Analisis de algoritmos

Eficiencia

- Es la capacidad de disponer de un recurso.
- En el caso de los algoritmos, la eficiencia se logra haciendo el mejor uso posible de los recursos del sistema.

Recursos

- ¿Qué recursos?
  - Tiempo
  - Memoria
  - 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

- Se toman como unidad las operaciones elementales: asignación y comparación.
- Luego se puede multiplicar el resultado por el tiempo que llevan estas unidades.

Ejemplo: mínimo de tres valores.

Método 1:

```
m:= a;
If b<m then m:= b;
If c<m then m:= c;
```

Método 2:

```
If a <= b then
  if a <= c then m:= a
  else m:= c
Else if b<= c then m:= b
           else m:= c;
```

Ejemplo: mínimo de tres valores.

Método 3:

```
If (a<=b)and(a<=c) then m:= a;
If (b<=a)and(b<=c) then m:= b;
If (c<=a)and(c<=b) then m:= c;
```

Método 4:

```
If (a <= b)and(a<=c) then
  m:=a
else if b <= c then
  m:= b
  else m:= c;
```

Ejemplo: mínimo de tres valores.

- Método 1:

```
m:= a;
If b<m then m:= b;
If c<m then m:= c;
```

  - Comparaciones: 2
  - 1 <= Asignaciones <= 3

Ejemplo: mínimo de tres valores.

Método 2:

```
If a <= b then
   if a <= c then m:= a
   else m:= c
Else if b<= c then m:= b
              else m:= c;
```

- Comparaciones: 2
- Asignaciones: 1

Ejemplo: mínimo de tres valores.

Método 3:

```
If (a<=b)and(a<=c) then m:= a;
If (b<=a)and(b<=c) then m:= b;
If (c<=a)and(c<=b) then m:= c;
```

- Comparaciones: 6
- Asignaciones: 1

Ejemplo: mínimo de tres valores.

Método 4:

```
If (a <= b)and(a<=c) then
  m:=a
else if b <= c then
  m:= b
  else m:= c;
```

- 2 <= Comparaciones <=3
- Asignaciones: 1

Ejemplo: mínimo de tres valores.

¿La diferencia parece poco relevante?

Ejemplo: mínimo de tres valores.

Considere la repetición 500 veces de estas comparaciones: resultan 1000, 1500 o 3000 comparaciones.

Si por ejemplo tuviera que compararse los 500 elementos de tres vectores, determinando para cada *i* cuál de los tres es el menor.

Análisis de algoritmos

- Se realiza tomando los siguientes casos:
  - El mejor
  - El peor
  - El caso promedio

El primero no presenta mayor interés. En general son más útiles los otros dos.

El caso promedio

- Volvamos sobre el método 4.

```
If (a <= b)and(a<=c) then
   m:=a
else if b <= c then
   m:= b
   else m:= c;
```

  - En el mejor de los casos: 2 comparaciones.
  - En el peor de los casos: 3 comparaciones.

El caso promedio

- Examinemos todos los casos posibles:
  - a <= b <= c   2 comparaciones
  - a <= c <= b   2 comparaciones
  - b <= a <= c   3 comparaciones
  - b <= c <= a   3 comparaciones
  - c <= a <= b   3 comparaciones
  - c <= b <= a   3 comparaciones

  Promedio: (2+2+3+3+3+3)/6 = 8/3 (2,66)

Ejemplo: evitar repetición innecesaria de cálculos

- Método 1:

```
T:= x * x * x;
Y:= 0;
For n:= 1 to 2000 do y := y + 1 / (t-n)
```

- Método 2:

```
Y:= 0;
For n:= 1 to 2000 do y := y + 1 / (x * x * x - n)
```

¿Cuál es más eficiente y por qué?

Análisis de algoritmos

- Hay dos maneras de estimar el tiempo de ejecución:
  - Análisis empírico: se mide el tiempo de respuesta para distintos juegos de datos.
  - Análisis teórico: se calculan el número de comparaciones y asignaciones que efectúa el algoritmo.

Análisis de algoritmos

- Si bien las técnicas empíricas son más sencillas de aplicar presentan algunos inconvenientes:
  - Están afectadas por la potencia del equipo en el que se mide el tiempo.
  - Presentan variaciones según las características de los datos de entrada

- Cuando es posible, se prefiere el análisis teórico del algoritmo.

