Majority voting: characterization and algorithms

H. Zantema

RUU-CS-88-32
October 1988

Rijksuniversiteit Utrecht

Vakgroep informatica

Padualaan 14 3584 CH Utrecht
Corr. adres: Postbus 80.089, 3508 TB Utrecht
Telefoon 030-531454
The Netherlands

Majority voting: characterization and algorithms

H. Zantema

Technical Report RUU-CS-88-32
October 1988

Department of Computer Science
University of Utrecht
P.O.Box 80.089, 3508 TB Utrecht
The Netherlands

Majority voting: characterization and algorithms

H. Zantema

Abstract

A complete characterisation of the majority function on bags is given. A linear algorithm to compute it is derived, this algorithm is optimal with respect to the number of element comparisons. Finally, a very simple $(n \log k)$ algorithm is given to find the set of k-majority elements of a bag with n elements.

Contents

1 Introduction

Let D be any set and let $Bag(D)$ be the set of finite bags over D. We give a complete characterization of the partial majority function on $Bag(D)$: if some value in the bag occurs more than all other values together, then this value is called the majority, otherwise no majority exists. To avoid undefinedness we consider the result of the majority function as a subset of D containing at most one element: if the result contains an element, then this element is the majority, and if the result is the empty set, then no majority exists.

The precise definition of the majority function is as follows. For $B \in Bag(D)$ and p a predicate on D let $N(B,p)$ be the number of elements of B satisfying p. Let $\mathcal{P}(D)$ be the set of subsets of D. Define

$$maj : Bag(D) \to \mathcal{P}(D)$$

by

$$maj(B) = \{x \in B \mid N(B,= x) > N(B,\neq x)\}.$$

Clearly $maj(B)$ contains at most one element.

An immediate drawback of this definition is the occurrence of $N(B,p)$. Intuitively it has to be possible to define majority without counting: a bag has majority x if for each possible sequence of pairs of distinct elements which can be removed from the bag, always at least one element x remains. We try to catch this idea in a small set of properties of the majority function which together characterize the majority function completely. In section 2 we give such a characterization in four properties and show that none of these four properties may be left out.

The goal of such a characterization is that it may be helpful to derive an algorithm to compute the characterized function and to prove its correctness. The idea to do so is from A. J. M. van Gasteren. Much of this work has been done in collaboration with her. She has given a slightly different characterization of the majority function in terms of predicates in [1].

In section 3 the main part of a very simple linear algorithm to compute the majority function is derived from the characterization of section 2, together with its correctness proof. The derivation does not depend on the completeness of the characterization. In the literature this algorithm is referred to as the Boyer-Moore algorithm. The operational idea of our particular algorithm in which no real counting occurs (only increment, decrement and compare with zero) was suggested by S. D. Swierstra; this work was inspired by the approach of [3].

In section 4 the simple algorithm is optimized until the majority of a bag with N elements can be computed with never more than $\lceil \frac{3N}{2} \rceil - 2$ element comparisons. M. Fischer has proven in [2] by an adversary argument that no better result is possible. In the same note S. Salzberg has described a rather different algorithm with the same worst case number of comparisons.

Finally, in section 5 a simple generalization of the algorithm from section 3 is given. For a constant positive integer k the result of this algorithm is the set of k-majority elements, where a k-majority element of a bag B is an element occurring more than $\#B/k$ times in D. For $k = 2$ this corresponds to the ordinary majority. The complexity of this algorithm is $k \log \#B$. Essentially the same algorithm was given as (3) in [4]; we present it to show the similarity.

2 A characterization

Let $\sqcup$ denote bag union and let $[x, y]$ denote the bag consisting of x and y for $x, y \in D$. Both set membership and bag membership are denoted by $\in$; the empty set and the empty bag are both written as $\emptyset$.

Proposition 1 *Let D be any set and let*

$$f : Bag(D) \to \mathcal{P}(D)$$

be any function satisfying for all $x, y \in D$ and for all $B \in Bag(D)$:

1. $(x \in B \wedge (\forall z \in B : x = z)) \to f(B) = \{x\}$;
2. $f(\emptyset) = \emptyset$;
3. $(x \neq y) \to f(B \sqcup [x, y]) \subseteq f(B)$;
4. $(x \neq y \wedge f(B) = \{x\}) \to f(B \sqcup [x, y]) = \{x\}$.

Then $f = maj$.

