



# Complexity

Lab05



# Measuring the Run Time of an Algorithm

---

- ▶ One way to measure the time cost of an algorithm is to use computer's clock to obtain actual run time
  - ▶ Can use `time()` in `time` module
- ▶ Another technique is to **count the instructions** executed with different problem sizes
- ▶ A **primitive Operation** takes a unit of time. The actual length of time will depend on external factors such as the hardware and software environment
  - ▶ Each of these kinds of operation would take the same amount of time on a given hardware and software environment
    - ▶ Assigning a value to a variable
    - ▶ Calling a method.
    - ▶ Performing an arithmetic operation.
    - ▶ Comparing two numbers.
    - ▶ Indexing a list element.
    - ▶ Returning from a function

► Consider the following function:

```
def rate(number):  
    i = 0  
    while i < 10:  
        i += 1
```

You should count the line "while i < n:" as being executed each time the condition is checked. Note that a loop condition is checked 1 time more than the loop body is executed.

- $i=0$  <- one step
- while loop -> 11 comparisons + 10 increments = 21 steps
- Total = 22 steps

► Consider the following function:

```
def rate(n):  
    i = 0  
    while i < n:  
        i = i + 1
```

- $i=0$  <- one step
- while loop ->  $n+1$  comparisons +  $n$  increments =  $2n + 1$  steps
- Total =  $2n + 2$  steps
- If  $n$  is 10, total = 22, If  $n$  is 100, total = 202 ...



# loops

- ▶ Consider the following function:
  - ▶  $i=1 \rightarrow 1$  step
  - ▶  $total=0 \rightarrow 1$  step
  - ▶ while loop  $\rightarrow$ 
    - ▶  $i=1, 2, 4, 8 \rightarrow 4$  comparisons
    - ▶ Calculate the total = execute 3 times
    - ▶ Increment step = execute 3 times
    - ▶ Total of the while loop =  $(3 + 1 + 3 + 3) = 10$
  - ▶ return  $\rightarrow 1$  step
  - ▶ Total =  $2 + 10 + 1 = 13$  steps

```
def rate(n):  
    i = 1  
    total = 0  
    while i < 8:  
        total = i + total  
        i *= 2  
    return total
```

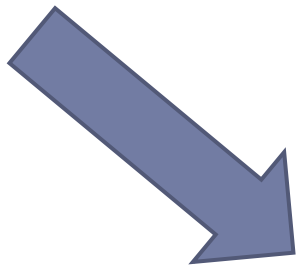
# Programming Question

- ▶ Consider the following example:

```
def rate(n):  
    i = 0  
    while i < n:  
        i = i + 1
```

n=10, number of operations = 22  
n=100, number of operations = 202

- ▶ Modify the function to print the number of operations



```
def rate(n):  
    count = 2  
    i = 0  
    while i < n:  
        count += 2  
        i += 1  
    print(...format(count))
```

Set up initial value:  
2 steps: i=0, comparison step

while loop:  
1 comparison and 1 increment

# Calculating Nested Complexity

```
▶ def my_nested_function(n):  
    i = 0                                # runs 1 time  
    total = 1                            # runs 1 time  
    while i < n:                          # runs n + 1 times  
        total += n                       # runs n times  
        j = 0                            # runs n times  
        while j < n:                    # runs n times ???  
            total += j  
            j += 1  
        i += 1  
    return total
```

▶ j = 0

```
▶ while j < n:  
    total += n  
    j += 1
```

# runs n + 1 times  
# runs n times  
# runs n times

# Calculating Nested Complexity

► def my\_nested\_function(n):

    i = 0

        # runs 1 time

    total = 1

        # runs 1 time

    while i < n:

        # runs n + 1 times

        total += n

        # runs n times

        j = 0

        # runs n times

        while j < n:

            # runs n(n + 1) times

            total += n

            # runs n times

            j += 1

            # runs n \* n times

        i += 1

        # runs n times

    return total

        # runs 1 time

# runs  $3n^2 + 5n + 4$  times in total

► j = 0

► while j < n:

    # runs n(n + 1) times

    total += n

    # runs n times

    j += 1

    # runs n times

Multiply by n



# Calculating Nested Complexity

## The Fast Way! (Big O)

```
▶ def my_nested_function(n):
```

```
    i = 0
```

```
    total = 1
```

```
    while i < n:
```

```
        total += n
```

# this outer while runs ~ n times

```
        j = 0
```

```
        while j < n:
```

```
            total += n
```

# By itself, this inner while loop runs ~ n times

```
            j += 1
```

```
        i += 1
```

```
    return total
```

# We multiply the inner while loop by the outer loop and get  $n^2$  as the rough number of operations in this code.

# Nothing more complicated seems to be happening giving the program an  $O(n^2)$  complexity





# Different Big O complexities

---

- ▶ We have only looked at the cases where  $i$  increases by 1 during each iteration of the while loop.
- ▶ If  $i$  changes in other ways we get different complexities:
  - ▶  $i -= 1$  Still gives  $O(n)$  complexity
  - ▶  $i *= 2$  Gives  $O(\log(n))$  complexity
  - ▶  $i /= 2$  Gives  $O(\log(n))$  complexity