# Complejidad algorítmica

# Algoritmos

En muchos casos es deseable que los algoritmos hagan uso eficiente de recursos:

- Tiempo
- Memoria
- Tiempo del procesador
- Medios de almacenamiento secundario
- Red

# Análisis de eficiencia de algoritmos

- Se puede plantear una función

  *Uso_de_Recurso_R(Tamaño_del_Problema)*

  que relacione cuanto de un recurso determinado se utiliza con el tamaño del problema al cual se aplica.

- En este caso tomaremos como recurso de estudio el tiempo, con lo cual la función sería

  *Tiempo_de_ejecución(Tamaño_del_problema)*

# Magnitudes: Tamaño del problema

- Llamamos *n* a la medida del tamaño del problema o de los datos a procesar.
- Qué es lo que mide *n* depende de cada problema en particular.
  - Ordenamiento: *n* es la cantidad de elementos a ordenar.
  - Factorial: *n* coincide con el operando.
  - Determinante de una matriz: *n* es el orden de la matriz.

# Magnitudes: Tiempo de ejecución

- *T(n)* o el tiempo de ejecución de un programa en función de su *n* se podría:
  - Medir físicamente, ejecutando el programa y tomando el tiempo
  - Contando las instrucciones a ejecutar y multiplicando por el tiempo requerido por cada instrucción

pero…

# Magnitudes: Tiempo de ejecución

- … los métodos anteriores dependen del modelo de recursos disponibles y esto imposibilita la comparación con otros algoritmos.

  Entonces…

# Magnitudes: Tiempo de ejecución

- El modelo que más comúnmente se adopta consiste en una máquina genérica con un conjunto de operaciones básicas que el procesador puede ejecutar.
- Para determinar el tiempo de ejecución de un algoritmo se cuentan las operaciones básicas que componen el mismo y se multiplican por el tiempo que lleva cada una.

# T($n$) versus $T_{prom}(n)$

- T($n$) es el tiempo de ejecución en el *peor caso*.
- $T_{prom}(n)$ es el valor medio del tiempo de ejecución de todas las entradas de tamaño $n$.
- Aunque parezca más razonable $T_{prom}(n)$, puede ser engañoso suponer que todas las entradas son igualmente probables.
- Casi siempre es más difícil calcular $T_{prom}(n)$, ya que el análisis se hace intratable en matemáticas y la noción de entrada *promedio* puede carecer de un significado claro.

# Asíntotas

- Los problemas pequeños en general se pueden resolver de cualquier forma.
- En cambio son los problemas grandes los que plantean desafíos y requieren de la administración cuidadosa de los recursos.
- Estudiaremos entonces el *comportamiento asintótico* de los algoritmos, es decir qué sucede con los mismos cuando *n* tiende a infinito.

# Notación asintótica *O*

Para hacer referencia a la velocidad de crecimiento de una función se puede utilizar la *notación asintótica* u *O* (“o mayúscula) y que señala qué función se comporta como “techo” de crecimiento o cota superior de la primera.

# Notación asintótica *O*

Por ejemplo, si se describe que el tiempo de ejecución *T(n)* de un programa es $O(n^2)$ (se lee “o mayúscula de n al cuadrado”) esto quiere decir que a partir de un cierto tamaño de problema, *T(n)* siempre será menor o igual que $n^2$ (o que $n^2$ multiplicada por una constante).

Más formalmente:

Existen constantes positivas *c* y *n0* tales que para $n \geq n_0$ se tiene que $T(n) \leq c \cdot n^2$

