# Data Structures and Algorithms

Lesson 3. Principles of Algorithm Design

Alexandra Ciobotaru

3 mar 2022

## Syllabus

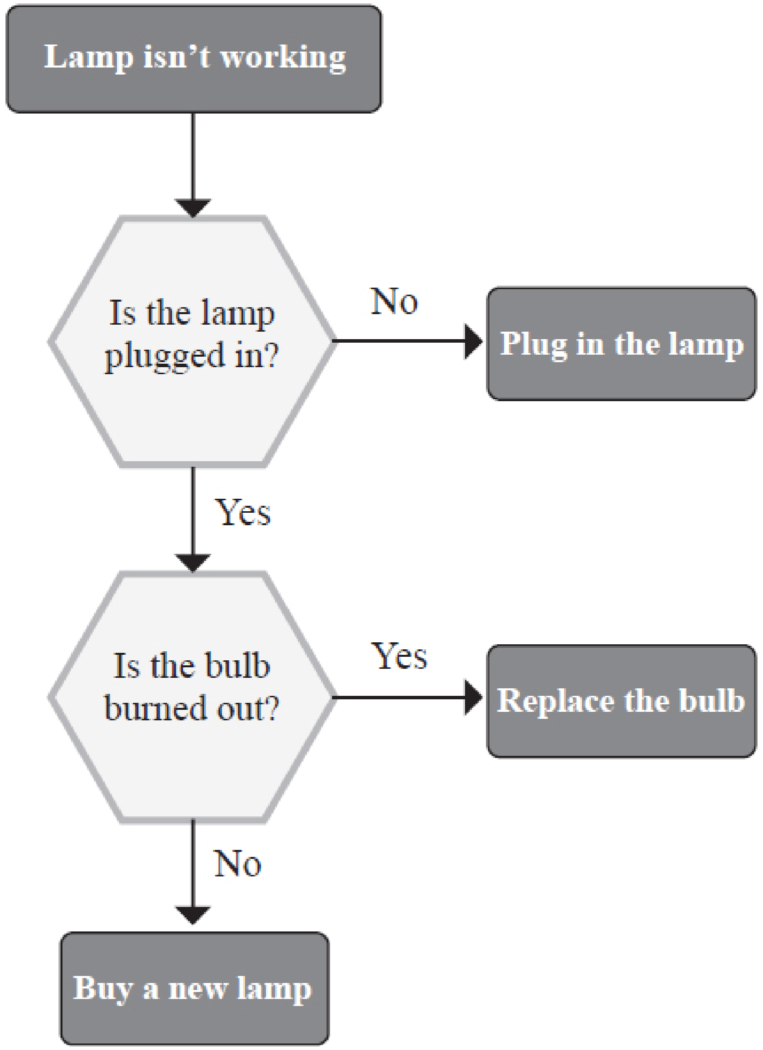
- **Lesson 1. Introduction: Algorithms, Data Structures and Cognitive Science.**
- **Lesson 2. Python Data Types and Structures.**
- **Lesson 3. Principles of Algorithm Design.**
- Lesson 4. Lists and Pointer Structures.
- Lesson 5. Stacks and Queues.
- Lesson 6. Trees.
- Lesson 7. Hashing and Symbol Tables.
- Lesson 8. Graphs and Other Algorithms.
- Lesson 9. Searching.
- Lesson 10. Sorting. Mid-term evaluation quiz (30%)
- Lesson 11. Selection Algorithms.
- Lesson 12. String Algorithms and Techniques.
- Lesson 13. Design Techniques and Strategies.
- Lesson 14. Implementations, Applications and Tools.
- Lab (20%) + Project (50%).

## What is an Algorithm, Exactly?

- Algorithms are not a special type of operation, necessarily. They are conceptual, a set of steps that you take in code to reach a specific goal – **to solve a specific problem**
  - They are essential for computer science and *intelligent* systems
  - They are important in many other domains (computational biology, economics,
  - ecology, communications, ecology, physics, and so on)
  - They play a role in technology innovation
  - They improve problem-solving and analytical thinking
- There are mainly two important aspects to solve a given problem:
  - 1. Firstly, we need an efficient mechanism to store, manage, and retrieve the data,
  - 2. secondly, we require an efficient algorithm which is a finite set of instructions to solve that problem.

