Algorithmic Workshop

The Algorithm

An **Algorithm** is a sequence of steps to solve a problem.

An algorithm is a set of instructions designed to perform a specific task.

This can be a simple task (multiplying two numbers) or a complex task (video compression)

Each search engine (such as Google) has its own "*algorithm*" that ranks websites for each keyword or combination of keywords (in milliseconds , seconds or in minutes)

The word algorithm is derived from the last name of Muhamed ibn Musa Al-Khwarizmi

Learning Outcomes from the course

Skills	Learning Outcomes
To Design an algorithm	Design an algorithm in a language-independent way
	Implement efficiently an algorithm in Python
To Understand complexity	Evaluate the time needed for a given algorithm to complete
	Evaluate the memory needed for a given algorithm to complete
	Compute or Find the complexity (space and time) of an algorithm
To Choose the best algorithm & Demonstrate a familiarity with major algorithms	Decide whether it is worth coding this algorithm or find another, more performant
	Comparing different algorithms to solve a problem
Apply important algorithmic design paradigms	Describe the Divide-And-Conquer paradigm and explain when an algorithmic design situation calls for it.
	Describe the Dynamic-Programming paradigm and explain when an algorithmic design situation calls for it.
	Describe the Greedy paradigm and explain when an algorithmic design situation calls for it.
	Explain what an approximation algorithm is, and the benefit of using approximation algorithms. Be familiar with some approximation algorithms
	Explain the different ways to design a randomized algorithm
Graph algorithms	Explain the major graph algorithms and their analyses.
	Design specific algorithms on sparse very large graphs.

Introduction

- **Factors that affects program performance:**
  - Hardware
  - Compilers
  - Programming Language
  - Operating Systems
  - Data structures
  - **Algorithms**

Algorithm design and algorithm analysis

- **Algorithm Design is** a specific instructions for completing a task.

“algorithms design are patterns for completing a task in an efficient way.”

- **Algorithms Analysis is** the determination of the amount of resources (such as time and storage) necessary to execute the algorithm.

“usually described as (**time complexity**) and (**space complexity**) of an algorithm”

(time complexity) and (space complexity)

- *Time complexity* is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm

- *Space complexity* is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm

- **Big O notation** is the most common metric for calculating time complexity.

Time Complexity (Big O notation)

- **Big O notation** describes the execution time of the algorithm in relation to the number of steps required to complete it.

- Usually is the **worst case running time of an algorithm**

- Used in computer science in the analysis of **algorithms complexity**

How To Evaluate Efficiency of Programs

GOAL: to evaluate different algorithms

- Measure with a timer
- Count the operations

Measure with a timer(1)

```
import time
t0 = time.clock()
sum=0
for i in range(10000000):
    sum+=i
t1 = time.clock() -t0
print("Time of Running : ",t1)
```

import time

- use time module
- importing means to bring in that class into your own file

t0 = time.clock()

- Startclock

for i in range(10000000):

- Loop

- Running time varies implementation (Programming Languages)
- Running time varies between computers

Counting Operations(2)

- Assume these steps take constant time:
  - mathematical operations
  - comparisons
  - assignments
  - accessing objects in memory

Counting Operations(2)

```
def mysum(n):
	total = 0
	for i in range(n+1):
		total += i
	return total
```

1 Operation

1 Operation(n times)

2 Operation(n times)

Total operations (1+3n) operation

What is the number of steps in the worst case?

```
def program1(n):
    total = 0
    for i in range(1000):
        total += i

    while n > 0:
        n -= 1
        total += n

    return total
```

1

1000

2000

1n+1

2n

2n

1

5n+3003

In the worst case scenario, n is a large positive number.

We first execute the assignment total = 0

1. for i in range(1000) loop. This loop is executed 1000 times and has three steps
   a) one for the assignment of i each time through the loop
   b) two for the += operation on each iteration.
2. the second loop (while n > 0) n times. This loop has five step
   a) conditional check, n > 0
   b) and two each for the -= and += When we finally get to the point where n = 0.
3. We next check if n > 0 - it is not so we do not enter the loop.
4. Adding one more step for the return statement.

in the worst case = 5*n + 3003 steps.

Complexity=O(n)

Big O notation

1) 600
2) $\log(n) + 4$
3) $\log(n) + n + 4$
4) $0.0001 * n * \log(n) + 300n$
5) $n^2 + 2n + 2$
6) $n^2 + 100000n + 31000$
7) $2n^{30} + 3^n$

1) $O(1)$ constant
2) $O(\log n)$ logarithmic
3) $O(n)$ linear
4) $O(n\log n)$ log-linear
5) $O(n^2)$ polynomial
6) $O(n^2)$ polynomial
7) $O(3^n)$ exponential

Big O notation

```
for i in range(n):
    print('a')
for j in range(n*n):
    print('b')
```

Complexity: O(n^2)

```
for i inrange(n):
    for j inrange(n):
        print('a')
```

Complexity: O(n^2)

```
for i inrange(100):
    print('a')
```

Complexity: O(1)

Linear Search

3	6	7	10	4	12	9	5	8

1.) let's find 5 with linear search algorithm

https://ramnauth.github.io/cs%20101/2018/10/17/sorting-searching/

Motivation (Binary Search)

