Algorithm Complexity:
An Introduction

Lesson Outcomes

At the end of this lesson, you should be able to:

- State the concept of time complexity in algorithm implementation
- Explain the Big O notation
- Describe how Big O can be used to compare time complexity of algorithms

Program Complexity Evaluation

Recall **Algorithms**

- step-by-step instructions given to computer on how to solve a given problem

Sequence: Step 1 → Step 2 → Step 3 …

- multiple possible algorithms as well as implementations
- how to compare them?
- how to evaluate which is the ‘better’ algorithm?

Program Complexity Evaluation (Cont'd)

Comparison in terms of execution time

00:00:01 **vs.** 00:00:10

Shorter execution time ⇒ better?

But execution time depends on many factors

- the speed of the computer
- the way the algorithm is implemented
  - pre-compute lookup table
  - loop unrolling technique
  - input data value and input data size

Instruction Steps

Count the number of instructions it takes to execute the algorithm

- independent of the computer
- more steps ⇒ longer execution time

But the number of steps may still depend on the data involved in the computation

Example

```
def linearSearch(List, item_x):
    for i in range(len(List)):
        if List[i] == item_x:
            return i
    return -1
```

Asymptotic Behavior

More important to consider the worst case situation

- when item_x is not in the List
- number of instruction steps will be the most

As the number of entries in the List increases

- number of steps under worst case situation also increases

Note that

- ‘input’ data size ≡ number of entries in the List

In time complexity analysis

- growth pattern of the number of steps as input data size increases indefinitely
- asymptotic behavior of running time – Big O

Example

```
def linearSearch(List, item_x):
        for i in range(len(List)):
            if List[i] == item_x:
                return i
        return -1
```

Big O

BIG O Notation

- measure and compare the time complexity of algorithms
- pattern of execution time of algorithm as input data size grows

Execution Time

- in terms of the number of instruction steps
- for worst case situation

Big O gives an upper bound on the asymptotic growth of an algorithm

Analysis of a Linear Search Algorithm

Assumption

- each line of the code statement can be executed in one step

Worst case

- item_x not in the list (of length n)
- T(n) = 2 + 4n + 1 = 3 + 4n

T(n) = 3 + 4n

Example

```
def linearSearch(List, item_x):
    max_pos = len(List)
    i = 0
    while i < max_pos:
        if List[i] == item_x:
            return i
        i = i + 1
    if i == max_pos:
        return - 1
```

1
1
n times
1 x n
-
1 x n
1 x n
1

n = ‘Input’ Data size ≡ number of entries in the List

Growth Order of T(n)

T(n) = 3 + 4n

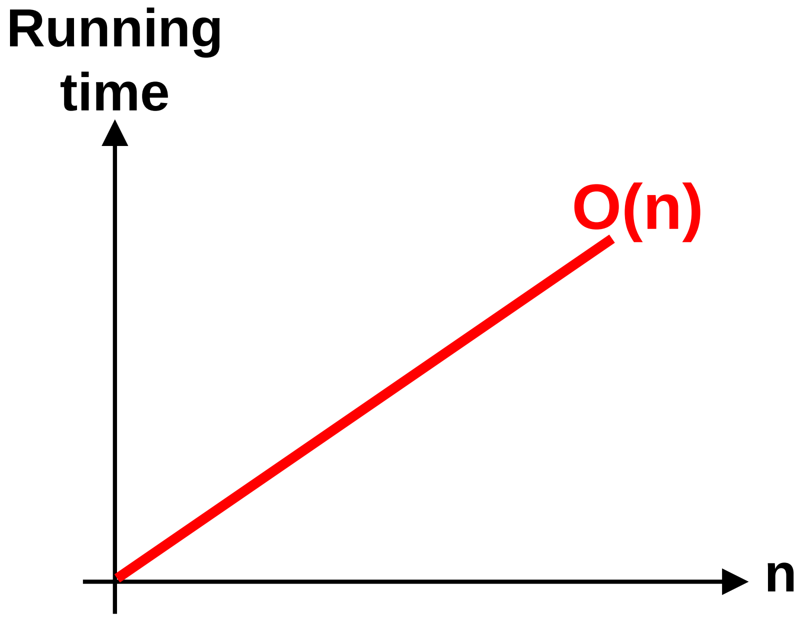
- minimum value of T = 3
- when n >> 3, T → 4n
  - e.g. for n = 1000
    T = 3 + 4000 = 4003 ≈ 4000
- T(n) ≈ 4n for large n

