ALGORITHMS ON MAJORITY PROBLEM

A THESIS IN
Computer Science

Presented to the Faculty of the University
of Missouri Kansas City in Partial fulfillment
of the requirements for the degree

MASTER OF SCIENCE

by
RAJARSHI TARAxFDAR

B.Tech. WESTBENGAL UNIVERSITY OF TECHNOLOGY, INDIA, 2015

Kansas City, Missouri
2015
ABSTRACT

The main idea of the paper is to give solutions to the majority problem where we are counting the number of occurrences of the majority element more than half of the total number of the elements in the input set and also for the number of occurrences of the element at least half of the total number of the elements in the input set. In the model we use elements that cannot be used to index into an array and there is no order for the input elements. Thus the outcome of the comparison of two elements can only be either equal or not equal and cannot be greater than or smaller than. The focus of the paper is to propose algorithms for these problems and analyze their time complexity. For both versions we show $O(n)$ time algorithms. These results could be compared with cases whose elements can be ordered. The paper has also been modified to give the solution to the majority problem where the number of occurrences of an item exceeds (at least) more than $n/k$ times in a multiset of $n$ elements. In the model we used elements cannot be used to index into an array and there is no order for the input elements. The focus of this thesis is on the solutions and analyze the time complexity. We have achieved time $O(nk)$ for this problem and we believe this is the optimal time complexity for this problem. Moreover an algorithm is suggested in this paper to find the majority elements which is only applicable for integers.

Keywords: Algorithm, Majority, Complexity, Recursion, Group
APPROVAL

The faculty listed below, appointed by the Dean of School of Computing and Engineering, have examined a thesis titled “Algorithms on Majority Problem” presented by Rajarshi Tarafdar, candidate for the Master of Science degree, and certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Yijie Han, Ph.D., Chair
School of Computing and Engineering.

Appie van de Liefvoort, Ph.D., Co-Advisor
School of Computing and Engineering.

Deep Medhi, Ph.D.
School of Computing and Engineering
# CONTENTS

ABSTRACT......................................................................................................................... iii

ILLUSTRATIONS .................................................................................................................. vii

ACKNOWLEDGEMENTS ...................................................................................................... viii

INTRODUCTION .................................................................................................................. 1

AIM OF THE PAPER ........................................................................................................... 2

Chapter

1. ALGORITHMS FOR MAJORITY PROBLEM................................................................. 3

   1.1 Introduction .................................................................................................................. 3

   1.2 Example with more than half the total number of votes ........................................... 4

   1.3 Example with at least half the total number of votes ................................................. 4

   1.4 Approaches ............................................................................................................... 4

   1.5 Algorithm First Version, More than Half Case ....................................................... 5

   1.6 Example: Comparing potential majority elements with all elements ....................... 8

   1.7 Time Complexity First Version .............................................................................. 8

   1.8 Algorithm Second Version, At least Half Case ...................................................... 9

   1.9 Example .................................................................................................................... 11

   1.10 Example of Correctness ......................................................................................... 12

   1.11 Time Complexity Second Version ....................................................................... 12

   1.12 Further Improvement ............................................................................................. 13
2. COUNTING MAJORITY ELEMENTS AT LEAST N/K OCCURRENCES ............ 19
   2.1 Introduction ........................................................................... 19
   2.2 Example with more than n/k times ......................................... 20
   2.3 Example at least n/k cardinality of the input set .................... 20
   2.4 An O(nklogn) Time Algorithm ............................................... 20
   2.5 Algorithm 1 ......................................................................... 21
   2.6 Time complexity analysis ...................................................... 23
   2.7 Modified Approach ............................................................... 24
   2.8 Algorithm 2 ......................................................................... 26
   2.9 Time Complexity Analysis Algorithm 2 ................................ 26
   2.10 Algorithm 3 ......................................................................... 27
   2.11 Time Complexity Analysis Algorithm 3 .............................. 28
   2.12 An O(nk) Time Algorithm ................................................... 28
   2.13 Time Complexity Analysis ................................................... 29
3. FINDING MAJORITY INTEGER ELEMENT .................................. 30
   3.1 Algorithm 1 ......................................................................... 30
   3.2 Time Complexity .................................................................. 32
CONCLUSION ................................................................................. 33
REFERENCES ................................................................................ 34
VITA ............................................................................................. 35
# ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Algorithm First Version with Even number of elements</td>
<td>7</td>
</tr>
<tr>
<td>1.2</td>
<td>Algorithm First Version with Odd number of elements</td>
<td>7</td>
</tr>
<tr>
<td>1.3</td>
<td>Example returning 2 majority elements with at least half occurrences</td>
<td>11</td>
</tr>
<tr>
<td>1.4</td>
<td>Example with odd number of elements</td>
<td>16</td>
</tr>
<tr>
<td>1.5</td>
<td>Example with even number of elements</td>
<td>16</td>
</tr>
<tr>
<td>1.6</td>
<td>Example</td>
<td>17</td>
</tr>
<tr>
<td>2.1</td>
<td>Example illustration with n=10 and k=3</td>
<td>22</td>
</tr>
<tr>
<td>2.2</td>
<td>Example illustration with n=6 and k=3</td>
<td>22</td>
</tr>
<tr>
<td>2.3</td>
<td>Example illustrating comparing potential majority elements with all elements</td>
<td>23</td>
</tr>
<tr>
<td>2.4</td>
<td>Example Illustrating Modified Approach</td>
<td>24</td>
</tr>
<tr>
<td>3.1</td>
<td>Example illustrating the limitations of the algorithm by ordered partitioning</td>
<td>30</td>
</tr>
<tr>
<td>3.2</td>
<td>Example of finding majority element with ordered partitioning</td>
<td>31</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