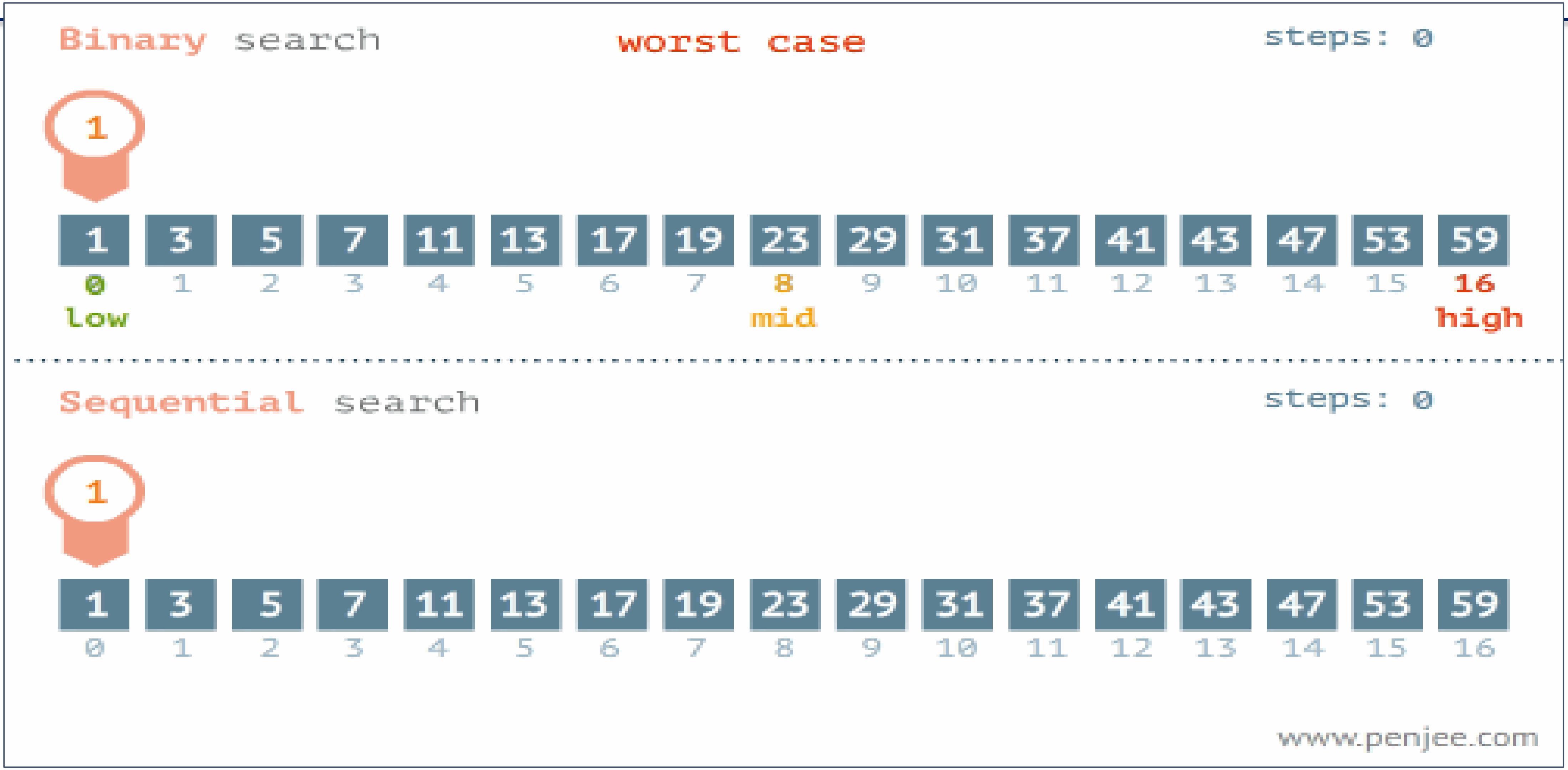
Binary Search

3	4	5	7	8	9	10	12	15	19	20	21	22	24	25

2.) let's find 22 with binary search algorithm

- https://ramnauth.github.io/cs%20101/2018/10/17/sorting-searching/

Linear vs Binary Search (worst case)



- https://blog.penjee.com/binary-vs-linear-search-animated-gifs/

Linear vs Binary Search (best case)

- https://blog.penjee.com/binary-vs-linear-search-animated-gifs/

Linear vs Binary Search

- Choose One:
  - Linear Search
  - Binary Search(Sort the array)

  - Linear Search Big O notation is O(n)
  - Binary Search Big O notation is **O**(log N)

Comparison of *N*, *logN* and N^2

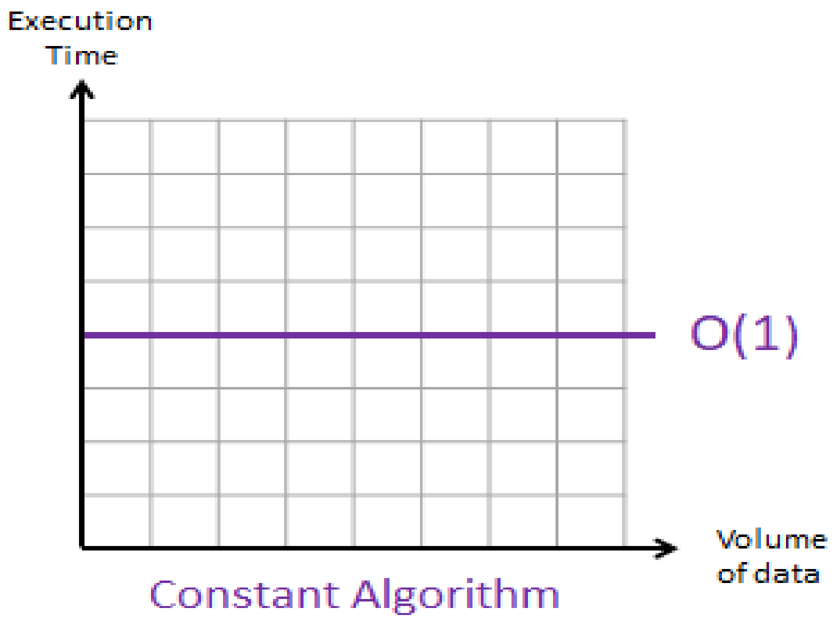
N	O(LogN)	O(N^2)
16	4	256
64	6	4K
256	8	64K
1,024	10	1M
16,384	14	256M
131,072	17	16G
262,144	18	6.87E+10
524,288	19	2.74E+11
1,048,576	20	1.09E+12
1,073,741,824	30	1.15E+18

Example on Constant Notation: O(1)

- The constant notation describes an algorithm that will always execute in the same execution time regardless of the size of the data set.

