NOTACIÓN O GRANDE

- El análisis de algoritmos estima el consumo de recursos de un algoritmo.

- Esto nos permite comparar los costos relativos de dos o más algoritmos para resolver el mismo problema.

- El análisis de algoritmos también les da una herramienta a los diseñadores de algoritmos para estimar si una solución propuesta es probable que satisfaga las restricciones de recursos de un problema.

- El concepto de **razón de crecimiento**, es la razón a la cual el costo de un algoritmo crece conforme el tamaño de la entrada crece.

Introducción

- ¿Cómo comparar dos algoritmos para resolver un mismo problema en términos de eficiencia?

- El análisis de algoritmos mide la eficiencia de un algoritmo, conforme crece el tamaño de la entrada.

- Usualmente se mide el tiempo de ejecución de un algoritmo, y el almacenamiento primario y secundario que consume.

- De consideración principal para estimar el desempeño de un algoritmo, es el número de operaciones básicas requeridas por el algoritmo para procesar una entrada de cierto tamaño.

- Ejemplo: Algoritmo de búsqueda secuencial del máximo. $T(n) = cn$ (donde c es el tiempo que lleva examinar una variable).

```
int maximo (int* arreglo, int n)
{ int mayor=0;
   for(int i = 0; i < n; i++)
     if(arreglo[i] > mayor)
       mayor = arreglo[i];
     return mayor;
}
```

- Ejemplo: el tiempo requerido para copiar la primera posición de un arreglo es siempre $c1$ (independientemente de n). Así $T(n) = c1$.

- Otro ejemplo :

```
Sum=0 ;
for(i=0 ; i < n ; i++)
  for(j=0; j < n; j++)
    sum++;
```

¿Cuál es el tiempo de ejecución de este fragmento de código?

$$T(n) = c2\ n^2$$

(*c2* es el tiempo en incrementar una variable).

- El concepto de **razón de crecimiento** es extremadamente importante. Nos permite comparar el tiempo de ejecución de dos algoritmos sin realmente escribir dos programas y ejecutarlas en la misma máquina.

- Una razón de crecimiento de *cn* se le llama a menudo razón de *crecimiento lineal*.

- Si la razón de crecimiento tiene el factor n^2, se dice que tiene una *razón de crecimiento cuadrático*.

- Si el tiempo es del orden $2n$ se dice que tiene una razón de *crecimiento exponencial*.

notemos que $2^n > 2n^2 > \log n$

también para toda

$a, b > 1, n^a > (\log n)^b$ y $n^a > \log n^b$

para toda $a, b > 1, a^n > n^b$

Mejor, peor y caso promedio

- Para algunos algoritmos, diferentes entradas (inputs) para un tamaño dado pueden requerir diferentes cantidades de tiempo.

- Por ejemplo, consideremos el problema de encontrar la posición particular de un valor K, dentro de un arreglo de n elementos. (suponiendo que sólo ocurre una vez). Comentar sobre el mejor, peor y caso promedio.

- ¿Cuál es la ventaja de analizar cada caso? Si examinamos el peor de los casos, sabemos que al menos el algoritmo se desempeñara de esa forma.

- En cambio, cuando un algoritmo se ejecuta muchas veces en muchos tipos de entrada, estamos interesados en el

comportamiento promedio o típico. Desafortunadamente, esto supone que sabemos cómo están distribuidos los datos.

- Si conocemos la distribución de los datos, podemos sacar provecho de esto, para un mejor análisis y diseño del algoritmo. Por otra parte, si no conocemos la distribución, entonces lo mejor es considerar el peor de los casos.

¿Una computadora más rápida o un algoritmo más rápido?

- Si compramos una computadora diez veces más rápida, ¿en qué tiempo podremos ahora ejecutar un algoritmo?
- La respuesta depende del tamaño de la entrada de datos, así como en la razón de crecimiento del algoritmo.
- Si la razón de crecimiento es lineal (como $T(n)=cn$) entonces por ejemplo, 100,000 números serán procesados en la nueva máquina en el mismo tiempo que 10,000 números en la antigua computadora.
- ¿De que tamaño (valor de n) es el problema que podemos resolver con una computadora X veces más rápida (en un intervalo de tiempo fijo)?
- Por ejemplo, supongamos que una computadora resuelve un problema de tamaño n en una hora. Ahora supongamos que tenemos una

computadora 10 veces más rápida, ¿de que tamaño es el problema que podemos resolver?

$f(n) = 10{,}000$

f(n)	n (para 10,000 op)	n´ (para 100,000 op)	cambio	n´/n
10n	1,000	10,000	n´=10n	10
20n	500	5,000	n´=10n	10
5n log n	250	1842	$\sqrt{(10)}n < n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{(10)}n$	3.16
2^n	13	16	n´=n+3	----

- En la tabla de arriba, *f(n)* es la razón de crecimiento de un algoritmo.
- Supongamos que tenemos una computadora que puede ejecutar 10,000 operaciones básicas en una hora.
- La segunda columna muestra el máximo valor de *n* que puede ejecutarse con 10,000 operaciones básicas en una hora.

