COMPSCI 105 S1 2017
Principles of Computer Science

Algorithm Analysis/Complexity

Agenda & Reading

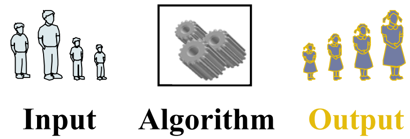
- Agenda:
  - Introduction
  - Counting Operations
  - Big-O Definition
  - Properties of Big-O
  - Calculating Big-O
  - Growth Rate Examples
  - Big-O Performance of Python Lists
  - Big-O Performance of Python Dictionaries
- Reading:
  - Problem Solving with Algorithms and Data Structures
    - Chapter 2

1 Introduction
What Is Algorithm Analysis?

- How to **compare** programs with one another?
- When two programs solve the same problem but look different, is one program **better** than the other?
- What **criteria** are we using to compare them?
  - Readability?
  - Efficient?
- Why do we need **algorithm analysis/complexity** ?
  - Writing a working program is not good enough
  - The program may be inefficient!
  - If the program is run on a large data set, then the running time becomes an issue

1 Introduction
Data Structures & Algorithm

- Data Structures:
  - A systematic way of **organizing** and **accessing** data.
  - No single data structure works well for **ALL** purposes.



- Algorithm
  - An algorithm is a step-by-step procedure for solving a problem in a finite amount of time.
- Program
  - is an algorithm that has been encoded into some programming language.
  - Program = data structures + algorithms

1 Introduction

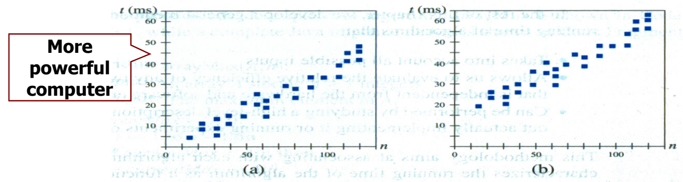
Algorithm Analysis/Complexity

- When we analyze the **performance** of an algorithm, we are interested in how **much** of a given **resource** the algorithm uses to solve a problem.
- The most common resources are **time** (how many steps it takes to solve a problem) and **space** (how much memory it takes).
- We are going to be mainly interested in **how long** our programs take to **run**, as time is generally a more precious resource than space.

1 Introduction

Efficiency of Algorithms

- For example, the following graphs show the execution time, in milliseconds, against sample size, n of a given problem in **different computers**



- The actual running time of a program depends not only on the efficiency of the algorithm, but on many other variables:
  - Processor speed & type
  - Operating system
  - … etc.

1 Introduction

Running-time of Algorithms

- In order to compare algorithm speeds experimentally
  - All other variables must be kept constant, i.e.
    - independent of **specific implementations**,
    - independent of **computers** used, and,
    - independent of the **data** on which the program runs
  - Involved a lot of work (better to have some theoretical means of predicting algorithm speed)

1 Introduction

Example 1

- Task:
  - Complete the `sum_of_n()` function which calculates the sum of the first n natural numbers.
    - **Arguments**: an integer
    - **Returns**: the sum of the first n natural numbers
- Cases:

`sum_of_n(5)` → 15

`sum_of_n(100000)` → 5000050000

Algorithm 1

- sum_of_n

Set the_sum = 0

Add each value to the_sum using a for loop

Return the_sum

```
time_start = time.time()

the_sum = 0
for i in range(1,n+1):
    the_sum = the_sum + I

time_end = time.time()
time_taken = time_end - time_start
```

The timing calls embedded before and after the summation to calculate the time required for the calculation

Algorithm 2

- sum_of_n_2

Set the_sum = 0

Use the equation (n(n + 1))/2, to calculate the total

Return the_sum

```
time_start = time.clock()

the_sum = 0
the_sum = (n * (n+1) ) / 2

time_end = time.clock()
time_taken = time_end - time_start)
```


Experimental Result

- Using 4 different values for n: [10000, 100000, 1000000, 10000000]

n	sum_of_n (for loop)	sum_of_n_2 (equation)
10000	0.0033	0.00000181
100000	0.0291	0.00000131
1000000	0.3045	0.00000107
10000000	2.7145	0.00000123

Time Consuming Process!

Time increase as we increase the value of n.

NO impacted by the number of integers being added.