Asymptotic behavior

- T(n) increases proportionally with n
  - e.g. T(n) doubles when n is doubled
- Growth order: f(n) = n

Complexity using Big O notation: O(f(n)) = O(n)

- Linear complexity



Program Complexity Types

Common Types of Complexity

$O(1)$	**Constant complexity**, where $f(n) = 1$
$O(n)$	**Linear complexity**, where $f(n) = n$
$O(\log n)$	**Logarithmic complexity**, where $f(n) = \log n$
$O(n^K)$	**Polynomial complexity**, where $f(n) = n^k$, with k = constant e.g. $O(n^2)$ = Quadratic complexity
$O(K^n)$	**Exponential complexity**, where $f(n) = k^n$

Constant Complexity O(1)

$O(1) \equiv$ Constant Complexity
Algorithm always uses the same amount of time to execute for all inputs

Example: *Pre-compute lookup table-based algorithm*

Example 1

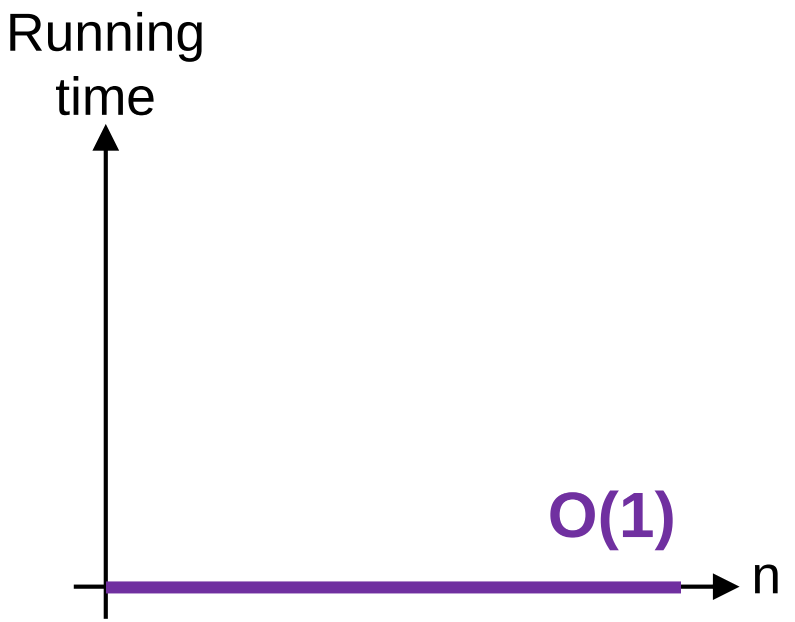
```
def lookup(position)
       return
lookupTable(position)
```

$T(n) = 1 \equiv 1.1 \Rightarrow f(n) = 1;$
$O(fn) = O(1)$

Example 2

```
def lookup(position)
       print("Entry position = " , position)
       return lookupTable(position)
```