- Es decir *f(n)=total* de operaciones básicas en un intervalo de tiempo
- Por ejemplo, f(n)=10,000 operaciones b. por hr.
- Si suponemos que tenemos una computadora 10 veces más rápida, entonces podremos ejecutar 100,000 operaciones básicas en una hora.
- Por lo cual *f(n´)=100,000* op. bas. por hr.
- Observar la relación *n´/n* según el incremento de velocidad de la computadora y la razón de crecimiento del algoritmo en cuestión.
- Nota: los factores constantes nunca afectan la mejora relativa obtenida por una computadora más rápida.

Análisis Asintótico

- Comparemos la razón de crecimiento entre *10n*, *20n*, y $2n^2$, notaremos que $2n^2$, supera eventualmente a *10n* y *20n*, difiriendo solamente en un valor mayor de *n*, donde ocurre el corte.

- Por las razones anteriores, usualmente se ignoran las constantes cuando queremos una estimación del tiempo de ejecución u otros requerimientos de recursos del algoritmo. Esto simplifica el análisis y nos mantiene pensando en el aspecto más importante: *la razón de crecimiento*. A esto se le llama **análisis asintótico del algoritmo**.

- En términos más precisos, el análisis asintótico se refiere al estudio de un algoritmo conforme el tamaño de entrada "se **vuelve grande**" o alcanza un límite (en el sentido del cálculo).

- Sin embargo, no siempre es razonable ignorar las constantes, cuando se compara algoritmos que van a ejecutar en valores relativamente pequeños de *n*.

Cotas superiores

- La **cota superior de un algoritmo**, indica una cota o la máxima razón de crecimiento que un algoritmo puede tener. Generalmente hay que especificar si es para el mejor, peor o caso promedio.

- Por ejemplo, podemos decir: "este algoritmo tiene una cota superior a su razón de crecimiento de n^2 en el caso promedio".

- Se adopta una **notación** especial llamada **O-grande** (big-Oh), por ejemplo ***O(f(n))*** para indicar que la cota superior del algoritmo es $f(n)$.

- En términos precisos, si $T(n)$ representa el tiempo de ejecución de un algoritmo, y $f(n)$ es alguna expresión para su cota superior, $T(n)$ está en el conjunto $O(f(n))$, si existen dos constantes positivas c y n_0 tales que $|T(n)| \leq c\,|f(n)|$ para todo $n > n_0$

- Ejemplo: Consideremos el algoritmo de búsqueda secuencial para encontrar un valor especificado en un arreglo. Si el visitar y comparar contra un valor en el arreglo, requiere c_s pasos, entonces en el caso promedio $T(n)=c_s\, n/2$. Para todos los valores $n>1$

  $|c_s\ n/2| \leq c_s|n|$. Por lo tanto, por definición, $T(n)$ está en $O(n)$ para $n_0=1$, y $c=c_s$.

- El sólo saber que algo es $O(f(n))$ sólo nos dice que tan mal se pueden poner las cosas. Quizás la situación no es tan mala. De la definición podemos ver que si $T(n)$ está en $O(n)$, también está en $O(n^2)$ y $O(n^3)$, etc.
- Por lo cual se trata en general de definir la mínima cota superior.

Cota inferior

- Existe una **notación** similar para indicar la mínima cantidad de recursos que un algoritmo necesita para alguna clase de entrada. La **cota inferior de un algoritmo**, denotada por el símbolo Ω, pronunciado "**Gran Omega**" u "**Omega**", tiene la siguiente definición:

- $T(n)$ está en el conjunto $\Omega(g(n))$, si existen dos constantes positivas c y n_0 tales que $|T(n)| \geq c|g(n)|$ para todo $n > n_0$.

- Ejemplo: Si $T(n)=c1n2+c2n$ para $c1$ y $c2 > 0$, entonces:

$$|c_1n^2+c_2n| \geq |c_1\ n^2| \geq \ c_1|n^2|$$

Por lo tanto, $T(n)$ está en $\Omega(n^2)$.

Notación Θ

- Cuando las cotas superior e inferior son la misma, indicamos esto usando la **notación** Θ (big-Theta). Se dice que una algoritmo es $\Theta(h(n))$, si está en $O(h(n))$ y está en $\Omega(h(n))$.

- Por ejemplo, como un algoritmo de búsqueda secuencial está tanto en $O(n)$, como en $\Omega(n)$ en el caso promedio, decimos que es $\Theta(n)$ en el caso promedio.

- Dada una expresión aritmética que describe los requerimientos de tiempo para un algoritmo, las cotas inferior y superior siempre coinciden. En general se usa la **notación** O y Ω , cuando no conocemos exactamente, sino sólo acotado de un algoritmo.