- We shall **count** the number of basic operations of an algorithm, and **generalise** the count.

Advantages of Learning Analysis

- Predict the running-time during the design phase
  - The running time should be **independent** of the type of input
  - The running time should be **independent** of the hardware and software environment
- Save your time and effort
  - The algorithm does not need to be **coded** and **debugged**
- Help you to write more efficient code

2 Counting Operations
Basic Operations

- We need to **estimate** the running time as a function of problem size n.
- 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

2 Counting Operations
Example 2A

- Example: Calculating a sum of first 10 elements in the list

1 assignment ->
1 assignment ->
11 comparisons ->
10 plus/assignments ->
10 plus/assignments ->
1 return ->

```
def count1(numbers):
    the_sum = 0
    index = 0
    while index < 10:
        the_sum = the_sum + numbers[index]
        index += 1
    return the_sum
```

- Total = 34 operations

2 Counting Operations
Example 2B

- Example: Calculating the sum of elements in the list.

1 assignment ->
1 assignment ->
1 assignment ->
n +1 comparisons ->
n plus/assignments ->
n plus/assignments ->
1 return

```
def count2(numbers):
    n = len(numbers)
    the_sum = 0
    index = 0
    while index < n:
        the_sum = the_sum + numbers[index]
        index += 1
    return the_sum
```

- Total = 3n + 5 operations
- We need to measure an algorithm's time requirement as a function of the problem size, e.g. in the example above the problem size is the number of elements in the list.

2 Counting Operations
Problem size

- Performance is usually measured by the **rate** at which the running time increases as the problem size gets bigger,
  - ie. we are interested in the relationship between the running time and the problem size.
  - It is very important that we identify what the problem size is.
    - For example, if we are analyzing an algorithm that processes a list, the problem size is the **size** of the list.
- In many cases, the problem size will be the **value** of a variable, where the running time of the program depends on how big that value is.

2 Counting Operations
Exercise 1

- How many operations are required to do the following tasks?
  - a) Adding an element to the end of a list   ?
  - b) Printing each element of a list containing n elements   ?

2 Counting Operations
Example 3

- Consider the following two algorithms:
  - Algorithm A:
    - Outer Loop: n operations
    - Inner Loop: $\frac{n}{5}$ operations
    - Total = $(n * \frac{n}{5})$ = $(\frac{n^2}{5})$ operations

```
for i in range(0, n):
   for j in range(0, n, 5):
      print (i,j)
```

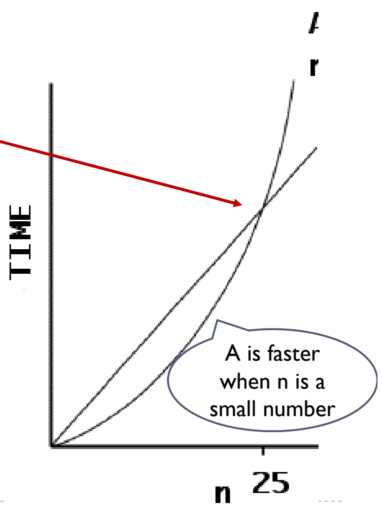
  - Algorithm B:
    - Outer Loop: n operations
    - Inner Loop: 5 operations
    - Total = n * 5 = 5*n operations

```
for i in range(0, n):
   for j in range(0, 5):
      print (i,j)
```


2 Counting Operations
Growth Rate Function – A or B?

n	5	10	15	20	24	25	26	30
A	5	20	45	80	115	125	135	180
B	25	50	75	100	120	125	130	150

- If n is 10^6,
  - Algorithm A's time requirement is
    - $(\frac{n^2}{5}) = (\frac{10^{12}}{5}) = 2 * 10^{11}$
  - Algorithm B's time requirement is
    - $5*n = 5 * 10^6$
- What does the growth rate tell us about the running time of the program?



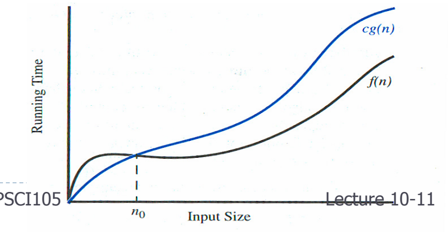
2 Counting Operations
Growth Rate Function – A or B?

