Departamento de Programación
Facultad de Informática
Universidad Nacional del Comahue

Apunte: Eficiencia

1- Análisis de algoritmos:

Un algoritmo es un conjunto finito de instrucciones claramente especificadas que la computadora debe seguir para resolver un problema. El código debe ser eficaz y eficiente. Eficaz para asegurar que el algoritmo realice lo que se supone que haga y que lo haga bien. Por ejemplo un algoritmo que ordena números, lo debe realizar ordenando todos los números correctamente. Eficiente para que el asegurar utilizar la menor cantidad de recursos posible.

Los recursos a los cuales se hace referencia son espacio y tiempo. Si un algoritmo utiliza varios gigabytes de memoria principal para resolverse no es útil en la mayoría de la máquinas actuales. De la misma manera si un algoritmo utiliza mucho tiempo de procesamiento tampoco es útil, aunque sea completamente correcto.

El tiempo necesario para ejecutar un algoritmo depende casi siempre de la cantidad de datos que el mismo debe procesar. Es de esperar, por ejemplo, que ordenar diez mil elemento requerirá mucho más tiempo de procesamiento que ordenar cien. El tiempo de ejecución de un algoritmo es por lo tanto en función del tamaño de la entrada de datos. Si bien el valor exacto de esta función depende de muchos factores, tales como la velocidad de la máquina, la calidad del compilador, y en algunos casos, de la calidad del programa.

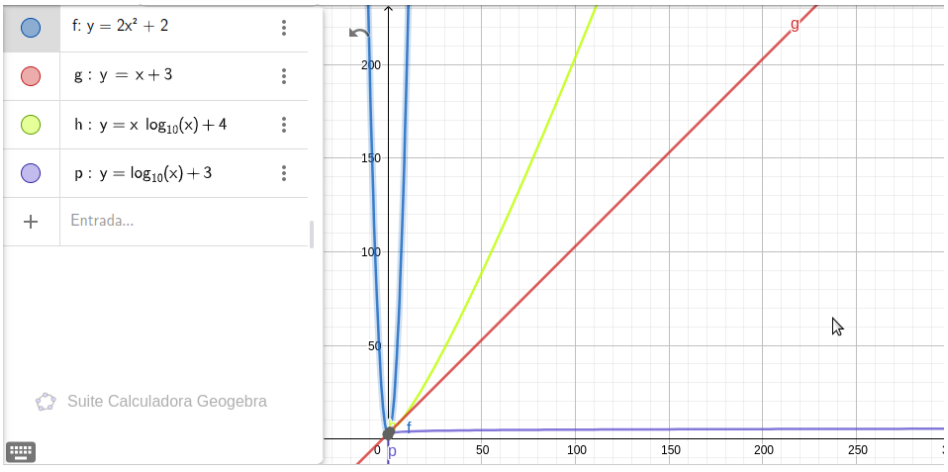
2-Relaciones del tamaño de entrada de datos:

Es posible definir una función de tiempo que toma parta de la cantidad de datos a procesar y de un tiempo necesario para ejecutar dicho programa. Se puede entonces dibujar la gráfica que representa la función del tiempo de ejecución de un algoritmo.

Es posible graficar las curvas que representan cuatro funciones típicas del análisis de algoritmos: función lineal, logarìtmica, cuadrática y cúbica.

El tamaño de la entrada n varia de 1 a 100 elementos, y los tiempos de ejecución asociados varian de 0 a 10 milisegundos.

Un simple vistaso sugiere que para un valor n muy grande la curva lineal y (n log n) crece mucho más despacio que la cuadrática y cúbica.



Recordemos la notación de estos términos en el siguiente polinomio que representa una función donde a es el factor del término cuadrático, b del término lineal y c del término independiente.

$$y = an^2 + bn + c$$

El problema de descargar un fichero de Internet nos sirve como ejemplo. Supongamos que hay un retraso inicial de 2 segundos (para establecer la conexión), después del cual la descarga se lleva a cabo a 1,6 K/seg. Entonces, si el tamaño del fichero es N kilobytes, el tiempo de descarga viene descrito por la fórmula T(n): n/1,6 + 2. Esta es una función lineal. Podemos ver que la descarga de un fichero de 80K tardaría aproximadamente 52 segundos, mientras que descargar un archivo del doble de grande (160K) consumiría alrededor de 102 segundos, es decir casi el doble.

