5.3 Majority Element

> Sections (like this one) which marked with a vertical line on the left side are commentary — discussions about the algorithm development process that wouldn't be part of a writeup.

Establish the problem.

> Defining — and understanding — the precise task to be solved is an essential prerequisite for algorithm development.

- *Specifications.*

  > State complete specifications for the problem. What is the problem? What do you start with (input) and what is the end result (output)? What are the legal input instances and the required output for each?

  An array A has a majority element if more than half of its values are the same.

  Task: determine the majority element in A, if any
  Input: array A with n elements
  Output: majority element or that none exists

- *Examples.*

  > If needed, give examples (specific inputs and the corresponding outputs) of typical and special cases to clarify the specifications.

Identify avenues of attack.

> Determine the algorithmic approach(es) to try.

- *Targets.*

  > Divide-and-conquer algorithms often seek to improve on a polynomial-time iterative brute-force running time. If there aren't other targets, identify the brute force algorithm and its running time.

  Brute force: for each element, count the number of occurrences of that element. This is $O(n^2)$.

  > This step isn't about coming up with clever algorithms, but in this case another simple algorithm may easily come to mind...

  Alternatively, count the occurrences of each element and see if any appear more than $n/2$ times. This requires one pass through the array, checking for and updating each element's counter. Storing/accessing labels is $O(1)$ expected for a hash map, so the running time for this algorithm is $\Theta(n)$.

  A better implementation would be to use a hash map to store counts, requiring only one pass through the array and retrieve/update operations for each element — $O(n)$.

- *Patterns.*

  > Consider the divide-and-conquer patterns.

  > There are two main patterns: easy split and easy merge. The goal in this step is to think about how this task might fit into the patterns — we're not trying to come up with an algorithm, just to identify what pieces need to be sorted out to complete an algorithm in a particular vein.
  >
  > Easy split is potentially applicable when the input is a collection of elements — split the elements into two groups (first half, second half). Easy merge is potentially applicable when the output is a collection of elements — have one friend produce the first part of the solution and the other friend produce the second part.

  Easy split: The input is an array of elements, so the easy split is to split the array in half. Then have friends find the majority element (if any) in each half, and use that to determine to overall majority element (if any).

  Easy merge is not applicable.

> Easy merge is not applicable because the result is a single thing (a majority element or not exists).

Define the algorithm.

> Assemble the pieces of the algorithm according to the template for the particular paradigm and pattern, culminating in a statement of the algorithm itself.

- *Size.*

  > What is the size of the problem, and what constitutes a smaller problem? What is the simplest/smallest instance of the problem?

  The size is the number of elements in A i.e. the number of elements.

- *Generalize / define subproblems.*

  > Define the subproblem — the task, the input, and the output. Write this as a function definition where parameters and return values are explicitly identified.

  > The friends always have the same task as we do — if we are finding the majority element, they are finding the majority element. But it is a generalized version of the original task — if the original task is to find a majority element for the whole array, a generalized version is to find one for some portion of the array. This is generalized because if you can find the majority element in a portion of the array, you can also find the majority element in the whole array — but if you can only handle the whole array, there's no way to only handle a portion.

  Subproblem task: determine the majority element in $A[low..high]$ (inclusive), if any
  Subproblem input: array A, indexes $low \leq high$
  Subproblem output: Output: majority element or that none exists

- *Base case(s).*

  > Address how to solve the smallest problem(s). This is often trivial, or is solved via brute force.

  For $n = 0$ ($low = high + 1$), return "no majority" — the array is empty, so there is no majority element.

  For $n = 1$ ($low = high$), return $A[low]$ — only one element, so that element constitutes more than half of the values.

- *Main case.*

  > Address how to solve a typical large problem instance (one that is not a base case). Specify how many friends are needed and what subproblem is handed to each friend, as well as how the results the friends hand back are combined to solve the problem.

  > The easy split pattern gives the outline of the main case: split the array S into the first half and the second half, have the friends determine the majority element in each half, and then (somehow) figure out how to determine the overall majoriy element given the results from the friends.
  >
  > But how to determine the overall majority element? Ultimately specific tactics depend on the particular problem. In this case, there are a small number of possibilities for what the friends return so consider each in turn.
  >
  > - *Both friends report "no majority element".* There's no majority element overall — nothing occurs more than $n/4$ times in either half, so nothing can appear more than $n/2$ times in the whole array.
  > - *Both friends report the same element as the majority element.* That's the majority element — it occurs more than $n/4$ times in each half and thus more than $n/2$ times overall.
  > - *One friend reports "no majority element" or both friends report a majority element, but different ones.* If there's a majority overall, it's one of the elements reported — no other

element can occur more than $n/4$ times in either half, so no other element can appear more than $n/2$ times in the whole array.

