SIAM J. COMPUT.
Vol. 2, No. 4, December 1973

SET MERGING ALGORITHMS*

J. E. HOPCROFT† AND J. D. ULLMAN‡

Abstract. This paper considers the problem of merging sets formed from a total of n items in such a way that at any time, the name of a set containing a given item can be ascertained. Two algorithms using different data structures are discussed. The execution times of both algorithms are bounded by a constant times $nG(n)$, where $G(n)$ is a function whose asymptotic growth rate is less than that of any finite number of logarithms of n.

Key words. algorithm, algorithmic analysis, computational complexity, data structure, equivalence algorithm, merging, property grammar, set, spanning tree

1. Introduction. Let us consider the problem of efficiently merging sets according to an initially unknown sequence of instructions, while at the same time being able to determine the set containing a given element rapidly. This problem appears as the essential part of several less abstract problems. For example, in [1] the problem of "equivalencing" symbolic addresses by an assembler was considered. Initially, each name is in a set by itself, i.e., it is equivalent to no other name. An assembly language statement that sets name A equivalent to name B by implication makes C equivalent to D if A and C were equivalent and B and D were likewise equivalent. Thus, to make A and B equivalent, we must find the sets (equivalence classes) of which A and B are currently members and merge these sets, i.e., replace them by their union.

Another setting for this problem is the construction of spanning trees for an undirected graph [2]. Initially, each vertex is in a set (connected component) by itself. We find edges (n, m) by some strategy and determine the connected components containing n and m. If these differ, we add (n, m) to the tree being constructed and merge the components containing n and m, which now are connected by the tree being formed. If n and m are already in the same component, we throw away (n, m) and find a new edge.

A third application [3] is the implementation of property grammars [4], and many others suggest themselves when it is realized that the task we discuss here can be done in less than $O(n \log n)$ time.

By way of introduction, let us consider some of the more obvious data structures whereby objects could be kept in disjoint sets, these sets could be merged, and the name of the set containing a given object could be determined. One possibility is to represent each set by a tree. Each vertex of the tree would correspond to an object in the set. Each object would have a pointer to the vertex representing it, and each vertex would have a pointer to its father. If the vertex is a root, however, the pointer would be zero to indicate the absence of a father. The name of the set is attached to the root.

* Received by the editors August 10, 1972, and in revised form May 18, 1973. This research was supported by the National Science Foundation under Grant GJ-1052 and the Office of Naval Research under Contract N00014-67-A-0071-0021.

† Department of Computer Sciences, Cornell University, Ithaca, New York 14850.

‡ Department of Electrical Engineering, Princeton University, Princeton, New Jersey 08540.

Given the roots of two trees, one can replace the representation of two sets by a representation for their union by making the pointer at one root point to the other root and, if necessary, updating the name at the root of the combined tree. Thus two structures can be merged in a fixed number of steps, independent of the size of the sets. The name of the set containing a given object can be found, given the vertex corresponding to the object, by following pointers to the root.

However, by starting with n trees, each consisting of a single vertex, and successively merging the trees together until a single tree is obtained, it is possible to obtain a representation for a set of size n, which consists of a chain of n vertices. Thus, in the worst case, it requires time proportional to the size of the set to determine which set an object is in.

For purposes of comparison, assume that initially there are n sets, each containing exactly one object, and that sets are merged in some order until all items are in one set. Interspersed with the mergers are n instructions to find the set containing a given object. Then the tree structure defined above has a total cost of n for the merging operation and a cost bounded by n for determining which set contains a given object (total cost n^2 for n look ups).

Methods based on maintaining balanced trees (see [5], e.g.) have a total cost of $n \log n$ for the merging operations and a cost bounded by $\log n$ for determining which set contains a given object (total cost $n \log n$ for n look ups).

A distinct approach is to use a linear array to indicate which set contains a given object. This strategy makes the task of determining which set contains a given object finite. By renaming the smaller of the two sets in the merging process, the total cost of merging can be bounded by $n \log n$.

