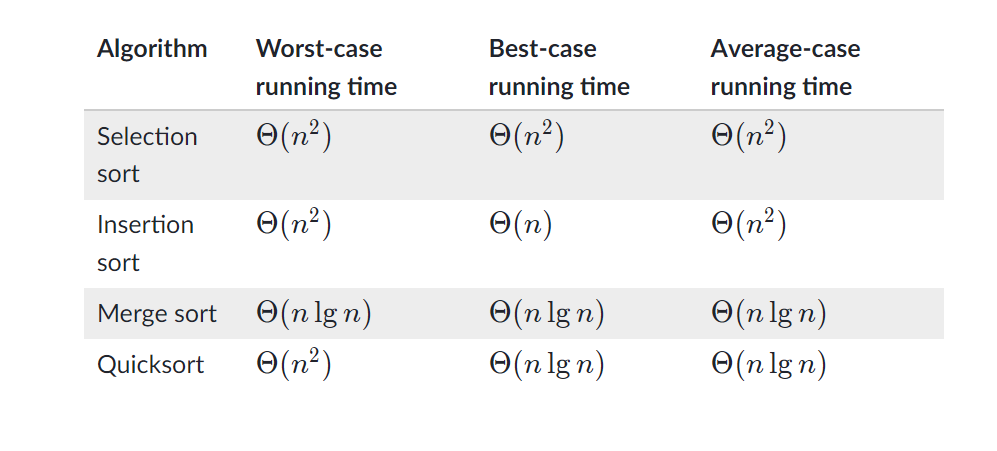
QUICK SORT

Divide and conquer (D&C) is a well-known recursive technique for solving problems. Divide and conquer is a strategy for solving a problem. A strategy is an approach or a design for solving a problem.

