A Proof of a Heavy Hitters Algorithm

Jayadev Misra

February 7, 2018

1 Introduction

This note is meant to teach program development and formal proof in an introductory computer science course. Given a bag of items consider finding all items that occur more often than a given fraction of the bag size, $1/k$ for some integer k, $k \geq 2$. This is known as the *heavy hitters* problem. An excellent introduction to the problem, along with the more recent advances such as ϵ-approximate heavy hitters, appear in Roughgarden and Valiant [3]. The first algorithm for this problem, due to Misra and Gries [2], is used as the motivation for the note.

There are two special cases of the problem that are worth noting. For a bag of size n, setting k to n yields all elements that have duplicates. The heavy-hitters algorithm is not particularly efficient at solving the duplicates problem; the algorithm works well only for small values of k. In particular, setting k to 2 finds the *majority* element of a bag, if one exists, efficiently.

In the following section we treat the problem of finding the majority element. The original algorithm for this special case is due to Boyer and Moore [1]. They were interested in a mechanical proof of the algorithm that was coded in Fortran, making its analysis particularly hard. Our description of the majority algorithm avoids using any specific programming language though there is enough detail that a student can easily code it in a common imperative language.

In Section 3 we treat the problem for general k. The general algorithm follows the same pattern as the algorithm for the majority problem.

2 The Boyer-Moore Majority Finding Algorithm

Given is a finite bag B of items. An *item* may occur multiple times in a bag. A *majority* item is one that occurs more often in the bag than all other items combined. Call occurrence of an item in the bag an *element* of the bag.

It is required to find the majority item of B, if one exists, or indicate that there is no such item. The Boyer-Moore algorithm solves this problem by making two passes over the elements of B where each pass runs in linear time. The first pass suggests a *candidate* item for majority, or that no such candidate exists (so there is no majority item). The second pass determines if the suggested candidate is indeed a majority item. The second pass is straightforward, by counting the number of occurrences of the candidate item in B, so we consider only the first pass in this note. The algorithm is based on the next proposition.

Proposition 1 Removing two distinct elements from a bag does not alter the majority item, if the bag has one.

Proof: Suppose two distinct elements from A are removed to get bag A'. The number of occurrences of a majority item in A is more than $|A|/2$. It is reduced by at most one in A' since the removed elements are distinct. So, the number of its occurrences in A' is more than $|A|/2 - 1 = (|A| - 2)/2 = |A'|/2$. So, it is a majority item of A'. □

Proposition 1 can be written formally as follows. Overload the usual set notation to apply to bags. Let $maj(A)$ be the set of majority items of A. That is, if A has a majority item —and there can be at most one— $maj(A)$ is a singleton set, otherwise it is the empty set. Given that x and y are two distinct elements of A, the proposition says that: $maj(A) \subseteq maj(A - \{x, y\})$. Observe that $maj(A) = maj(A - \{x, y\})$ does not hold: consider $A = \{1, 2, 3\}$ from which two distinct elements, 1 and 2, are removed; then, $maj(\{1, 2, 3\}) \neq maj(\{3\})$.

2.1 An Abstract Algorithm

An abstract algorithm, based on this proposition, is as follows: continue removing pairs of distinct elements from bag B as long as it is possible. At termination either the bag is empty —so, there is no majority item— or it has one or more instances of a single item that is a candidate for the majority.

A simple implementation of this idea is as follows: initially, remove any item from B and hold it in a separate bag, c. In each subsequent step, remove an item from B; if it differs from an item of c then cast out both elements, and if it is identical add the item to c (formally, any element matches every element of an empty bag). Steps are repeated until B becomes empty. From the construction, all the elements in bag c are identical, and from Proposition 1, the majority item, if any, belongs to c at the termination.

Below, the reduced bag B is called b during the run of the program. Symbol ϕ is the empty bag.

```
initially  $b, c := B, \phi$
while  $b \neq \phi$ do
   pick any $x$ from $b$  $\{x \in b\}$
   if  there exists $y$ in $c$ such that $x \neq y$
      then  $b, c := b - \{x\}, c - \{y\}$
      else  $\{$for all $y$ in $c$, $x = y\}$ $b, c := b - \{x\}, c \cup \{x\}$
enddo
```

2.2 Correctness

We claim that the algorithm terminates, and on termination either c is empty or it consists of a single item, m, perhaps multiple times. If B has a majority item then it is m. The converse does not hold, i.e., if c includes m then m is not necessarily the majority item because B may have none. So, a second pass determines if indeed m is the majority item.