A more sophisticated version of the linear array replaces the set names in the array by pointers to header elements. This method, based on the work of Stearns and Rosenkrantz [3], uses $n \underbrace{\log \log \cdots \log}_{k}(n)$ steps for the merging process and a fixed number of steps independent of n for determining which set contains a given element. Here k is a parameter of the method and can be any fixed integer.

In what follows, we shall make use of a very rapidly growing function and a very slowly growing function which we define here. Let $F(n)$ be defined by

$$F(0) = 1,$$

$$F(i) = F(i-1)2^{F(i-1)} \quad \text{for} \quad i \geqq 1.$$

The first five values of F are 1, 2, 8, 2048 and 2^{2059}.

The slowly growing function $G(n)$ is defined for $n \geqq 0$ to be the least number k such that $F(k) \geqq n$. Both set merging algorithms presented here have asymptotic growth rates of at most $O(nG(n))$.

Initially, let $S_i = \{i\}$, $1 \leqq i \leqq n$. The S_i's are *set names*. Given a sequence of two types of instructions,

$$\text{(i) MERGE}(i, j), \qquad \text{(ii) FIND}(i),$$

where i and j are distinct integers between 1 and n, we wish to execute the sequence from left to right as follows. Each time an instruction of the form MERGE(i, j) occurs, replace sets S_i and S_j by the set $S_i \cup S_j$ and call the resulting set S_j. Each time an instruction of the form FIND(i) occurs, print the name of the set currently containing the integer i. It is assumed that the length of the sequence of instructions is bounded by a constant times n. Both set merging algorithms presented here have asymptotic growth rates bounded by $nG(n)$. The first algorithm actually requires $nG(n)$ steps for certain sequences. For the second algorithm, it is unknown whether $nG(n)$ is in fact its worst case performance. Recently, Tarjan [6] has shown the algorithm to be worse than $O(n)$.

2. The first set merging algorithm. The first set merging algorithm represents a set by a data structure which is a generalization of that used in [3] to implement property grammars. The basic structure is a tree similar to that shown in Fig. 1, where the elements of the set are stored at the leaves of the tree. Links are assumed to point in both directions.

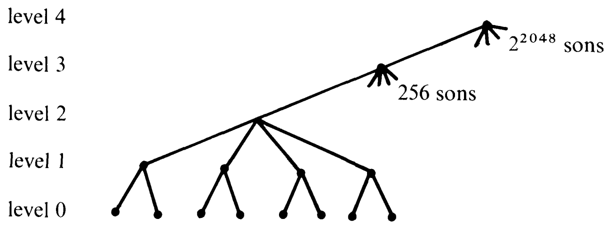

FIG. 1. *Data structure for set representation*

Each vertex at level i, $i \geqq 1$, has between one and $2^{F(i-1)}$ sons and thus at most $F(i)$ descendants which are leaves. We define a *complete* vertex as follows:

(i) Any vertex at level 0 is complete.

(ii) A vertex at level $l \geqq 1$ is complete if and only if it has $2^{F(l-1)}$ sons, all of which are complete.

Otherwise, a vertex is *incomplete*.

The data structure is always maintained, so that no vertex has more than one incomplete son. Furthermore, the incomplete son, if it exists, will always be leftmost, so it may be easily found. Attached to each vertex is the level number, the number of sons, and the number of descendant leaves. The name of the set is attached to the root.

The procedure COMBINE, given below, takes two data structures of the same level l and combines them to produce a single structure of level l. If there are too many leaves for a structure of level l, then COMBINE produces two structures of level l, one with a complete root and one with the remaining leaves. To simplify understanding of the algorithm, the updating of set names, level numbers, number of sons and the number of descendant leaves for each vertex has been omitted.

Procedure COMBINE (s_1, s_2).

Assume without loss of generality that s_2 has no more sons than s_1 (otherwise permute the arguments).