- For smaller values of n, the differences between algorithm A ($n^2/5$) and algorithm B (5n) are not very big. But the differences are very evident for larger problem sizes such as for n > 1,000,000
  - $2 * 10^{11}$ Vs $5 * 10^6$
- **Bigger** problem size, produces **bigger** differences
- Algorithm efficiency is a concern for **large** problem sizes

3 Big-O
Definition

- Let $f(n)$ and $g(n)$ be functions that map nonnegative integers to real numbers. We say that $f(n)$ is O($g(n)$) if there is a real constant, c, where c > 0 and an integer constant n_0, where $n_0 \geq 1$ such that $f(n) \leq c * g(n)$ for every integer $n \geq n_0$.
  - $f(n)$ describe the actual time of the program
  - $g(n)$ is a much simpler function than $f(n)$
  - With assumptions and approximations, we can use $g(n)$ to describe the complexity i.e. O($g(n)$)

Big-O Notation is a mathematical formula that best describes an algorithm's performance



3 Big-O
Notation

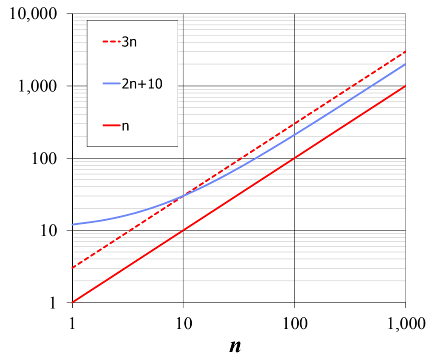
- We use Big-O notation (capital letter O) to specify the order of complexity of an algorithm
  - e.g., O(n^2) , O(n^3) , O(n).
  - If a problem of size n requires time that is directly **proportional** to n, the problem is O(n) – that is, order n.
  - If the time requirement is directly **proportional** to n^2, the problem is O(n^2), etc.

3 Big-O
Big-Oh Notation (Formal Definition)

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is O($g(n)$) if there are positive constants, c and n_0, such that

$f(n) \leq c * g(n)$ for every integer $n \geq n_0$.

- Example: 2n + 10 is O(n)
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick c = 3 and $n_0 = 10$



3 Big-O
Examples

$f(n) \leq c * g(n)$ for every integer n >= n_0.

- Suppose an algorithm requires
  - 7n-2 operations to solve a problem of size n

    $7n-2 \leq 7 * n$ for all $n_0 \geq 1$
    i.e. c = 7, n_0 = 1

    O(n)

  - n^2 - 3 * n + 10 operations to solve a problem of size n

    $n^2 - 3 * n + 10 < 3 * n^2$ for all $n_0 \geq 2$
    i.e. c = 3, n_0 = 2

    O(n^2)

  - $3n^3 + 20n^2 + 5$ operations to solve a problem of size n

    $3n^3 + 20n^2 + 5 < 4 * n^3$ for all $n_0 \geq 21$
    i.e. c = 4, n_0 = 21

    O(n^3)

4 Properties of Big-O

Properties of Big-O

- There are three properties of Big-O
  - Ignore **low order terms** in the function (smaller terms)
    - O(f(n)) + O(g(n)) = O(max of f(n) and g(n))
  - Ignore any **constants** in the high-order term of the function
    - C* O(f(n)) = O(f(n))
  - **Combine** growth-rate functions
    - O(f(n)) * O(g(n)) = O(f(n)*g(n))
    - O(f(n)) + O(g(n)) = O(f(n)+g(n))

4 Properties of Big-O

Ignore low order terms

- Consider the function: $f(n) = n^2 + 100n + \log10n + 1000$
  - For small values of n the last term, 1000, dominates.
  - When n is around 10, the terms 100n + 1000 dominate.
  - When n is around 100, the terms n^2 and 100n dominate.
  - When n gets much larger than 100, the n^2 dominates all others.
  - So it would be safe to say that this function is $O(n^2)$ for values of n > 100
- Consider another function:

  $f(n) = n^3 + n^2 + n + 5000$

  - Big-O is $O(n^3)$
- And consider another function:

  $f(n) = n + n^2 + 5000$

  - Big-O is $O(n^2)$

4 Properties of Big-O

Ignore any Constant Multiplications

- Consider the function:

  $f(n) = 254 * n^2 + n$

  - Big-O is $O(n^2)$
- Consider another function:

  $f(n) = n / 30$

  - Big-O is O(n)
- And consider another function:

  $f(n) = 3n + 1000$

  - Big-O is O(n)

4 Properties of Big-O

Combine growth-rate functions

- Consider the function:

  $f(n) = n * \log n$

  - Big-O is O(n log n)
- Consider another function:

  $f(n) = n^2 * n$

  - Big-O is $O(n^3)$

Exercise 2

- What is the Big-O performance of the following growth functions?
  - $T(n) = n + \log(n)$?
  - $T(n) = n^4 + n*\log(n) + 300n^3$?
  - $T(n) = 300n + 60 * n \; * \log(n) + 342$?

