# Ch. 3 Algorithms

Sunday, April 19, 2020 3:08 PM

Fundamentals Ch. 3

In [ ]:
```
# if we could just print(my_die) and have the value of the die show up without hc
import random

class MSDie:
    """
    Multi-sided die

    Instance Variables:
        current_value
        num_sides

    """

    def __init__(self, num_sides):
        self.num_sides = num_sides
        self.current_value = self.roll()

    def roll(self):
        self.current_value = random.randrange(1,self.num_sides+1)
        return self.current_value

    def __str__(self):
        return str(self.current_value)

    def __repr__(self):
        return "MSDie({}) : {}".format(self.num_sides, self.current_value)


my_die = MSDie(6)
for i in range(5):
    print(my_die)
    my_die.roll()

d_list = [MSDie(6), MSDie(20)]
print(d_list)
```

## 3.1 Algorithm Analysis

Algorithm: Step-by-step list of instructions for solving any instance of a problem. Program: An algorithm that has been encoded into some programming language.

#### A better algorithm

- More readable (i.e. better variable name)
- Efficient in use of computing resources (i.e. space, memory)

Typesetting math: 0%

In [18]:
```
# Tracking the execution time for a function
import time


# One algorithmic approach
def sumOfN2(n):
   start = time.time()

   theSum = 0
   for i in range(1,n+1):
      theSum = theSum + i

   end = time.time()

# By calling this function twice, at the beginning and at the end, and then compu
   return theSum,end-start

# Test with n = 1 million
test = sumOfN2(10000000)
print("Using the first algorithmic approach, the sum is %d required %10.7f seconc
print()

#second algorithmic approach
def sumofN3(n):
    start = time.time()
    answer = (n*(n+1))/2
    end = time.time()

    return answer, end-start


test2 = sumofN3(10000000)
print("Using the first algorithmic approach, the sum is %d required %10.7f seconc


#The second algorithm is clearly faster and more efficient (in our language and c
```

```
Using the first algorithmic approach, the sum is 50000005000000 required  1.803
8225 seconds

Using the first algorithmic approach, the sum is 50000005000000 required  0.000
0000 seconds
```

## 3.3 Big-O Notation

A benchmark that judges algorithm alone from machine, program, time of day, compiler, and programming language.

Typesetting math: 0% The question is how do we distinguish the size of a computer problem, and our goal then is to show how the algorithm's execution time changes with respect to the size of the problem.

Big O Notation is an approximation of the number of steps it takes an algorithm to perform a function.

**Common Functions for Big O**

| O(n) | Name |
|---|---|
| 1 | Constant |
| log *n* | Logarithmic |
| *n* | Linear |
| *n*log*n* | Log Linear |
| $n^2$ | Quadratic |
| $n^3$ | Cubic |
| 2^n | Exponential |

Notice that when n is small, the functions are not very well defined with respect to one another. It is hard to tell which is dominant. However, as n grows, there is a definite relationship and it is easy to see how they compare with one another.