1. If both s_1 and s_2 have incomplete sons, say v_1 and v_2, then call COMBINE (v_1, v_2). On completion of this recursive call of COMBINE, the original data structure is modified as follows. If originally the total number of leaves on the subtrees with roots v_1 and v_2 did not exceed the number of leaves for a complete subtree, then the new subtree with root v_1 contains all leaves of the original two subtrees. The new subtree with root v_2 consists only of the vertex v_2.

If originally the total number of leaves exceeded the number for a complete subtree, then the new subtree with root v_1 is a complete subtree whose leaves are former leaves of the original subtrees with roots v_1 and v_2, and the new subtree with root v_2 is an incomplete subtree with the remaining leaves. If on completion of the recursive call of COMBINE, vertex v_2 has no sons, then delete vertex v_2 from the data structure.

2. Make sons of s_2 be sons of s_1, until either s_2 has no more sons or s_1 has $2^{F(l-1)}$ sons.

3. If s_2 still has sons, then interchange the lone incomplete vertex at level $l - 1$, if any, with the leftmost son of s_2. Otherwise, interchange the incomplete vertex with the leftmost son of s_1.

We now consider the first algorithm for the MERGE–FIND problem.

ALGORITHM 1.

1. Initially n vertices numbered 1 to n are created and treated as structures of level 0. Each vertex has information giving the name of its set, its number of descendant leaves (1), its level (0) and the number of its sons (0). A linear array of size n is created, such that the ith cell contains a pointer to vertex i.

2. The sequence of instructions of the forms MERGE(i, j) and FIND(i) are processed in order of occurrence.
   (a) For an instruction of the form FIND(i), go to the ith cell of the array, then to the vertex pointed to by that cell, then proceed from the vertex to the root and print the name at the root.
   (b) For an instruction of the form MERGE(i, j), if the roots for structures i and j are not of the same level, then add a chain of vertices above the root of lower level so that the roots of the two trees will have the same level. Let s_1 and s_2 be the two roots. Execute COMBINE(s_1, s_2). If after the execution of COMBINE s_2 has no sons, then discard s_2 and we are finished. Otherwise, create a new vertex whose left son is s_1 and whose right son is s_2.

We now show that Algorithm 1 requires time bounded by a constant times $nG(n)$, provided that the length of the sequence of instructions is itself first bounded by a constant times n. The first step is to observe that Algorithm 1 preserves the property that at most one son of any vertex is incomplete.

LEMMA 1. *At any time during the stimulation of a sequence of* MERGE *and* FIND *instructions by Algorithm* 1, *each vertex has at most one incomplete son. Furthermore, the incomplete son, if it exists, will always be leftmost.*

Proof. The proof proceeds by induction on the number of MERGE instructions simulated. The basis zero is trivial, since each leaf is complete by definition.

For the inductive step, we observe by induction on the number of applications of COMBINE that applying COMBINE to two trees, each of which have the desired property, will result in either producing a single tree with the desired property plus a single vertex or producing a complete tree plus another tree with the desired property. Thus, no call of COMBINE can give a vertex two incomplete sons if no vertex had two such sons previously. Furthermore, if two trees are produced, neither consisting of a single vertex, then one must be complete.

As a consequence of the above observation, if a new vertex with sons s_1 and s_2 is created in step 2(b) of Algorithm 1, the subtree with root s_1 is complete, and the property holds for the new vertex.

LEMMA 2. *If we begin with n objects, no vertex created during Algorithm* 1 *has level greater than* $G(n)$.

Proof. If there were such a vertex v, it could only be created in step 2(b). It would, by Lemma 1, have a complete descendant at level $G(n)$. An easy induction on l shows that a complete vertex at level l has $F(l)$ descendant leaves. Thus, v has at least $F(G(n)) + 1$ descendant leaves, which implies $F(G(n)) + 1 \leqq n$. The latter is false by definition of G.