Proposition 2 Invariant: All elements of c are identical.

Proof: The invariant holds initially because c is empty. In each step either an element is removed from c, or an element that equals every element of c, is added to c. Therefore, a step preserves the invariant. □

The following invariant encodes Proposition 1.

Proposition 3 Invariant: $maj(B) \subseteq maj(b \cup c)$.

Proof: The invariant holds initially because $b \cup c = B$. Next, we show that each step preserves the invariant. The step corresponding to the "else" clause does not alter $b \cup c$. For the step corresponding to the "then" part, the required annotation is shown below.

$$\begin{array}{l} \{maj(B) \subseteq maj(b \cup c) \wedge x \in b \wedge y \in c \wedge x \neq y\} \\ \qquad b, c := b - \{x\}, c - \{y\} \\ \qquad \{maj(B) \subseteq maj(b \cup c)\} \end{array}$$

Using the axiom of assignment, we need to show,

$$\begin{array}{ll} & maj(B) \subseteq maj(b \cup c) \wedge x \in b \wedge y \in c \wedge x \neq y \\ \Rightarrow & maj(B) \subseteq maj((b - \{x\}) \cup (c - \{y\})). \end{array}$$

The proof is as follows.

$$\begin{array}{ll} & maj(B) \subseteq maj(b \cup c) \wedge x \in b \wedge y \in c \wedge x \neq y \\ \Rightarrow & \{\text{use Proposition 1: } maj(b \cup c) \subseteq maj(b \cup c - \{x, y\}\ \} \\ & maj(B) \subseteq maj(b \cup c - \{x, y\}) \wedge x \in b \wedge y \in c \wedge x \neq y \\ \Rightarrow & \{x \neq y; \text{ so } (b \cup c) - \{x, y\} = (b - \{x\}) \cup (c - \{y\})\} \\ & maj(B) \subseteq maj((b - \{x\}) \cup (c - \{y\})) \end{array}$$

Proposition 4 The algorithm terminates.

Proof: Each iteration of the loop reduces the size of b by 1; so, the number of iterations is exactly $|B|$. □

Proposition 5 Suppose B has a majority item m. Then, at termination c consists of one or more copies of m.

Proof: From the invariant $maj(B) \subseteq maj(b \cup c)$, and at termination $b = \phi$. So, $maj(B) \subseteq maj(c)$. If $maj(B) = \{m\}$ then $\{m\} \subseteq maj(c)$. And, from Proposition 2, all elements of c are identical. So, c consists of one or more copies of m. □

2.3 Algorithm Optimization

Bag c consists of zero or more copies of a single item at any point during the algorithm. Therefore, it may be encoded by two values i and n, item i occurs n times in c (if $n = 0$, i is irrelevant). The algorithm can then be written as follows.

```
initially  $b, i, n := B, \_ , 0$
while  $b \neq \phi$ do
   pick any $x$ from $b$  $\{x \in b\}$
   if  $n > 0 \wedge x \neq i$ $\{c$ is non-empty, so $i$ has a value$\}$
      then  $b, n := b - \{x\}, n - 1$
      else  $\{n = 0 \vee (n > 0 \wedge x = i)\}$ $b, i, n := b - \{x\}, x, n + 1$
enddo
```

Note: i is assigned to x in the last step of the program because preceding this step if $n = 0$ then i may have an irrelevant value.

3 The Heavy Hitters Problem

It is required to find every item that occurs more than $|B|/k$ times in bag B for a given k, $k \geq 2$; these items are the, so called, *heavy hitters*.

The special case of $k = 2$, the majority problem, has been treated in the previous section. The general case closely follows the development of the majority problem. The fundamental observation for the majority problem, Proposition 1, is generalized in Proposition 6. Let $hh(A)$ be the set of heavy hitters in bag A, where the parameter k is implicit.

