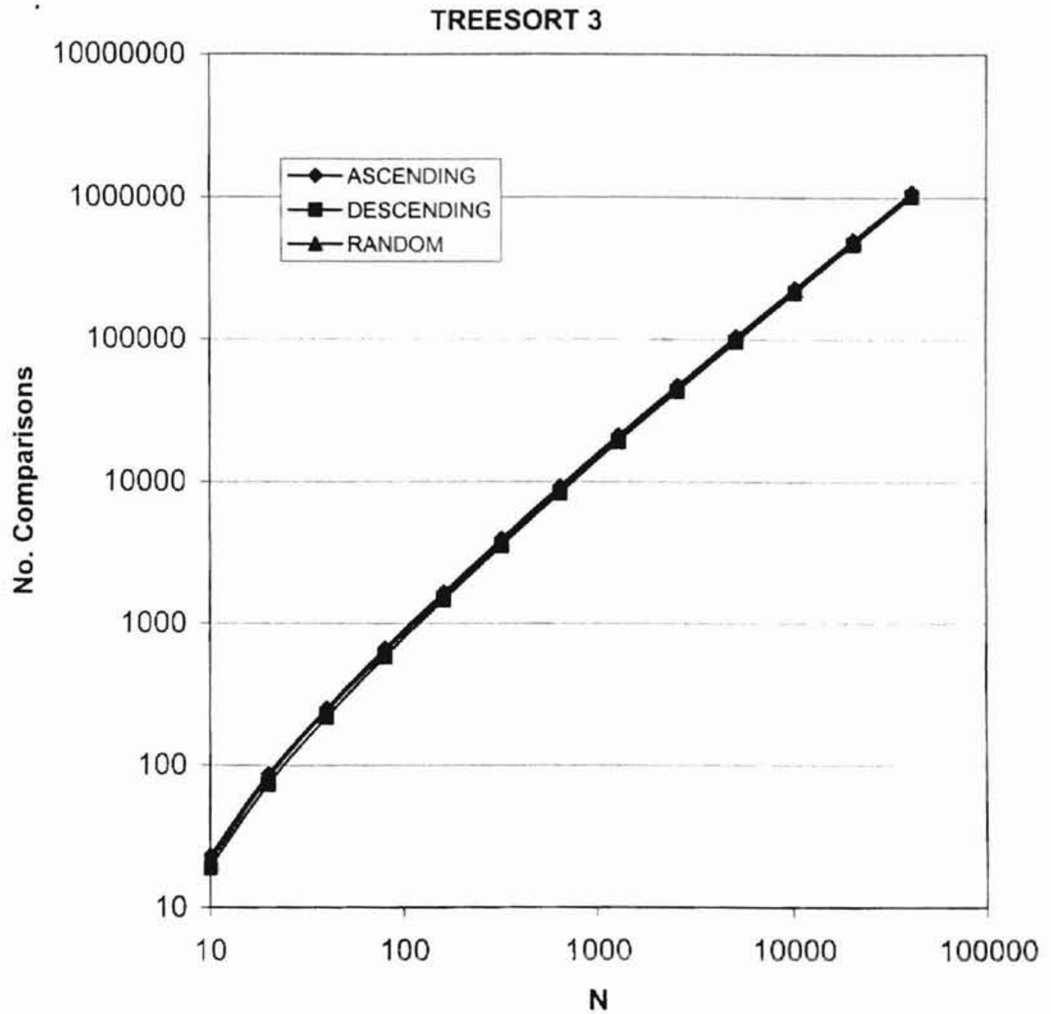


Figure 15 – Three different cases for heapsort

In Figure 16 below, we plot the behavior of FIND with random, ascending-order and descending-order input. Its worst-case performance is achieved with descending-order input with ascending-order input not much better. FIND clearly performs its best with random input.