I would like to take this opportunity to thank the following people who have directly or indirectly helped me in the academic achievements. Firstly, I would like to thank Dr. Yijie Han, my advisor, for his continuous support and guidance throughout my Masters program in Computer Science. Without his tireless effort and advise, this personal achievement wouldn’t be possible. I sincerely thank Dr. Appie van de Liefvoort for his help as co-advisor and providing references and also thank Dr. Deep Medhi for accepting to be a part of my thesis committee and making time for me from their busy schedule.

I would like to dedicate my thesis to my parents who constantly inspired me to pursue higher studies. I would like to thank my family, my dad Mr Gautom Tarafdar and my mom Mrs. Kakali Tarafdar who stood behind me all these years during my degree.

Finally I would like to thank my teachers, educational administrators, present and past and all who helped me to achieve this academic goal.
INTRODUCTION

Boyer and Moore in 1981 has proposed an algorithm. “Boyer Moore Majority Vote Algorithm” which determines majority from a sequence of elements using linear time and constant space.

The idea in this paper is based on the modification of that algorithm.

Moreover weighted majority algorithm is proposed which can be used as predication algorithm from a pool of algorithms. It assumes that we do not have prior knowledge about the algorithms in pool.

One of the algorithm is expert advise problem. Let us imagine we are investing in stock of one type. The stock price can be observed as binary, up/down. Here we are presuming we are taking the prediction of n experts. The algorithm’s goal is to limit the cumulative losses or bad predictions.

The weighted majority problem maintains the weighting of the experts. Initially all have equal weights. As time progress some experts predict better than other and the algorithms increases their weight proportionately.
AIM OF THE PAPER

The field of computer science focuses on two important topics, data structures and algorithms. These topics are particularly very important because the efficiency of our program depends on CPU time and memory usage. In other words less the time required for larger input size more efficient our algorithm is. This thesis focuses on the majority problem. We have already discussed few previous working in this topic. The majority algorithm is also known as voting algorithm which is proposed by Boyer Moore in 1981. This first chapter of this thesis focuses on how to find the majority element which has more than half of the total occurrences in the input set. Moreover it is modified for at least half of the total occurrences of the input. This can be valid for any choices, not only for integers. Optimal time complexity is proposed for both the algorithms.

The second chapter of the thesis focuses on how to find the majority element which has at least or more than n/k occurrences of the input set where n is the size of the input and the value of k varies. Improvements has been shown for these algorithms in terms of time complexity. Optimal time complexity is also achieved here.

The third chapter focuses on how to find the majority integer element in nearly optimal time complexity. This algorithm is only valid for integers.
CHAPTER 1
ALGORITHMS FOR MAJORITY PROBLEM

In this chapter we review the algorithm proposed by Boyer and Moore and has proposed modified algorithms.

1.1 Introduction

The majority problem [8] is to find the element with occurrences at least half of the number of elements in the input set. This element found is called the majority element. Majority problem is an interesting problem in algorithms field [1][2][3][4][6][7][8]. We study two versions of this problem. The first version is the more than half majority version in which the majority element occurs more than half the number of times of the cardinality of the input set. The other version is the at least half version in which the majority element occurs at least half the number of times of the cardinality of the input set. In this case there might exist two majority elements The first version is studied in [1] and an O(n) time algorithm is presented there. In the studying of the majority problem we assume the model that input elements can not be ordered and they can not be used to index an element in an array. The result of the comparison between two elements can only be equal or not equal. That is the comparison will not return the result of one element is ‘larger’ or ‘smaller’ than the other element.
1.2 Example of majority with more than half of votes [8]

Majority with more than half the number of times of the cardinality of the input set: James stares at the pile of papers in front of him. His class has just finished voting in the election of class representative. All pupils have written the name of their preferred candidate on a piece of paper, and James has volunteered to count the votes and determine the result of the election. Prior to the election the class has agreed that a candidate should become class representative only if more than half the class voted for him or her. If none of the candidates wins the absolute majority of the votes, the election will have to be repeated. James task is now to find out whether any candidate has received more than half of the total votes.

1.3 Example of majority with at least half the total number of votes

Majority with at least half the number of times of the cardinality of the input set: In the previous case James is counting the number of votes of the candidates to check whether they have received votes more than half of the total votes. But there may exist certain circumstances where one or two candidate(s) received the half of the total number of the votes. The candidate(s) must be the winner if he/she received half of the total number of the votes.