Proposition 6 Remove k distinct elements from A to get bag A'. Then all heavy hitters of A are heavy hitters of A', i.e., $hh(A) \subseteq hh(A')$.

Proof: The number of occurrences of any heavy hitter in A is more than $|A|/k$. It is reduced by at most one in A' since removed elements are distinct. Then, the number of its occurrences in A' is more than $|A|/k - 1 = (|A| - k)/k = |A'|/k$. So, it is a heavy hitter of A'. □

An Abstract Algorithm The abstract algorithm, corresponding to the one in Section 2.1, is similar. Bag b is as before, and c contains fewer than k items at any point in the algorithm.

```
initially  $b, c := B, \phi$
while  $b \neq \phi$ do
   pick any $x$ from $b$  $\{x \in b\}$
   if  $c \cup \{x\}$ has $k$ distinct items
      then  $b := b - \{x\}$; remove $k$ distinct elements from $c \cup \{x\}$
      else  $\{c \cup \{x\}$ has fewer than $k$ distinct items$\}$ $b, c := b - \{x\}, c \cup \{x\}$
enddo
```

Correctness We assert that the algorithm terminates, and at termination c includes all heavy hitters of B. Each proposition for the majority case has a counterpart here.

1. Bag c contains fewer than k distinct items at any point during the algorithm: This assertion holds initially. In the iteration if the "then" clause holds then, inductively, c has fewer than k distinct items and the step may remove, but does not add any element to c. So, c continues to hold fewer than k distinct items after the step. For the "else" clause, the precondition is $c \cup \{x\}$ has fewer than k distinct items; so, assigning $c := c \cup \{x\}$ satisfies the assertion after the step.

2. $hh(B) \subseteq hh(b \cup c)$ is invariant: The invariant holds initially because $b \cup c = B$. Next, we show that each step of the loop preserves the invariant. The step corresponding to the "else" clause does not alter $b \cup c$. For the step corresponding to the "then" part, apply Proposition 6.

3. The algorithm terminates because each step decreases the size of b.

4. At termination, c includes all the heavy hitters of B, i.e., $hh(B) \subseteq c$: Actually, we prove the stronger result, $hh(B) \subseteq hh(c)$. From (2), above, $hh(B) \subseteq hh(b \cup c)$ is invariant. And, at termination $b = \phi$, so, the result follows.

A subtle difference between majority and heavy hitters algorithms For the majority problem, at termination c is either empty or it consists of identical elements. If c is non-empty, the item in it is the sole candidate for majority. For heavy hitters we have a stronger result. Since $hh(B) \subseteq hh(c)$ at termination, we may discard all items that are *not* heavy hitters in c from further consideration.

To see that some elements of c can be discarded, consider $c = \{1, 1, 1, 2\}$ at termination with $k = 3$. Now, $hh(c) = \{1\}$ because only item 1 occurs more than $4/3$ times in c. Therefore, 2 can be discarded as a heavy hitter candidate for B, just by considering c. This c could have resulted from $B = \{1, 2, 3, 1, 1, 1, 2\}$ where the elements of B are picked in the algorithm in the order shown.

Finding the heavy hitters of c can be very efficient. Bag c can be stored as a set of pairs (i, n) where item i occurs n times in c. At termination i is a heavy hitter of c if $n > |c|/k$.

Implementation We do not discuss the appropriate data structure for c so that the crucial test "$c \cup \{x\}$ has k distinct items" and removing k distinct items from $c \cup \{x\}$ can be done efficiently. The algorithm in Misra and Gries [2] shows a data structure that takes $\emptyset(\log k)$ time for each iteration, so the total time is $|B| \times \emptyset(\log k)$.

References

[1] R.S. Boyer and J Moore. MJRTY - a fast majority vote algorithm. In R.S. Boyer, editor, *Automated Reasoning: Essays in Honor of Woody Bledsoe*, Automated Reasoning Series, pages 105–117. Kluwer Academic Publishers, Dordrecht, The Netherlands, 1991.

[2] Jayadev Misra and David Gries. Finding repeated elements. *Science of Computer Programming*, 2:143–152, 1982.

[3] Tim Roughgarden and Gregory Valiant. CS168: The modern algorithmic toolbox, lecture 2: Approximate heavy hitters and the count-min sketch, 2015.