- For instance, an algorithm to retrieve the first value of a data set, will always be completed in one step, regardless of the number of values in the data set.

```
FUNCTION getFirstElemnt(list)
    RETURN list[0]
END FUNCTION
```

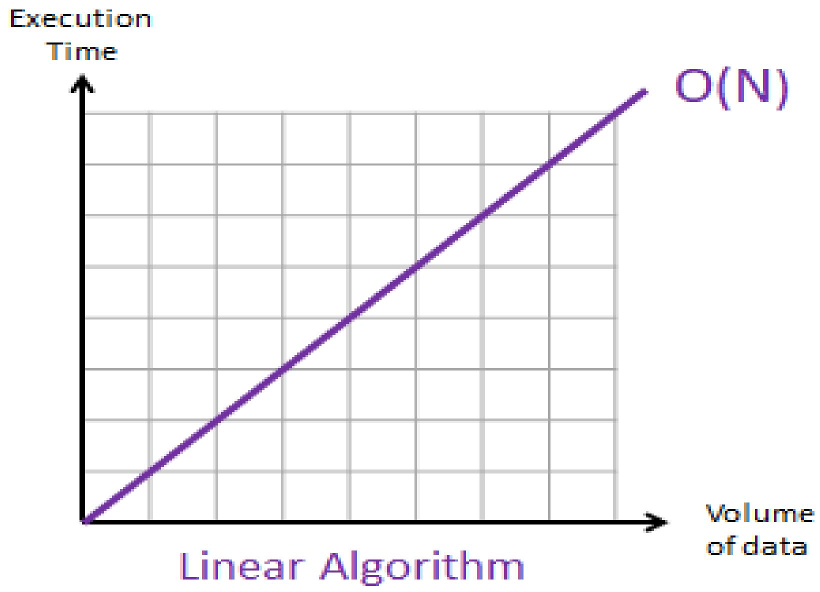


Example on Linear Notation: O(N)

- A linear algorithm is used when the execution time of an algorithm grows **in direct proportion** to the size of the data set it is processing.

- Algorithms, such as the linear search.

```
FUNCTION linearSearch(list, value)
    FOR EACH element IN list
        { IF (element == value)
            RETURN true
        }
RETURN false
END FUNCTION
```

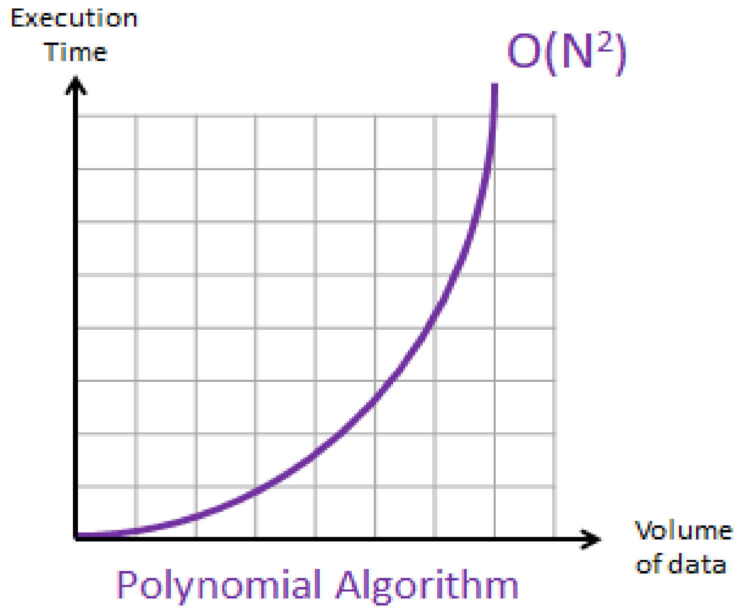


Example on Polynomial Notation: O(N²), O(N³), etc.

- $O(N^2)$ when the complexity of the algorithm proportional to the square of the size of the data set.
- $O(N^3)$ when the complexity of the algorithm proportional to the cube of the size of the data set.

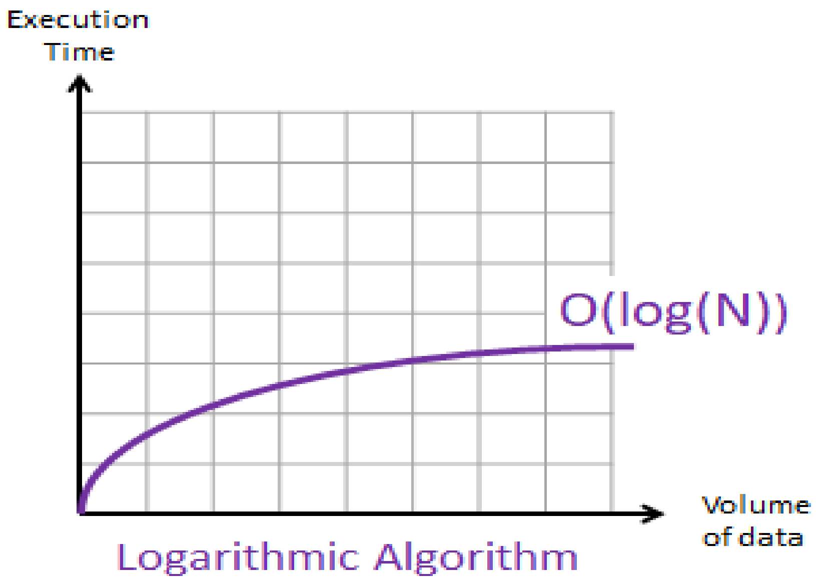
- Algorithms which are based on **nested loops**

```
PROCEDURE displayTimesTable()
    FOR i FROM 1 TO 10
        FOR j FROM 1 TO 10
            product = i*j
            OUTPUT i + " times " + j + " equals " + product
        NEXT j
    NEXT i
END PROCEDURE
```