$m \leftarrow high - low + 1$ `// number of elements`

```
// split the input
mid ← (low + high)/2

// recursively solve the two parts
maj1 ← majority(A,low,mid)
maj2 ← majority(A,mid+1,high)

// return the answer
if neither maj1 nor maj2 exist then
  report no majority
else if maj1 == maj2 then
  report maj1 as the majority
otherwise
  count the number of occurrences of maj1 and maj2 (if they exist)
  report the element with more than m/2 occurrences if there is one,
    or no majority element otherwise
```

- *Top level.*

  The top level puts the context around the recursion.

  - *Initial subproblem.*

    Specify the inputs and parameters for the initial subproblem — the one whose solution solves the original problem.

    `majority(A,0,n-1)`

  - *Setup.*

    Whatever must happen before the initial subproblem is solved.

    Nothing to do — find the majority element in A as given.

  - *Wrapup.*

    Whatever must happen to get the final answer after the solution for the initial subproblem is obtained.

    Nothing to do — the majority element in A is the desired result.

- *Special cases.*

  Make sure the algorithm works for all legal inputs — identify the cases that need to be handled and address how that handling is incorporated into the previous steps (if not already accounted for).

  Typical special cases to consider for collections are empty collections and collections with a single element. Also consider if particular values could cause problems.

  Nothing to consider — $n = 0$ and $n = 1$ have already been addressed.

- *Algorithm.*

  Assemble the algorithm from the previous steps and state it.

  There shouldn't be new elements here, instead bring together the base case(s), main case, and top level with any handling needed for special cases and state the whole algorithm.

```
algorithm majority(A)
 Task: determine the majority element in A, if any
 Input: array A with n elements
 Output: majority element or that none exists

 majority(A,0,n-1)
```

```
algorithm majority(A,low,high)

 Task: determine the majority element in A[low..high] (inclusive), if any
 Input: array A, indexes low ≤ high
 Output: Output: majority element or that none exists

 if low = high + 1 then      // no elements
   report no majority

 else if low = high then   // one element
   report A[low] as the majority

 otherwise
   m ← high − low + 1    // number of elements

   // split the input
   mid ← (low + high)/2

   // recursively solve the two parts
   maj1 ← majority(A,low,mid)
   maj2 ← majority(A,mid+1,high)

   // return the answer
   if neither maj1 nor maj2 exist then
     report no majority
   else if maj1 == maj2 then
     report maj1 as the majority
   otherwise
     count the number of occurrences of maj1 and maj2 (if they exist)
     report the element with more than m/2 occurrences if there is one,
      or no majority element otherwise
```

Show termination and correctness. Show that the algorithm produces a correct solution.

- *Termination.*

  Show that the recursion — and thus the algorithm — always terminates.

  - *Making progress.*

    Explain why what each of your friends get is a smaller instance of the problem.

    The problem is only split when $n \geq 2$ so we have $high \geq low + 1$ and thus

$$\begin{aligned} mid &= (low + high)/2 \\ &\geq (low + low + 1)/2 \\ &\geq low \end{aligned}$$

and

$$\begin{aligned} mid &= (low + high)/2 \\ &\leq (high - 1 + high)/2 \\ &< high \end{aligned}$$

This means that both subproblems $A[low..mid]$ and $A[mid + 1..high]$ have at least one element. Since there is no overlap in the ranges $low..mid$ and $mid + 1..high$, at least one element in one subproblem means at most $n - 1$ in the other, ensuring that both subproblems are smaller.

– *The end is reached.*
Explain why a base case is always reached.

The base cases are $low = high + 1$ ($n = 0$) and $low = high$ ($n = 1$) and the main case is $low < high$. To miss a base case would require $low > high + 1$ which would in turn require $low > mid + 1$ or $mid + 1 > high + 1$ in the parent case, which is impossible because $low \leq mid = (low + high)/2 \leq high$ if $low < high$.

- *Correctness.*
Show that the algorithm is correct.

– *Establish the base case(s).*
Explain why the solution is correct for each base case.

For $n = 0$, there are no elements, let alone a majority element.

For $n = 1$, there is one element and $1 > 1/2$ so it meets the definition of a majority element.

– *Show the main case.*
Assume that the friends return the correct results for their subproblem, and explain why the correct answer is then produced from those results.

We already reasoned through the ideas when figuring out the main case, but here we make the arguments more rigorous.

Let m, m_1, and m_2 be the number of elements in $A[low..high]$, $A[low..mid]$, and $A[mid+1..high]$, respectively.

Two observations:

* If some element x is neither the majority element in $A[low..mid]$ nor the majority element in $A[mid + 1..high]$, then x cannot be the majority element in $A[low..high]$. This is because if x is not the majority element, it can occur at most $m_1/2$ times in $A[low..mid]$ and at most $m_2/2$ times in $A[mid + 1..high]$.

$$\begin{aligned} m_1/2 + m_2/2 &= (m_1 + m_2)/2 \\ &= m/2 \end{aligned}$$

but $m/2$ times isn't enough to be a majority element in $A[low..high]$.

