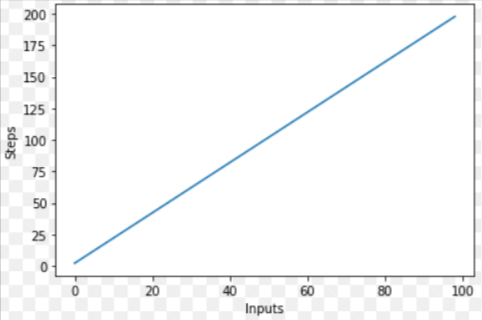
**DATA STRUCTURES AND ALGORITHMS IN PYTHON**

* Best case is always denoted as **OMEGA.**
* Average case is always called **THETA.**
* Worst case is always called as **Big(O).**

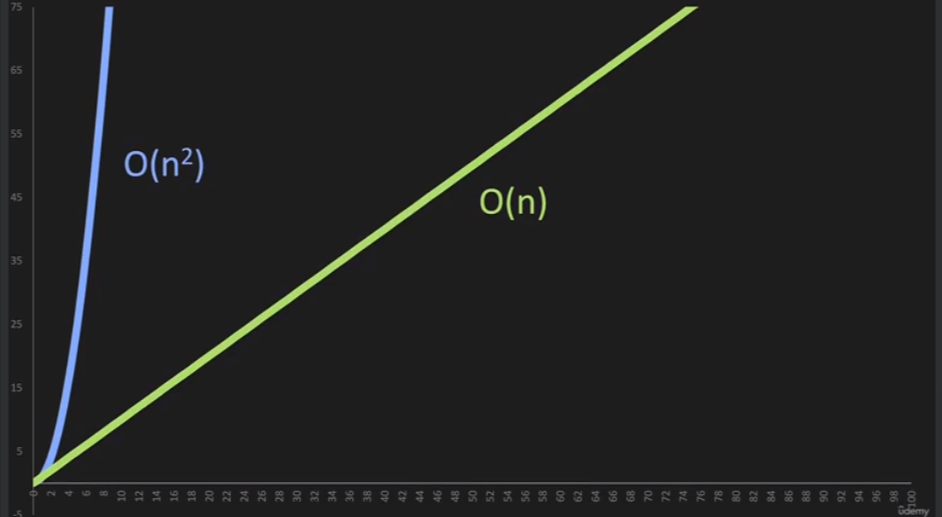
**Big(O)**

* We can express algorithmic complexity using the big-O notation.
* Big(O) means always the worst case scenario. There is no best case or worst case for the Big(O).

**O(n):**

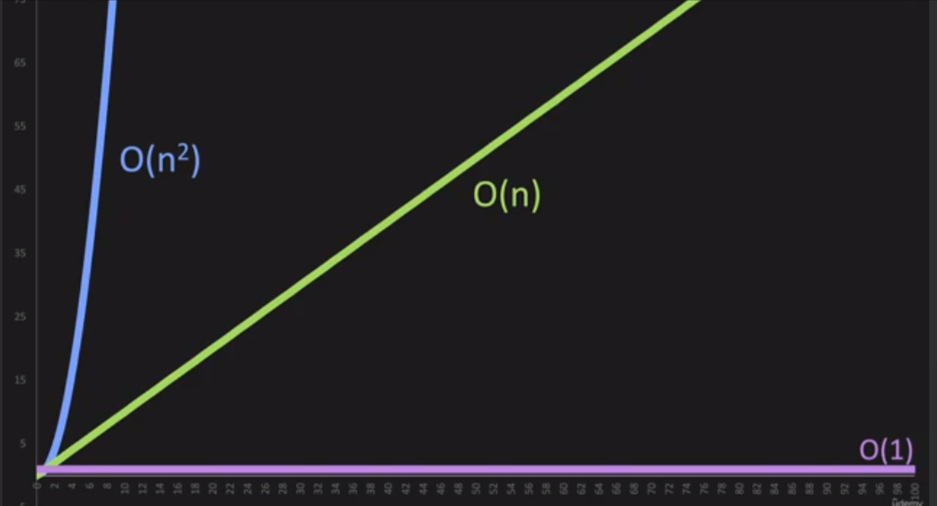
* Algorithm takes number of steps directly proportional to the size of the input.
* If input size is 5 then algorithm take the 5 steps to finish.
* def print\_items(n):  
   for i in range(n):  
   print(i,end=" ")  
  n = 10  
  print\_items(n)
* output: **0 1 2 3 4 5 6 7 8 9**
* Graph is always be the straight lite.
* ****

**O(n^2):**

* We are using the two loops here and those loops are nested because of that we got the **O(n^2).**
* def print\_items(n):  
   for i in range(n):  
   for j in range(n):  
   print(i,j)  
    
  print\_items(2)
* ****

**O(1):**

* It related to the number of operation.
* As the n increases the number of operation remain constant
* def add\_items(n):  
   return n+n+n  
  print(add\_items(5))
* abo we can see that n get increases but the operations remain same.