THEOREM 1. *If Algorithm* 1 *is executed on n objects and a sequence of at most m* MERGE *and* FIND *instructions,* $m \geqq n$, *then the time spent is* $O(mG(n))$.

Proof. By Lemma 2, each FIND may be executed in $O(G(n))$ steps, for a total cost of $O(mG(n))$ for the FIND's. Since there can be at most $n - 1$ MERGE's, the cost of simulating the MERGE operations, exclusive of calls to COMBINE, is $O(n)$. Moreover, by Lemma 2, each MERGE can result in at most $G(n)$ recursive calls to COMBINE, as each call is with a pair of vertices at a lower level than previously. Thus, the cost of all calls to COMBINE is $O(nG(n))$ plus the cost of shifting vertices from one father to another in step 2 of COMBINE.

It remains only to show that the total number of shifts of vertices over all calls of COMBINE is bounded by a constant times $nG(n)$. In executing an instruction MERGE(s_1, s_2), no more than $G(n)$ incomplete vertices are shifted, at most one at each level. Thus we need only count shifts of complete vertices.

Consider step 2 of COMBINE. The new subtree with root s_2 is referred to as the CARRY unless the subtree consists solely of the vertex v_2 in which case we say, "there is no CARRY." The new subtree with root s_1 is referred to as the RESULT. The number of shifts of complete vertices is counted as follows. If a complete vertex is shifted, and there is no CARRY at this execution of COMBINE, charge the cost to the vertex shifted. If there is a CARRY, set the cost of each vertex in the CARRY to zero, and distribute the cost uniformly among the vertices in the RESULT.

Each time a vertex is moved, either its new father has at least twice as many sons as its old father, or there is a CARRY. Thus, a vertex at level i is moved at most $F(i)$ times before a CARRY is produced. Once a CARRY is produced, the root of the RESULT is complete, and its sons are never moved again. Hence, a vertex at level i can accumulate a cost of at most $2F(i)$, that is, $F(i)$ due to charges to itself and $F(i)$ for its share of the costs previously charged to the sons of the root of CARRY.

To compute the total cost of shifting complete vertices, note that a complete vertex at level i has $F(i)$ descendant leaves. Redistribute the cost of each complete vertex uniformly among its descendant leaves. Since no leaf has more than $G(n)$ ancestors, the cost charged to any leaf is bounded by $2G(n)$. Hence, the complete cost of moving vertices is $O(nG(n))$. Since $m \geqq n$ is assumed, we have our result.

COROLLARY. *If we may assume $m \leqq an$ for some fixed constant a, then Algorithm 1 is $O(nG(n))$.*

It should be observed that the set merging algorithm can be modified to handle a problem which is in some sense the inverse of set merging. Initially given a set consisting of integers $\{1, 2, \cdots, n\}$, execute a sequence of two types of instructions:

(i) PARTITION(i), (ii) FIND(i),

where i is an integer between 1 and n. Each time an instruction of the form PARTITION(i) occurs, replace the set S containing i into two sets:

$$S_1 = \{j | j \in S \text{ and } j \leqq i\},$$
$$S_2 = \{j | j \in S \text{ and } j > i\}.$$

Each set is named by the largest integer in the set. Each time an instruction of the form FIND(i) occurs, print the name of the set currently containing the integer i. To handle this partitioning problem, use the same data structure as before, but start with a single tree with n leaves. The leaves in order from left to right correspond to the integers from 1 to n. To execute PARTITION(i), start at the leaf corresponding to the integer i and traverse a path to the root of the tree to partition the tree into two trees.

All vertices to the left of the path are placed on one tree, all vertices to the right of the path are placed on the other. Vertices on the path are replaced by two vertices, one for each subtree, unless the vertex i is the rightmost descendant leaf of the vertex, in which case the vertex is placed on the left subtree. Assume that vertices v and w are on the path and w is a son of v. The sons of v are partitioned as follows. Simultaneously, start counting the sons of v to the left of w, including w, starting with the leftmost son of v, and start counting the sons of v to the right of w. Cease counting as soon as one of the two counts is complete. (The reason for the simultaneous counting is to make the cost of counting proportional to the smaller of the two counts.) The sons of v can now be partitioned at a cost proportional to the smaller of the two sets by moving the smaller number of sons.

