# Divide and Conquer

This chapter is going to be just an introduction about the Divide and Conquer, we are going to study algorithms based on the divide on conquer in the next three chapters.

### So, why an entire chapter just for the introduction of Divide and Conquer?

Indeed, Divide and Conquer is a very useful technique but direct jumping into the algorithms might feel difficult for beginners. So, why not first see what basically this technique is in a detailed way and then implement it to the algorithms.

The name of this technique tells a lot about the technique itself. We literally divide the problems into smaller subproblems and then conquer (or solve) the smaller subproblems first. After this, we combine the solution of the smaller subproblems to get the solution for the original problem.

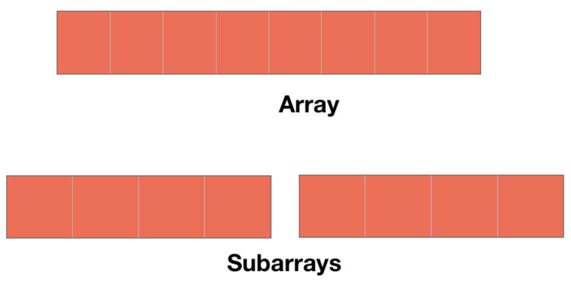
Thus, Divide and Conquer is a three-step process:

1. **Divide** → The first step is to break the problem into smaller subproblems. For example, take an example of any big organization. It would be quite difficult for a single person to directly handle all the work of the organization himself. So, the organization is divided into several departments and different people are appointed to assist those departments. In short, we are breaking our problem into smaller subproblems.
2. **Conquer** → This is basically solving of the smaller subproblems. In the example of the organizations, the problems of the departments will be solved individually by the departments.
3. **Combine** → In the last step, we combine the solutions of the smaller subproblems to get the solution of the bigger problem. For example, the whole point of an organization would be to earn profit, so whatever the departments are doing, in last, they would sum up to generate some profit for the organization.

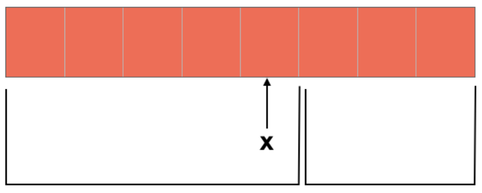
Now, let's take these steps to the domain we are concerned with i.e., programming and start our discussion with the 'divide' first.

## Implementation of Divide and Conquer

In Divide and Conquer, we generally deal with arrays and our task is to first divide the array into smaller subarrays.

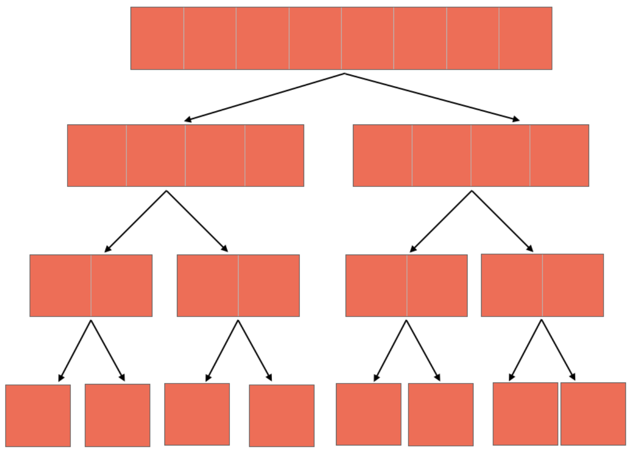


At first, it might seem that we are going to make new smaller subarrays for this purpose but this is not the case. Suppose we have to divide an array into two subarrays, we would introduce a variable which will point to the element from which we are breaking the array.



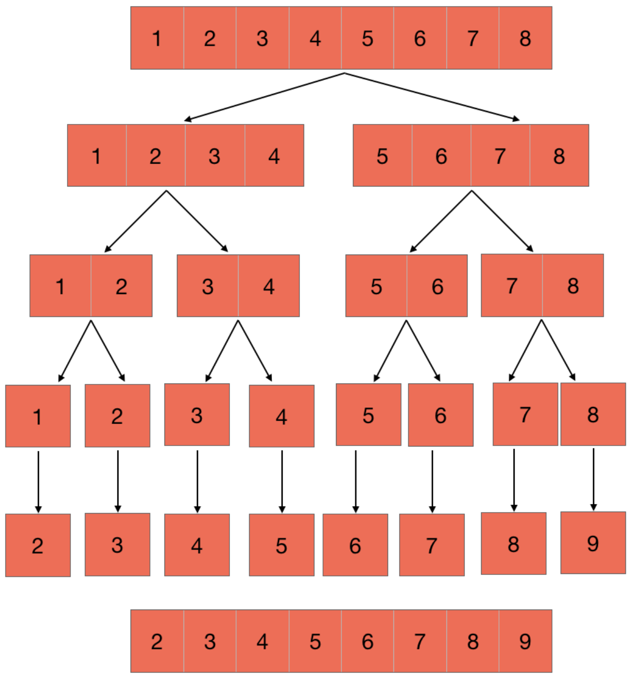
So, now we will deal with the array from 'start' to 'middle' as one smaller array and the array from 'middle+1' to 'end' as the second subarray.

