Stack-Based Sorting Algorithms

Reda A. Ammar
Computer Science and Engineering Department, University of Connecticut, Storrs, Connecticut

The emergence of stacks as a hardware device in stack machines implies the recognition of the importance of using stacks in different computer applications and the need to make use of them in others. This paper uses stacks to solve the sorting problem. Two stack-based sorting algorithms are introduced. The first is based upon sorting by the insertion technique, whereas the second is based upon sorting by the exchange technique. Their analysis and performance are derived when stack computers are used to run them. A comparison study with other sorting algorithms is presented. This study shows that both algorithms have the best performance with a wide margin relative to other sorting algorithms when stack computers are used.

1. INTRODUCTION

The stacks or "last in first out" data structure has been widely used in several different ways in a computer [7]. In compiling programs, it can be used in parsing. In executing assignment statements, it can be used with zero-address instructions to manage temporary results. In carrying out dynamic storage allocation, it can be used to activate and deactivate arrays within one user's program. Furthermore, the operating system can use stacks to activate and deactivate different users.

Stacks can be implemented in any computer by software means, but in some machines a good deal of special hardware is provided to carry out stack operations. Besides the B5000-B6700 [8, 12] series that can be considered as the paradigm of the stack computers [11], hardware stacks are present in the ICL 2900, a British entry in the computer market, in a number of minicomputers such as the Hewlett Packard HP3000 [2], and in a number of microprocessors such as the Intel 8080, Intel iAPX 432 [9], and National PACE [13].

When a computer architecture is altered to include or facilitate some software device, it is a sign that the device has become generally accepted, far past the experimental stage, as an essential feature of the computer system. From this sense, the emergence of stacks as a hardware device in stack machines implies the recognition of the the important of using stacks in different computer applications and the need to make use of them in others. In this paper, we expand the computer applications that use stacks to include solving the sorting problem.

Internal sorting algorithms have been extensively studied [1, 10, 14, 18]. Through this study, very interesting techniques have been discovered such as sorting by insertion, by exchanging, by selection, and by merging. Each of these techniques use different approaches. As an example, sorting by insertion may use straight insertion, binary insertion, two-way insertion, and so on. In these approaches, the array and pointer data structure are mainly used. Other data structures, such as stacks and queues, are used as auxiliaries to keep track of the pointer sequences. In this paper, two algorithms using a stack data structure are introduced. The first is based upon sorting by the insertion technique, whereas the second is based upon sorting by the exchanging technique. The second algorithm is a modification of the algorithm by Yuen [19].

This paper is organized as follows. In the next section, we briefly present the methodology used to study and compare the performance of the proposed algorithms with other algorithms. In Section 3, the stack insertion sort algorithm is described, and its performance is derived. Section 4 does the same as Section 3 but for the stack partition sort algorithm. Finally, Section 5 will show the importance of using the stack to sort a set of elements on a stack machine instead of the traditional sorting algorithms.

2. THE METHODOLOGY USED TO STUDY AN ALGORITHM'S PERFORMANCE

Our methodology consists of constructing the time formulas for a given stack algorithm and the ones that use the same technique but are implemented by arrays if they are run on the same machine. Time formulas are

Address correspondence to Reda A. Ammar, Assistant Professor, U155, Computer Science and Engineering Department, University of Connecticut, Storrs, CT 06268.

© 1989 Elsevier Science Publishing Co., Inc.
0164-1212/89/$3.50

symbolic formulas that express the execution times as functions of a set of performance parameters [3].

Our time formulas are generated by a performance-analysis software tool called "PASS" [4, 15]. PASS is an analytic tool. It uses an analytic approach to predict the performance of a given computation. The input to PASS is a computation that is described by a program written in a language supported by PASS. PASS first constructs the computation structure model [16] of the given computation. It then uses the flow-analysis technique [17] to derive the time formula of the computation. Using this technique, the time formula will be function in flow counts. Some of the flows are dependent flows that can be expressed in terms of a set of independent flows. In addition, there may exist some relations among the independent flows defined by the information being processed, thus the independent flows can often be expressed as functions of some externally observable parameters [3].

It is the user's responsibility to specify the performance parameters and use these parameters to define the flow counts for the independent flows in the computation. In most cases in this paper, we use an analytical approach to derive the relationship between a flow count and the performance parameters. In some cases, it is difficult to use the analytical approach to derive the required relationship. In these cases, an experimental approach can be used.

Not only is the time formula a function of the performance parameters, but it is also machine dependent. PASS assumes that all operations carried in a high-level language, such as Pascal, can be implemented using the following eight primitive operations: (1) dereference, (2) addition, (3) multiplication, (4) relation, (5) logic NOT, (6) logic AND, (7) logic OR, (8) and assignment. The cost of a high-level operation is then represented by a vector that indicates how many times a primitive operation is used to accomplish the operation. The time cost of each of these primitive operations is usually given in the manufacturer's specification tables [5, 6]. These tables, for some interesting machines, are stored in a knowledge base accessed by the PASS system. Given this vector and specifying the target machine, PASS evaluates the cost of each high-level operation. On the other hand, if an assembly language is used, the time formula is a function of the cost of differently used instructions. The cost of an instruction depends upon the instruction type and the different operands modes. The cost of executing an instruction is also given in the manufacturer's specification tables [5, 6].

Specifying the machine and the relationships between the independent flows and the performance parameters, PASS finally evaluates the time formula to answer the user's questions about performance (e.g., the mean cost of the computation, the profile of the computation, etc.).

To have adequate comparison results for different algorithms, they should be designed and run in their most matching environment. Since the stack sorting algorithms are designed to run on stack computers, the target machine should be a stack computer or at least a machine that can simulate the activities of a stack computer. PDP11 is a widely used machine that supports the main two stack operations PUSH and POP. At the same time, PDP11 is a suitable machine to run array sorting algorithms. Consequently, PDP11 was used as the target machine throughout this paper.

3. STACK-INSERTION SORT ALGORITHM

3.1 Description of the Algorithm

The basic idea in sorting by insertion is to insert an element a_{i+1} into a sequence of ordered elements $a_1, a_2, \cdots, a_i$ in such a way that the resulting sequence of size $(i + 1)$ is also ordered. The process of finding the appropriate place for the element a_{i+1} is accomplished via searching the ordered list $a_1, a_2, \cdots, a_i$. Typically, a linear search or a binary search technique can be used to find the element's proper location. A detailed description of these two algorithms can be found in Refs. 1, 10, 14, and 18. Both algorithms make use of an array and pointers.