1.4 Approaches

Approaches for solving the majority problem the first version, Example 1): How should James approach this task? He doesn’t think about it much and decides to use the most straightforward method, namely putting down the names of all candidates that receive votes on a piece of paper, and keeping a tally of how many votes each of them has received. He picks up each of the ballot papers in turn to see which name has been written on it. If the name is not yet on his sheet, he writes down the name and puts one tally mark next to it. If the name is already on his sheet, he simply adds an extra tally mark next to that name.
But he thinks something different. He thinks that the complexity can be improved. so he come up with an algorithm that will reduce the complexity. Below is the proposed algorithm

1.5 Algorithm First Version, More Than Half Case

Procedure Majority1 (A [0..n-1]) /* First version, more than half case. */ Input: Array A [0..n-1] of n objects
Output: Majority element a if it exist. Otherwise report no majority element.

Main
{
for (i=0; i<n; i++) B[i]=A[i];
a=Find (A[0..n-1]);
if(a=NO-MAJORITY-ELEMENT;
then return NO-MAJORITY-ELEMENT;
count=0;
for (i=0; i<n; i++)
{
    if(B[i]==a) count++;
}
if (count > n/2) return a;
else return NO-MAJORITY-ELEMENT;
}
Sub Find(A[0..n-1])
{
if (n==1) return A[0];
if(n == 2) {
    else return NO-MAJORITY-ELEMENT;
}
if(n==3) {
    else return NO-MAJORITY-ELEMENT;
}
if (n%2==0) then n1=n;
else n1=n-1;
j=0;
for(i=0; i<n1/2; i++)
{
    {
        j++;
    }
}
if(n%2 ==1) then A[j]=A[n-1];
else j--;
Call Find(A[0..j]);

}

Note that if Find returns a which is not NO-MAJORITY-ELEMENT, a may not be the majority element.

Let us take even number of elements in the input say 6 elements.
The sequence is <4, 4, 3, 3, 4, 4>
We compare the elements in pairs {4, 4}, {3, 3}, {4, 4} thereby returning 4, 3, 4 as the elements.
Recursively comparing all the elements returned we get 4 as the potential majority element. We then compare the potential majority element 4 will all the elements in the sequence thereby returning 4 as the majority element.

Figure 1.1: Algorithm First Version with Even number of elements

Let us take odd number of elements in the input say 5 elements. The sequence is <2, 3, 2, 3, 2>
We will compare elements by pair wise. In this case compare {2, 3} and {2, 3}.
As they are not same elements discard them. We will therefore put the last element 2 in the list of majority element. Then we compare 2 with all the elements in the sequence to find out the majority element. This algorithm will return 2 as the majority element.

Figure 1.2: Algorithm First Version with Odd number of elements
1.6 Example: Comparing potential majority element with all elements.

Let the input sequence be \(<b, a, c, a, a, d, e, a, f, f>\). It consists of 10 elements. The Find procedure will return \(f\). But \(f\) is not a majority element. Thus after we got \(f\) from Find we have to compare it to all the \(n\) input elements to make sure that it is/is not the majority element.

In the above sequence we compare \(b\) with \(a\). As they are not same we will discard them. Next we will compare \(c\) with \(a\) and the same action repeats. We will continue till the last element. We will compare last two elements in the sequence. As they are same so we will keep one. We can not assume that \(f\) is the majority element. It may or may not be majority element. To come to the conclusion we must compare \(f\) with all the input elements. If the number of occurrences of \(f\) in the sequence exceed more than half of the number of elements (here >5) then \(f\) is the majority element otherwise it is not.

1.7 Time Complexity First Version

Every iteration indexed by \(i\) in procedure Find reduces the number of element to at most half. Thus the recursion is called for at most the half of number of elements. Thus the recursion relationship in this algorithm is \(T(n)=T(n/2)+O(n)\) because each recursion, with a call to problem of half size. So the solution is \(T(n)=O(n)\).
PROCEDURE Find Majority 2 (Second version, at least half case)

Input: Array A[0..n-1] of n objects.

Output: Majority element (a, b). If a and/or b is nil then it is not a majority element.

Main:
{
if(n==1)
return (A[0], nil);
if(n == 2)
{
if(A[0] == A[1])
return (A[0], nil)
else return (A[0], A[1]);
}
n1=(n+n%2)/2;
(a, b)=Find Majority 2(A[0..n1]);
(c, d)=Find Majority 2(A[n1+1..n-1]);
counta=0;
countb=0;
countc=0;
countd=0;
for(i=0; i<n; i++)
if(A[i]==a) counta++;
if(A[i]==b) countb++;
if(A[i]==c) countc++;
if(A[i]==d) countd++;
}
(e, f)=(nil, nil); count=0;
if (counta >= (n+n%2)/2) { e=a; count++; } if(countb >= (n+n%2)/2) {
if(count==0) {e=b; count++; }
else f=b;
}
if(countc >= (n+n%2)/2)
{
if(count==0) { e=c; count++; }
else f=c;
if(countd >= (n+n%2)/2)
{
if(count==0) { e=d; count++; }
else f=d;
}
return (e, f);
Note that in the second case where we are looking for majority element(s) that occur at least half of the times there may exist two majority elements.