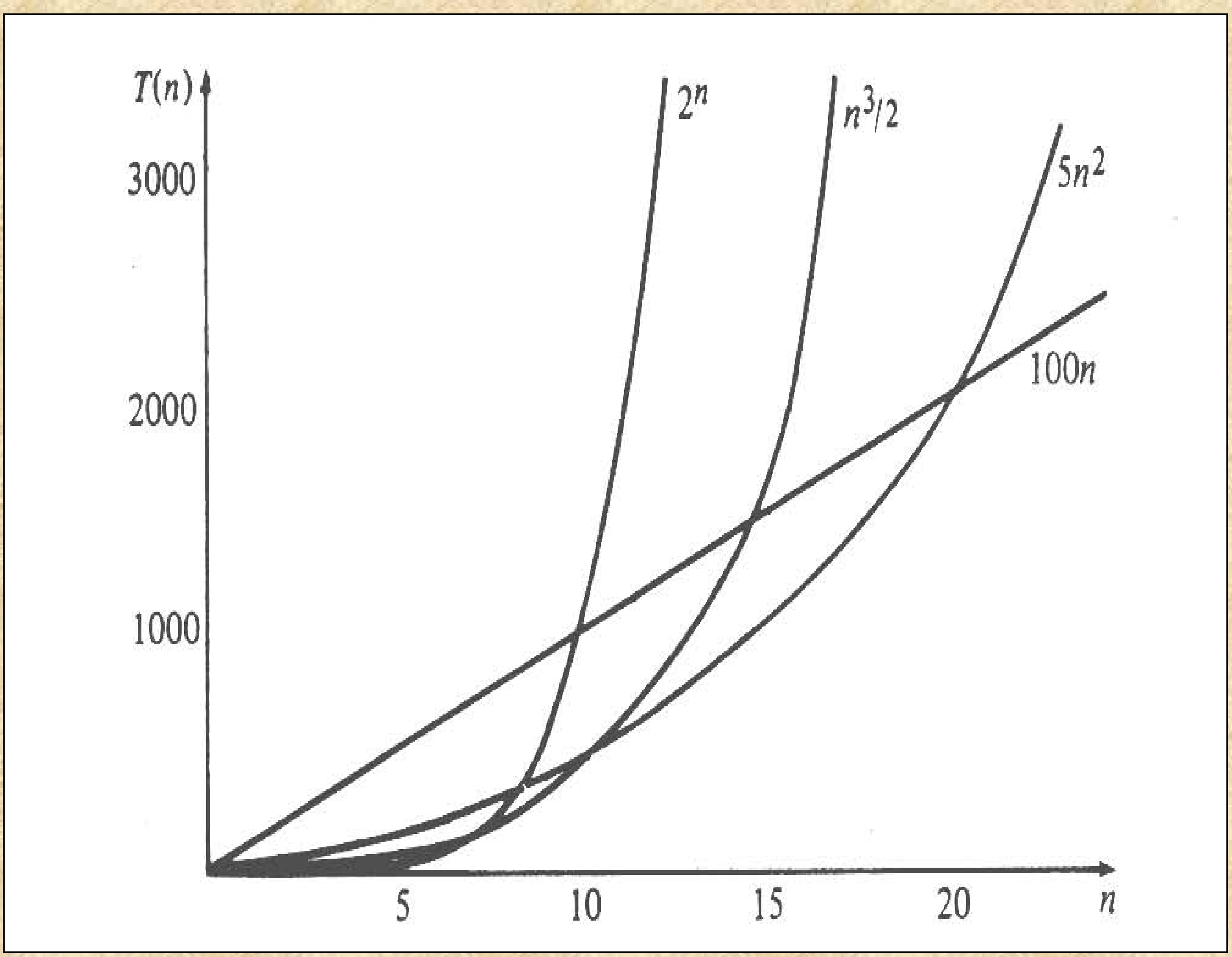
# Órdenes de complejidad

- *O(f(n))* define un *orden de complejidad*, es decir una “familia” de funciones que crecen de una determinada manera. Así tenemos los siguientes órdenes de complejidad.

# Órdenes de complejidad

| | |
|---|---|
| • *O(1)* | Orden constante |
| • *O(log n)* | Orden logarítmico |
| • *O(n)* | Orden lineal |
| • O(n log n) | Orden lineal * logaritmo |
| • $O(n^2)$ | Orden cuadrático |
| • $O(n^a)$ | Orden polinomial (*a>2)* |
| • $O(a^n)$ | Orden exponencial (a>2) |
| • *O(n!)* | Orden factorial |

# Órdenes de complejidad



# Órdenes de complejidad

- Esta tabla además coincide con un orden jerárquico ya que aquellas funciones *O(1)*, es decir que tienen como "techo" una función constante también serán *O(log n), O(n), O(n log n)*, etc., ya que también tendrán como cota superior una función logarítmica, lineal, *n log n*, etc., aunque estas sean menos descriptivas de su comportamiento.