Example on Logarithmic Notation: O(log(N))

- A **binary search** is a typical example of logarithmic algorithm.
- In a binary search, half of the data set is discarded after each iteration. Which means that an algorithm which searches through 2,000,000 values will just need one more iteration to search in 1,000,000 values only then in 500,000 then in 250,000 then in 125,000 (in 4 iterations only)and so on.



Example on Exponential Notation: $O(2^n)$

The exponential notation $O(2^N)$ describes an algorithm whose growth doubles with each addition to the data set.

Example of exponential algorithm: An algorithm to list all the possible binary permutations depending on the number of bits.

2 bits => 4 permutations (2^2)

[0, 0]

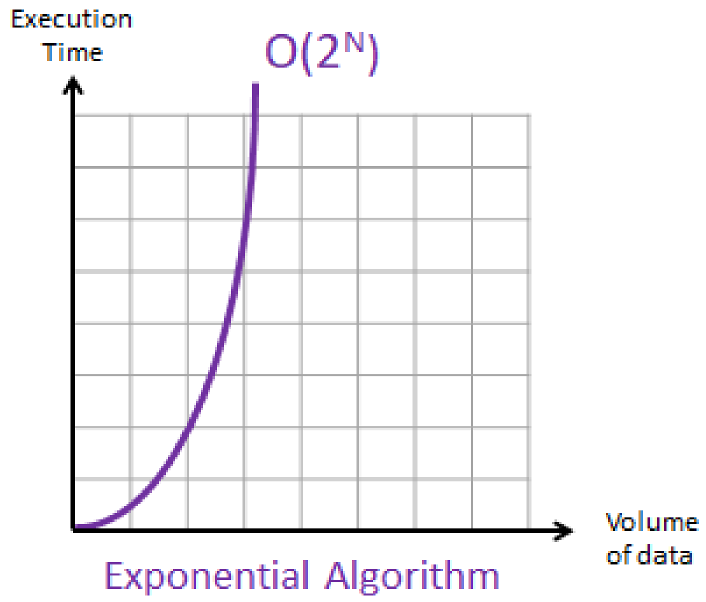
[1, 0]

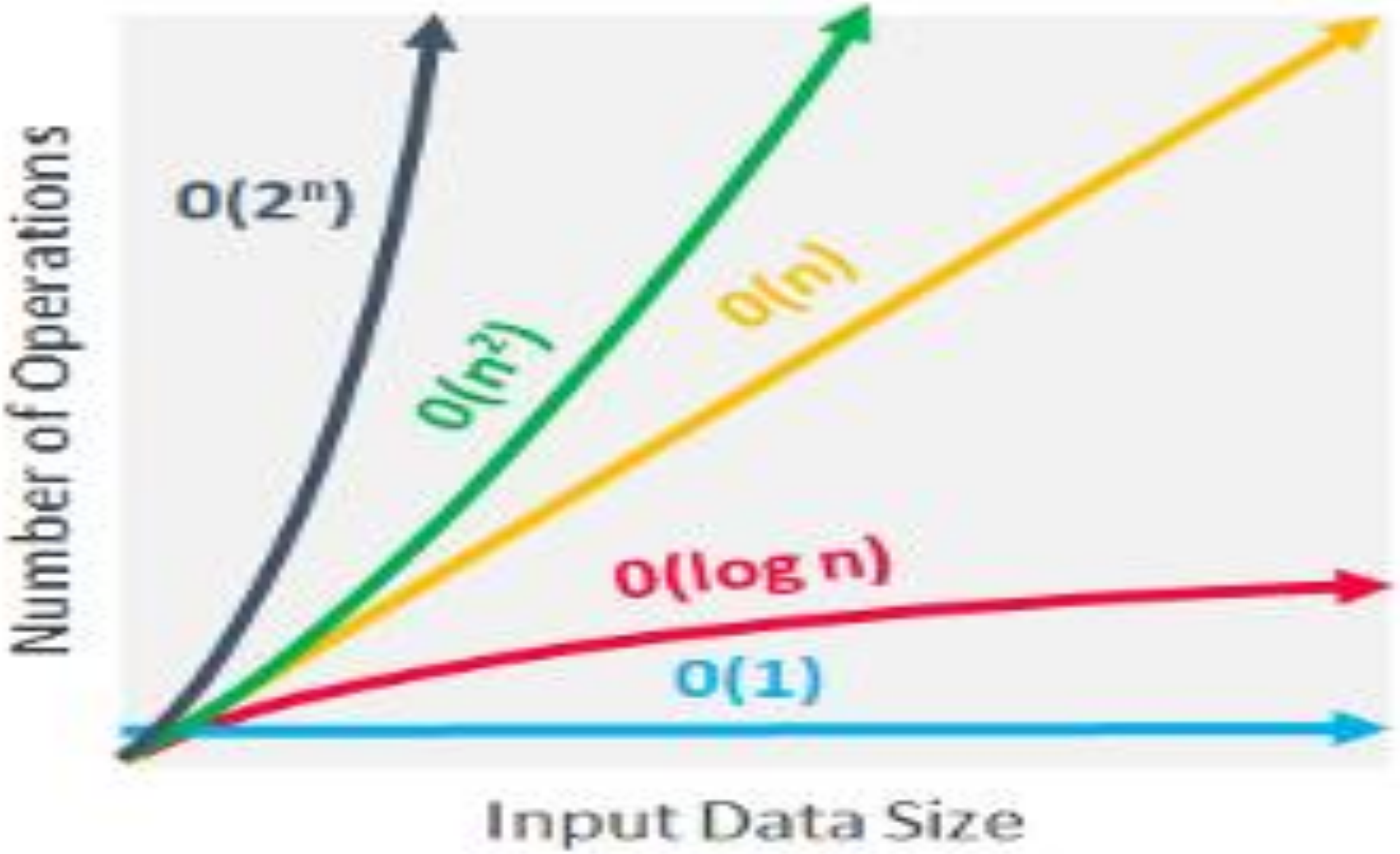
…….