Best, average & worst-case complexity

- In some cases, it may need to consider the best, worst and/or average performance of an algorithm
- For example, if we are required to sort a list of numbers an ascending order
  - **Worst-case:**
    - if it is in reverse order
  - **Best-case:**
    - if it is already in order
  - **Average-case**
    - Determine the average amount of time that an algorithm requires to solve problems of size n
    - More difficult to perform the analysis
    - Difficult to determine the relative probabilities of encountering various problems of a given size
    - Difficult to determine the distribution of various data values

Calculating Big-O

- Rules for finding out the time complexity of a piece of code
  - Straight-line code
  - Loops
  - Nested Loops
  - Consecutive statements
  - If-then-else statements
  - Logarithmic complexity

Rules

- Rule 1: Straight-line code
  - Big-O = Constant time O(1)
  - Does not vary with the size of the input
  - Example:
    - Assigning a value to a variable
    - Performing an arithmetic operation.
    - Indexing a list element.

```
x = a + b
i = y[2]
```

- Rule 2: Loops
  - The running time of the statements inside the loop (including tests) times the number of iterations
  - Example:
    - Constant time * n
    - = c * n = O(n)

```
for i in range(n):
    print(i)
```

Executed n times

Constant time

5 Calculating Big-O
Rules (con't)

- Rule 3: Nested Loop
  - Analyze inside out. Total running time is the product of the sizes of all the loops.
  - Example:
    - constant * (inner loop: n)*(outer loop: n)
    - Total time = c * n * n = $c*n^2$ = $O(n^2)$

Outer loop: Executed n times

```
for i in range(n):
    for j in range(n):
        k = i + j
```

Inner loop: Executed n times

- Rule 4: Consecutive statements
  - Add the time complexities of each statement
  - Example:
    - Constant time + n times * constant time
    - $c_0 + c_1 n$
    - Big-O = O(f(n) + g(n))
    - = O(max (f(n) + g(n)))
    - = O(n)

Constant time

Executed n times

```
x = x + 1
for i in range(n):
    m = m + 2;
```


5 Calculating Big-O
Rules (con't)

- Rule 5: if-else statement
  - Worst-case running time: the test, plus either the `if` part or the `else` part (whichever is the larger).
  - Example:
    - c_0 + Max(c_1, (n * ($c_2 + c_3$)))
    - Total time = c_0 * n($c_2 + c_3$) = O(n)
  - Assumption:
    - The condition can be evaluated in constant time. If it is not, we need to add the time to evaluate the expression.

Test: Constant time c_0

```
if len(a) != len(b):
    return False
else:
    for index in range(len(a)):
        if a[index] != b[index]:
            return False
```

True case: Constant c_1

False case: Executed n times

Another if: constant c_2 + constant c_3

5 Calculating Big-O
Rules (con't)

- Rule 6: Logarithmic
  - An algorithm is O(log n) if it takes a constant time to cut the problem size by a fraction (usually by ½)
  - Example:
    - Finding a word in a dictionary of n pages
      - Look at the centre point in the dictionary
      - Is word to left or right of centre?
      - Repeat process with left or right part of dictionary until the word is found
  - Example:

```
size = n
while size > 1:
  // O(1) stuff
  size = size / 2
```

