Introduction to Big O Notation and Time Complexity

Instructor: Yiyang (Ian) Wang

How much time does it take to run this function?

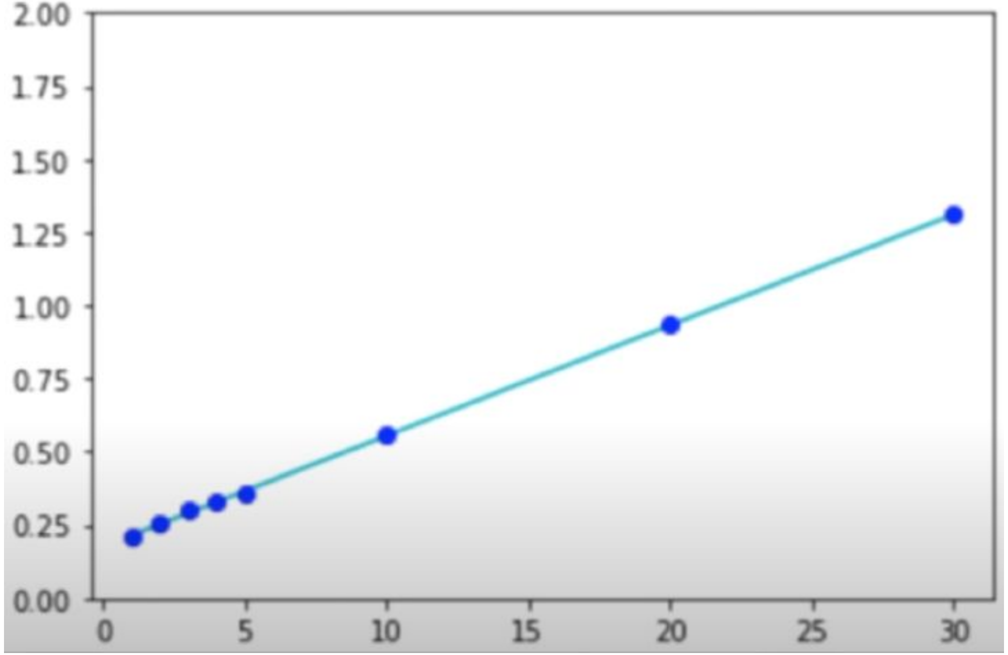
```
given_array = [1, 4, 3, 2, …., 10]

def find_sum(given_array):

    total = 0

    for each i in given_array:

        total +=i

    return total
```

How does the runtime of this function grow?

```
given_array = [1, 4, 3, 2, …., 10]

def find_sum(given_array):

    total = 0

    for each i in given_array:

        total +=i

    return total
```

Size of the input vs. Time it took to run the function

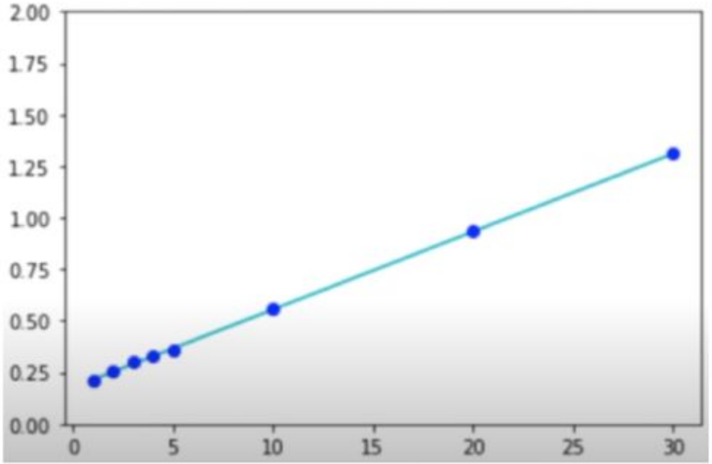
Microsecond
(one million of a second)



Number of elements in the input array

Time Complexity: Linear Time

Microsecond
(one million of a second)