# An efficient algorithm should have the following characteristics:

- It should be as specific as possible
- It should have each instruction properly defined
- There should not be any ambiguous instruction
- All the instructions of the algorithm should be executable in a finite amount of time and in a finite number of steps
- It should have clear input and output to solve the problem
- Each instruction of the algorithm should be important in solving the given problem



## Algorithm design paradigms

- In general, we can discern three broad approaches to algorithm design. They are:

1. **Divide and conquer** = breaking a problem into smaller simple sub-problems, solving these sub-problems, and then combining the results to obtain a global optimal solution. Examples:
    - Merge sort (https://www.geeksforgeeks.org/merge-sort/)
    - Binary search
    - Quick sort
    - Karatsuba – used for multiplication
    - Strassen's matrix multiplication
    - Closest pair of points

- 2. **Greedy algorithms** - the objective is to obtain the best optimum solution from many possible solutions in each step, and we try to get the local optimum solution which may eventually lead to the overall optimum solution; Greedy algorithms are mainly used for optimization. Examples:
    - Kruskal's minimum spanning tree
    - Dijkstra's shortest path
    - Knapsack problem
    - Prim's minimal spanning tree algorithm
    - Travelling salesman problem (greedy approach - choose the closest destination first)
- 3. **Dynamic programming** – uses recursion, but allows comparisons in different stages of the execution
    - Any recursive algorithm can be optimized through dynamic programming
    - The idea is to store the results of sub-problems, so that we do not have to re-compute them when needed later. This simple optimization reduces time complexities from exponential to polynomial.
    - For example, if we write simple recursive solution for Fibonacci Numbers, we get exponential time complexity and if we optimize it by storing solutions of subproblems, time complexity reduces to linear.

# Compute Fibonnaci recursively

Fibonacci Sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144…

```
def recur_fibo(n):
   if n <= 1:
       return n
   else:
       return(recur_fibo(n-1) + recur_fibo(n-2))
# take input from the user
nterms = int(input("How many terms? "))
# check if the number of terms is valid
if nterms <= 0:
   print("Plese enter a positive integer")
else:
   print("Fibonacci sequence:")
   for i in range(nterms):
       print(recur_fibo(i))
```

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n \geq 2 \end{cases}$$

`CONS`: for each number calculated, it needs to calculate all the previous numbers more than once – it takes a lot of time to compute

# Compute Fibonnaci with dynamic programming recurssion

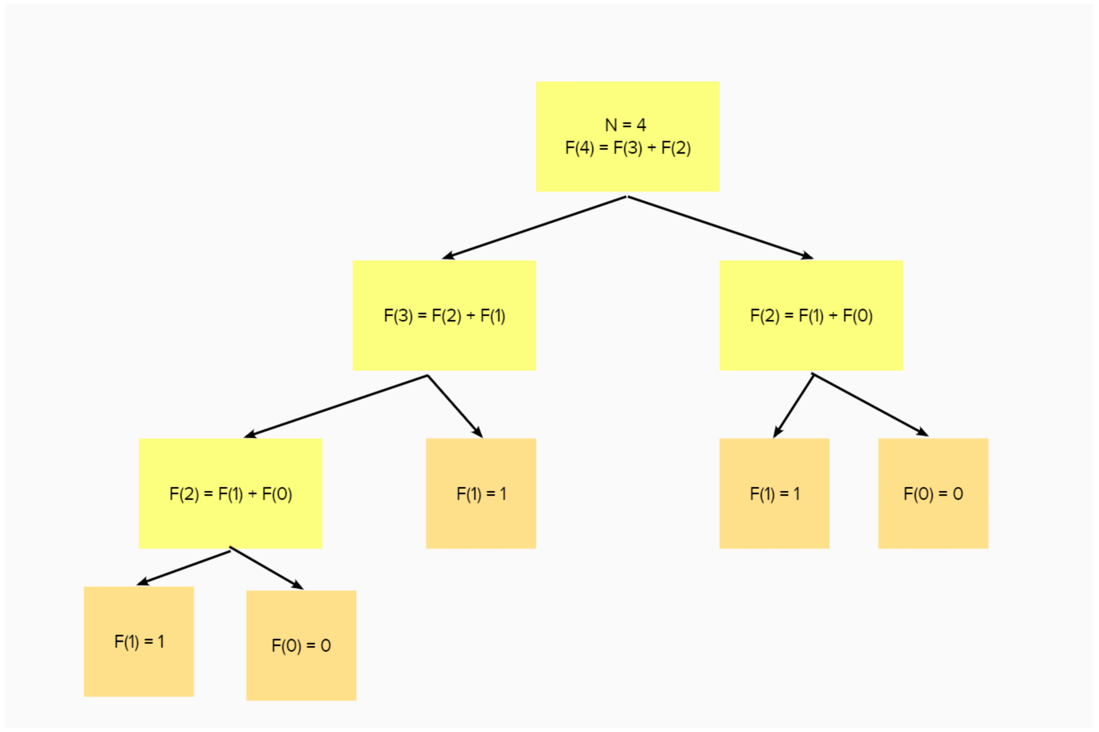
```
def fib(n):
    # base case
    if n == 0:
        return(0)
    if n == 1:
        return(1)
    # create empty dynamic programming array to store intermediate and temporary results
    dp = [0] * (n + 1)
    # find patterns
    dp[0] = 0
    dp[1] = 1
    print("Pre-generated DP Array = ", dp)
    # dp[2] = dp[1] + dp[0]
    # dp[i] = dp[i-1] + dp[i-2]
    for i in range(2,n+1):
        dp[i] = dp[i-1] + dp[i-2]
    print("Generated DP Array after filling = ", dp)
    return(dp[n])
print(fib(56))
```

