2.2.4.3 Valores frontera en algoritmos

- Generalmente es importante determinar los valores frontera de un algoritmo, para ello se utilizan 3 valores

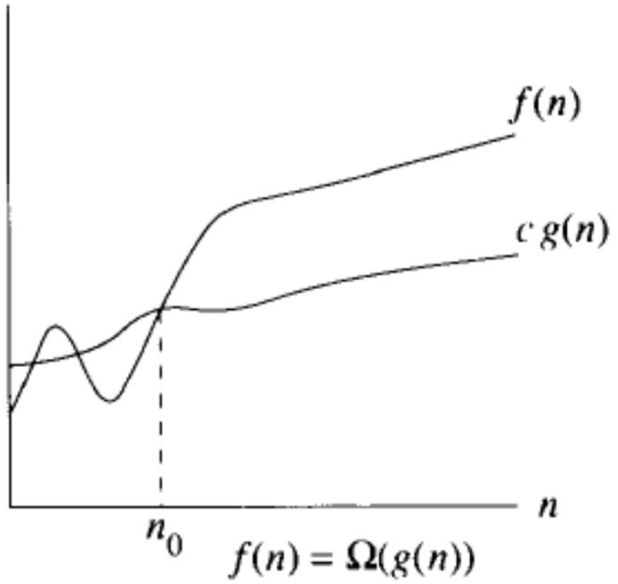
✓El **peor caso** es aquel ejemplar que exige más recursos para ser resuelto

✓El **caso promedio** representa el tiempo aproximado para un ejemplar “típico”. Se calcula la media del tiempo para todos los posibles ejemplares de tamaño n asumiendo distribución uniforme.

✓El **mejor caso** es aquella instancia que requiere menos recursos

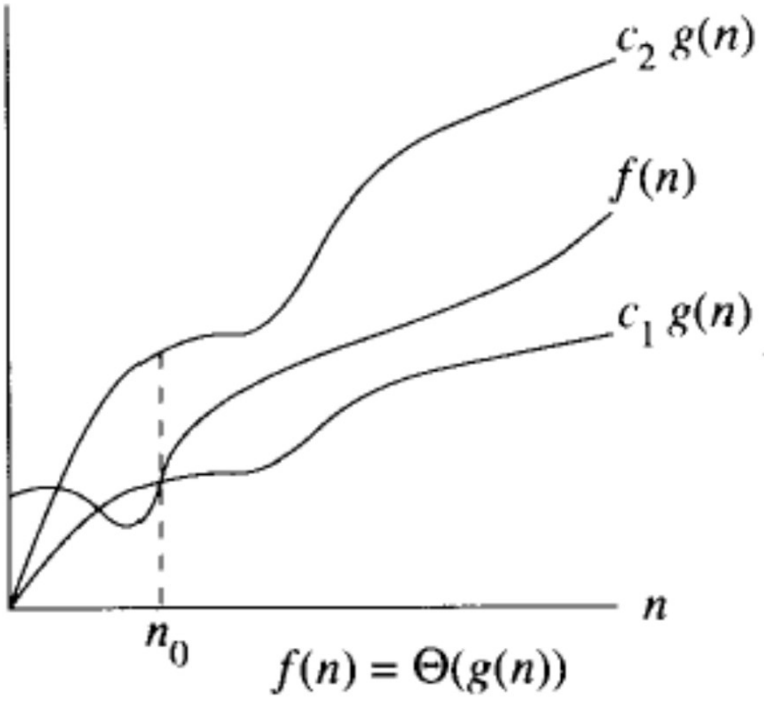
2.2.4.4 Notación Ω y Θ

- En ocasiones, sobre un algoritmo se tiene que precisar ***por lo menos*** una cierta cantidad de tiempo.
- La cota inferior se define con la notación Ω



2.2.4.4 Notación Ω y Θ

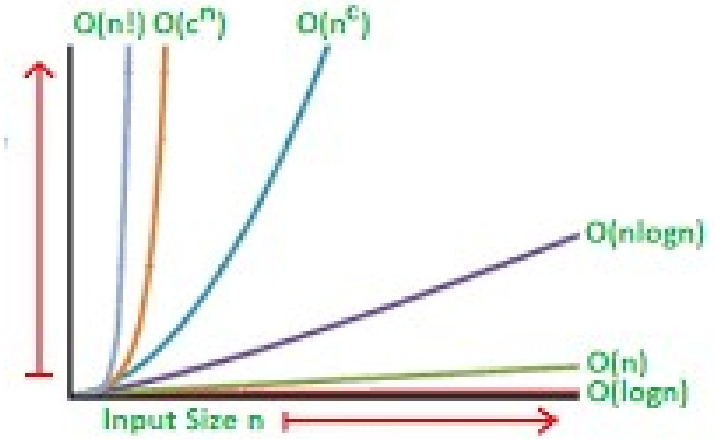
- Cuando se usa la notación ***θ*** grande, se tiene una **cota asintóticamente ajustada** sobre el tiempo de ejecución.
- "Asintóticamente" porque importa solo para valores grandes de ***n***.
- "Cota ajustada" porque ajustamos el tiempo de ejecución dentro del rango de una constante hacia arriba y hacia abajo.