4 bits => 16 permutations (2^4)

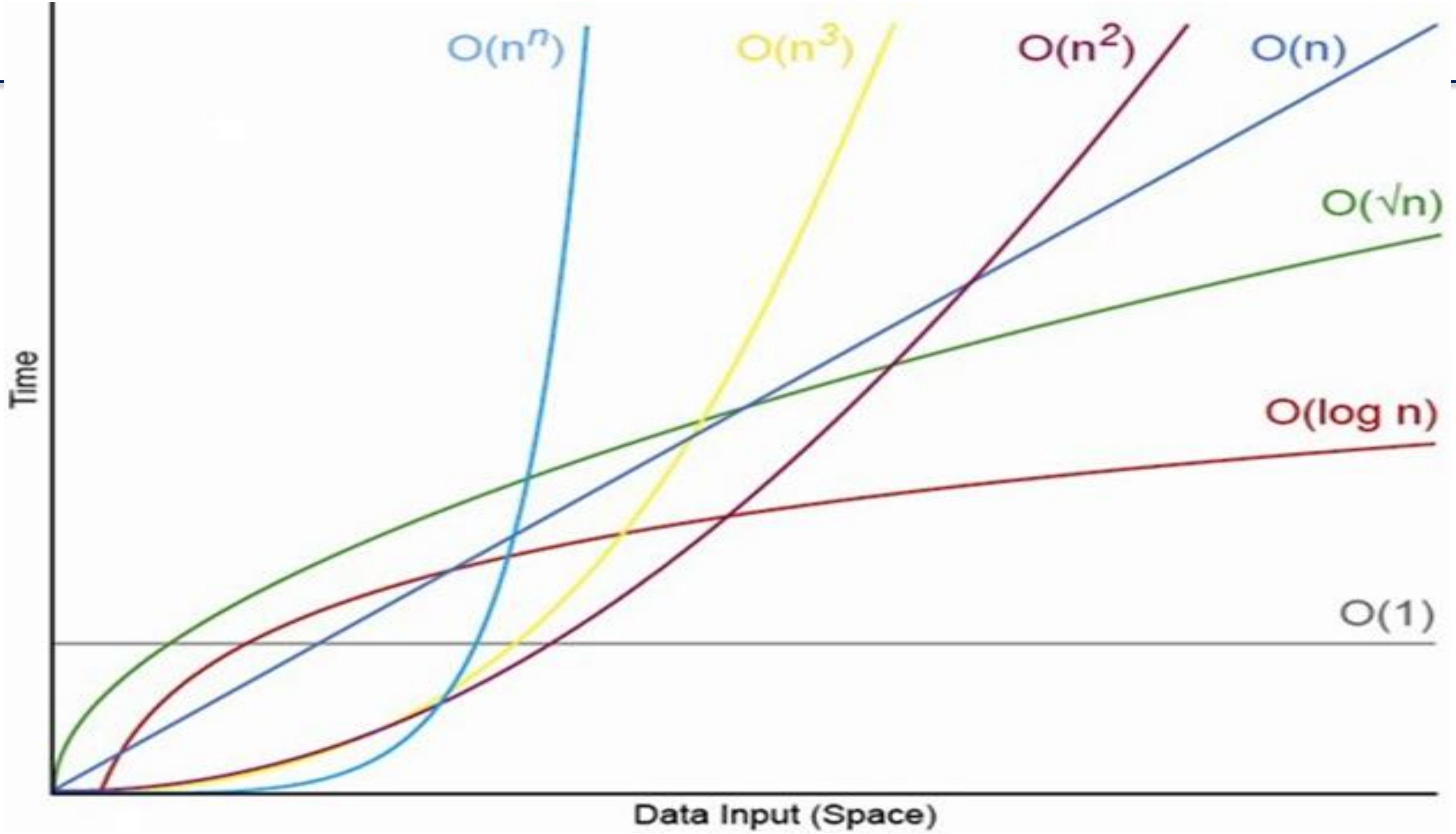
[0, 0, 0, 0]

[1, 0, 0, 0]