In words (1) says that the majority of a constant non-empty bag is the element of that bag; while (2) says that the empty bag has no majority.

The remaining requirements are less trivial: (3) states that the majority of a bag – if existing – remains majority after removing two distinct elements. On the other hand (4) states that a bag having a majority keeps the same majority after adding two distinct elements one of which is the majority.

Applying the definition of the majority function maj it can be verified directly that maj satisfies these four properties. Although we shall not use it, we mention that (3) is a particular case of the more general property

$$maj(B \sqcup C) \subseteq maj(B) \cup maj(C)$$

for all $B, C \in Bag(D)$, which can also be verified directly from the definition.

Conversely, if we have a function f satisfying (1), (2), (3) and (4) we have to prove that $f = maj$. To be able to give such a proof we need two lemmas.

Lemma 1 *Let $f : Bag(D) \to \mathcal{P}(D)$ be any function satisfying (1), (2) and (3). Then each element of $f(C)$ is contained in C for each $C \in Bag(D)$.*

Proof: We prove the assertion by induction on the number of elements of C. If all elements of C are equal then the assertion is immediate from (1) and (2). So we may assume that C can be written as

$$B \sqcup [x, y] \quad \text{with} \quad x \neq y.$$

According to (3) we then have

$$f(C) \subseteq f(B).$$

From the induction hypothesis we conclude that each element of $f(B)$ is contained in B, so also in C. Hence each element of $f(C)$ is contained in C. □

Lemma 2 *Let $f : Bag(D) \to \mathcal{P}(D)$ be any function satisfying (1), (2) and (3). Then $f(C)$ contains at most one element and*

$$f(C) \subseteq maj(C)$$

for each $C \in Bag(D)$.

Proof: We shall prove by induction on the number of elements of C that $x \in maj(C)$ for each $x \in f(C)$. Assume $x \in f(C)$. According to lemma 1 x is an element of C. If all elements of C re equal to x then the assertion is immediate from (1). So we may assume that C can be written as

$$B \sqcup [x, y] \quad \text{with} \quad x \neq y.$$

According to (3) we have

$$f(C) \subseteq f(B)$$

and from the induction hypothesis we conclude

$$f(B) \subseteq maj(B).$$

So x is contained in $maj(B)$. Now we can write

$$N(C, = x) = N(B, = x) + 1 > N(B, \neq x) + 1 = N(C, \neq x),$$

so $x \in maj(C)$, which we had to prove. □

Let f be any function (1), (2), (3) and (4). We shall prove that

$$f(C) = maj(C)$$

for each $C \in Bag(D)$, again by induction on the number of elements of C. If $maj(C)$ is empty then the equality holds by lemma 2, so we may assume that

$$maj(C) = \{x\}$$

for some element $x \in C$. If C contains only elements x then by (1) we also have $f(C) = \{x\}$ and the equality holds, so we may assume that C contains also an element y with $x \neq y$. Define B by

$$C = B \sqcup [x, y].$$

Since (3) holds for maj we conclude that $maj(C) \subseteq maj(B)$. Since $maj(B)$ contains at most one element we have

$$maj(B) = \{x\}.$$

Since B is smaller than C we may conclude from the induction hypothesis that

$$f(B) = \{x\}.$$

Finally, applying (4) to this result we conclude that

$$f(C) = \{x\} = maj(C),$$

which we had to prove. So the majority function maj is the only function satisfying (1), (2), (3) and (4), and proposition 1 has been proved. □

None of these four requirements can be left away. This is simply shown by four examples of functions f which are not equal to maj:

- Let $f(B) = \emptyset$ for all bags B, then f satisfies (2), (3) and (4).
- Choose an element $d \in D$; let

  $$f(B) = maj(B) \text{ if } N(B, = d) \neq N(B, \neq d)$$

  and

  $$f(B) = \{d\} \text{ if } N(B, = d) = N(B, \neq d).$$

  Then f satisfies (1), (3) and (4).
- Let

  $$f(B) = \{x \in B \mid \forall z \in B : N(B, = x) > N(B, = z)\}.$$

  Then f satisfies (1), (2) and (4). By the way, this f can not be computed faster than quadratic in the number of elements of B, because this f can be used to check whether an arbitrary sequence contains some element more than once. By an adversary argument it can be shown that this needs at least a quadratic number of comparisons if no order on the elements is available.
- Let

  $$f(B) = \{x \in B \mid \forall z \in B : x = z\}.$$

  Then f satisfies (1), (2) and (3).