$T(n) = 2 \equiv 2.1 \Rightarrow f(n) = 1; O(fn) = O(1)$



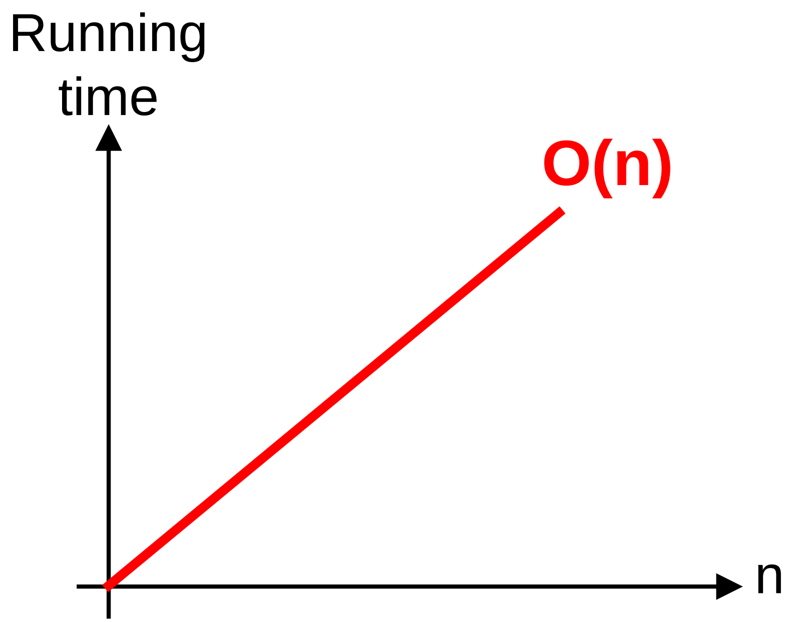
Linear Complexity O(n)

O(n) ≡ Linear Complexity
Algorithm execution time increases linearly in proportion with (input) data size

Example

```
def linearSearch(List, item_x):
     for i in range(len(List)):
         if List[i] == item_x:
             return i
     return -1
```

f(n) = len(List)   O(fn) = O(len(List)) = O(n)



Polynomial Complexity $O(n^K)$

$O(n^K) \equiv$ Polynomial Complexity

Occurs for algorithm that contains nested loops

Example: *Quadratic complexity $O(n^2)$*

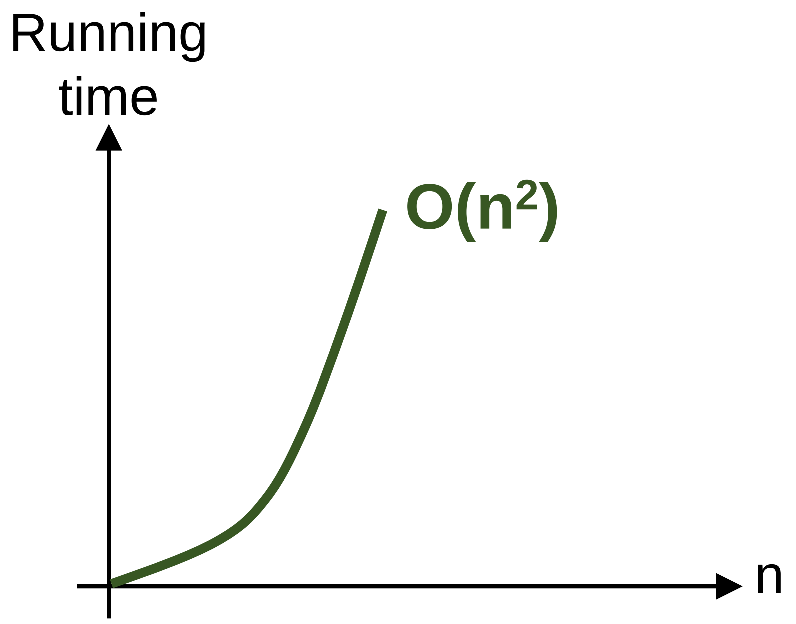
- execution time is proportional to the square of the input data size

Example: Quadratic Complexity $O(n^2)$

```
def checkDuplicate(List1, List2):
    for i in range(len(List1)):
  for j in range(len(List2)):
            if List1[i] == List2[j]:
                return List1[i]
    return -1
```

$f_{in}(n)$ = len(List2); $f_{out}(n)$ = len(List2)

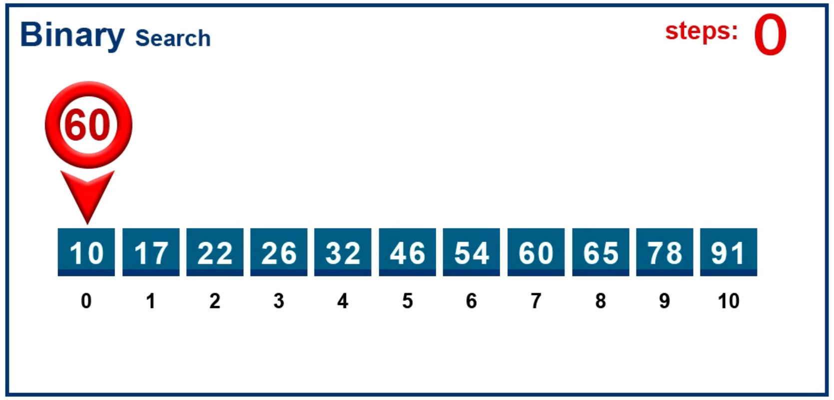
$O(fn) = O(f_{in}(n).\ f_{out}(n)) \equiv O(n.n) = O(n^2)$