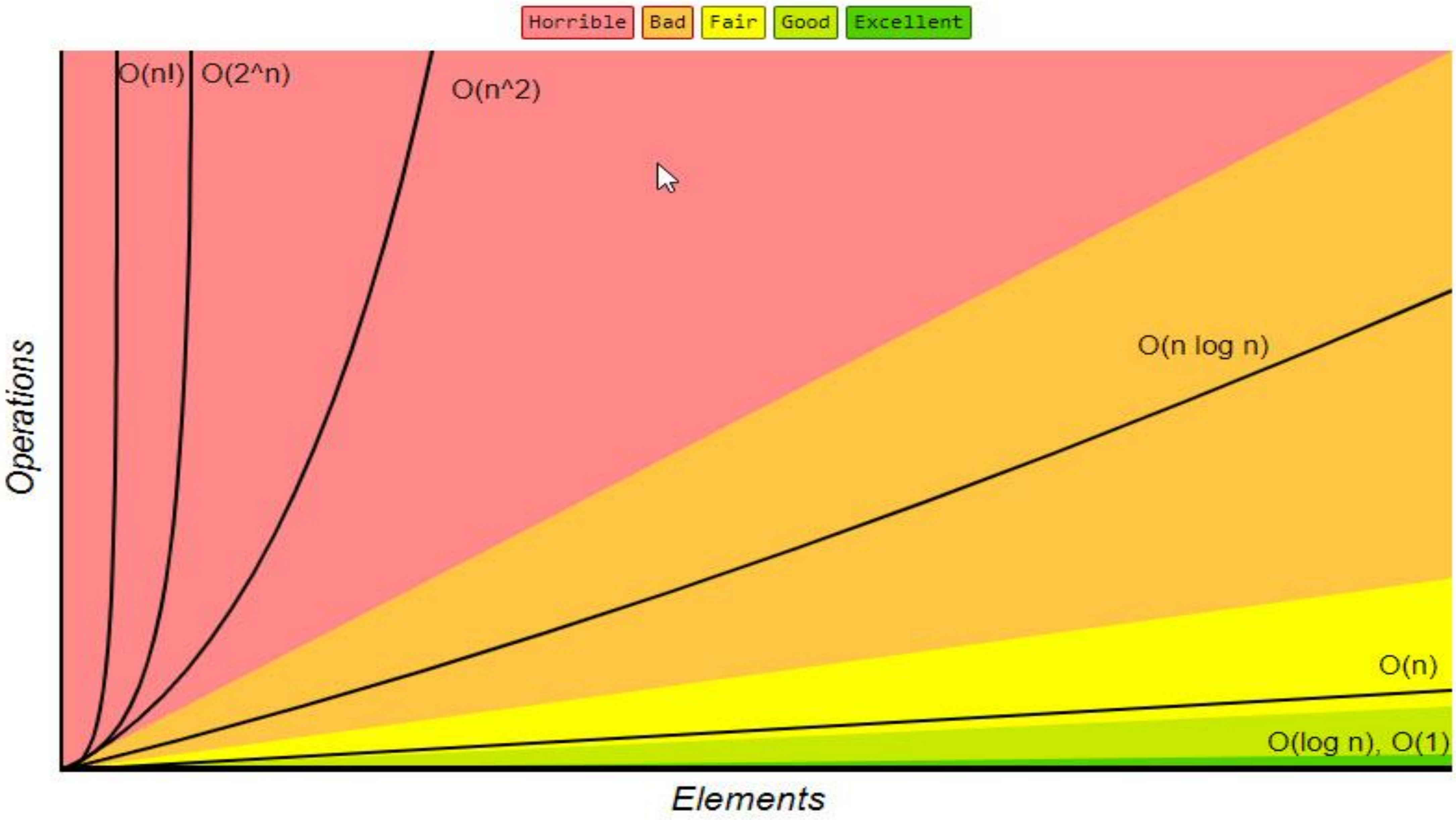


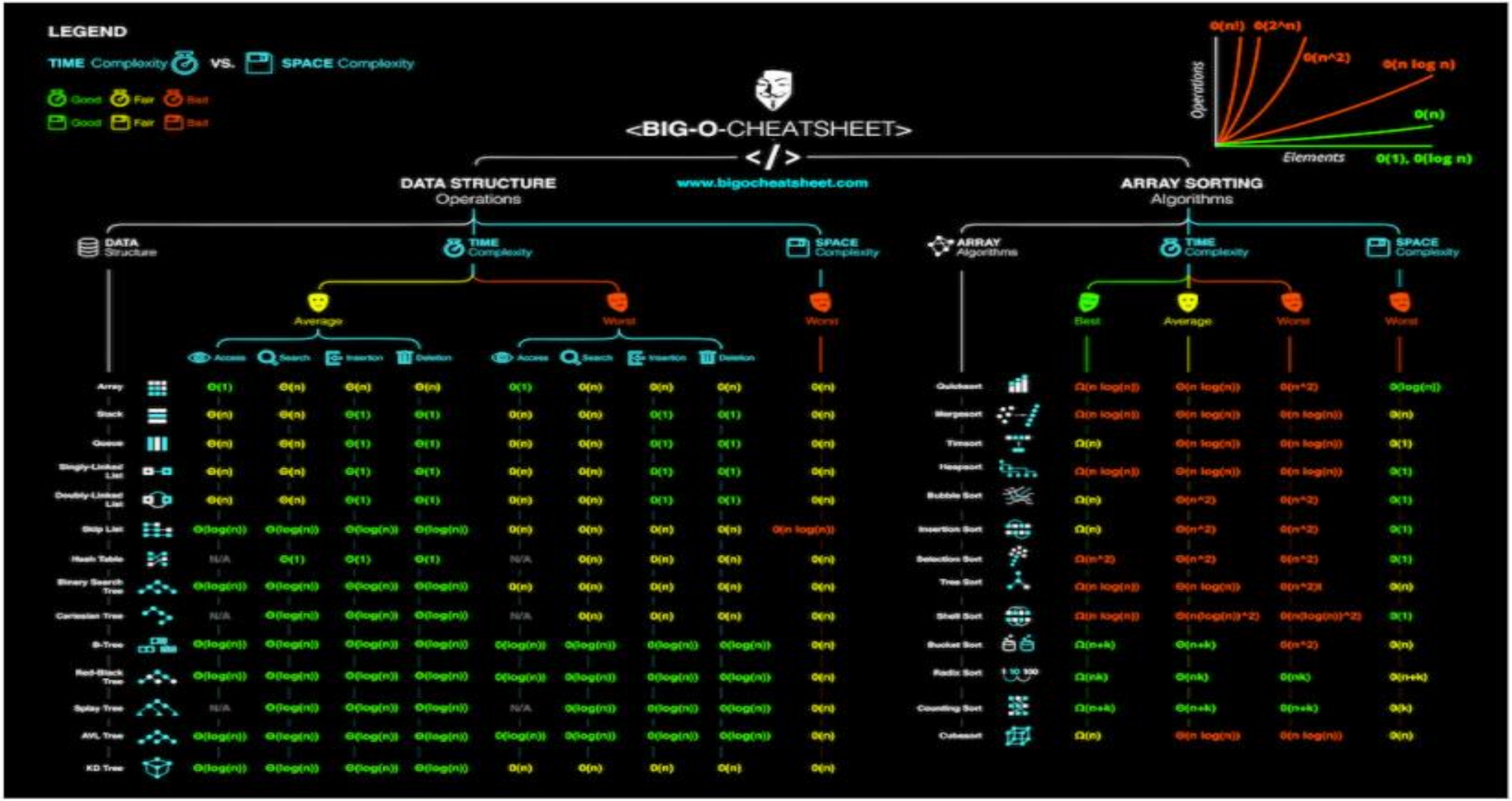


O(2^n)
O(n^2)
O(n)
O(log n)
O(1)
Number of Operations
Input Data Size


O(n^n)
O(n^3)
O(n^2)
O(n)
O(√n)
O(log n)
O(1)
Time
Data Input (Space)

Big-O Complexity Chart





Ref: https://www.bigocheatsheet.com/

Questions