The analysis of the running time is similar to that of the set merging algorithm, and thus only a brief sketch is given. Note that a vertex can have at most two incomplete sons, only one of which can be moved in the execution of any PARTITION instruction. Thus at most $G(n)$ incomplete vertices are moved in executing one PARTITION instruction.

To bound the cost of moving a complete vertex, note that each time a vertex is moved, its new father has at most half as many sons as the old father. Thus a vertex at level i can be moved at most $F(i)$ times. Since a complete vertex at level i has $F(i)$ leaves, distributing to its leaves the cost of all moves of a given vertex while it is complete gives at most a cost of one to each of its leaves. Since a leaf has at most $G(n)$ ancestors, the cost of moving all complete vertices is bounded by $nG(n)$.

3. The second set merging algorithm.

We now consider a second algorithm to simulate a sequence of MERGE and FIND instructions.

This algorithm also uses a tree data structure to represent a set. But here, all vertices of the tree, rather than just the leaves, correspond to elements in the set. Moreover, tree links point only from son to father.

ALGORITHM 2.

1. Initially each set $\{i\}$, $1 \leqq i \leqq n$, is represented by a tree consisting of a single vertex with the name of the set at the vertex.

2. To merge two sets, the corresponding trees are merged by making the root of the tree with fewer vertices a son of the root of the other tree. Ties can be broken arbitrarily. Attach the appropriate name to the remaining root.

3. To execute FIND(i), follow the path from vertex i to the root of its tree and print the name of the set. Make each vertex encountered on the path a son of the root. (This is where the algorithm differs substantially from balanced tree schemes!)

The above algorithm, except for the balancing feature of merging small trees into large, was suggested by Knuth [7] and is attributed by him to Tritter. The entire algorithm, including this feature, was implemented by McIlroy [2] and Morris in a spanning tree algorithm. The analysis of the algorithm without the balancing feature was completed by Fischer [8], who showed that $O(n \log n)$ is a lower bound, and Paterson [9], who showed it to be an upper bound. Our analysis shows that the algorithm with the balancing scheme is $O(nG(n))$ at worst. Thus it is substantially better than the one without the balancing.

We now introduce certain concepts needed to analyze Algorithm 2. Let α be a fixed sequence of MERGE and FIND instructions. Let v be a vertex. Define the *rank* of v, with respect to α, denoted $R(v)$, as follows. If in the execution of α by Algorithm 2, v never receives a son, then $R(v) = 0$. If v receives sons $v_1, v_2, \cdots, v_k$ at any time during the execution, then the rank of v is $\max\{R(v_i)\} + 1$. It is not hard to prove that the rank of v is equal to the length of the longest path from v to a descendant leaf in the tree that would occur if the FIND instructions and their attendant movement of vertices were ignored in the execution of α.

LEMMA 3. *If the rank of v with respect to α is r, then at some time during the execution of α, v was the root of a tree with at least 2^r vertices.*

Proof. The proof is by induction on the value of r. The case $r = 0$ is trivial. Assume the lemma to be true for all values up to $r - 1$, $r \geqq 1$. If v is of rank r, at some point v must become the father of some vertex u of rank $r - 1$. This event could not occur during a FIND, for if it did, then u would have previously been a descendant of v with some vertex w between them. But then w is of rank at least r and v of rank at least $r + 1$ by the definition of rank.

Thus, u must be the root of a tree T which is merged with a tree T' having root v. By the inductive hypothesis, T has at least 2^{r-1} vertices, since a root cannot lose descendants and a nonroot cannot gain descendants, and hereafter u will no longer be a root. By step 2 of Algorithm 2, T' has at least as many descendants as T. The resulting tree has at least 2^r vertices and has v as root.

