Science of Computer Programming 2 (1982) 143–152
North-Holland Publishing Company

FINDING REPEATED ELEMENTS*

J. MISRA
Department of Computer Science, University of Texas at Austin, Austin, TX78712, U.S.A.

David GRIES
Department of Computer Science, Cornell University, Ithaca, NY 14853, U.S.A.

Communicated by J. Nievergelt
Received May 1982
Revised November 1982

Abstract. Two algorithms are presented for finding the values that occur more than $n \div k$ times in an array $b[0:n-1]$. The second one requires time proportional to $n * \log(k)$ and extra space proportional to k. A theorem suggests that this algorithm is optimal among algorithms that are based on comparing array elements. Thus, finding the element that occurs more than $n \div 2$ times requires linear time, while determining whether there is a duplicate – the case $k = n$ – requires time proportional to $n * \log n$.

The algorithms may be interesting from a standpoint of programming methodology; each was developed as an extension of the algorithm for the simple case $k = 2$.

1. Introduction

We begin by introducing an algorithm that, given an array $b[0:n-1]$, $1 \leq n$, determines whether there is a majority value – whether any value occurs more than $n \div 2$ times in b. The algorithm works in two passes. First, it finds a single likely candidate v for the majority element; second, it scans b again to count the number of occurrences of v to see whether v occurs more than $n \div 2$ times. The second pass is simple and clearly takes time $O(n)$, and we shall not concern ourselves with it further.

The following algorithm for the first pass, which is clearly linear in n, appears in [1]. We present it in Dijkstra's guarded command notation [2, 3], along with the multiple assignment [3]. A multiple assignment $x_1, \ldots, x_m := e_1, \ldots, e_m$ can be executed by determining the variables x_i being assigned, evaluating the expressions e_i, and then assigning the values to the variables in left-to-right order:

(1) $\quad i, c := 0, 0;$
$\quad\quad$ **do** $i \neq n \rightarrow$
$\quad\quad\quad$ **if** $v = b[i] \qquad \rightarrow c, i := c+2, i+1$

* This work was supported under Air Force grant AFOSR81-0205 A at Austin and NSF grant MCS81-03605 at Cornell.

$\square\ c = i \quad\quad \rightarrow c, i, v := c+2, i+1, b[i]$
$\square\ c \neq i \wedge v \neq b[i] \rightarrow i := i+1$
fi
od
$\{R$: only v may occur more than $n \div 2$ times in $b[0:n-1]\}$

Termination is obvious, using the bound function n–i. But how can one understand that R is true upon termination? The easiest way is to introduce the following invariant:

P: $0 \leq i \leq n$
$\wedge\ v$ occurs at most $c \div 2$ times in $b[0:i-1] \wedge i \leq c \wedge$ even (c)
$\wedge$ each other value occurs at most $i - c \div 2$ times in $b[0:i-1]$

P is true after the initialization $i, c := 0, 0$, no matter what value is initially in v, because $b[0:i-1]$ is empty. And, from the truth of P and the falsity of the loop guard $i \neq n$ upon termination, we conclude that result R holds. The following arguments show that P is indeed an invariant, so that the loop is correct.

Consider the first alternative of the loop body. If guard $v = b[i]$ is true, then v occurs one more time in $b[0:i]$ than it does in $b[0:i-1]$. Hence, increasing i by 1 requires increasing c by 2 so that the upper bound $c \div 2$ on occurrences of v increases by 1. Note that execution of the command leaves the upper bound $i - c \div 2$ of the number of occurrences of each other value the same.

Consider the second alternative. If $c = i$ then i is even and $i - c \div 2 = i \div 2$. Hence, no value occurs more than $i \div 2$ times in $b[0:i-1]$. Therefore, the only value that might possibly (it need not) occur more than $i \div 2$ times in $b[0:i]$ is $b[i]$. From this, it follows that execution of the second guarded command maintains the truth of P.