A stack can also be used to implement sorting by insertion. In this case, two stacks called LEFT and RIGHT are used. The LEFT stack is used to push items in ascending order, whereas the RIGHT stack is used to push items in the descending order. The top of each stack at step i represents the insert point of element i. As a new item is being processed, the two stacks are shuffled to being the insert point to the top. After all the input elements are exhausted, the two stacks are combined onto the ascending-order stack. A minimum value (say $-\infty$) is used to indicate the end of the LEFT stack, and a minimum value (say $+\infty$) is used to indicate the end of the RIGHT stack.

The steps of the latter algorithm are shown in an example of seven numbers chosen at random in Figure 1. Figure 2 shows the algorithm written in PDP11 assembly language.

3.2 Analysis of the Algorithm and Its Time Formula

In this section, we present the asymptotic behavior and the detailed analysis of the stack-insertion sort algorithm. The reader should refer to Refs. 1, 10, 14, and 18 for the analysis of the straight-insertion sort and the binary-insertion sort algorithms. The analysis of these

	Initial key	:	20 67 07 31 53 11 6 55
	stack1	:	0
	stack2	:	100
step1	keys	:	67 07 31 53 11 6 55
	stack1	:	0
	stack2	:	100 20
step2	keys	:	07 31 53 11 6 55
	stack1	:	0 20
	stack2	:	100 67
step3	keys	:	31 53 11 6 55
	stack1	:	0 07
	stack2	:	100 67 20
step4	keys	:	53 11 6 55
	stack1	:	0 07 20
	stack2	:	100 67 31
step5	keys	:	11 06 55
	stack1	:	0 07 20 31
	stack2	:	100 67 53
step6	keys	:	06 55
	stack1	:	0 07 11
	stack2	:	100 67 53 31 20
step7	keys	:	55
	stack1	:	0 06
	stack2	:	100 67 53 31 20 11 07
step8	keys	:	--
	stack1	:	0 06 07 11 20 31 53
	stack2	:	100 67 55
step9	keys	:	--
	stack1	:	0 06 07 11 20 31 53 55 67
	stack2	:	100

Figure 1. An example of the stack-insertion sort algorithm.

```
         MOV #N1,R1        ; R1 IS THE LEFT STACK POINTER
         MOV #N2,R2        ; R2 IS THE RIGHT STACK POINTER
         MOV #MIN,(R1)     ; MIN IS THE LEFT BOTTOM ELEMENT
         MOV #MAX,(R2)     ; MAX IS THE RIGHT BOTTOM ELEMENT
         MOV #N,R3         ; N IS THE NUMBER OF ELEMENTS TO
                           ;       BE SORTED
         MOV #LOC,R4       ; R4 POINTS TO THE ELEMENTS TO
                           ;       BE SORTED
L1:      TST R3            ; IF R3 IS 0, TERMINATE
         BEQ L5
L2:      CMP (R4), (R1)    ; IF THE CURRENT ELEMENT IS > THE LEFT TOP
         BGT L3            ;   CHECK IT WITH THE RIGHT TOP
         MOV (R1)+,-(R2)   ; OTHERWISE, MOVE ELEMENTS FROM THE LEFT TO
         JMP L2            ;   THE RIGHT UNTIL THE CURRENT ELEMENT
                           ;   BECOMES > THE LEFT TOP
L3:      CMP (R4), (R2)    ; IF THE CURRENT ELEMENT IS < THE RIGHT TOP
         BGT L4            ;   THEN THIS THE INSERTION POINT
         MOV (R2)+,-(R1)   ; OTHERWISE, MOVE ELEMENTS FROM THE RIGHT TO
         JMP L3            ;   THE LEFT UNTIL THE CURRENT ELEMENT
                           ;   BECOMES < THE RIGHT TOP
L4:      MOV (R4)+,-(R2)   ; PUSH THE CURRENT ELEMENT INTO THE LEFT STACK
         DEC R3            ; PROCESS NEXT ELEMENT
         JMP L1
L5:      CMP (R1),#MIN     ; MOVE THE CONTENTS OF THE LEFT TO
         BEQ TERM          ;   TO THE RIGHT IF THERE IS ANY
         MOV (R1)+,-(R2)
         JMP L5
TERM :   HALT
```

Figure 2. Stack-insertion sort procedure.

algorithms provides the performance parameters necessary to construct their time formulas.

For the stack-insertion sort algorithm, the best case is when the input items are in order. In this case, each item is pushed with no movements at all and the algorithm best performance is of $O(n)$, where n is the number of elements to be sorted. The worst case is when each item has to be inserted at the bottom of the stack that contains most of the previously inserted elements. In this case, the number of comparisons and the number of movements are about $n^2/2$ and the algorithm performance will be of $O(n^2)$.

To obtain the performance equation of an algorithm, the performance parameters of the different steps must be expressed as a function of the number of elements. For the stack-insertion sort algorithm, the required number of iterations to complete the sorting part (loop L1) is equal to the table size n. During this sorting stage, elements may be moved from the LEFT to the RIGHT or vice versa (loop L2 and loop L3, respectively). Since the cost of a movement is the same in both directions (a pop + a push), it is enough to have the total number of movements. In the best case, this number 0, whereas in the worst case this number if equal to

$$0+1+1+3+3+5+5+\cdots+(n-2)+(n-2), \quad \text{if } n \text{ is odd}$$

and

$$0+1+1+3+3+5+5+\cdots+(n-3)+(n-3)+(n-1), \quad \text{if } n \text{ is even}$$

which is $(n-1)^2/2$ if n is odd, and $(n-2)^2/2+(n-1)$ if n is even. This can be written as ROUND $(n^2-2n/2+0.5)$ for all values of n. After sorting is completed, the elements of the LEFT are popped and pushed into the RIGHT (loop L5). The number of these elements in the worst case depends also upon whether n is even or odd. It is 1 if n is even and n if n is odd. This can be represented as $[(n+1) \bmod 2 + n*(n \bmod 2)]$.

These performance parameters were used to find the upper and lower bounds using PASS, and assuming that the PDP11 machine was the target machine. The same step was done for the straight-insertion sort and the

Table 1. Cost Expressions of Upper and Lower Bounds for Sorting by Insertion Algorithms[a]

	Best case (μsec)	Worst case (μsec)
Straight insertion	$102.1n-49.57$	$37.385n^2+64.72n-49.57$
Binary insertion	$129.7ni+53.3n-100.19$	$33.12n^2+129.66ni+149.84n-229.85$
Stack insertion	$41.65n+49$	$9.975n^2+41.65n+49$

[a] $i = \log_2 n$.

Table 2. Average Cost Expressions for Sorting by Insertion Algorithms[a]

	Average cost (μsec)
Straight insertion	$18.69n^2+158.18n-124.34$
Binary insertion	$16.56n^2+129.66ni+101.57n-165.02$
Stack insertion	$3.325n^2+48.3n+49$

[a] $i = \log_2 n$.

binary-insertion sort algorithms to determine the upper and lower bounds of each of them. Table 1 gives the performance equations of these boundaries for the three algorithms. Figure 3 represents these performance equations graphically.

