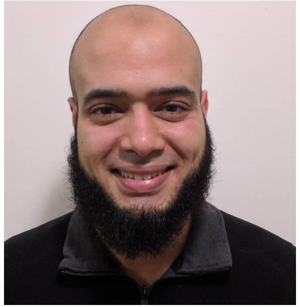
# Data Structures

# Asymptotic Complexity 4

**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)



# Worst-Case Analysis

- Sometimes, the total number of steps in a function f() can vary **significantly** based on the input
- O() is intended to be an **upper bound**.
- In other words; focusing on the **worst case**
    - That is why we find the **largest term,** C, and use it
    - This is perfect most of the time
    - But it can be **misleading**

# Average-Case Analysis

- This type computes the expected order.
- It involves **probability** and considers **different cases**
  - It is usually harder to analyze
- Sometimes we need it because the O() is actually much bigger than the actual performance
  - An example for that is *Quick sort algorithm*
- There is also the **best-case analysis**, but this is less useful
  - Imagine an algorithm that may do O(n) or O(n^3) depending on the conditions
  - Its best case is O(n)

# Amortized Analysis

- What is the time complexity of append?
- It depends on how Python is implemented behind the scenes
- Let’s assume **CPython**

```
def f1(n):        # O(n)
    lst = []
    for i in range(n):
        lst.append([i])  # O(1)


def f2(n):        # O(n)
    lst = [0] * n
    for i in range(n):
        lst[i] = i
```

# Amortized Analysis

- What is time complexity of that append function with the capacity trick?
    - Seems for some N steps, we have O(1)
    - Then suddenly we do O(N) steps **for a single append**
    - And so on
    - Most of the steps are fast, and single step will be slow - and this pattern repeats
    - If you think from the average perspective, this means after N steps, we have O(N) in total
    - This is what we call amortized analysis
    - Let's avoid more details beyond that
- Take home message
    - Consider this thinking when operations vary
    - C/C++ helps you understand the complexity/design in more detail
    - It is important to know the time complexity of built-in data-structures

# List

| Operation | Average Case | Amortized Worst Case |
| --- | --- | --- |
| Copy | O(n) | O(n) |
| Append[1] | O(1) | O(1) |
| Pop last | O(1) | O(1) |
| Pop intermediate[2] | O(n) | O(n) |
| Insert | O(n) | O(n) |
| Get Item | O(1) | O(1) |
| Set Item | O(1) | O(1) |
| Delete Item | O(n) | O(n) |
| Iteration | O(n) | O(n) |
| Get Slice | O(k) | O(k) |
| Del Slice | O(n) | O(n) |
| Set Slice | O(k+n) | O(k+n) |
| Extend[1] | O(k) | O(k) |
| Sort | O(n log n) | O(n log n) |
| Multiply | O(nk) | O(nk) |
| x in s | O(n) | |
| min(s), max(s) | O(n) | |
| Get Length | O(1) | O(1) |

# collections.deque

- Copied from wiki
- A deque (**double-ended queue**) is represented internally as a doubly linked list.
    - Well, a list of arrays rather than objects, for greater efficiency.
- Both ends are accessible, but even *looking at* the middle is slow, and adding to or removing from the middle is slower still.

| Operation | Average Case | Amortized Worst Case |
|---|---|---|
| Copy | O(n) | O(n) |
| append | O(1) | O(1) |
| appendleft | O(1) | O(1) |
| pop | O(1) | O(1) |
| popleft | O(1) | O(1) |
| extend | O(k) | O(k) |
| extendleft | O(k) | O(k) |
| rotate | O(k) | O(k) |
| remove | O(n) | O(n) |

# Set

| Operation | Average case | Worst Case | notes |
|---|---|---|---|
| x in s | O(1) | O(n) | |
| Union s\|t | O(len(s)+len(t)) | | |
| Intersection s&t | O(min(len(s), len(t)) | O(len(s) * len(t)) | replace "min" with "max" if t is not a set |
| Multiple intersection s1&s2&..&sn | | (n-1)*O(l) where l is max(len(s1),..,len(sn)) | |
| Difference s-t | O(len(s)) | | |
| s.difference_update(t) | O(len(t)) | | |
| Symmetric Difference s^t | O(len(s)) | O(len(s) * len(t)) | |
| s.symmetric_difference_update(t) | O(len(t)) | O(len(t) * len(s)) | |

# Dict

| Operation | Average case | Worst Case | notes |
|---|---|---|---|
| x in s | O(1) | O(n) | |
| Union s\|t | O(len(s)+len(t)) | | |
| Intersection s&t | O(min(len(s), len(t)) | O(len(s) * len(t)) | replace "min" with "max" if t is not a set |
| Multiple intersection s1&s2&..&sn | | (n-1)*O(l) where l is max(len(s1),..,len(sn)) | |
| Difference s-t | O(len(s)) | | |
| s.difference_update(t) | O(len(t)) | | |
| Symmetric Difference s^t | O(len(s)) | O(len(s) * len(t)) | |
| s.symmetric_difference_update(t) | O(len(t)) | O(len(t) * len(s)) | |

# Don’t worry

- During the course, we will study how to implement hash tables (used for set/dict) and linked lists (used for deque)
- Then you will easily realize the reason for these time complexities
- For the list, you have to think in terms of arrays (fixed)
  - For example, to delete an element we can either:
  - 1) Create a new array and copy the old data, which is O(n)
  - 2) Or shift to the right all the elements after the deleted element
    - [1, 2, 3, 5, 6, 7, 8, 9]. Delete 6. Shift right [7, 8, 9]
    - It's still O(n), O(n) but more efficient
  - The same logic for insert
    - Shift the subarray left (if the subarray is selected specifically) then add the value ⇒ O(n)

# Tip

- Python order confuses beginners, as many things happens behind the scenes
- Do your best to guess how something is implemented
- If number in list:
    - print(‘found’)
- This looks like an uncomplicated piece of code, but behind the scenes, Python will loop through the whole list
    - Which is O(n)
- C++/Java programmers are not tricked, as they see all of the operations

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."