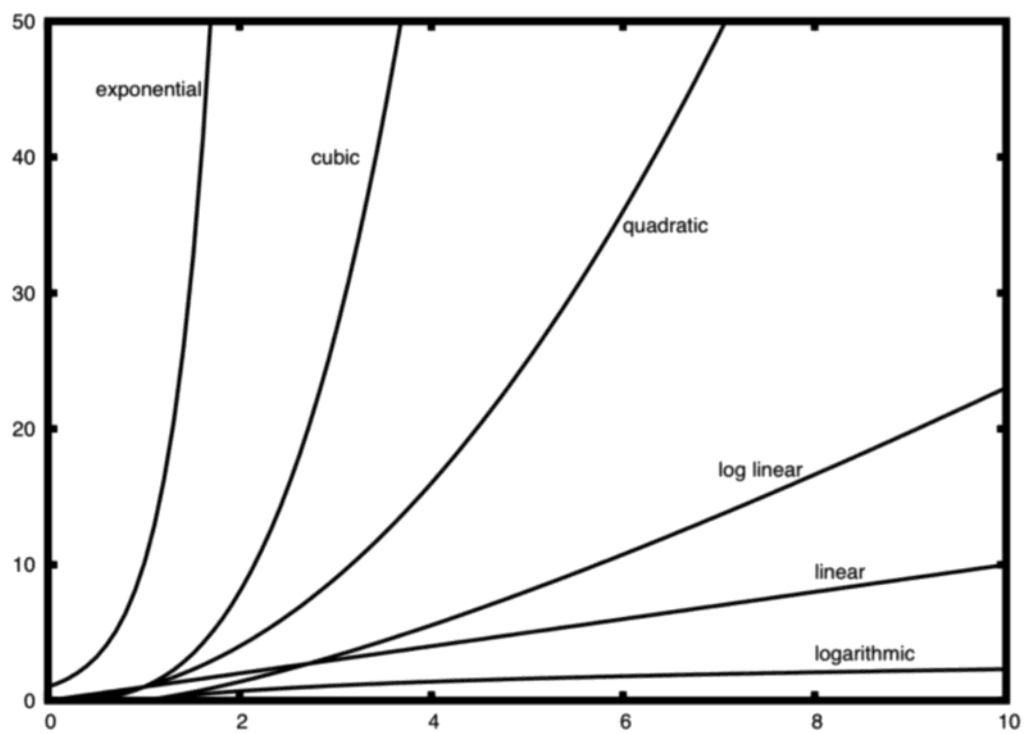


Figure 1: Plot of Common Big-O Functions

## Summary

Typesetting math: 0%

- N represents the size of the problem

- The bigger the O(n) value, the longer the growth of run time
- To check O(n)
  - Count the nested loops, esp if nested loops depend on N. Single nested loop is O(n^2)
  - Ask yourself how the problem grows with an increase of 1 for N (how many more iterations need to be done)
  - If the value of i is cut in half each time through the loop it will only take log n iterations.

#### Anagram Problem

Big-O Notation for Different Algorithms for an Anagram Checker (https://runestone.academy/runestone/books/published/pythonds/AlgorithmAnalysis/AnAnagramDete

The Final Solution

"Again, the solution has a number of iterations. However, unlike the first solution, none of them are nested. The first two iterations used to count the characters are both based on n. The third iteration, comparing the two lists of counts, always takes 26 steps since there are 26 possible characters in the strings. Adding it all up gives us T(n)=2n+26 steps. That is O(n). We have found a linear order of magnitude algorithm for solving this problem."

In other words, the run time never changes even as the word gets longer. The **worst case** scenario never changes the run time.

#### Cheat Sheet of O(n) of operations

Typesetting math: 0%

**Table 2: Big-O Efficiency of Python List Operators**

| Operation | Big-O Efficiency |
|---|---|
| index [] | O(1) |
| index assignment | O(1) |
| append | O(1) |
| pop() | O(1) |
| pop(i) | O(n) |
| insert(i,item) | O(n) |
| del operator | O(n) |
| iteration | O(n) |
| contains (in) | O(n) |
| get slice [x:y] | O(k) |
| del slice | O(n) |
| set slice | O(n+k) |
| reverse | O(n) |
| concatenate | O(k) |
| sort | O(n log n) |
| multiply | O(nk) |

## 3.5 Lists and Dictionaries Operating Time

Typesetting math: 0%

In [20]:
```
import timeit

#Adding list items by concatenating existing list l with items from another list
def test1():
    l = []
    for i in range(1000):
        l = l + [i]

#Appending
def test2():
    l = []
    for i in range(1000):
        l.append(i)

#List Comprehension: For Loop inside a List
def test3():
    l = [i for i in range(1000)]

#List Constructor Function
def test4():
    l = list(range(1000))

#Empty Function for Experimental Purity
def test0():
    pass



#Timing Test- importing tests from __main__ declutters the stray variables/functi
#First baseline: overhead time it takes to call an empty function
t0 = timeit.Timer("test0()","from __main__ import test0")
overhead = int(t0.timeit(number=1000))
t1 = timeit.Timer("test1()", "from __main__ import test1")
print("concat: ",t1.timeit(number=1000) - overhead, "milliseconds")
t2 = timeit.Timer("test2()", "from __main__ import test2")
print("append: ",t2.timeit(number=1000) - overhead, "milliseconds")
t3 = timeit.Timer("test3()", "from __main__ import test3")
print("comprehension: ",t3.timeit(number=1000) - overhead, "milliseconds")
t4 = timeit.Timer("test4()", "from __main__ import test4")
print("list range: ",t4.timeit(number=1000) - overhead, "milliseconds")
```

```
concat:  4.765215999999782 milliseconds
append:  0.32262719999971523 milliseconds
comprehension:  0.1275946999999178 milliseconds
list range:  0.03649310000037076 milliseconds
```

## 3.7 Dictionary

Dictionaries differ from lists in that you can access items in a dictionary by a key rather than a position

Typesetting math: 0%

- Checking to see whether a key is in the dictionary or not is also O(1
- Get item and set item operations on a dictionary are O(1)

**Table 3: Big-O Efficiency of Python Dictionary Operations**

| operation | Big-O Efficiency |
|---|---|
| copy | O(n) |
| get item | O(1) |
| set item | O(1) |
| delete item | O(1) |
| contains (in) | O(1) |
| iteration | O(n) |

# 4 Data Structure

**Content**

- Stack

Stack: "an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end."

```
* Last in First Out
* Staks are fundamentally important since they reverse the order of items
* i.e. Your browser has a back button to retrieve the last (most recent p
age)
```

Typesetting math: 0%