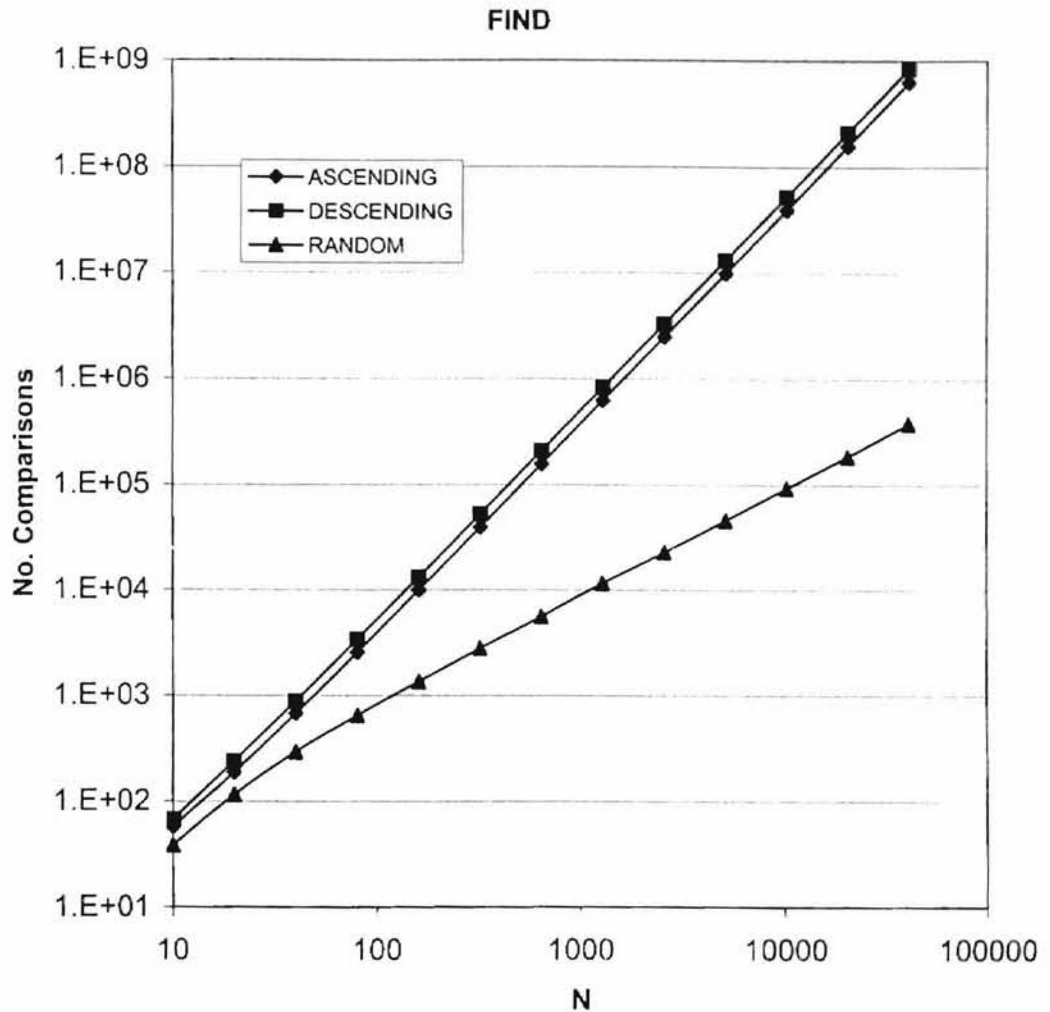


Figure 16 – Three different cases for FIND

As was previously mentioned regarding Dromey's improvement to Hoare's FIND algorithm, the number of comparisons is reduced quite a bit in both the average and worst case. The equation for the algorithm's worst-case behavior appears to be complicated, as empirical testing shows it is not a simple increasing or decreasing sequence. Unlike FIND, however, its running time is $\Theta(N)$ when presented with pre-sorted input. For ascending input, the maximum number of comparisons is $N - 1$. For descending input,

the maximum number of comparisons is 1.5N. As was done with PICK and SELECT, the Monte Carlo method was employed for creating nearly worst-case behavior.

In Table 9 below, the average-case and worst-case comparisons for Dromey's improvement to FIND is listed. For large values of N, the worst known case appears to be better than FIND's average case.

Number Of Comparisons		
N	**Average**	**Worst known**
10	16	31
20	36	94
40	83	290
80	174	721
160	364	2182
320	780	3945
640	1719	6125
1280	3475	10825
2560	6726	18286
5120	14224	42304
10240	27531	87377
20480	55720	174997
40960	112553	251027

Table 9 – Dromey's improvement to FIND

In Figure 17 below, we plot the average-case behavior for FIND and for Dromey's improvement. While an improvement, SELECT is still the best-performing algorithm in the average case. The line in FIND's data series is there to simply *guide the eye* and is susceptible to statistical noise.

