A sidebar indicates the "answer" for each step — it is content that is part of following the algorithm development process, separate from commentary about the process or the solution itself.

Majority Element

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? For optimization problems, distinguish between legal solutions and optimal ones.

> An array A has a majority element if more than half of its values are the same. Determine whether A has a majority element and, if so, report its value.
>
> Input: array `A` containing n elements
>
> Output: the majority element, or that no such element exists
>
> Note: your algorithm should not assume that the elements can be ordered (so sorting is not an option). However, comparison of two elements to determine whether they are the same is possible in constant time.

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

> "Majority" means $> n/2$, so an even-length array with $n/2$ As would not have a majority element (because there aren't enough As and there can't be more than $n/2$ of anything else).

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 of the problem is n, the number of elements. A smaller problem is fewer elements. The smallest problems are $n = 0$ (no majority) and $n = 1$ (the element is the majority element).

Targets. What are the time and space requirements for your solution?

For divide-and-conquer, the goal is often to beat the brute force algorithm.

> Brute force algorithm: for each element, loop through the array, counting the number of occurrences of that element. $\Theta(n^2)$
>
> Another approach: loop through the array, counting the number of occurrence of each element — if at any point, the count for an element is greater than $n/2$, the majority element has been found. (If no such count occurs, there is no majority element.) $\Theta(n)$ using a hashtable implementation of a Map for the counts. Note that this approach requires being able to compute a hashcode for each element.

Tactics. The time and space constraints may narrow down the algorithmic options and/or may guide you in particular directions. Consider both things you can and can't do.

> Note: your algorithm should not assume that the elements can be ordered (so sorting is not an option). However, comparison of two elements to determine whether they are the same is possible in constant time.

Approaches. Consider specific patterns — what would they look like if applied to this problem?

> A *process input* approach would mean splitting the input in half, having friends produce the solution for each half — i.e. the majority element if there is one — and then compute the overall majority if there is one.

A *produce output* approach has each friend producing part of the solution, but that's not applicable here — the solution isn't a collection of things.

Narrowing the search space isn't applicable — while we are looking for element(s) that are part of the input, a majority element can't be identified without the context of the rest of the input.

Generalize / define subproblems. Friends get smaller versions of the original problem. Define the task, input, and output for the subproblems.

> Original problem: `majority(A)` — the majority element in `A`, if there is one
>
> Subproblem / generalized problem: `majority(A,low,high)` — the majority element in `A[low..high]`, if there is one (inclusive)
>
> Input: n-element array `A` and indexes `low` $\leq$ `high`
>
> Output: the majority element in `A[low..high]` if one exists

Base case(s). Address how to solve the smallest problem(s).

> For $n = 0$ (`high` $=$ `low` $- 1$), there is no majority element.
>
> For $n = 1$ (`high` $=$ `low`), `A[low]` is the majority element.

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

> With no elements, there is no element to be a majority. With one element, $1 > 1/2$ so that element is the majority element.

Main case. Split the problem, solve the subproblems, combine the subproblem solutions into the solution for this problem: the main case addresses how to solve a typical large problem instance. Specify how many friends are needed and what is handed to each friend, as well as how to generate what is handed to the friends and how the results the friends hand back are combined to solve the problem.

Process input gives the following structure:

```
split A[low..high] in half:  mid ← (low+high)/2
maj1 ← majority(A,low,mid)
maj2 ← majority(A,mid+1,high)
return ??  as the majority or that there isn't one
```

How to determine the majority from `maj1` and `maj2`? Consider the cases for what can happen:

- neither `maj1` and `maj2` exist — there's no majority element
- one of `maj1` and `maj2` exist — if there's a majority element, it's this element
- both `maj1` and `maj2` exist, and they are the same element — this is the majority element
- both `maj1` and `maj2` exist, and they are the different elements — if there's a majority element, it's one of these

(What to do in each case can take some thinking, and that it is the right thing to do is justified in the correctness steps.)

This leads to an algorithm –

```
split A[low..high] in half:  mid ← (low+high)/2
maj1 ← majority(A,low,mid)
maj2 ← majority(A,mid+1,high)
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
otherwise
```

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.

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.

Observe: 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 $m_1/2$ 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]`.