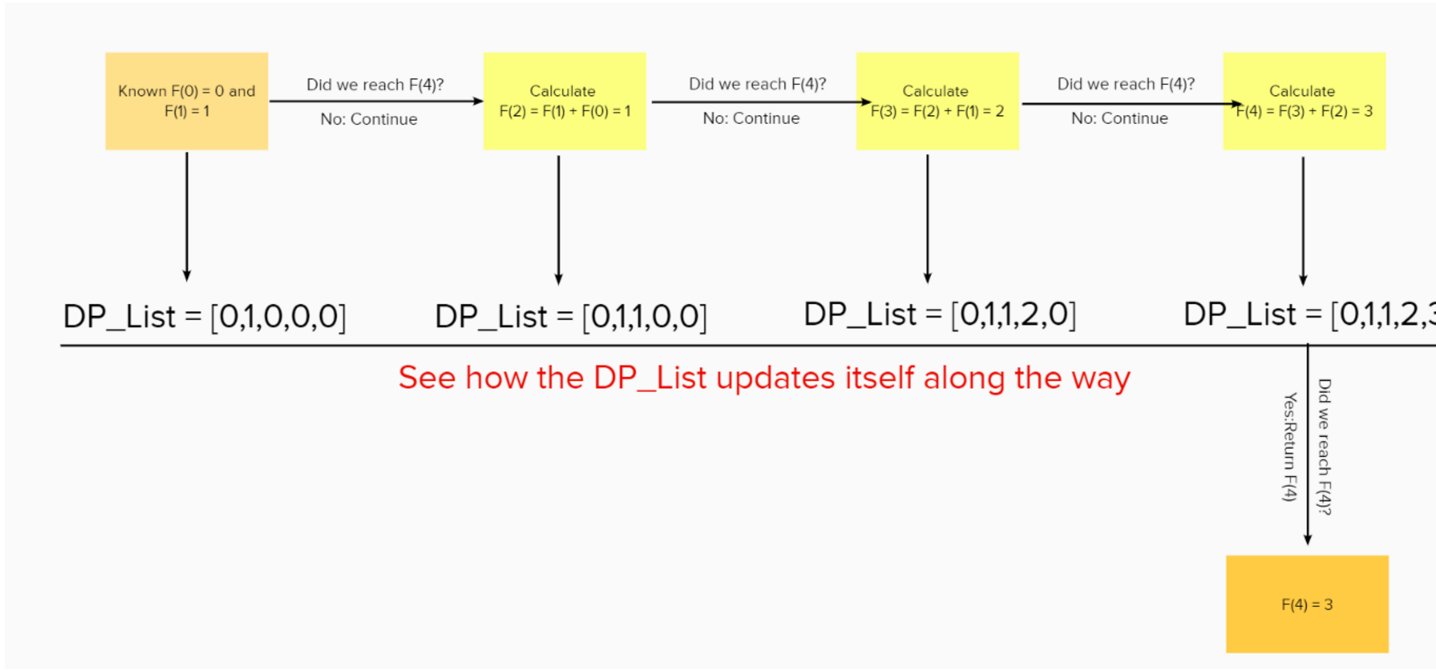
The base case tell the recursion when to terminate,
meaning the recursion is stopped once the base condition is met

And this is actually the major difference between dynamic programming with recursion and simple Recursion: in recursion, we do not store any intermediate results vs in dynamic programming, we do store all intermediate steps.

