Recurrences

As told in the previous chapter, this chapter is going to be about the analysis of recursive algorithms. To analyze an algorithm, we first need to derive the cost function (growth function) for that algorithm as we did in the chapter Analyze Your Algorithm (/course/algorithms-analyze-your-algorithm/) with the iterative algorithms and then we solve it. So, it is a two-step process:

i. We first derive a recurrence equation of our algorithm.
ii. Then we solve the equation to get the order of growth of the algorithm.

You will be understand the first step after going through few examples.

The second step is to solve the recurrence equation and we are going to study 3 different methods in this course to do so:

i. Iteration Method
ii. Recursion Tree Method
iii. Master's Theorem

Deriving the Recurrence Equation

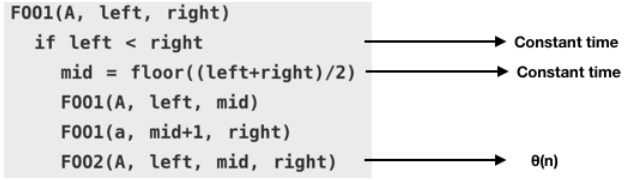
So, let's start with the first step and try to form a recurrence equation of the algorithm given below.

```
FOO1(A, left, right)
  if left < right
    mid = floor((left+right)/2)
    FOO1(A, left, mid)
    FOO1(a, mid+1, right)
    FOO2(A, left, mid, right)
```

</>
If the above code doesn't seem familiar, don't worry, we are going to explain each and every line of it.

`FOO1` is a function which takes an array 'A', its starting index i.e., 'left' and the last index i.e., 'right'.

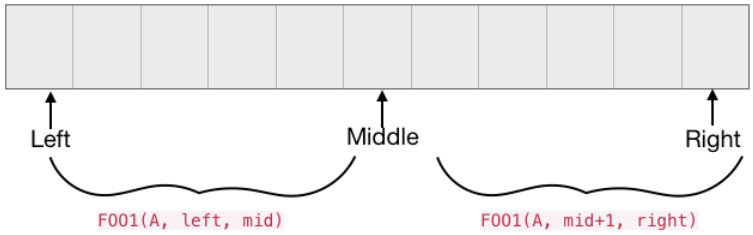
Let's first take the case when 'left' is less than 'right' i.e., `left<right`, the calculation of 'mid' is going to take a constant amount of time and not going to depend upon the input size (According to the RAM model, arithmetic operations takes a constant amount of time). So, its time will be $\Theta(1)$. And the variable 'mid' is basically the middle element of the array 'A' indexed from 'left' to 'right'. Also, the checking of the condition i.e., `if left < right` will take constant time.



</>
'floor' is the floor function. So, floor(2.2) will return 2, floor (3.8) will return 3, etc.

Before proceeding further, let say our function `FOO1` is going to take $T(n)$ amount of time for the input size 'n' (number of elements in the array A, in this case) i.e., $T(n)$ is the running time of our function `FOO1`. So, our task is to basically derive this $T(n)$ function and that will be our growth function.

Now, there are two recursive calls of `FOO1` - `FOO1(A, left, mid)` and `FOO1(a, mid+1, right)`. Take note that these recursive calls are made on the half of the size of the array 'A'.



So, the first call `FOO1(A, left, mid)` is made on the 'left' half (indexed from 'left' to 'mid') and the second call `FOO1(a, mid+1, right)` is made on the right half (indexed from 'mid+1' to 'right'). So, each of these calls are going to take $T(\frac{n}{2})$ time because the input size is half and we have assumed that `FOO1` takes $T(n)$ time for the input size of 'n'. Thus the total time taken for the execution of both the functions would be $2T(\frac{n}{2})$.

```
FOO1(A, left, right)
  if left < right                    → Constant time
    mid = floor((left+right)/2)      → Constant time
    FOO1(A, left, mid)               → T(n/2)
    FOO1(a, mid+1, right)            → T(n/2)
    FOO2(A, left, mid, right)        → θ(n)
```

There is one more line - `FOO2(A, left, mid, right)`. For now, let's assume `FOO2` is any arbitrary function which takes $\Theta(n)$ time. There can be many functions which take $\Theta(n)$ time. For example, let's take a function to print all the elements of an array.

	Cost	Times
PRINT(A)		
for i in 1 to A.length	c_1	n+1
print(A[i])	c_2	n

From the previous knowledge of analysis of iterative algorithms, one can easily find out that this takes $\Theta(n)$ time $(c_1(n+2)+c_2n => n(c_1+c_2)+2c_1)$.

Similar to this, take `FOO2` to be some arbitrary function which takes $\Theta(n)$ time.

Thus, the total time taken by the function for the case `left<right` can be written as:

$$T(n) = \Theta(1) + 2T\left(\frac{n}{2}\right) + \Theta(n)$$

And for the case `left >= right`, only the condition check will occur i.e., `if left < right` and thus the function will complete in $\Theta(1)$ time.

So, we can write $T(n)$ as:

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

And this is the recurrence equation we were talking about and now the second step is to solve this equation and obtain the order of growth.

</>
We are using arbitrary codes to explain the concepts. However, these are not just random algorithms but extracted from some common algorithms which you are going to study further.

Before going further to learn how to solve this recurrence equation, let's look at one more example of making the recurrence equation.

```
FOO(A, low, high, x)
  if (low > high)
    return False
  mid = floor((high+low)/2)
  if (x == A[mid])
    return True
  if (x < A[mid])
    return FOO(A, low, mid-1, x)
  if (x > A[mid])
    return FOO(A, mid+1, high, x)
```

This code is an algorithm called Binary Search. As stated earlier, these algorithms will be discussed later in this course, so let's just focus on deriving the recurrence equation of the algorithm.

This example is also similar to the previous one. The condition checking like `if (low > high)`, `if (x == A[mid])`, etc are going to take constant time. Also, the arithmetic operation (`mid = floor((high+low)/2)`) is going to take constant time. And we are calling the `FOO` function on the half of the array but in this example, **either `FOO(A, low, mid-1, x)` or `return FOO(A, mid+1, high, x)` will be executed**.

Thus, if $T(n)$ is the running time of the `FOO` function with the input size 'n', then the execution of `FOO(A, low, mid-1, x)` and `return FOO(A, mid+1, high, x)` is going to take $T\left(\frac{n}{2}\right)$ (it was $2T\left(\frac{n}{2}\right)$ in the previous example as both the recursive functions were executed).

Thus, the running time of this algorithm can be written as:

$$T(n) = \begin{cases} \Theta(1), & n = 1 \ (\text{low} = \text{high}) \\ T\left(\frac{n}{2}\right) + \Theta(1), & \text{if } n > 1 \end{cases}$$

Now, we know how to make a recurrence equation, so we are now left to learn to solve these equations to obtain the order of the growth. As mentioned above, we are going to learn three methods to solve recurrence equations.

i. Iteration Method
ii. Recursion Tree Method
iii. Master's Theorem

So, let's proceed to the next chapter to learn these methods.

> “ To understand recursion, one must first understand recursion. ”

PREV (/course/algorithms-analyze-iterative-algorithms/) (/course/algorithms-lets-iterate/) NEXT

Download Our App.