1.9 Example

Let the input be \(<a, b>\)

This algorithm will return a and b as the majority elements.

Let us consider a sequence consists of even number of elements in the input set say 8.\(< 3, 3, 3, 3, 4, 4, 4, 4>\). It is clearly seen that here 2 elements 3 and 4 needs to be returned as the majority element since the count of 3’s and 4’s is at least equal to the half of the total number of elements in the input set .

The first recursive call will return 3 as the potential majority element.

The second recursive call will return 4 as the potential majority element.

Count the occurrences of 3 and 4 in the whole input thereby returning 3 and 4 as the majority elements since the occurrences are at least half the total number of occurrences in the given input.

Figure 1.3: Example returning 2 majority elements with at least half occurrences.
1.10 Example Correctness

<4 4 3 3 4 3 4>

Let us consider the above sequence which consists of 8 elements. The above algorithm is to study the majority element equal or greater than the half. In this sequence the number of 4’s=the number of 3’s. The above algorithm will return two majority elements. (Here 4 and 3).

The correctness of Majority 2 follows from the fact that at each recursion level we check to make sure that the elements returned are/not majority elements.

1.11 Time Complexity Second Version

Find Majority 2(A[0..n-1]) have recursive calls Find Majority 2(A[0..n1]) and Find Majority 2(A[n1+1..n-1]), thus the recursion formula for this algorithm is T(n)=2T(n/2)+O(n).The solution for the time complexity is O(n log n).
Algorithm Find Majority 3(A)

Input: set A of n elements.

Output: Majority element (a, b). If a and/or b is nil then it is not a majority element.

Main

{ }

for(i=0; i<n; i++) B[i]=A[i];

(a, b)= Find (A[0..n-1]);

counta =0; countb=0;

for(i=0; i<n; i++)

{ }

if(B[i]==a) counta++;

if(B[i]==b) countb++;

}

if(counta && countb == n/2)

return (a, b);

else if(counta >= n/2)

return (a, nil);

else if(countb >=n/2)

return (b, nil);

}
sub Find
{
if(n%2==1) call Majority 1(A)
else (n%4==2) Group every 4 elements in a group with the last 2 elements form another group.
else /* n%4 ==0 */ Group every 4 elements in a group.
for every group of 4 elements a, b, c, d do
{
if (a==b && a==c && a==d)
{
Discard any two elements and put the other two elements into set S;
}
else if(three elements (say a, b, c) are equal and d is different)
{
Discard b, c, d and put a into S;
}
else if(there are two pairs and elements within each pair are equal (say a==b && c==d))
{
Discard b and d and put a and c in S;
}
else if(there a pair of elements that are equal and other two elements are not equal (say a==b & & a != c & & a != d & c != d)
{
Discard b, c, d and put a in S;
}
else /* No two elements are equal among 4 elements */ {
    Discard all 4 elements;
}

for the group of two elements a and b do
{
    if(a==b)
    {
        Discard a and put b in S;
    }
    Else
    {
        Discard both a and b;
    }
}

Call Find Majority 3(S);
Let us take the sequence <2, 3, 2, 3, 2> which consists of 5 elements. Since the sequence consists of 5 elements that is odd number so we will call algorithm 1 which will return 2 as the majority element.

Figure 1.4: Example with odd number of elements

Let us take the sequence which consists of 8 elements <4, 4, 3, 3, 4, 3, 4, 4>

Group the elements in sub groups consisting of 4 elements in each group.

G1={4,4,3,3}
G2={3,4,3,4}

G1 will be reduced to {4, 3} as potential majority elements as per the algorithm.
G2 will be reduced to {4, 3} as potential majority elements as per the algorithm.

The recursive call is done in {4, 3, 4, 3} and will return {4, 3} as the potential majority elements.

Compare potential majority elements {4, 3} with all the elements to compute the majority elements.

The total number of 4’s and 3’s is 4 each which is half of the total occurrences of the input elements.

Figure 1.5: Example with even number of elements
Let us take a sequence which consists of 6 elements <4, 4, 3, 3, 4, 4>

Group elements into G1, G2 as per the algorithm.

G1 = {4, 4, 3, 3}
G2 = {4, 4}

G1 will return {4, 3} as the potential majority elements.
G2 will return 4 as the potential majority

The recursive call on {4, 3, 4} will return 4 as the final potential element.

Compare 4 with all the elements which will return 4 as the majority element.

Figure 1.6: Example

Theorem 1: Majority 3 finds the majority element(s) in O(n) time.

Proof: The correctness of Algorithm Majority 3 follows from the facts as shown below:

Let us consider a set consisting of 4 elements which has 4 same elements

{a, a, a, a}

So the algorithm will be designed in such a way such that if all the 4 elements are equal then we should keep two element (here a) and discard the remaining elements. Because at each recursion level we reduce the total number of elements to at most half. Thus if a is a majority element it will remain to be the majority element.
If the set consists of elements where there are 3 equal elements such as in \{a, a, a, b\} then we will keep one of the equal elements and discard the remaining elements. Because a \neq b thus throw away the pair a and b will not affect the selection of the majority element. For 2 a’s we kept one and because the number of elements is reduced to at most half, thus if a is a majority element it will remain to be the majority element.