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 que 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:

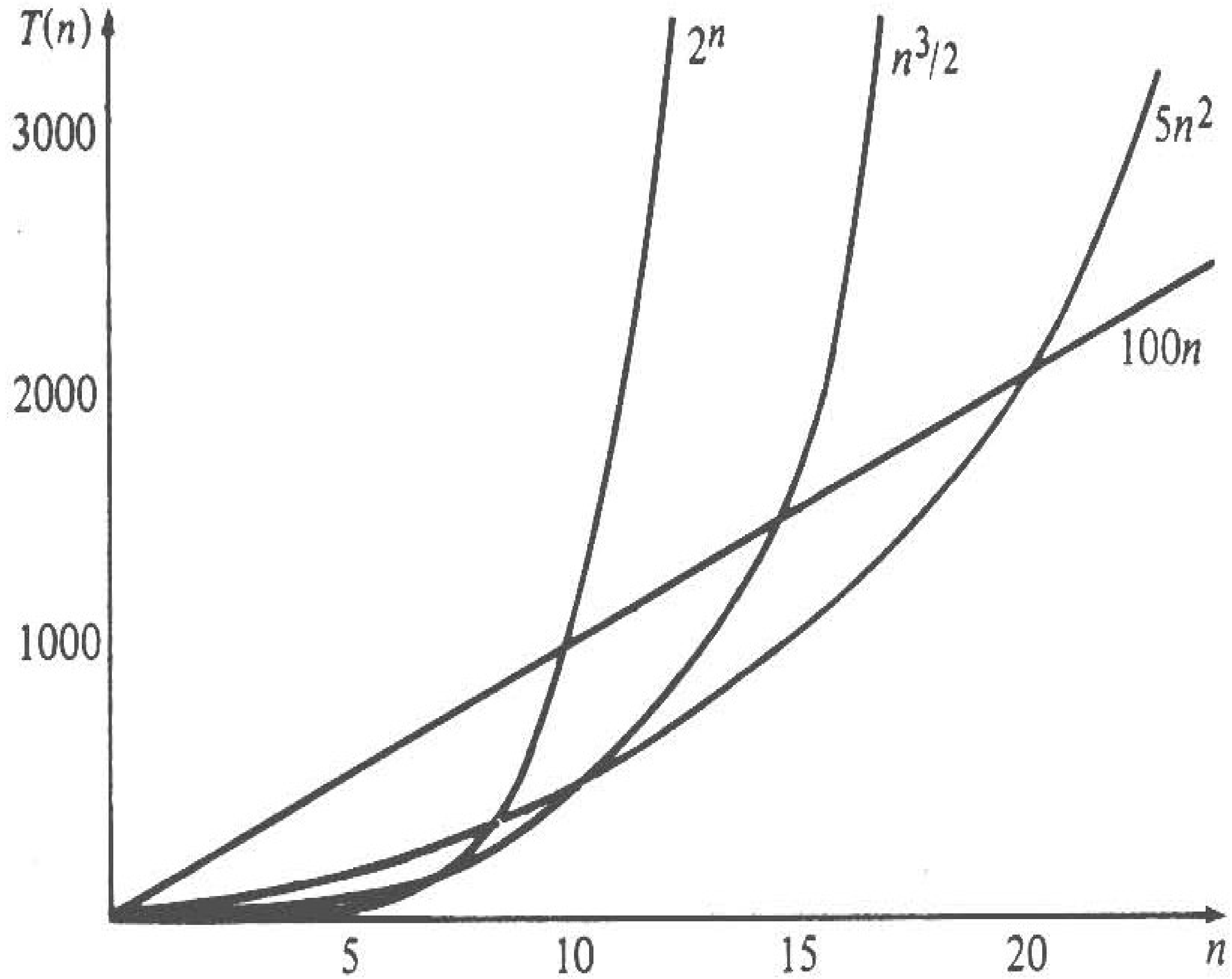
Existen constantes positivas *c* y *n0* tales que para $n \geq n_0$ se tiene que $T(n) \leq c \cdot n^2$

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)
- *O(* n^2 *)* Orden cuadrático
- *O(* n^a *)* Orden polinomial (*a>2)*
- *O(* a^n *)* Orden exponencial (a>2)
- *O(n!)* Orden factorial


T(n)
3000
2000
1000
2^n
n^3/2
5n^2
100n
5
10
15
20
n

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.

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.

- Estructuras alternativas: 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

- for (i= 0 to K) { algo_de_O(1) } => *K*O(1) = O(1)*
- for (i= 0 to N) { algo_de_O(1) } => *N * O(1) = O(n)*
- for (i= 0 to N) do

  for (j= 0 to N) do

  algo_de_O(1)

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

Ejemplo de cálculo de complejidad de estructuras repetitivas

- for (in i= 0; i < N; i++) {

  for (int 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 + ... + 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 (int 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)

Casos de aplicación

- Algoritmos de búsqueda
- Algoritmos de ordenamiento

Algoritmos de búsqueda: búsqueda lineal (I)

```
Function Buscar(a: tvector; n:indice; x:tipoelemento):indice;
Var j,k:indice;
Begin
    j:=0;
    for k:= 1 to n do
            if x= a[k] then j:=k;
    buscar:= j;
End;
```

Algoritmos de búsqueda: búsqueda lineal (I)

- El vector pasa por valor. Costo adicional en cuanto a memoria y tiempo.
- El vector es recorrido hasta el final aunque se encuentre el elemento antes.

