MA-INF 4209 - Seminar Principles of Data Mining and Learning Algorithms. Final talk: Finding repeated elements

Yauheni Selivonchyk[1], Dr. Tamas Horvath[2]

[1]Student of Master Computer Science,
University of Bonn

[2]Department of Computer Science III,
University of Bonn

January 28, 2016

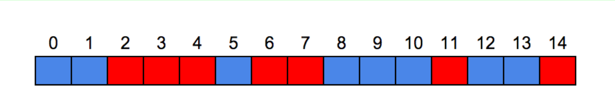
Outline

Outline

Trivial majority vote

Find the element of an array that appears more than $\left\lfloor \frac{n}{2} \right\rfloor$ times.

Majority vote example 1.1

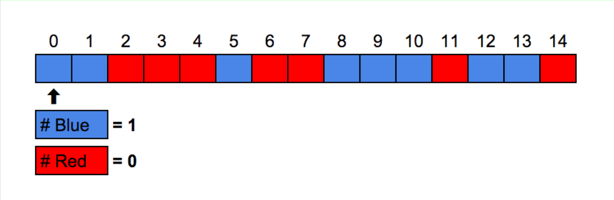


Trivial majority vote

Find the element of an array that appears more than $\left\lfloor \frac{n}{2} \right\rfloor$ times.

Use a counter for every distinct element of an array.

Majority vote example 1.2

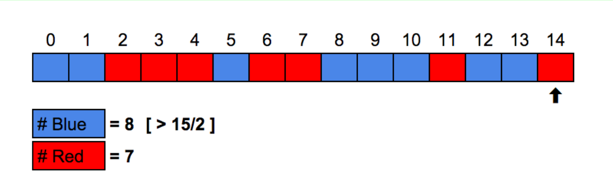


Trivial majority vote

Find the element of an array that appears more than $\left\lfloor \frac{n}{2} \right\rfloor$ times.

Use a counter for every distinct element of an array.

Majority vote example 1.3



Trivial majority vote

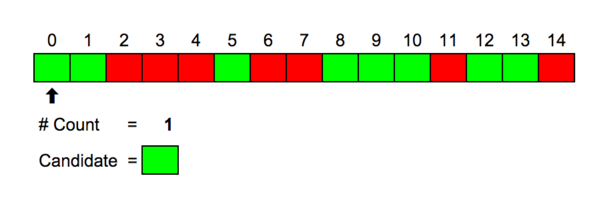
Individual counters for every distinct input requires space proportional to size of the universe$|U|$.

Can we solve the problem using single counter?

Trivial majority vote

Find the element of an array that appears more than $\left\lfloor \frac{n}{2} \right\rfloor$ times.

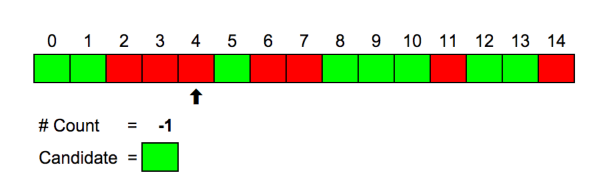
Majority vote example 2.1



Trivial majority vote

Find the element of an array that appears more than $\left\lfloor \frac{n}{2} \right\rfloor$ times.

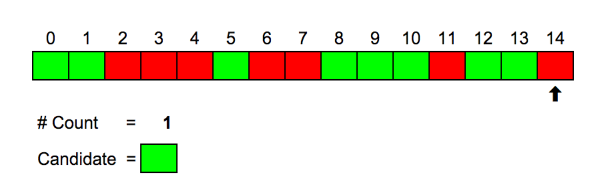
Majority vote example 2.2



Trivial majority vote

Find the element of an array that appears more than $\left\lfloor \frac{n}{2} \right\rfloor$ times.

Majority vote example 2.3



Trivial majority vote

Find the element of an array that appears more than $\left\lfloor \frac{n}{2} \right\rfloor$ times.