# Impacto práctico

- Supongamos diferentes algoritmos de diferentes complejidades para resolver un mismo problema.

| *T(n)* | **Tamaño máximo del problema para 1000 segundos.** | **Tamaño máximo del problema para 10000 segundos.** | **Incremento en el tamaño máximo del problema.** |
|---|---|---|---|
| *100 n* | 10 | 100 | 10 |
| *5 n^2* | 14 | 45 | 3.2 |
| *(n^3) / 2* | 12 | 27 | 2.3 |
| *2^n* | 10 | 13 | 1.3 |

# Impacto práctico

- Nótese en la segunda columna que para un tiempo de 10^3 segundos, el tamaño máximo de problema es similar para todos los algoritmos.
- En la tercera se observa el tamaño máximo de problema si disponemos de 10 veces más tiempo, o si incrementamos en la misma proporción la potencia del procesador.

# Impacto práctico

- La cuarta columna pone en evidencia entre otras cosas:
  - Los beneficios de un algoritmo de complejidad lineal, que permite el incremento del tamaño máximo de problema en la misma proporción que el incremento del recurso tiempo.
  - Que independientemente de la potencia del procesador o del tiempo de ejecución del que se disponga un algoritmo exponencial sólo permitirá resolver problemas pequeños.

# Desempeño de los algoritmos

- Para los algoritmos *O(n)* y *O(n log n)*, aproximadamente a doble de tiempo, doble de datos procesados.
- Los algoritmos de complejidad logarítmica tienen una muy buena tasa de crecimiento, ya que en el doble de tiempo permiten resolver problemas notablemente mayores.

# Desempeño de los algoritmos

- Los algoritmos de crecimiento polinómico están en la frontera de lo tratable: los de orden cuadrático o cúbico suelen serlo, mientras que grados cercanos a $O(n^{100})$ no lo son
- Algoritmos de órdenes de complejidad superiores a los polinómicos son considerados intratables, excepto para $n$ muy pequeños.

# Criterios para ponderar la complejidad algorítmica

- Si bien siempre se debe tener en cuenta la complejidad de un algoritmo a veces influyen otros factores para inclinarse por una solución:
  - Si un programa será utilizado pocas veces y con entradas relativamente pequeñas el tiempo de desarrollo es más relevante que el de ejecución.
  - Si el programa tendrá larga vida se tiene que tener en cuenta en el costo de mantenimiento, y entonces deberá privilegiarse su legibilidad.
  - Para *n* pequeños puede ser mejor un algoritmo más complejo. Veamos…

# Criterios para ponderar la complejidad algorítmica

- El algoritmo f tiene $T(n) = 100\, n$ (es $O(n)$)
- El algoritmo g tiene $T(n) = n^2$ (es $O(n^2)$)

Sin dudas para *n* grandes f es mejor, sin embargo para valores de *n* menores a 100 es preferible el uso del algoritmo g.

# Problemas P, NP y NP-completos

- La complejidad de un **problema** es la del mejor **algoritmo** conocido para resolverlo.
- A los problemas se los puede clasificar según su complejidad.
- Los más difíciles son los *clase P, clase NP* y *clase NP-completos*

# Problemas clase P

- Son aquellos de complejidad polinomial.
- Se dice que son tratables porque suelen ser abordables en la práctica.
- Los de complejidades superiores son problemas intratables. Para estos sería interesante encontrar una solución polinómica (o mejor) que permitiera resolverlos.