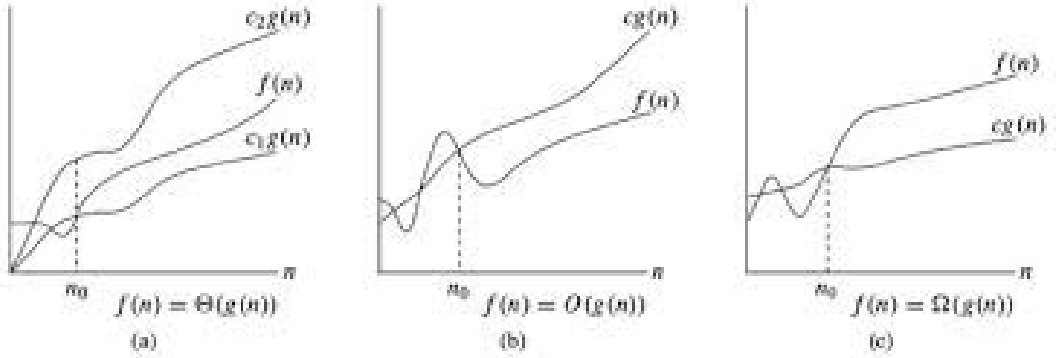


Complejidad Algoritmos de Ordenamiento

Algorithm	Time Complexity		
	Best	Average	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))
Bubble Sort	O(n)	O(n^2)	O(n^2)
Insertion Sort	O(n)	O(n^2)	O(n^2)
Select Sort	O(n^2)	O(n^2)	O(n^2)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)
Radix Sort	O(nk)	O(nk)	O(nk)

2.2.5 Ejemplos de análisis de complejidad





2.2.5 Ejemplos de análisis de complejidad

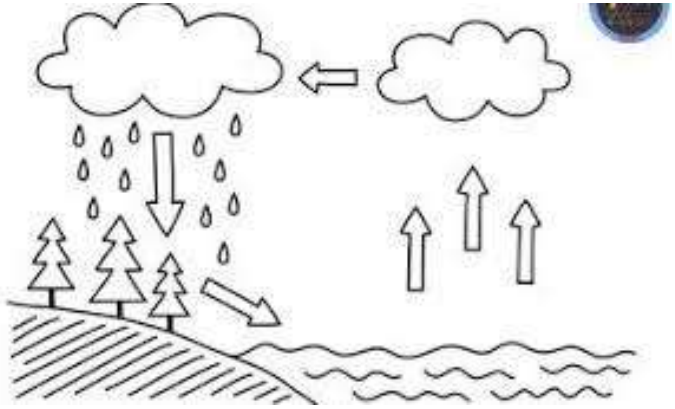
- **Instrucciones simples**

```
int main(){
    int a;
    a=5;
    a++;
    if (a==10)
        print (“hola”);
    else
        print (“adios);
}
```

“it’s easy if you try”

2.2.5 Ejemplos de análisis de complejidad

- **Ciclos de repetición “simples”**



```
int main(){
    printf(“ingresa a”)
    scanf(“%d”,a);
    a*=2;
    for(i=0;i<n;i++){
        //instrucciones simples
    }
}
```

*Se toma en cuenta el alcance del ciclo multiplicado por la cantidad de instrucciones que contenga

2.2.5 Ejemplos de análisis de complejidad

- **Ciclos de repetición “simples”**

```
for(int a=0, i=0 ; i<n ; i++){
    a+=2;
}
for(i=5 ; i<n-10 ; i++){
    //
}
for(i=0 ; i<10’000 ; i++){
    //
}
```

2.2.5 Ejemplos de análisis de complejidad

- **Ciclos anidados**

```
int main(){
    for(i=0;i<n;i++){
        for(int j=0; j<n; j++)
            //instrucciones simples
    }
}
```

2.2.5 Ejemplos de análisis de complejidad

- **Ciclos anidados**

```
int main(){
    for(i=0;i<n;i++){
        //instrucciones?
        for(int j=0; j<n; j++)
            //instrucciones simples
    }
}
```