LEMMA 4. *Each time a vertex v gets a new father w during the execution of Algorithm 2, that father has a rank higher than any previous father $u \neq w$ of v.*

Proof. If v, formerly a son of u, becomes a son of w, it must be during a FIND. Then w is an ancestor of u and hence of higher rank by definition.

LEMMA 5. *For each vertex v and rank r, there is at most one vertex u of rank r which is ever an ancestor of v.*

Proof. Suppose there were two such u's, say u_1 and u_2. Assume, without loss of generality, that v first becomes a descendant of u_1. Then u_2 is of higher rank than u_1 by Lemma 4 and the fact that at all times, paths up trees are of monotonically increasing rank. This contradicts the assumption that u_1 and u_2 were of rank r.

LEMMA 6. *There are at most $n/2^r$ vertices of rank r.*

Proof. Each vertex of rank r at some point is the root of a tree with at least 2^r vertices by Lemma 3. No vertex could be the descendant of two vertices of rank r by Lemma 5. This implies that there are at most $n/2^r$ vertices of rank r.

For $j \geqq 1$, define $\mathscr{g}_j$, the *j-th rank group*, as follows:[1]

$$\mathscr{g}_j = \{v \mid \log^{j+1}(n) < R(v) \leqq \log^j(n)\}.$$

Note that the higher the rank group, the lower the ranks of the vertices it contains.

LEMMA 7. $|\mathscr{g}_j| \leqq 2n/\log^j(n)$.

Proof. Since there are at most $n/2^r$ vertices of rank r, we have

$$|\mathscr{g}_j| \leqq \sum_{i=\log^{j+1}n}^{\log^j n} \frac{n}{2^i} \leqq \frac{n}{\log^j n}(1 + \tfrac{1}{2} + \tfrac{1}{4} + \cdots) \leqq \frac{2n}{\log^j n}\,.$$

LEMMA 8. *Each vertex is in some $\mathscr{g}_j$ for $1 \leqq j \leqq G(n) + 1$.*

Proof. By Lemma 6, no vertex has rank greater than $\log n$, so $j \geqq 1$ may be assumed. Thus to prove $j \leqq G(n) + 1$, we need only show that $\log^{G(n)+2}(n) < 0$. From the definition of $G(n)$, $n \leqq F(G(n))$, and so

$$\log^{G(n)+2}(n) \leqq \log^{G(n)+2}F(G(n)).$$

Thus it suffices to show that $\log^{i+2}F(i) < 0$. We shall actually show that $\log^{i+2}2F(i) \leqq 0$. The proof is by induction on i. For $i = 0$, the result is obvious. Assume the induction hypothesis true up to $i - 1$. Then $\log^{i+2}2F(i) \leqq \log^{i+1}(1 + \log F(i)) = \log^{i+1}(1 + \log F(i-1) + F(i-1))$, which is less than or equal to $\log^{i+1}2F(i-1)$, since $F(i-1) \geqq 1$ for $i \geqq 1$ and since $\log x \leqq x - 1$ for integer $x \geqq 1$. Thus by the inductive hypothesis, $\log^{i+2}2F(i) \leqq 0$. From this it follows that $\log^{i+1}F(i) < 0$, and the proof is complete.

THEOREM 2. *Given $n \geqq 2$ objects, the time necessary to execute any sequence of $m \geqq n$* MERGE *and* FIND *instructions on these objects by Algorithm* 2 *is $O(mG(n))$.*

Proof. The cost of the MERGE instructions is clearly $O(n)$. The cost of executing all FIND instructions is proportional to the number of instructions plus the sum, over all FIND's, of the number of vertices moved by that FIND. We now show that this sum is bounded by $O(mG(n))$.

Let v be a vertex in $\mathscr{g}_j$. If before a move of v the father of v is in a lower rank group (smaller value of j), assign the cost to the FIND instruction. Otherwise, assign the cost to v itself. For an instruction FIND(i), consider the path from vertex i to the root of its tree. The ranks of vertices along the path to the root are monotonically increasing, and hence there can be on the path at most $G(n)$ vertices whose fathers are in a lower rank group. Hence no FIND instruction is assigned a cost more than $G(n)$.