Before dealing with the average cost expression of an algorithm, it is necessary to have the probability distribution of the inputs. From this distribution, the probabilistic properties of different performance parameters can be derived and used to find the average cost. For sorting, a natural assumption is that every permutation of the sequence to be sorted is equally likely to appear as an input.

We will use this assumption henceforth. This does not mean that it is the only assumption we can make. In some situations, the inputs may have other properties as being partly sorted or in reverse order. Such properties will heavily influence the expected performance of an algorithm and the designer's decision.

For the stack-insertion sort algorithm, only two performance parameters are influenced by the input distribution; namely, the number of movements between the LEFT and the RIGHT stacks during the sorting step (loop L2 and loop L3) and the number of elements in the LEFT stack after the sorting is completed (loop L5). To find the probability distribution of these two random numbers, an experimental study was used to measure them. It was found that the total number of movements between the two stacks during the sorting process is a positively skewed binomial distribution with p close to 0.5. This is expected since in every step the new element is pushed into the LEFT stack. The distribution will be symmetric if the element is pushed one time into the LEFT stack and another time into the RIGHT stack. The average number of movements is modeled as $(n^2 - 1)/6$. It is also found that the number of elements left in the LEFT stack after the sorting is completed is uniformly distributed between 1 and n.

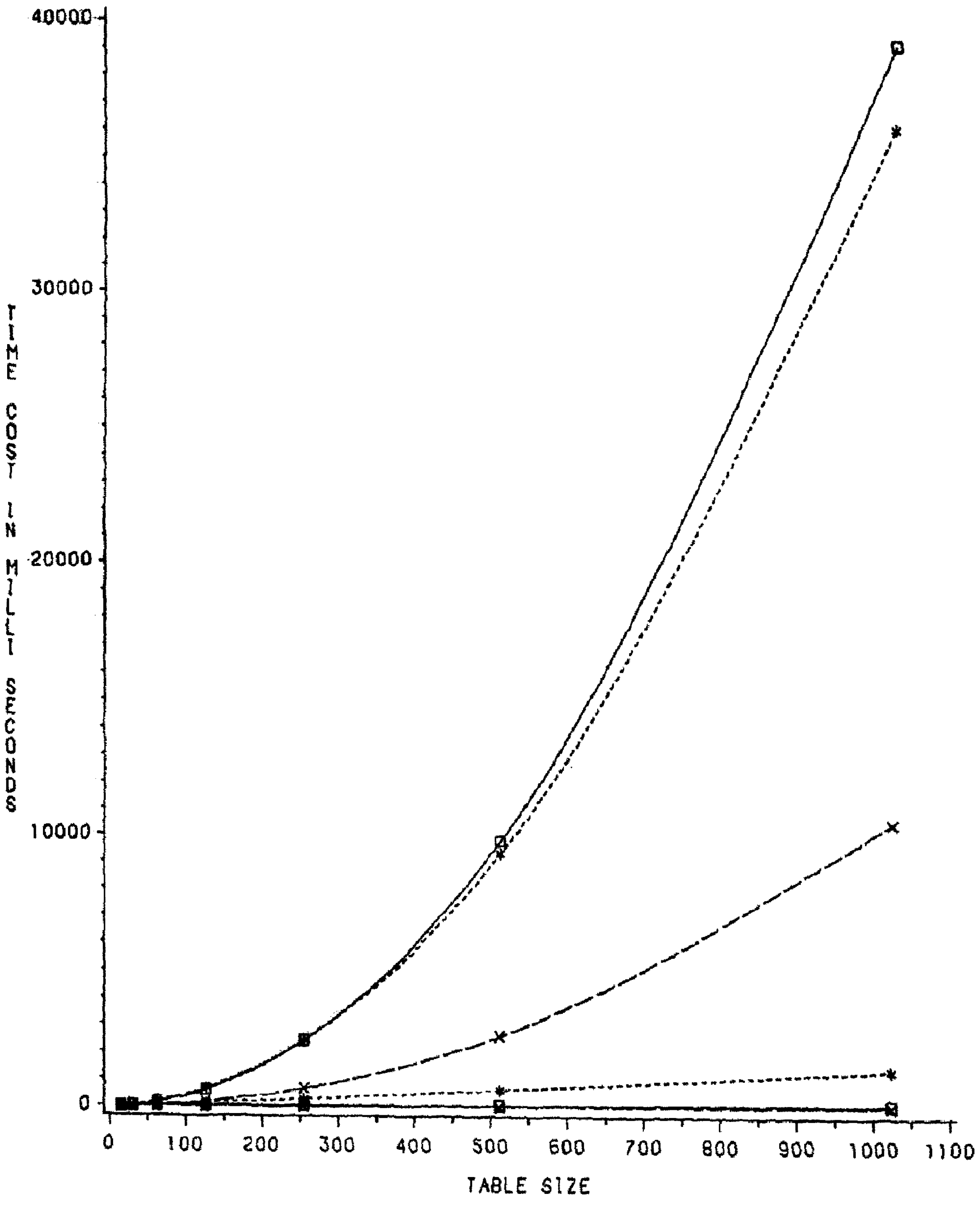
Using these results, the average performance equation for the stack-insertion sort algorithm is as shown in Table 2. Similar studies for the straight-insertion sort algorithm and binary-insertion sort algorithm produce the average performance equations shown in Table 2. Figure 4 represents these performance equations graphically. Needless to say, the stack-insertion sort algorithm has the best performance among the sorting-by-insertion algorithms for the three working conditions; namely, the best, the worst, and the average cases.