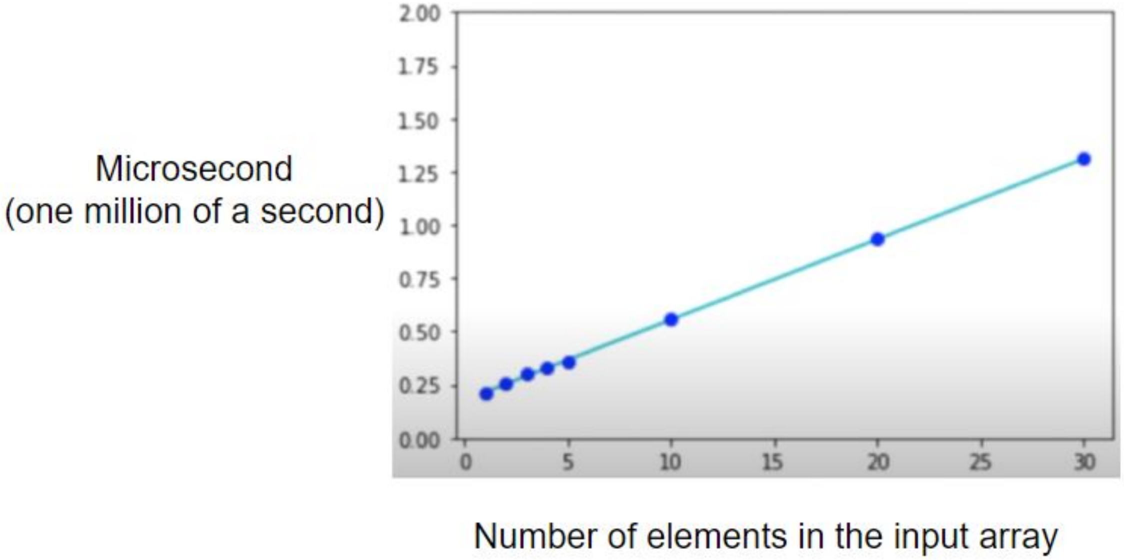
Number of elements in the input array

Time Complexity: a way of showing how the runtime of a function increases **as the size of the input increases**

Big O Notation

- Linear Time: $O(n)$
- Constant Time: $O(1)$
- Quadratic Time: $O(n^2)$

How Do We Find the Time Complexity?



$T = an + b$

1. Find the fastest growing term

2. Take out the coefficient

Excise: What is the Time Complexity?

Example 1:

given_array = [1, 4, 3, 2,..., 10]

```
def silly_function(given_array):
        total = 0
        return total
```

Excise: What is the Time Complexity?

Example 1:

given_array = [1, 4, 3, 2,..., 10]

def silly_function(given_array):

total = 0 $\Rightarrow$ O(1)

return total $\Rightarrow$ O(1)

T = O(1) + O(1) = c1 + c2 = c3 = O(1)

Excise: What is the Time Complexity?

Example 2:

```
given_array = [1, 4, 3, 2,..., 10]

def find_sum(given_array):

    total = 0

    for each i in given_array:

        total +=i

    return total
```

Excise: What is the Time Complexity?

Example 2:

given_array = [1, 4, 3, 2,..., 10]

def find_sum(given_array):

total = 0 ⇨ O(1)

for each i in given_array:

total +=i ⇨ O(1)

return total ⇨ O(1)

T = O(1) + **n**O(1) + O(1) = O(n)

Excise: What is the Time Complexity?

Example 3:

```
array_2d = [[1, 4, 3],[3,1,9],[0,5,2]]

def find_sum_2d (array_2d):
    total = 0
    for each row in array_2d:
        for each i in row:
            total +=i
    return total
```

Excise: What is the Time Complexity?