Majority vote example 2.4

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	
Input:																
i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
c:	0	1	2	1	0	-1	0	-1	-2	-1	0	1	0	1	2	1
Candidate:																**x**

Problem generalization

Find the element of an array that appears more than $\left\lfloor \frac{n}{2} \right\rfloor$ times.

- Can we solve it for arbitrary array using limited amount of memory?
- Can we do the same for arbitrary k (currently $k = 2$)?

Majority vote example 3

Problem definition

Given an array of n elements $b[0 : n-1]$ and integer value k s.t. $1 \leq k \leq n$ list all elements of b that occur more than $\left\lfloor \frac{n}{k} \right\rfloor$ times.

For $k = n$ problem equals to finding duplicates in the input array.

Results reflected in the publication

- Adaptation of Fast Majority Vote algorithm for arbitrary k.
- The second algorithm that solves the problem in $O(n * log(k))$ time and uses $O(k)$ extra space.
- Proves that second algorithm is optimal among algorithms based on comparing array elements.

Outline

Algorithm listing

Fast majority vote algorithm finds the only element that appears more than $\left\lfloor \frac{n}{2} \right\rfloor$ times in the input array.

Fast majority vote

```
i,c := 0,0
do i ≠ n
  if  v=b[i]         → i,c    := i+1,c+2,i+1
      c=i            → i,c,v  := i+1,c+2,b[i]
      c≠i∧v≠b[i]     → i      := i+1
  fi
od
```

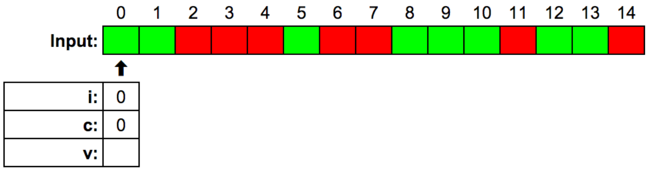
$\{R$: only v may occur more than $\left\lfloor \frac{n}{2} \right\rfloor$ in $b[0, n-1]\ \}$

Example 1

Fast majority vote alg. Loop body

$$
\begin{array}{llcll}
\textbf{if} & v = b[i] & \rightarrow & i, c & := i+1, c+2 \\
 & c = i & \rightarrow & i, c, v & := i+1, c+2, b[i] \\
 & c \neq i \wedge v \neq b[i] & \rightarrow & i & := i+1
\end{array}
$$

Fast majority vote example 1

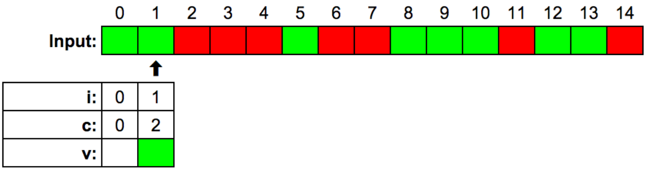


Example 1

Fast majority vote alg. Loop body

```
if   v=b[i]          → i,c     := i+1,c+2
     c=i             → i,c,v   := i+1,c+2,b[i]
     c≠i∧v≠b[i]      → i       := i+1
```

Fast majority vote example 1

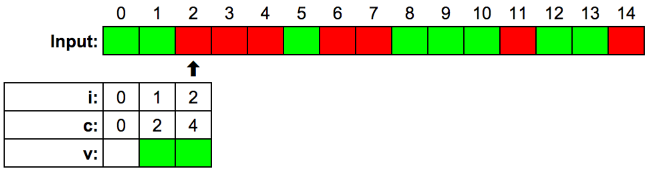


Example 1

Fast majority vote alg. Loop body

```
if   v=b[i]       → i,c    := i+1,c+2
     c=i          → i,c,v  := i+1,c+2,b[i]
     c≠i∧v≠b[i]   → i      := i+1
```