4. STACK-PARTITION SORT ALGORITHM

4.1 Description of the Algorithm

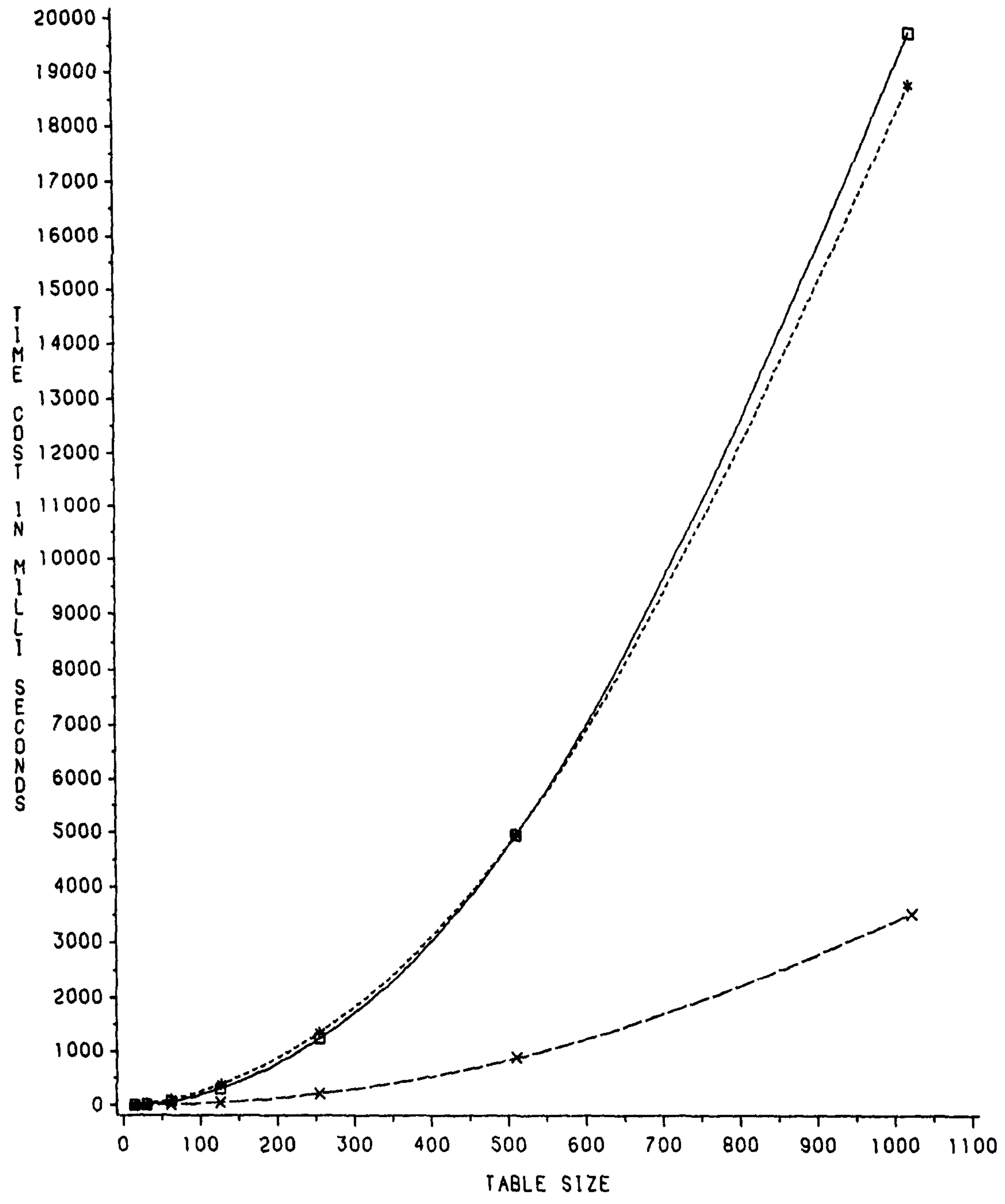
In this class of sorting algorithms, out-of-order pairs of elements interchange their positions until no more such pairs exist. The most popular algorithm using this technique is Hoare's Quicksort that selects an element as a reference (pivot) and partitions the elements into two sections, such that all elements smaller than the pivot are in the first section and all elements larger than the pivot are in the second section with the pivot itself between the two . This ensures that the pivot is in its correct location. The same process is applied recursively to both sections until all elements are in their correct locations. A detailed description of the quick sort algorithm can be found in Refs. 1, 10, 14, and 18.

The quick sort algorithm is implemented using an array to store the input, pointers to keep track of the start and the end of each partition, and an auxiliary stack to implement recursion. Another implementation, using stacks only, is due to Yuen [19]. In Yuen's version, three stacks called SOURCE, FIELD, and BASE are used to implement the quick sort algorithm. In the beginning, the whole vector is in the SOURCE stack. The top element is read out and used as a PIVOT and the rest of the elements are then divided between BASE and FIELD according to their value relative to the PIVOT (e.g., larger elements go to BASE and smaller ones to to FIELD). When SOURCE is empty, the PIVOT is then pushed on BASE and it is tagged to indicate that it is in its correction location.



——— straight -------- binary — — — — stack

Figure 3. Upper and lower bounds of sorting by insertion algorithms.



———— straight - - - - - - - - binary — — — — — stack

Figure 4. Average performance of sorting by insertion algorithms.

The SOURCE stack is exchanged with the FIELD stack and the same process is repeated until both the SOURCE stack and the FIELD stack become empty. At this moment, we exchange the SOURCE with the BASE and repeat the process again. Two things, however, are now different. First, the SOURCE stack has a number of sections separated by tagged elements. These elements cannot be selected as PIVOT again. Whenever a tagged element is at the top of the SOURCE, it is poped and pushed into the BASE until an untagged element is at the top. This latter element is used as a PIVOT and the partitioning process is repeated until a tagged element is met again. Second, because stacking a vector reverse the direction, we have to change the rule of allocating elements to BASE or FIELD (i.e., smaller elements go the BASE and larger ones go the FIELD). In other words, every time we exchange the SOURCE and the BASE, we have to change the direction of flow. This is controlled by a variable PITCH whose value may be either 0 or 1.

Two modifications are added to Yuen's algorithm. A test is added to see if the SOURCE is empty or not before trying to pop an element as a PIVOT. This is because the SOURCE sometimes contains only tagged elements and no more unsorted sections. After the algorithm is completed, the elements are in the BASE in order or in the reverse order. This depends upon table size and elements permutation. The user has to check the variable PITCH to decide about this before loading the elements back to the given array.

The above algorithm is illustrated by an example shown in Figure 5, whereas the algorithm written in PDP11 assembly language is shown in Figure 6.

4.2 Analysis of the Algorithm and Its Time Formula

No analysis is reported in Yuen's technical report for the stack-partition sort algorithm. Yuen, however, mentions that this algorithm shares the advantages and the disadvantages of the Quicksort algorithm. In order to verify Yuen's statement, the analysis of the algorithm is done by first investigating the behavior of the partitioning process. From this investigation, we can determine the input permutation that causes the best and the worst cases.

The best case occurs when the pivot is correctly positioned every time such that the number of the elements that are smaller than the pivot is equal to the number of elements that are larger than it (i.e., half of a partition goes to the stack BASE and the other half goes to the stack FIELD). Table 3 shows such sequences for n = 3, 7, 15, and 31. Such behavior is exactly similar to the best case for the Quicksort algorithm [18]. This leads to $O(n \log_2 n)$ as the best performance of the stack-partition sort algorithm.