2.2.5 Ejemplos de análisis de complejidad

- **Ciclos anidados**

```
for(int i=0 ; i<n ; i++){
    for(int j=n; j<i; j++){
        //instrucciones simples
    }
}

for(int i=0 ; i<n ; i++){
    for(int j=0; j<500; j++){
        //instrucciones simples
    }
}
```

2.2.5 Ejemplos de análisis de complejidad

```
//pseudocódigo
funcion ejemplo(n: entero):entero
empieza
   variables: a, j, k enteros
   para j desde 1 hasta n hacer
      a=a+j
   fin para
   para k desde n hasta 1 hacer
      a=a-1
      a=a*2
   fin para
   devolver a
termina
```

2.2.5 Ejemplos de análisis de complejidad

```
funcionZ(){
     int b;
     b=5;
     b++;
     for(i=0;i<n;i++){
          //Instrucciones simples
     }
     for(i=0;i<n;i++){
          for(j=0;j<n;j++){
               //instrucciones simples
          }
     }
}
```

2.2.5 Ejemplos de análisis de complejidad

```
funcionX(int x){
     if(x==7){
          for(i=0;i<n;i++){
               for(j=0;j<n;j++){
                    //instrucciones simples
               }
          }
     }
     else{
          for(i=0;i<n;i++){
               //instrucciones simples
          }
     }
}
```

2.2.5 Ejemplos de análisis de complejidad


```
int recursiveFun1(int n)
    {
        if (n <= 0)
        return 1;
    else
        return 1 + recursiveFun1(n-1);
    }
```

```
void recursiveFun4(int n, int m, int o)
{
    if (n <= 0)
    {
        printf("%d, %d\n",m, o);
    }
    else
    {
        recursiveFun4(n-1, m+1, o);
        recursiveFun4(n-1, m, o+1);
    }
}
```

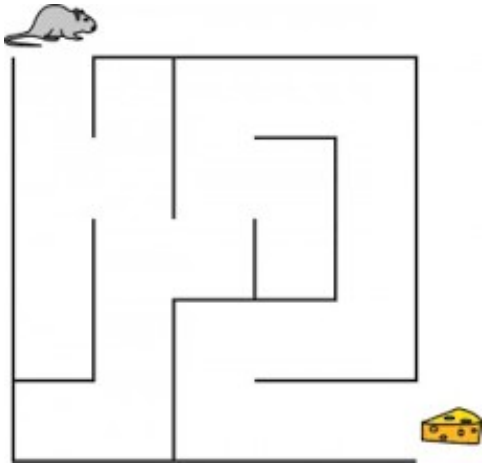
TAREA

Buscar en libros de algoritmos en el tema de análisis de complejidad:

- 5 ejemplos de códigos o algoritmos con su respectivo análisis de complejidad (notación BIG - O)
- Ejemplos diferentes a los vistos en clase

2.2.6 Medidas empíricas de rendimiento

- Se podría pensar que un algoritmo “simple” o con pocas instrucciones no es eficiente.
- Sin embargo, la simplicidad es una característica difícil de obtener en el momento en el que se diseña un algoritmo, pero que si se logra, facilita su verificación, el estudio de su complejidad y su mantenimiento.

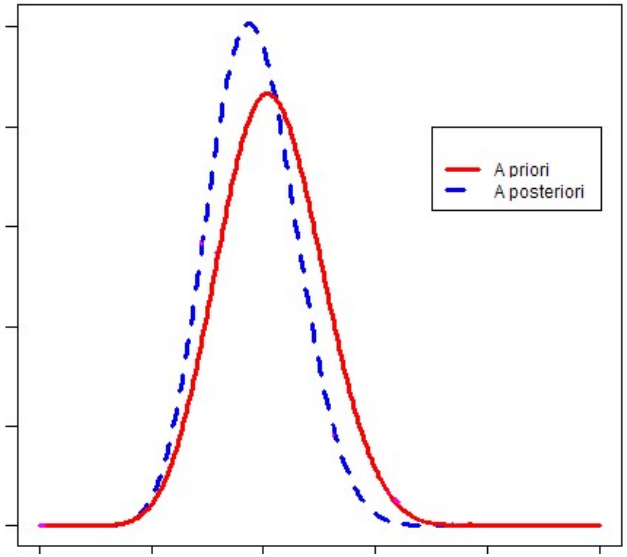


2.2.6 Medidas empíricas de rendimiento

• Las dos aproximaciones en cuanto a criterios de medición son:

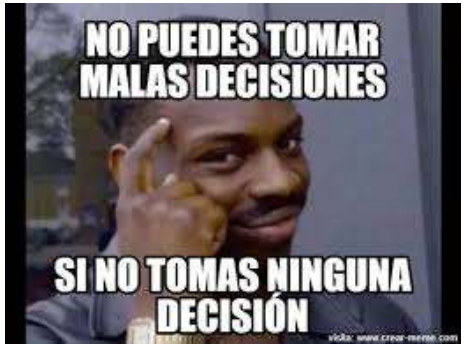
Una **medida teórica** (a priori) que consiste en obtener una función que acote el recurso que utiliza el algoritmo.