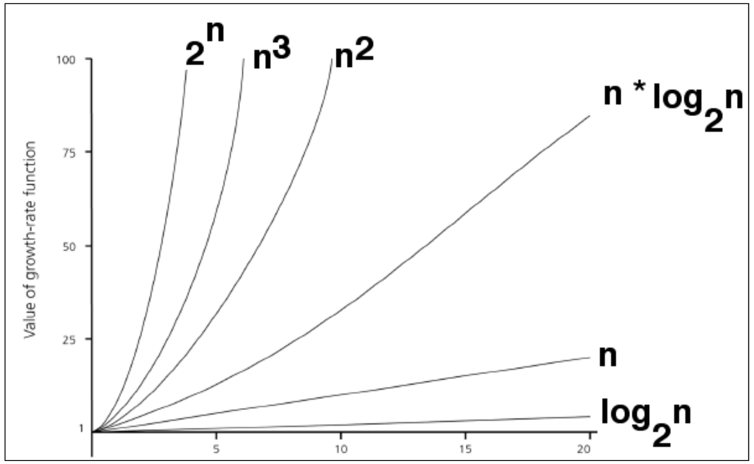
  - Size: n, n/2, n/4, n/8, n/16, . . . 2, 1
  - If $n = 2^K$, it would be approximately k steps. The loop will execute log k in the worst case ($\log_2 n = k$). Big-O = O(log n)
  - Note: we don't need to indicate the base. The logarithms to different bases differ only by a constant factor.

6 Growth Rate Examples
Hypothetical Running Time

- The running time on a hypothetical computer that computes 10^6 operations per second for varies problem sizes

Notation		n 10	10^2	10^3	10^4	10^5	10^6
O(1)	Constant	1 µsec	1 µsec	1 µsec	1 µsec	1 µsec	1 µsec
O(log(n))	Logarithmic	3 µsec	7 µsec	10 µsec	13 µsec	17 µsec	20 µsec
O(n)	Linear	10 µsec	100 µsec	1 msec	10 msec	100 msec	1 sec
O(nlog(n))	N log N	33 µsec	664 µsec	10 msec	13.3 msec	1.6 sec	20 sec
$O(n^2)$	Quadratic	100 µsec	10 msec	1 sec	1.7 min	16.7 min	11.6 days
$O(n^3)$	Cubic	1 msec	1 sec	16.7 min	11.6 days	31.7 years	31709 years
$O(2^n)$	Exponential	10 msec	3e17 years				

6 Growth Rate Examples
Comparison of Growth Rate

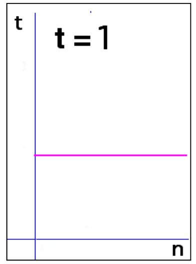


A comparison of growth-rate functions in graphical form

6 Growth Rate Examples
Constant Growth Rate - O(1)

- Time requirement is constant and, therefore, independent of the problem's size n.

```
def rate1(n):
    s = "SWEAR"
    for i in range(25):
        print("I must not ", s)
```

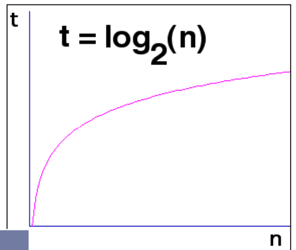


n	10^1	10^2	10^3	10^4	10^5	10^6
O(1)	1	1	1	1	1	1

6 Growth Rate Examples
Logarithmic Growth Rate - O(log n)

- Increase slowly as the problem size increases
- If you square the problem size, you only double its time requirement
- The base of the log does not affect a log growth rate, so you can omit it.

```
def rate2(n):
    s = "YELL"
    i = 1
    while i < n:
        print("I must not ", s)
        i = i * 2
```

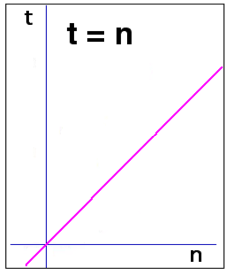


n	10^1	10^2	10^3	10^4	10^5	10^6
O($\log_2$ n)	3	6	9	13	16	19

6 Growth Rate Examples
Linear Growth Rate - O(n)

- The time increases directly with the sizes of the problem.
- If you square the problem size, you also square its time requirement

```
def rate3(n):
    s = "FIGHT"
    for i in range(n):
        print("I must not ", s)
```

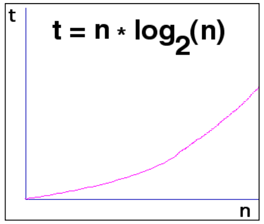


n	10^1	10^2	10^3	10^4	10^5	10^6
O(n)	10	10^2	10^3	10^4	10^5	10^6

6 Growth Rate Examples
n* log n Growth Rate - O(n log(n))

- The time requirement increases more rapidly than a linear algorithm.
- Such algorithms usually divide a problem into smaller problem that are each solved separately.

```
def rate4(n):
    s = "HIT"
    for i in range(n):
        j = n
        while j > 1:
            print("I must not ", s)
            j = j // 2
```