Fast majority vote example 1

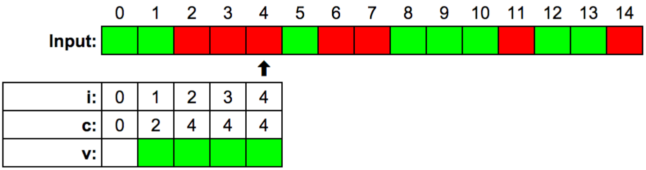


Example 1

Fast majority vote alg. Loop body

```
if  v=b[i]        → i,c    := i+1,c+2
    c=i           → i,c,v  := i+1,c+2,b[i]
    c≠i∧v≠b[i]    → i      := i+1
```

Fast majority vote example 1



Example 1

Fast majority vote alg. Loop body

```
if  v=b[i]          → i,c    := i+1,c+2
    c=i             → i,c,v  := i+1,c+2,b[i]
    c≠i∧v≠b[i]      → i      := i+1
```

Fast majority vote example 1

Input: 

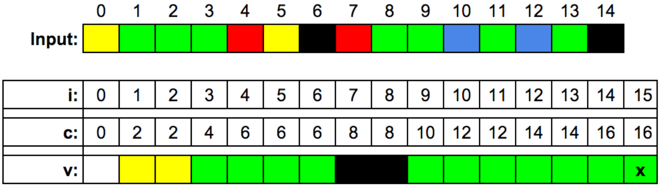
i:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
c:	0	2	4	4	4	6	6	8	8	10	10	12	12	14	16	16
v:																x

Example 2

Fast majority vote alg. Loop body

```
if  v=b[i]        → i,c    := i+1,c+2
    c=i           → i,c,v  := i+1,c+2,b[i]
    c≠i∧v≠b[i]    → i      := i+1
```

Fast majority vote example 2



Invariant

$$
\begin{aligned}
P : 0 \leq i \leq n \\
&\wedge v \text{ occurs at most } \left\lfloor \frac{c}{2} \right\rfloor \text{ times in } b[0 : i-1] \\
&\wedge i \leq c \\
&\wedge even(c) \\
&\wedge \text{ each other value occurs at most } i - \left\lfloor \frac{c}{2} \right\rfloor \text{ times in } b[0 : i-1]
\end{aligned}
$$

Invariant. continues

$$P : 0 \le i \le n$$
$$\wedge v \text{ occurs at most } \left\lfloor \frac{c}{2} \right\rfloor \text{ times in } b[0 : i-1]$$
$$\wedge i \le c$$
$$\wedge even(c)$$
$$\wedge \text{ each other value occurs at most } i - \left\lfloor \frac{c}{2} \right\rfloor \text{ times in } b[0 : i-1]$$

Fast majority vote alg. Loop body

```
if   v=b[i]          →  i,c     :=  i+1,c+2
     c=i             →  i,c,v   :=  i+1,c+2,b[i]
     c≠i∧v≠b[i]      →  i       :=  i+1
fi
```

The First Algorithm

The first algorithm extends Fast Majority Vote Algorithm to arbitrary k.

- Data structure t stores list of potential candidates
- Upper bound c is replaced with upper bound s.

Invariant

$$P : 0 \leq i \leq n$$
$$\wedge(\forall v, c : (v, c)\epsilon t \text{ occurs at most } \left\lfloor \frac{c}{k} \right\rfloor \text{ times in } b[0 : i - 1]$$
$$\text{c}\wedge c > i \wedge k \text{ divides } c)$$
$$\wedge \text{ any value not the first component of a pair in } t$$
$$\text{occurs at most } \left\lfloor \frac{s}{k} \right\rfloor \text{ times in } b[0, i - 1]$$
$$\wedge 0 \leq s \leq i$$
$$\wedge k \text{ divides } s$$

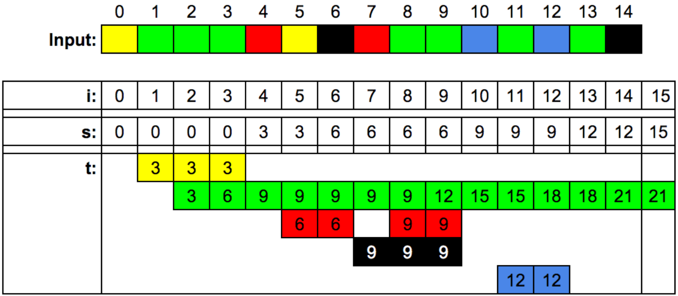
Listing

The first algorithm

```
i,s,t := 0,0,{}
do i ≠ n →
  Let j be the index of a pair v_j, c_j in t s.t. v_j = b[i]
  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 = 1 and, if any are deleted, set s to i
od {R}
```

