Guia 3.1 Python

Objetivo General: Realizar mediciones de eficiencia en los programas.

Objetivos Específicos:

1. Mostrar el uso de la notacion O grande.
2. Mostrar el uso de herramientas de medicion de eficiencia de algoritmos.

Notacion O grande

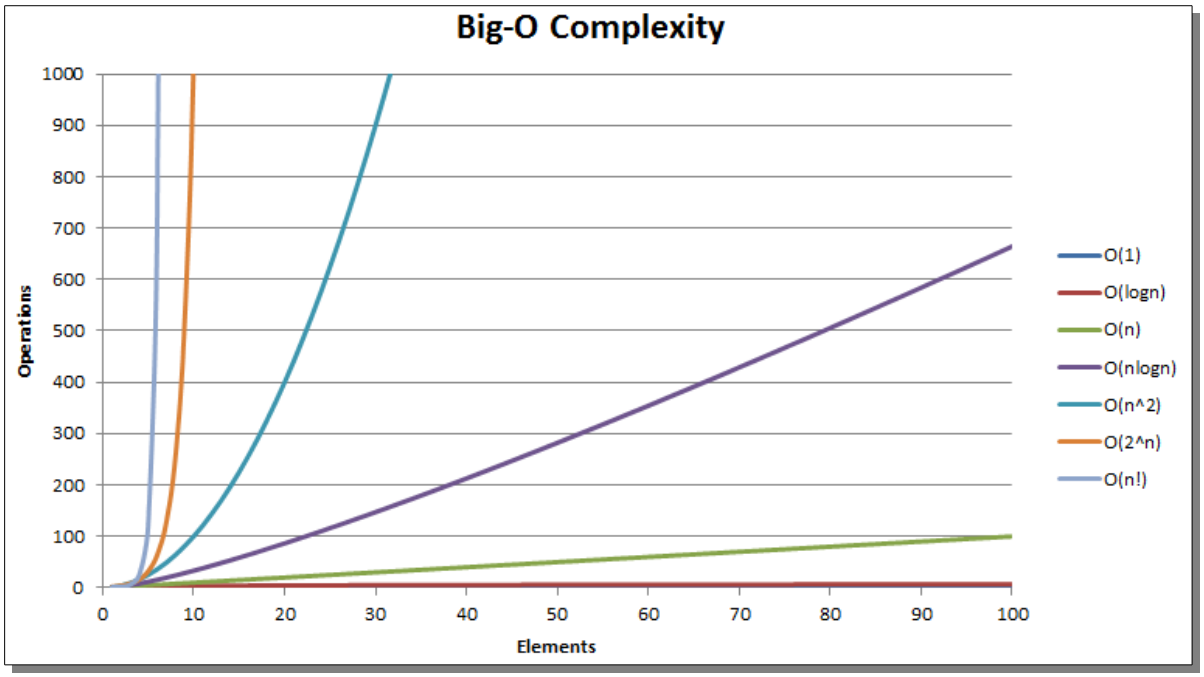
Teoría.

La notacion O grande, es un simbolismo usado en las ciencias de la computacion para describir el comportamiento de los algoritmos, consiste en identificar la velocidad de crecimiento de una funcion.

Cuando se analiza algun algoritmo, se puede encontrar que el numero de pasos que se necesita para completar el problema de tamaño “n” está dado por $T(n)=25n^5+4n^2+10$. Si ignoramos las constantes (el crecimiento que aportan es poco, si no se considera el hardware) y los exponentes mas pequeños, se puede concluir “T(n) crece en el orden de n^5 ” y escribir $T(n)=O(n^5)$.

Lista de las clases de las funciones que se encuentran normalmente cuando se analizan algoritmos, ordenadas de crecimiento lento a crecimiento rápido. “c” es una constante.

Notacion	Nombre
$O(1)$	Constante
$O(\log(n))$	Logaritmica
$O((\log(n))^c)$	Polilogaritmica
$O(n)$	Linear
$O(n^2)$	Cuadratica
$O(n^c)$	Polinomial
$O(c^n)$	Exponencial



1

La eficiencia de un algoritmo incluye:

1. Uso de CPU (tiempo)
2. Uso de memoria.
3. Uso de disco.
4. Uso de red.

Todos son importantes, pero lo que la notacion O grande mide es la complejidad en tiempo (uso de CPU). Es necesario diferenciar:

- Eficiencia: cuanto utiliza un programa cuando ejecuta.
- Complejidad: como cambia el uso de recursos.

El tiempo requerido por un algoritmo es proporcional al numero de operaciones básicas que realiza:

- Operaciones aritméticas.
- Asignaciones.
- Test.
- Lecturas.
- Escrituras.

Cuando estamos interecados en encontrar la complejidad de un programa no se busca la cantidad exacta de operaciones que se realizan, se busca la relacion de el numero de operaciones a realizar con el tamaño del problema. Típicamente el peor de los casos es el número máximo de operaciones.

1 http://bigocheatsheet.com/

Determinación de complejidad

Secuencia de sentencias.

```
sentencia1;
sentencia2;
sentencia3;
...
sentenciak;
```

El tiempo total está dado por:

$$tiempo\,total = \sum_{i=1}^{k} tiempo(sentencia\,1)$$

If – Else.

```
if(condicion):
    secuencia1;
elif:
    secuencia2;
elif
    ...
else:
    secuenciak;
```

El tiempo total está dado por:

$$tiempo\,total = max(tiempo(secuencia\,1), [tiempo(secuencia\,2), \ldots, tiempo(secuenciak)])$$

Ciclos

```
for(i in range(N)):
    sentencia1;
```

El tiempo total está dado por:

$$tiempo\,total = N * tiempo(sentencia\,1)$$

Ciclos anidados

```
for i in range(N):
    for i in range(M):
        sentencia1;
```

El tiempo total está dado por:

$$tiempo\,total = N * M * tiempo\left(sentencia\,1\right)$$

Ejercicio - CriptoAlfabeto.

El problema consiste en encontrar que digitos deben reemplazar a que letras para que la ecuacion sea correcta. Ejemplo:

$$\mathrm{ODD+ODD = EVEN}$$

Para este problema hay multiples soluciones:

Solucion de fuerza bruta

Esta solucion consiste en escribir todas las combinaciones posibles de reemplazo y probarlas hasta encontrar una o todas las que resuelven el acertijo.

El diseño sera el siguiente:

1. Se tiene una cadena que representa el acertijo. Ej: 'ODD+ODD==EVEN'.
2. A esa cadena se generarán todos los reemplazos posibles, por ejemplo: sustituir O=1, D=2, E=3, V=4, N=5; la cadena quedaría '122+122==3435'
3. evaluar la cadena, si es True es una solucion, si es False probar con otra permutacion.

Ahora que se tiene un enfoque de desarrollo, completar el archivo «semana_3/cripto_aritmetica/fuerza_bruta.py»