The worst performance occurs when the pivot is greater than (or smaller than) all of the other elements of a partition. This happens when the input elements are either in the reverse order or in order. In this case, the algorithm is acting as the worst case of the straight-insertion sort algorithm and the worst case of the Quicksort algorithm. This means that its worst performance is of $O(n^2)$. Thus, the asymptotic boundaries of the stack-partition sort algorithm are identical to those of the Quicksort algorithm. Therefore, to compare the two algorithms, we must derive the performance equations for both. The analysis of the Quicksort algorithm is covered by different authors [1, 10, 14, 18]. For the stack-partition sort algorithm, it is necessary to model the performance parameters as a function of the array size n.

In the worst case, each iteration starts by selecting a pivot, sends all of the other elements to the BASE, and then exchanges the BASE with the SOURCE. Therefore, the number of times in which the FIELD is used is zero (loop L1), and the number of times in which the BASE and the SOURCE are exchanged is n (loop L0). After a pivot has been pushed into its proper location, loop L1 is executed one more time to pop all of the tagged elements left in the SOURCE and push them into the BASE. This occurs for all times except in the first two iterations where loop L1 is executed only once. This causes loop L1 to be executed.

$$2(n-2)+2=2(n-1) \text{ times}$$

It is also easy to observe that the number of tagged elements after the ith iteration is i and half of them are at the top of the SOURCE and the other half are at the bottom. From this observation, loop L2 is activated twice per iteration and it is executed i times in both activations. Hence, loop L2 is executed.

$$1+2+3+\cdots+(n-1)=\frac{n(n-1)}{2} \text{ times}$$

Similarly, the number of untagged elements after the ith iteration is $(n - i)$. The top of these elements is to be selected as pivot and the other $(n - i - 1)$ elements will be pushed into the BASE via loop L4. Therefore, loop L4 will be executed.

Table 3. Examples of Best Case Sequences

n	Elements
3	3 1 2
7	6 5 7 2 1 3 4
15	12 11 9 10 15 13 14 4 3 1 2 7 5 6 8
31	24 22 21 23 18 17 19 20 30 29 31 26 27 28 8 6 5 7 2 1 3 4 14 13 15 10 9 11 12 16

```
PITCH = 1          S   4 9 1 3 7 2 8
                   F
PIVOT = 4          B
-----------------------------------------------
PITCH = 1          F
                   S   2 3 1
PIVOT = 2          B   4* 8 7 9
-----------------------------------------------
PITCH = 1          S   1
                   F
PIVOT = 1          B   2* 3 4* 8 7 9
-----------------------------------------------
PITCH = 1          S
                   F
PIVOT = --         B   1* 2* 3 4* 8 7 9
-----------------------------------------------
PITCH = 0          B   2* 1*
                   F
PIVOT = 3          S   3 4* 8 7 9
-----------------------------------------------
PITCH = 0          B   4* 3* 2* 1*
                   F
PIVOT = 8          S   8 7 9
-----------------------------------------------
PITCH = 0          B   8* 7 4* 3* 2* 1*
                   S   9
PIVOT = 9          F
-----------------------------------------------
PITCH = 1          S   9* 8* 7 4* 3* 2* 1*
                   B
PIVOT = --         F
-----------------------------------------------
PITCH = 1          S   7 4* 3* 2* 1*
                   B   8* 9*
PIVOT = 7          F
-----------------------------------------------
PITCH = 1          S
                   B   1* 2* 3* 4* 7* 8* 9*
PIVOT = ---        F
```

Figure 5. An example of the stack-partition sort algorithm.

$$(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2} \text{ times}$$

Loop L3 is executed n times since there are n pivots selected. Test L12 is executed only once when we end with the PITCH equaling 0. This occurs if the table size is even. The mod function can be used to represent such a relation as $[(n + 1) \bmod 2]$. Finally, loop L15 will be executed n times if test L12 is true. Therefore, loop L15 will be executed $n*[(n + 1) \bmod 2]$ times.

It is not important to model the number of execution times of other branches since they do not appear in the

```
        MOV #NS,R0        ; R0 IS THE SOURCE STACK POINTER
        MOV #NB,R1        ; R2 IS THE BASE STACK POINTER
        MOV #NF,R2        ; R2 IS THE FIELD STACK POINTER
        MOV #N,R3         ; N IS THE NUMBER OF ELEMENTS TO
                          ;          BE SORTED
        MOV #1,R5         ; SET THE PITCH
        MOV #FLAG,(R1)    ; FLAG INDICATES THE BASE BOTTOM
        MOV #FLAG,(R2)    ; FLAG INDICATES THE FIELD BOTTOM

L0:     TST R3            ; ARE ALL ELEMENTS PROCESSED?
        BEQ L12           ; IF YES, EXIT
L1:     CMP (R0),#FLAG    ; IS THE SOURCE EMPTY?
        BEQ L10           ; IF YES, LOOK AT THE BASE STACK
L2:     BIT (R0),#040000  ; IS THE SOURCE TOP TAGGED?
        BEQ L3            ; IF NO, SELECT THE PIVOT
        MOV (R0)+,-(R1)   ; PUSH THE SOURCE TOP INTO THE BASE
        JMP L2            ; TEST NEXT SOURCE ELEMENT
L3:     CMP (R0),#FLAG    ; IS THERE MORE ELEMENTS IN THE SOURCE?
        BEQ L9            ; IF NO, EXCHANGE THE SOURCE AND THE BASE
        MOV (R0)+,R4      ; GET THE PIVOT
L4:     CMP (R0),#FLAG    ; IS THERE MORE ELEMENTS IN THE THE SOURCE?
        BEQ L8            ; IF NO, PUSH THE PIVOT INTO THE BASE
        BIT (R0),#040000  ; IF YES, IS THE SOURCE TOP TAGGED?
        BEQ L8            ; IF YES, PUSH THE PIVOT INTO THE BASE
        TST R5            ; TEST THE PITCH'S VALUE
        BEQ L6            ; IF 0 GO TO L6
        CMP (R0),R4       ; IF THE SOURCE TOP LESS THAN
        BLT L5            ; THE PIVOT, SEND IT INTO
        MOV (R0)+,-(R1)   ; THE BASE
        JMP L4            ; BACK TO PROCESS NEXT ONE
L5:     MOV (R0)+,-(R2)   ; OTHERWISE SEND IT INTO THE FIELD
        JMP L4            ; BACK TO PROCESS NEXT ONE
L6:     CMP (R0),R4       ; IF THE SOURCE TOP GREATER THAN
        BGT L7            ; THE PIVOT, SEND IT INTO
        MOV (R0)+,-(R1)   ; THE BASE
        JMP L4            ; BACK TO PROCESS NEXT ONE
L7:     MOV (R0)+,-(R2)   ; OTHERWISE SEND IT INTO THE FIELD
        JMP L4            ; BACK TO PROCESS NEXT ONE
L8:     BIS #040000,R4    ; TAG THE PIVOT
        MOV R4,-(R1)      ; PUSH THE PIVOT INTO THE BASE
        DEC R3            ; DECREMENT THE COUNTER
        CMP (R2), #FLAG   ; IS THE FIELD EMPTY?
        BEQ L9            ; YES, BACK TO TEST IF THE SOURCE CONDITION
        MOV R0,R4         ; NO, EXCHANGE THE SOURCE
        MOV R2,R0         ; WITH THE FIELD
        MOV R4,R2
L9:     JMP L1
L10:    MOV R0,R4         ; EXCHANGE THE SOURCE
        MOV R1,R0         ; WITH THE BASE
        MOV R4,R1
        TST R5            ; CHANGE THE PITCH'S VALUE
        BEQ L11
        CLR R5
        JMP L0            ; AND GO BACK FOR NEXT PASS
L11:    INC R5
        JMP L0
L12:    TST R5            ; IS PITCH = 0?
        BNE L13           ; NO, HALT
L15:    CMP (R0),#FLAG    ; YES, MOVE ALL ELEMENTS FROM
        BEQ L14           ; THE SOURCE TO THE BASE
        MOV (R0)+,-(R1)
        JMP 15
L14:    MOV R0,R4         ; THEN EXCHANGE THE SOURCE
        MOV R1,R0         ; WITH THE BASE
        MOV R4,R1
L13:    HALT              ; STOP
```