The first algorithm example

Iterations of the first algorithm for k=3



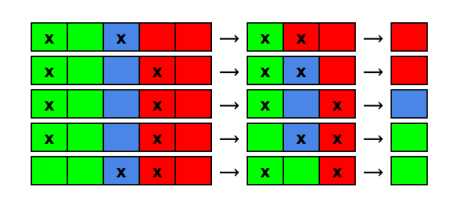
K-reduced bag

Def. A **k-reduced bag** of multiset B is a subset of B produced by repeated deletions of k distinct elements from B.

K-reduced bag

Def. A **k-reduced bag** of multiset B is a subset of B produced by repeated deletions of k distinct elements from B.

2-reduced bag example



Theorem 1

Theorem 1

Let bag B contain N items. The only value that may occur more than $\left\lfloor \frac{n}{k} \right\rfloor$ times in B are the values in a k-reduced bag for B.

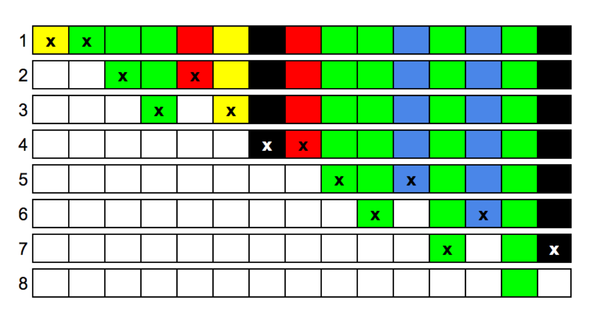
Listing

The second algorithm

```
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 {R}
```

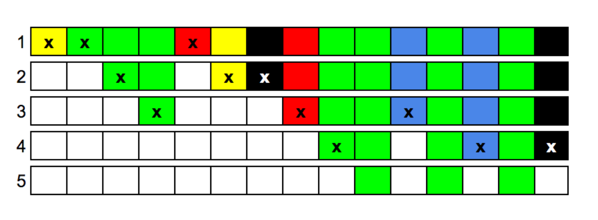
The second algorithm example

Iterations of the second algorithm for k=2



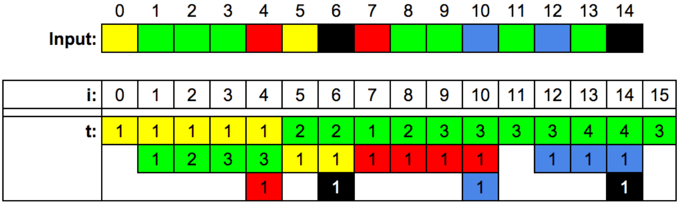
The second algorithm example

Iterations of the second algorithm for k=3



The second algorithm example

Iterations of the second algorithm for k=3



Invariant

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

Algorithm complexity

Operation	Complexity	Iterations	Impact
Initialization	O(1)	1	O(1)
Search for index $b[i]$	?	n	?
Insert element into t	?	n	?
Delete k elements from t	?	$\left\lfloor \frac{n}{k} \right\rfloor$	?

Table: Complexity of the second algorithm operations

Implementation details

Use AVL tree as a data structure for t.

Store key-value pairs $< v_i, c_i >$ where c_i is a counter. c_i is increased by one on insert and decreased on delete operation.