Finally, it is easily seen that execution of the third command, $i := i+1$, when guard $c \neq i \wedge v \neq b[i]$ is true maintains P. Hence, P is indeed a loop invariant.

This algorithm and its invariant led us to develop two different algorithms for detecting values that could possibly occur more than $n \div k$ times in $b[0:n-1]$, for a given k, $2 \leq k \leq n$. Both algorithms work in two passes: the first pass determines a set t of values that may occur more than $n \div k$ times in b; the second pass scans b to determine how many times each value in t actually occurs. The second pass can be performed in time $O(n \log(|t|))$, and we are interested only in describing the first pass.

2. The first algorithm

We want to generalize the above problem and algorithm. Given k and n, $2 \leq k \leq n$, and array $b[0:n-1]$, we want to find values that may occur more than $n \div k$ times in b. For the case $k = 2$, we were able to identify a single possible value; for the more general case, where $2 \leq k \leq n$, up to $k-1$ distinct values may occur more than $n \div k$ times in b. The simplest extension of R for the case $k=2$ is the following. Execution is to store in a set variable t a set of pairs (v, c) such that

$$\begin{aligned} R:\ & (\forall v, c: (v, c) \in t: v \text{ occurs at most } c \div k \text{ times in } b[0:n-1] \\ & \qquad\qquad\qquad \wedge c > n \wedge k \text{ divides } c) \\ & \wedge \text{ each other value occurs at most } n \div k \text{ times in } b[0:n-1] \end{aligned}$$

To develop the algorithm, we choose an invariant P that weakens R in a useful manner, using the solution for the case $k=2$ for insight:

$$\begin{aligned} P:\ & 0 \leqslant i \leqslant n \\ & \wedge (\forall v, c: (v, c) \in t: v \text{ occurs at most } c \div k \text{ times in } b[0:i-1] \\ & \qquad\qquad\qquad \wedge c > i \wedge k \text{ divides } c) \\ & \wedge \text{ any value not the first component of a pair in } t \\ & \quad \text{occurs at most } s \div k \text{ times in } b[0:i-1] \\ & \wedge 0 \leqslant s \leqslant i \wedge k \text{ divides } s \end{aligned}$$

P was developed after several different trials. The part concerning set t was fairly easy. The difficulty was in discovering a suitable upper bound $s \div k$ on the number of occurrences of other values. A straightforward extension of the case $k=2$ gave $i-(\sum v, c: (v, c) \in t: c)$ for this upper bound; this at first seemed reasonable, since each distinct value v in t could occur up to $c \div k$ times. However, adding a new pair (v, c) to t would cause this upper bound to decrease far too much. Variable s was introduced simply in the hope that a better upper bound could be computed at each iteration, and trial and error led to its definition as given in P. Algorithm (2) was developed hand-in-hand with P:

```
(2)   i, s, t := 0, 0, { };
      do i ≠ n →
         Let j be the index of a pair v_j, c_j in t satisfying v_j = b[i]
            – if no such pair exists let j = 0;
         if j = 0 ∧ s + k ≤ i + 1  → i, s := i + 1, s + k
         □ j = 0 ∧ s + k > i + 1  → i, t := i + 1, t ∪ {(b[i], s + k)}
         □ j ≠ 0                  → i, c_j := i + 1, c_j + k
         fi;
         Delete all pairs (v_j, c_j) from t for which
            c_j = i and, if any are deleted, set s to i
      od {R}
```

It is clear that the initialization establishes P, that the algorithm terminates, and that upon termination the result holds (if P is true). It remains to show the invariance of P under execution of the loop body.

Consider the first alternative of the alternative command. Condition $j=0$ means that $b[i]$ is not the first component of a pair in c. Hence, there is no need to change the counts c_j of components in t when i is increased by 1. However, s must be decreased by k so that $s \div k$ remains an upper bound on the number of occurrences of values not in t. The conjunct $s+k \leq i+1$ ensures that execution maintains $s \leq i$.

