I) Big O

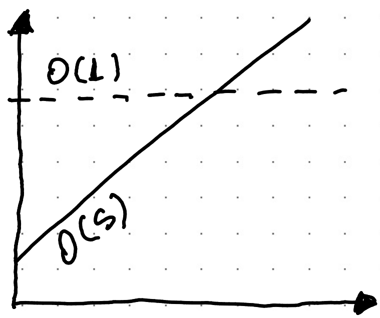
Big O time = language / metric we use to define algorithm efficiency

Time complexity / asymptotic runtime

eg. data transfer algorithm

- Electronic transfer : $O(s)$, s = size of file
- Airplane transfer : $O(1)$ - time is constant

* No matter how big the constant is or how slow the linear increase → linear will at one point surpass constant



! common asymptotic runtimes :

$O(\log n)$, $O(N \log N)$

$O(n)$, $O(n^2)$

$O(2^n)$

* multiple variables in your runtime

eg. time to paint a fence w - wide, h - high

$\boxed{O(wh)}$, if p = no. of layers

=> $O(whp)$

Big O, Big Theta, Big Omega

slowest
- O (big O) = upper bound on the time / at least as fast as described (<=) less than equal - not useful but you can describe an alg. with a higher Big O
- Ω (big omega) = lower bound on time (>=)

fastest
- Θ (big theta) = both O and Ω

if an algorithm is $\Theta(n)$ => it's both $O(n)$ and $\Omega(n)$ → tight bound on runtime

* industry : $\Theta \simeq O$ => tightest bound

Best case, worst case and expected case

eg. Quick Sort - pick random element as a pivot, swaps values in the array => elements less than the pivot appear before elements greater than pivot → partial sort => recursively sorts the left and right side using a similar process

- Best case : if elements are equal => QS will traverse the array only once => $\boxed{O(n)}$

- Worst case : $O(n^2)$
- Expected case : $O(n \log n)$

$\log_{10}(1000) = 3$

$10^3 = 1000$

$$\boxed{\begin{array}{l} \log_b(x) = e \\ b^e = x \end{array}}$$

Space Complexity

- time is not the only thing that matters in an algorithm - memory (or space)
- space complexity is a parallel concept to time complexity

eg. if we need to create an array of size n this will require $O(n)$ space.

eg. two dimensional array of size $n \times n \Rightarrow O(n^2)$

Drop the constants

- it's possible for $O(n)$ code to run faster than $O(1)$ code for specific big inputs.
- Big O describes the rate of increase - we drop constants in runtime $O(2n) \rightarrow O(n)$

eg. 2 non nested for loops $\rightarrow \underline{O(2n)}$

Drop the Non-Dominant Terms

$O(N^2 + N)$ => N is not a constant

eg. $O(N^2 + N^2) = O(2N^2) = O(N^2)$

drop constants

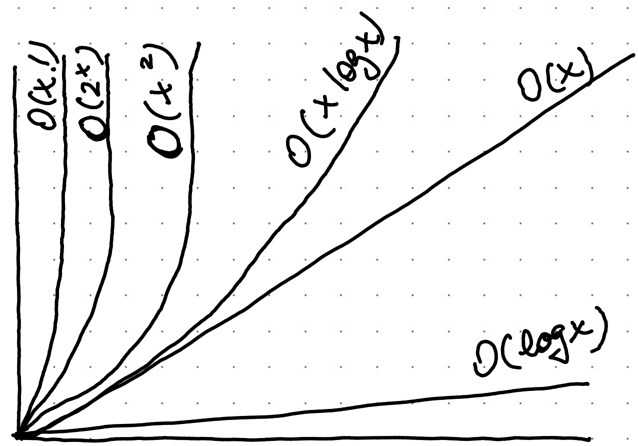
* you should drop the non dominant terms

$O(N^2 + N) \longrightarrow O(N^2)$

$O(N + \log N) \longrightarrow O(N)$

$O(5 \cdot 2^N + 1000N^{100}) \longrightarrow O(2^N)$

eg. rate of increase for some of the common big O times.



$O(x^x)$ or $O(2^x \cdot x!)$ are worse than $\boxed{O(x!)}$

Multi-part algorithms: Add vs. Multiply

- if you have an algorithm with 2 steps: when do you multiply and when do you add the runtime?

Amortized Time

ArrayList - dinamically resizing array

- is implemented using an array: when the array hits capacity → ArrayList class will create a new array with double the capacity and copy the elements.