3 An algorithm

Let us examine how the four properties

1. $(x \in B \wedge (\forall z \in B : x = z)) \rightarrow f(B) = \{x\}$;
2. $f(\emptyset) = \emptyset$;
3. $(x \neq y) \rightarrow f(B \sqcup [x, y]) \subseteq f(B)$;
4. $(x \neq y \wedge f(B) = \{x\}) \rightarrow f(B \sqcup [x, y]) = \{x\}$

of proposition 1 can be used for deriving an algorithm computing the majority of a given non-empty bag C.

Property 3 states that the majority of a bag – if existing – remains majority after removing two distinct elements. This is the only of the four properties that can be used to say something about the majority of a non-constant bag in terms of a smaller bag; we shall use the following property which immediately follows from property 3:

if $x \neq y$ and

$$f(C) \subseteq f(B \sqcup [x, y]),$$

then also

$$f(C) \subseteq f(B).$$

This suggests to choose as a loop invariant:

$$f(C) \subseteq f(\text{some bag}).$$

What can we choose for *some bag*? Let B' be the bag of elements that have not been read yet. We may expect that B' will occur in *some bag*, on the other hand we may expect that elements will be read until *some bag* is small enough to be considered by property 1 or 2: it has to be constant. Further when using the property the bag will get smaller two elements at a time. We choose as a loop invariant:

$$f(C) \subseteq f(B' \sqcup x^k),$$

where x^k denotes a constant bag consisting of k elements all equal to x; here k is a non-negative integer.

If an element y is read, then this element y is removed from B', how can the invariant remain to hold? There are three possibilities:

- if $y = x$ then k can be incremented by one;
- if $k = 0$ then x can be given the value y and k the value one;

- if $y \neq x$ and $k \neq 0$ then property 3 can be applied: k can be decremented by one.

Note that for each of these possibilities the value of k remains non-negative. Choosing the notation $B' := B' \setminus [x]$ for reading an element x and $B' \neq \emptyset$ for checking if there are still elements to be read, this yields the program:

$$
\begin{array}{ll}
 & k, B' := 1, C \setminus [x] \\
; & \textbf{do } B' \neq \emptyset \rightarrow B' := B' \setminus [y] \\
 & \quad ; \ \textbf{if } y = x \qquad\quad \rightarrow k := k + 1 \\
 & \quad\ \ [\!] \ k = 0 \qquad\quad \rightarrow x, k := y, 1 \\
 & \quad\ \ [\!] \ y \neq x \wedge k \neq 0 \ \rightarrow k := k - 1 \\
 & \quad\ \ \textbf{fi} \\
 & \textbf{od}
\end{array}
$$

Written functionally, if bags are implemented as lists and an arbitrary start value x is chosen, the same program reads:

$$
\begin{array}{lll}
 & mj\ 0\ x\ C & \\
\textbf{where} & mj\ k\ x\ [\,] & = (x, k) \\
 & mj\ k\ y\ (y : B') & = mj\ (k+1)\ y\ B' \\
 & mj\ 0\ x\ (y : B') & = mj\ 1\ y\ B' \\
 & \textbf{if } \ y \neq x \wedge k \neq 0 \ \textbf{then} & \\
 & mj\ k\ x\ (y : B') & = mj\ (k-1)\ x\ B'.
\end{array}
$$

Finally, using the notation of [5] the same program can even be written as

$$\otimes \not\rightarrow (x, 0)$$

for arbitrary $x \in D$, where $\otimes$ is defined by

$$(x, k) \otimes y = \begin{cases} (x, k+1) & \text{if } x = y \\ (y, 1) & \text{if } k = 0 \\ (x, k-1) & \text{otherwise.} \end{cases}$$

After running this program the bag B' is empty, so the invariant then yields:

$$f(C) \subseteq f(x^k).$$

In the case of $k = 0$ this is equivalent by $f(C) = \emptyset$ according to property 2, so then we have established that the given bag C has no majority and we are done. In the other case, if $k > 0$, we obtain by property 1:

$$f(C) \subseteq \{x\},$$

so then we know that either the majority of C equals x or it does not exist.