If the set consists of two sets of equal elements such as \{a, a, b, b\} we will discard one a and b and keep the remaining two. Because recursion reduced the set to a set of at most half the number of elements and thus if a and/or b are majority elements they are kept.

If the set consists of two equal elements \{a, a, b, c\} then we discard a, b, c and keep one of the equal elements (here a). Removing b and c will not affect the counting of majority because b \neq c.

We kept one a for the number of elements is reduced to at most half and therefore if a is majority element it will be kept.

If the set consists of 4 different elements \{a, b, c, d\} we will discard all of them. In this case a and b and a\neq b and thus we throw them away. And we pair c and d and c\neq d and thus we throw them away. These actions will not affect the counting of majority elements.

**Time Complexity**

The recursion formula for this algorithm is \(T(n)=T(n/2)+O(n)\).

Therefore the time complexity is \(O(n)\).
CHAPTER 2

COUNTING MAJORITY ELEMENTS AT LEAST N/K OCCURRENCES

2.1 Introduction

The majority problem [7] is to find elements with occurrences at least n/k times in the input multiset with n elements. These elements are called majority elements. Majority problem is an interesting problem in algorithms field [1][2][3][4][6][7][8]. We study two versions of this problem. The first version is the more than n/k majority version in which the majority element occurs more than n/k times. The other version is the at least n/k version in which the majority element occurs at least n/k times. In the first version there may exist at most one majority element if k=2 while in the second version there may exist two majority elements when k=2. In both versions there may exist more than two but no more than k majority elements when k>2.

In the studying of the majority problem we assume the model that there is no order among input elements and input elements cannot be used to index an element in an array. The result of the comparison between two elements can only be equal or not equal and cannot be greater than or less than. That is, the comparison will not return the result of one element is “larger” or “smaller” than the other element.
2.2 Example [8] of majority with more than n/k times.

Majority with more than n/k times: James stares at the pile of papers in front of him. His class has just finished voting in the election of a class representative. All pupils have written the name of their preferred candidate on a piece of paper, and James has volunteered to count the votes and determine the result of the election. Prior to the election the class agreed that a candidate should become class representative only if more than n/k of the classmates voted for him or her. If none of the candidates wins the absolute majority of the votes, the election will have to be repeated. James’s task is now to find out whether any candidate has received more than n/k of all the votes.

2.3 Example at least n/k cardinality of the input set

Majority with at least n/k the number of times of the cardinality of the input set: In the previous case James is counting the number of votes of the candidates to check whether they have received votes more than n/k of the total votes. But there may exist circumstances where some candidate(s) received exactly n/k of the total number of the votes. The candidate(s) must be the winner if he/she received n/k of the total number of the votes. What will be the approach that James will follow to elect the class representative who received at least n/k the total number of the votes?

2.4 An O(nklogn) Time Algorithm

Approaches for solving the majority problem (the first version, Example 1): How should James approach this task? He doesn’t think about it much and decides to use the most straightforward method, namely putting down the names of all candidates that receive votes on a piece of paper, and keeping a tally of how many votes each of them has received. He picks up each of the ballot
papers in turn to see which name has been written on it. If the name is not yet on his sheet, he writes down the name and puts one tally mark next to it. If the name is already on his sheet, he simply adds an extra tally mark next to that name.

But he thinks something different. He thinks that the complexity can be improved. so he comes up with an algorithm that will reduce the complexity. Below is the proposed algorithm.

2.5 Algorithm 1

Input: Array A[0..n-1] of n objects and k.
Output: Majority element (a₀,a₁,…aₖ₋₁).  If a is nil then it is not a majority element.

Main:

1. Call Find(A[0,..n-1]) to get (a₀, a₁,…, aₖ₋₁). If the majority elements we are to find for the at least n/k case, then (a₀, a₁,…, aₖ₋₁) are the results and we are done. If the majority elements we are to find are for the more than n/k case then go to Step 2.

2. Compare each aᵢ, 0≤i<n, to every element in A[0..n-1] to find the number of occurrence nᵢ of aᵢ. If nᵢ>n/k then output aᵢ as a majority element.

sub Find(A[0..n-1])

1. If n ≤ 2k then compare every pair of elements to find the majority elements of A[0..n-1], else go to Step 2.

2. Make a copy of A[0..n-1] and call it B[0..n-1].

3. Call Find(A[0..n/2]) and Find(A[n/2+1..n-1]). Let (a₀, a₁,…, aₖ₋₁) be the majority elements returned by Find(A[0..n/2]) and let (b₀,b₁,…, bₖ₋₁) be the majority elements returned by Find(A[n/2+1..n-1]), where aᵢ and bᵢ could be nil if there are less than k majority elements.
4. Compare \( a_i \) (b), \( 0 \leq i < n \), to every element in \( A[0..n-1] \) to find the number of occurrences of it. If the number occurrences is \( \geq n/k \) then output it as a majority element.

