1. Problem:

Let A [1..n] be an array of n integers. An integer is a majority element in A if it appears strictly more than n/2 times in A. We give an algorithm that can decide whether an array A [1..n] contains a majority element and if so finds it. The algorithm runs in linear time in the worst case and the only comparisons made between the elements of A are tests of Equality.

2. Approach:

A list of elements is given in an array. We consider the elements one by one and find the majority element of the whole array.

We consider the first element of the array as the majority element in the array containing the first element only. Then we include the second element into the array, and we compare this element with the first element. If they are equal we retain the two equal elements as majority element else we discard both elements and we decide that there is no majority element in the array containing the first two elements. If in case we discarded the both elements, we consider the third element of the array as the majority element (as there is a possibility of it becoming the majority element in the subsequent arrays) of the array considered so far and we proceed likewise as we have done before.

When we are retaining ‘k’ equal elements and if the next element is different from the retained elements then we reduce the retained elements by one and we retain ‘k-1’ elements as the majority element. If it is equal to the retained elements then we increase the retained elements by one to ‘k+1’.

If we continue this way we will end up with one or more retained equal elements if the array contains a majority element, otherwise there will be no retained elements or we end up with some non-majority elements.

To avoid the second condition we again traverse the array to find the count of the retained elements. If that count is greater than n/2 then we say that there is a majority element and it is the same as the retained elements.

Implementing the above approach we will result in the following algorithm.
3. Algorithm:

Procedure MAJORITY (A[1…n])
// This algorithm finds the majority element if one exists in the given
// array.
Begin
    i←1;
    k←0;
    While (i <= n){
        If (k=0) then {
            k←k+1;
            SWAP (A [k], A [i]);
        }
        Else if (A [k]=A [i]) then {
            k←k+1;
            SWAP (A [k], A [i]);
        }
        Else k←k-1;
        i←i+1;
    }
    Count← 0;
    If (k=0) then {
        Print (“No majority element in the given array”);
        Return;
    }
    For i←1 to n step by 1 do {
        If (A [i]=A [k]) then Count←Count+1;
    }
    If (Count > n/2) then
        Print (“The majority element in the given array is”, A [k]);
    Else
        Print (“No majority element in the given array”);
End MAJORITY;

4. Proof of the algorithm:

We will prove the above program using the Programming Logic approach as follows:

Let the predicates P and LS be as follows:

P=\{n>0 \land (A [1..n] \text{ contains ‘n’ elements})\}

LS=\{1 \leq k \leq n \land \text{If a majority element exists in A then it is A [k]}\}
We need to prove the following triple a valid theorem.

\[
\{ P \}
\]

\[
k \leftarrow 0; \quad i \leftarrow 1;
\]

While \( i \leq n \) \{ 

\quad \text{Body of the Loop;}

\}

\{ LS \}

In the above algorithm, the only part that is not obvious is the correctness of the WHILE LOOP. All the remaining parts are obvious. Therefore we concentrate on the WHILE LOOP. The Loop Invariant, \( I \) is as follows.

\[
I = \{ P \land (0 \leq k \leq i-1) \land \\
(\text{If a majority element is present in A [1..i-1] then it is A [k]} ) \land \\
(A[j]=A[k] \text{ for all } 0<j<k) \land \\
(A[k+1...i-1] \text{ does not contain a majority element} ) \land \\
(\text{If k=0 then there is no majority element in the array A [1..i-1]} ) \}
\]

Using the above loop invariant we will prove the correctness of the while loop in the majority element algorithm.

\[
\{ I \land i \leq n \}
\]

If \( k=0 \) \( \rightarrow k=k+1; \text{SWAP (A [k], A [i])}; \)

\( (A \ [k]=A \ [i] \rightarrow k=k+1; \text{SWAP (A [k], A [i])}; \)

Else \( \rightarrow k=k-1; \)

fi

\( i \leftarrow i+1; \)

\{ I \}

The body of the loop can be written as above and is a valid theorem because

The Loop invariant, I is true before each iteration of the loop.

