Introducción al análisis de algoritmos

Dr. Luis Carlos González Gurrola

lcgonzalez@uach.mx

http://www.gonzalezgurrola.com

¿Por qué analizar un algoritmo

¿Por qué analizar un algoritmo

- Para descubrir características que nos permitan evaluar su aplicación o compararlo con otros algoritmos
  - Tiempos
  - Espacio

- Para decidir cuál entorno de programación es más favorecedor

- Puede pasar que el tiempo y espacio, no sean de nuestro interés, e.g. Cómputo móviles, pero si el uso de la batería

Análisis de algoritmos

Este término ha sido usado para describir 2 ideas:

1. Determinar el orden de crecimiento del desempeño para el peor caso (Aho, Hopcroft, et al.)
2. Caracterizar el mejor caso, caso promedio y peor caso (Knuth)

¿Por qué analizar un algoritmo

- El análisis de algoritmos nos ayuda a entenderlo mejor, y puede sugerir mejoras

- No es inusual que un algoritmo pase por una transformación que lo haga más sencillo y más elegante durante el proceso de análisis

Teoría de algoritmos

El objetivo primario de la teoría de algoritmos es clasificar a los algoritmos de acuerdo a sus características de desempeño

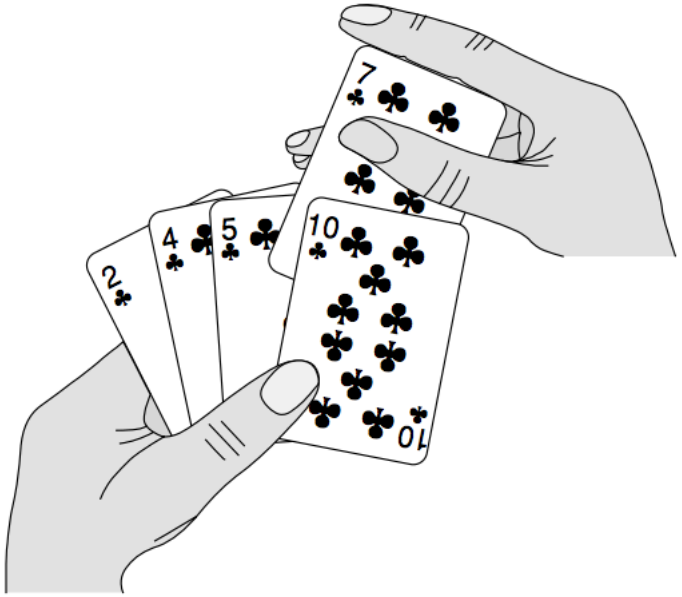
Para lograr este propósito, las siguientes nociones matemáticas son necesarias

$O(f(N))$ denotes the set of all $g(N)$ such that $|g(N)/f(N)|$ is bounded from above as $N \to \infty$.

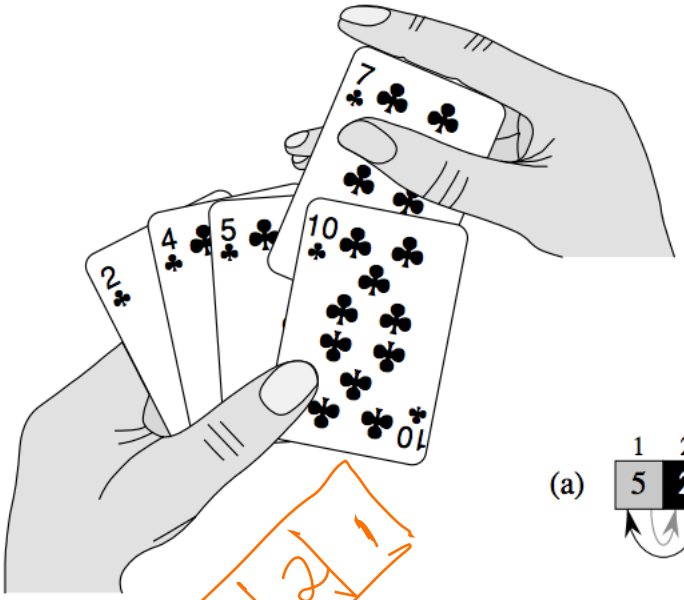
$\Omega(f(N))$ denotes the set of all $g(N)$ such that $|g(N)/f(N)|$ is bounded from below by a (strictly) positive number as $N \to \infty$.