- What about big O notation with (Hard) Real time Systems algorithms ?

- The big O notation is valid with Machine learning and Deep learning algorithms ?

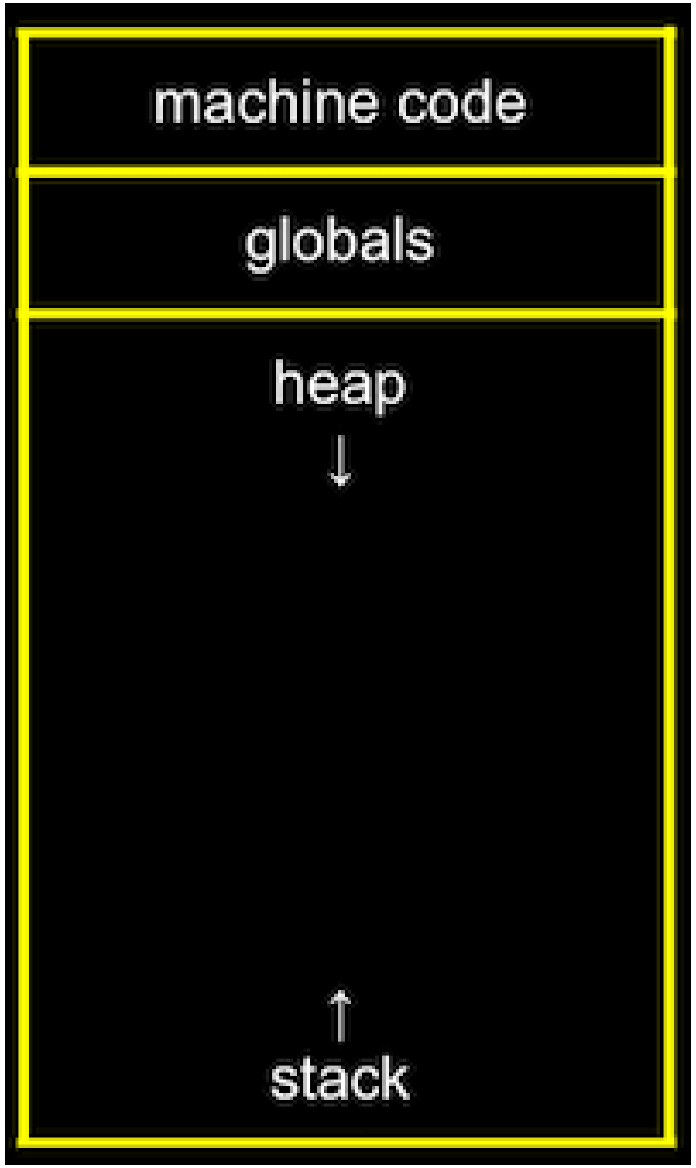
Recursive Factorial

C++

```
int fact (int x){
    if (x == 0){
        return 1;
    } else
        return n * fact(n - 1);
    }
}
```