Also observe: 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 neither `maj1` nor `maj2` exist, there is no majority element in `A[low..high]`. This follows from the first observation — no element can be the majority in `A[low..high]` if it is not the majority in `A[low..mid]` or `A[mid+1..high]`.
- If only one of `maj1` or `maj2` exist, the majority element in `A[low..high]`, if it exists, is that majority element. This again follows from the first observation — no other element can be the majority because no element can be the majority in `A[low..high]` if it is not the majority in `A[low..mid]` or `A[mid+1..high]`.
- If both `maj1` and `maj2` exist and are the same element, that element is the majority element in `A[low..high]`. This follows from the second observation.
- If both `maj1` and `maj2` exist and but are different, the majority element in `A[low..high]`, if it exists, will be either `maj1` or `maj2`. This follows from the first observation — no other element can be the majority because no element can be the majority in `A[low..high]` if it is not the majority in `A[low..mid]` or `A[mid+1..high]` — and that they can't both be majority elements because only one element can occur more than $m/2$ times.

Top level. The top level puts the context around the recursion. Specify the inputs and parameters for the initial subproblem — the one whose solution solves the original problem. Setup covers whatever must happen before the initial subproblem is solved, and wrapup covers whatever must happen to get the final answer after the solution for the initial subproblem is obtained.

Initial subproblem. `majority(A,0,n-1)`

Setup. Nothing else is needed.

Wrapup. Nothing else is needed – the result of `majority(A,0,n-1)` is the majority element if there is one, as desired.

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.

There isn't any setup or wrapup, and the initial subproblem `majority(A,0,n-1)` includes all of the original elements so the answer to that problem is the overall answer.

Special cases. Make sure the algorithm works for all legal inputs — revise the previous steps to add handling for special cases as needed.

The base cases and main case account for all $n \geq 0$, and there isn't any special handling needed for particular elements.

Algorithm. Assemble the algorithm from the previous steps and state it.

`majority(A)` — the majority element in `A`, if there is one

Input: n-element array `A`

Output: the majority element in `A` if one exists

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

`majority(A,low,high)` — the majority element in `A[low..high]`, if there is one (inclusive)

Input: n-element array `A` and indexes `low` $\leq$ `high`

Output: the majority element in `A[low..high]` if one exists

```
if high = low-1
  report no majority
else if high = low
   report A[low] as the majority element
else
   split A[low..high] in half:  mid ← (low+high)/2
   maj1 ← majority(A,low,mid)
   maj2 ← majority(A,mid+1,high)
   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 otherwise
```

Termination. Show that the recursion ends. For making progress, explain why each subproblem is smaller. For reaching the end, show that a base case is always reached.

Making progress. At each step the range `low..high` is cut in half. Since $n \geq 2$ for the main case, there is at least one element for each half and thus both subproblems are smaller.

Reaching the end. Splitting a problem of size $n \geq 2$ means there's at least one element in each half, so the base case of $n = 1$ is always reached — it isn't possible to split a problem $n \geq 2$ and end up with a problem of size $n < 1$.

Implementation. Identify data structures and, as necessary, specific implementations of those data structures to efficiently support the algorithm. Also fill in any algorithmic steps that haven't been specified.

With `A` already specified to be an array, there aren't any other implementation details that affect running time.

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

The base cases take O(1) time each. For the main case (n elements), consider each step:

- Dividing the array is O(1) (to compute (low+high)/2).
- Two problems of size $n/2$ are generated.
- Counting the number of occurrences of up to two particular elements is O(n).
- Once the counts are known, determining which of those elements, if any, is a majority element is O(1).

This leads to the recurrence relation $T(n) = 2T(n/2) + n$. With T(1) = 1, this yields $T(n) = \Theta(n \log n)$.

Additional space requirements are O(1) for a few temporary variables.

Is this good? It's better than the $O(n^2)$ brute force algorithm... But can we do better? It doesn't seem plausible to be able to do better than $O(n)$, since it seems like we'd have to at least look at every element once to determine how many times it occurs in the array, but is $O(n)$ possible? (It certainly is when using an auxiliary data structure to simply count the number of occurrences of all elements.)

The work in the algorithm given above comes in counting the number of occurrences of the potential majority element(s). Could the friends also return a count along with their majority element? They'd have to return the counts for *all* elements, not just their majority, because the number of occurrences of the majority element in the other half is needed to determine if it is an overall majority. But for us to return the combined counts for the two halves is O(n), so that doesn't save any work over counting for ourselves.

So, another tactic... It is worth noting that "produce output" could be viewed as a single friend producing the whole solution (rather than multiple friends producing oart of the solution), and that "narrowing the search space"can still work if there's a way to eliminate things that definitely aren't what is being looked for. These amount to the same idea — how to give one friend a smaller set of elements where the majority, if there is majority, is still the majority.

Consider: if there is a majority element m, when the n elements are divided into $n/2$ pairs, there has to be at least one pair (m,m) — because there are $< n/2$ copies of m, there can't be $n/2$ pairs of elements with at most one m 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: $T(n) = T(n/2) + \Theta(n) = \Theta(n)$ 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 m 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.)