# Problemas clase NP

- Algunos problemas intratables tienen como característica que puede aplicarse un algoritmo polinomial para comprobar si una posible solución es válida o no.
- Esto se emplea con métodos heurísticos para obtener soluciones hipotéticas que se van aceptando o desestimando a ritmo polinómico

# Problemas clase NP-completos

- Son problemas NP de extrema complejidad.
- Se caracterizan por ser “todos iguales”, en el sentido de que si se descubriera una solución para uno de ellos, esta sería aplicable a todos ellos.
- Si se hallara esta solución se resolverían fácilmente todos los problemas NP.
- Se supone que no existe tal solución, pero no se ha demostrado su inexistencia.

# Reglas prácticas para determinar la complejidad de un algoritmo.

- Si bien no existe una receta para encontrar la O más descriptiva de un algoritmo, muchas veces se pueden aplicar las siguientes reglas.

# Reglas prácticas para determinar la complejidad de un algoritmo.

- Sentencias sencillas: instrucciones de asignación o de E/S siempre que no involucren estructuras complejas. Tienen complejidad constante (*O(1)*)
- Secuencia: una serie de instrucciones de un programa tiene el orden de la suma de las complejidades de estas.
  - La suma de dos o más complejidades da como resultado la mayor de ellas.

# Reglas prácticas para determinar la complejidad de un algoritmo.

- <u>Estructuras alternativas</u>: se debe sumar la complejidad de la condición con la de las ramas.
  - La condición suele ser de orden *O(1)*
  - De las ramas se toma la peor complejidad.

# Reglas prácticas para determinar la complejidad de un algoritmo.

- Estructuras repetitivas: se deben distinguir dos casos diferentes.
  - Cuando *n* no tiene que ver con la cantidad de veces que se ejecuta el bucle, la repetición introduce una constante multiplicativa que termina absorbiéndose.
  - Cuando *n* determina de alguna manera la cantidad de iteraciones sí modificará la complejidad. Veamos algunos ejemplos.

# Ejemplo de cálculo de complejidad de estructuras repetitivas (N = tamaño del problema)

- for (i=0; i<=K; i++)  // algo_de_O(1) =>
  *K*O(1) = O(K)*   // Orden Constante

- for (i=0; i<=N; i++)  // algo_de_O(1)  =>
  *N * O(1) = O(n)* // Orden Lineal

- for (i=0; i<=N; i++)
  for (j=0; j<=N; j++)
  algo_de_O(1)   // Orden cuadrático

tendremos $N * N * O(1) = O(n^2)$

# Ejemplo de cálculo de complejidad de estructuras repetitivas

- ```
  for (  i=0; i<=N; i++)
       for  (j=0; j<=i; j++)
            algo_de_O(1)
  ```
- el bucle exterior se realiza N veces, mientras que el interior se realiza 1, 2, 3, ... N veces respectivamente.
- En total, $1 + 2 + 3 + \ldots + N = N*(1+N)/2$ -> $O(n^2)$
  ```

# Ejemplo de cálculo de complejidad de estructuras repetitivas

A veces aparecen bucles multiplicativos, donde la evolución de la variable de control no es lineal (como en los casos anteriores)

```
c = 1;
while (c < N) {
        algo_de_O(1);
        c = 2 * c;
}
```

El valor inicial de "c" es 1, siendo "$2^k$" al cabo de "k" iteraciones. El número de iteraciones es tal que $2^k >= N => k= eis (log_2(N))$ [el entero inmediato superior] y, por tanto, la complejidad del bucle es *O(log n)*.

# Ejemplo de cálculo de complejidad de estructuras repetitivas

- ```
  c = N;
  while (c > 1) {
        algo_de_O(1);
        c = c / 2;
  }
  ```

Un razonamiento análogo nos lleva a $log_2(N)$ iteraciones y, por tanto, a un orden *O(log n)* de complejidad.
  ```

# Ejemplo de cálculo de complejidad de estructuras repetitivas

```
for (i=0; i<= N; i++) {
        c = i;
        while (c > 1) {
                algo_de_O(1);
                c = c / 2;
        }
}
```

tenemos un bucle interno de orden O(log n) que se ejecuta N veces, luego el conjunto es de orden O(n log n)