Example 3:

array_2d = [[1, 4, 3],[3,1,9],[0,5,2]]

```
def find_sum_2d (array_2d):
    total = 0                     → O(1)
    for each row in array_2d:
        for each i in row:
            total +=i             → O(1)
    return total                  → O(1)
```

$$T = O(1) + \mathbf{n^2} O(1) + O(1) = O(n^2)$$

Big O Notation is used to describe the worst-case complexity

- Big O notation describes the **upper bound** of its growth rate
  - The worst-case scenario
    - The maximum amount of time or space the algorithm could require when given a particular input size
  - It may not represent the exact performance of an algorithm but rather an estimation of how it will behave in the worst-case scenario
- Big Omega (Ω)
  - The optimal scenario (lower bound)
- Big Theta (Θ)
  - The average scenario

10^6 instructions/sec, runtimes

N	O(log N)	O(N)	O(N log N)	O(N²)
10	0.000003	0.00001	0.000033	0.0001
100	0.000007	0.00010	0.000664	0.1000
1,000	0.000010	0.00100	0.010000	1.0
10,000	0.000013	0.01000	0.132900	1.7 min
100,000	0.000017	0.10000	1.661000	2.78 hr
1,000,000	0.000020	1.0	19.9	11.6 day
1,000,000,000	0.000030	16.7 min	18.3 hr	318 centuries

Python Big O List

Operation	Example	Complexity Class	Notes
Index	l[i]	O(1)	
Store	l[i] = 0	O(1)	
Length	len(l)	O(1)	
Append	l.append(5)	O(1)	mostly: ICS-46 covers details
Pop	l.pop()	O(1)	same as l.pop(-1), popping at end
Clear	l.clear()	O(1)	similar to l = []
Slice	l[a:b]	O(b-a)	l[1:5]:O(l)/l[:]:O(len(l)-0)=O(N)
Extend	l.extend(...)	O(len(...))	depends only on len of extension
Construction	list(...)	O(len(...))	depends on length of ... iterable
check ==, !=	l1 == l2	O(N)	
Insert	l[a:b] = ...	O(N)	
Delete	del l[i]	O(N)	depends on i; O(N) in worst case
Containment	x in/not in l	O(N)	linearly searches list
Copy	l.copy()	O(N)	Same as l[:] which is O(N)
Remove	l.remove(...)	O(N)	
Pop	l.pop(i)	O(N)	O(N-i): l.pop(0):O(N) (see above)
Extreme value	min(l)/max(l)	O(N)	linearly searches list for value
Reverse	l.reverse()	O(N)	
Iteration	for v in l:	O(N)	Worst: no return/break in loop
Sort	l.sort()	O(N Log N)	key/reverse mostly doesn't change
Multiply	k*l	O(k N)	5*l is O(N): len(l)*l is O(N**2)

Python Sets Big O

```
                                  Complexity
Operation     | Example        | Class           | Notes
--------------+----------------+-----------------+-------------------------------
Length        | len(s)         | O(1)            |
Add           | s.add(5)       | O(1)            |
Containment   | x in/not in s  | O(1)            | compare to list/tuple - O(N)
Remove        | s.remove(..)   | O(1)            | compare to list/tuple - O(N)
Discard       | s.discard(..)  | O(1)            |
Pop           | s.pop()        | O(1)            | popped value "randomly" selected
Clear         | s.clear()      | O(1)            | similar to s = set()

Construction  | set(...)       | O(len(...))     | depends on length of ... iterable
check ==, !=  | s != t         | O(len(s))       | same as len(t); False in O(1) if
                                                   the lengths are different
<=/<          | s <= t         | O(len(s))       | issubset
>=/>          | s >= t         | O(len(t))       | issuperset s <= t == t >= s
Union         | s | t          | O(len(s)+len(t))
Intersection  | s & t          | O(len(s)+len(t))
Difference    | s - t          | O(len(s)+len(t))
Symmetric Diff| s ^ t          | O(len(s)+len(t))

Iteration     | for v in s:    | O(N)            | Worst: no return/break in loop
Copy          | s.copy()       | O(N)            |
```