Let the input sequence be \(<a\ a\ f\ a\ f\ d\ e\ a\ f\ f>\). It consists of 10 elements. Here let the value of \( k \) be 3. The main procedure calls the Find procedure will return a and f. Thus a and f are the majority elements in the at least case. After we got f and a from Find we have to compare it to all the n input elements to make sure that it is/is not the majority element in the more than case.

In the above sequence we compare all elements with a and find the number of occurrences of a is 4. We will also compare all elements with f and find the number of occurrences of f is 4. Since \( n/k = 10/3 \) and thus both a and f are majority elements.

Figure 2.1: Example illustration with \( n=10 \) and \( k=3 \)

Let us take the input consisting of 6 elements \(<c, b, a, a, c, b>\)

So \( n=6 \) and \( k=3 \).

Since here \( n \leq 2k \) is satisfied compare the input in pairs.

The algorithm will return as a, b, c as the potential majority elements.

To find at least \( n/k \) elements a, b, c will become majority elements.

Figure 2.2: Example illustration with \( n=6 \) and \( k=3 \)

22
Let us take another case study of this algorithm.

\(<c, b, a, a, a, c>\)

Here \(n=6\) and \(k=3\).

The algorithm will return \(a, c\) as potential majority elements by comparing in pairs.

To find majority elements at least \(n/k\) the algorithm will return \(a\) and \(c\) as the majority elements

To find majority element greater than \(n/k\) compare potential majority elements will all the inputs
in the sequence which returns \(a\) as the majority element.

Figure 2.3: Example illustrating comparing potential majority elements with all elements

2.6 Time complexity analysis

The recursion relationship in this algorithm is \(T(n)=2T(n/2)+O(nk)\). So the solution is

\(T(n)=O(nk\log n)\).

Theorem 1: Majority elements with more than (at least) \(n/k\) occurrences can be computed in
\(O(nk\log n)\) time.
2.7 Modified approach

We propose a solution to improve the previous algorithm. We will form groups of elements with 2k elements in a group. Then we will either delete all 2k elements in a group or combine elements in the group and assigned them labels which indicates the account of the element. The proposed algorithm is described below with an illustration.

Let us consider a sequence consisting of elements of \{2, 2, 3, 4, 4, 4, 2, 2, 3, 4, 4, 3, 1, 2, 5, 4, 6, 2, 2, 2, 5, 4, 5\}. The sequence consists of 24 elements. Let \(k=3\), and \(n/k=8\). Let us divide the elements in groups of 6 (that is \(2k\)) elements. The main aim is to compute the majority element with at least \(n/k\) occurrences. We will group 6 elements into a group. If every element in a group appears only once we will remove all element in the group. If there is an element appear more than once we will combine elements and assign labels to them. The groups of 6 elements are \{2, 2, 3, 4, 4, 4\}, \{2, 2, 2, 3, 4, 4\}, \{3, 1, 2, 5, 4, 6\}, \{2, 2, 2, 5, 4, 5\}. Every element appear only once in the third group and therefore we will remove all elements in the third group and change the value of \(n\) to \(n'=n-6=18\). We will group elements in the first group and we get 2 with label 2 (occurs twice), 3 with label 1, and 4 with label 3. We will group the elements in the second group and we get 2 with label 2, 3 with label 2 and 4 with label 2. We get 2 with label 3, 4 with label 1 and 5 with label 2 in the last group. After elements receive labels we will put elements with label 2 in a set. Note here that element 4 receive label 3 and thus we will break 4 into two elements, 4 with label 2 and 4 with label 1. After the combining and labeling we have sets:

\[S_1 \text{ with label } 1: 3, 4, 4, 2\]
We will repeatedly grouping elements for any set that has at least 2k elements. The in set $S_2$ there are more than 6 elements and we will put any 6 elements into a group and removing or combining elements. Say we put $\{2, 2, 4, 2, 3, 4\}$ into a group. Then this group will combine elements and we will get 2 with label 6 ($3 \times 2$), 4 with label 4 and 3 with label 2. We then break 2 into 2 with label 4 and 2 with label 2. After this is done we have

- $S_1$ with label 1: 3, 4, 4, 2
- $S_2$ with label 2: 2, 5, 3
- $S_4$ with label 4: 2, 4.

If in this stage we form a group with every element appear only once in the group we will also remove all elements in the group and decrease $n'$.

Because now all sets have less than 2k elements we will stop combining.

If at the time we stop combining the number of elements remaining is $n'$, then there are no more than $\log n'$ sets. We will measure elements with occurrences at least $n' / k$ among these $n'$ elements as a potential majority element. Elements appear only in sets $S_1, S_2, \ldots$.

$S_{n'/(4k^2)}$ cannot be potential majority element as the total number of occurrences of all elements in these set is $< n'/k$. Thus there can be only $O(2k \log k)$ elements in $2 \log k + 1$ sets $S_{n'/(2k^2)}, \ldots, S_{n'}$ can be potential majority elements. We compare these $O(2k \log k)$ elements in these $2 \log k + 1$ sets to all $n$ input elements in $O(nk \log k)$ time to determine whether any of them are majority elements.