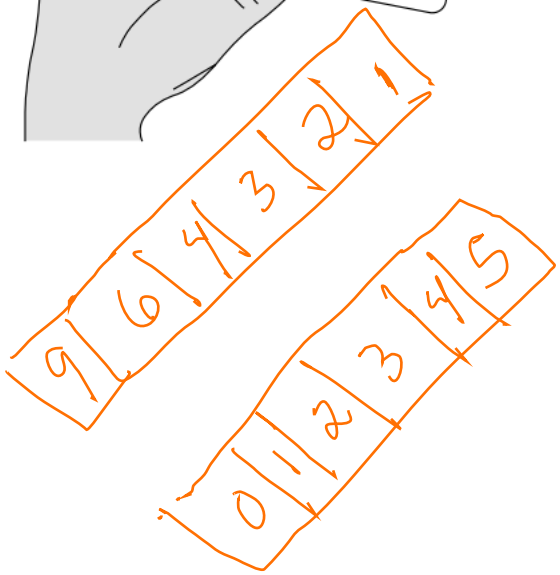
$\Theta(f(N))$ denotes the set of all $g(N)$ such that $|g(N)/f(N)|$ is bounded from both above and below as $N \to \infty$.

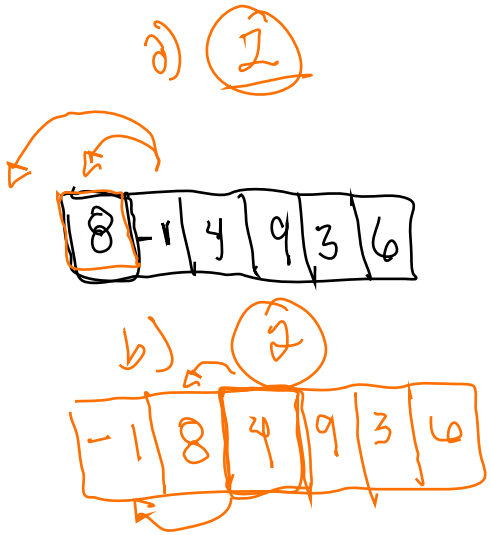
Insertion Sort

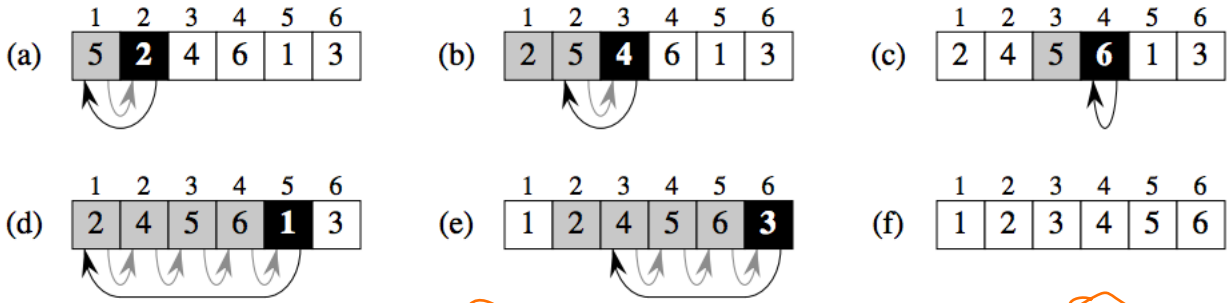


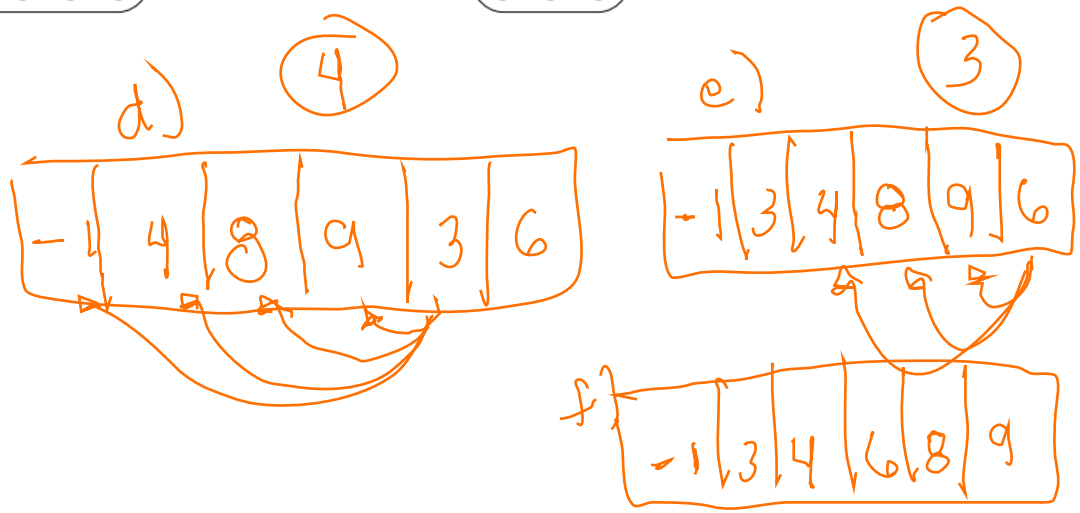
Insertion Sort











```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2       do key ← A[j]
3          ▷ Insert A[j] into the sorted sequence A[1 . . j − 1].
4          i ← j − 1
5          while i > 0 and A[i] > key
6               do A[i + 1] ← A[i]
7                  i ← i − 1
8          A[i + 1] ← key
```

¿Podemos caracterizar el tiempo de ejecución de este algoritmo?

Insertion Sort

El tiempo de ejecución de un algoritmo sobre una entrada en particular es el número de operaciones primitivas o “pasos” ejecutados

Es conveniente definir la noción de paso de tal manera que sea independiente de la máquina en la cual el algoritmo se ejecute

Un tiempo de ejecución constante se requiere para ejecutar cada línea del pseudo código

Insertion Sort

Una línea puede tomar un monto diferente de tiempo que otra línea, pero supondremos que cada ejecución de la línea i toma el tiempo ci, donde ci es constante

Insertion Sort

	INSERTION-SORT(A)	*cost*	*times*
1	**for** $j \leftarrow 2$ **to** *length*$[A]$	c_1	n
2	**do** *key* $\leftarrow A[j]$	c_2	$n-1$
3	$\triangleright$ Insert $A[j]$ into the sorted sequence $A[1 \,..\, j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] >$ *key*	c_5	$\sum_{j=2}^{n} t_j$
6	**do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	$A[i+1] \leftarrow$ *key*	c_8	$n-1$

Insertion Sort

INSERTION-SORT(A)	*cost*	*times*
