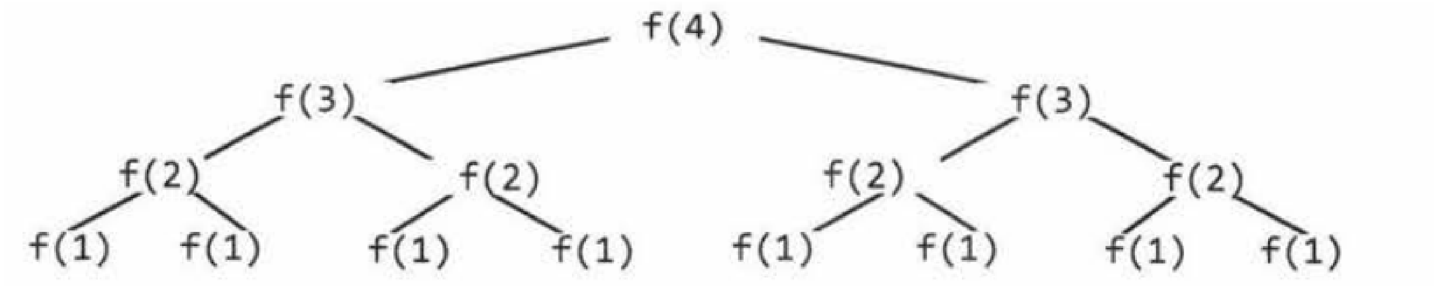
Time complexity resume:
big 0 describes an upper bound on the time.
-common runtimes:
> O(1) , constant
> O(log n), expected cases
> O(n), linear time , best case
> O(n log n)
> O(n^2) , worst cases
> O(2^n)
> O(n!)

TIPS
*If your algorithm is in the form "do this, then, when you're all done, do that" then you add the runtimes.
*If your algorithm is in the form "do this for each time you do that" then you multiply the runtimes.
* When you see a problem where the number of elements in the problem space gets halved each time, that will likely be a O( log base 2 de N) runtime. La base depende de por cuanto se divida.

$2^4 = 16 \rightarrow \log_2 16 = 4$
$\log_2 N = k \rightarrow 2^k = N$

* When you have a recursive function that makes multiple calls, the runtime will often (but not always) look like O( branches^depth ), where branches is the number of times each recursive call branches. In this case, this gives us 0 (2N) .




| Level | # Nodes | Also expressed as... | Or... |
|---|---|---|---|
| 0 | 1 | | $2^0$ |
| 1 | 2 | 2 * previous level = 2 | $2^1$ |
| 2 | 4 | 2 * previous level = $2 * 2^1 = 2^2$ | $2^2$ |
| 3 | 8 | 2 * previous level = $2 * 2^2 = 2^3$ | $2^3$ |
| 4 | 16 | 2 * previous level = $2 * 2^3 = 2^4$ | $2^4$ |

*Binary search is O( log n)
We commonly see O( log N) in runtimes. Where does this come from? Let's look at binary search as an example. In binary search, we are looking for an example x in an N-element sorted array. We first compare x to the midpoint of the array. If x == middle, then we return. If x < middle, then we search on the left side of the array. If x > middle, then we search on the right side of the array.

search 9 within {1, 5, 8, 9, 11, 13, 15, 19, 21}
compare 9 to 11 -> smaller.
search 9 within {1, 5, 8, 9, 11}
compare 9 to 8 -> bigger
search 9 within {9, 11}
compare 9 to 9
return

*Suma de los primeros N numeros :

$$(N-1) + (N-2) + (N-3) + \ldots + 2 + 1$$

$$= 1 + 2 + 3 + \ldots + N-1$$
$$= \text{sum of 1 through } N-1$$

The sum of 1 through N-1 is $\frac{N(N-1)}{2}$

Que termina siendo O (n^2)

*Si recorremos dos arrays distintos en un for anidado, la complejidad del numero de operaciones es O(ab) no O(n^2), ya que no recorren el mismo array.