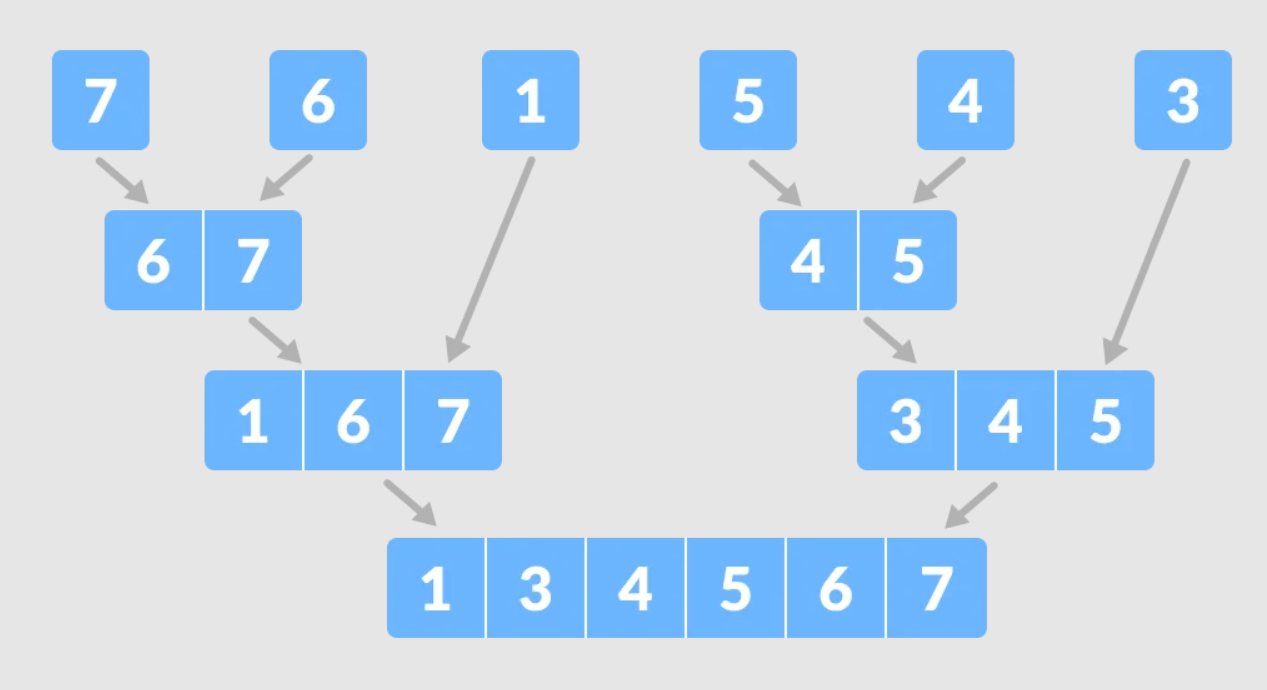
**Divide & conquer**

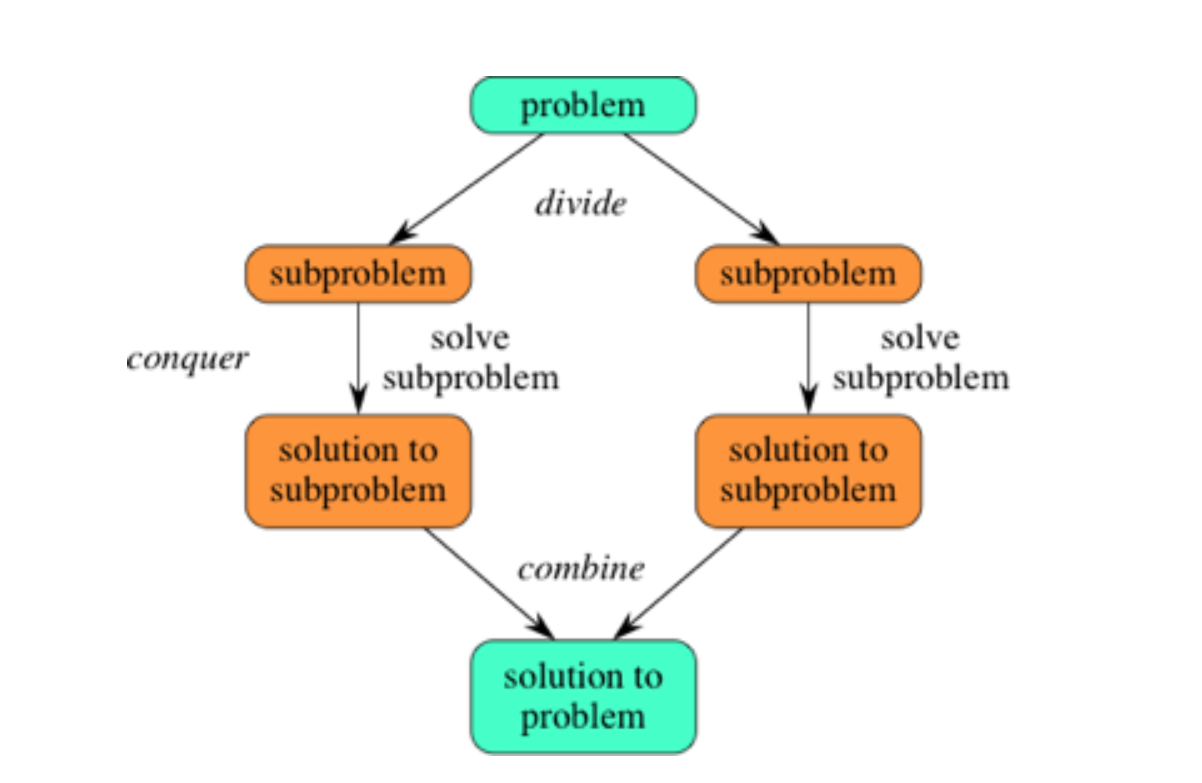
****

Both merge sort and quicksort employ a common algorithmic paradigm based on **recursion**. This paradigm, **divide-and-conquer**, breaks a problem into subproblems that are similar to the original problem, recursively solves the subproblems, and finally combines the solutions to the subproblems to solve the original problem. Because divide-and-conquer solves subproblems recursively, each subproblem must be smaller than the original problem, and there must be a base case for subproblems. You should think of a divide-and-conquer algorithm as having three parts:

1. **Divide** the problem into a number of subproblems that are smaller instances of the same problem.
2. **Conquer** the subproblems by solving them recursively. If they are small enough, solve the subproblems as base cases.
3. **Combine** the solutions to the subproblems into the solution for the original problem.