--- If k=0, we are in a situation where we are not having any majority element in the array A [1..i-1], there we are assuming that A [i] as the majority element in the array A [1..i]. As we have no majority element in the array it doesn’t matter what we consider as a majority element because the invariant says only that when there is a majority element then it is A [k]. We are swapping A [i] and A [++k] (that is A [1]), so A
[k] becomes the majority element of the array A [1..i]. Here A[k+1]th element is swapped with A[i]th element and k is increased by one so the elements of A[k+1..i-1] previously are now equivalent to A[k+1..i] though they may be in different order. So, A[k+1..i] does not contain a majority element.

--- If A[k]=A[i], we have one more instance of the majority element so far considered, we will increase k by one and swap A[k] and A[i] and our previous assumption that A[k] is a majority element still holds as new element A[i] is added and is equal to the previously considered majority element. So, A[k] is the majority element of the array A [1...i]. Here A[k+1]th element is swapped with A[i]th element and k is increased by one so the elements of A[k+1..i-1] previously are now equivalent to A[k+1..i] though they may be in different order. So, A[k+1..i] does not contain a majority element.

--- Else (A[i] != A[k] & k != 0), we know from the Loop Invariant, I (which is true before each iteration of the loop) that A[k] is the majority element if one exists for the array A [1..i-1] and A[j]=A[k] for all 0<j<k.

We have A[i] != A[k] from the else condition, that is we got an instance of non-majority element at A[i], so we reduce the number of instances of majority element left. That is k now becomes k-1, So that we can still have A[k] (if k is still greater than zero) as the majority element of the array A [1..i]. Here k is decreased by one and new element A[i] is added to the array A[k+1..i-1], so our array becomes now A[k+1..i] with two new unequal elements inserted. Since two unequal elements are added here the array doesn’t contain a majority element even after adding the two elements.

Now, we are increasing the variable i by one, So, A[k] becomes the majority element of the array A [1..i-1] and A[k+1..i-1] doesn’t contain a majority element. Thus the loop invariant, I holds at the end of the loop.

So, the loop invariant I is not changed by execution of the loop body. And hence the following is a valid triple by Iterative Rule.

```
{I}
While (i<=n){
  {I ∧ i<=n}
  Body of the loop;
  {I}
}
{I ∧ i=n+1}
```

Now, the complete proof outline of the program can be written as follows:
\[P\]
\[\begin{align*}
i & \leftarrow 1; \quad k \leftarrow 0; \\
\{ P \land i=1 \land k=0 \} \\
\text{While} \quad (i \leq n) \{ \\
\{ I \land i \leq n \} \\
\text{If} \quad (k=0) \rightarrow k \leftarrow k+1; \text{SWAP}(A[k], A[i]); \\
\quad (A[k]=A[i] \rightarrow k \leftarrow k+1; \text{SWAP}(A[k], A[i])); \\
\text{Else} \rightarrow k \leftarrow k-1; \\
\} \\
i & \leftarrow i+1; \\
\{I\} \\
\{I \land i=n+1\} \\
\{LS\}
\end{align*}\]

Since the triple given above is a valid theorem, the majority element algorithm is partially correct. And we have \(i=1\) initially and we are increasing ‘i’ by one at each iteration of the loop, So, ‘i’ reaches a value equal to \(n+1\) eventually. Thus the loop terminates as \(i > n\), and subsequently the program terminates.

Hence the program is totally correct as it is partially correct and it terminates.

5. Time Complexity:

The while loop in the above algorithm is iterated ‘n’ times and if we assume that the swapping of two elements takes constant time then complexity of the loop is in the order of ‘n’

Complexity of the While Loop \(\in \Theta(n)\).

Then we have one for loop in the algorithm, which takes time in the order of ‘n’

Complexity of the For Loop \(\in \Theta(n)\).

Now, the total time complexity of the algorithm is given by

Time Complexity, \(T(n) \in \Theta(n + n).\)
\(\in \Theta(n)\)

References: Fundamentals of Algorithms, Brassard and Bratley, 
Introduction to Algorithms, Cormen, et al., McGraw Hill, and
Concurrent Programming (Principles & Practices), by G.R.Andrews