1 **for** $j \leftarrow 2$ **to** *length*$[A]$	c_1	n
2 **do** *key* $\leftarrow A[j]$	c_2	$n-1$
3 ▷ Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
4 $i \leftarrow j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] >$ *key*	c_5	$\sum_{j=2}^{n} t_j$
6 **do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7 $i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8 $A[i+1] \leftarrow$ *key*	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n} (t_j - 1)$$
$$+ c_7 \sum_{j=2}^{n} (t_j - 1) + c_8(n-1) \,.$$

Mejor caso

Si todos los elementos están totalmente ordenados, el paso 5, A[i]<= key siempre, por lo tanto el renglón 5 sólo se ejecutará 1 vez para cada valor de j

INSERTION-SORT(A)		*cost*	*times*
1	**for** $j \leftarrow 2$ **to** *length*$[A]$	c_1	n
2	**do** *key* $\leftarrow A[j]$	c_2	$n-1$
3	$\triangleright$ Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] >$ *key*	c_5	$\sum_{j=2}^{n} t_j$
6	**do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n}(t_j-1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n}(t_j-1)$
8	$A[i+1] \leftarrow$ *key*	c_8	$n-1$

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\
&= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)\,.
\end{aligned}
$$

Mejor caso

INSERTION-SORT(A)		*cost*	*times*
1	**for** $j \leftarrow 2$ **to** *length*$[A]$	c_1	n
2	**do** *key* $\leftarrow A[j]$	c_2	$n-1$
3	$\triangleright$ Insert $A[j]$ into the sorted sequence $A[1\,..\,j-1]$.	0	$n-1$
4	$i \leftarrow j-1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] >$ *key*	c_5	$\sum_{j=2}^{n} t_j$
6	**do** $A[i+1] \leftarrow A[i]$	c_6	$\sum_{j=2}^{n} (t_j - 1)$
7	$i \leftarrow i-1$	c_7	$\sum_{j=2}^{n} (t_j - 1)$
8	$A[i+1] \leftarrow$ *key*	c_8	$n-1$