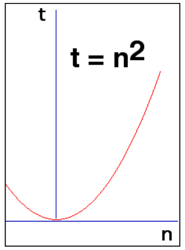


n	10^1	10^2	10^3	10^4	10^5	10^6
O(nlog(n))	30	664	9965	10^5	10^6	10^7

6 Growth Rate Examples
Quadratic Growth Rate - O(n^2)

- The time requirement increases rapidly with the size of the problem.
- Algorithms that use two nested loops are often quadratic.

```
def rate5(n):
    s = "LIE"
    for i in range(n):
        for j in range(n):
            print("I must not ", s)
```

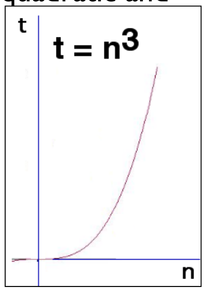


n	10^1	10^2	10^3	10^4	10^5	10^6
O(n^2)	10^2	10^4	10^6	10^8	10^{10}	10^{12}

6 Growth Rate Examples
Cubic Growth Rate - O(n^3)

- The time requirement increases more rapidly with the size of the problem than the time requirement for a quadratic algorithm
- Algorithms that use three nested loops are often quadratic and are practical only for small problems.

```
def rate6(n):
    s = "SULK"
    for i in range(n):
        for j in range(n):
            for k in range(n):
                print("I must not ", s)
```

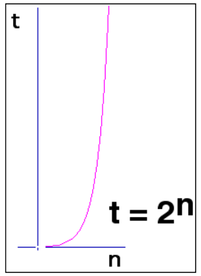


n	10^1	10^2	10^3	10^4	10^5	10^6
O(n^3)	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}

6 Growth Rate Examples
Exponential Growth Rate - O(2^n)

- As the size of a problem increases, the time requirement usually increases too rapidly to be practical.

```
def rate7(n):
    s = "POKE OUT MY TONGUE"
    for i in range(2 ** n):
        print("I must not ", s)
```



n	10^1	10^2	10^3	10^4	10^5	10^6
O(2^n)	10^3	10^{30}	10^{301}	10^{3010}	10^{30103}	10^{301030}

Exercise 3

- What is the Big-O of the following statements?

```
for i in range(n):
    for j in range(10):
        print (i,j)
```

Executed n times — Executed 10 times — Constant time

- Running time = n * 10 * 1 =10n, Big-O = ?

- What is the Big-O of the following statements?

```
for i in range(n):
    for j in range(n):
        print(i,j)
for k in range(n):
    print(k)
```

Executed n times — Executed n times — Executed n times

- The first set of nested loops is $O(n^2)$ and the second loop is O(n). This is $O(max(n^2,n))$ Big-O = ?

Exercise 3

- What is the Big-O of the following statements?

```
for i in range(n):
    for j in range(i+1, n):
        print(i,j)
```

- When i is 0, the inner loop executes (n-1) times. When i is 1, the inner loop executes n-2 times. When i is n-2, the inner loop execute once.
- The number of times the inner loop statements execute:
  - (n-1) + (n-2) ... + 2 + 1
- Running time = n*(n-1)/2,
- Big-O = ?

7 Performance of Python Lists
Performance of Python Data Structures

- We have a general idea of
  - Big-O notation and
  - the differences between the different functions,
- Now, we will look at the Big-O performance for the operations on Python **<u>lists</u>** and **<u>dictionaries.</u>**
- It is important to **<u>understand</u>** the **<u>efficiency</u>** of these Python data structures
- In later chapters we will see some possible **implementations** of both lists and dictionaries and how the **performance** depends on the implementation.

7 Performance of Python Lists
Review

- Python lists are ordered sequences of items.
- Specific values in the sequence can be referenced using subscripts.
- Python lists are:
  - **dynamic**. They can grow and shrink on demand.
  - **heterogeneous**, a single list can hold arbitrary data types.
  - **mutable** sequences of arbitrary objects.