By Lemma 7, there are at most $2n/\log^j n$ vertices in $\mathscr{g}_j$, and by Lemma 4, each vertex in $\mathscr{g}_j$ can be moved at most $\log^j n$ times before its new father is in $\mathscr{g}_{j-1}$ or a lower rank group. Thus, the total cost of moving vertices in $\mathscr{g}_j$, not counting moves of a vertex whose father is in a lower rank group, is $2n$. Since there are at most $G(n) + 1$ rank groups, the total cost exclusive of that charged to FIND's is $O(nG(n))$. Hence, the total cost of executing the sequence of MERGE and FIND instructions is $O(mG(n))$.

[1] $\log^0(n)$ is n and $\log^{j+1}(n) = \log(\log^j(n)) = \log^j(\log(n))$. Base 2 logarithms are assumed throughout.

COROLLARY. *If $m \leqq an$ for fixed constant a, then Algorithm 2 requires $O(nG(n))$ time.*

4. An application. One application of the set merging algorithms is to process a sequence of instructions of the forms INSERT(i), $1 \leqq i \leqq n$, and MIN. Start with a set S which is initially the empty set. Each time an instruction INSERT(i) is encountered, adjoin the integer i to the set S. Each time a MIN instruction is executed, delete the minimum element from the set S and print it. We assume that for each i, the instruction INSERT(i) appears at most once in the sequence of instructions, and at no time does the number of MIN instructions executed exceed the number of INSERT instructions executed. Note that as a special case, we could sort k integers from one to n by executing INSERT instructions for each integer, followed by k MIN instructions.

The algorithm which we shall give for this problem is *off-line*, in the sense that the entire sequence of instructions must be present before processing can begin. In contrast, Algorithms 1 and 2 are *on-line*, as they can execute instructions without knowing the subsequent instructions with which they will be presented.

Let $I_1, I_2, \cdots, I_r$ be the sequence of INSERT and MIN instructions to be executed. Note that $r \leqq 2n$. Let l of the instructions be MIN. We shall set up a MERGE-FIND problem whose on-line solution will allow us to simulate the INSERT-MIN instructions. We create l objects M_i, $1 \leqq i \leqq l$, where M_i "represents" the ith MIN instruction. We also create n objects X_i, $1 \leqq i \leqq n$, where X_i "represents" the integer i. Suppose, in addition, that there are two arrays which, given i, enable us to find M_i or X_i in a single step. The following algorithm determines for each i, that MIN instruction, if any, which causes i to be printed. Once we have that information, a list of the integers printed by $I_1, I_2, \cdots, I_r$ is easily obtained in $O(n)$ steps.

ALGORITHM 3.

```
1. Initially use MERGE instructions to create the following sets:
     (i) $S_i$, $2 \leqq i \leqq l$, consists of object $M_{i-1}$ and all those $X_j$ such that
         INSERT($j$) appears between the $i - 1$st and $i$th MIN instructions.
    (ii) $S_1$ consists of all $X_j$ such that INSERT($j$) appears prior to the first
         MIN.
   (iii) $S_\infty$ consists of $M_l$ and all $X_j$'s not placed by (i) or (ii).
2. for $i \leftarrow 1$ until $n$ do
     begin
       FIND($X_i$);[2]
       let $X_i$ be in $S_j$;
       if $j \neq \infty$ then
       begin
         TIME($i$) $\leftarrow j$;
         FIND($M_j$);
         let $M_j$ be in $S_k$;
         MERGE($S_j, S_k$)
       end
     end
```

[2] We are taking the liberty of using X_i, M_i and S_i as arguments of FIND and MERGE, rather than integers, as these instructions were originally defined. It is easy to see that objects and set names could be indexed, so no confusion should arise.