$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\
&= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \,.
\end{aligned}
$$

Este tiempo de ejecución se puede expresar como *an+b* para constantes a y b que dependen del costo de las instrucciones c_i; por lo tanto es una función lineal

Peor caso

Pero, ¿qué característica tendría que tener la entrada para el peor tiempo de ejecución posible?

Si el arreglo se encuentra en orden inverso, forma decreciente, cada elemento A[i] se tendría que comparar con el subarreglo ordenado A[*i*…*j*-1], así que $t_j = j$ para j=2…n, y obtendríamos:

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

and

$$\sum_{j=2}^{n} (j-1) = \frac{n(n-1)}{2}$$

Peor caso

En ese caso, el tiempo de ejecución

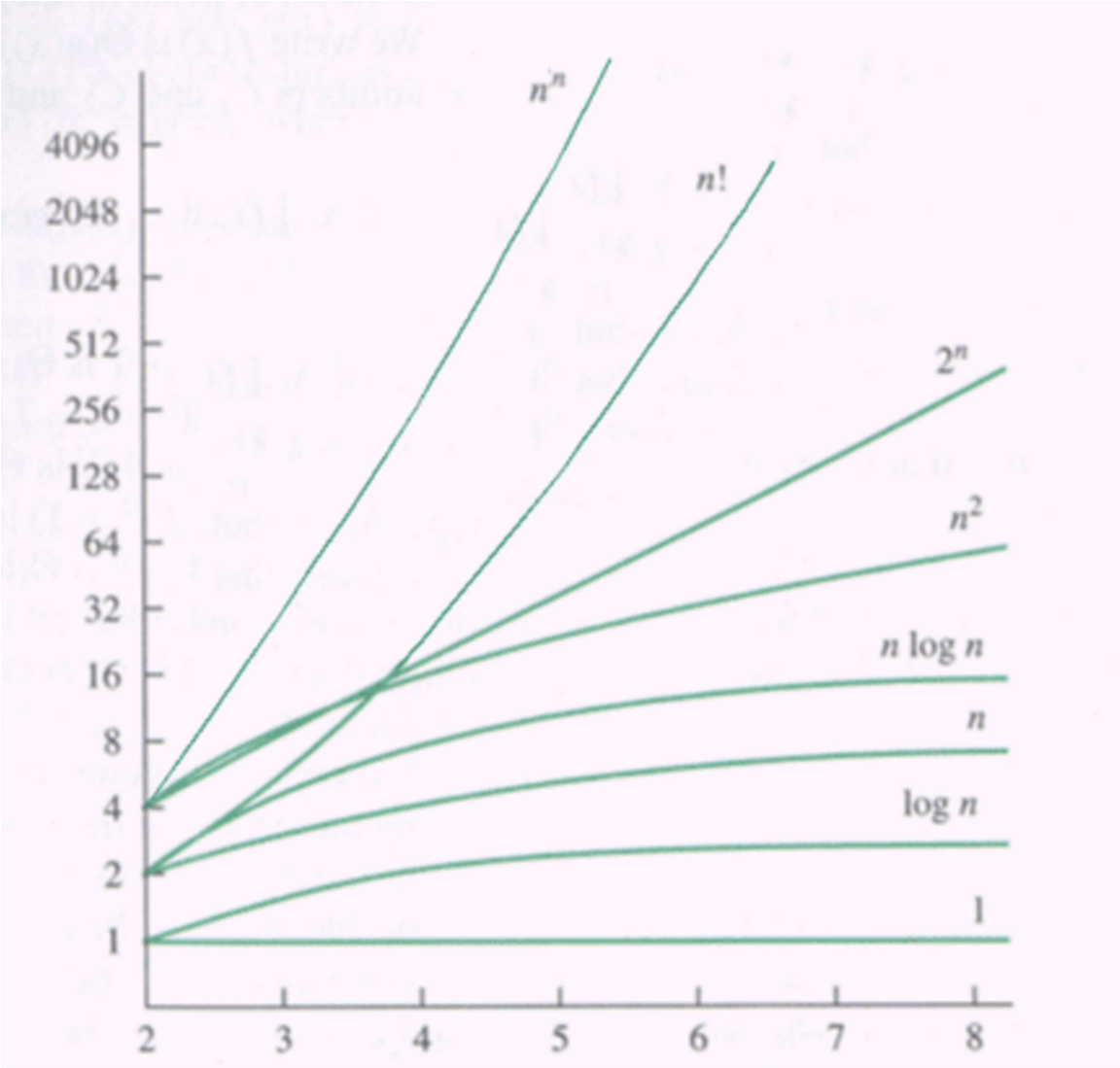
$$
\begin{aligned}
T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\
&\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\
&= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right) n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right) n \\
&\quad - (c_2 + c_4 + c_5 + c_8) \, .
\end{aligned}
$$