****

**Pointers in python:**

num1 = 11  
num2 = num1  
print(f"Value of num1 is {num1}")  
print(f"Value of num2 is {num2}")

print(f"Memory location of num1: {id(num1)} ")  
print(f"Memory location of num2: {id(num2)} ")

# output

Value of num1 is 11

Value of num2 is 11

Memory location of num1: 140705953997928

Memory location of num2: 140705953997928

* If we assign the num2 = num1 for integers it point to the same memory location
* If we change the value of num2 them it point out the different memory location.
* num1 = 11  
  num2 = num1  
  print(f"Value of num1 is {num1}")  
  print(f"Value of num2 is {num2}")  
    
  print(f"Memory location of num1: {id(num1)} ")  
  print(f"Memory location of num2: {id(num2)} ")  
    
  print(f"--------------After changing the value-------------------")  
  num2 = 22  
  print(f"Value of num1 is {num1}")  
  print(f"Value of num2 is {num2}")  
    
  print(f"Memory location of num1: {id(num1)} ")  
  print(f"Memory location of num2: {id(num2)} ")
* #output
* Value of num1 is 11
* Value of num2 is 11
* Memory location of num1: 140705953997928
* Memory location of num2: 140705953997928
* --------------After changing the value-------------------
* Value of num1 is 11
* Value of num2 is 22
* Memory location of num1: 140705953997928
* Memory location of num2: 140705953998280
* But with dictionary it completely different.

dict1 = {  
 "value":11  
}  
dict2 = dict1  
print(f"Value of dict1: {dict1}")  
print(f"Value of dict2: {dict2}")  
print(f"Memory location of dict1: {id(dict1)}")  
print(f"Memory location of dict2: {id(dict2)}")  
print(f"--------after changing the value-------------")  
dict2["value"] = 55  
print(f"Value of dict1: {dict1}")  
print(f"Value of dict2: {dict2}")  
print(f"Memory location of dict1: {id(dict1)}")  
print(f"Memory location of dict2: {id(dict2)}")

#output

Value of dict1: {'value': 11}

Value of dict2: {'value': 11}

Memory location of dict1: 2462686071872

Memory location of dict2: 2462686071872

--------after changing the value-------------

Value of dict1: {'value': 55}

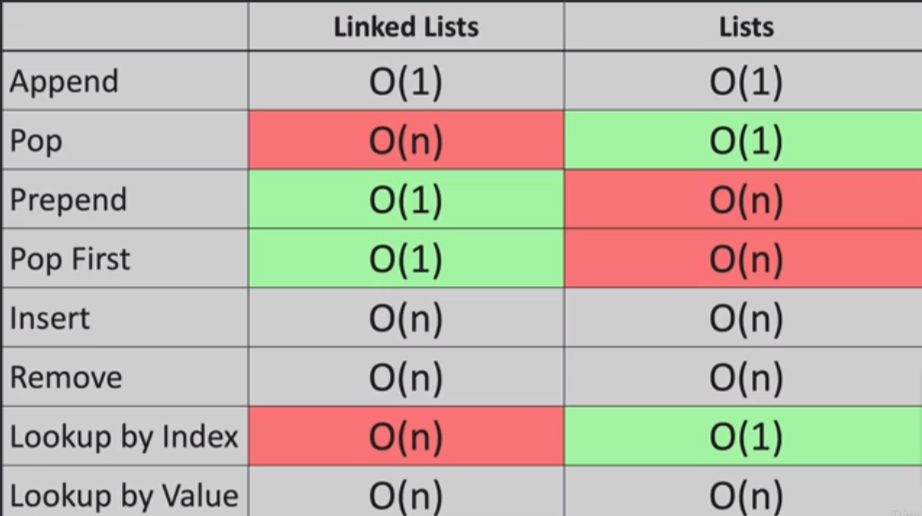
Value of dict2: {'value': 55}

Memory location of dict1: 2462686071872

Memory location of dict2: 2462686071872

* It point put the same memory location but value is updated.
* It does not create the new dictionary.