Let's take an array and write a program to break it into a smaller subarrays of size 1 as shown in the first picture. We are first breaking the array into two subarrays and then again breaking those arrays into 4 subarrays and so on. So, it looks like a process of breaking an array into two halves and then repeatedly applying this process to smaller arrays to again break them in two halves.

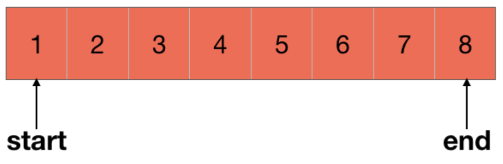


Thus, we can make a function for the process of breaking the array and then recursively pass smaller arrays to it.

Let's write a function which breaks an array into smaller arrays and when the size of smaller array reaches to 1, it increases their value by 1.



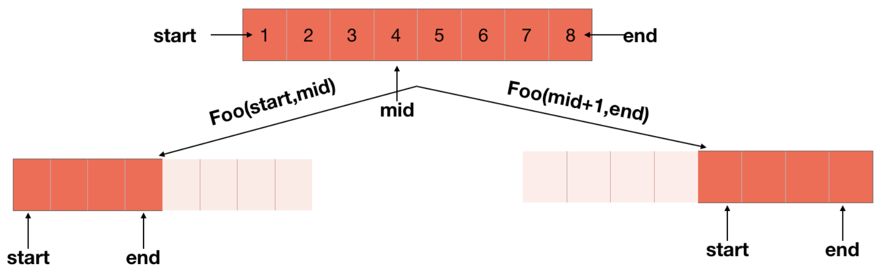
As discussed above, we are not going to make any new arrays for smaller arrays, instead, we will use variables to divide the array. It means that our function should only treat the region marked by the variables as an array and that is how we will break the array. So, we will use two variables 'start' and 'end' to represent an array. Even if the array is bigger than whatever is enclosed in 'start' and 'end', inside the function, only this region will be treated as the array.



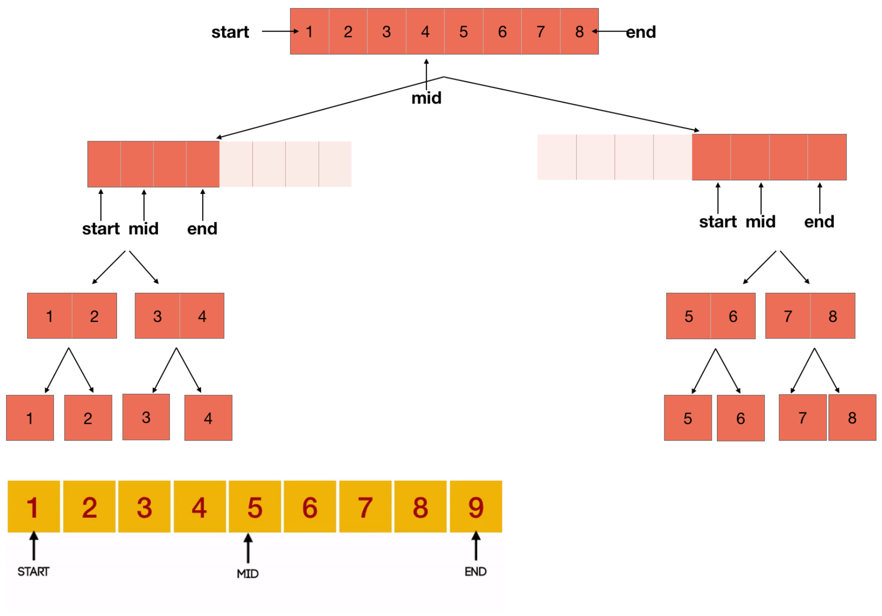
So, let's start writing the function - `DEMO(A, start, end)` → 'DEMO' is the name of the function, 'A' is the array passed to it and 'start' and 'end' are the starting and the ending indices respectively.

