Big O notation

Es el nivel de complejidad o la cantidad de espacio, tiempo que le tomará a una función completarse según lo que se realiza dentro de ella.

Ejemplo O(n):

```
def item_in_list(to_check,the_list):
      for item in the_list:
            if to_check == item:
                  return True
            return False
```

Por lo tanto el crecimiento es lineal o "n" = al número de ítems dentro del arreglo. Pasamos por todos los elementos de la lista y si encontramos el primer argumento retorno **True** si llego al final y no lo encuentro retorno **False**.

Ejemplo O(1):

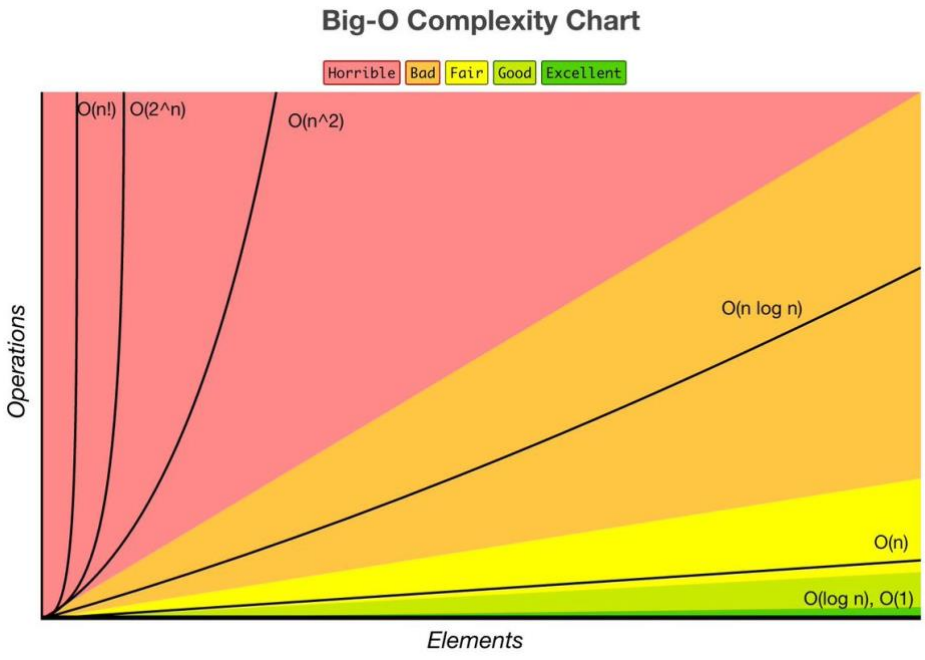
```
def is_none(item):
      return item is none
```

No importa la longitud del input siempre toma la misma cantidad de tiempo calcularlo

Ejemplo O(n^2):

```
def all_combinations(the_list):
  results = []
  for item in the_list:
    for inner_item in the_list:
      results.append(item, inner_item)
  return results
```

Este hace match con todos y cada uno de los elementos de la lista si le damos [1,2,3]



Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

Algoritmos de Ordenamiento

Selection Sort:

Pasa n veces por la lista ajustando así el valor más largo cada pasada:

```
def selectionSort(array):
  for locone in range(len(array)-1,0):
    max = 0
    for loctwo in range(1,locone+1):
      if array[loctwo] > array[max]:
        max = loctwo
    temp = array[locone]
    array[locone] = array[max]
    array[max] = temp

array = [54,26,93,17,77,31,44,55,20]
selectionSort(array)
print(array)
```

Insertion Sort:

Mantiene una sublista ordenada de las posiciones menores de la lista, cada nuevo valor es "reinsertado" en la sublista de tal manera que la sublista siempre tiene un valor más. Es mejor porque sólo se hace una asignación.

```
def insertionSort(array):
  for index in range(1,len(array)):
    curval = array[index]
    pos = index
    while pos > 0 and array[pos-1] > curval:
      array[pos] = array[pos-1]
      pos = pos-1
    array[pos]=curval

array = [54,26,93,17,77,31,44,55,20]
insertionSort(array)
print(array)
```

Quick Sort:

Utiliza el paradigma de divide y vencerás para ganar las mismas ventajas del merge sin usar almacenaje adicional. Sin embargo puede suceder que el pivote en la primer instancia no sea acomodado a la mitad y eso pueda hacer que el desempeño sea malo.

Primero seleccionamos un valor que usamos como pivote, se puede seleccionar cualquiera pero para este ejemplo voy a seleccionar el primer valor. El pivote lo tenemos que acomodar en su lugar en la lista, la idea es que al final queden del lado derecho todos los valores mayores al pivote (en desorden) y del lado izq todos los menores al pivote (en desorden) y en estas nuevas subsecciones de la lista aplicamos recursivamente este paradigma hasta que todos los valores queden acomodados.

Para acomodar el pivote me ayudo de otros 2 indicadores que llamaré izq y der. Empezamos incrementando el indicador izq hasta encontrar un valor más alto que el pivote, entonces movemos der hasta que encontramos un valor que es menor al pivote. En este punto ya encontramos 2 valores que están desacomodados y los cambiamos de lugar y repetimos el proceso, hasta que der, la posición es menor que izq ahí nos detenemos.

