FINDING REPEATED ELEMENTS*

J. Misra+
David Gries++

TR 82-505
July 1982

Department of Computer Science
Cornell University
Ithaca, New York 14853

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

+University of Texas at Austin

++Cornell University

Finding Repeated Elements

J. Misra
University of Texas at Austin+

David Gries
Cornell University+

July 1982

Keywords Majority detection, repeated elements

Abstract

Two algorithms are presented for finding the values that occur more than n÷k times in array b[0:n-1]. The second algorithm requires time O(n*log(k)) and extra space O(k). We prove that O(n*log(k)) is a lower bound on the time required for any algorithm based on comparing array elements, so that the second algorithm is optimal. As special cases, determining whether a value occurs more than n÷2 times requires linear time, but determining whether there are duplicates —the case k = n — requires time O(n*log(n)).

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

1. Introduction

Given is an array b[0:n-1], where n > 0, and an integer k, 0 < k ≤ n. We consider the problem of finding the values that occur more than n÷k times in b. The more general problem of finding values that occur more than r times, for 0 < r < n, can be solved in terms of the original problem by taking k as the smallest integer satisfying n÷k ≤ r. Thus, if n = 10 and r = 4, use k = 3; find the values that occur more than 3, instead of 4, times; then count how many times each actually occurs in b.

We begin by considering the case k = 2. The following algorithm identifies a value v: upon termination, no value except v occurs more than n÷2 times, but the occurrences of v in b must be counted to determine whether v occurs more than n÷2 times. The algorithm, which is linear in n, appears in [1].

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

```
(1)   i, c:= 0, 0;
      do i ≠ n →
         if c≠0 ∧ v = b[i] → c, i:= c+1, i+1
          [] c≠0 ∧ v ≠ b[i] → c, i:= c-1, i+1
          [] c=0           → c, i, v:= c+1, i+1, b[i]
         fi
      od
      {only  v  may occur more than  n÷2  times in  b[0:n-1]}
```

The algorithm may be understood most easily using the following loop invariant.

P: $0 \le i \le n \wedge 0 \le c \wedge$ even(i+c) $\wedge$
   v occurs at most $(i+c)\div 2$ times in b[0:i-1] $\wedge$
   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. It is easy to see that the first two alternatives of the alternative command of the loop body maintain the truth of P; each increases one of $(i+c)\div 2$ and $(i-c)\div 2$ and leaves the other unchanged, depending on whether v = b[i].

Now consider the third alternative. Suppose the guard is true: c = 0. Then $(i+c)\div 2 = (i-c)\div 2 = i$. Further, i is even and no value occurs more than $i\div 2$ times in b[0:i-1]. Therefore, the only value that can occur more times in b[0:i] is b[i]. From this, it follows that execution of the last guarded command maintains the truth of P.

Upon termination, the truth of P and falsity of the loop guard imply the desired result. Termination is obvious, using the bound function n-i.

This algorithm and its invariant led us to develop two different algorithms for the case $n\div k$ instead of $n\div 2$. Both algorithms determine a set t of values that may occur more than $n\div k$ times in b. To determine whether they do occur more times, one must actually count the number of occurrences in b of each one. This counting can be performed in time $O(n*\log(|t|))$.

2. The First Algorithm

Given k and n, $0 < k \le n$, and array b[0:n-1], we want to find the values that may occur more than $n\div k$ times in b. We formulate the result assertion of the algorithm as follows. Execution is to store in a set variable t a set of pairs (v, c) such that

R: $(\forall$ v,c: $(v, c) \in t$: v occurs at most $c\div k$ times in b[0:n-1] $\wedge$
   $c > n \wedge k$ divides c) $\wedge$
   no other value occurs more than $n\div k$ times in b

To develop the algorithm, we choose an invariant P that weakens result assertion R in a useful manner, using solution (1) for insight. P was developed after several different trials. It required the replacement of constant n by a variable i and the introduction of a fresh integer variable s.

