<https://www.tellabs.com/>

<https://www.nixu.com/>

<https://github.com/TT00FE39-3001/lecture2>

55d41251

<https://cpp.sh/>

<https://www.youtube.com/watch?v=aGjL7YXI31Q>

# Algorithms Lecture 1 -- Introduction to asymptotic notations

<https://www.youtube.com/watch?v=zWg7U0OEAoE>

# Lecture - 1 Introduction to Data Structures and Algorithms

# Topics

* Review
* Algorithm techniques
  + Brute Force
  + Decrease-and-Conquer
  + Divide-and-Conquer
* Analyses tool(s): Big �

## Part 0: Review

* Course mechanics
  + Activities: Group + Individual (at home),
  + Due before next lecture
  + Approach: We try to present the material in an object-based approach (not OOP), therefore we will not use inheritance etc.s
* Previous lecture:
  + [Abstract Data Types(ADT)](https://en.wikipedia.org/wiki/Abstract_data_type)
  + [Data Structures](https://en.wikipedia.org/wiki/Data_structure)
  + **Array-based** [Stacks](https://www.softwaretestinghelp.com/stack-in-cpp/)
  + **Array-based** [Queues](https://www.softwaretestinghelp.com/queue-in-cpp/)

## Part 1: Brute Force, Decrease-and-Conquer

* Algorithm techniques:
  + [Brute Force](https://www.geeksforgeeks.org/brute-force-approach-and-its-pros-and-cons/): [Linear Search](https://www.softwaretestinghelp.com/searching-algorithms-in-cpp/) of lists
  + [Decrease-and-Conquer](https://www.geeksforgeeks.org/decrease-and-conquer/):
    - [Binary Search](https://www.softwaretestinghelp.com/searching-algorithms-in-cpp/) of sorted lists
    - [Running time of binary search](https://www.khanacademy.org/computing/computer-science/algorithms/binary-search/a/running-time-of-binary-search)
* [Recursion vs Iteration](https://www.geeksforgeeks.org/introduction-to-recursion-data-structure-and-algorithm-tutorials/)
* [Activity 1](https://github.com/TT00FE39-3001/lecture2/blob/main/activity1/README.md)

## Part 2: Analyses tools

* Math review: [Logarithms](https://www.mathsisfun.com/algebra/logarithms.html)
* [Big $O$](https://justin.abrah.ms/computer-science/big-o-notation-explained.html)
* Tools: [Graphing online](https://www.mathway.com/graph)
* [Activity 2](https://github.com/TT00FE39-3001/lecture2/blob/main/activity2/README.md)

## Part 3: Algorithm techniques

* [Divide-and-Conquer](https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm)
* [Activity 3](https://github.com/TT00FE39-3001/lecture2/blob/main/activity3/README.md)

## Misc

* [GitHub workflow](https://github.com/TT00FE39-3001/lecture2/blob/main/github.md)
* [Links](https://github.com/TT00FE39-3001/lecture2/blob/main/links.md)

ACTIVITY 2

**# Activities**

**## Task 1**

Refer to the first [link](#links).

- Why is Algorithm Analysis Important?

- Explain the Big $O$ notation

- Compare `Linear`, `Logarithmic`, `Quadratic` and `Constant` complexities.

**## Task2**

Refer to the first [link](#links).

- Write a simple algorithm in C++ that finds the square of the first item in a list and then prints it on the screen.

- What is the complexity of the algorithm?

**## Task 3**

Refer to the first [link](#links).

- Write a simple program that displays all items in a list to the console.

- What is the complexity of the algorithm?

**## Task 4: : Individual, at home**

Refer to this [pdf](./big\_o.pdf):

- What is the difference between complexity and performance:

- Does complexity affects performance bor is it the other way around?

- Restate the formal definition of big $O$ in plain English

**## Links**

- [Big O Notation and Algorithm Analysis ](https://stackabuse.com/big-o-notation-and-algorithm-analysis-with-python-examples/)

- [Visualization](https://www.cs.usfca.edu/~galles/visualization/Search.html)

- [Big-O notation explained by a self-taught programmer](https://justin.abrah.ms/computer-science/big-o-notation-explained.html)

- [Big-O is easy to calculate, if you know how](https://justin.abrah.ms/computer-science/how-to-calculate-big-o.html)

- https://cpp.sh/

ANSWERS

**## Task 1**

Refer to the first [link](#links)

.