Operation	Complexity
Insert	O(log(k))
Look up $b[i]$	O(log(k))
Delete (decrease + delete)	O(log(k) + log(k)) = O(log(k))

Table: Complexity of the second algorithm operations

Algorithm complexity

Operation	Complexity	Iterations	Impact
Initialization	O(1)	1	O(1)
Search for index $b[i]$	O(log(k))	n	O(n*log(k))
Insert element into t	O(log(k))	n	O(n*log(k))
Delete k elements from t	O(k*log(k))	$\left\lfloor \frac{n}{k} \right\rfloor$	O(n*log(k))

Table: Complexity of the second algorithm operations

The second algorithm worst case time complexity is $O(n * log(k))$.

Outline

Decision problem

Original problem definition

Given an array of n elements $b[0 : n-1]$ and integer value k s.t. $1 \leq k \leq n$ list all elements of b that occur more than $\left\lfloor \frac{n}{k} \right\rfloor$ times.

Decision problem definition

Given an array of n elements $b[0 : n-1]$ and integer value k s.t. $1 \leq k \leq n$ determine wether there **exists an element** $v \epsilon b$ that occurs more than $\left\lfloor \frac{n}{k} \right\rfloor$ times.

Given solution of the original problem solution for the decision problem for the same input can be found in constant time.

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 $\left\lfloor \frac{n}{k} \right\rfloor$ times in $b[0 : n-1]$.

Proof outline

- Introduce decision-tree representation of an algorithm
- Construct set of inputs (r-lists)
- Show the impact of the inputs on the size of the tree
- Argue about the size of the resulting decision tree.

Proof of the theorem 2

Algorithm based on comparing array elements can be thought of as decision-tree algorithms.

Def. A **decision-tree algorithm** is a finite decision tree D together with an algorithm for descending along the tree.
Characteristics of decision tree D:

- Every nonterminal node has a label (i, j) where $0 \leq i, j < n$ (refer to array elements $b[i], b[j]$)
- Every nonterminal node has 3 branches with labels $<$, $=$, $>$
- Every terminal node has a label *YES* or *NO*
- Algorithm for descending along the tree (ommited).

r-lists

Def. An **r-list** is a list of n elements in which each of the values $[0, 1, \ldots, \left\lfloor \frac{n}{r} \right\rfloor - 1]$ occurs r times and value $\left\lfloor \frac{n}{r} \right\rfloor$ occurs $(n \mod r)$ times.

r-list examples

n=10, r=4: [0, 0, 0, 0, 1, 1, 1, 1, **2, 2**]

n=10, r=3: [0, 0, 0, 1, 1, 1, 2, 2, 2, **3**]

n=10, r=3: [2, 1, 0, 2, 1, 0, 3, 2, 1, 0]

r-list count

Number of unique r-list equals to number of unique array of size n divided by number of possible transpositions of repeated elements

$$\frac{n!}{r!^{\left\lfloor \frac{n}{r} \right\rfloor} * (n \mod r)!} \tag{1}$$

By lemma 1 and 2 (refer to original paper) we can show that this number is no smaller than:

$$(\frac{n/e}{r})^n \geq (k/e)^n, \text{where } k = \left\lfloor \frac{n}{r} \right\rfloor \tag{2}$$

Decision tree alg. and r-lists

Lemma 3.

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

Note: r-list does not contain any element with frequency greater than r. Therefore, for $k = \left\lfloor \frac{n}{r} \right\rfloor$ algorithm must terminate at node labeled "NO".

Proof of lemma 3

Proof by contradiction.

Lets assume that there exist two non-equal r-list L1 and L2 that terminate on the same node of decision tree.
Than we construct list L = L1 * L2 as follows:

$$L[i] = min(L1[i], L2[i]) \text{ for } \forall i | 0 \leq i < n$$

Than L satisfies next properties:

$$\begin{aligned} L1[i] < L1[j] \wedge L2[i] < L2[j] &=> L[i] < L[j], \\ L1[i] = L1[j] \wedge L2[i] = L2[j] &=> L[i] = L[j], \\ L1[i] > L1[j] \wedge L2[i] > L2[j] &=> L[i] > L[j]. \end{aligned} \tag{3}$$

Lemma 4

Lemma 4.

