Data Structures and Algorithms

algorithm is simply a procedure or formula for solving problems

some problems have names as well as some procedures being common enough that the algorithm

associated with it also has a name

**Big-O notation**

Big-O notation describes how quickly runtime will grow relative to the input as the input get arbitrarily large

Different types of Big-O functions

1 constant

log(n) logartithmic

n linear

nlog(n) Log linear

n^2 Quadratic

n^3 cubic

2^n Exponential

**Constant:**

When using constant type function of Big –O , for any number of input the value returned by the function is a single value eg below

**def** func\_constant(values):

*'''*

*Prints first item in a list of values.*

*'''*

print values[0]

func\_constant([1,2,3])

Ouput 1 , because we are accessing the first value in the list , whatever may be the length of the since we are accessing the first element in the list the values will be always constant

Note how this function is constant because regardless of the list size, the function will only ever take a constant step size, in this case 1, printing the first value from a list. so we can see here that an input list of 100 values will print just 1 item, a list of 10,000 values will print just 1 item, and a list of n values will print just 1 item!

**Linear**

When using linear type of function , for n number of inputs the function runs n times , for all the values of n.

**def** func\_lin(lst):

*'''*

*Takes in list and prints out all values*

*'''*

**for** val **in** lst:

print val

func\_lin([1,2,3])

Output: 1

2

3

This function runs in O(n) (linear time). This means that the number of operations taking place scales linearly with n, so we can see here that an input list of 100 values will print 100 times, a list of 10,000 values will print 10,000 times, and a list of n values will print n times.

**Quadratic:**

While using quadratic the values in the list are in order of n \* n i.e , we need to perform two looping in order to get the get the values twice .

**def** func\_quad(lst):

*'''*

*Prints pairs for every item in list.*

*'''*

**for** item\_1 **in** lst:

**for** item\_2 **in** lst:

print item\_1,item\_2

lst = [0, 1, 2, 3]

func\_quad(lst)

Output: 0 1 in the next line 0 1 in the next line 0 2 in the next line 0 3 and so on upto 3 3.

Note how we now have two loops, one nested inside another. This means that for a list of n items, we will have to perform n operations for every item in the list! This means in total, we will perform n times n assignments, or n^2. So a list of 10 items will have 10^2, or 100 operations. You can see how dangerous this can get for very large inputs! This is why Big-O is so important to be aware of!

**Calculating Scale of Big-O**

In this section we will discuss how insignificant terms drop out of Big-O notation.

When it comes to Big O notation we only care about the most significant terms, remember as the input grows larger only the fastest growing terms will matter. If you've taken a calculus class before, this will reminf you of taking limits towards infinity. Let's see an example of how to drop constants:

**def** print\_once(lst):

*'''*

*Prints all items once*

*'''*

**for** val **in** lst:

print val

print\_once(lst)

0

1

2

3

The print\_once() function is O(n) since it will scale linearly with the input. What about the next example?

**def** print\_3(lst):

*'''*

*Prints all items three times*

*'''*

**for** val **in** lst:

print val

**for** val **in** lst:

print val

**for** val **in** lst:

print val

print\_3(lst)

0

1

2

3

0

1

2

3

0

1

2

3

We can see that the first function will print O(n) items and the second will print O(3n) items. However for n going to inifinity the constant can be dropped, since it will not have a large effect, so both functions are O(n).

Let's see a more complex example of this:

**def** comp(lst):

*'''*

*This function prints the first item O(1)*

*Then is prints the first 1/2 of the list O(n/2)*

*Then prints a string 10 times O(10)*

*'''*

print lst[0]

midpoint = len(lst)/2

**for** val **in** lst[:midpoint]:

print val

**for** x **in** range(10):

print 'number'

lst = [1,2,3,4,5,6,7,8,9,10]

comp(lst)

1

1

2

3

4

5

number

number

number

number

number

number

number

number

number

number

So let's break down the operations here. We can combine each operation to get the total Big-O of the function:

O(1+n/2+10)O(1+n/2+10)

We can see that as n grows larger the 1 and 10 terms become insignificant and the 1/2 term multiplied against n will also not have much of an effect as n goes towards infinity. This means the function is simply O(n)!

**Worst Case vs Best Case**

Many times we are only concerned with the worst possible case of an algorithm, but in an interview setting its important to keep in mind that worst case and best case scenarios may be completely different Big-O times. For example, consider the following function:

**def** matcher(lst,match):

*'''*

*Given a list lst, return a boolean indicating if match item is in the list*

*'''*

**for** item **in** lst:

**if** item == match:

**return** **True**

**return** **False**

lst= [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

matcher(lst,1)

True

matcher(lst,11)

False

Note that in the first scenario, the best case was actually O(1), since the match was found at the first element. In the case where there is no match, every element must be checked, this results in a worst case time of O(n). Later on we will also discuss average case time.

Finally let's introduce the concept of space complexity.

**Space Complexity**

Many times we are also concerned with how much memory/space an algorithm uses. The notation of space complexity is the same, but instead of checking the time of operations, we check the size of the allocation of memory.

Let's see a few examples:

**def** printer(n=10):

*'''*

*Prints "hello world!" n times*

*'''*

**for** x **in** range(n):

print 'Hello World!'

printer()

Hello World!

