Geraldo Braho

Comp 5327

Discussion 2

Please discuss asymptotic order of growth Big-Oh, Big-Omega, and Big-Theta using an algorithm of your choice

Definition

Big Oh Notation, O

$O(f(n)) = \{ g(n) : \text{there exists } c > 0 \text{ and } n0 \text{ such that } f(n) \leq c.g(n) \text{ for all } n > n0. \}$

Omega Notation, Ω

$\Omega(f(n)) \geq \{ g(n) : \text{there exists } c > 0 \text{ and } n0 \text{ such that } g(n) \leq c.f(n) \text{ for all } n > n0. \}$

Theta Notation, θ

$\theta(f(n)) = \{ g(n) \text{ if and only if } g(n) = O(f(n)) \text{ and } g(n) = \Omega(f(n)) \text{ for all } n > n0. \}$

Python is a wide language and it has many rich libraries that we can inport and work with them. On of them is BIG_O library.big_O is a Python module to estimate the time complexity of Python code from its execution time.

In [3]:
```
import big_o
import numpy as np
big_o.big_o(np.zeros, big_o.datagen.n_, max_n=100000, n_repeats=100)
```

Out[3]:
```
(<big_o.complexities.Linear at 0x10b2d2470>,
 {<big_o.complexities.Constant at 0x10b2d2160>: 6.7063845854587176e-
06,
  <big_o.complexities.Linear at 0x10b2d2470>: 1.7706947047647937e-07
,
  <big_o.complexities.Quadratic at 0x10b2d24a8>: 6.2716584102630193e
-07,
  <big_o.complexities.Cubic at 0x10b2d24e0>: 1.2085146863108874e-06,
  <big_o.complexities.Polynomial at 0x10b2d2518>: 0.2176646212604116
4,
  <big_o.complexities.Logarithmic at 0x10b2d2550>: 2.413837229040560
9e-06,
  <big_o.complexities.Linearithmic at 0x10b2d2588>: 1.96661493652106
87e-07,
  <big_o.complexities.Exponential at 0x10b2d25c0>: 2.189825350454839
3})
```

Big Oh Notation, O

Big O is used to measure the performance or complexity of an algorithm. In more mathematical term, it is the upper bound of the growth rate of a function, or that if a function g(x) grows no faster than a function f(x), then g is said to be a member of O(f).In general, it is used to express the upper bound of an algorithm and which gives the measure for the worst time complexity or the longest time an algorithm possibly take to complete.

Big Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

Big Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time.

Notation is used to determine the complexity of various algorithm's

Big O notation is mostly used, and it is used mainly all the times to find the upper bound of an algorithm whereas the Big θ notation is sometimes used which is used to detect the average case and the Ω notation is the least used notation among the three. You will be seeing the examples of the notation used in the algorithm to determine the complexity for that particular algorithm. For example for a quick sort: Quick sort is a Divide and Conquer algorithm where it is used for sorting. It serves as a systematic method of placing elements in the order, i.e. For example arranging elements or number in the array in ascending or descending order. This algorithm picks the pivot or the index that is chosen from the given array. Also, the pivot can be selected in the different ways. The example that is implemented below is the pivot element selected with the last element. The main crux of the Quick sort is the partition. From an array, a partition element is chosen, and then the partition element(i.e.pit for example) is kept in correct position and then the element greater to the partition is kept at the right of the pit, and the smaller element is kept at the left of the pit.

In [4]:
```
#The last element will be taken as a pivot by the use of the function
#The smaller element is placed left to the pivot
#The greater element is placed to the right of the pivot
def partition(array,low,high):
    i = ( low-1 )         # index of smaller element is chosen
    pivot = array[high]     # pivot is chosen

    for j in range(low , high):

        #Is the element less or equal to the pivot
        if   array[j] <= pivot:

            # increment index of smaller element
            i = i+1
            array[i],array[j] = array[j],array[i]

    array[i+1],array[high] = array[high],array[i+1]
    return ( i+1 )

# The main crux of the problem that implements Quick sort is
#array[] is to be sorted
#high is the ending index
#low is the starting index

# Function to do Quick sort
def quickSort(array,low,high):
    if low < high:

       #pit is the partitioning index
        pit = partition(array,low,high)

      #Element sorted before and after partition
        quickSort(array, low, pit-1)
        quickSort(array, pit+1, high)

array=[2,4,6,8,10,12]
n = len(array)
quickSort(array,0,n-1)
print ("The Sorted array is:")
for i in range(n):
    print ("%d" %array[i]),
```

```
The Sorted array is:
2
4
6
8
10
12
```

Now it is time to analyze the time complexity. At first,

- The best case is: Ω(n log(n))
- The average case is: Θ(n log(n))
- The worst case is: O(n^2) Now, let's analyze the above code. Best Case: It is the case where the partition element picks the element which is middle to be a pivot i.e.Since the algorithm will invoke recursively on first and second half. Thus - the total number of steps needed is the number of times it will take to reach from n to 1 if you divide the problem by 2 each step. So,there are n/2/2/2/2/..../2=1 *k times But, note that the equation is actually: n / 2^k = 1. Since 2^logn = n, we get k = long. So the number of steps (iterations) the algorithm requires is O(log), which will make the algorithm O(n log n) - since each iteration is O(n).

Worst case: In the worst case, if the first element is chosen as the pivot the Worst case complexity is chosen when the input to be sorted is in increasing or decreasing order. The reason for the worst case is that the after the partitioning, the partition size will be 1 and the other size will be n-1. Here,T(n) is the function: So, the Time to quick sort 'n' elements T(n) = Time it takes to partition 'n' elements O(n) + Time to quick sort 'n-1' elements T(n-1) So, T(n) = T(n-1) + O(n) => T(n) = O(n^2)