Logarithmic Complexity O(log n)

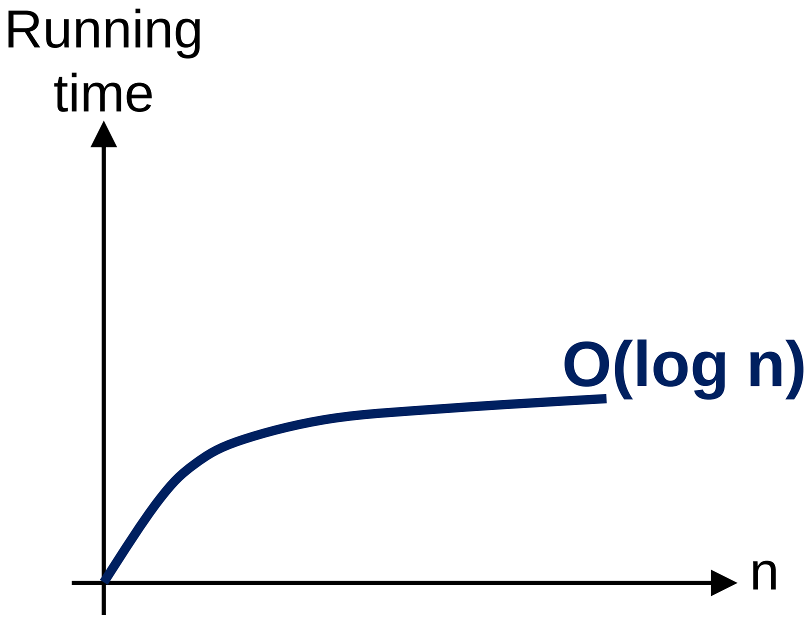
O(log n) ≡ Logarithmic Complexity
Execution time grows as the log of input

Example: Binary Search



$(n) = \log_2(\text{len}(\text{List}))$

$O(fn) = O(\log_2(\text{len}(\text{List})\ ) \equiv O(\log n)$



Exponential Complexity $O(k^n)$

$O(k^n)$ ≡ Exponential Complexity | Occurs for algorithm that contains recursive call

Example:

Fibonacci sequence computation

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 …..

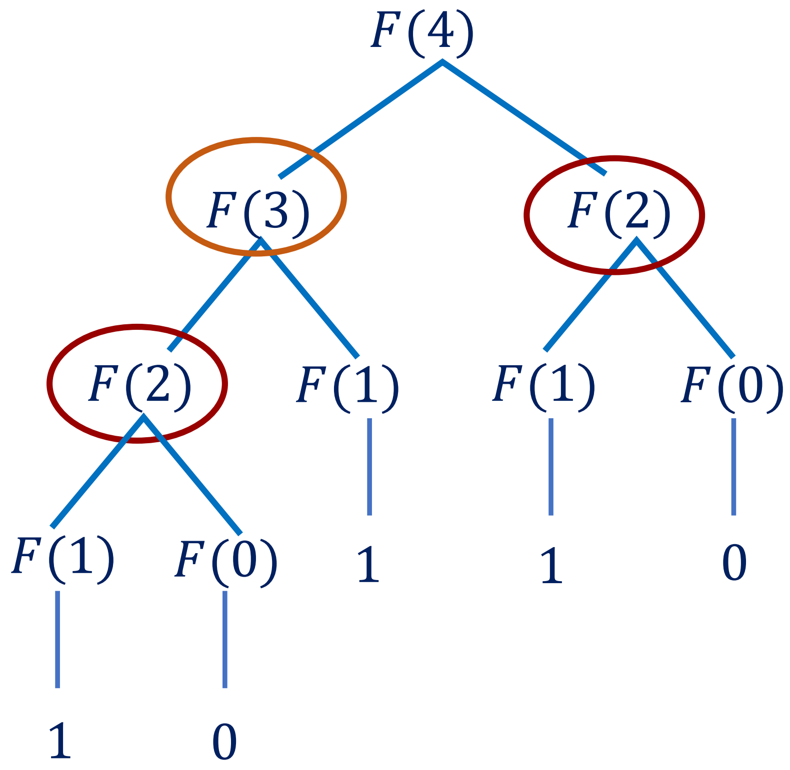
Each number can be derived based on the formula:

$$F(n) = F(n-1) + F(n-2)\,, n \geq 2$$

Example

```
def rFib (n):
     if (n == 0):
          return 0
     elif (n == 1):
          return 1
     elif (n > 1):
          return (rFib(n-1) + rFib(n-2))
     else:
          return -1
```

Recursion Fibonacci

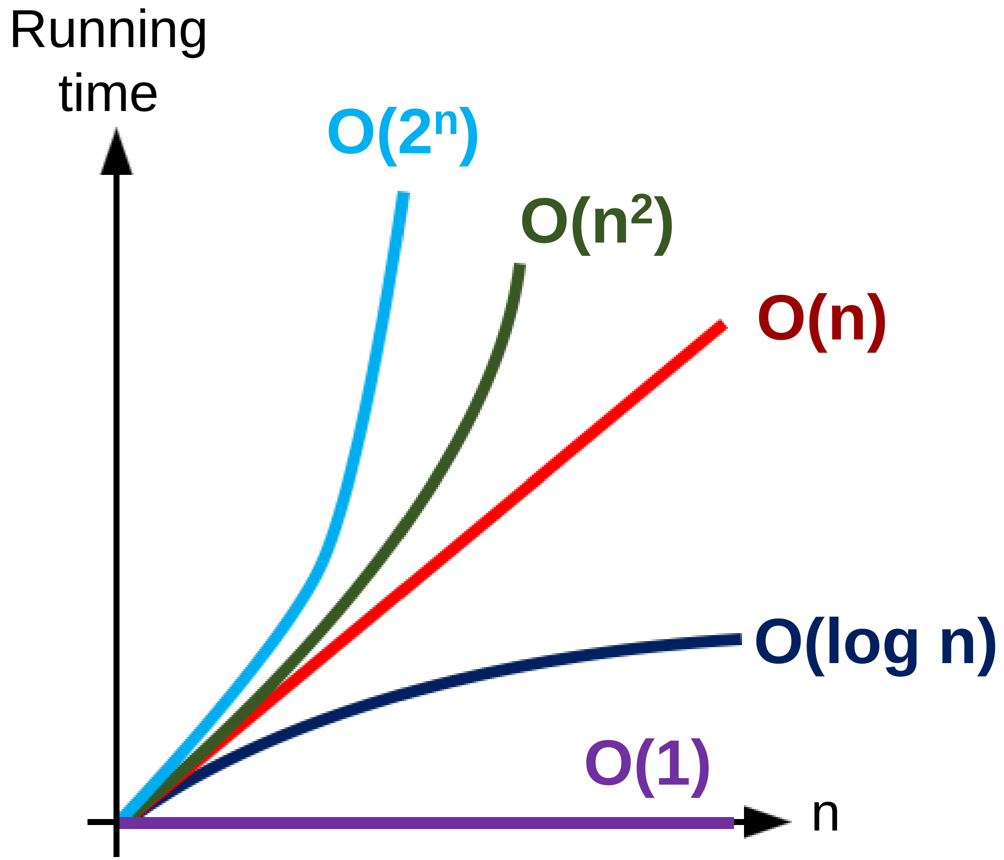


f(n) = 2^n O(f(n)) = O(2^n)

Example

```
def rFib (n):
      if (n == 0):
            return 0
      elif (n == 1):
            return 1
      elif (n > 1):
            return (rFib(n-1) + rFib(n-2))
      else:
            return -1
```

Rate of Growth



Iteration Fibonacci

An alternative algorithm for Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 …..