Este tiempo de ejecución se puede expresar como an^2+bn+c para constantes a,b y c que dependen del costo de las instrucciones c_i; por lo tanto es una función cuadrática!

Crecimiento de funciones

El orden de crecimiento del tiempo de ejecución de un algoritmo permite una caracterización de la eficiencia del algoritmo y también permite comparar su desempeño contra otros algoritmos

T
I
E
M
P
O



Tamaño del vector de entrada

Notación asintótica

Una vez que el tamaño de la entrada es lo suficientemente grande, el algoritmo merge sort, con su peor caso θ(n lg n) le gana al insertion sort, cuyo peor caso es θ (n^2)

Para entradas muy grandes, las constantes multiplicativas y términos de bajo orden son dominados por los efectos del tamaño de la entrada!

Cuando analizamos tamaños de la entrada lo suficientemente grande para hacer que sólo el orden de crecimiento del tiempo de ejecución sea relevante, estamos estudiando la eficiencia **asintótica** de los algoritmos

Notación asintótica

Es decir, estamos interesados en cómo el tiempo de ejecución de un algoritmo se incrementa en el límite

Usualmente, un algoritmo que es asintóticamente más eficiente será la mejor opción, excepto para entradas extremadamente pequeñas

Notación θ

Previamente, habíamos dicho que el peor tiempo de ejecución del Insertion Sort era $T(n)=\theta(n^2)$, pero, ¿Qué significa esta notación?

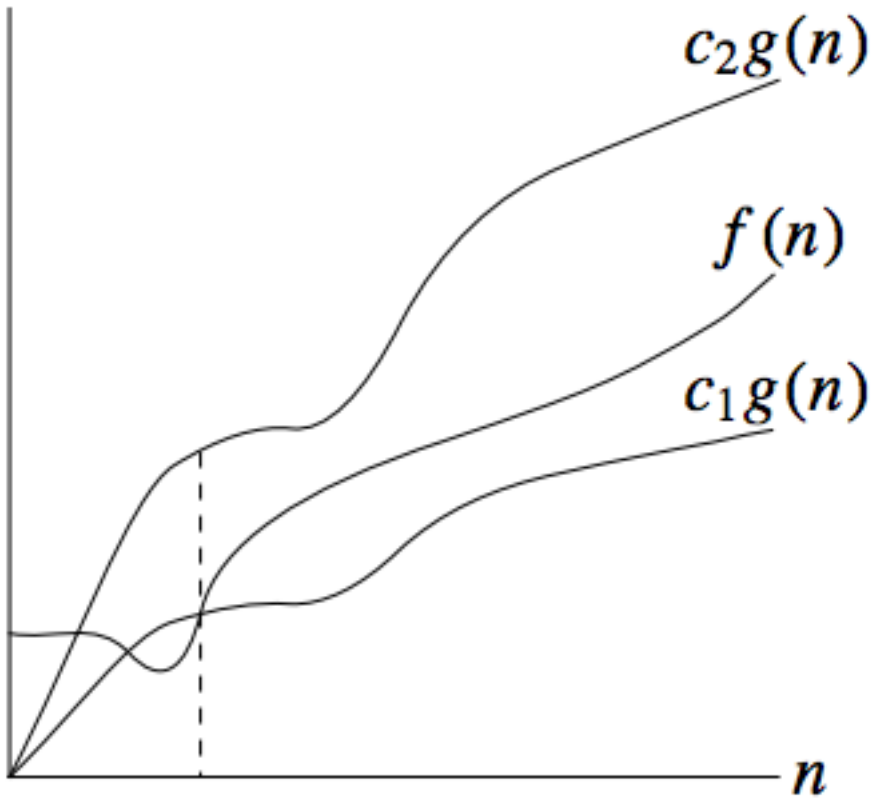
Para una función g(n), denotamos $\theta(g(n))$ como el conjunto de funciones