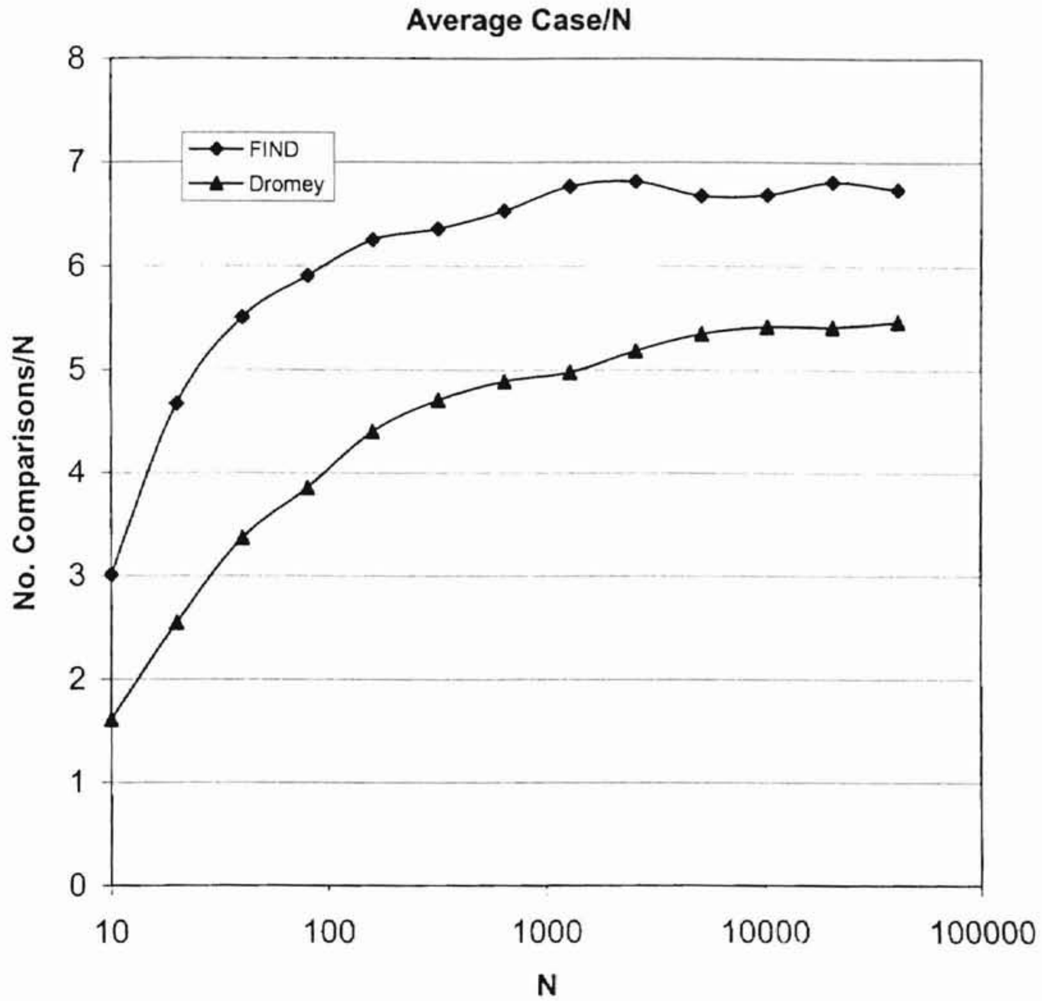


Figure 17 - Average case for FIND and Dromey's improvement/N

In Figure 18 below, we plot the worst-case behavior for FIND and the worst-known-case using Dromey's improvement. The improvement over FIND is quite evident. Although worst-case input for Dromey's improvement could *not* be achieved through empirical testing, it appears to perform no worse than PICK in the worst case.