7 Performance of Python Lists
List Operations

- Using operators:

Operator	Meaning
<seq> + <seq>	Concatenation
<seq> * <int-expr>	Repetition
<seq>[]	Indexing
len(<seq>)	Length
<seq>[:]	Slicing
for <var> in <seq>:	Iteration
<expr> in <seq>	Membership (Boolean)

```
my_list = [1,2,3,4]
print (2 in my_list)

zeroes = [0] * 20
print (zeroes)
```

True
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

7 Performance of Python Lists
List Operations

- Using Methods:

Method	Meaning
<list>.append(x)	Add element x to end of list.
<list>.sort()	Sort (order) the list. A comparison function may be passed as a parameter.
<list>.reverse()	Reverse the list.
<list>.index(x)	Returns index of first occurrence of x.
<list>.insert(i, x)	Insert x into list at index i.
<list>.count(x)	Returns the number of occurrences of x in list.
<list>.remove(x)	Deletes the first occurrence of x in list.
<list>.pop(i)	Deletes the ith element of the list and returns its value.

7 Performance of Python Lists
Examples

```
my_list = [3, 1, 4, 1, 5, 9]
my_list.append(2)
my_list.sort()
my_list.reverse()
```

[3, 1, 4, 1, 5, 9, 2]
[1, 1, 2, 3, 4, 5, 9]
[9, 5, 4, 3, 2, 1, 1]

```
print (my_list.index(4))
```

2

Index of the first occurrence of the parameter

```
my_list.insert(4, "Hello")
print (my_list)
```

[9, 5, 4, 3, 'Hello', 2, 1, 1]

```
print (my_list.count(1))
```

2

The number of occurrence of the parameter

```
my_list.remove(1)
print (my_list)

print(my_list.pop(3))
print (my_list)
```

[9, 5, 4, 3, 'Hello', 2, 1]

3
[9, 5, 4, 'Hello', 2, 1]

7 Performance of Python Lists
List Operations

- The **del** statement
  - Remove an item from a list given its index instead of its value
  - Used to remove slices from a list or clear the entire list

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

Big-O Efficiency of List Operators

index[]	O(1)
index assignment	O(1)
append	O(1)
pop()	O(1)
pop(i)	O(n)
insert(i,item)	O(n)
del operator	O(n)
iteration	O(n)
contains (in)	O(n)
get slice [x:y]	O(k)
del slice	O(n)
set slice	O($n + k$)
reverse	O(n)
concatenate	O(k)
sort	O($n \log n$)
multiply	O(nk)

O(1) - Constant

- Operations for **indexing** and **assigning** to an index position
  - Big-O = O(1)
  - It takes the same amount of time no matter how large the list becomes.
  - i.e. independent of the size of the list

Inserting elements to a List

- There are two ways to create a longer list.
  - Use the **append** method or the **concatenation** operator
- Big-O for the append method is ***O*(1)**.
- Big-O for the concatenation operator is ***O*(k)** where k is the size of the list that is being concatenated.

4 Experiments

- Four different ways to generate a list of n numbers starting with 0.
  - Example 1:
    - Using a for loop and create the list by concatenation

```
for i in range(n):
  my_list = my_list + [i]
```

  - Example 2:
    - Using a for loop and the append method

```
for i in range(n):
  my_list.append(i)
```

  - Example 3:
    - Using list comprehension

```
my_list = [i for i in range(n)]
```

  - Example 4:
    - Using the range function wrapped by a call to the list constructor.

```
my_list = list(range(n))
```

7 Performance of Python Lists
The Result

- From the results of our experiment:

Append: Big-O is O(1)
Concatenation: Big-O is O(k)

  - 1) Using for loop
    - The append operation is much faster than concatenation
  - 2) Two additional methods for creating a list
    - Using the list constructor with a call to range is much **faster** than a list comprehension
  - It is interesting to note that the list comprehension is **twice** as fast as a for loop with an append operation.

```
for i in range(n):
  my_list = my_list + [i]
```

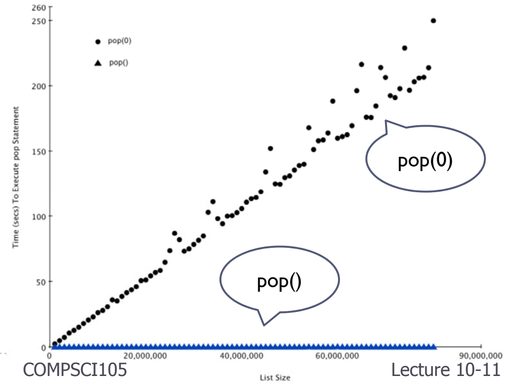
```
my_list = [i for i in range(n)]
```

```
for i in range(n):
  my_list.append(i)
```

```
my_list = list(range(n))
```