- Why is Algorithm Analysis Important?

- Explain the Big $O$ notation

- Compare `Linear`, `Logarithmic`, `Quadratic` and `Constant` complexities.

- Why is Algorithm Analysis Important?

### Introduction

There are usually multiple ways to solve the problem using a computer program. For instance, there are several ways to sort items in an array - you can use [merge sort](https://stackabuse.com/merge-sort-in-python/), [bubble sort](https://stackabuse.com/bubble-sort-in-python/), [insertion sort](https://stackabuse.com/insertion-sort-in-python/), and so on. All of these algorithms have their own pros and cons and the developer's job is to weigh them to be able to choose the best algorithm to use in any use case. In other words, the main question is which algorithm to use to solve a specific problem when there exist multiple solutions to the problem.

Algorithm analysis refers to the analysis of the complexity of different algorithms and finding the most efficient algorithm to solve the problem at hand. Big-O notation is a statistical measure used to describe the complexity of the algorithm.

*In this guide, we'll first take a brief review of algorithm analysis and then take a deeper look at the Big-O notation. We will see how Big-O notation can be used to find algorithm complexity with the help of different Python functions.*

**Note:** Big-O notation is one of the measures used for algorithmic complexity. Some others include Big-Theta and Big-Omega. Big-Omega, Big-Theta and Big-O are intuitively equal to the **best**, **average** and **worst** time complexity an algorithm can achieve. We typically use Big-O as a measure, instead of the other two, because it we can guarantee that an algorithm runs in an acceptable complexity in its worst case, it'll work in the average and best case as well, but not vice versa.

### Why is Algorithm Analysis Important?

To understand why algorithm analysis is important, we will take the help of a simple example. Suppose a manager gives a task to two of his employees to design an algorithm in Python that calculates the factorial of a number entered by the user. The algorithm developed by the first employee looks like this:

def fact(n):

product = 1

for i in range(n):

product = product \* (i+1)

return product

print(fact(5))

Notice that the algorithm simply takes an integer as an argument. Inside the fact() function a variable named product is initialized to 1. A loop executes from 1 to n and during each iteration, the value in the product is multiplied by the number being iterated by the loop and the result is stored in the product variable again. After the loop executes, the product variable will contain the factorial.

Similarly, the second employee also developed an algorithm that calculates the factorial of a number. The second employee used a recursive function to calculate the factorial of the number n:

def fact2(n):

if n == 0:

return 1

else:

return n \* fact2(n-1)

print(fact2(5))

The manager has to decide which algorithm to use. To do so, they've decided to choose which algorithm runs faster. One way to do so is by finding the time required to execute the code on the same input.

In the Jupyter notebook, you can use the %timeit literal followed by the function call to find the time taken by the function to execute:

%timeit fact(50)

This will give us:

9 µs ± 405 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

The output says that the algorithm takes 9 microseconds (plus/minus 45 nanoseconds) per loop.

Similarly, we can calculate how much time the second approach takes to execute:

%timeit fact2(50)

This will result in:

15.7 µs ± 427 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)

The second algorithm involving recursion takes 15 microseconds (plus/minus 427 nanoseconds).

The execution time shows that the first algorithm is faster compared to the second algorithm involving recursion. When dealing with large inputs, the performance difference can become more significant.

However, execution time is not a good metric to measure the complexity of an algorithm since it depends upon the hardware. A more objective complexity analysis metric for an algorithm is needed. This is where the Big O notation comes to play.

**## Task 3**

Refer to the first [link](#links).

- Write a simple program that displays all items in a list to the console.

- What is the complexity of the algorithm?

#include <algorithm>

#include <iostream>

#include <list>

int main() {

std::list<int> my\_list = { 12, 5, 10, 9 };

for (int x : my\_list) {

std::cout << x << '\n';

}

}

complexity is N