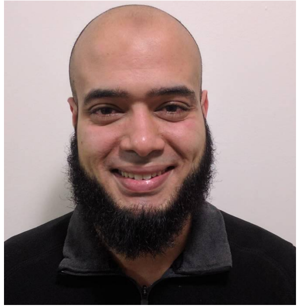
Data Structures

Asymptotic Complexity 3

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)



Big O notation: Little math

- Assume your code takes: 9N+17 steps ⇒ Order O(N)
- There is some **constant C** where for any input size **N** ⇒ 9N + 17 < CN
- Actually O(n) means there is some constant multiplied in this n
  - For example, let C = 30
  - Then 9N + 17 < 10N for **ANY** N
- What does this imply?
- Big O is an **Upper limit** to the number of steps regardless these constants and factors in 9N+17
  - So 30N is **always bigger** than 9N + 17. So It is O(n)

Big O notation: an upper bound

- Assume we have function F(N).
- Its total number of steps T(N) = N + 2N + 5N^2
    - Clearly T(N) is O(N^2), but what is proper C?
    - Let’s try C = 6. This means T(N) < 6N^2 for any N   ?

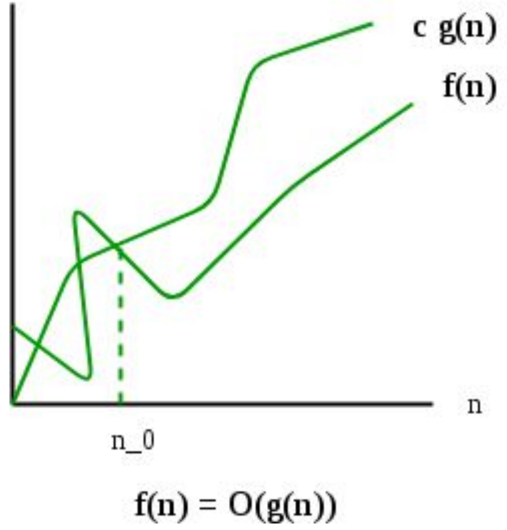
	T(N) = N + 2N + 5N^2	6N^2
N = 1	1 + 2 x 1 + 5 x 1 x 1 = 8	6 x 1 x 1 = 6 ⇒ 8 < 6? NO
N = 2	2 + 2 x 2 + 5 x 2 x 2 = 26	6 * 2 * 2 = 24 ⇒ 26 < 24? NO
N = 3	3 + 2 x 3 + 5 x 3 x 3 = 54	6 x 3 x 3 = 54 ⇒ 54 < 54? No
N = 4	4 + 2 x 4 + 5 x 4 x 4 = 92	6 x 4 x 4 = 96 ⇒ 92 < 96? YES
N = 5	5 + 2 x 5 + 5 x 5 x 5 = 140	6 x 5 x 5 = 150 ⇒ YES

Big O notation: an upper bound

- In the previous table, N = 1, 2, 3 our C was not good
- But starting from 4, always T(N) < 6 N^2
- Let's state O() more **formally**
  - T(N) is O(G(N)) IFF we could find:
    - n0 < N
    - Constant C such that T(N) < C * F(N) for any N > n0
  - In our case:
    - T(N) = N + 2N + 5N^2 ⇒ Total number of steps
    - G(N) = N^2 ⇒ Our guessed order O(N^2)
    - **n0 = 3** ⇒ The starting point
    - **C = 6** ⇒ The constant

Big O notation: an upper bound

- As you see, starting from some point n0
- Our order function g(n) is always higher than f(n) with a specific C
  - So it is an **upper** function
- Note if some C is working well, any higher also
  - E.g. previously C = 6 is good
  - Then C = 7, 8, 9 and higher are good too!
- Note if g(n) is good, then higher is good
  - E.g. previously, it is O(n^2)
  - Then O(N^3) and O(N^4) and higher are good too
  - But we use the **tightest** one



Img src

Enough math

- In practice
    - We don’t compute or care a lot about this X
    - Just follow last lecture tips to compute the order like a pro!
- C idea is cool to understand order is an **upper function**
- If you did not understand the previous slides well = totally ok
    - Skip and repeat by the end of the course

Same order

- Consider the 2 functions f1 and f2
- Both of them are O(n^3)
- This means they **grow cubic** in time, which is too much!
- But in practice, does they take the same amount of time?

```
void f1(int n = 1000) {     // O(n^3)
    int cnt = 0;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < n; ++k)
                cnt++;
}

void f2(int n = 1000) {     // O(n^3)
    int cnt = 0;
    for (int i = 0; i < n; ++i)
        for (int j = i; j < n; ++j)
            for (int k = j; k < n; ++k)
                cnt++;
}
```

Same order

- In terms of operations:
  - For n = 1000
  - F1 = 1000,000,000 * some c
  - F2 = 167,167000 * some c
- The moral of that
  - We can have 2 code of the same order, e.g. O(n^3)
  - But still one of them is faster
  - E.g. C1 = 7 but C2 = 2
  - Smaller constant ⇒ faster
- Tip: build code with small C :)

```
void f1(int n = 1000) {      // O(n^3)
    int cnt = 0;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j)
            for (int k = 0; k < n; ++k)
                cnt++;
}

void f2(int n = 1000) {      // O(n^3)
    int cnt = 0;
    for (int i = 0; i < n; ++i)
        for (int j = i; j < n; ++j)
            for (int k = j; k < n; ++k)
                cnt++;
}
```

Worst-Case and Average-Case Analysis

- Sometimes total number of steps of f() **varies differently** based on input
- O() is intended to be an **upper bound**. In other words considers **worst case**
  - That is why we find the **largest term** and use it
  - This is perfect most of the time
  - But sometimes is **misleading** (partially like previous slide)
- Another type is called: Average-Case Analysis
  - This one computed the expected order. It involves **probability** and consider **different cases**
  - It is usually **harder** analysis
  - Sometimes we need it because the O() is actually much bigger than actual performance
    - An example for that is *Quick sort algorithm*
  - There is also best-case analysis, but less useful

“Acquire knowledge and impart it to the people.”

“Seek knowledge from the Cradle to the Grave.”