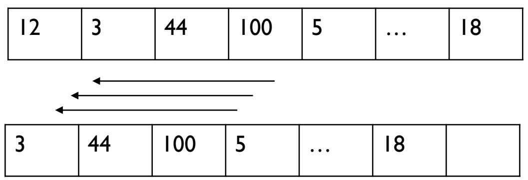

7 Performance of Python Lists
Pop() vs Pop(0)

- From the results of our experiment:
  - As the list gets longer and longer the time it takes to pop(0) also increases
  - the time for pop stays very flat.
  - pop(0): Big-O is O(n)
  - pop(): Big-O is O(1)
  - Why?



7 Performance of Python Lists
Pop() vs Pop(0)

- pop():
  - Removes element from the end of the list
- pop(0)
  - Removes from the beginning of the list.
  - Big-O is O(n) as we will need to shift all elements from space to the beginning of the list



Exercise 4

- Which of the following list operations is not $O(1)$?

1. list.pop(0)
2. list.pop()
3. list.append()
4. list[10]

?

Introduction

- Dictionaries store a mapping between a set of **keys** and a set of **values**
  - Keys can be any immutable type.
  - Values can be any type
  - A single dictionary can store values of different types
- You can define, modify, view, lookup or delete the key-value pairs in the dictionary
- Dictionaries are unordered
- Note:
  - Dictionaries differ from lists in that you can access items in a dictionary by a **key** rather than a **position**.

Examples:

```
capitals = {'Iowa':'DesMoines','Wisconsin':'Madison'}
print(capitals['Iowa'])
capitals['Utah']='SaltLakeCity'
print(capitals)
capitals['California']='Sacramento'
print(len(capitals))
for k in capitals:
    print(capitals[k]," is the capital of ", k)
```

DesMoines
{'Wisconsin': 'Madison', 'Iowa': 'DesMoines', 'Utah': 'SaltLakeCity'}
4
Sacramento is the capital of California
Madison is the capital of Wisconsin
DesMoines is the capital of Iowa
SaltLakeCity is the capital of Utah

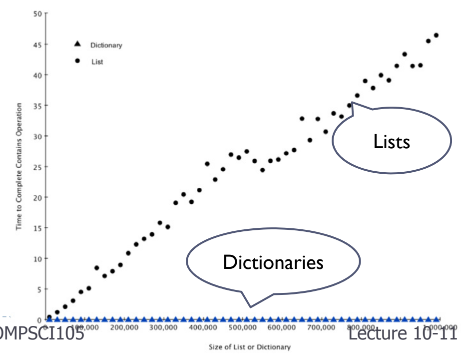
Big-O Efficiency of Operators

- Table 2.3

Operation	Big-O Efficiency
Copy	$O(n)$
get item	$O(1)$
set item	$O(1)$
delete item	$O(1)$
contains (in)	$O(1)$
iteration	$O(n)$

Contains between lists and dictionaries

- From the results
  - The time it takes for the **contains** operator on the list **grows** linearly with the size of the list.
  - The time for the contains operator on a dictionary is **constant** even as the dictionary size grows
- Lists, big-O is **O(n)**
- Dictionaries, big-O is **O(1)**



Quizzes

- Complete the Big-O performance of the following dictionary operations

1. 'x' in my_dict
2. del my_dict['x']
3. my_dict['x'] == 10
4. my_dict['x'] = my_dict['x'] + 1

?

Summary

- Complexity Analysis measure an algorithm's time requirement as a function of the problem size by using a growth-rate function.
  - It is an **implementation-independent** way of measuring an algorithm
- Complexity analysis focuses on **large** problems
- Worst-case analysis considers the **maximum** amount of work an algorithm will require on a problem of a given size
  - Average-case analysis considers the **expected** amount of work that it will require.
  - Generally we want to know the worst-case running time.
    - It provides the upper bound on time requirements
    - We may need average or the best case
    - Normally we assume worst-case analysis, unless told otherwise.