Una **medida real** (a posteriori), consistente en medir el recurso que utiliza el algoritmo para ciertos valores de entrada dados en un dispositivo físico.



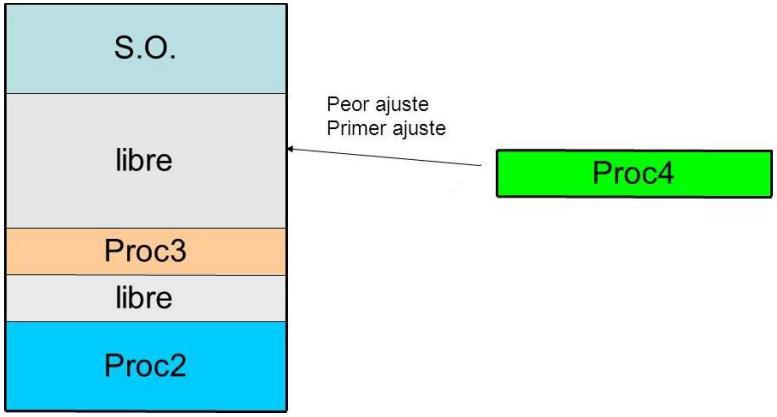
2.2.7 Compromiso espacio-tiempo

- El compromiso espacio-tiempo es una situación en la que en el diseño de un algoritmo para resolver un problema, se tiene que tomar una decisión respecto a qué recurso se debe optimizar.
- Si se decide optimizar el tiempo entonces la ejecución será más rápida pero se utilizará una mayor cantidad de memoria.
- Si se decide optimizar memoria, entonces el programa será mas lento.



2.2.7 Compromiso espacio-tiempo

- Un ejemplo de ello son los archivos comprimidos, cuando ocupan un menor espacio en memoria es más lento acceder a ellos.
- Otro ejemplo de esto es la asignación de memoria por “Peor ajuste” vista en el tema 1.

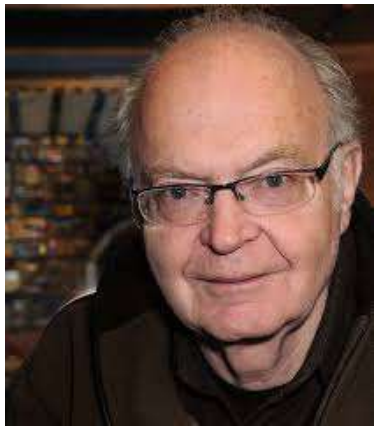


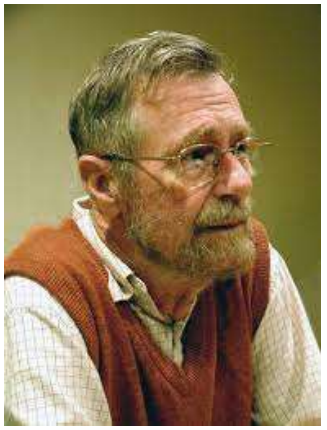
2.2.8 Herramientas de SW y diseño avanzado

- En el software lo que se mide son atributos propios del mismo, normalmente se descompone un atributo general en otros más simples de medir.
- Algunos de los atributos que se suelen medir son:
  - Funcionalidad
  - Número de errores durante un periodo determinado
  - Capacidad de respuesta frente a errores externos
  - Nivel de seguridad
  - Errores en la codificación.
  - Tamaño de un producto informático.

2.2.8 Herramientas de SW y diseño avanzado

- Los algoritmos son la parte fundamental de todos los aspectos de la ciencia e ingeniería de la computación.
- Un buen diseño de algoritmos y estructuras de datos es esencial para el buen desempeño de un sistema de información.
- Un conocimiento profundo de las propiedades teóricas de los algoritmos es esencial para cualquier científico de la computación.





2.2.8 Herramientas de SW y diseño avanzado

- La investigación teórica de los algoritmos conduce a una comprensión más profunda de la estructura de los problemas; el conocimiento de una gran variedad de tipos de algoritmos permite observar un nuevo problema desde diferentes ángulos