The strategy behind Algorithm 3 is to let M_k after the ith iteration of step 2 lie in that set S_j such that the jth MIN instruction is the first one following the kth MIN having the property that none of the integers up to i are printed by the jth MIN. Likewise, X_m will be in S_j if and only if the jth MIN is the first MIN following INSERT(m) which does not print any integer up to i. We can formalize these ideas in the next lemma.

LEMMA 9. (a) *After the i-th iteration of step* 2 *in Algorithm* 3, $S_j, j \neq \infty$, *contains* X_m (*resp.* M_k) *if and only if j is the smallest number such that the j-th* MIN *follows* INSERT(m) (*resp. the k-th* MIN), *and none of* 1, 2, $\cdots$, *i is printed by the j-th* MIN.

(b) TIME(i) $\leftarrow j$ *by Algorithm* 3 *if and only if i is printed by the j-th* MIN.

Proof. We show (a) and (b) simultaneously by induction on i. The basis, $i = 0$, is true by step 1 of Algorithm 3. Part (b) of the induction holds, since if X_i is in S_j when the ith iteration begins, it must be that the jth MIN follows INSERT(i) but does not cause any integer smaller than i to be printed. However, by hypotheses, any MIN's between INSERT(i) and the jth MIN do print out smaller integers. Thus i is the smallest integer available when the jth MIN is executed.

For part (a) of the induction, we need only observe that the set in which an X_m or M_k belongs does not change at iteration i unless it was in S_j. Then, since the jth MIN prints i, Algorithm 3 correctly merges S_j with the set containing M_j, that is the set of the next available MIN (or S_∞ if no further MIN's are available).

THEOREM 3. *A sequence of* INSERT *and* MIN *instructions on integers up to n can be simulated off-line in* $O(nG(n))$ *time.*

Proof. We use Algorithm 3 to generate a sequence of MERGE and FIND instructions. At most $2n$ MERGE's are needed in step 1 and at most $2n$ FIND's and n MERGE's are needed in step 2, a total of at most $5n$ instructions. We can thus, by the corollary to either Theorem 1 or 2, simulate this sequence on-line in $O(nG(n))$ steps. In doing so, we shall obtain the array TIME(i). The order in which integers are printed by the MIN instructions can be obtained in a straightforward manner from TIME in $O(n)$ steps.

Acknowledgment. We are indebted to M. J. Fischer for bringing to our attention an error in an earlier analysis of the second algorithm which attempted to show that the time bound was linear.

REFERENCES

[1] B. A. GALLER AND M. J. FISCHER, *An improved equivalence algorithm*, Comm. ACM, 7 (1964), pp. 301–303.

[2] M. D. MCILROY, Private communication, Sept. 1971.

[3] R. E. STEARNS AND D. J. ROSENKRANTZ, *Table machine simulation*, 10th SWAT, 1969, pp. 118–128.

[4] R. E. STEARNS AND P. M. LEWIS, *Property grammars and table machines*, Information and Control 14 (1969), pp. 524–549.

[5] D. E. KNUTH, *The Art of Computer Programming*, vol. III, Addison-Wesley, Reading, Mass., 1973.

[6] R. E. TARJAN, *On the efficiency of a good but not linear set union algorithm*, Tech. Rep. 72–148, Dept. of Comp. Sci., Cornell University, Ithaca, N.Y., 1972.

[7] D. E. KNUTH, *The Art of Computer Programming*, vol. II, Addison-Wesley, Reading, Mass., 1969. See also L. Guibas and D. Plaisted, *Some combinatorial research problems with a computer science flavor*, Combinatorial Seminar, Stanford Univ., Stanford, Calif., 1972.

[8] M. J. FISCHER, *Efficiency of equivalence algorithms*, Complexity of Computer Computations, Miller et al., eds., Plenum Press, New York, 1972, pp. 153–167.

[9] M. Paterson, unpublished report, University of Warwick, Coventry, Great Britain.