Figure 6. Stack-partition sort procedure.

algorithm's performance equation. However, it is straightforward to model them as a function of the table size n.

In the best case, each iteration starts by popping the untagged top element of an unsorted section of the SOURCE as a pivot whose value is such that exactly half of the section elements are larger than it and half are smaller. After this splitting process, half of the section under consideration is in the FIELD and the other half is in the BASE separated from other sections (if any) by one or more tagged elements. FIELD and SOURCE are exchanged if FIELD is not empty, and the splitting process continues until SOURCE become empty. In this case, BASE and SOURCE are exchanged and a new iteration starts.

Following the above procedure, if n is equal to $(2^k - 1)$, where $k = 1, 2, 3, \cdots$, SOURCE will have $(k - 1)$ sections after the first iteration whose lengths are $(2^{k-1} - 1)$, $(2^{k-2} - 1)$, $\cdots$, 1. During each iteration, SOURCE's unsorted sections are partitioned in a similar fashion into a number of unsorted sections. If a section length is 1, the partition process for this section terminates and the element is in its proper location. This sorting process terminates when all unsorted sections terminates. Based upon this partitioning process, the number of execution times of different algorithm steps in the best case are evaluated. We will now consider, without loss of generality, that n is equal to $(2^i - 1)$, where $i = 1, 2, 3, \cdots$.

The number of iterations required for a given size n is equal to the number of times necessary for all sections to be reduced to length 1, which is $\log_2 n$.

The algorithm goes through the loop L1 whenever the SOURCE is not empty. In this case, L1 will be executed a number of times equaling the number of the selected pivots during the iteration plus one. Therefore, the total number of times loop L1 is executed, is $(n + \log n - 2)$ times.

In each iteration of L1, loop L2 will be executed a number of times equaling to the number of the tagged elements produced after the previous iteration. This number is 0 for the first iteration, $\log n$ for the second, $[\log_2 n + \log n/2 + \cdots + 1]$ for the third and so on. This can be expressed in the following recurrence equation:

$$T_2(i) = T_2(i-1) + T_2(i-2) + \cdots + T_2(1) + i(i-1)$$
$$+ \sum_{j=2}^{i} (i-j)(2^{j-1}-1), \qquad T_2(1) = 0$$

whose solution is $n(k - 1)/2$, where $k = \log_2 n$.

The number of pivots used for a section of length m is an iteration is $\log_2 m$. Therefore, loop L3 is executed a number of times described by the following recurrence equation:

$$T_3(i) = T_3(i-1) + T_3(i-2) + \cdots + T_3(1) + i,$$
$$T_3(1) = 1$$

whose solution is $T_3 = n$.

The algorithm goes through the loop L4 $(m - 1)$ times for a given section of length m. This leads to the following recurrence equation:

$$T_4(i) = T_4(i-1) + T_4(i-2) + \cdots + T_4(1)$$
$$+ \sum_{j=0}^{\log_2 n} \left(\frac{n}{2^j} - 1\right), \qquad T_4(1) = 0$$

whose solution is $(k - 2)(n + 1)/2$, where $k = \log_2 n$.

The number of exchanges between FIELD and SOURCE during processing of a given section depends not only upon the section length but also upon the section location in the SOURCE. This number, which is equal to the $\log_2 m$ for the sections whose length m, is such that $\log_2 m$ is odd wherever they are. However, when $\log_2 m$ is even, this number becomes equal to $\log_2 m$ if the section is the last unsorted section in the SOURCE, and $\log_2 m$ otherwise. This proposes to have two cases for the analysis: one when $k = \log_2 n$ is even, and one when it is odd. By considering different section sizes for a given n, and using the above observation, each of the two cases can be represented by a recurrence equation as follows:

Even case:

$$T_8(i) = T_8(i-2) + 2^{(i-1)}, \qquad T_8(2) = 1$$

Odd case

$$T_8(i) = T_8(i-2) + 2^{(i-1)}, \qquad T_8(3) = 3$$

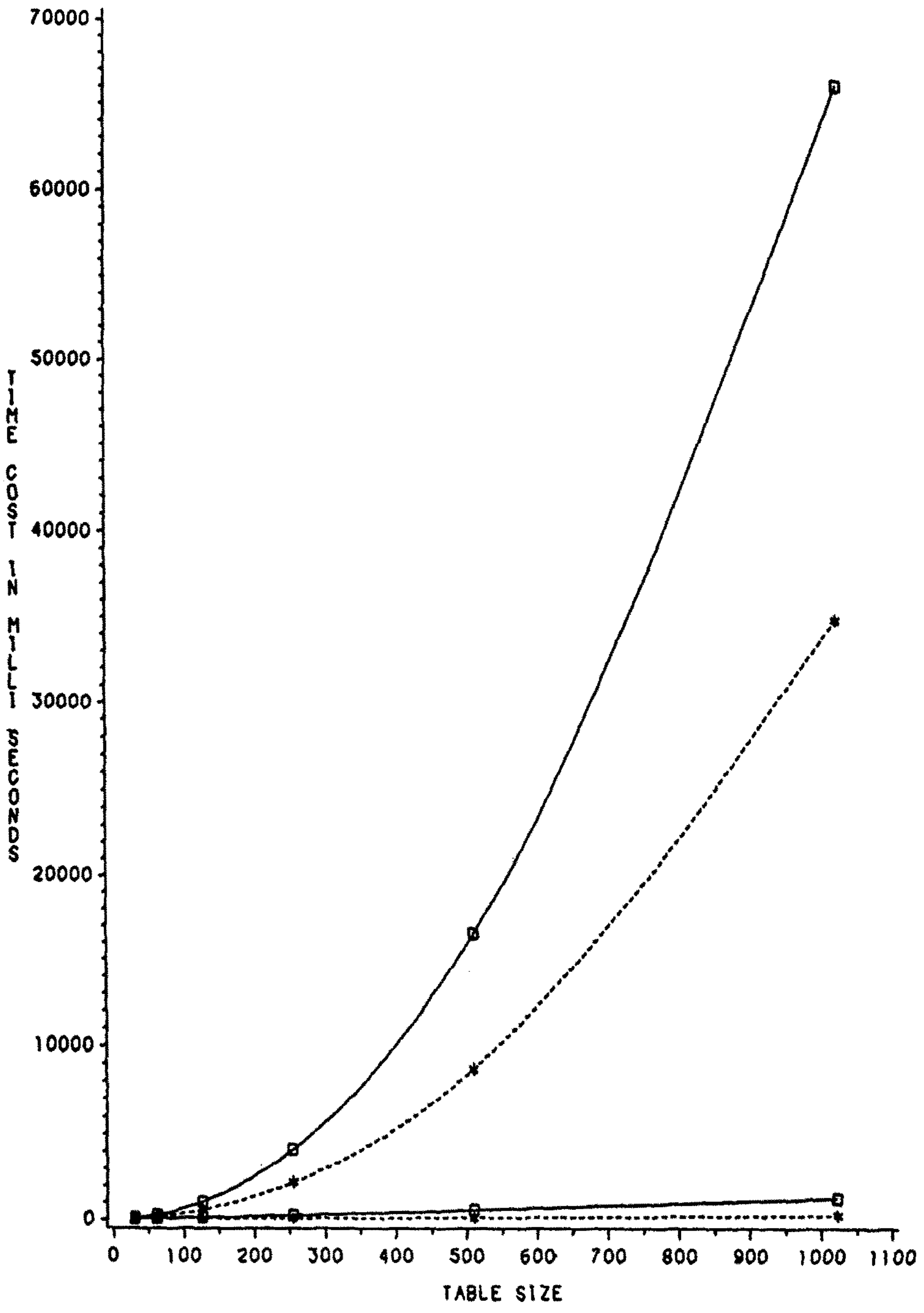
Both relations are solved, and the two solutions are combined together into TRUNC $(2n - 3/3)$.

Finally, test L12 is executed only once when we end with the PITCH equaling 0. This occurs when $\log_2 n$ is odd. Loop L15 is executed n times if test L12 is true. Therefore, loop L15 is executed $n*[(\log_2 n) \bmod 2]$.

As mentioned in the analysis of the worst case, the number of times for executing the other branches are not important since they do not appear in the performance equation.

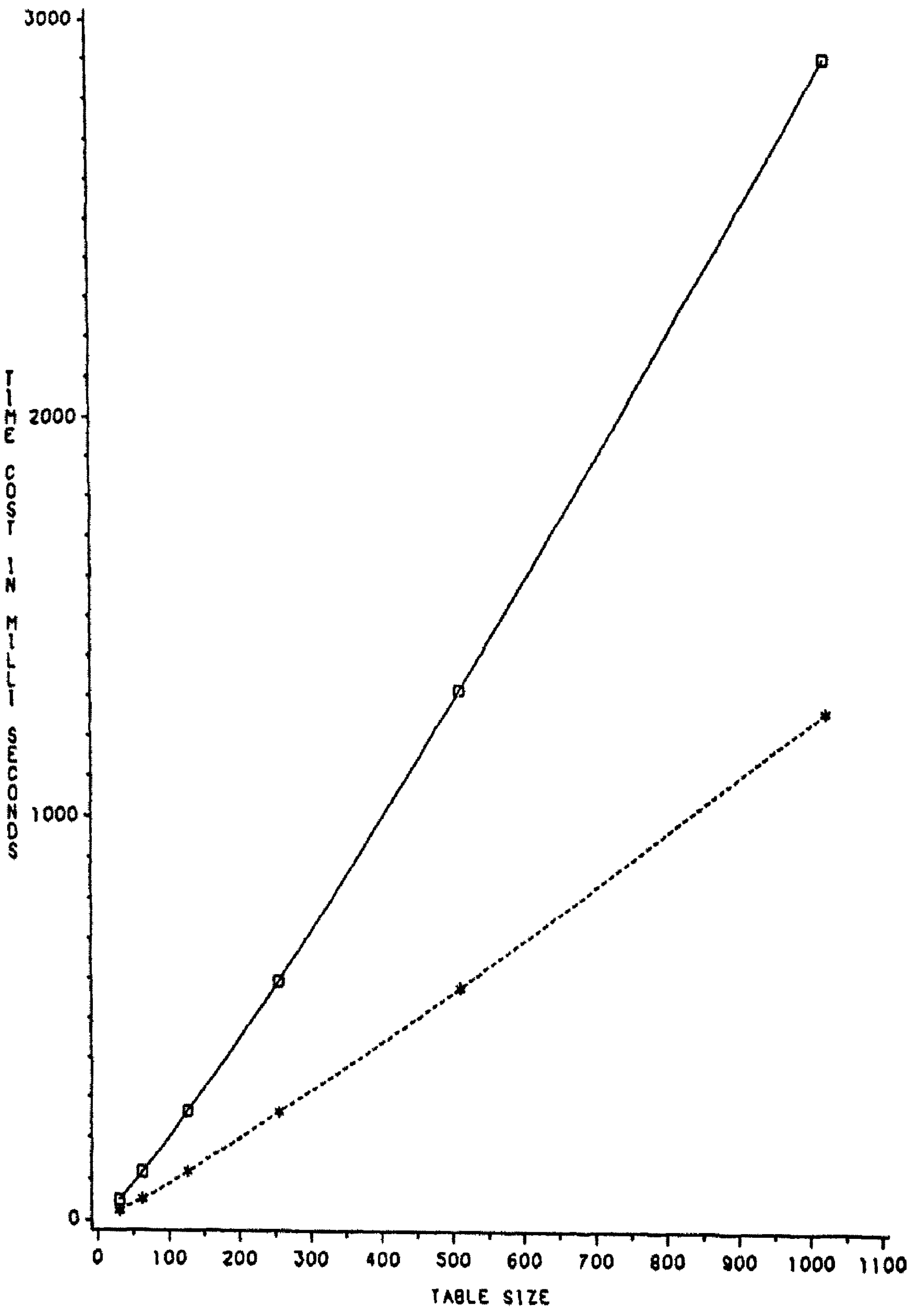
The performance parameters modeled above, in both the worst case and the best case, were used to find the upper and the lower bounds of the algorithm using PASS, and assuming that the PDP11 machine was the target machine. The same step was done for the Quicksort algorithm. Table 4 gives the performance equations of these boundaries for both algorithms. The same results are represented graphically in Figure 7.

