

Properties of Algorithms

- **Correctness** (no mistakes)
- **Completeness** (no blind spots)
- **Efficiency** (no time wasting)

Computational complexity

In computer science, the computational complexity or simply complexity of an algorithm is the amount of resources required to run it. Particular focus is given to computation time (generally measured by the number of needed elementary operations) and memory storage requirements. The complexity of a problem is the complexity of the best algorithms that allow solving the problem.

Computational complexity

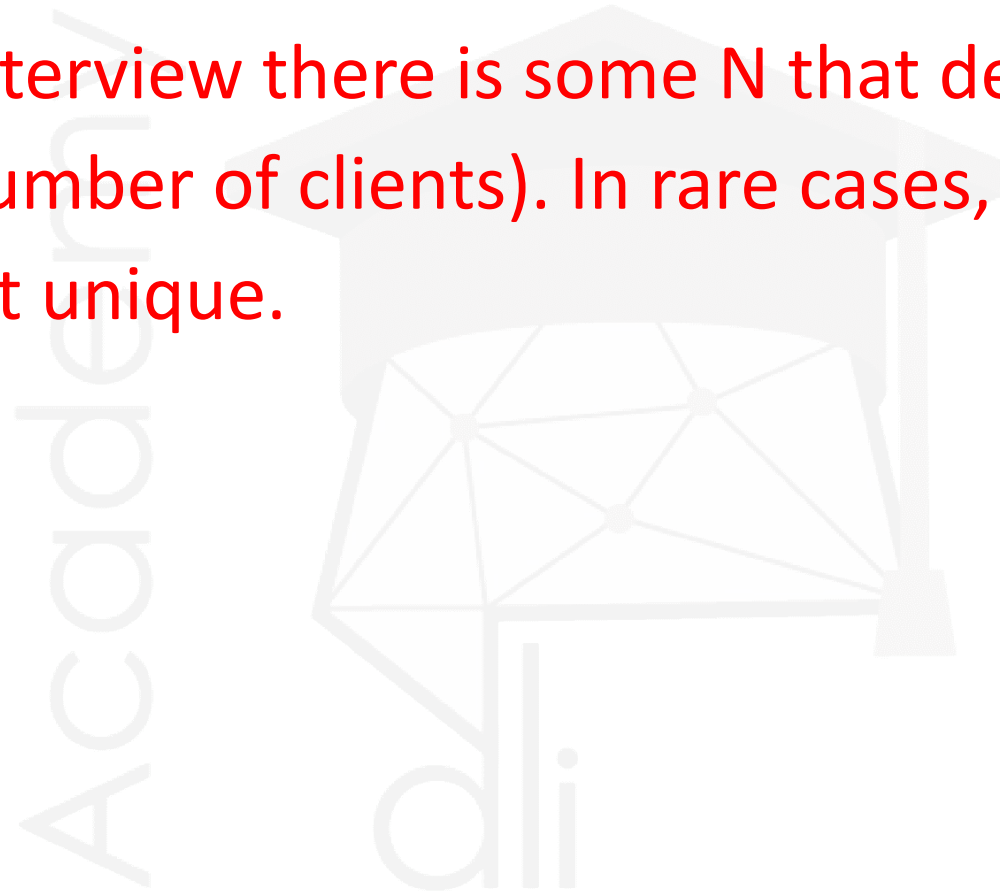
Given an algorithm, evaluate how much time and storage are required w.r.t. the input size.

In layman terms: suppose that executing the algorithm on a given input X^1 requires T^1 time and M^1 memory. What if we run the same algorithm on an input X^2 , which is $X^1 * 2$? Can we estimate T^2 and M^2 ?

Computational complexity is a formal/theoretical approach that allows to deal with the super-pragmatical topic of scalability.

Computational complexity: input

In any coding interview there is some N that defines the input size (e.g., the number of clients). In rare cases, the N might be not trivial or not unique.



Computational complexity: space and time

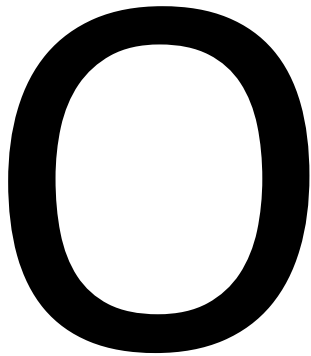
For any algorithm, one can analyse the computational complexity w.r.t. space and time resource allocation. In both cases, we don't care about exact numbers, but we look for a function.

Time complexity is not about exact computational time (in seconds), but is aimed at describing (and bounding) the number of elementary operations w.r.t. the input size N .

Similarly, space complexity does not focus on estimating the exact memory requirements (e.g. in kilobytes) of an algorithm, but tries to define a function that allows to estimate the requirement growth w.r.t. the input size.

In some cases, different algorithms might produce a space/time trade-off: highlighting and discussing this during the interview is surely worth bonus points!