Hello World!

Hello World!

Hello World!

Hello World!

Hello World!

Hello World!

Hello World!

Hello World!

Hello World!

Note how we only assign the 'hello world!' variable once, not every time we print. So the algorithm has O(1) **space** complexity and an O(n) **time** complexity.

Let's see an example of O(n) **space** complexity:

**def** create\_list(n):

new\_list = []

**for** num **in** range(n):

new\_list.append('new')

**return** new\_list

print create\_list(5)

['new', 'new', 'new', 'new', 'new']

Note how the size of the new\_list object scales with the input **n**, this shows that it is an O(n) algorithm with regards to **space** complexity.

Thats it for this lecture, before continuing on, make sure to complete the homework assignment below:

Dynamic Arrays:

The dynamic array is one dynamically doubles the size when the size of the array increases. We can construct the dynamic array, but it uses order O(n) methodology.

Eg of creating a dynamic array

import ctypes

import sys

class DynamicArray(object):

def \_\_init\_\_(self):

self.n =0

self.capacity = 1

self.A = self.make\_array(self.capacity)

def \_\_len\_\_(self):

return self.n

def \_\_getitem\_\_(self, k):

if not 0 <= k < self.n:

return IndexError('k is out of bounds!')

return self.A[k]

def append(self, ele):

if self.n == self.capacity:

self.\_resize(2 \* self.capacity)

""" The new capacity variable which is passed in the resize and makenew array method is the the value

passed in the resize function in append."""

self.A[self.n] = ele

self.n +=1

def \_resize(self, new\_cap):

B = self.make\_array(new\_cap)

for k in range(self.n):

B[k] =self.A[k]

self.A = B

self.capacity = new\_cap

def make\_array(self, new\_cap):

return (new\_cap \* ctypes.py\_object)()

arr = DynamicArray()

arr.append(1)

print(len(arr)) , Ouptut: we can get add elements to the array using the append function we can get the length and the index value of the array.

Amortization

By using this algorithm pattern we can show that performing a sequence of such append operations on a dynamic array is actually quite efficient by using with the Amortization we can optimize the dynamic array to O(1)

Stacks Deques and Queue:

# Implementation of Stack

## Stack Attributes and Methods

Before we implement our own Stack class, let's review the properties and methods of a Stack.

The stack abstract data type is defined by the following structure and operations. A stack is structured, as described above, as an ordered collection of items where items are added to and removed from the end called the “top.” Stacks are ordered LIFO. The stack operations are given below.

* Stack() creates a new stack that is empty. It needs no parameters and returns an empty stack.
* push(item) adds a new item to the top of the stack. It needs the item and returns nothing.
* pop() removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.
* peek() returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.
* isEmpty() tests to see whether the stack is empty. It needs no parameters and returns a boolean value.
* size() returns the number of items on the stack. It needs no parameters and returns an integer.

# Queues Overview

In this lecture we will get an overview of what a Queue is, in the next lecture we will implement our own Queue class.

A **queue** is an ordered collection of items where the addition of new items happens at one end, called the “rear,” and the removal of existing items occurs at the other end, commonly called the “front.” As an element enters the queue it starts at the rear and makes its way toward the front, waiting until that time when it is the next element to be removed.

The most recently added item in the queue must wait at the end of the collection. The item that has been in the collection the longest is at the front. This ordering principle is sometimes called **FIFO, first-in first-out**. It is also known as “first-come first-served.”

The simplest example of a queue is the typical line that we all participate in from time to time. We wait in a line for a movie, we wait in the check-out line at a grocery store, and we wait in the cafeteria line. The first person in that line is also the first person to get serviced/helped.

Note how we have two terms here, **Enqueue** and **Dequeue**. The enqueue term describes when we add a new item to the rear of the queue. The dequeue term describes removing the front item from the queue.

# Deques Overview

A deque, also known as a double-ended queue, is an ordered collection of items similar to the queue. It has two ends, a front and a rear, and the items remain positioned in the collection. What makes a deque different is the unrestrictive nature of adding and removing items. New items can be added at either the front or the rear. Likewise, existing items can be removed from either end. In a sense, this hybrid linear structure provides all the capabilities of stacks and queues in a single data structure.

It is important to note that even though the deque can assume many of the characteristics of stacks and queues, it does not require the LIFO and FIFO orderings that are enforced by those data structures. It is up to you to make consistent use of the addition and removal operations.

Singly Linked Lists

A singly linked list, in its simplest form, is a collection of nodes that collectively form a linear sequence.

Each node stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list.

Doubly Linked Lists

In a doubly linked list, we define a linked list in which each node keeps an explicit reference to the node before it and a reference to the node after it.

These lists allow a greater variety of O(1)-time update operations, including insertions and deletions.

We continue to use the term “next” for the reference to the node that follows another.

We have a new term “prev” for the reference to the node that precedes it.

## Recursion

There are two main instances of recursion. The first is when recursion is used as a technique in which a function makes one or more calls to itself. The second is when a data structure uses smaller instances of the exact same type of data structure when it represents itself. Both of these instances are use cases of recursion.

Recursion actually occurs in the real world, such as fractal patterns seen in plants!