Esta propiedad, por la cual el tiempo de ejecución es esencialmente proporcional al tamaño de la entrada, caracteriza a un algoritmo lineal, y es el escenario más favorable. Como puede verse, en las gráficas, algunas curvas no lineales conducen a tiempo de ejecución mayores. Esto indica, en particular, que un algoritmo lineal es mucho mejor y tardaría mucho menos que uno cúbico.

La función $10\,n^3 + n^2 + 40\,n + 80$ es una función cúbica. De igual manera, una funci#on $6\,n^2 + 5\,n + 4$ es cuadrática ya que su mayor término es n^2 por alguna constante, de la misma manera para la función lineal. El logaritmo es una función que crece lentamente, por ejemplo el logaritmo de 1.000.000 (en base 2) es solo 20. La función logaritmo crece más lentamente que cualquier función raíz.

Dadas dos funciones, una cualquiera puede ser menor que la otra en un punto dado, por lo que en general no tiene sentido afirmar f(n) < g(n). No obstante nosotros medimos el índice de crecimiento de las funciones. Esto viene justificado por tres razones:

Los valores pequeños de n generalmente no son importantes. Para n=20 todos los algoritmos terminan antes de 5 ms. La diferencia entre el mejor y el peor algoritmo es menor que un parpadeo de ojos.

3- Tiempo de ejecución

Es el tiempo que tarda en ejecutarse una cantidad de sentencias en un algoritmo hasta alcanzar la solucion. Existen dos recursos fundamentales en los cuales se basa la eficiencia de un algoritmo: Espacio y Tiempo. Ambos determinan la eficiencia de un algoritmo, el espacio se refiere a la cantidad de memoria que utiliza al ejecutarse, el tiempo, a la duración de la ejecución del algoritmo desde que se comienza a ejecutar hasta su finalización.

Nos enfoacremos en el tiempo. La utilizacion de recursos en la ejecucion de un programa depende del tamaño de los datos de entrada. De esta manera podemos decir que el tiempo de ejecución de un programa establece una relación de dependencia entre el número que representa el tamaño de los datos de entrada y un numero real que representa el tiempo de ejecucion.

Se establece una función matemática respecto al tiempo de ejecución

$f : \mathbf{N} \rightarrow \mathbf{R+}$ Donde el **N** (dominio) representa el tamaño de la entrada y corresponde a números naturales,

y **R+** (imagen) cdescribe la cantidad de veces que se ejecuta de acuerdo a ese tamaño de entrada.

Entonces para analizar la eficiencia de los algoritmos se pueden utilizar dos enfoques: empírico y teórico.

- Enfoque empírico: (a posteriori) Prueba y error. Son técnicas para testear programas en base a lotes de datos de prueba.
- Enfoque teórico:(a priori) Determina matemáticamente la cantidad de recursos necesarios en función del tamaño de los datos de entrada,

Existen distintos tipos de análisis teóricos:

- Del Peor Caso: se considera el máximo uso de recursos para todas las instancias de tamaño n.

- Del Caso Promedio: se considera un promedio de uso de recursos para todas las instancias de tamaño n.
- Probabilístico: se considera el uso de recursos de cada instancia de tamaño n en funcion de su probabilidad de ser ejecutada. (es muy complejo)
- Del Mejor Caso: se considera el minimo uso de recursos para todas las instancias de tamaño n

Analizaremos el peor caso porque constituye una cota superior al total de los recursos utilizados por el algoritmo. Conocerla nos asegura que no superara esa cantidad, además en algunos casos, el peor caso es el que ocurre más seguido.

3.1. Reglas Generales para calcular tiempos de ejecución

Las operaciones elementales corresponden a un tiempo de ejecución acotado por una constante que depende de la implementación, por convención se toma la unidad.

1. Las proposiciones elementales se suman.
2. Proposiciones consecutivas se suman. .
3. Las proposiciones alternativas se suma el costo de la comparación con el mayor costo de las proposiciones de la alternativa. roposiciones consecutivas se suman.
4. Los ciclos repetitivos, se calcula la cantidad de iteraciones por lo que cuesta ejecutar el interior más las comparaciones y el incremento.
5. Ciclos repetitivos anidados, de adentro hacia afuera, se calcula el costo de la proposición interior multiplizado por el de la exterior.

