Programación II
Práctica 01: Complejidad algorítmica

Notación O

$f \in O(g) \Leftrightarrow \exists\ n_0, c$ tales que para todo $n > n_0$
$f(n) < c\ g(n)$

Donde n, es el tamaño de la instancia que es parámetro del algoritmo.

Reglas y Álgebra de Ordenes

1) $O(1) \leq O(\log n) \leq O(n) \leq O(n^k) \leq O(k^n) \leq O(n!) \leq O(n^n)$

2) $O(f) + O(g) = O(f + g) = O(\max\{f, g\})$

3) $O(f)\ O(g) = O(f\ g)$

4) $O(k) = O(1)$ para todo k constante.

Ejemplo: O(2n)
= O(2) * O(n) // por regla 3
= O(1) * O(n) // por regla 4
= O(n)

5) $\sum_{i=1}^{k} O(f) = O(\sum_{i=1}^{k} f) = O(k\ f)$
Si k es constante, entonces vale O(f)

6) $\sum_{i=1}^{k} i = \frac{k*(k+1)}{2} = O(k^2)$

Calcular el Orden de complejidad algorítmica para el peor caso y para el caso promedio de los siguientes ejercicios.

Ejercicio1 (Obligatorio): Determinar la complejidad para el peor de caso de los siguientes algoritmos

a)

```
Void función(int n)
      if n == 1
            for (i = 1; i<n; i++ )
      else
            for (i = 1; i<n^2; i++ )
```

b)

```
Void función(int n)
      if n != 1
            for (i = 1; i<n; i++ )
      else
            for (i = 1; i<n^2; i++ )
```

c)

```
Void función1(int x)
      if f(x)
            g(x)
      else
            h(x)
```

Ejercicio2 (Obligatorio): Burbujeo

a)

```
for(int i = 0; i < n; i++) {
      for(int z = 0; z < n; z++)    {
            if(vector[z] > vector[z + 1]){
            aux = vector[z];
            vector[z] = vector[z + 1];
            vector[z + 1] = aux;
            }
      }
}
```

b) ¿Como cambia el orden si cambiamos z = 0 por z = i, en el segundo “for”?

Variables de complejidad

Antes de continuar identificaremos la o las variables que representan el tamaño de los datos de entrada el algoritmo.
La función f() que me mide la complejidad , estará en función de dichas variables.

Ejemplo1

```
Void función1(k, h)
        for (i=0, i < k, i++)
                h++
```

Los candidatos a variables son k y h, pero como h++ esta en O(1) , la complejidad no depende de h
Asi que la variable de complejidad es k y esto surge de que el ciclo se repite k veces.

Variable: k
La complejidad de funcion1 es O(k)

```
Void función2(k, h)
        for (i=0, i < 2^k, i++)
                h++
```

Notar que el ciclo no termina en k pasos.

La complejidad de funcion2 es $O(2^k)$

Elemplo2

```
Void función3(k, h)
        for (i=0, i < k, i++) { }

        for (i=0, i < h, i++) { }
```

El primer ciclo depende de k, pero el segundo cido depende de h.

Entonces que variable usaremos?

En este caso se necesitan las dos

Pero cual es la complejidad? O(k) o O(h). Como saber si k > h o si h > k ?

La realidad es que en general no se sabe.

Entonces tendremos que poner alguna de las siguientes expresiones

(1) O(k + h)
(2) O(max(k, h))

La segunda expresión es más precisa.

Supongamos como ejemplo que k > h. claramente k +h sigue siendo mayor que k.

Entonces como puede ser que la complejidad sea O(k)

Vamos a buscar una cota:

Como k > h, sabemos que 2k > k + h

Entonces la complejidad es O(2k) , pero por el álgebra de la complejidad es lo mismo que O(k)

O(2k) = O(2) O(k) = O(1) O(k) = O(k)

Ejercicio3 *: Test de primalidad

1)

```
boolean esPrimo1(n){
      int i = 1
      int divisores  = 0
      while i < n
            if divide(i,n)
                  divisores ++
            i++
      return (divisores < 2)
}
```

2) Para la parte 2 utilizar la siguiente proposición P que dice:
Solo hace falta chequear los divisores hasta $\sqrt{n}$ para ver si esPrimo(n)

Demostración

Demostraremos por absurdo que eso no es necesario.
Asumiremos que No P es cierto.
Hipótesis(No P): Vamos a suponer que existe un divisor mayor que $\sqrt{n}$ que hace falta ver.