**Linked List**

* Linked list does not have the index value instead of they have pointers value.
* All the elements are store`s the pointer of the next value.
* First element in the linked list is called as **head.**
* Last element in the list is called as **Tail.**
* List in python has the index value which can we used to identify the elements.
* 
* Node is just dictionary link structure.
* {  
   "value":11,  
   "next":-->  
  }
* 🡪 means pointer.
* Structure of the linked list
* head = {  
   "value":11,  
   "next":{  
   "value":12,  
   "next":{  
   "value":22,  
   "next":{  
   "value":55,  
   "next":{  
   "value":44, #----------------> Tail  
   "next":None  
   }  
   }  
   }  
   }  
  }
* accessing the values
* print(head["next"]["next"]["next"])
* #output
* 22

Constructor for Linked List :-

* Methods like append ,prepend, and insert have common operation is creating the node.
* Node is just a dictionary.
* {  
   "value": 4,  
   "next": None  
  }
* First we need to create the node.
* class Node:  
   def \_\_init\_\_(self,value):  
   self.value = value  
   self.next = None
* Above Node() Class create the node.
* class Node:  
   def \_\_init\_\_(self, value):  
   self.value = value  
   self.next = None  
    
  class LinkedList:  
   def \_\_init\_\_(self, value):  
   new\_node = Node(value)  
   self.head = new\_node  
   self.tail = new\_node  
   self.length = 1  
    
  my\_linked\_list = LinkedList(4)  
  print(my\_linked\_list.head.value)
* above code creates the new node and head and tail is pointed to same node.
* Self.length is count the nodes in linked list.

2. Stack Data Structure

* Stack is a linear data structure.
* It stores the data in the LIFO(last in first out ) Or FILO (First in last out).
* Example: Undo feature in editor, back button in browser, etc.
* In stack new element is inserted only a the end and also remove from only at the end.
* We can perform only operation on stack
  + Push (Add the element at the end of the list)
  + Pop (Remove the element at the end of the list)
* Following are the methods of the stack
  + **empty () -** It returns true, it the stack is empty. The time complexity is O(1).
  + **size() -** It returns the length of the stack. The time complexity is O(1).
  + **top() -** This method returns an address of the last element of the stack. The time complexity is O(1).
  + **push(g) -** This method adds the element 'g' at the end of the stack - The time complexity is O(1).
  + **pop() -** This method removes the topmost element of the stack. The time complexity is O(1).
* In python we can implement the stack using the list and Collections.deque Or queue.LifoQueue
  + **Implementing using the list**
    - List is dynamic collection of the data.
    - Python list uses the **append()**  method to insert element in the list where stack uses the **push().**
    - List also provides the **pop()**  method to remove element, but list becomes slow when it grows.
    - If we have the 10 elements in the list and we want to insert the 11th element then it will copy the all the list data and allocate the new memory to the entire list.
    - This causes the problem of the memory allocations that means we need more memory if we use the list as the stack
* stack = []  
  stack.append("a")  
  stack.append("b")  
  stack.append("c")  
  print("Initial Stack--------")  
  print(stack)  
    
  print("\nElements are pop from the stack----")  
  print(stack.pop())  
  print(stack.pop())  
  print(stack.pop())  
    
  print("\nStack after ekements are pop----")  
  print(stack)
  + - Output  
      Initial Stack--------
    - ['a', 'b', 'c']
    - Elements are pop from the stack----
    - c
    - b
    - a
    - Stack after elements are pop----
    - [ ]
  + Implementing Using the **collections.deque**
    - Collection module provides the deque class, which is used to creating the Python Stack.
    - The deque pronounce as **“deck”**.
    - The deque can be preferred over the list because it performs append and pop operation faster than the list. The time complexity is O(1), where the list takes O(n).
* from collections import deque
* my\_stack = deque()
* # append() function is used to push
* # element in the my\_stack
* my\_stack.append('a')
* my\_stack.append('b')
* my\_stack.append('c')
* print('Initial my\_stack:')
* print(my\_stack)
* # pop() function is used to pop
* # element from my\_stack in
* # LIFO order
* print('\nElements poped from my\_stack:')
* print(my\_stack.pop())
* print(my\_stack.pop())
* print(my\_stack.pop())
* print('\nmy\_stack after elements are poped:')
* print(my\_stack)
* **Output:**
* Initial my\_stack:
* deque(['a', 'b', 'c'])
* Elements poped from my\_stack:
* c
* b
* a
* my\_stack after elements are poped:
* deque([])