## Recursion

## Dynamic Programming





Source: https://towardsdatascience.com/dynamic-programming-i-python-8b20387870f5

## Compute Fibonnaci Iteratively

```
def fib_to(n):

    fibs = [0, 1]
    for i in range(2, n+1):
        fibs.append(fibs[-1] + fibs[-2])
    return fibs

print(fib_to(56))
```

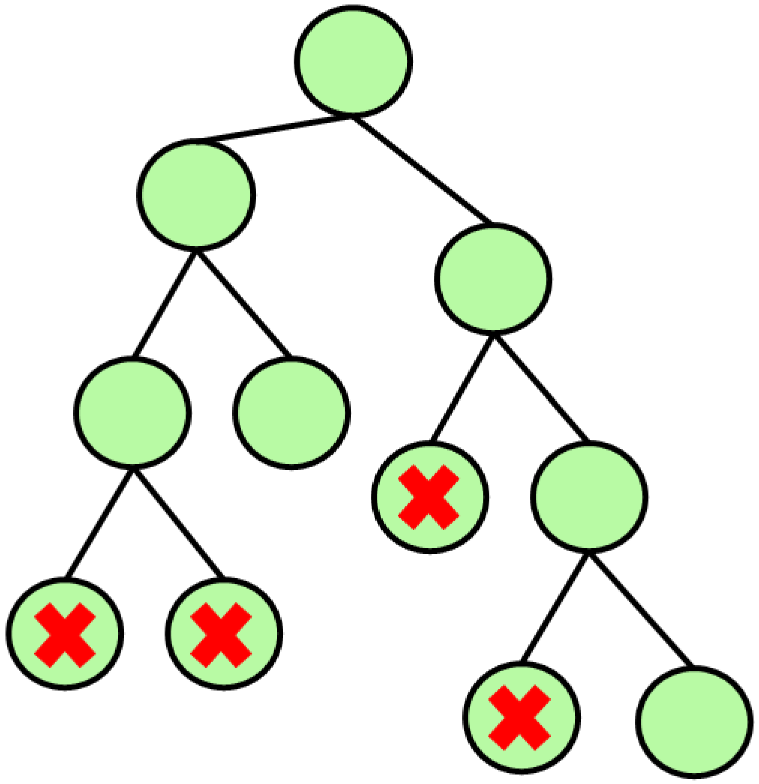
## Recursion vs. Iteration

| Recursion | Iteration |
|---|---|
| The function calls itself. | A set of instructions are executed repeatedly in the loop. |
| It stops when the termination condition is met. | It stops execution when the loop condition is met. |
| Infinite recursive calls may give an error related to stack overflow. | An infinite iteration will run indefinitely until the hardware is powered. |
| Each recursive call needs memory space. | Each iteration does not require memory storage. |
| Recursion is generally slower than iteration. | It is faster as it does not require a stack. |

## Backtracking

- Backtracking is a form of recursion that is particularly useful for types of problems such as traversing tree structures, where we are presented with a number of options for each node, from which we must chose one
- Depending on the series of choices made, we reach either a goal state, either a dead end
- If we reach a dead end, we must backtrack to a previous node and traverse a different branch
- **A backtracking algorithm is a problem-solving algorithm that uses a brute force approach for finding the desired output. The Brute force approach tries out all the possible solutions and chooses the desired/best solutions.**
- Backtracking is a *divide and conquer* method used for solving problems with multiple solutions(like searching)
- Also, it **prunes** branches that cannot give a result

# State-space tree



**State Space Tree** – A state-space tree is a tree representing all the possible states (solution or nonsolution) of the problem from the root as an initial state to the leaf as a terminal state.

Backtracking follows the following approach:

- It checks whether the given node is a valid solution or not.

- Discards several invalid solutions with one iteration.

- Enumerates all subtree of the node to find the valid solution.