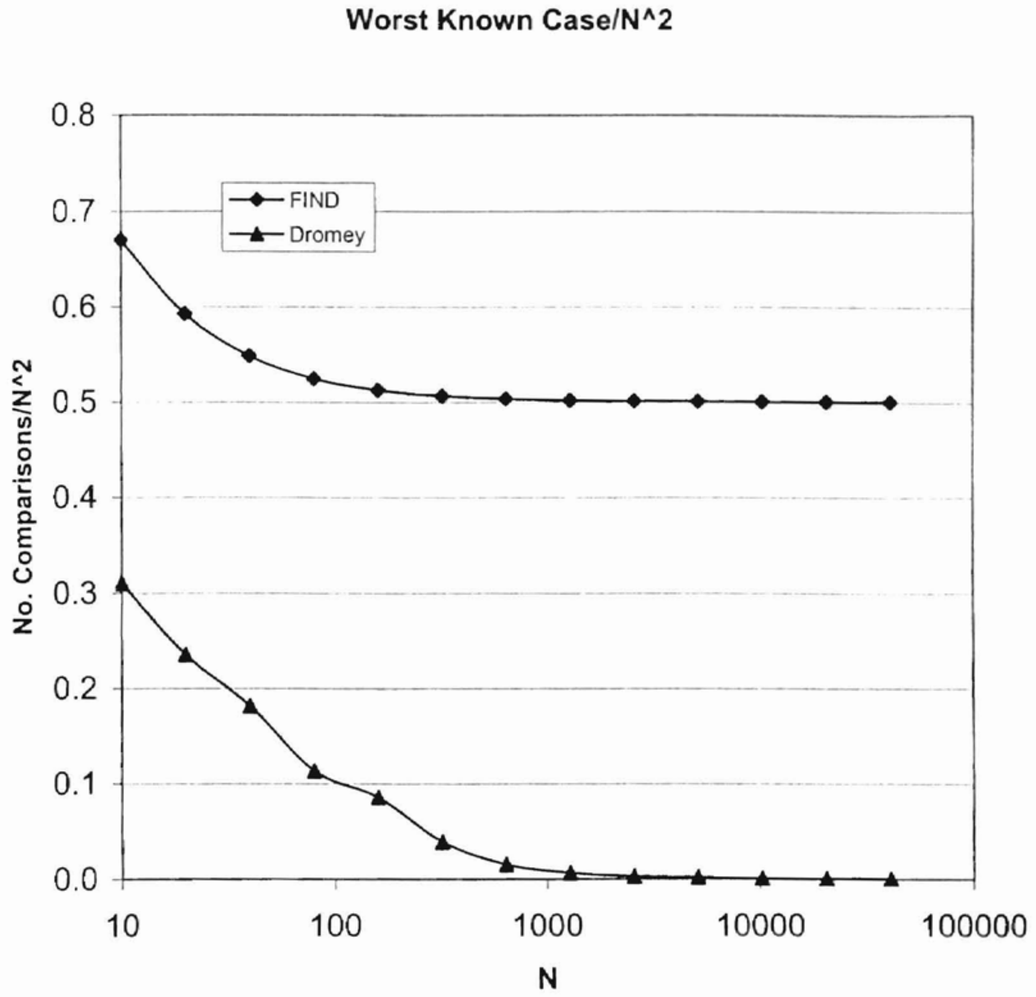


Figure 18 - Worst case for FIND and worst known case using Dromey's improvement/N^2

Chapter 5. Summary, Conclusions and Future Work

In summary, PICK is the only known linear algorithm in the worst case for large values of N. In addition, it is not negatively impacted by pre-sorted input.

SELECT is the best performing algorithm in the average case for all values of N. However, it's sampling scheme is negatively impacted by descending-order input. This can be avoided by first shuffling the input in O(N) time.

Because its average and worst-case running times are both $\Theta(N \log(N))$, heapsort performs consistently with random and pre-sorted input, but it is not competitive with SELECT, FIND or PICK in the average case, or with PICK in the worst case.

FIND performs poorly with pre-sorted input, with its worst-case behavior of $O(N^2)$ obtained with descending-order input. This case can be avoided by first shuffling the input in O(N) time. Because of its recursive nature as implemented by Hoare, input that approaches worst-case behavior can produce stack overflow. An easy workaround was provided for this.

Future work should include finding the worst-case input for PICK and for Dromey's improvement to FIND.

References

[1] ...... Blum, Manuel and Floyd, Robert W. and Pratt, Vaughan and Rivest, Ronald L. and Tarjan, Robert E., Time Bounds for Selection, *Journal of Computer and System Sciences 7*, (August 1973), 448-461.

[2] ...... Brown, Theodore, Remark on Algorithm 489: *ACM Transactions on Mathematical Software 3*, No. 2, (September 1976), 301-304.

[3] ...... Cormen, Thomas and Leiserson, Charles and Rivest, Ronald L., *Introduction to Algorithms*, McGraw-Hill, (1990), 189-191.

[4] ...... Dor, D. and Zwick, U., Selecting the Median, *6th Symposium On Discrete Algorithms*, San Francisco, (1995), 28-37.

[5] ...... Dromey, R. Geoff, An Algorithm for The Selection Problem, *Software – Practice and Experience 16*, No. 11, (November 1986), 981-986.

[6] ...... Eppstein, David, Deterministic Selection, http://www.ics.uci.edu/~eppstein/161/960130.html, (1996 lecture notes).

[7] ...... Floyd, Robert W., Algorithm 113: TREESORT, *Comm. ACM 5*, (August 1962), 434.

[8] ...... Floyd, Robert W., Algorithm 245: TREESORT 3, *Comm. ACM 7*, (December 1964), 701.

[9] ...... Floyd, Robert W. and Rivest, Ronald L., Expected Time Bounds for Selection, *Comm. ACM 18*, 3, (March 1975), 165-172.

[10] .... Ford, Lester R. and Johnson, Selmer M., A Tournament Problem, *The Mathematical Monthly 66*, (May 1959), 387-389.

[11] .... Hadian, Abdollah and Sobel, Milton. *Selecting the tth largest using binary errorless comparisons,* Technical Report No. 121, Department of Statistics, University of Minnesota.

[12] .... Hoare, C.A.R., Algorithm 63: PARTITION, *Comm. ACM 4*, 7, (July 1961), 321.

[13] .... Hoare, C.A.R., Algorithm 64: Quicksort, *Comm. ACM 4*, (1961), 321.

[14] .... Hoare, C.A.R., Algorithm 65: FIND, *Comm. ACM 4*, 7, (July 1961), 321-322.

[15] .... Hoare, C.A.R., Proof of a Program: FIND, *Comm. ACM 14*, 1, (January 1971), 39-45.

[16] .... Knuth, Donald E., *The Art of Computer Programming, vol. 3: Sorting and Searching*, Addison-Wesley, (1973), 136, 381, 618.

