

BIG O NOTATION

We say $f(n)$ is Big O of $g(n)$

If there is some constant c such that $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \leq c$

example

$$\begin{array}{l} f(n) = n \\ g(n) = n^2 \end{array} \quad \lim_{n \rightarrow \infty} \frac{n}{n^2} = 0 \Rightarrow n = O(n^2)$$

$$\begin{array}{l} f(n) = n^3 \\ g(n) = n^2 \end{array} \quad \lim_{n \rightarrow \infty} \frac{n^3}{n^2} = \infty \quad n^3 \neq O(n^2)$$

$$\begin{array}{l} 3n^2 + 5n + 1 \\ 2n^2 + 10n + 10 \end{array} \Rightarrow \text{Big O of each other}$$

Big O classifies functions by their fastest growing term. For large n , constants and lower order terms do not affect the growth class.

Ω - Big Omega

O = asymptotic upper bound
 Ω = asymptotic lower bound

$$f(n) = \Omega(g(n)) \text{ if}$$

$$g(n) = O(f(n))$$

Where " O " is an upper-bound, " Ω " is a lower bound

$$n^2 = \Omega(n)$$

Θ ~ "Big Theta"

$$f(n) = \Theta(g(n)) \text{ if } f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n))$$

$\Theta \sim$ an asymptotic " $=$ " sign

Industry Question

- * What is the worst case runtime of a given algorithm
- * What is the average-case runtime of a given algorithm
- * Is this algorithm correct?
- * What is the best case runtime of a given algorithm.
- * Is there an algorithm that correctly solves the problem.
- * What algorithm guarantees the worst case runtime for a given problem.
- * How hard is this problem.

Correctness is the most important question: an algorithm is correct only if it produces the desired output for every valid input

What ~~does~~ it mean for an algorithm to be correct

- It works on every single valid input

$$1 + 2 + 3 + \dots + (n-1) + n = \frac{n(n+1)}{2}$$

Base case: $n=1$

$$1 = \frac{1(1+1)}{2}$$

PROVE BY INDUCTION

Inductive step:

- Assume I know the formula holds for some particular value n .
- Now show it works for $n+1$

1. Base case: show $P(1)$ is true
2. Inductive hypothesis: Assume $P(n)$ is true
3. Inductive step: Using that assumption, prove $P(n+1)$ is true.

Base case gives us a starting point.

$P(n) \Rightarrow P(n+1)$ lets us continue forever.

$$P(1) \Rightarrow P(2) \Rightarrow P(3) \Rightarrow \dots$$

$$[1+2+3+\dots+(n-1)+n] + (n+1)$$

$$= \frac{n(n+1)}{2} + n+1$$

$$= \frac{n^2+n+2n+2}{2}$$

$$= \frac{n^2+3n+2}{2}$$

$$= \frac{(n+1)(n+2)}{2}$$

$$1+2+4+8+\dots+2^{n-1} = 2^n - 1$$

Base case ($n=1$)

$$2^{1-1} = 2^1 - 1$$

$$1 = 1$$

Induction step (show that it works for $n+1$)

$$[1+2+\dots+2^{n-1}] + 2^n$$

$$2^n - 1 + 2^n$$

$$2 \cdot 2^n - 1$$

$$= 2^{n+1} - 1$$

Induction sort (Input $A[1:n+1]$)

For $i = 1, \dots, n$

$j = i$

while $j > 1$ and $A[j] < A[j-1]$

swap $A[j]$ and $A[j-1]$

After the iteration i , the first i elements are sorted

Base case: $i = 1$

On iteration $i = 1$, nothing happens in the loop, but first "one" element is sorted

Induction step: If it worked for the i th iteration, show it it works on the $(i+1)$ th iteration.

Induction hypothesis: Assume after iteration i , the first i elements are sorted.

[sorted] [x] [remaining]

On iteration $i+1$, insertion sort moves the new element x left until it reaches its correct position among the already-sorted first i elements.

The original i elements remain in sorted order, and x is inserted in the correct position

On the final iteration, $i = n$. Therefore, the first n elements (the entire array) are sorted.