We will break the array from the middle element but this is always not the case. For example, some algorithms require to break the array from a specific pivot point like the median, a random element, etc. So, we just need to calculate the middle element i.e., $\lfloor (start + end)/2 \rfloor$.

Now, we want the array to be broken from this middle element. To do this, we will simply pass the array to the function again but this time once we will pass array with starting index 'start' and ending index 'middle' and again with the starting index 'middle+1' and ending index 'end' i.e., `DEMO(A, start, middle)` and `DEMO(A, middle+1, end)`.



So, we have broken our array and this function will again break the array into even smaller subarrays.



Now, this should stop when the subarray reaches the size of 1, and in that case, the 'start' and the 'end' will point to the same element. So, we will call `DEMO(A, start, middle)` and `DEMO(A, middle+1, end)` only when the 'start' is less than the 'right' because if they are equal then the array has only one element in it and there is no need of further breaking it.

```
DEMO(A, start, end)
  if start < right
    middle = floor((start+end)/2)
    DEMO(A, start, middle)
    DEMO(A, middle+1, end)
```

And thus, we are done with the dividing part. Now let's focus on the conquer part, and here we have to increase the value of each element by 1. We can easily do this after reaching the base case i.e. when 'start' is not less than 'end', we will increase the value of the element by 1 `A[start] = A[start]+1`.

```
DEMO(A, start, end)
  if start < right
    ...
  else
    A[start] = A[start]+1
```

```
DEMO(A, start, end)
  if start < right
    middle = floor((start+end)/2)
    DEMO(A, start, middle)
    DEMO(A, middle+1, end)
  else
    A[start] = A[start]+1
```

**C** **Python** **Java**

```c
#include <stdio.h>

void demo(int a[], int start, int end) {
  int middle;

  if(start < end) {
    middle = (start+end)/2;
    demo(a, start, middle);
    demo(a, middle+1, end);
  }
  else {
    a[start] = a[start]+1;
  }
}

int main() {
  int a[] = {4, 8, 1, 3, 10, 9, 2, 11, 5, 6};
  demo(a, 0, 9);

  //printing array
  int i;
  for(i=0; i<10; i++) {
    printf("%d ",a[i]);
  }
  printf("\n");
  return 0;
}
```

That's it. This is how we implement Divide and Conquer and the basic idea behind it. Let's analyze the above code.

## Analysis

Let's recall the functions whose analysis were done in the Recurrences chapter, isn't it similar to one of those functions? Actually, it is. Let's implement what we learned in the Recurrence chapter.

The base case is when the size of the input is 1, in that case, the statement of `else` will be executed and will take constant time. So, the running time is $\Theta(1)$ when $n = 1$. Otherwise, the problem is broken down into two smaller subproblems i.e., $2T\left(\frac{n}{2}\right)$. Rest of the statements will take constant time i.e., $\Theta(1)$.

So, the running time will be:

$$T(n) = \begin{cases} c_1, & n = 1 \text{ (start = end)} \\ 2T\left(\frac{n}{2}\right) + c_2, & \text{if } n > 1 \end{cases}$$

Applying Master's theorem, $a = 2$, $b = 2$, $n^{\log_b a} = n^{\log_2 2} = n$ and $f(n) = c_2$ (a constant).

By case 1 of the Master's theorem, $T(n) = \Theta(n^{\log_b a}) = \Theta(n)$.

And it seems logical also, we have visited each element of the array once and thus, we should be able to do this in a linear time i.e., $\Theta(n)$.

We have seen how we basically implement divide part of the Divide and Conquer in our program and then solve the smaller problems. Sometimes, we require to proceed one step further and combine these solutions of the subproblems in a particular way to get the solution of the original problem. We are going to this in the next chapter i.e., Merge Sort (/course/algorithms-merge-sort/). Now, we will focus only on the concepts of the Merge sort because we already have a clear idea of the implementation of the Divide and Conquer. So, let's move to the next chapter.

“ Winning isn't everything--but wanting to win is. ”

- Vince Lombardi

PREV (/course/algorithms-heapsort/) (/course/algorithms-merge-sort/) NEXT

## # Further Readings

- Divide and Conquer (/blog/article/divide-and-conquer/)
- Maximum Subarray Sum Using Divide and Conquer (/blog/article/maximum-subarray-sum-using-divide-and-conquer/)

**New Questions**

setting up an ide for mac.. - Cpp

(/discussion/setting-up-an-ide-for-mac)

Please fill the blanks and help me am stuck - Java

(/discussion/fill-the-blanks-and-help-me-am-stuck)