[17] .... Weiss, Mark A., *Data Structures and Algorithm Analysis in C, Second Edition*, Addison-Wesley, (1996), 239-240, 260, 373-376.

[18] .... Williams, J.W.J., Algorithm 232: Heapsort, *Comm. ACM* 7, (1964), 347-348.

Appendix A

C source code for Treesort 3

```
// A[i] is moved upward in the subtree of A[0:N-1] of which it is
// the root
void SiftUp( int A[], int i, int N )
{
    int     nCopy,
            j;


    nCopy = A[i];

    do
    {
        j = 2 * i + 1;
        if (j < N)
        {
            if (j < N - 1)
            {
                if (A[j + 1] > A[j])
                    j = j + 1;
            }

            if (A[j] > nCopy)
            {
                A[i] = A[j];
                i = j;
            }
        }
    } while (j < N && A[j] > nCopy);

    A[i] = nCopy;
}

//==============================================================

// TreeSort 3 is a major revision of TreeSort [R. W. Floyd, Alg.
// 113, Comm. ACM 5 (Aug 1962), 434] suggested by HeapSort [J. W.
// J. Williams, Alg. 232, Comm. ACM 7 (June 1964), 347] from
// which it differs in being an in-place sort.  It is shorter and
// probably faster, requiring fewer comparisons and only one
// division.  It sorts the array A[0:N-1], requiring no more than
// 2 * (2^(p-2)) * (p - 1), or approximately 2 * N * log2(N) - 1)
// comparisons and half as many exchanges in the worst case to
// sort N = 2^p - 1 items.  The algorithm is most easily followed
// if A is thought of as a tree, with A[j ÷ 2] the father of A[j]
// for 0 < j < N
void TreeSort3( int A[], int N )
{
    int i;


    for (i = N / 2 - 1; i > 0; i--)
        SiftUp(A, i, N);
```

```
    for (i = N - 1; i > 0; i--)
    {
        SiftUp(A, 0, i + 1);
        Swap(&A[0], &A[i]);
    }
};
```

C source code for FIND

```
// will assign to A[K] the value which it would have if the array
// A[M : N] had been sorted.  the array A will be partly sorted
// and subsequent entries will be faster than the first
void FIND( int A[], int M, int N, int K )
{
    int   I,
          J;


    if (M < N)
    {
        PARTITION(A, M, N, &I, &J);

        if (K <= J)
            FIND(A, M, J, K);
        else if (I <= K)
            FIND(A, I, N, K);
    }

    // use this code if the above causes a stack overflow due to
    // recursion
    /*
    while (M < N)
    {
        PARTITION(A, M, N, &I, &J);

        if (K <= J)
            N = J;
        else if (I <= K)
            M = I;
        else
            M = N;
    }
    */
}

//=================================================================

// we want a number between M and N
#define Random1(M,N) ((Random() % (N - M + 1)) + M)

// I and J are output variables and A is the array (with
// subscript bounds M:N) which is operated upon by this
// procedure.  partition() takes the value X of a random element
// of the array A and rearranges the values of the elements of
// the array in such a way that there exist integers I and J with
// the following properties:
```

```
//      M <= J < I <= N provided M < N
//      A[R] <= X for M <= R <= J
//      A[R]  = X for J <  R <  I
//      A[R] >= X for I <= R <= N
// the procedure uses an integer procedure random(M,N) which
// chooses equiprobably a random integer F between M and N and
// also a procedure Swap(), which exchanges the values of its two
// parameters
void PARTITION( int A[], int M, int N, int *I, int *J )
{
    int     X,
            F;


    F = Random1(M, N);

    X  = A[F];
    *I = M;
    *J = N;

up:
    for (*I = *I; *I <= N; *I = *I + 1)
    {
        if (X < A[*I])
            goto down;
    }

    *I = N;
down:
    for (*J = *J; *J >= M; *J = *J - 1)
    {
        if (A[*J] < X)
            goto change;
    }

    *J = M;
change:
    if (*I < *J)
    {
        Swap(&A[*I], &A[*J]);

        *I = *I + 1;
        *J = *J - 1;

        goto up;
    }
    else if (*I < F)
    {
        Swap(&A[*I], &A[F]);
        *I = *I + 1;
    }
    else if (F < *J)
    {
        Swap(&A[F], &A[*J]);
        *J = *J - 1;
    }
}
```

C source Code for Dromey's Improvement to FIND

```
void Dromey( int A[], int l, int r, int k )
{
    int     i,
            j,
            x;

    while (l < r)
    {
        i = l;
        j = r;
        x = A[k];

        // if either of the indexes cross 'k', the choice for 'k'
        // was wrong
        while (i <= k && j >= k)
        {
            do
            {
                if (A[i] < x)
                    i = i + 1;
            } while (A[i] < x);

            do
            {
                if (A[j] > x)
                    j = j - 1;
            } while (A[j] > x);

            Swap(&A[i], &A[j]);
            i = i + 1;
            j = j - 1;
        }

        if (j < k)
            l = i;

        if (i > k)
            r = j;
    }
}
```

