# Big O Notation

# Algorithm efficiency

Websites and applications can deal with small to huge amounts of data

Example: Data used by small local restaurant website vs. Google search engine

An inefficient algorithm used with a large set of data will incur high costs in runtime

We do not measure algorithm speed/efficiency by real-time minutes/seconds

This is because computer speeds can vary drastically

Instead, we use Big O Notation

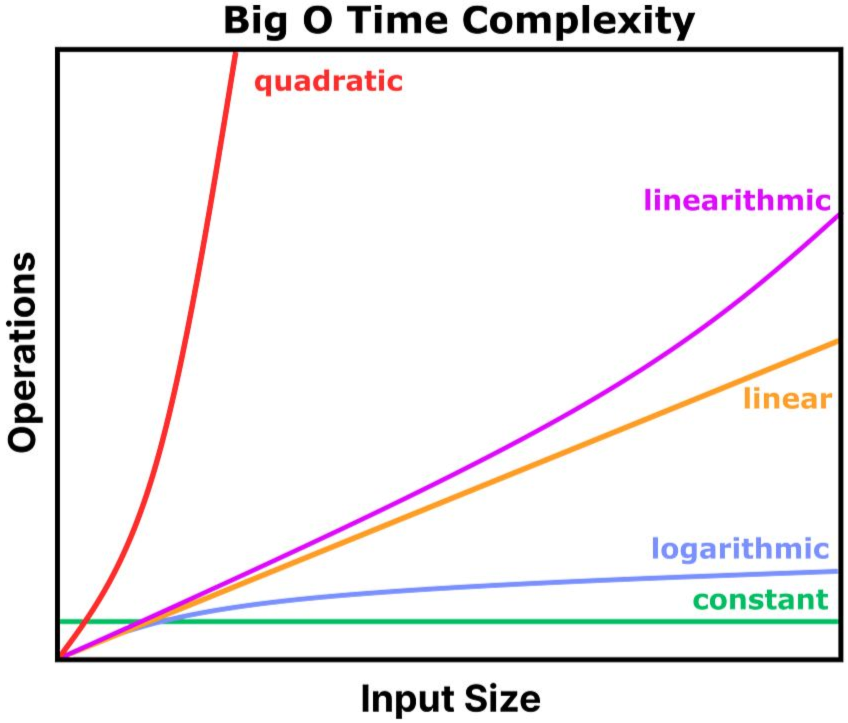
# Big O Notation

Used to analyze algorithms for efficiency

Looks at how increasing the size of inputs given to an algorithm affects the number of operations, or time complexity