* If some element x is the majority element in both $A[low..mid]$ and $A[mid + 1..high]$, x is the majority element in $A[low..high]$. This is because if x is the majority element, it occurs more than $m_1/2$ times in $A[low..mid]$ and more than $m_2/2$ times in $A[mid + 1..high]$ so it occurs more than $m_1/2 + m_2/2 = m/2$ times in $A[low..high]$.

Now consider the cases:

* *If both friends report "no majority element", there's no majority element overall.* This follows from the first observation that an element x cannot be the overall majority element without being the majority element for at least one of the halves. Since no element is the majority element for at least one of the halves, no element can be the overall majority element. So, reporting "no majority element" is the right thing to do in this case.

    * *If both friends report the same element as the majority element, that's the overall majority element.* This is the second observation. So, reporting that element as the majority element is the right thing to do in this case.
    * *If one friend reports "no majority element" or both friends report different majority elements, the majority element, if any, is one of the ones reported.* This again follows from the first observation — no other element can be the majority element. We don't know which of the element(s) reported is the overall majority, or even if either of them are, but only those elements need to be checked. Counting the occurrences of each provides an answer as to whether either element is the majority element.
  - *Final answer.*

    > Explain why the top level — the setup plus a correct solution to the initial subproblem followed by the wrapup — means that the final result is a correct answer to the problem.

    `majority(A,0,n-1)` covers the whole array so the majority element found for the subproblem is the overall majority element, and if no majority element was found, there isn't one because there aren't any elements excluded from the initial subproblem. There isn't any setup or wrapup.

Determine efficiency.

> Evaluate the running time and space requirements of the algorithm.

- *Implementation.*

  > Identify data structures and, as necessary, specific implementations of those data structures to efficiently support the algorithm. Also fill in any algorithmic details that are needed in order to establish the running time.

  With A already specified to be an array, random access is $\Theta(1)$ and there aren't any other implementation details that affect the running time.

- *Time and space.*

  > Assess the running time and space requirements of the algorithm given the implementation identified.

  > For a recursive algorithm, write a recurrence relation for the running time. Use the table discussed in class to find a big-Oh for each recurrence relation.

  The only step that isn't $\Theta(1)$ in the main case is counting the number of occurrences of $maj1$ and/or $maj2$. That requires two traversals of $A[low..high]$, for a total of $\Theta(high - low + 1)$ work.
  $T(n) = 2T(n/2) + O(n) = \Theta(n \log n)$.

- *Room for improvement.*

  > Are the targets met? Is it necessary to do better? If improvements in running time and/or space are needed, identify possible avenues for improvement: Can you leverage the work the friends are doing and have them hand back more in order to reduce your computation? Do you need to pass all of the elements off to friends or can some be eliminated?

  $\Theta(n \log n)$ is better than the nested-loop brute force algorithm but not better than the counting strategy, though it doesn't require an additional data structure (Map) for holding the elements.

  > To do better means decreasing the number and/or size of the subproblems or improving the work done outside of the recursion itself.

  The expensive part of the additional work is counting the occurences of particular elements when the friends return different things. Having the friends return the desired counts doesn't work in this case because the subproblems would no longer be independent — friend 1 would need to know about $maj2$ in order to count it, and friend 2 would need to know about $maj1$ in order to count it.

> Having to count occurrences is a consequence of the friends returning different things — a situation which could be avoided entirely by eliminating one friend and having only one subproblem. Since subproblems must always be smaller in order to make progress, that means needing to identify one or more elements to eliminate from further consideration while ensuring that the majority element for the subproblem is still the majority element (or if there is none for the subproblem, there isn't one overall either).
>
> This is now a decrease-and-conquer algorithm, and our divide-and-conquer template no longer helps with algorithm development since the basic framework — split, solve, combine — no longer applies. Some creativity is required...

Consider: if there is a majority element x, when the n elements are divided into $n/2$ pairs, there has to be at least one pair (x,x) — because there are $> n/2$ copies of x, there can't be $n/2$ pairs of elements with at most one x each. So, a main case: split the n elements into $n/2$ pairs (p,q), keep one element from pairs where $p = q$, and discard the rest. The friend then finds the majority of the remaining elements. The base cases are the same: $n = 1$ (the element is the majority) and $n = 0$ (no majority). This yields a better running time: worst case, $T(n) = T(n/2) + \Theta(n) = \Theta(n)$ because it takes $\Theta(n)$ time to pair up the elements and at most $n/2$ elements (one from each pair) are passed on to the friend.

> This is just a sketch of the idea. To complete the algorithm development, correctness needs to be established — if x is the majority element, will it still be the majority element in the elements passed on to the friend? Special cases — an odd number of elements, so they can't be paired up exactly — also need to be addressed.