$\theta(g(n))=\{f(n)$: existen constantes enteras positivas c_1, c_2 y n_0, tales que $0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ para todo $n \geq n_0\}$

Notación θ

Una función f(n) pertenece al conjunto θ (g(n)) si las constantes positivas c_1 y c_2 la encierran (o hacen sandwich o torta) entre $c_1 g(n)$ y $c_2 g(n)$, para una *n* suficientemente grande

Aunque θ (g(n)) es un conjunto, escribimos "f(n)= θ(g(n))" para indicar que f(n) es miembro de θ(g(n))



$$f(n) = \Theta(g(n))$$

Ejemplo

$$\frac{1}{2}n^2 - 3n = \theta(n^2)?$$

Ejemplo

$$\frac{1}{2}n^2 - 3n = \theta(n^2)?$$

Para demostrarlo debemos encontrar constantes positivas c_1, c_2 y n_0 tales que

$$c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$$

Para toda $n \geq n_0$

Ejemplo

Dividiendo por n^2 tenemos

$$c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$$

La parte derecha de la desigualdad se mantiene para cualquier valor de $n \geq 1$ al elegir $c_2 \geq ½$

De la misma manera, el lado izquierdo de la desigualdad se puede mantener para cualquier valor de $n \geq 7$ al elegir $c_1 \leq 1/14$

Así, al elegir $c_1=1/14$, $c_2=1/2$ y $n_0 =7$ podemos comprobar que

$$\frac{1}{2}n^2 - 3n = \theta(n^2)$$

Ciertamente, otras opciones de valores existen, pero lo importante es que existen valores

Note que estas constantes dependen de la función f(n); una función diferente que pertenezca a θ (n^2) usualmente requerirá diferentes constantes

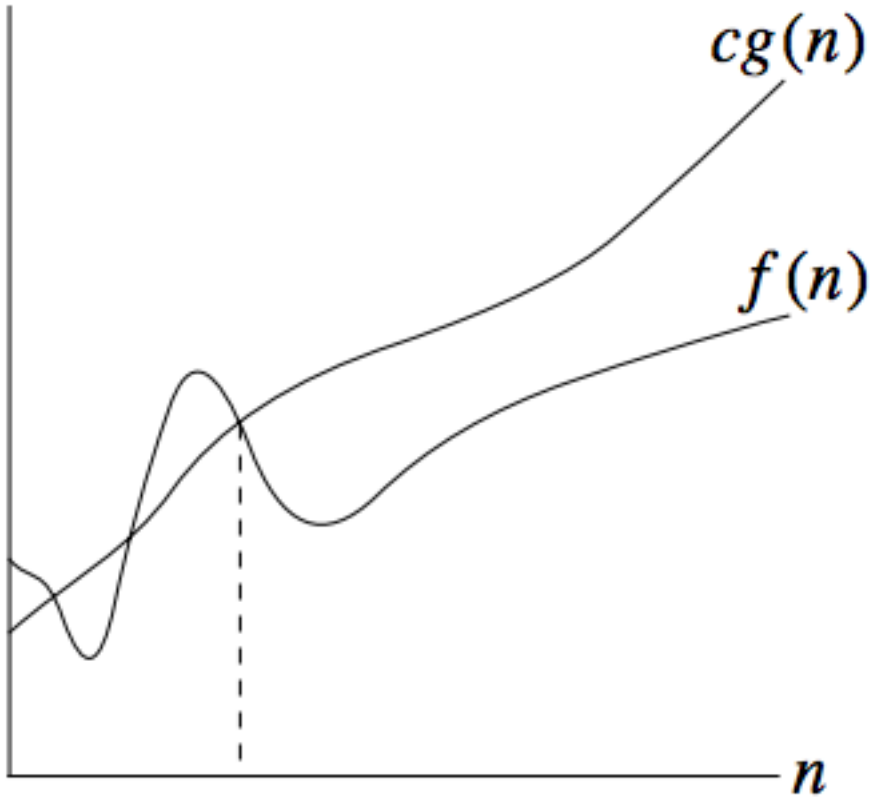
Notación O

La notación θ encierra asintóticamente una función por arriba y por abajo

Cuando únicamente tenemos la cota superior usamos la notación O (O mayúscula). Para una función dada g(n), denotamos por =(g(n)) el conjunto de funciones