Consider the second alternative. Again, $j=0$ means there is no need to change the counts c_j of components in t. However, s cannot be changed as i is increased because $s \leq i$ would be violated. In this case, the component $b[i]$ might occur $(s+k) \div k$ times in $b[0:i]$, and so $b[i]$ must be placed in t along with the maximum number of times it might occur.

In the case of the third alternative, $b[i]$ is the first component of a pair (v_j, c_j) in t. Hence, v_j occurs one more time in $b[0:i]$ than it does in $b[0:i-1]$, and c_j is increased accordingly.

The third statement of the loop body deletes certain members from set t so that pairs (v_j, c_j) of t satisfy $c_j > i$. In this case, however, the upper bound on the number of occurrences of values not in t must be changed. Hence the change in s.

This ends the discussion of the invariance of P.

The execution speed of algorithm (2) depends on the size and implementation of set t. Unfortunately, we have been unable to determine a useful upper bound on the size of t. We conjecture that it is a function of k, and not i. We also conjecture that t may become its largest if b has roughly the following form: it ends with k distinct values, preceded by $k \div 2$ values, each occurring twice, preceded by $k \div 3$ values, each occurring thrice, etc. Hence, $|t|$ could possibly become as large as $O(k * \log(k))$.

3. The second algorithm

The second algorithm rests on some extremely simple theory. Consider a bag – i.e. a collection of elements, with duplicates possible[1] – and consider the operation of deleting k *distinct* elements from it. This operation may be performed several times. A *k-reduced bag for* bag B is a bag derived from B by repeating this operation until no longer possible. Note that the k-reduced bag is not unique. For example, for bag {1,1,2,3,3}, one can arrive at three different 2-reduced bags using 5 different deletion sequences. We show these sequences below; in each bag the elements to be deleted next are barred.

$$
\begin{array}{llll}
\{\bar{1}, 1, \bar{2}, 3, 3\}, & \text{then } \{\bar{1}, \bar{3}, 3\}, & \text{then } \{3\}, & \\
\{\bar{1}, 1, 2, \bar{3}, 3\}, & \text{then } \{\bar{1}, \bar{2}, 3\}, & \text{then } \{3\}, & \\
\{\bar{1}, 1, 2, \bar{3}, 3\}, & \text{then } \{\bar{1}, 2, \bar{3}\}, & \text{then } \{2\}, & \\
\{\bar{1}, 1, 2, \bar{3}, 3\}, & \text{then } \{1, \bar{2}, \bar{3}\}, & \text{then } \{1\}, & \text{and} \\
\{1, 1, \bar{2}, \bar{3}, 3\}, & \text{then } \{\bar{1}, 1, \bar{3}\}, & \text{then } \{1\} &
\end{array}
$$

[1] We use set notation for bags, e.g. $b \cup \{v\}$ denotes the bag consisting of the elements of bag b together with the element v.

Suppose bag B has N elements. The operation of deleting k distinct elements can be performed at most $N \div k$ times, for after that the set will contain fewer than k elements. Only values that occur in a k-reduced bag for B can occur more than $N \div k$ times in B; the other values have been deleted at most $N \div k$ times each and don't appear any more, so they could have appeared at most $N \div k$ times in B. This proves the following theorem:

Theorem 1. *Let bag B contain N items. The only values that may occur more than $N \div k$ times in B are the values in a k-reduced bag for B.*

Considering $b[0:n-1]$ to be a bag, we use Theorem 1 to develop an algorithm as follows. The result assertion is

$$R: t \text{ is a } k\text{-reduced bag for } b[0:n-1]$$

A loop invariant is found by replacing constant n by a variable i and introducing a second variable d for efficiency purposes:

$$\begin{aligned} P:\ & 0 \leqslant i \leqslant n \\ & \wedge t \text{ is a } k\text{-reduced bag for } b[0:i-1] \\ & \wedge d \text{ is the number of distinct elements of } t \end{aligned}$$