3.2. Tiempo de ejecución para secuencias de instrucciones

Se suman las operaciones de cada sentencia

```
….
        cant ← arreglo[i] * 2
        int i ← sum + 2
        return i
```

Ejemplo de Operaciones Elementales

*cant ← arreglo[i] * 2* Tres operaciones elementales,
una asignación,
un acceso al valor del arreglo iésimo del arreglo y
una multiplicación.

int i ← sum + 2 Dos operaciones elementales,
una asignación,
una suma

return i Una operación elemental,

Requiere: `Tiempo = 3 + 2 + 1 = 6`

3.3. Tiempo de ejecución para alternativa simple

```
Si (cond) ENTONCES
     …...S1
FIN SI
```

Requiere: Tiempo Si = $t_{cond} + t_1$

3.4. Tiempo de ejecución para alternativa compuesta

```
Si (cond) ENTONCES
    S1
SINO
    S2
FIN SI
```

Requiere: Tiempo Si/Sino = $t_{cond} + max(t_1, t_2)$

El tiempo de ejecución del peor caso $T_w(n)$ **es** *el mayor tiempo de ejecución de* ***S1*** *y* ***S2***

3.5. Tiempo de ejecución para iterativas tipo PARA

```
PARA i ← 1 HASTA n HACER
    S1
FIN PARA
```

Requiere:Tiempo para = $t_{ini} + cantIt * (t_{cond} + t_{int} + t_{inc}) + t_{cond}$

- *La inicialización de la variable =* T_{ini}
- *La evaluación de la condición =* T_{cond}
- *La evaluación de las condiciones para decidir la finalización se realiza una cantidad de iteraciones cantIt de veces en true y la última vez es falsa.*
- *Los incrementos se evalúan con el tiempo* T_{inc}
- *El tiempo interno del ciclo es la serie de sentencias internas S1* T_{int} *)*

Ejemplo de Repetitiva

```
PARA j ← 4 HASTA n PASO 2 HACER
    a[j] ← a[j] + n
FIN PARA
```

Requiere:Tiempo para = $t_{ini} + cantIt * (t_{cond} + t_{int} + t_{inc}) + t_{cond}$

- *La inicialización de la variable =* T_{ini} *(j = 4) es un tiempo de asignación*
- *La evaluación de la condición =* T_{cond} *(j < n) es un tiempo de asignación*
- *La evaluación de las condiciones para decidir la finalización se realiza una cantidad de iteraciones*

  cantIt *= (valor Final – valor inicial) /paso, en nuestro caso (n-4)/2*

- *Tiempo interno =* T_{int}
  - *1 asignación + 1 suma + 2 acceso al vector = 4 tiempos*
  - *Cantidad de Itenaciones = (n-5)/2 veces*

- *Los incrementos se evalúan con el tiempo* T_{inc} *(j++) son dos tiempos de asignación*

Tiempo Para = t_{ini} + ***cantIt****(t_{cond} + t_{int} + t_{inc}) + t_{cond} = 1 + (n-4)/2* (1 + 4 + 2) + 1 = **.....**

Repetitiva PARA: se tiene en cuenta el tiempo de la inicialización de la variable de control T_{ini} , luego por cada repetición del ciclo $\sum_{min}^{max}(T_i)$ (Pmax min), donde min es el valor de inicialización y max es el último valor para el que la condición da true; sumamos el tiempo para evaluar la condición T_{cond} , el tiempo del cuerpo de sentencias interno del bucle T_{int} , y el tiempo del incremento de la variable de control T_{inc} . Además consideramos que la evaluación de la condición T_{cond} se ejecuta una vez más que las que se ejecuta el bloque interno (cuando la condición es falsa).

