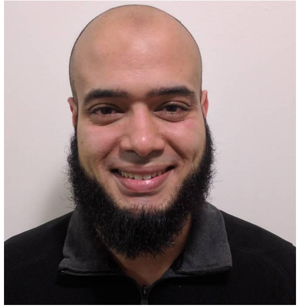
Data Structures

Asymptotic Complexity 1

Mostafa S. Ibrahim
Teaching, Training and Coaching since more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)



How much time? How much memory?

- Some services (e.g. from Google) are very fast, but many are slow
- Some of your computer programs consume your memory
- Our code consumes time and memory!
- When we develop our code and run it, it will take some time
- But how can we **estimate** how good our code might be in terms of time and memory?
- We might think of different ways to do so
  - E.g. run the function and compute total time, but this is hardware dependent
- **Asymptotic Complexity**: a field that answers this question
- During DSA courses, we need to learn how efficient are our codes

My educational approach

- The formal introduction to this field involves many abstract concepts and mathematics
- Initially, many students find this intimidating and inconvenient - or even incomprehensible!
- I prefer an informal treatment
  - Focusing exclusively on a subset that are commonly used in the real world (i.e. in industry)
  - Gaining **incremental** experience in computing the complexity
- This will make it much easier when you deal with the logic behind it

How many steps **approximately**?

- Let's pretend these approximations are valid: 9n+4, where n is list's size
- It takes a **linear number** of steps!

```python
def compute(numbers_lst, f1, f2):                # 9n+4
    for idx in range(len(numbers_lst)):          # 6n+1
        x = f1 - 1
        numbers_lst[idx] += x * (idx+1)

    sum = 0                                  # 1
    for number in numbers_lst:               # 3n+1
        sum += number * f2

    return sum  # 1
```

With large N

- Let's say we have function f(n)
- We computed the exact number of steps: n + 19
- Think in terms of **large** n = 10^9 (million)
  - Does it matter if it is 1,000,000,000 steps or 1,000,000,019 steps? Clearly not!
- It is more **intuitive** to just think about it in terms of n steps
- What if it is 5n? Again 1 billion steps vs 5 billion steps is not that significant
  - Both are extremely slow
- With a **large** N, we actually don't care about these factors
  - All of the following are close, time-wise: n,   10n+19,   13n+20,   n+1000
  - In all of them, the code takes a **linear number** of steps

Big O notation

- If a code takes **9n+17 steps**, we say it is O(n) code
    - Order of N
    - O is capital. *There is another o small*
- It means the code runs in **linear time** relative to n

Big O notation: Guidelines

- Remember, we always think in terms of a very **large** n
- Practically, the **largest** term in the **equation** is the one that dominates
  - All others are **negligible** with a large n with large N
- Assume your code takes the following number of steps:
  - 5n^2 + 10n ⇒ O(n^2). So we selected the **largest term** (n^2) as it will dominate
  - n * (n+1) / 2 ⇒ O(n^2), Again, expand the expression to find the **biggest**
  - 2n^4 + 5n^3 + n + 9 ⇒ O(n^4): Again the **largest**
  - 17 ⇒ O(1): We say this is constant time (**largest** is n^0).
    - It always confuses students.
    - If an algorithm takes 10^6 fixed steps, it is again O(1). We don’t have this in practice
    - Observe: what matters is n, as it affects total steps for a LARGE n

How can skilled programmers find the order
VERY quickly?

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."