Considering the average case, we will assume again



——————— **quick** - - - - - - - **stack**

Figure 7. Upper and lower bounds of sorting by exchange algorithms.



———— quick - - - - - - stack

Figure 8. Average performance of sorting by exchange algorithms.

Table 4. Cost Expressions of Upper and Lower Bounds for Sorting by Exchange Algorithms[a]

	Best case (μsec)	Worst case (μsec)
Stack	$23.1ni-13.65n+121.1i+10.5y+20.65z-56.7$	$33.25n^2+77n+20.65x-10.5$
Quick	$126.48ni-35.04+47.02$	$63.24n^2+16.22n-67.48$

[a] $i = \log_2 n$. $x = (n+1) \bmod 2$. $z = i \bmod 2$. $y = \text{trunc}(2n/3-1)$.

that all of the input permutations are equally likely. Instead of modeling all of the performance variables, we will use the recursion nature of the algorithm to get rid of most of them [1] and model the necessary ones only using measured values of the performance variables.

Initially, let us ignore the cost of deleting the tagged elements whenever they are found in the SOURCE (loop L2) and the cost of reversing the direction of the sorted elements when necessary (test L12 and loop L15). The average cost of these two parts will be added after finding the average cost of the remaining algorithm steps. Let $T(n)$ be the expected time required by the stack-partition algorithm to sort a sequence of n elements. Suppose that the element chosen as a pivot is the jth smallest element of the n elements. Then the other elements are partitioned between the FIELD and the BASE. This partition process has the cost $(a + bn)$, where a and b are constants. We can imagine the exchange operation between the SOURCE and the FIELD and between the SOURCE and the BASE as two recursive calls of the sorting algorithm. The first call is with an array of size $(j - 1)$, and the second call is with an array of size $(n - j)$. The cost of the first call is $T(j - 1)$ whereas, the cost of the second call is $T(n - j)$. Since j is equally likely to take on any value between 1 and n, we have the following relationship:

$$T(n)=(a+bn)+\frac{1}{n}\sum_{j=1}^{n}[T(j-1)+T(n-j)]$$

$$=(a+bn)+\frac{2}{n}\sum_{j=0}^{n-1}T(j)$$

The next step is to find the average cost of the two ignored parts under the assumption that all of the permutation are equally likely. This step requires measuring the behavior of the two performance parameters; loop L2 and test L12. From this measurement phase, the average number of iterations in loop L2 is approximated by $[0.5n(\log_2 n) - 1]$, whereas the average number of times we need to change the order of

Table 5. Average Cost Expressions of Sorting by Exchange Algorithms[a]

	Average case (μsec)
Stack	$102.55ni+92.4i+210.525n+233.05$
Quick	$252.96ni+252.96i+298.6n+472.18$

[a] $i = \log_2 n$.

the sorted elements is almost a constant for $n > 6(0.43)$ and close to 0.5 for $n \leqslant 6$.

Using these results, the average performance equation is as shown in Table 5. A similar study for the Quicksort algorithm produces the average cost shown in the same table. Figure 8 represents the same results graphically. Again, it is straightforward to notice that the stack-partition sort algorithm has better performance than the quicksort algorithm for the three working conditions; namely, the best, the worst, and the average cases.

5. CONCLUSIONS

Computer architects design a high-performance system out of the same components that the rest of us would use to build a common one. They do it through an understanding of how the system will actually be programmed. One result of this insight is stack computers. Our job is to use these architectures efficiently in different computer applications.

Having this motivation, this paper has introduced two sorting algorithms that are most suitable for stack computers than the traditional sorting algorithms. The first is based upon sorting by the insertion technique, whereas the second is based upon sorting by the exchange technique. The analysis of both algorithms shows that they score the best performance when run on stack computers as compared with other array sorting algorithms.

REFERENCES

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Massachusetts, 1976,
2. R. P. Blake, Exploring a Stack Architecture, *Computer,* 10, May 1977.
3. T. L. Booth, and C. A. Wiecek, Performance Abstract Data Types as a Tool in Software Performance Analysis and Design, *IEEE Trans. Software Eng.* (1980).
4. T. L. Booth, et al., PASS: A Performance-Analysis Software System to Aid the Design of High-Performance Software, *Proc. 1st Int. Conf. Computers and Applications;* June 1984,
5. Digital Equipment Corporation, *PDP11 Processor Handbook,* Mayard, Massachusetts, 1979,

6. Digital Equipment Corporation, *Microcomputer Processor Handbook,* Mayard, Massachusetts, 1980,
7. R. Doran, Architecture of stack machines, in *High-Level Language Computer Architecture,* (Yaohan Chu, ed.), Academic Press, New York, 1975.
8. E. A. Hauck, and B. A. Dent, Burroughs B6500/B7500 Stack Mechanism, *Proc. AFIPS 1968 Spring Joint Computer Conf.,* 32, AFIPS Press, Montvale, New Jersey, 1968,
9. Intel, *iAPX 432 General Data Processor Architecture Reference Manual,* Preliminary edition, Intel Corp., Aloha, Oregon, 1981,
10. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching,* Addison-Wesley, Reading, Massachusetts, 1973,
11. W. M. McKeeman, Stack computer, in *Introduction to Computer Architecture,* (Harold S. Stone, ed.), Science Research Associates, Inc., Chicago, Illinois, 1975.
12. E. I. Organick, *Computer System Organization: The B5700/B6700 Series,* Academic Press, New York, 1973,
13. A. Osborne, *An Introduction to Microcomputers,* Adam Osborne and Associates, Berkeley, California, 1975,
14. P. W. Purdom, and C. A. Brown, *The Analysis of Algorithms,* Holt, Rinehart and Winston, New York, 1985,
15. B. Qin, PASS—A Performance Analysis Tool for Software Designs, M.Sc. Thesis, Computer Science Department, University of Connecticut, Connecticut, 1984.
16. H. A. Sholl, and T. L. Booth, Software Performance Modeling Using Computations Structure, *IEEE Trans. Software Eng.* (1975).
17. T. T. Wetmore, The Performance Compiler—A Tool for Software Design, M.Sc. Thesis, Computer Science Department, University of Connecticut, Connecticut, 1980.
18. N. Wirth, *Algorithms + Data Structures* = Programs, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1976,
19. C. K. Yuen, A Stack-Based Method For Sorting, TR-A3-83, Centre of Hong Kong, 1983.