3.6. Tiempo de ejecución para iterativas tipo MIENTRAS

```
MIENTRAS cond HACER
    S1
FIN MIENTRAS
```

Requiere:Tiempo mientras = ***cantIt *(t_{cond} + t_{int})+ t_{cond}***

- *La evaluación de la condición =* T_{cond}
- *La evaluación de las condiciones para decidir la finalización se realiza una cantidad de iteraciones cantIt de veces en true y la última vez es falsa.*
- El tiempo interno del ciclo es la serie de sentencias internas S1 T_{int}

3.7. Tiempo de ejecución para iterativas tipo REPETIR HASTA

```
REPETIR
    S2
HASTA cond
```

Requiere:Tiempo REPETIR HASTA = **+ *cantIt *(t_{int} + t_{cond})***

- *La evaluación de la condición =* T_{cond}
- *La evaluación de las condiciones para decidir la finalización se realiza una cantidad de iteraciones* ***cantIt***
- *Tiempo interno =* T_{int}

3.8. Tiempo de ejecución para bucles anidados

```
PARA i ← 1 HASTA n-1 HACER
    PARA j ← 1 HASTA n HACER
        S1
    FIN PARA
FIN PARA
```

$$T_n=\sum_{i=1}^{n-1}\sum_{j=1}^{n} k=\sum_{i=1}^{n-1} kn=k\sum_{i=1}^{n-1} n=k\,n\,(n-1)=kn^2-kn$$

3.9. Ejemplos para analizar los tiempos de ejecución de un algoritmo

Primer ejemplo

```
private static void ejemplo1 (int n)    0.
{ int i,j,k=0;                          1.
  for (i = 1: i <= n-1; i++)            2.
     { for (j = 1: j < n-1; j++)       3.
        {  k= i*j;    }                 4.
      }                                 5.
}                                       6.
```

$$T(n)=T_{s1}+T_{s2}$$

$$T_{s1}=1$$

$$T_{s2}=1+\sum_{i=1}^{n-1}(2+T_{s3}+2)+2$$

$$T_{s4}=1+1=2$$

$$T_{s3}=1+\sum_{j=1}^{n-2}(2+T_{s4}+2)+2$$
$$T_{s3}=3+\sum_{j=1}^{n-2}(T_{s4}+4)$$
$$T_{s3}=3+\sum_{j=1}^{n-2}(2+4)$$
$$T_{s3}=3+\sum_{j=1}^{n-2}(6)$$
$$T_{s3}=3+6*(n-2)$$
$$T_{s3}=3+6n-12$$
$$T_{s3}=6n-9$$

$$T_{s2}=1+\sum_{i=1}^{n-1}(2+T_{s3}+2)+2$$
$$T_{s2}=3+\sum_{i=1}^{n-1}(2+T_{s3}+2)+2$$
$$T_{s2}=3+\sum_{i=1}^{n-1}(2+6n-9+2)$$
$$T_{s2}=3+\sum_{i=1}^{n-1}(6n-5)$$
$$T_{s2}=3+(6n-5)(n-1)$$
$$T_{s2}=3+6n^2-6n-5n+5$$
$$T_{s2}=6n^2-11n+8$$

$$T_n=T_{s1}+T_{s2}$$

$$T_n=1+6n^2-11n+8$$

$$T_n = 6n^2 - 11n + 9$$

Segundo ejemplo

```
private static int ejemplo2 (int n)     0.
{ int i,j,k, cantA =0, cantB = 0;       1.
  for (i = 1: i <= n; i++)              2.
     { for (j = 1: j < i; j++){         3.
           contA++;                     4.
           contB++;    }                5.
       for (k = i: k < n; k++){         6.
           contA++;                     7.
           contB+= contA;    }          8.
      }                                 9.
 return contB;                         10.
}                                      11.
```