If r-lists L1 and L2 are different, then there exists a value v that occurs more than r times in $L = L1 * L2$.

Lemma 4. Continued

Proof of lemma 4 (scetch).

1. Let $v^* = min\{L1[i], L2[i] \mid L1[i] \neq L2[i],\ \forall i \epsilon [0, n)\}$;
   $\exists v^*$ by definition of L1 and L2;
2. $v^* \neq r$ $(r = max\{v \epsilon L1, L2\})$, while $L1[i] < r$ or $L2[i] < r$,
   therefore $|i \epsilon [0, n)|L1[i] = v^*| = r$;
3. Let $I = \{i \epsilon [0, n)\ s.t.\ L1[i] = v^*\}$
   and $J = \{j \epsilon [0, n)\ s.t.\ L2[j] = v^*\}$
4. $v*$ appears in L exactly $|I \cap J|$ times by properties 3.
   Since $I \neq J$ by definition of v^*, $|I \cap J| > |I| = r$.

Proof of lemma 3. continued

Lemma 3.

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

If algorithm for r-lists L1 and L2, where $L1 \neq L2$ terminates on the same node, than algorithm must terminate on the very same node for list L = L1*L2 since every test on every node along the path would yield the same outcome for L as for L1, L2, by properties 3.

Since by Lemma 3 L terminates at node labeled "YES" and on node labeled "NO" for L1 and L2 by construction of r-lists we face a **contrudiction**.

Proof of theorem 2. continued

Recap:

- By lemma 1 number of different r-lists $\geq (k/e)^n$
- By lemma 3 execution of alg. terminates on different nodes for different r-list. Therefore there exist at least $(k/e)^n$ terminal nodes.
- Longest path of a tree can not be shorter than $log(N)$, where N is a number of terminal nodes.

Proof of theorem 2. continued

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 $\left\lfloor \frac{n}{k} \right\rfloor$ times in $b[0 : n-1]$.

Length of the longest path is at least: $log_3(k/e)^n$.
At least one comparison operation is required for every node.
Therefore:

$$\begin{aligned} O(problem) &\geq O(log_3(k/e)^n) \\ &= O(n * (log(k) - log(e))/log3) \\ &= O(n * log(k)). \end{aligned} \tag{4}$$

Summary

- Two different algorithm for finding repeated elements are described in the paper.
- The second algorithm can be implemented with time complexity $O(n * log(k))$ and extra space proportional to k.
- Prove that asymptotic time complexity of the second algorithm is optimal among algorithms based on comparing array elements is provided.

Thank you!

Lemma1

$$\frac{n!}{r!^{\left\lfloor \frac{n}{r} \right\rfloor} * (n \mod r)!} \tag{5}$$

Let $x = n \mod r$. Then $\left\lfloor \frac{n}{r} \right\rfloor = (n - x)/r$.

$$\begin{aligned} r!^{\left\lfloor \frac{n}{r} \right\rfloor} * (n \mod r) &= r!^{(n-x)/r} * x! \\ &= (r!^{n-x} * x!^{r})^{1/r} \\ 1 &\leq (r!^{n-x} * r!^{x})^{1/r} \quad \text{(Lemma 2)} \end{aligned} \tag{6}$$

Lemma1. Continued

$$
\begin{aligned}
&\leq (r!^{n-x} * r!^{x})^{1/r} \quad \text{(Lemma 2)} \\
&= r!^{n/r} \\
&\leq (r^r)^{n/r} \\
&= r^n
\end{aligned}
\tag{7}
$$

Lemma1. Continued

Given: $r!^{\left\lfloor \frac{n}{r} \right\rfloor} * (n \mod r) \leq r^n$;

$$\begin{aligned} \frac{n!}{r!^{\left\lfloor \frac{n}{r} \right\rfloor} * (n \mod r)} &\geq \frac{n!}{r^n} \\ &\geq \frac{(n/r)^n}{r^n} \text{ (Using Stirling's formula)} \\ &\geq (k/e)^n. \end{aligned} \tag{8}$$