In this latter case a scan over the whole bag has still to be executed to establish whether the element x is indeed the majority of C or not. Note that we didn't use property 4 until now. One should want to derive a second loop checking whether x is indeed the majority only using the properties including property 4. However, we don't do so since the shape and the correctness of this second loop is evident from the definition of majority. The whole program is

$$
\begin{array}{ll}
 & k, B' := 1, C \setminus [x] \\
; & \textbf{do } B' \neq \emptyset \rightarrow B' := B' \setminus [y] \\
 & \quad ; \ \textbf{if } y = x \rightarrow k := k + 1 \\
 & \quad \ [\!] \ k = 0 \rightarrow x, k := y, 1 \\
 & \quad \ [\!] \ y \neq x \wedge k \neq 0 \rightarrow k := k - 1 \\
 & \quad \ \textbf{fi} \\
 & \textbf{od} \\
; & \textbf{if } k = 0 \rightarrow \textit{no majority} \\
 & [\!] \ k \neq 0 \rightarrow B', n := C, 0 \\
 & \quad ; \ \textbf{do } B' \neq \emptyset \rightarrow B' := B' \setminus [y] \\
 & \quad \quad ; \ \textbf{if } y = x \rightarrow n := n + 1 \\
 & \quad \quad \ [\!] \ y \neq x \rightarrow n := n - 1 \\
 & \quad \quad \ \textbf{fi} \\
 & \quad \textbf{od} \\
 & \quad ; \ \textbf{if } n > 0 \rightarrow x \textit{ is the majority} \\
 & \quad \ [\!] \ n \leq 0 \rightarrow \textit{no majority} \\
 & \quad \ \textbf{fi} \\
 & \textbf{fi}
\end{array}
$$

4 An optimal algorithm

The goal of this section is to minimize the number of element comparisons. Let N be the number of elements of the bag. Until now the algorithm consists of two loops, each executing at most N element comparisons, so the total number of element comparisons will not exceed $2N$. In [2] an algorithm is described consisting of two subsequent loops. The first one scans the bag to find a possible majority x and arranges the elements in a list in such a way that two subsequent elements of that list are always distinct. The second loop again establishes whether this element x is indeed the majority or not. This is done by scanning the list just built; using that subsequent elements are different this never needs more than $\frac{N}{2}$ element comparisons.

In this section, we shall arrive at a different optimal algorithm only by optimizing the algorithm of the former section. First note that the selection in the first loop is non-deterministic: if both $k = 0$ and $y = x$ a choice can be made. Since we are trying to optimize the number of element comparisons and we do not count the number of equal-zero comparisons, the next choice is preferred:

$$
\begin{array}{ll}
 & k, B' := 1, C \setminus [x] \\
; & \textbf{do } B' \neq \emptyset \rightarrow B' := B' \setminus [y] \\
 & \quad ; \ \textbf{if } k = 0 \rightarrow x, k := y, 1 \\
 & \quad\ \ [\!] \ k \neq 0 \rightarrow \textbf{if } y = x \rightarrow k := k + 1 \\
 & \quad\quad\quad\quad\quad\ \ [\!] \ y \neq x \rightarrow k := k - 1 \\
 & \quad\quad\quad\quad\quad\ \ \textbf{fi} \\
 & \quad\ \ \textbf{fi} \\
 & \textbf{od}
\end{array}
$$

In this first loop the worst case number of element comparisons remains $N - 1$, but it can be far better. The next optimization concerns a more efficient way to represent a bag. After the first scan an element x is found with the property that the majority of the whole bag is either x or does not exist. Using the ordinary bag data type where bag elements are read one at a time, we may not expect to be able to check whether x is indeed the majority or not in less than N comparisons.

Consider a bag X of pairs (y, n), where y is an ordinary bag element and n is a positive integer. Let $B(X)$ be the bag union of all corresponding bags y^n. If not all n are equal to 1, then X contains fewer elements than $B(X)$. The idea now is to check whether an element x is a majority by scanning X instead of an ordinary bag. So let us try to build a bag X of pairs in the first scan. As a notation we choose $X := \emptyset$ for starting with an empty bag and $X := X \sqcup (x, n)$ for adding a pair (x, n) to X. As an extra invariant of the first loop we choose

$$B(X) \sqcup x^n \sqcup B' = C,$$

where C represents the whole bag and B' the bag of elements not yet read. Now the first loop becomes:

$$
\begin{array}{ll}
 & k, n, B', X := 1, 1, C \setminus [x], \emptyset \\
; & \textbf{do } B' \neq \emptyset \rightarrow B' := B' \setminus [y] \\
 & \quad ; \ \textbf{if } k = 0 \rightarrow X, k, n, x := X \sqcup (x, n), 1, 1, y \\
 & \quad\ \ [\!] \ k \neq 0 \rightarrow \textbf{if } y = x \rightarrow k, n := k + 1, n + 1 \\
 & \quad\quad\quad\quad\quad\ \ [\!] \ y \neq x \rightarrow X, k, n := X \sqcup (y, 1), k - 1, n + 1 \\
 & \quad\quad\quad\quad\quad\ \ \textbf{fi} \\
 & \quad\ \ \textbf{fi} \\
 & \textbf{od}
\end{array}
$$

After running this loop we have by the invariant

$$B(X) \sqcup x^n = C$$

and either $k = 0$ and C has no majority, or $k \neq 0$ and the majority of C is x or does not exist. For the latter case we can scan X after running the first loop. The whole program is:

```
   k, n, B', X := 1, 1, C \ [x], ∅
;  do B' ≠ ∅ → B' := B' \ [y]
              ;  if k = 0 → X, k, n, x := X ⊔ (x, n), 1, 1, y
                 [] k ≠ 0 → if y = x → k, n := k + 1, n + 1
                                [] y ≠ x → X, k, n := X ⊔ (y, 1), k − 1, n + 1
                                fi
                 fi
   od
;  if k = 0 → no majority
   [] k ≠ 0 → do X ≠ ∅ → X := X \ [(y, i)]
                       ;  if y = x → n := n + i
                          [] y ≠ x → n := n − i
                          fi
               od
            ;  if n > 0 → x is the majority
               [] n ≤ 0 → no majority
               fi
   fi
```

Let us compute the number of element comparisons executed by this program. Let *noc* be the number of '$x = y$'-comparisons done in the first loop, and let $N = \#C$. As an invariant of the first loop we have

$$\frac{3(N - \#B')}{2} = noc + \#X + \frac{k}{2} + 1, \tag{1}$$

which can easily be checked for each of the three cases. Let *tnoc* be the total number of element comparisons. The number of element comparisons in the second loop is equal to $\#X$. Remember that k is non-negative, so clearly

$$tnoc < \frac{3N}{2}.$$

By some case distinction we now prove that even

$$tnoc \leq \lceil \frac{3N}{2} \rceil - 2,$$

the result which, in [2], has been proved to be the best possible. If $k = 0$ after the first loop, then X is not empty since $k + \#X > 0$ is invariant in the first loop. So by 1 we obtain:

$$tnoc = noc \leq noc + \#X - 1 \leq \frac{3N}{2} - 2 \leq \lceil \frac{3N}{2} \rceil - 2.$$

If N is odd we conclude from 1 that k is odd too and:

$$tnoc \leq noc + \#X \leq \frac{3N}{2} - \frac{3}{2} = \lceil \frac{3N}{2} \rceil - 2.$$

Finally, if N is even and $k > 0$ after the first loop, then $k \geq 2$ since k is even and

$$tnoc \leq noc + \#X \leq \frac{3N}{2} - 2 = \lceil \frac{3N}{2} \rceil - 2,$$

which completes the proof.

5 Generalization to k-majority

In this section we generalize the algorithm from section 3 to an algorithm computing the k-majority of a given bag, which is defined to be the set of elements occurring in the bag with a frequency of more than $1/k$. More precisely, for an integer $k > 1$ the function

$$maj_k : Bag(D) \to \mathcal{P}(D)$$

is defined by

$$maj_k(B) = \{x \in B \mid k * N(B, = x) > \#B\}$$

for each $B \in Bag(D)$. Clearly the original majority function equals maj_2, and $maj_k(B)$ contains at most $k - 1$ elements. Similar to property 3 in the characterization of maj we have

Key property of maj_k:

If $x_1, x_2, \ldots, x_k$ are k distinct elements of D, then

$$maj_k(B \sqcup [x_1, x_2, \ldots, x_k]) \subseteq maj_k(B)$$

for each $B \in Bag(D)$.