Python Dictionary Big O

Operation	Example	Complexity Class	Notes
Index	d[k]	O(1)	
Store	d[k] = v	O(1)	
Length	len(d)	O(1)	
Delete	del d[k]	O(1)	
get/setdefault	d.get(k)	O(1)	
Pop	d.pop(k)	O(1)	
Pop item	d.popitem()	O(1)	popped item "randomly" selected
Clear	d.clear()	O(1)	similar to s = {} or = dict()
View	d.keys()	O(1)	same for d.values()
Construction	dict(...)	O(len(...))	depends # (key,value) 2-tuples
Iteration	for k in d:	O(N)	all forms: keys, values, items
			Worst: no return/break in loop

Magic Commands

magic commands are special commands that start with a % (for line magics) or %% (for cell magics) and provide various convenient functions for controlling the environment, running code, and accessing system-related information. For example:

- %run is used to run external Python scripts.
- %matplotlib inline is used to display Matplotlib plots inline within the notebook.
- %%time is used to measure the execution time of a code cell.
- %load_ext and %reload_ext are used to load and reload extensions.

Magic Commands

```
%time sum(range(10000))
```

```
CPU times: user 236 µs, sys: 35 µs, total: 271 µs
Wall time: 283 µs
49995000
```

```
%%time
total = 0
for i in range(10000):
    total += i
```

```
CPU times: user 2.17 ms, sys: 0 ns, total: 2.17 ms
Wall time: 2.21 ms
```

User time, Sys time, and Wall time

- **User time**
    - User Time refers to the amount of time the CPU spends executing your program's code (or user code). This includes the time the CPU spends running your code, as well as the time spent in functions called by your code.
    - This is useful to measure how much processing time is being used by your program's instructions.
    - If your program performs a lot of computations or data processing, the User Time will represent the time spent performing those calculations.

User time, Sys time, and Wall time

- **System Time (Sys Time):**
  - System Time refers to the amount of time the CPU spends executing system calls on behalf of your program. This includes time spent in the kernel mode of the operating system, such as handling I/O operations (e.g., reading from or writing to a disk, managing memory).
  - This is useful for understanding how much time your program spends interacting with the operating system, such as reading or writing files, or waiting for system resources.
  - If your program is performing a lot of file I/O or other system-level operations, System Time will be higher.

User time, Sys time, and Wall time

- **Wall time**
  - Wall Time (also known as Real Time) is the total elapsed time from the start to the end of a program's execution. It includes all time: User Time, System Time, as well as time when the CPU is idle or waiting (e.g., waiting for I/O operations to complete).
  - Wall Time is the actual time that has passed, as you would measure with a stopwatch. It is the time the user experiences.
  - If your program waits for user input, performs network operations, or reads from a slow disk, these waiting periods will increase Wall Time but not necessarily increase User Time or System Time.

User time, Sys time, and Wall time

- Wall Time = User Time + System Time + Time spent waiting (e.g., for I/O, network delays).
- User Time and System Time represent the CPU's perspective of how busy it was with your program, while Wall Time represents the actual clock time that passed from the beginning to the end of your program's execution.

Other Magic Commands

- %run ./mycode.py
  - Runs the command in a shell
- %system
  - Provides a shell
- %time
  - Times the current line of code
- %load_ext autoreload
- %autoreload 2
  - Reloads modules before executing user code
- %matplotlib inline
  - Makes sure Jupyter will show your plots
- %who
  - Lists all variables in the global scope
- %pdb
  - Python debugger
- %store data
- %store –r data (in the other notebook)
  - Passes variables between notebooks

Space Complexity

- What is it?

  Time complexity measures the time to run program, space complexity measures ***memory usage*** of a specific program.

- What makes space complexity increase?
  - Assigning variables
  - Creating new data structures
  - Function calling and allocation

Space Complexity in Python

```
# Notice how much overhead Python objects have
# A raw integer should be 64 bits or 8 bytes only

print sys.getsizeof(1)
print sys.getsizeof(1234567890123456789012345678901234567890)
print sys.getsizeof(3.14)
print sys.getsizeof(3j)
print sys.getsizeof('a')
print sys.getsizeof('hello world')
```

```
24
44
24
32
38
48
```

How much space does this data take?

- Use sys.getsizeof()

Returns the number of bytes

Let’s try it!

- Download, following the instructions in the “Python Benchmarking” notebook