Sea k un divisor de n tal que k > $\sqrt{n}$ =>
Existe q, otro divisor, pues q = n / k

Si q $\leq \sqrt{n}$ => no era necesario ver k.
Abs
Con q me alcanza para el test.

Si q > $\sqrt{n}$ => k * q > $\sqrt{n} * \sqrt{n}$
Pero $\sqrt{n} * \sqrt{n}$ = n =>
k * q > n
Abs

=> P es cierto: No hace falta ver divisores mayores que $\sqrt{n}$

```
boolean esPrimo2(n){
      int i = 1
      int divisores  = 0
      while i < Math.SQRT(n)  //por propiedad1
            if divide(i,n)
                  divisoreus ++
            i++
      return (divisores < 2)
}
```

3) Utilizar la siguiente propiedad:
La cantidad de primos en los primeros n números no es mayor que n / ln(n).
Solo en este caso ln esta en base e, en lugar de base 2 como en el resto de las practicas.
Por ej: $\ln_e 32 = 3{,}46$; mientras que $\ln_2 32 = 5$

Implementar esPrimo3, el cual recibe la lista de números primos menores que $\sqrt{n}$ y n como parámetros.
Calcular el nuevo Orden de complejidad.

Ayuda:
Calcular la lista de primos a mano para realizar los ejemplos.
La complejidad tiene que ser mejor que el ejercicio 2

```
boolean esPrimo3(listaPrimos,n)
```

Opcional 4) Implementar algún otro test de primalidad y calcular el orden.
http://es.wikipedia.org/wiki/Test_de_primalidad

Ejercicio4 Fósforos

Se tiene una caja de fósforos con n fósforos nuevos.
Cada vez que quiera utilizar uno, el procedimiento es el siguiente:
Tomo un fosforo de la caja. Si está quemado, tomo otro, y así hasta encontrar uno nuevo.
Luego utilizo el fosforo y lo mezclo junto con los otros fósforos usados en la caja

a) ¿Cuál es la complejidad de encontrar un fosforo sin quemar dado que ya consumí la mitad de la caja?
b) ¿Cuál es la complejidad de consumir n fósforos?

Ejercicio5 (obligatorio): Definición de O

Utilizando la definición de O ($f \in O(g) \Leftrightarrow \exists\ n_0, c$ tales que para todo $n > n_0 \Rightarrow f(n) < c\ g(n)$),
Encontrar n_0 y c para justificar el orden de los siguientes tiempos de ejecución.
Decidir en que Orden(el mas chico) están.

a) $n^2 - n^2 + 100$
b) $n^{3/2} + n^{1/2} + 100$
c) $n^3 + 2n^2 + 10$
d) $\sqrt{n} + \log n + 1000$
e) $n^n + n^{10} + 10$

Ejercicio6 Varias variables

Implementar un algoritmo que recorra y muestre una matriz de n filas y k columnas.
Calcular la complejidad de dicho algoritmo utilizando la definición de O para dos variables

$f_{n,k} \in O(g_{n,k}) \Leftrightarrow \exists\ n_0, k_0, c$ tales que para todo $n > n_0, k > k_0 \Rightarrow f(n,k) < c\ g(n,k)$

Ayuda: Nombrar las variables antes de calcular la complejidad

Ejercicio7 Combinatoria

En una habitación hay **n** personas que se quieren saludar entre si.
Las reglas para saludarse son:

1) Cada persona puede saludar solo una persona a la vez.
2) El saludo es simétrico: Si a saluda a b, se considera que b saluda a a en la mismo saludo.
3) Cada saludo demora 1 segundo
4) Si hay n personas, puede haber hasta n/2 saludos simultáneos.

a) Cuanto tiempo(en segundos) se necesita que todos queden saludados?

Ayuda: Hacer una tabla para el caso de 4 y de 8 personas.

b) Cuantos saludos hay en total?

c) Como cambia a) si cambiamos la regla 4) de manera que solo puede haber un saludo a la vez?

d) Como cambia a) si cambiamos la regla 1) de manera que cada persona puede saludar a mas de una persona a la vez?

e) Como cambia a) si quitamos la regla 1) y además, cada persona que es saludada sale de la habitación.

Ayuda: Hacer una tabla para el caso de 4 y de 8 personas.

Bibliografía:

[Cormen1990]: http://www.amazon.com/exec/obidos/ISBN=0262031418/none01A/

[Cormen2001]: Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford: Introduction to Algorithms. Sección 31.8: "Primality testing", pp.887–896. MIT Press & McGraw-Hill, 2a edición,2001. (ISBN 0-262-03293-7.)