$$T_n = T_{s1} + T_{s2} + T_{10}$$

$$T_{s1} = 2$$

$$T_{s10} = 1$$

$$T_{s2} = 1 + \sum_{i=1}^{n} (1 + T_{s3} + T_{s6} + 2) + 1$$

$$T_{s2} = 2 + \sum_{i=1}^{n1} (T_{s3} + T_{s6} + 3)$$

Primero calculo T_{s3} **y** T_{s4}

$$T_{s3} = 1 + \sum_{j=1}^{i-1} (1 + T_{s4} + T_{s5} + 2) + 1$$

$$T_{s4} = 2$$

$$T_{s5} = 2$$

$$T_{s3} = 2 + \sum_{j=1}^{i-1} (1 + 4 + 2)$$

$$T_{s3} = 2 + \sum_{j=1}^{i-1} (7) = 2 + 7(i-1) = 7i - 5$$

$$T_{s6} = 1 + \sum_{k=i}^{n-1} (1 + T_{s7} + T_{s8} + 2) + 1$$

$$T_{s6} = 2 + \sum_{k=i}^{n-1} (T_{s7} + T_{s8} + 3)$$

$$T_{s7} = 2$$

$$T_{s8} = 2$$

$$T_{s6} = 2 + \sum_{k=i}^{n-1} (2 + 2 + 3)$$

$$T_{s6} = 2 + \sum_{k=i}^{n-1} (7)$$

Aplicando la regla correspondiente

$$T_{s6} = 2 + \sum_{k=1}^{n-1} (7) - \sum_{k=1}^{i-1} (7)$$

$$T_{s6} = 2 + \sum_{k=1}^{n-1} (7) - \sum_{k=1}^{i-1} (7)$$

$$T_{s6} = 2 + (7)(n-1) - (7)(i-1)$$

$$T_{s6} = 2 + 7\mathrm{n} - 7 - 7\mathrm{i} + 7$$

$$T_{s6} = 2 + 7\mathrm{n} - 7\mathrm{i}$$

$$T_{s2} = 1 + \sum_{i=1}^{n} (T_{s3} + T_{s6} + 3) + 1$$

$$T_{s2} = 2 + \sum_{i=1}^{n} (1 + 7\mathrm{i} - 5 + 7\mathrm{n} - 7\mathrm{i} + 2 + 2)$$

$$T_{s2} = 2 + \sum_{i=1}^{n} (7\mathrm{n})$$

$$T_{s2} = 2 + (n)(7\mathrm{n}) = 2 + 7n^2$$

$$T(n) = T_{s1} + T_{s2} + T_{s9}$$

$$T(n) = 2 + 7n^2 + 1 = 7n^2 + 3$$

4- Notación asintótica

La notacion asintotica, es una funcion que sirve de cota superior o techo de un conjunto de funciones y es de gran utilidad para clasificar la eficiencia de los algoritmos.

Se utilizará la notacion O para representar el índice de crecimiento. Por ejemplo, el tiempo de ejecución de un algoritmo cuadrático se describe como $O(n^2)$ (pronunciado del orden de n al cuadrado). La notacion O nos permite establecer el orden relativo entre funciones, comparando los términos dominantes.

La notación O no garantiza una respuesta asintótica exacta sino sólo una cota superior.

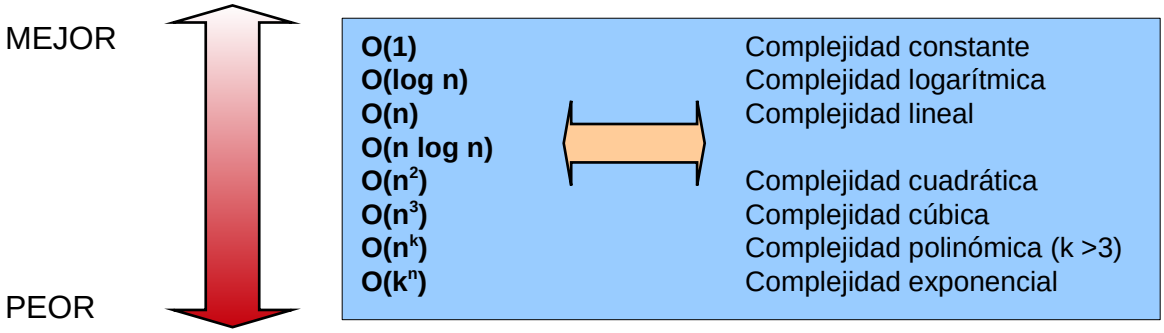
Para valores pequeños de n (por ej. aquellos menores que 30) la figura muestra que hay puntos para los cuales una curva es inicialmente mejor a otra, aunque luego esto deja de ser cierto.

Por ejemplo, la curva cuadrática es mejor inicialmente que la curva O(n log n), pero cuando n se hace suficientemente grande el algoritmo cuadrático pierde esta ventaja.