The algorithm is then written as follows: it should be compared to algorithm (2), and it should need no further explanation:

```
(3)     i, d, t := 0, 0, { };
        do i ≠ n →
            if b[i] ∉ t → t, d := t ∪ {b[i]}, d + 1;
                          if d = k → Delete k distinct values
                                     from t and update d
                          □ d < k → skip
                          fi
            □ b[i] ∈ t → t := t ∪ {b[i]}
            fi;
            i := i + 1
        od
```

In algorithm (2), we were not able to determine the size of set t. In algorithm (3), t has at most k distinct elements, and it has at most $k-1$ distinct elements before and after each iteration. We will show later how to implement t so that algorithm (3) runs in time $O(n * \log(k))$.

Both algorithms use a bag t of elements. It is only in the definition of t that they differ. Both were developed by trying to extend the algorithm for the case $k=2$ given in the Introduction.

4. Implementing bag t of algorithm (3)

Bag t of algorithm (3) has at most n elements and d distinct elements, $d \leq k$. The operations performed on t and d are:

1. $t := \{\ \}$. Performed once.
2. Search t for an element $b[i]$. Performed n times.
3. Insert an element into t. Performed at most n times.
4. Delete k distinct elements from t and update d. Performed at most $n \div k$ times and only when t has exactly k distinct elements.

We implement t using an AVL tree T with d nodes; each node is a pair (v_j, c_j), where v_j is one of the distinct elements of t and c_j is the number of times v_j occurs in t. This requires $O(k)$ space.

Operation 1 calls for initializing T to an empty tree – a constant-time operation. Operation 2, searching for an element in t, requires time $O(\log(k))$, since T has at most k nodes. In total, operation 2 contributes time $O(n * \log(k))$. Operation 3, inserting an element into t, calls for finding a value in a node j of T and adding 1 to c_j, or, if the element is not in t, adding it with count 1. In any case, the time is no worse than $O(\log(k))$, and operation 3 contributes time $O(n * \log(k))$.

Operation 4, deleting k distinct elements from t when t has exactly k distinct elements, calls for subtracting 1 from count c_j for each node j of *AVL* tree T and, if c_j becomes 0, deleting node j from T. This takes time at most $O(k * \log(k))$. Since operation 4 is performed at most $n \div k$ times, the total time spent in it is $O((n \div k) * k * \log(k))$, which is $O(n * \log(k))$.

Hence, the total time spent in operations dealing with bag t is $O(n * \log(k))$.

5. On the complexity of detecting repeated elements

We introduce a *decision-tree algorithm* (see e.g. [4]) for the problem of determining whether any value occurs more than $n \div k$ times in $b[0:n-1]$. We show that the algorithm takes time $O(n * \log(k))$ (all times given are worst-case times). All algorithms for the problem that are based on comparing elements of b can be thought of as decision-tree algorithms, which leads to the suggestion that algorithm (3) has optimal execution time.

A decision-tree algorithm for the problem is a *decision tree D* together with algorithm (4), given below; the decision tree D is a finite tree with the following characteristics:

1. Every nonterminal node of D has a label (i, j), where $0 \leq i, j < n$. i and j are used to refer to elements $b[i]$ and $b[j]$.
2. Every nonterminal node has three branches, with labels $<$, $=$ and $>$.
3. Every terminal node has a label YES or NO.

4. Given $b[0:n-1]$ and k, execution of algorithm (4) begins with x as the root of the tree and terminates with x being a terminal node; the label of x is YES if some value occurs more than $n \div k$ times and NO otherwise.