C source code for SELECT

```
#define sign(x)   ((x >= 0.0) ? 1.0 : -1.0)
#define SAMPLING (600)

// rearrange the values of array segment X[L : R] so that X[K]
// (for some given K; L <= K <= R) will contain the
// (K - L + 1)-th smallest value,
// L <= 1 <= K will imply X[I] <= X[K] and K <= I <= R will imply
// X[I] >= X[K]
void SELECT( int A[], int L, int R, int K )
{
```

```
int     N,
        I,
        J,
        S,
        SD,
        LL,
        RR,
        T;
double  Z;


while (R > L)
{
    if (R - L + 1 > SAMPLING)
    {
        // call SELECT() recursively on a sample of size S to
        // get an estimate for the (K - L + 1)-th smallest
        // element into X[K], biased slightly so that the
        // (K - L + 1)-th element is expected to lie in the
        // smaller set after partitioning
        N  = R - L + 1;
        I  = K - L + 1;
        Z  = log((double) N);
        S  = (int) (0.5 * exp(2.0 * Z / 3.0));
        SD = (int) (0.5 * sqrt(Z * S * (N - S) / N) *
             sign(I - N / 2.0));
        LL = __max(L, K - I * S / N + SD);
        RR = __min(R, K + (N - 1) * S / N + SD);

        SELECT(A, LL, RR, K);
    }

    T = A[K];

    // the following code partitions X[L : R] about T.
    // it is similar to PARTITION but will run faster on most
    // machines since subscript range checking I and J has
    // been eliminated
    I = L;
    J = R;

    // if A[K] >= A[L] and A[R], then the swaps below will
    // assign to A[L] the value we are partitioning around.
    // otherwise, if A[K] <= A[L] and A[R], then the swaps
    // will assign to A[R] the value we are partitioning
    // around.

    // put the element we are partitioning around at the
    // left end
    Swap(&A[L], &A[K]);

    // ensure that the element at the left end is larger than
    // the element at the right end because of the initial
    // exchange in the loop below
    if (T < A[R])
    {
        Swap(&A[R], &A[L]);
```

```
        }

        while (I < J)
        {
            Swap(&A[I], &A[J]);
            I = I + 1;
            J = J - 1;

            do
            {
                if (A[I] < T)
                    I = I + 1;
            } while (A[I] < T);

            //while (A[I] < T)
                //I = I + 1;

            do
            {
                if (A[J] > T)
                    J = J - 1;
            } while (A[J] > T);

            //while (A[J] > T)
                //J = J - 1;
        }

        if (A[L] == T)
            // prior to the above 'while' loop, the element we
            // are partitioning around was initially at A[R].
            // it then got exchanged with A[L] and stayed there
            Swap(&A[L], &A[J]);
        else
        {
            // prior to the above 'while' loop, the element we
            // are partitioning around was initially at A[L].
            // it then got exchanged with A[R] and stayed there
            J = J + 1;
            Swap(&A[J], &A[R]);
        }

        // now adjust L, R so they surround the subset containing
        // the (K - L + 1)-th smallest element
        if (J <= K)
            // the sought item is in the right side so adjust the
            // left boundary
            L = J + 1;

        if (K <= J)
            // the sought item is in the left side so adjust the
            // right boundary
            R = J - 1;
    }
}
```