Figure 2.4: Example Illustrating Modified Approach
2.8 Algorithm 2

Input: Array A[0..n-1] of n objects and k.

Output: Majority element (a_0, a_1, ..., a_{k-1}). If a is nil then it is not a majority element.

1. Let n’=n. Group every 2k elements in a group G with each element’s label set at 1.
2. If every element appears only once in G, then we discard all the elements in G and reduce the value of n to n’(n’=n’-L(G)*2k), where L(G) is the label of G.
3. If there are elements appear more than once in G, then we will label each element with the number of occurrences. However, we will group occurrences in groups with label that is a power of 2. Put elements with the same label L into set S_L.
4. If there are at least 2k elements in any set S_i, we group 2k of them into group G and go to Step 2. Else go to Step 5.
5. Let S_1, S_2, ..., S_{n’} be the sets of remaining elements. Compare every element in sets S_{n’/(2k^2)}, S_{2n’/(2k^2)}, ..., S_{n’} with all the input n elements to determine whether any of them are majority elements.

2.9 Time complexity analysis Algorithm 2

Whenever we work on a group, we either remove all elements in the group or reduce 2k elements to at most 2k-1 elements by combining elements. The when we worked on all n input elements we reduce them to n(1-1/(2k)) elements in O(nk) time (the factor k comes from where we have to pair-wise compare all elements in a group). Thus after 2k rounds with time O(nk^2) the number of elements becomes n(1-1/(2k))^{2k} \approx n/e, where e is the natural number. Thus the next 2k rounds, and the next 2k rounds, ..., will make the number of elements becomes a geometric series: n, n/e, n/e^2, n/e^3, .... Thus the time complexity is dominated by the first round which is O(nk^2). Step 5
compare $O(k \log k)$ elements to $n$ elements and thus have time $O(nk \log k)$. Thus the overall time complexity is $O(nk^2)$.

Theorem 2: Majority elements with more than (at least) $n/k$ occurrences can be computed in $O(nk^2)$ time.

Improving the Time Complexity

In this section we further improve the time complexity of our algorithm.

2.10 Algorithm 3

Input: Array $A[0..n-1]$ of $n$ objects.

Output: Majority element $(a_0, a_1, \ldots, a_{k-1})$. If $a$ is nil then it is not a majority element.

1. Divide sequence of $n$ elements in groups each consisting of $k^2$ elements.
2. Call Algorithm 1 for each group which will return $k$ elements for each group since $k^2$ elements in each group will be reduced to $k$ elements.

Note: Algorithm 1 will work for $k^2$ elements instead of $k$ elements.

3. Let $A[0..n/k]$ be the remaining elements, Call Algorithm 2;
2.11 Time complexity analysis Algorithm 3

The time complexity of returning k elements by computing with $k^2$ elements in $n/k^2$ groups is $O((n/k^2)k^3\log k)=O(nk\log k)$. The time complexity to find the majority elements among $n/k$ by grouping in $2k$ elements is $O((n/k)k^2)=O(n/k)$. Therefore the overall complexity is $O(nk\log k)$.

**Theorem 3:** Majority elements with more than (at least) $n/k$ occurrences can be computed in $O(nk\log k)$ time.

2.12 An $O(nk)$ Time Algorithm

In this section we show an $O(nk)$ time algorithm for the majority problem. We believe that $O(nk)$ time complexity is the optimal time complexity for the majority problem.

**Algorithm 4**

**Input:** Array $A[0..n-1]$ of $n$ objects and $2k$ elements in each group

**Output:** Majority element $(a_0, a_1, \ldots, a_{k-1})$. If $a$ is nil then it is not a majority element

1. Divide $n$ elements in elements containing each groups of $2k$ elements and split into $n/2k$ groups.
2. We will compare every pair of elements in each group of $2k$ elements in $O(k^2)$ time.
3. In each group if more than $k$ elements occurs only once in the group of $2k$ elements we discard $k$ of these elements which occur only once. $2k$ elements is therefore reduced to $k$ elements.
4. If there are at most $k$ elements occurs only once, there will be at least $a>k$ elements which occur more than once. We combine these $a$ elements with labels and reduce them to at most $2a/3$ elements. Note here the worst combining ratio is to combine 3 equal elements to 1 element with label $2L$ and 1 element with label $1L$, where $L$ is their label before combining.
5. Because of Step 3 and Step 4, \(2k\) elements will therefore be reduced to at most \((k+2k/3)=5k/3\) elements.

6. Run steps 1 to 5 repeatedly until we find out the majority elements.

2.13 Time complexity analysis:

\(2k\) elements is reduced to \(5k/3\) elements in each group. Therefore \(n\) elements will be reduced to \(5n/6\) elements. Comparison in one group to find out elements occurring once or more than once can be done in \(O(k^2)\) time. There are total \(n/2k\) groups and therefore the time becomes \(O(nk)\).

After the 1st iteration the time complexity will be \((5/6)(n/2k)\times O(k^2)\). Similarly after the \(i\)th iteration the time complexity will be \((5/6)^i \times (n/2k) \times O(k^2)\). Therefore the overall time complexity will be \(O(nk)\) which we believe is the optimal time complexity.