Computation based on iteration:

$$\boldsymbol{F(n) = F(n-1) + F(n-2)\,,\, n \geq 2}$$

```
def iFib(n):
      if (n == 0):
          return 0
      elif (n == 1):
          return 1
      elif (n == 2):
          return 1
      elif: # n > 2
      #continue in next
```

```
elif :   # n > 2
          fn0 = 0
          fn1 = 1
          fn2 = 1
          for i in range(n-2):
              fn0 = fn1
              fn1 = fn2
              fn2 = fn0+fn1
          return fn2
```

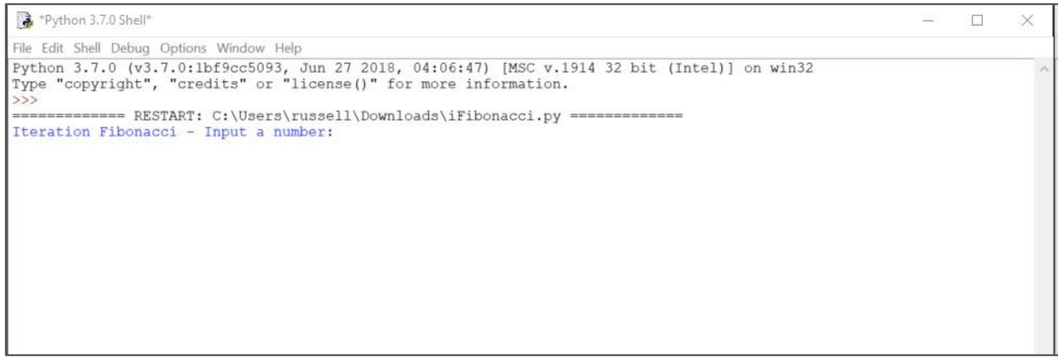
T(n) = 4n + 4

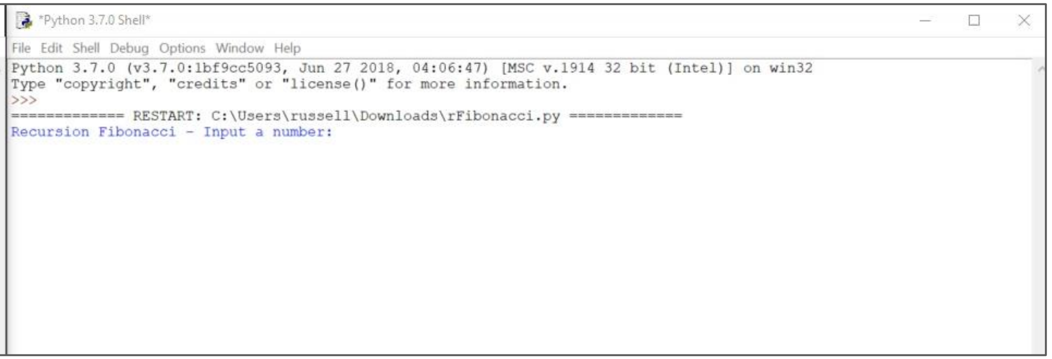
f(n) = n

$\Rightarrow$ O(n)

i.e. Linear Complexity

Iteration vs. Recursion Fibonacci Computation





Iteration vs. Recursion Fibonacci Computation (Cont’d)

Summary

Algorithm complexity can be analyzed

- time complexity

Big O

- worst case analysis
- order of growth pattern as input size grows

Asymptotic behavior of algorithms

- useful for comparing and classifying algorithms

Different algorithms for same problem

- compared based on Big O

References for Images

No.	Slide No.	Image	Reference
1	4		Question problem [Online Image]. Retrieved April 18, 2018 from https://pixabay.com/en/question-problem-think-thinking-622164/.
2	4		Search [Online Image]. Retrieved April 18, 2018 from https://pixabay.com/en/database-search-database-search-icon-2797375/.
3	5		Search [Online Image]. Retrieved April 18, 2018 from https://pixabay.com/en/stopwatch-timer-watch-seconds-34107/.