Para valores pequeños del tamaño de la entrada, es difícil comparar las funciones porque las constantes multiplicativas llegan a ser muy significativas.

Ejemplo: La función n + 2.500 es mayor que n^2 cuando n es menor que 50. A partir de un punto, la función lineal será

siempre menor que la función cuadrática.



Lo que es más importante, para tamaños pequeños de la entrada los tiempos de ejecución llegan a ser insignificantes, por lo que no tenemos que preocuparnos de ellos las limitaciones aparecen al trabajar con cantidades de datos grandes.

La característica más notoria de estas curvas es que los algoritmos cuadráticos y cúbicos no pueden competir con los restantes para tamaños de la entrada razonablemente grandes. Siempre un algoritmo lineal será mucho mas eficiente que uno cuadrático. Por tanto lo primero que se debe hacer es optimizar el algoritmo.

Los algoritmos de complejidad $O(n)$ y $O\ (n * log\ n)$ son los que muestran un comportamiento más natural, prácticamente a doble cantidad de datos procesados, doble de tiempo necesario. Los algoritmos de complejidad logarítmica son excelentes pues para resolver un problema el doble de grande sólo hace falta un poco más de tiempo. Sobre los algoritmos de tipo polinómico, mientras complejidades del orden $O(n^2)$ y $O(n^3)$ suelen ser aceptables, nadie acepta algoritmos de orden mayor. Finalmente, cualquier algoritmo por encima de una complejidad polinómica se dice "intratable" y sólo será aplicable a problemas muy pequeños.

Un algoritmo lineal resuelve el problema de tamaño 10.000 en una pequeña fracción de segundos. El algoritmo O(n log n) utiliza aproximadamente 10 veces ese tiempo.

4.1. Definición Matemática:

$$O(f(n)) = \{t: \mathbb{N} \rightarrow \mathbb{R}+ \mid (\exists\ c \in \mathbb{R}+, \exists\ n_0 \in \mathbb{N}\ n_0 : \forall\ n >= n_0,\ \ t(n) \leq c\ f(n)]\}$$

El *Orden* ***f(n)*** corresponde al conjunto de las funciones ***t*** que van de Naturales a Reales positivos, tal que existe una constante ***c*** arbitraria que pertenece a los Reales positivos donde *para todo* ***n*** que pertenece a los Naturales mayor a un valor n_0 , la función ***t(n)*** es menor o igual que ***c*** *por* ***f(n)***.

Decimos que f(n) es una función que acota superiormente a t(n)

Visualmente buscamos una función que partir de un valor dado, multiplicada por una constante es techo de la función t.

La figura siguiente ordena en orden creciente de tasa de crecimiento distintas funciones que describen comúnmente el tiempo de ejecución de los algoritmos.

Matemáticamente hablando, cuando N tiende a infinito. Es decir, lo que nos interesa conocer es el comportamiento asintótico de la función del tiempo de ejecución. El conjunto de funciones que comparten un mismo comportamiento asintótico comparten un orden de complejidad. El orden de complejidad lo escribimos $O(f(n))$ donde $f(n)$ representa la velocidad de crecimiento de la función del tiempo de ejecución. Para calcularlo podemos hacerlo de dos maneras:

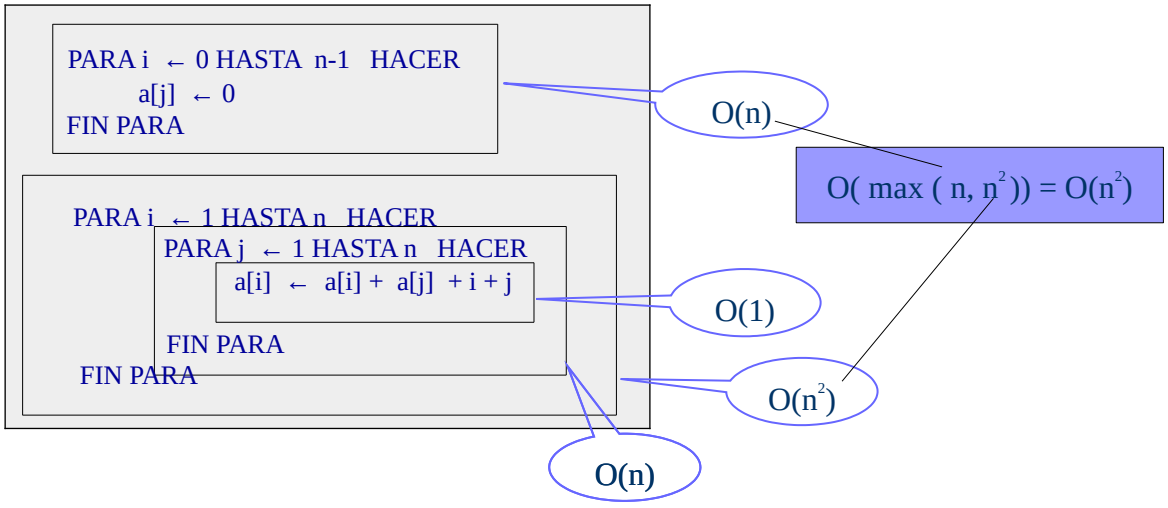
- A partir de la función de tiempo de ejecución: cuando se conoce la función del tiempo de ejecución, se aísla el término que crece más rápido, despreciando las constantes que lo multiplican. Ejemplo, si $T(n) = 4n^2 + 5n + 8$ su orden es $O(n^2)$.
- A partir del código (o seudocódigo), simplificando las sentencias de tiempo constante (orden $O(1)$) y aplicando las siguientes reglas:
  - Sean $T_1(n)$ y $T_2(n)$ las funciones que expresan los tiempos de ejecución de dos fragmentos de un programa,

y sus órdenes $T_1(n) \in O(f_1(n))$ y $T_2(n) \in O(f_2(n))$

- Regla de la suma: El orden de la suma de dos bloques de código secuenciales es el máximo de sus órdenes. $O(f_1(n)) + O(f_2(n)) = O(max(f_1(n), f_2(n)))$
- Regla del producto: El orden del producto de ambos bloques es el producto de sus órdenes.

  $O(f_1(n)) * O(f_2(n)) = O(f_1(n) * f_2(n))$

4.2. Cálculo del Orden sobre un algoritmo



4.2. Validación empírica

Es posible realizar una validación empírica para verificar si un algoritmo es *O(f(n))*. Esto se puede realizar calculando los valores de *T(n)/F(n)* para una serie de valores de n, donde *T(n)* es el tiempo de ejecución observado empíricamente.

Si *f(n)* es una respuesta ajustada al tiempo de ejecución, entonces los valores convergen a una constante positiva. Si *f(n)* es una sobreestimación, los valores convergen a cero. Si *f(n)* es una subestimación, y por lo tanto errónea, los valores divergen.

Comparamos dos funciones, una respecto de la otra. Siempre se puede determinar las velocidades de crecimiento relativas de dos funciones ***f(n)*** y ***g(n)***

Computar $\lim_{N \to \infty} f(n)/g(n)$

- 0 g tiene mayor vel. de crecimiento
- ∞ f tiene mayor vel. De crecimiento
- ≠ 0 f y g tienen igual velocidad de crecimiento

Veamos como ejemplo $f(n) = \log n$ y $g(n) = n^2 + 2$

$\lim_{N \to \infty} f(n)/g(n) = \lim_{N \to \infty} (\log n/n^2 + 2) = \lim_{N \to \infty} (1/n\ /2n) = \lim_{N \to \infty} (1/2n^2) = 0$ g tiene mayor vel. de crecimiento

5- Fórmulas útiles

Las siguientes fòrmulas se pueden aplicar a sumatorias

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

$$\sum_{i=1}^{n-1} 5 = 5(n-1) = 5n - 5$$

$$\sum_{i=1}^{n} i^2 = \frac{(n)(n+1)(2n+1)}{6}$$

$$\sum_{i=1}^{n} f(n) = n\, f(n)$$

$$\sum_{i=a}^{b} i = \frac{(b)(b+1) - (a)(a-1)}{2}$$

$$\sum_{i=n_0}^{n} f(i) = \sum_{i=1}^{n} f(i) - \sum_{i=1}^{n_0-1} f(i)$$