You can easily remember the steps of a divide-and-conquer algorithm as *divide, conquer, and combine*. Here's how to view one step, assuming that each divide step creates two subproblems (though some divide-and-conquer algorithms create more than two):







here’s how D&C works:

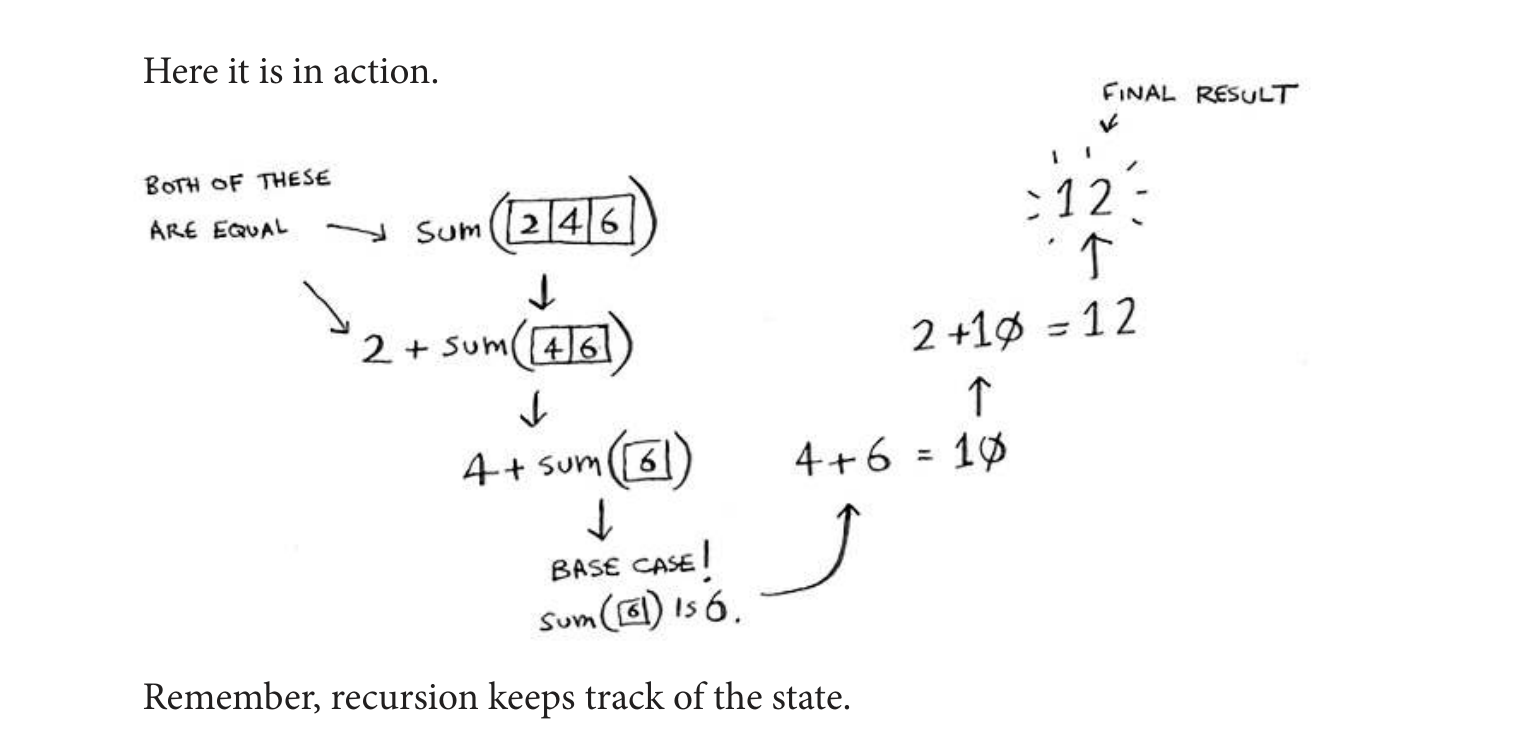
1. Figure out a simple case as the base case.