(4) $x :=$ root of D;
do x is a nonterminal node with label $(i, j) \rightarrow$
  $b[i]$ **op** $b[j]$ must hold, where **op** is either $<$, $=$, or $>$. Let y be the son of node x that is reached via a branch labelled **op**. Follow the branch from x to this son y, i.e. execute $x := y$
od

Execution of algorithm (4) begins at the root of the decision tree and proceeds along some path to a terminal node, and the label at the terminal node indicates whether a value occurs more than $n \div k$ times in b. All algorithms for solving the problem that are based on comparing elements of b can be thought of as decision-tree algorithms, for they proceed by comparing array elements in some order that can be given by a decision tree. Further, decision trees enjoy the advantage that the next action following a comparison can depend on *all* previous comparisons, without incurring the attendant cost.

As defined, tree D allows the comparisons $<$, $>$ and $=$. The same results follow if one allows instead only binary trees with labels $=$ and $\neq$.

We now proceed as follows. Let $r = n \div k$. Hence, $n \div (r+1) < k \leq n \div r$. We introduce a set of lists, called r-lists, each with n elements. Each r-list contains a list of values that could appear in array $b[0:n-1]$ upon which our algorithms can be run. We show (Lemma 1) that there are at least $(k/e)^n$ different r-lists.[2] Next, we show (Lemma 3) that execution of the decision-tree algorithm (with a given decision tree) terminates at a distinct terminal node for each assignment of an r-list to b. Hence, a decision tree has at least as many terminal nodes as there are r-lists, so that the longest path length in a decision tree is at least

$$O(\log((k/e)^n) = O(n * \log(k) - n * \log(e))$$
$$= O(n * \log(k)).$$

This proves

Theorem 2. *For a given k, $2 \leq k \leq n$, any algorithm based on comparing array elements requires at least $O(n * \log(k))$ comparisons to determine whether some value(s) occurs more than $n \div k$ times in $b[0:n-1]$.*

Definition 1. An *r-list* is a list of n elements in which each of the values $0, 1, \ldots, n \div r - 1$ occurs r times and the value $n \div r$ occurs n **mod** r times.

[2] e is the base of natural logarithms.

Lemma 1. *There are at least $(k/e)^n$ different r-lists.*

Proof. An r-list can be constructed as follows. Choose any r indices out of n and store the value 0 there; choose any r indices out of the remaining $n-r$ possible indices and store the value 1 there; ... ; after $r*(n \div r)$ values have been stored, store the value $n \div r$ in the remaining $n \textbf{ mod } r$ positions. The number of different r-lists corresponds to the number of different possible choices in this procedure, which is

$$\prod_{i=0}^{n\div r-1} \binom{n-i*r}{r} = \frac{n!}{r!^{n\div r} * (n \textbf{ mod } r)!}.$$

Let $x = n \textbf{ mod } r$. Then $n \div r = (n-x)/r$. So

$$\begin{aligned} r!^{n\div r} * (n \textbf{ mod } r)! &= r!^{(n-x)/r} * x! \\ &= (r!^{n-x} * x!^r)^{1/r} \\ &\leqslant (r!^{n-x} * r!^x)^{1/r} \quad \text{(Lemma 2)} \\ &= r!^{n/r} \\ &\leqslant (r^r)^{n/r} \\ &= r^n. \end{aligned}$$

Hence, the number of different r-lists is bounded below by

$$\begin{aligned} \frac{n!}{r!^{n\div r} * (n \textbf{ mod } r)!} &\geqslant \frac{n!}{r^n} \\ &\geqslant \frac{(n/e)^n}{r^n} \quad \text{(using Stirling's formula)} \\ &\geqslant (k/e)^n. \quad \Box \end{aligned}$$

Lemma 2. *If $r > p$ then $r!^p \geqslant p!^r$.*

Proof. Let $r = p+q$. Then

$$\begin{aligned} r! &= p! * (p+1) * (p+2) * \cdots * (p+q) \\ &\geqslant p! * p^q. \end{aligned}$$

Therefore,
$$\begin{aligned} r!^p &\geqslant (p! * p^q)^p \\ &= p!^p * (p^p)^q \\ &\geqslant p!^p * p!^q \\ &= p!^r. \quad \Box \end{aligned}$$