O(g(n))={f(n): existe constantes positiva c y n_0 tales que

$0 \leq f(n) \leq cg(n)$ para toda $n \geq n_0$}



$f(n) = O(g(n))$

Algoritmo Mergesort

- Divide el arreglo a la mitad, ordena las 2 mitades (en forma recursiva), y una las mitades ordenadas

- Es un algoritmo prototipo del paradigma "**dividir y vencer**", en donde el problema se resuelve (recursivamente) al resolver pequeños subproblemas y usando esas soluciones para solucionar el problema original

Video extraído de https://en.wikipedia.org/wiki/Merge_sort

```
private void mergesort(int[] a, int lo, int hi)
{
   if (hi <= lo) return;
   int mid = lo + (hi - lo) / 2;
   mergesort(a, lo, mid);
   mergesort(a, mid + 1, hi);
   for (int k = lo; k <= mid; k++)
      b[k-lo] = a[k];
   for (int k = mid+1; k <= hi; k++)
      c[k-mid-1] = a[k];
   b[mid-lo+1] = INFTY; c[hi - mid] = INFTY;
   int i = 0, j = 0;
   for (int k = lo; k <= hi; k++)
      if (c[j] < b[i]) a[k] = c[j++];
      else             a[k] = b[i++];
}
```

Program 1.1 Mergesort

```
private void mergesort(int[] a, int lo, int hi)
{
   if (hi <= lo) return;
   int mid = lo + (hi - lo) / 2;
   mergesort(a, lo, mid);
   mergesort(a, mid + 1, hi);
   for (int k = lo; k <= mid; k++)
      b[k-lo] = a[k];
   for (int k = mid+1; k <= hi; k++)
      c[k-mid-1] = a[k];
   b[mid-lo+1] = INFTY; c[hi - mid] = INFTY;
   int i = 0, j = 0;
   for (int k = lo; k <= hi; k++)
      if (c[j] < b[i]) a[k] = c[j++];
      else             a[k] = b[i++];
}
```

Usa 2 arreglos auxiliares b y c para guardar los sub arreglos

Program 1.1 Mergesort

```
private void mergesort(int[] a, int lo, int hi)
{
   if (hi <= lo) return;
   int mid = lo + (hi - lo) / 2;
   mergesort(a, lo, mid);
   mergesort(a, mid + 1, hi);
   for (int k = lo; k <= mid; k++)
      b[k-lo] = a[k];
   for (int k = mid+1; k <= hi; k++)
      c[k-mid-1] = a[k];
   b[mid-lo+1] = INFTY; c[hi - mid] = INFTY;
   int i = 0, j = 0;
   for (int k = lo; k <= hi; k++)
      if (c[j] < b[i]) a[k] = c[j++];
      else             a[k] = b[i++];
}
```

Se usa un valor "centinela" que se supone es más grande que cualquiera de los elementos de los arreglos

Program 1.1 Mergesort

```
private void mergesort(int[] a, int lo, int hi)
{
   if (hi <= lo) return;
   int mid = lo + (hi - lo) / 2;
   mergesort(a, lo, mid);
   mergesort(a, mid + 1, hi);
   for (int k = lo; k <= mid; k++)
      b[k-lo] = a[k];
   for (int k = mid+1; k <= hi; k++)
      c[k-mid-1] = a[k];
   b[mid-lo+1] = INFTY; c[hi - mid] = INFTY;
   int i = 0, j = 0;
   for (int k = lo; k <= hi; k++)
      if (c[j] < b[i]) a[k] = c[j++];
      else             a[k] = b[i++];
}
```

La unión (merge) se logra, para cada k, mover el más pequeño de los elementos b[i] y c[j], después se incrementa k e i ó j

Program 1.1 Mergesort