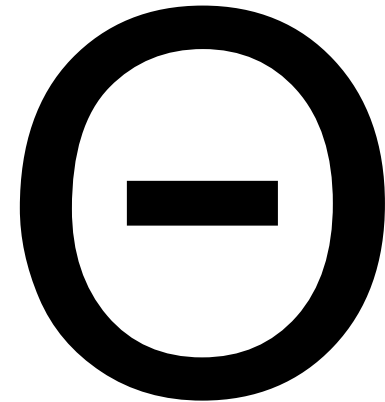
Computational complexity: notation



Big O
Worst case
Upper Bound

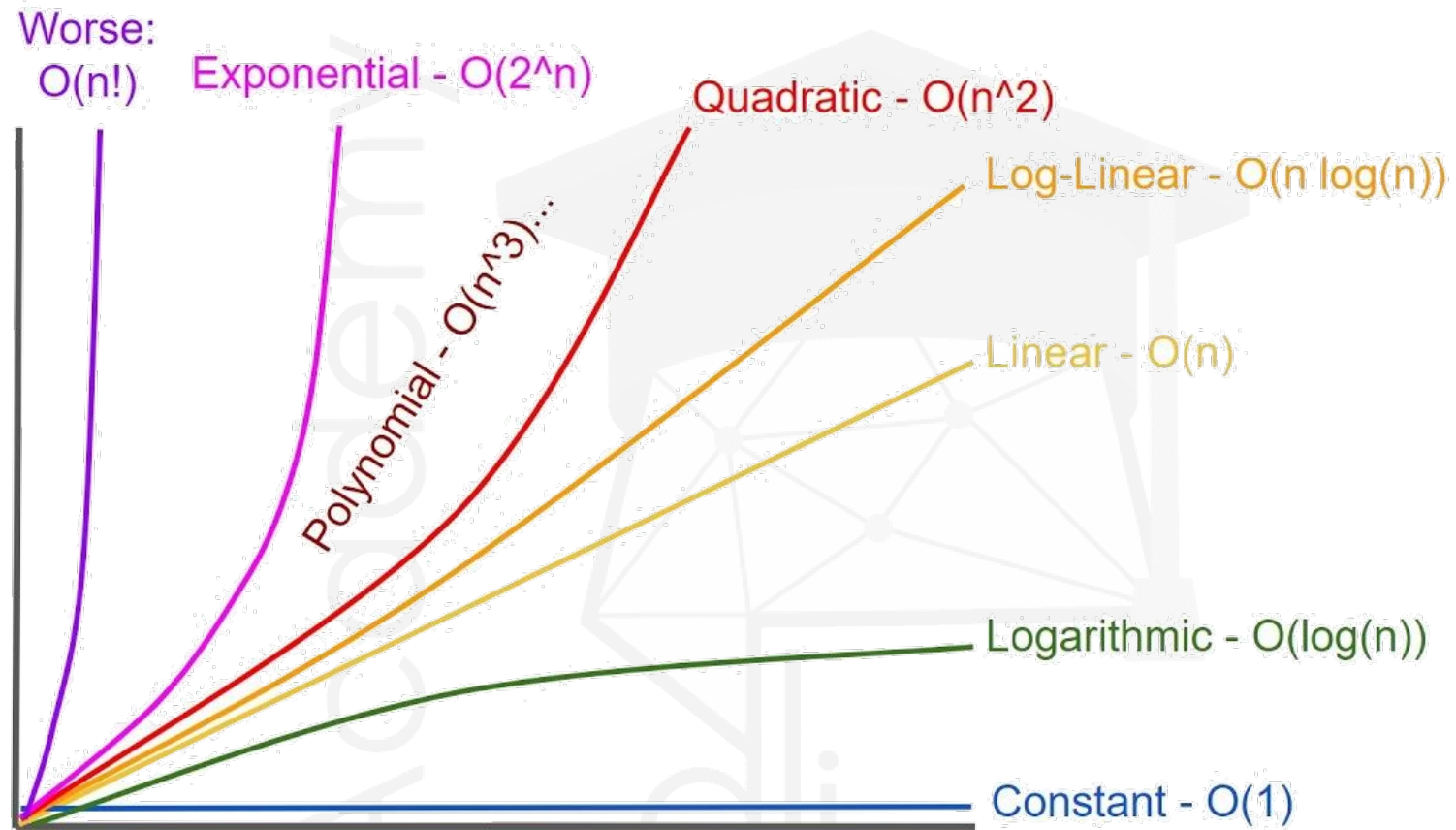


Omega
Best case
Lower Bound



Theta
Exact case
Tight Bound

Complexity classes



Let's ignore coefficients and all terms besides the worst one!
A function with complexity $6N^3 + 2000N \log N + 4$ is $O(N^3)$

Example

- **Given an array of clients, check whether two share the same age.**
- **N: number of clients**

```
def same_age_clients_1(clients):  
    n = len(clients)  
    for i in range(n):  
        for j in range(i+1,n):  
            if clients[i]==clients[j]:  
                return True  
    return False
```

Space: $O(N)$, $\Omega(N)$, $\Theta(N)$
Time: $O(N^2)$, $\Omega(1)$

```
def same_age_clients_2(clients):  
    n = len(clients)  
    clients = sorted(clients)  
    for index in range(1,n):  
        if clients[index]==clients[index-1]:  
            return True  
    return False
```

Space: $O(N)$, $\Omega(N)$, $\Theta(N)$
Time: $O(N\log N)$, $\Omega(N\log N)$, $\Theta(N\log N)$

Computational complexity: wrap-up

- Isolate the 'N' of the problem (is it unique?)
- Draft a solution
- Think about data structures (e.g. list vs array)
- Think about primitives you're using (e.g. sorting)
- Provide big-O for space and time
- Write the code!