P: $0 \le i \le n \wedge$
(∀ v,c: (v, c) ∈ t: v occurs at most c÷k times in b[0:i-1] ∧
c > i ∧ k divides c) ∧
$s \ge 0 \wedge$ k divides i-s ∧
any value not the first component of a pair in t
occurs at most (i-s)÷k times in b[0:i-1]

A discussion follows the algorithm:

(2) i, s, t := 0, 0, {};
do $i \neq n \rightarrow$
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 $j = 0 \wedge s \ge k-1 \rightarrow$ i, s := i+1, s-k+1
[] $j = 0 \wedge s < k-1 \rightarrow$ i, s, t := i+1, s+1, t ∪ {(b[i], i-s+k)}
[] $j \neq 0 \rightarrow$ i, s, c_j := i+1, s+1, c_j+k
fi;
Delete all pairs (v_j, c_j) from t for which
$c_j = i$; if any are deleted, set s to 0
od

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 two alternatives of the alternative command; $j = 0$ means that b[i] is not the first component of a pair in t. 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-1 so that the expression (i-s)÷k is increased by 1. The latter may be done only if s remains ≥0. If $s < k-1$, then b[i] might occur i÷k + 1 times in b[0:i], so b[i] must be placed in t, along with the maximum number of times it might occur. This is the purpose of the second alternative.

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. As i is increased, s is increased to keep the value of (i-s)÷k the same.

The third statement of the loop body deletes certain members from set t, so that pairs (v_j, c_j) of t satisfy $c_j > i$.

The execution speed of this algorithm 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 of i. We also conjecture that t becomes its largest if b has roughly the following form: it ends with k different values, preceded by k÷2 different values, each ocurring twice, preceded by k÷3 different values, each occurring thrice, etc. Hence |t| might 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[+]— 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:

{1,1,2,3,3}, then {1,3,3}, then {3},
{1,1,2,3,3}, then {1,2,3}, then {1},
{1,1,2,3,3}, then {1,2,3}, then {2},
{1,1,2,3,3}, then {1,2,3}, then {3}, and
{1,1,2,3,3}, then {1,1,3}, then {1}

Suppose bag B has N elements. The operation of deleting k distinct elements can be performed at most N÷k times, for after that B can contain at most N **mod** k elements, which is <k. Hence, the values that don't occur in a k-reduced bag for B can not occur more than N÷k times in B, —they have been deleted at most N÷k times and no longer appear. This leads directly to a simple theorem:

(3) Theorem. The only values that may occur more than N÷k times in bag B of size N are the elements in a k-reduced bag for B. □

Considering b[0:n-1] to be a bag, we use theorem (3) to develop an algorithm as follows. The result assertion is

R: t is a k-reduced bag for b[0:n-1]

so that upon termination t will contain at most k-1 distinct values that may occur more than N÷k times in b. The invariant of a loop is found by replacing constant n by a variable i and introducing a second variable d

+We use set notation for bags, e.g. b ∪ {v} denotes the bag consisting of the elements of bag b together with the element v.

for efficency purposes:

P: $0 \le i \le n \wedge$
t is a k-reduced bag for b[0:i-1] $\wedge$
d is the number of distinct elements of t

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

```
(4)  i, d, t := 0, 0, {};
     do i ≠ n →
         if b[i] ∉ t ∧ d < k-1 → t, d := t ∪ {b[i]}, d+1
          [] b[i] ∉ t ∧ d ≥ k-1 → t, d := t ∪ {b[i]}, d+1;
                                  Delete k distinct elements
                                  from t and update d
          [] b[i] ∈ t           → t := t ∪ {b[i]}
         fi
     od
```

For algorithm (2), we were not able to determine the size of set t. In algorithm (4), t has at most k distinct elements, and it has at most k-1 distinct elements before and after each iteration. We will subsequently show how to implement t so that, in total, the operations performed on it take no more than time O(n*log(k)).

Note the similarity of the algorithms; essentially, both use a bag t of elements and both have the same structure. 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 the Bag t of Algorithm (4)

Bag t of algorithm (4) has at most n elements and d distinct elements, $d \le k$. The operations to be performed on t and d are:

1. t := {}. Performed once.
2. Search t for a element v. 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÷k times and only when t has exactly k distinct elements.