C source code for PICK

```
#define GROUPSIZE   5

int PICK( int A[], int N, int k )
{
    int     nGroups = 0,
            L1,
            nIndex  = 0,
            nMedian = 0,
            nElementsInGroup,
            nOffset,
            *pMedians;


    if (N <= GROUPSIZE)
    {
        // sort this group and return the k-th element
        Sort(A, N);

        return (A[k]);
    }

    // calculate how many groups we're dealing with
    if ((N % GROUPSIZE) == 0)
        nGroups = N / GROUPSIZE;
    else
        nGroups = (N / GROUPSIZE) + 1;

    // allocate memory to hold the medians of the groups
    pMedians = new int[nGroups];

    for (nIndex = 0; nIndex < nGroups; nIndex++)
    {
        nElementsInGroup = __min(GROUPSIZE, N - (nIndex *
                                 GROUPSIZE));

        Sort(&A[nIndex * GROUPSIZE], nElementsInGroup);

        // will be 0, 1 or 2 when GROUPSIZE = 5
        nOffset = nElementsInGroup / 2;

        pMedians[nIndex] = A[(nIndex * GROUPSIZE) + nOffset];
    }

    // recursively find the median of the medians to use as the
    // pivot later
    nMedian = PICK(pMedians, nGroups, nGroups / 2);

    // we're done with the memory so free it up
    delete [] pMedians;

    // partition on the median found earlier
    L1 = MedianPartition(A, N, nMedian);

    if (k < L1)
    {
```

```
        // recurse on the left side
        return PICK(A, L1, k);
    }
    else if (k > L1)
    {
        // recurse on the right side
        return PICK(&A[L1 + 1], N - L1 - 1, k - L1 - 1);
    }
    else
        // we found it!
        return nMedian;
}

//==================================================================

int MedianPartition( int A[], int N, int nPivot )
{
    int i,
        j,
        F;


    i = -1;
    j = N;
    F = -1;

    while (i < j)
    {
        do
        {
            j = j - 1;
        } while (A[j] > nPivot);

        do
        {
            i = i + 1;
        } while (A[i] < nPivot);

        if (i < j)
        {
            // if the pivot element is moving, note where it is
            // moving to
            if (A[i] == nPivot)
                F = j;
            else if (A[j] == nPivot)
                F = i;

            Swap(&A[i], &A[j]);
        }
    }

    // if the pivot element moved, swap it so that all elements
    // to its left are smaller.  otherwise, it's just
    // intermingled with all of the smaller elements or larger
    // elements which is a bad thing
    if (-1 != F)
    {
```

```
        if (F > i)
        {
            // the pivot element needs to be moved back to the
            // left
            Swap(&A[i], &A[F]);
            j = i;
        }
        else if (F < j)
            // the pivot element needs to be moved back to the
            // right
            Swap(&A[F], &A[j]);
        else
            // the pivot element is right where it needs to be
            // but 'j' is one-off
            j = i;
    }

    // elements up to and including A[j] are less than or equal
    // to nPivot
    return j;
}
```

C source code for driver

```
void main( void )
{
    __int64     i64Total = 0;
    int         x,
                N;
                *A;

    for (N = 10; N <= 40960; N *= 2)
    {
        A = new int[N];
        if (NULL != A)
        {
            for (x = 0; x < N; x++)
                A[x] = Random();

            PICK(A, N, N / 2);

            delete [] A;
        }
    ;
;
```

Appendix B

These formulas, or abridged versions of them, are found in the various references but have been included here showing how one form has been simplified into another.

Formula for FIND's average case

$$T(N) \le \frac{1}{2}T(N) + \frac{1}{2}T\left(\frac{3N}{4}\right) + N$$

$$subtract\,\frac{1}{2}T(N)\,from\,both\,sides$$

$$T(N) - \frac{1}{2}T(N) \le \frac{1}{2}T(N) - \frac{1}{2}T(N) + \frac{1}{2}T\left(\frac{3N}{4}\right) + N$$

$$\frac{1}{2}T(N) \le \frac{1}{2}T\left(\frac{3N}{4}\right) + N$$

$$divide\,both\,sides\,by\,\frac{1}{2}$$

$$\frac{\frac{1}{2}T(N)}{\frac{1}{2}} \le \frac{\frac{1}{2}T\left(\frac{3N}{4}\right) + N}{\frac{1}{2}}$$

$$T(N) \le T\left(\frac{3N}{4}\right) + N$$

Formula for FIND's worst case

FIND's worst-case behavior takes the form of the linear equation:

$aN^2 + bN + c$ = maximum number of comparisons.

The values for a, b and c differ slightly for odd values of N versus even values of N. For $N = 4$, 6 and 8, we have the three equations:

E1: $16a + 4b + c = 13$

E2: $36a + 6b + c = 27$

E3: $64a + 8b + c = 45$

Subtract E2 from E3:

E1: $16a + 4b + c = 13$

E2: $36a + 6b + c = 27$

E3: $28a + 2b = 18$

Subtract E1 from E2:

E1: 16a + 4b + c = 13

E2: 20a + 2b = 14

E3: 28a + 2b = 18

Subtract E2 from E3:

E1: 16a + 4b + c = 13

E2: 20a + 2b = 14

E3: 8a = 4

a = 0.5

Plug this value into E2:

E2: 20(0.5) + 2b = 14

b = 2

Plug these values into E1:

E1: 16(0.5) + 4(2) + c = 13

c = -3

So, with a = 0.5, b = 2 and c = -3, we have $\frac{1}{2}N^2 + 2N - 3$.

For N = 3, 5 and 7, we have the three equations:

E1: 9a + 3b + c = 8

E2: 25a + 5b + c = 20

E3: 49a + 7b + c = 36

Subtract E2 from E3:

E1: 9a + 3b + c = 8

E2: 25a + 5b + c = 20

E3: 24a + 2b = 16

Subtract E1 from E2:

E1: 9a + 3b + c = 8

E2: 16a + 2b = 12

E3: 24a + 2b = 16

Subtract E2 from E3:

E1: 9a + 3b + c = 8

E2: 16a + 2b = 12

E3: 8a = 4

a = 0.5

Plug this value into E2:

E2: 16(0.5) + 2b = 12

b = 2

Plug these values into E1:

E1: 9(0.5) + 3(2) + c = 8

c = -2.5

So, with a = 0. 5, b = 2 and c = -2.5, we have $\frac{1}{2}N^2 + 2N - 2.5$.

