# CSCA48 Tutorial Week #11

TUT 0006

TA: Andrew Wang

Email: andrewpy.wang@mail.utoronto.ca

Website: http://individual.utoronto.ca/andrewwang/a48/

# This Week

Sorting algorithm: insertion sort

Algorithm complexity analysis

Big-Oh notation

# INSERTION SORT

The algorithm – to the board!

[6, 5, 3, 1, 8, 7, 2, 4]

The implementation – to Wing!

# ALGORITHM COMPLEXITY ANALYSIS

Assuming every operation in a computer uses same amount of time

e.g. computing the value of an expression,
querying a variable in the memory,
assigning value to a variable, etc.

We call each of those operations a "step"

Let's see how many step does every line of our insertion sort function takes!

# Algorithm Complexity Analysis

```
def insertion_sort(L):
    for i in range(1,len(L)):                # 5 steps
        current_val = L[i]                   # 3 steps
        j = i                                # 2 steps
        while(j > 0 and L[j-1] < current_val): # 8 steps
            L[j] = L[j-1]                    # 5 steps
            j = j -1                         # 3 steps
        L[j] = current_val                   # 3 steps
```

# Big-Oh Notation

Big O notation is used to describe the performance or complexity of an algorithm

Big O specifically describes the worst-case scenario

# Big-Oh Notation

**O(1)** describes an algorithm that will always execute in the same time regardless of the size of the input data set.

```
def is_empty(L):
   return len(L) == 0
```

**O(N)** describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set.

```
def contains(L, element):
    found = False
    i = 0
    while i < len(L) and not found:
        if L[i] == element:
            found = True
        i += 1
    return found
```

# Big-Oh Notation

**$O(N^2)$** represents an algorithm whose performance is directly proportional to the square of the size of the input data set.

```
def insertion_sort(L):
    for i in range(1,len(L)):
        current_val = L[i]
        j = i
        while(j > 0 and L[j-1] < current_val):
            L[j] = L[j-1]
            j = j -1
        L[j] = current_val
```

# Big-Oh Notation

**$O(2^N)$** denotes an algorithm whose growth doubles with each additon to the input data set. The growth curve of an $O(2^N)$ function is exponential.

```
def fib(n):
    if(n < 2):
        result = 1
    else:
        result = fib(n - 1) + fib(n - 2)
    return result
```

# Big-Oh Notation

**O(log N)** represents binary algorithms.

```
def binary_search(L, s):
    if(len(L) == 0):
        result = False
    else:
        mid_index = len(L) // 2
        mid_element = L[mid_index]
        if(mid_element == s):
            result = True
        elif(mid_element < s):
            result = binary_search(L[mid_index + 1:], s)
        else:
            result = binary_search(L[: mid_index], s)
    return result
```

# Big-Oh Notation

Just for fun: **O(∞)** denotes an algorithm which will never finish in the worst case scenario.

```
def bogo_sort(x):
    while not inorder(x):
        shuffle(x)
    return x
```

# Big-Oh Notation

$O(1)$

$O(N)$

$O(N^2)$

$O(2^N)$

$O(\log N)$