The correctness of this key property can be shown as follows. Taking

$$C = [x_1, x_2, \ldots, x_k]$$

we see that it is a direct consequence of the more general property:

$$maj_k(B \sqcup C) \subseteq maj_k(B) \cup maj_k(C)$$

for each $B, C \in Bag(D)$. To show this general property we have to verify that

$$\text{if} \quad k(b' + c') > b + c \quad \text{then} \quad kb' > b \vee kc' > c$$

for $b = \#B$, $b' = N(B, = x)$, $c = \#C$, $c' = N(C, = x)$, for each $x \in D$, which is evidently true.

The role of the bag x^k in the algorithm of section 3 will now be taken by a bag X which will be kept small in the sense that it contains at most $k-1$ distinct elements. As before, let C denote the whole bag to be examined and let B' be the bag of elements not yet read. As an invariant we choose

$$maj_k(C) \subseteq maj_k(B' \sqcup X) \ \bigwedge$$

$$n = \text{ number of distinct elements of } X \ \bigwedge \ n < k.$$

We obtain the program:

$n, B', X := 0, C, \emptyset$
; **do** $B' \neq \emptyset \rightarrow B' := B' \setminus [y]$
; **if** $y \in X \rightarrow X := X \sqcup [y]$
$[\!]\ y \notin X \wedge n < k-1 \rightarrow X := X \sqcup [y]$
; $n := n+1$
$[\!]\ y \notin X \wedge n = k-1 \rightarrow$ *strip*
fi
od

Here *strip* means that of each of the $k-1$ distinct elements of X one copy is removed from X and n is calculated anew. For the first two alternatives the invariance of the chosen invariant is trivial, for the third alternative it is a direct consequence of the key property of maj_k.

When all elements of C have been read then $B' = \emptyset$, then by the invariant the bag X satisfies:

$$maj_k(C) \subseteq maj_k(X).$$

To determine the set $maj_k(C)$ we need a second scan over the whole bag C to establish which of the at most $k-1$ values of $maj_k(X)$ are really k-majority elements and which are not, similar to the algorithm of section 3.

What can be said about the complexity of this algorithm? Assume a total order is defined on D. Let X be implemented as an ordered list of distinct values, each value decorated with the number of occurrences of that value. Let such an ordered list be implemented by a data structure in which both searching and inserting can be done in $O(\log n)$ steps, for example by an AVL-tree. Since X contains never more than $k-1$ distinct values, we may assume that the executions of both

$$y \in X \quad \text{and} \quad X := X \sqcup [y]$$

take $O(\log k)$ steps. For the complexity of the first scan we still need an estimate of the cost of *strip*. It is possible to implement *strip* linear in k or even faster, but for the overall complexity we only need the bound $k \log k$, which can easily be

reached for any possible tree-like implementation. For the first scan we add an extra invariant:

$$\text{number of steps done} \;\leq\; c(2n - 2\#B' - \#X)\log k,$$

for a sufficiently large constant c. Here $\#$ means the total number of elements, not the number of distinct values. If an element y is read then $\#B'$ is decremented by 1. By executing $X := X \sqcup [y]$ the value of $\#X$ is incremented by 1, by executing *strip* the value of $\#X$ is decremented by $k-1$. In any case the assertion remains invariant, so at the end of the first scan we have

$$\text{total number of steps done} \;\leq\; c(2n - \#X)\log k \;\leq\; 2cn\log k.$$

For the second scan the frequency of each of the at most $k-1$ values of X in the bag C with n elements has to be computed. Clearly this can also be done in $O(n\log k)$ steps. We conclude that the complexity of the total program computing the k-majority of an n element bag is $O(n\log k)$.

The argument in the first scan is a nice example of the following. To prove that the complexity of a loop consisting of n iterations is $O(nf(n))$ it is *not* always necessary to prove that the complexity of each iteration is $O(f(n))$.

References

[1] A.J.M. van Gasteren, *On the majority of a bag*, notes AvG79 and AvG79a, 1988.

[2] M.J. Fischer and S.L. Salzberg, problem 81-5 by J. Moore, Journal of Algorithms, volume 3, 1982, pp. 375 – 379.

[3] R. Backhouse, P. Chisholm and G. Malcolm, *Do-it-yourself Type Theory (part 2)*, EATCS bulletin 35, 1988, pp. 205 – 245.

[4] J. Misra and D. Gries, *Finding repeated elements*, Science of Computer Programming, volume 2, 1982, pp. 143 – 152.

[5] R.S. Bird, *An introduction to the theory of lists*, Logic of programming and calculi of discrete design (ed. M. Broy), Springer, 1987, pp. 3 – 42.