We assume that each operation takes a similar amount of time

More operations = more time



Python Fundamentals,
Data Structures, and Algorithms

# Time Complexity

nucamp

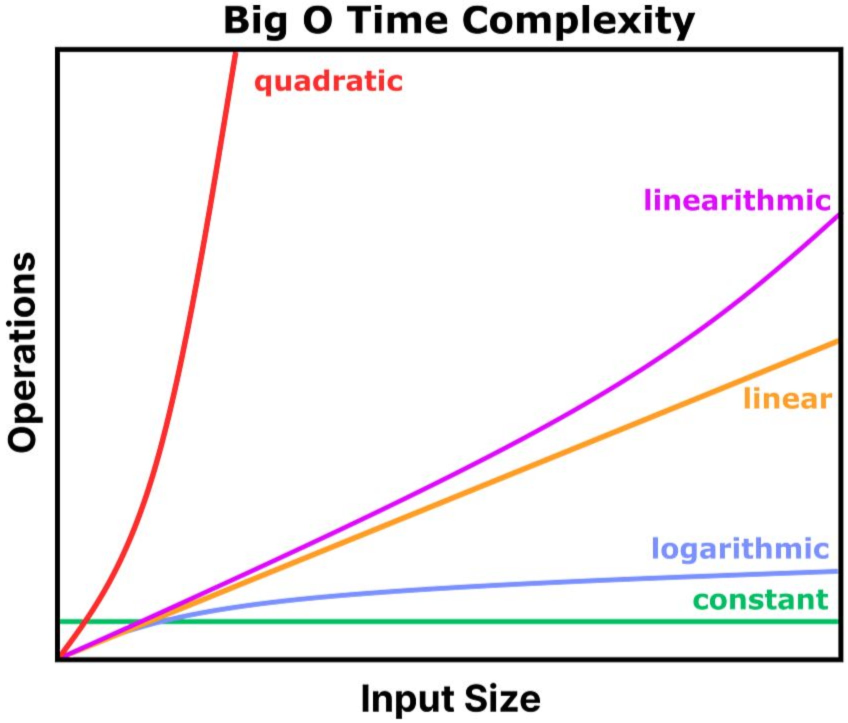
Constant Time: $O(1)$

Linear Time: $O(n)$

Logarithmic Time: $O(\log_2 n)$

Linearithmic Time: $O(n \log_2(n))$

Quadratic Time: $O(n^2)$

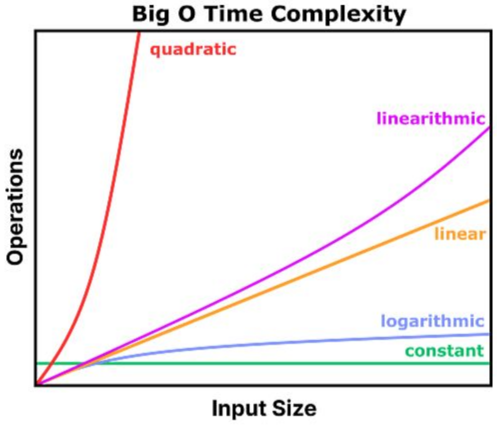


# Constant Time: O(1)

Does not depend on input size.

Examples:

- Declaring a variable
- Retrieving the value of an item in a List by its index
- Appending or popping last item in a List

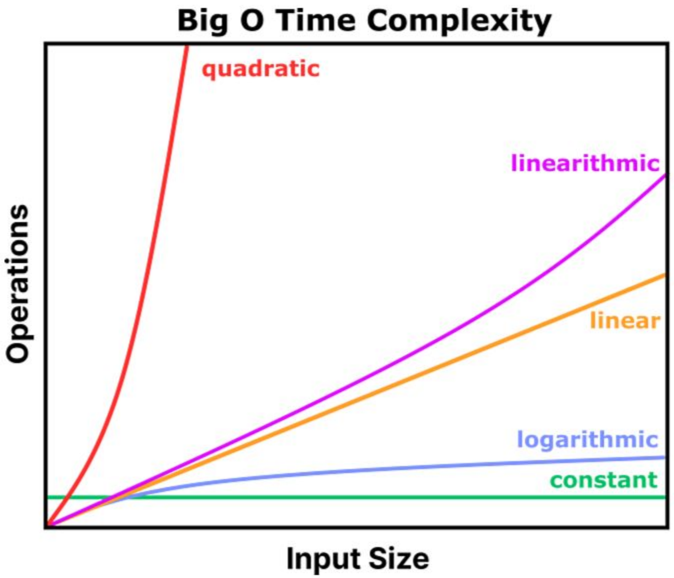


# Linear Time: O(n)

As input size grows, number of operations grows proportionally

Examples:

- Iterating every value in a List or String
- Linear Search
- Creating a list with n items:
  - *[x for x in range(n)]*
  - If n is 10, this will create: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  - If n is 10000, this will create: [0, ..., 9999]



File Edit Selection View Go Run Terminal Help linear_time.py - week5 - Visual Studio Code

linear_time.py

```python
import timeit

print(timeit.timeit("[x for x in range(1000000)]", number=1))
print(timeit.timeit("[x for x in range(10000000)]", number=1))
print(timeit.timeit("[x for x in range(100000000)]", number=1))

```

File Edit Selection View Go Run Terminal Help linear_time.py - week5 - Visual Studio Code

PROBLEMS OUTPUT TERMINAL DEBUG CONSOLE bash

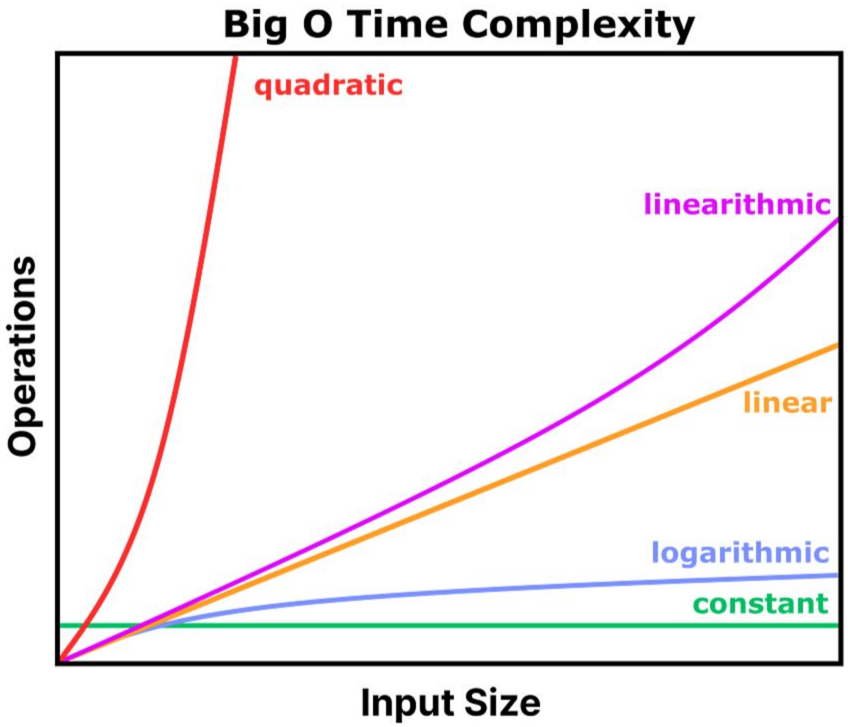
```
minae@eris MINGW64 ~/Desktop/NucampFolder/Python/1-Fundamentals/week5
$ python linear_time.py
0.0619355
0.7298738
9.512747000000001

minae@eris MINGW64 ~/Desktop/NucampFolder/Python/1-Fundamentals/week5
$ python linear_time.py
0.0577234
0.6458704000000001
8.6230712

minae@eris MINGW64 ~/Desktop/NucampFolder/Python/1-Fundamentals/week5
$ python linear_time.py
0.0649287
0.7255295
7.4745477000000005

minae@eris MINGW64 ~/Desktop/NucampFolder/Python/1-Fundamentals/week5
$ python linear_time.py
0.06432639999999999
0.6736384
8.0994443

minae@eris MINGW64 ~/Desktop/NucampFolder/Python/1-Fundamentals/week5
$
```