- The runtime of insertion? What if the array is full?
  - $O(N)$ if its full
  - $O(1)$ most of the times

Amortized Time - worst case happens once in a while. Once it happened, it won't happen again for so long that the cost is amortized.

$$\underbrace{\boxed{x + x/2 + x/4 + \dots + 1}}_{\simeq x} \simeq 2x$$

⇒ x insertions takes $O(2X)$, each ins. $O(1)$

eg $a = [1]$ we add 4 elements $O(8)$

step I: add 2 $\Rightarrow$ a doubles in size | 3

$a = [\_, \_]$ copy 1, add 2 $\Rightarrow a = [1, 2]$

step II: add 3 $\Rightarrow$ a doubles in size | 4

$a = [\_, \_, \_, \_]$
1 2 3

step III add 4 $\Rightarrow$ a is same size | 1

$a = [1, 2, 3, 4]$

---

step IV add 5 $\Rightarrow$ a doubles in size | 6

$a = [\_, \_, \_, \_, \_, \_, \_, \_]$
1 2 3 4 5

$O(1) \longleftrightarrow O(N)$

Log N Runtimes

$\boxed{O(\log N)}$? eg. Binary Search

- we start with N- element array

after the first step $N/2$, ... $N/\ldots$, 1

- The total runtime is the number of steps we can take until N becomes 1

$$\underbrace{1, 2}_{\times 2}, \underbrace{2, 4}_{\times 2}, \underbrace{4, 8}_{\times 2} \dots N$$

$$\boxed{2^k = N}$$

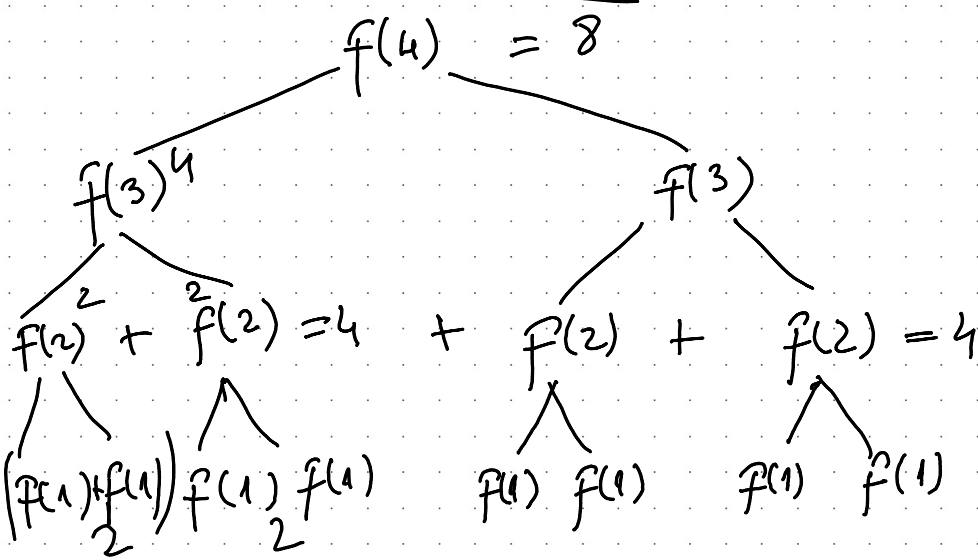
$$\boxed{\log_2 N = k}$$

if $\boxed{N = 16} \Rightarrow k = 4$ = steps

* if you see a problem where the number of elements gets halved each time => it will likely be a runtime of $O(\log N)$

* the base of the log doesn't matter

Recursive Runtimes



- the tree will have a depth of N; each node has two children

Level	Nodes
0	1
1	2
2	4
3	8
4	16
⋮	

$$\Rightarrow \boxed{2^0 + 2^1 + 2^2 + 2^3 + \dots 2^N}$$

$$= \boxed{2^{N+1} - 1}$$

nodes

* a recursive function with multiple calls

$$\boxed{O(\text{branches}^{\text{depth}})} \Rightarrow O(2^N)$$

* log doesn't matter for big O but the base of an exponent does matter.

- space complexity is $O(N)$ - only $O(N)$ exist at any given time on the call stack

---

<u>Geometric series</u>

$$\sum_{k=1}^{n} a r^{k-1} = a r^0 + a r^1 + a r^2 + \dots + a r^{n-1}$$

$$\boxed{\sum_{k=1}^{n} a \cdot r^{k-1} = \frac{a(1-r^n)}{1-r}}$$