Formula for the minimum elements discarded in PICK

This formula comes from page 191 of [3]. It simply shows how the ceiling "$\lceil\ \rceil$" function affects the ≥ and = operators

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{N}{5} \right\rceil \right\rceil - 2\right) \geq$$

$$multiply\, what's\, in\, parenthesis\, by\, 3$$

$$\left(\frac{3}{2}\left\lceil \frac{3N}{15} \right\rceil\right) - 6 \geq$$

$$multiple\, what's\, in \lceil\ \rceil by\, \frac{3}{2}$$

$$\frac{9N}{30} - 6 =$$

$$\frac{3N}{10} - 6$$

Formulas for the cost of the PICK algorithm

These formulas come from pages 450-451 of [1]. The first one shows how the recursive terms of the equation can be removed by following a few simple algebra rules.

$$P(n) \le \frac{n \times h(c)}{c} + P\left(\frac{n}{c}\right) + n + P\left(n - d \times \frac{n}{2c}\right)$$

$$P(n) \le \frac{n \times h(c)}{c} + \left(\frac{1}{c}\right)P(n) + n + \left(1 - \frac{d}{2c}\right)P(n)$$

$$P(n) \le \frac{n \times h(c)}{c} + n + \left(\frac{1}{c} + 1 - \frac{d}{2c}\right)P(n)$$

$$P(n) - \left(\frac{1}{c} + 1 - \frac{d}{2c}\right)P(n) \le \frac{n \times h(c)}{c} + n$$

$$P(n) + \left(-\frac{1}{c} - 1 + \frac{d}{2c}\right)P(n) \le \frac{n \times h(c)}{c} + n$$

$$\left(1 - \frac{1}{c} - 1 + \frac{d}{2c}\right)P(n) \le \frac{n \times h(c)}{c} + n$$

$$\left(1 - 1 - \frac{1}{c} + \frac{d}{2c}\right)P(n) \le \frac{n \times h(c)}{c} + \frac{c}{c}n$$

$$\left(-\frac{1}{c} + \frac{d}{2c}\right)P(n) \le \frac{n \times h(c)}{c} + \frac{c}{c}n$$

$$\left(-1 + \frac{d}{2}\right)P(n) \le n \times h(c) + cn$$

$$\left(-\frac{2}{2} + \frac{d}{2}\right)P(n) \le n\left(h(c) + c\right)$$

$$\left(\frac{d-2}{2}\right)P(n) \le n\left(h(c) + c\right)$$

$$P(n) \le \frac{n\left(h(c) + c\right)}{d - 2/2}$$

$$P(n) \le \frac{n \times 2 \times (h(c) + c)}{d - 2}$$

$$P(n) \le \left(\frac{2 \times (h(c) + c)}{d - 2}\right) \times n$$

Per the authors, we can substitute c=21, h(21)=66 and d=11 into the above formula to find the minimal cost of PICK. The correct answer of 19.3n is obtained rather than 19.6n, which is found in [1].

$$P(n) \le \frac{66n}{21} + P\left(\frac{n}{21}\right) + n + P\left(\frac{31n}{42}\right)$$

$$P(n) \le \left(\frac{66}{21}\right)n + \left(\frac{n}{21}\right)P(n) + n + P\left(\left(\frac{31}{42}\right)n\right)$$

$$P(n) \le \left(\frac{66}{21}\right)n + \left(\frac{1}{21}\right)P(n) + n + \left(\frac{31}{42}\right)P(n)$$

$$P(n) \le \left(\frac{87}{21}\right)n + \left(\frac{33}{42}\right)P(n)$$

$$\left(\frac{42}{42}\right)P(n) - \left(\frac{33}{42}\right)P(n) \le \left(\frac{87}{21}\right)n$$

$$\left(\frac{9}{42}\right)P(n) \le \left(\frac{87}{21}\right)n$$

$$P(n) \le \left(\frac{42}{9}\right)\left(\frac{87}{21}\right)n$$

$$P(n) \le \frac{174n}{9}$$

$$P(n) \le 19.\bar{3}n$$

2

VITA

David B. Crow

Candidate for the Degree of

Master of Science

Thesis: FINDING THE MEDIAN IN LINEAR WORST-CASE TIME

Major Field: Computer Science

Biographical:

Personal Data: Born in McAlester, Oklahoma on September 6, 1968.

Education: Graduated from McAlester High School, McAlester, Oklahoma in May 1986; received Associate of Science degree in Computer Science from Eastern Oklahoma State College, Wilburton, Oklahoma in May 1988; received Bachelor of Science degree in Computer Science from East Central University, Ada, Oklahoma in May 1990. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in December of 2001.

Experience: Have been employed in the computer and technology industry since 1988 as a programmer and software engineer.