Reglas de simplificación

Una vez que se determina la ecuación del tiempo de ejecución para un algoritmo, es relativamente sencillo derivar las expresiones para: **O-grande**, Ω y Θ.

Existen algunas reglas sencillas que nos permiten simplicar las expresiones:

1. Si *f(n)* está en *O(g(n))* y *g(n)* está en *O(h(n))*, entonces *f(n)* está en *O(h(n))*.

Esta regla nos dice que si alguna función *g(n)* es una cota superior para una función de costo, entonces cualquier cota superior para *g(n)*, también es una cota superior para la función de costo.

Nota: Hay una propiedad similar para la **notación** Ω y Θ.

2. Si $f(n)$ está en $O(k\ g(n))$ para cualquier constante $k>0$, entonces $f(n)$ está $O(g(n))$

El significado de la regla es que se puede ignorar cualquier constante multiplicativa en las ecuaciones, cuando se use **notación** de **O-grande**.

3. Si $f_1(n)$ está en $O(g_1(n))$ y $f_2(n)$ está en $O(g_2(n))$, entonces $f_1(n) + f_2(n)$ está en $O(max(g_1(n),g_2(n)))$.

La regla expresa que dadas dos partes de un programa ejecutadas en secuencia, sólo se necesita considerar la parte más cara.

4. Si $f_1(n)$ está en $O(g_1(n))$ y $f_2(n)$ está en $O(g_2(n))$, entonces $f_1(n)f_2(n)$ está en $O(g_1(n)g_2(n))$.

Esta regla se emplea para simplificar ciclos simples en programas. Si alguna

acción es repetida un cierto número de veces, y cada repetición tiene el mismo costo, entonces el costo total es el costo de la acción multiplicado por el número de veces que la acción tuvo lugar.

Tomando las tres primeras reglas colectivamente, se pueden ignorar todas las constantes y todos los términos de orden inferior para determinar la razón de crecimiento asintótico para cualquier función de costo, ya que los términos de orden superior pronto superan a los términos de orden inferior en su contribución en el costo total, conforme *n* se **vuelve grande**.

Ejemplos de cálculo del tiempo de ejecución de un programa

Veamos el análisis de un simple enunciado de asignación a una variable entera:

```
a = b;
```

Como el enunciado de asignación toma tiempo constante, está en $\Theta(1)$.

Consideremos un simple ciclo "for" :

```
sum=0;
for(i=0; i<n; i++)
    sum += n;
```

La primera línea es $\Theta(1)$. El ciclo "for" es repetido *n* veces. La tercera línea toma un tiempo constante también, por la regla de simplificación (4), el costo total por ejecutar las dos líneas que forman el ciclo "for" es $\Theta(n)$. Por la regla (3), el costo por el entero fragmento de código es también $\Theta(n)$.

Analicemos un fragmento de código con varios ciclos "for", algunos de los cuales están anidados.

```
sum=0;
for(j=0; j<n; j++)
    for(i=0; i<j; i++)
        sum++;
for(k=0; k<n; k++)
    A[k]= k-1;
```

Este código tiene tres partes: una asignación, y dos ciclos.

La asignación toma tiempo constante, llamémosla c_1. El segundo ciclo es similar al ejemplo anterior y toma $c_2n = \Theta(n)$.

Analicemos ahora el primer ciclo, que es un doble ciclo anidado. En este caso trabajemos de adentro hacia fuera.

La expresión *sum++* requiere tiempo constante, llamémosle c_3.

Como el ciclo interno es ejecutado j veces, por la regla (4), tiene un cost de c_3j. El ciclo exterior es ejecutado n veces, pero

cada vez el costo del ciclo interior es diferente.

El costo total del ciclo es c_3 veces la suma de los números 1 a n, es decir

$\Sigma^{n}_{i=1} j = n(n+1)/2$, que es $\Theta(n^2)$. Por la regla (3), $\Theta(c_1+c_2n+c_3n^2)$ es simplemente $\Theta(n^2)$.

Comparemos el análisis asintótico de los siguientes fragmentos de código:

```
sum1=0;
for(i=0; i<n; i++)
  for(j=0; j<n; j++)
    sum1++;

sum2=0;
for(i=0; i<n; i++)
  for(j=0; j<i; j++)
    sum2++;
```

El primer fragmento de ejecuta el enunciado *sum1++*, precisamente n^2 veces.

Por otra parte, el segundo fragmento de código tiene un costo aproximado de $\frac{1}{2} n^2$. Así ambos códigos tienen un costo de $\Theta(n^2)$.

Ejemplo, no todos los ciclos anidados son $\Theta(n^2)$:

```
sum1=0;
for(k=0; k<n; k*=2)
  for(j=0; j<n; j++)
       sum1++;
```