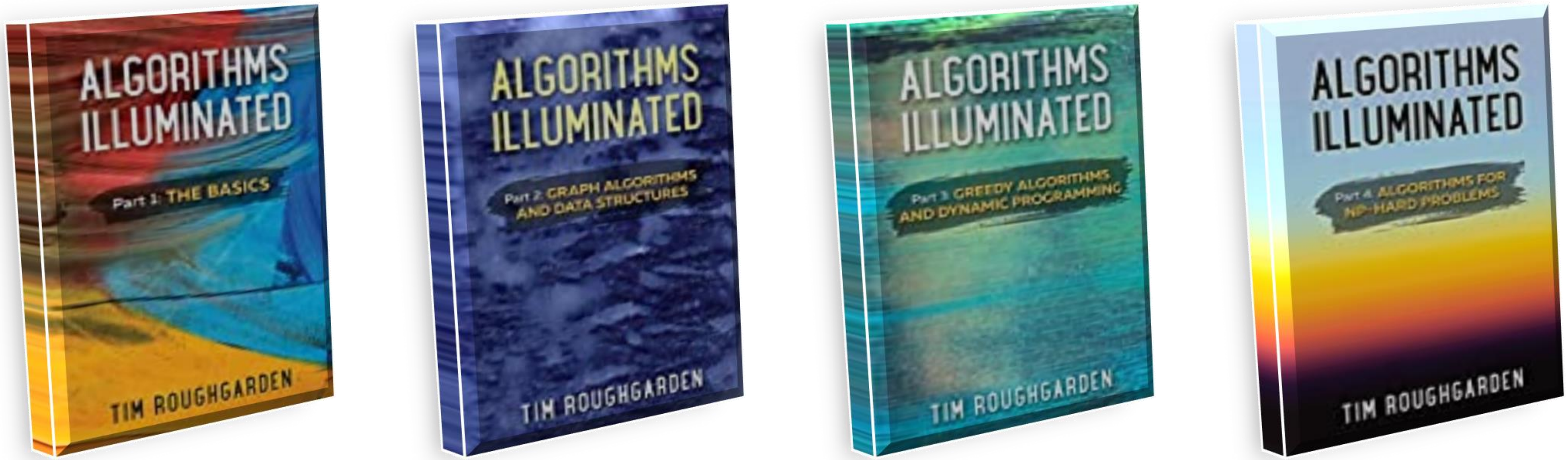
- Prove, using induction that the algorithm returns the values
  - **fact**(0) = 0! =1
  - **fact**(n) = **n**! = **n** * (**n** – 1) * … * 1 if **n** > 0

Memory layout



Textbooks

- Algorithms Illuminated , Tim Roughgarden:
  - Part 1: The Basics (September 2017).
  - Part 2: Graph Algorithms and Data Structures (August 2018)
  - Part 3: Greedy Algorithms and Dynamic Programming (May 2019)
  - Part 4: Algorithms for NP-Hard Problems (July 16, 2020)
  - http://algorithmsilluminated.org/



Reference

Introduction to Algorithms

3rd edition

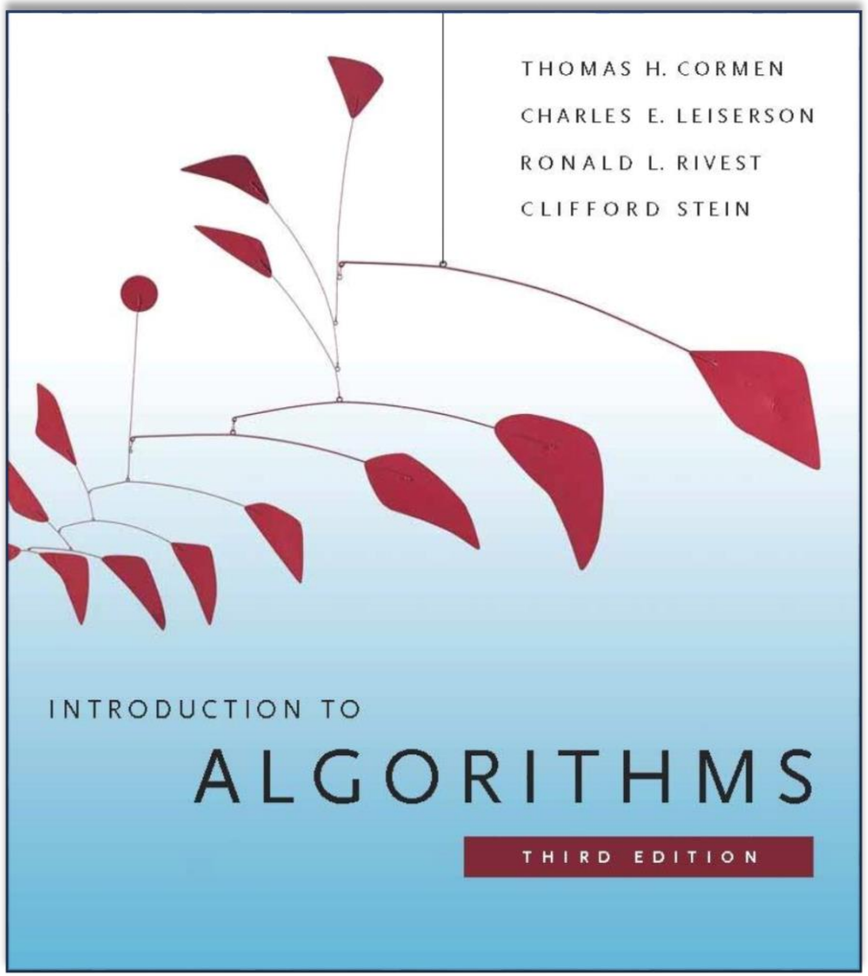
By: Thomas H. Cormen,

Charles E. Leiserson,

Ronald L. Rivest,

Clifford Stein

Published by The MIT Press, 2009



Recommended Video Lectures

1) MIT

"Introduction to Algorithms"

Prof. Erik Demaine,Prof. Srini Devadas

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/

2) MIT

"Introduction to Algorithms"

Prof. Charles Leiserson

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-introduction-to-algorithms-sma-5503-fall-2005/

3) Stanford-Coursera

"Algorithm Specialization"

Tim Roughgarden

https://www.coursera.org/specializations/algorithms

Recommended Video Lectures

4) UC San Diego and National Research University.

“Data Structures and Algorithms Specialization”

https://www.edx.org/micromasters/ucsandiegox-algorithms-and-data-structures

https://www.coursera.org/specializations/data-structures-algorithms

5) Ghassan Shobaki Computer Science Lectures

https://www.youtube.com/playlist?list=PL6KMWPQP_DM8t5pQmuLlarpmVc47DVXWd