# Logarithmic Time: $O(\log_2(n))$

Also known as log time

Most efficient time complexity after Constant Time

Often shortened to: O(log (n)) or O(log n)
– base 2 is assumed in computer science



# Logarithms



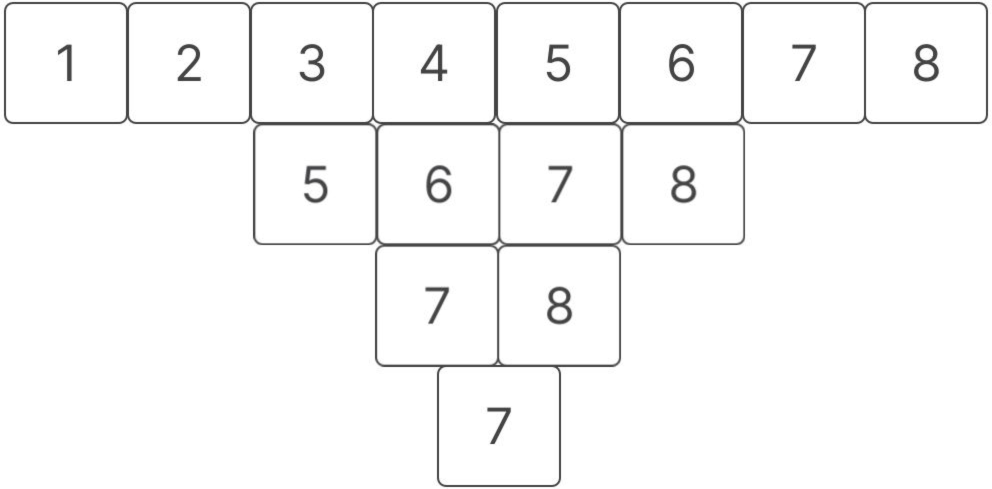
Compare to multiplication & division:
Multiplication: $4 * 5 = 20$
Division: $4 = 20 / 5$
Division is inverse function of multiplication

Logarithms vs exponents:
Exponent: $2^3 = 2 * 2 * 2 = 8$
Logarithm: $3 = \log_2(8)$
Logarithm is inverse function of exponent