Source: https://pythonwife.com/backtracking-in-python/

## Runtime analysis

- The performance of an algorithm is measured by the size of its input data and the time and memory space used by the algorithm
  - Time = key operations performed by the algorithm (it varies with the hardware used)
  - Space = storage needed to store the variables, constantsand instructions during execution
- An effective algorithm takes into consideration both space and time.
- Given two solutions, to determine which algorithm is better, a simple approach would be to run both programs, and the algorithm that takes the least time to execute for a given input is better than the other.

## Runtime analysis

- Worst-case complexity
  - The upper-bound complexity
  - The maximum running time required for an algorithm to execute
  - (the key operations are executed the maximum number of times)
- Best-case complexity
  - The lower-bound complexity
  - The minimum running time required for an algorithm to execute
  - (the key operations are be executed the minimum number of times)
- Average-case complexity
  - The average running time required for an algorithm to execute

## Asymptotic analysis

- Asymptotic analysis of an algorithm = computation of the running time of the algorithm in respect to the input
- It is possible that for a given input, one algorithm performs better than the other
- In asymptotic analysis we compare algorithms with respect to input size rather than the runtime, and we measure how the time of execution increase with the increase of the input size

# Asymptotic analysis

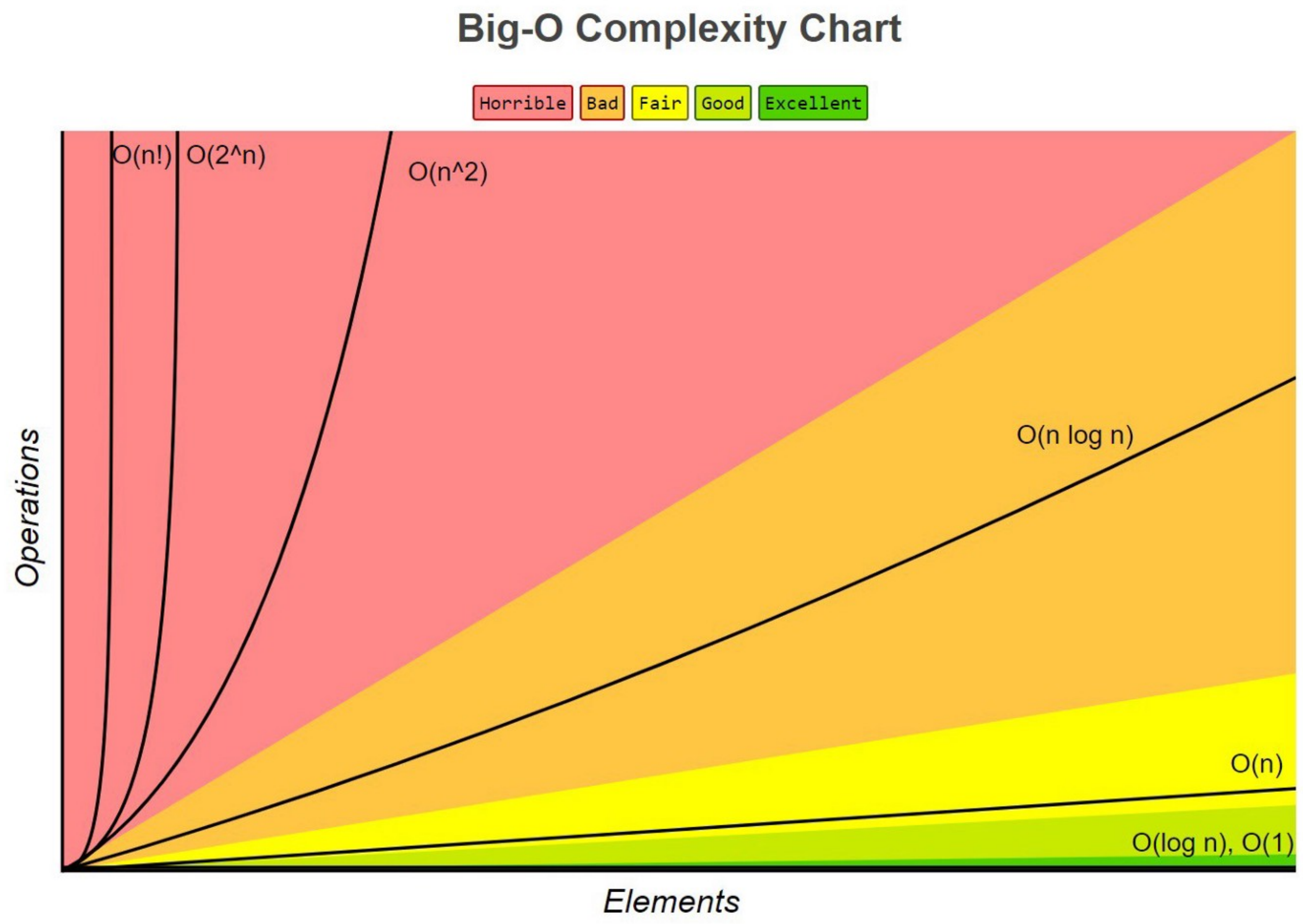
- Worst-case analysis
  - We consider the upper bound complexity in running time – the maximum time to be taken by the algorithm
  - Example: searching for an element in a list - the worst case happens when the element to be searched for is found in the last comparison, or not found at all -> in this case, the maximum number of comparisons will be the total numbers of elements in the list
- Average-case analysis
  - we consider all the possible cases where the element can be found in the list, and then we compute the average running time complexity
  - Example: searching for an element in a list:

$$average - case\ complexity = \frac{1+2+3+\cdots n}{n} = \frac{n(n+1)}{2}$$

- Best-case analysis
  - The lower bound complexity in running time - the minimum time needed for an algorithm to run
  - Example: searching for an element in a list – he best case is when the searched for element is found in the first comparison

## Big O notation

- Big O is simplified analysis of an algorithm's efficiency
- The letter in Big O stands for order – the order of the rate of growth
- It measures the worst-case running time complexity (the maximum time to be taken by the algorithm)
- We say that function T(n) is a big O of another function F(n), and we define it as:
- T(n) = O(F(n))



Source: https://www.bigocheatsheet.com/

## Constant time
## O(1)

- Independent of input size
- X = 5 + (15 * 20)

- When we have a sequence of equations with constant time:
- X = 5 + (15 * 20)
- Y = 15 – 2
- Print(X+Y)
- How do we compute Big O for this block of code?
- Total_time = O(1) + O(1) + O(1) = 3 * O(1) = O(1)
- (Big O computation doesn't care about constants)

## Linear time
## N * O(1) = O(N)

- For x in range(0, n):
  - Print(x) $\longrightarrow$ O(1)

Y = 5 + (15 * 20) $\longrightarrow$ O(1)

$$\left.\begin{array}{l}\text{For x in range(0, n):}\\ \qquad\qquad\text{print(x)}\end{array}\right\} \longrightarrow \text{O(N)}$$

Total_time = O(1) + O(N) = O(N)

(Big O computation doesn’t care about low order terms)

## Quadratic time

## $O(n^2)$

```
• For x in range(0, n):
•     for y in range(0,n):
•         print(x*y)
```

$$\begin{tikzcd} \text{print(x*y)} \arrow[r] & O(1) \arrow[r] & O(N) \arrow[r] & N * O(N) \end{tikzcd}$$

# O(?)

- x = 5 + (15 * 20) $\longrightarrow$ O(1)
- for x in range(0,n):
-     print(x)

$$\left.\begin{array}{l}\texttt{for x in range(0,n):}\\ \quad\texttt{print(x)}\end{array}\right\}\longrightarrow O(N)$$

- for x in range(0,n):
-     for y in range(0,n):
-         print(x*y)

$$\left.\begin{array}{l}\texttt{for x in range(0,n):}\\ \quad\texttt{for y in range(0,n):}\\ \quad\quad\texttt{print(x*y)}\end{array}\right\}\longrightarrow O(N^2)$$

- What is the total runtime?
- The max of the three: O($N^2$)

## O(?)

- if x>0:
-     print("x bigger") $\longrightarrow$ O(1)
- elif x<0:
-     print("x smaller") $\longrightarrow$ O(logN)
- else:
-     print("x equals 0") $\longrightarrow$ $O(N^2)$

| Complexity Class | Name | Example operations |
|---|---|---|
| O(1) | Constant | Append, get item, set item |
| O(logn) | Logarithmic | Find an element in a sorted list |
| O(n) | Linear | Copy, insert, delete |
| O(nLogn) | Linear-logarithmic | Sort a list |
| O($n^2$) | Quadratic | Find the shortest path between two nodes in a graph |
| O($n^3$) | Cubic | Matrix multiplication |
| O($2^n$) | Exponential | backtracking |

## Thank you!