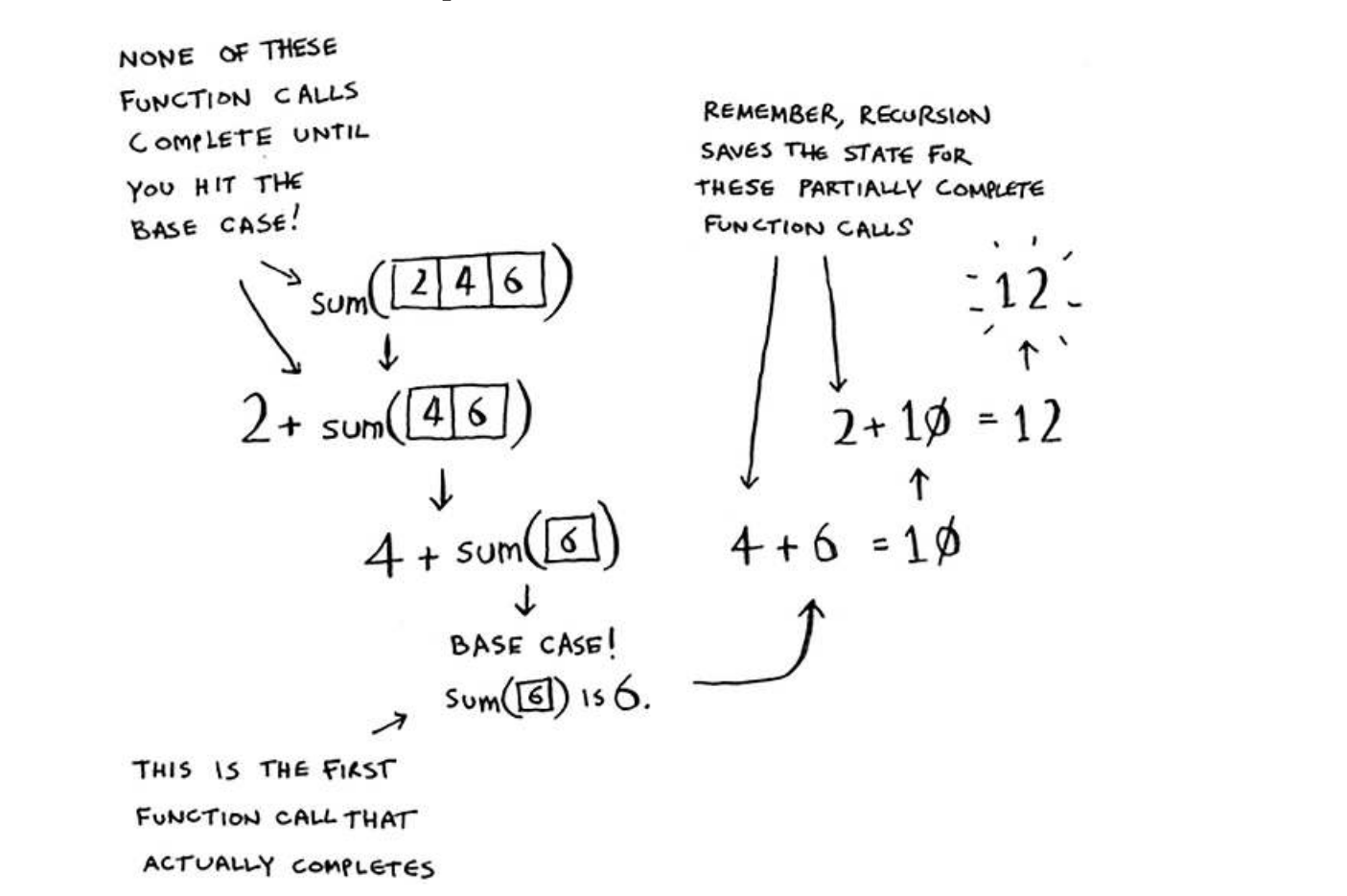
2. Figure out how to reduce your problem and get to the base case.

D&C isn’t a simple algorithm that you can apply to a problem. Instead, it’s a way to think about a problem. Let’s do one more example.

When you’re writing a recursive function involving an array, the base case is

often an empty array or an array with one element. If you’re stuck, try that irst.





First, pick an element from the array. his element is called the pivot. Now ind the elements smaller than the pivot and the elements larger

than the pivot.

his is called partitioning. Now you have

• A sub-array of all the numbers less than the pivot

• he pivot

• A sub-array of all the numbers greater than the pivot

he two sub-arrays aren’t sorted. hey’re just partitioned. But if they

were sorted, then sorting the whole array would be pretty easy.

If the sub-arrays are sorted, then you can combine the whole thing like

this—left array + pivot + right array—and you get a sorted

array. In this case, it’s [10, 15] + [33] + [] =

[10, 15, 33], which is a sorted array.

How do you sort the sub-arrays? Well, the quicksort base case already

knows how to sort arrays of two elements (the let sub-array) and

empty arrays (the right sub-array). So if you call quicksort on the two

sub-arrays and then combine the results, you get a sorted array!

quicksort([15, 10]) + [33] + quicksort([])

> [10, 15, 33]

A sorted array