# Logarithmic Time: $O(\log_2(n))$

Input is repeatedly partitioned

Example:
Binary Search

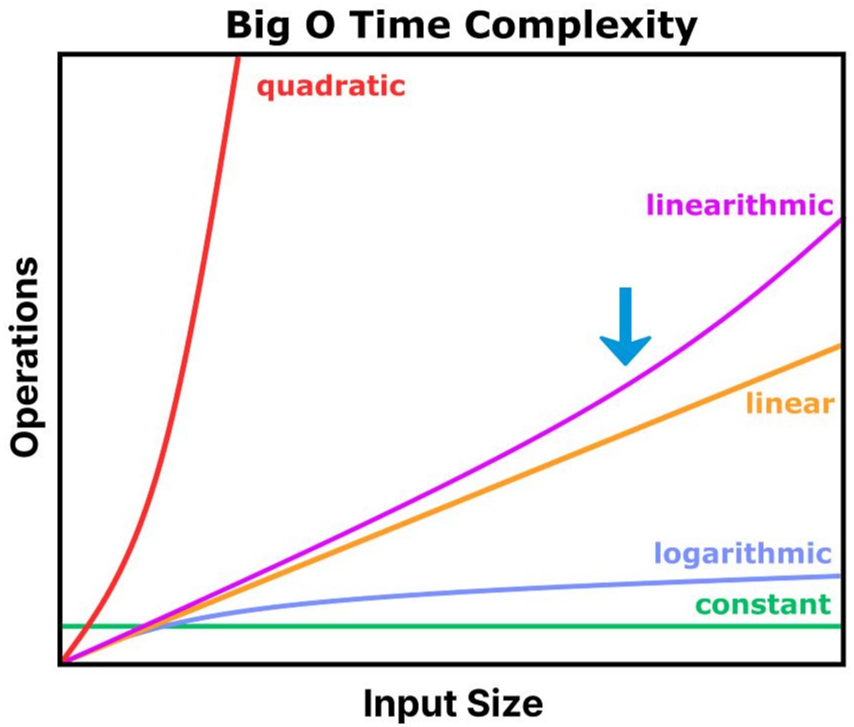


# Logarithmic Time: $O(\log_2(n))$

$$\frac{8}{2^3} = 1 \rightarrow 8 = 2^3 \rightarrow 3 = \log_2(8)$$

$$x = \log_2(n)$$

# Linearithmic Time: $O(n \cdot \log_2(n))$



# Linearithmic Time: $O(n \cdot \log_2(n))$

Depends on input size times the log of the input size

Example: Quicksort

Repeatedly divides in two parts: $\log_2(n)$

Compares pivot against each value: n

| | | | | |
|---|---|---|---|---|
| 65 | 3 | 100 | 10 | 9 |

Combined: $O(n \cdot \log_2(n))$

# Linearithmic Time: $O(n \cdot \log_2(n))$



Depends on input size times the log of the input size

Example: Quicksort

Repeatedly divides in two parts: $\log_2(n)$

Compares pivot against each value: n

| | | | | |
|---|---|---|---|---|
| 10 | 3 | 100 | 9 | 65 |

Combined: $O(n \cdot \log_2(n))$

# Linearithmic Time: $O(n \bullet \log_2(n))$

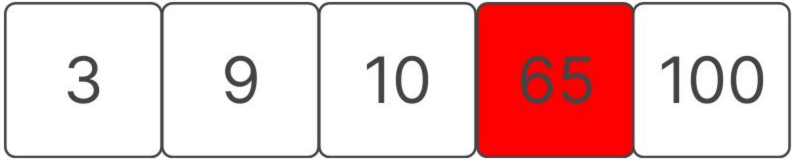
Depends on input size times the log of the input size

Example: Quicksort

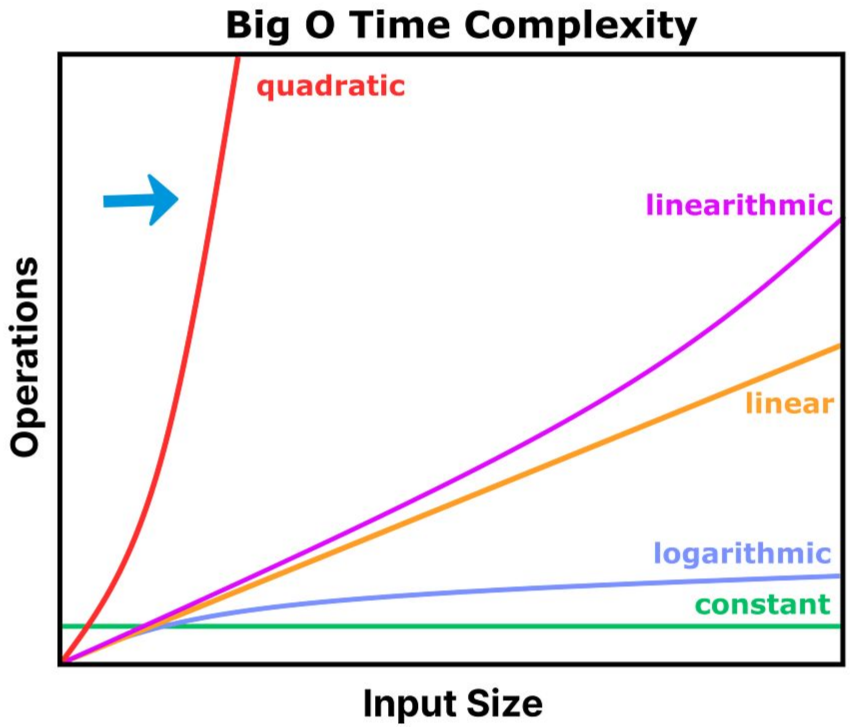
Repeatedly divides in two parts: $\log_2(n)$

Compares pivot against each value: n

Combined: $O(n \bullet \log_2(n))$



# Quadratic Time: O($n^2$)



# Quadratic Time: $O(n^2)$

Based on square of input size
Considered an inefficient algorithm

Example:
BubbleSort

$$(n - 1)(n - 1) = n^2 - 2n + 1$$

OR

$$O(n^2)$$