Lemma 3. *Consider a fixed decision tree. Execution of the decision-tree algorithm for different r-lists terminates at different nodes.*

Proof. No value occurs more than r times in an r-list; hence, execution of the decision-tree algorithm with an r-list terminates at a node labelled NO. Next, define a new list $L = L1 * L2$ from two different r-lists $L1$ and $L2$ as follows:

$$L[j] = \min(L1[j], L2[j]) \quad \text{for } 0 \leq j < n.$$

It is obvious that L satisfies the following, for any indices i and j:

$$\begin{aligned} &L1[i] < L1[j] \wedge L2[i] < L2[j] \Rightarrow L[i] < L[j], \\ &L1[i] = L1[j] \wedge L2[i] = L2[j] \Rightarrow L[i] = L[j], \\ &L1[i] > L1[j] \wedge L2[i] > L2[j] \Rightarrow L[i] > L[j]. \end{aligned} \tag{1}$$

Further, we show in Lemma 4 that if $L1$ and $L2$ are different then some value occurs more than r times in L, so that execution of the decision-tree algorithm with input L terminates on a node with label YES.

Now assume the contrary of the lemma: execution of the decision-tree algorithm terminates at the same node x for both $L1$ and $L2$. Hence, the executions for $L1$ and $L2$ follow the same path in the decision tree. By property (1), execution of the decision-tree algorithm on list L must follow that same path, and hence must end in a terminal node with label NO. Since some value occurs more than r times in L, this is a contradiction. Hence, the assumption that $L1$ and $L2$ land on the same node must be false, and the lemma is proved. □

Lemma 4. *If r-lists $L1$ and $L2$ are different, then a value occurs more than r times in $L = L1 * L2$.*

Proof. Let $s1(v)$ and $s2(v)$ be the set of indices (positions) in $L1$ and $L2$, respectively, where a value that is at most v appears:

$$s1(v) = \{j \mid L1[j] \leq v\}, \qquad s2(v) = \{j \mid L2[j] \leq v\}$$

Since $L1 \neq L2$, there is some v satisfying $s1(v) \neq s2(v)$. For $v \geq n \div r$, $s1(v) = s2(v) = \{1, 2, \ldots, n\}$. Hence, for some w, $w < n \div r$, $s1(w) \neq s2(w)$ holds.

Suppose $i \in s1(w) \cup s2(w)$. Then either $L1[i] \leq w$ or $L2[i] \leq w$, so that $L[i] = \min(L1[i], L2[i]) \leq w$. From the definition of r-list and the fact that $w < n \div r$, $|s1(w)| = |s2(w)| = (w+1) * r$ holds. Since $s1(w) \neq s2(w)$, $|s1(w)| \cup |s2(w)| > (w+1) * r$. By the pigeon-hole principle, some value that is at most w must appear more than r times in L. □

6. Finding whether values occur more than r times

Consider finding values that occur more than r times in $b[0:n-1]$, where $1 \leq r < n$. This problem can be solved in terms of the original problem by taking k as the smallest integer satisfying $n \div k \leq r$. Thus, if $n = 10$ and $r = 4$, take $k = 3$ and find a set of values that may occur more than 3, instead of 4, times. Then count the number of occurrences in b of each of these values to solve the original problem.

If n is not known – e.g. b is implemented as a linked list – then one can first search b to determine its length. This takes linear time, so that the algorithm remains $O(n * \log(k))$.

Acknowledgment

We wish to thank the referees for quickly providing many constructive criticisms.

References

[1] B. Boyer and J. Moore, MJRTY: A fast majority-vote algorithm, submitted for publication.
[2] E.W. Dijkstra, *A Discipline of Programming* (Prentice-Hall, Englewood Cliffs, NJ, 1976).
[3] D. Gries, *The Science of Programming* (Springer, New York, 1981).
[4] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Menlo Park, 1974).