We implement bag 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 the value in a node j of T and adding 1 to c_j, or, if the element is not in t, adding it to T 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 it has exactly k 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÷k times, the total time spent in operation 4 is O((n÷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 begin by introducing a class of algorithms, called decision-tree algorithms, for determining whether any value occurs more than n÷k times in b[0:n-1]. Each decision-tree algorithm consists of algorithm (5) (given below), together with a decision tree, which controls its execution. A decision tree D is a finite tree with the following characteristics:

1. Every nonterminal node of D has a label (i, j), where $0 \le i < n, 0 \le j < n$. The label is 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 an label YES or NO.
4. Given b[0:n-1], execution of algorithm (5) begins with c being the root of the tree and terminates with c being a terminal node; the label of c is YES if some value in b occurs more than n÷k times and NO otherwise.

```
(5) c:= root of D;
    do c is a nonterminal node with label (i, j) →
        Suppose b[i] op b[j], where op is one of the
        operators <, = , >, and let x be the son of
        node c that is labeled op. Execute c:= x
    od
```

Execution of algorithm (5) 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 some value occurs more than n÷k times in b. The path taken depends only on comparisons of array elements. All algorithms for solving the problem that are based on comparing elements of b can be thought of as decision-tree algorithms; further, decision trees enjoy the advantage

that the next action following a comparison can depend on all previous comparisons, without incurring the attendant cost.

We 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. We show (lemma (8)) that there are

$$\frac{n!}{r!^{n \div r} * (n \bmod r)}$$

different r-lists. Next, we show (lemma (9)) that execution of a 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

$$\begin{aligned} & O(\log(n! \;/\; (r!^{n \div r} * (n \bmod r)))) \\ &= O(n*\log(n) - (n \div r)*r*\log(r) - \log(n \bmod r)) \\ &= O(n*\log(n) - n*\log(r)) \\ &\geq O(n*\log(n \div r)) \\ &\geq O(n*\log(k)) \end{aligned}$$

This leads directly to

(6) Theorem. 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 $b[0:n-1]$. □

(7) Definition. 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 $n \bmod r$ times. □

(8) Lemma. There are $n!/(r!^{n \div r} * (n \bmod r))$ 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 $n \bmod r$ positions. The number of different r-lists corresponds to the number of different possible choices in the procedure given above, which is

$$\prod_{i=0}^{n \div r - 1} \binom{n - i*r}{r}$$

which simplifies to the expression given in the lemma. □

(9) Lemma. Consider a fixed decision tree. Execution of a 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 a decision-tree algorithm with an r-list terminates at a node labelled NO. Define a new list $L = L1 \oplus L2$ from different r-lists L1 and L2 as follows:

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

Obviously, L satisfies the following for any indices i and j:

(10) $L1[i] < L1[j] \wedge L2[i] < L2[j] \Rightarrow L[i] < L[j]$
$L1[i] = L1[j] \wedge L2[i] = L2[j] \Rightarrow L[i] = L[j]$
$L1[i] > L1[j] \wedge L2[i] > L2[j] \Rightarrow L[i] > L[j]$

Further, we show in lemma (11) that if L1 and L2 are different then some value in L occurs more that r times, 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 a decision-tree algorithm terminates at the same node x for both L1 and L2. Hence, the executions follow the same path in the decision tree. By property (10), 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. □

(11) Lemma. If r-lists L1 and L2 are different, then some value occurs more than r times in $L = L1 \oplus L2$.

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:

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

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

Suppose $i \in s1(w) \cup s2(w)$. Then either $L1[i] \le w$ or $L2[i] \le w$, so that $L[i] = \min(L1[i], L2[i]) \le w$. From the definition of r-list and the fact that $w < n*r$, $|s1(w)| = |s2(w)| = (w+1)*r$ 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. □

References

[1] Boyer, B. and J. Moore. MJRTY: a fast majority-vote algorithm. Submitted for publication.