**Theorem 4:** Majority elements with more than (at least) \(n/k\) occurrences can be computed in \(O(nk)\) time.
CHAPTER 3

FINDING MAJORITY INTEGER ELEMENT

3.1 Algorithm 1

1. Let \( i = \lceil \log(\log(n/\log(n/(2k))) \rceil \).

2. Ordered partition \( n \) elements into \( n^{1-1/2^i} \) ordered sets with each ordered set having size 
   \( n^{1/2^i} \leq n/(2k) \) by using the ordered partition algorithm in [5].

3. For each ordered set \( S_i \) do

4.   

5.   Pick an arbitrary element \( a \) in the set and compare every element in the set with \( a \).

6.   If any element in each set is not equal to \( a \)

7.   then discard that set.

8.   else compare \( a \) with every element in \( S_{i-1} \) and \( S_{i+1} \) to confirm whether the occurrences

   of \( a \) is at least \( n/k \).

9.   }

Let us take a make a ordered partition of a sequence. Suppose it is \(<2, 0, 3, 3><4, 3, 4, 3><5, 4, 6, 6><7, 8, 6, 7>\).Here 3 can be majority element but we could not find it. We need
majority element appearing more than \( n/(k) = (16/2) \) times to find it. Below is the illustration
given where we can find the majority element by this procedure.

Figure 3.1: Example illustrating the limitations of the algorithm by ordered partitioning.
Let us take the sequence <2, 3, 3, 3, 3, 3, 3, 4, 4, 7, 8, 8, 7>

There are 16 elements n=16.

Initially divide into \( n^{1/2} \) sets that is 4 sets.

Each set consists \( n^{1/2} \) elements that is 4 elements in each set.

Here the value of \( k \leq n^{1/2}/2 \). Here \( n^{1/2} \leq n/(2k) \)

Elements in 1\(^{st}\) set < Elements in 2\(^{nd}\) set < Elements in 3\(^{rd}\) set < Elements in 4\(^{th}\) set.

Note: The partition is ordered partition.

\( S_1 = <2, 3, 2, 3> \)

\( S_2 = <3, 3, 3, 3> \)

\( S_3 = <3, 3, 4, 4> \)

\( S_4 = <7, 8, 8, 7> \)

Compare an element with all the remaining elements in each set.

If all the elements in the set are not equal discard them.

Therefore we discard S1, S3 and S4 and found 3 as the only element in S2.

Compare 3 with all the elements in S1 and S3 and the occurrences are 8 which is at least \( n/k=8 \).

Figure 3.2: Example of finding majority element with ordered partitioning
3.2 Time Complexity

Initially n elements are divided into \( n^{1-1/2^i} \) sets by ordered partitioning [5] which takes \( O(ni) \) time complexity. In the next step we are pick up an arbitrary element \( a \) from each set \( S_i \) and compare every elements in \( S_i \) with \( a \). This takes \( |S_i| \) time for \( S_i \) and overall \( O(n) \) time for all sets. If we find all the elements in \( S_i \) are equal then we will compare this element to all elements in \( S_{i-1} \) and \( S_{i+1} \) and this takes \( 2|S_i| \) time. Below is the time complexity when value of \( k \) varies.

\( k \) can be \( n^{1/2} \), \( n^{3/4} \), \( n^{7/8} \), \( n^{15/16} \).

The size of each set in the partitions will be \( n^{1/2}, n^{1/4}, n^{1/8}, \ldots, n^{1/2^i} \).

The worst case scenario when the elements are sorted in each partition is as below.

\[ n^{1/2^i} = 1 \]

\[ 1/2^i \log n = \log 2 \]

\[ \log n = 2^i \]

Therefore \( i = \log(\log n) \)

The best case scenario will be as below.

\[ n^{1/2^i} \leq n/2k \]

\[ 1/2^i \log n \leq \log(n/2k) \]

\[ 2^i \geq (\log n)/(\log(n/2k)) \]

\[ i \geq \log(\log n/\log(n/2k)) \]

Thus when \( k < n^i \) we get linear time. When \( k = n \) the input integers needs to be sorted and that takes \( O(n \log \log n) \) time.
CONCLUSIONS

In this paper we have studied the majority problem. We have studied the algorithm for greater than \( n/2 \) case and also at least \( n/2 \) case we provide nearly optimal algorithms. For both the more than \( n/k \) case and the at least \( n/k \) case we provided efficient algorithms. We also presented an algorithm for finding the majority element for integer input elements. We note that the model used for the study of these algorithms is interesting and we plan to do further research on the majority problem in the near future.
REFERENCES


VITA

Rajarshi Tarafdar was born on January 03, 1993, in West Bengal, India. He completed his schooling from Sri Aurobindo Vidyamandir and Hooghly Collegiate School in 2011. He then completed his Bachelor’s degree in Computer Science and Engineering from Academy of Technology, Kolkata affiliated to West Bengal University of Technology in 2015. He has worked for Capgemini, Chennai, India as Software Engineer from August 2015 to July 2016. In Fall 2016, he came to US for his Masters in Computer Science and his research interest lies towards Software Engineering.