La posición donde se encuentra derecha es ahora el punto de división cambiamos el contenido del pivote por der, y ahora todos los valores a la derecha del punto de división son mayores y a la izq son menores. Ahora aplico lo mismo a estas 2 sublistas recursivamente.

```
def quickSort(array):
   quickSortHelper(array,0,len(array)-1)

def quickSortHelper(array,incio,fin):
   if inicio<fin:

       puntoDiv = particion(array,inicio,fin)

       quickSortHelper(array, inicio, puntoDiv-1)
       quickSortHelper(array, puntoDiv+1, fin)


def partition(array,inicio,fin):
   pivot = array[inicio]

   izq = inicio+1
   der = fin
   done = False
   while not done:

       while izq <= der and array[izq] <= pivot:
           izq = izq + 1

       while array[der] >= pivot and der >= izq:
           der = der - 1

       if der < izq:
           done = True
       else:
           temp = array[izq]
           array[izq] = array[der]
           array[der] = temp

   temp = array[inicio]
   array[inicio] = array[der]
   array[der] = temp

   return der

array = [54,26,93,17,77,31,44,55,20]
quickSort(array)
print(array)
```

Merge Sort:
Utiliza el paradigma de divide y vencerás, es un algoritmo recursivo que divide las listas a la mitad, si la lista está vacía o solo tiene un valor se considera ordenada. Si tiene más de un valor la dividimos, ya que las dos mitades han sido ordenadas usamos una operación de unirlas o merge. Las combinamos en una sola lista ordenada.

Se inicia preguntando longitud inicial, si la longitud de la lista es menor o igual a 0 entonces tenemos una lista ordenada y no se necesita mayor procesamiento. Calculamos la mitad y creamos 2 listas en las cuales llamamos de nuevo la función recursiva para dividir en otras listas y así hasta que queden divididas. Luego las volvemos a unir ya acomodadas y ordenadas.

```
def mergeSort(array):
    print("Dividiendo ",array)
    if len(array)>1:
        mitad = len(array)//2
        izq = array[:mitad]
        der = array[mitad:]

        mergeSort(izq)
        mergeSort(der)

        i=0
        j=0
        k=0
        while i < len(izq) and j < len(der):
            if izq[i] < der[j]:
                array[k]=izq[i]
                i=i+1
            else:
                array[k]=der[j]
                j=j+1
            k=k+1

        while i < len(izq):
            array[k]=izq[i]
            i=i+1
            k=k+1

        while j < len(der):
            array[k]=der[j]
            j=j+1
            k=k+1
    print("Merging ",array)

array = [54,26,93,17,77,31,44,55,20]
mergeSort(array)
print(array)
```

Binary Search:
En una lista ordenada inicia buscando en la mitad. Si ese era terminamos. Sí es mayor o menor a la mitad sabemos en que mitad dejar de buscar y repetimos el proceso con la mitad en la que debería de estar.

```
def binarySearch(alist, item):
  first = 0
  last = len(alist)-1
  found = False
  while first<=last and not found:
    midpoint = (first + last)//2
    if alist[midpoint] == item:
      found = True
    else:
      if item < alist[midpoint]:
        last = midpoint-1
      else:
        first = midpoint+1
  return found

testlist = [0, 1, 2, 8, 13, 17, 19, 32, 42,]
print(binarySearch(testlist, 3))
print(binarySearch(testlist, 13))
```

Hashtable:
Colección de ítems que se guardan en “slots”, cada elemento tiene “key” y “value” key es generalmente un integer generado de una operación resultado del value.

X ejemplo:

En una de tamaño 11 usando la operación value % 11

Item	HashValue
54	10
26	4
93	5
17	6

Stack:
Last in first out LIFO

- Push para agregar uno
- Pop para eliminar el ultimo que agregamos

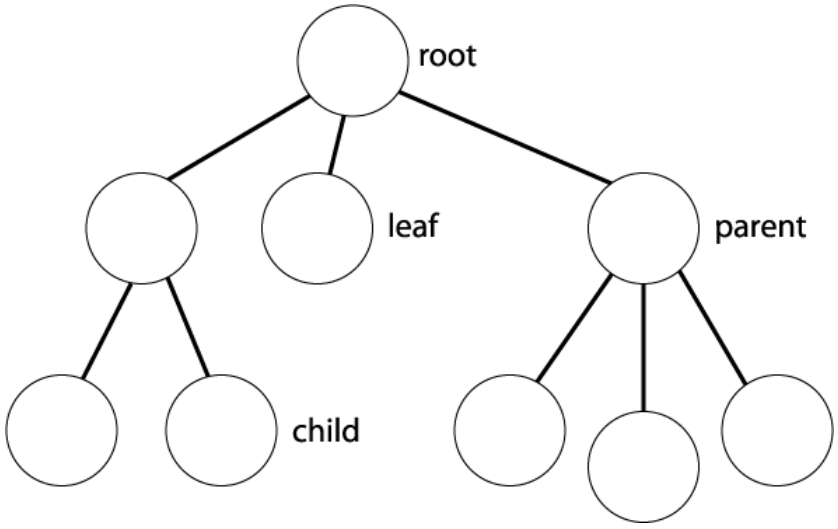
Queue:
First in First out FIFO

- Enqueue para agregar
- Dequeue para remover

Tree:

Consiste en una serie de nodos y aristas que conectan pares de nodos y tienen las siguientes propiedades.

- Uno de los nodos es designado como nodo raíz
- Cada nodo n, menos el nodo raíz, es conectado por una arista desde otro nodo p, donde p es padre de n.
- Un camino único cruza desde la raíz a cada nodo.
- Si cada nodo tiene un máximo de 2 hijos entonces puede llamarse **Binary Tree.**




```
Mytree = ["a",
          ["b",
          ["d",[],[]],
          ["e",[],[]]],
          ["c",
          ["f",[],[]],
          []]
          ]
```