Algoritmos de búsqueda: búsqueda lineal (II)

```
Function Buscar(a: tvector; n:indice; x:tipoelemento):indice;
Var j,k:indice;
Begin
    j:=0;
    k:=0;
    while (k<n) and (x<>a[k]) do
            k:= k + 1;
    if x=a[k] then
            j:=k;
    buscar:= j;
End;
```

Algoritmos de búsqueda: búsqueda lineal (II)

- Análisis teórico: caso promedio

Posición del elem. buscado en el vector	Cantidad de comparaciones
1	1
2	2
3	3
…	…
n	n

Algoritmos de búsqueda: búsqueda lineal (II)

- Análisis teórico: caso promedio.

  Cantidad de iteraciones:

$$\frac{1+2+3+...+n}{n}$$

$$\frac{1+n}{2}$$

Algoritmos de búsqueda: búsqueda lineal (II)

- Análisis teórico: el peor caso.

  Si el elemento está en la última posición del vector, se deberán realizar n comparaciones.

  $T(n) = c \cdot n$

  $T(n)$ es $O(n)$ -> Complejidad lineal

Algoritmos de búsqueda: búsqueda binaria

```
Function Buscar(a: tvector; n:indice; x:tipoelemento):indice;
Var j, pri, ult, medio:indice;
Begin
    j:=0; pri:=1; ult:=n;
    medio:=(pri+ult) div 2;
    while (pri<=ult) and (x<>a[medio]) do
    begin
        if x < a[medio] then
            ult:= medio -1
        else
            pri:= medio + 1;
        medio:= (pri + ulti) div 2;
    end;
    if x=a[medio] then
        j:=medio;
    buscar:= j;
End;
```

Algoritmos de búsqueda: búsqueda binaria

- Análisis teórico: el peor caso.

  El algoritmo sucesivamente dividirá el epacio de búsqueda a la mitad hasta que el mismo tenga longitud igual a uno.

  ¿Cuántas iteraciones lleva esto?

Algoritmos de búsqueda: búsqueda binaria

Iteración Nº	Longitud del vector
0	n
1	$n \cdot \frac{1}{2}$
2	$n \cdot \frac{1}{2} \cdot \frac{1}{2} = n \cdot \left(\frac{1}{2}\right)^2$
3	$n \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{1}{2} = n \cdot \left(\frac{1}{2}\right)^3$
...	$\dots$
n	$n \cdot \frac{1}{2} \cdot \frac{1}{2} \cdot \ldots \cdot \frac{1}{2} = n \cdot \left(\frac{1}{2}\right)^n$

Algoritmos de búsqueda: búsqueda binaria

$$\frac{n}{2^k} = 1$$

$$n = 2^k$$

$$log_2 n = log_2 2^k$$

$$k = log_2 n$$

Así vemos que el algoritmo de búsqueda binaria es *O(log n)*, es decir de complejidad logarítmica.

Algoritmos de búsqueda: búsqueda binaria

- Análisis teórico: el caso promedio.

  En promedio la cantidad de comparaciones será:

$$log_2(n+1) - 1 + (log_2(n+1)/n)$$

Algoritmos de búsqueda: búsqueda binaria

- El caso promedio: demostración.

  La suma de las comparaciones necesarias para hallar cada uno de los elementos de un vector de *n* elementos.

  $$s = (1 \cdot 1) + (2 \cdot 2) + (3 \cdot 4) + (4 \cdot 8) + ... + m \cdot 2^{m-1}$$

  $$2 \cdot s = (1 \cdot 2) + (2 \cdot 4) + (3 \cdot 8) + (4 \cdot 16) + ... + (m-1) \cdot 2^{m-1} + m \cdot 2^{m}$$

  Restando *s* de *2s* tenemos:

  Dado que $s = -1 - 2 - 4 - 8 - ... - 2^{m-1} + m \cdot 2^{m}$

  Se llega a que $n = 2^{m-1} = 1 + 2 + 4 + 8 - ... + 2^{m-1}$

  $$s = m \cdot 2^{m} - n$$

  o

  $$s = (n+1)log_2(n+1) - n$$

  El promedio es

  $$\frac{s}{n} = \frac{(n+1)log_2(n+1) - n}{n} = \frac{n \cdot log_2(n+1) + log_2(n+1) - n}{n}$$

  - Crease o no... $log_2(n+1) - 1 + \frac{log_2(n+1)}{n}$

Algoritmos de búsqueda: búsqueda binaria

- Mirando cuidadosamente la expresión vemos que es apenas ligeramente menor que la cantidad de comparaciones en el peor caso.

$$log_2(n+1) - 1 + (log_2(n+1)/n)$$

Algoritmos de Ordenamiento: Selección

```
Procedure Ordenar(var v: tvector; n:indice);
Var i, j, p:indice;
    item: tipoelem;
Begin
    for i:= 1 to n-1 do
        begin
            p:= i;
            for j:= i+1 to n do
                if v[j]<v[p] then
                    p:=j;
            item:= v[p];
            v[p]:= v[i];
            v[i]:= item;
        end;
End;
```

Algoritmos de Ordenamiento: Selección

- Caso promedio:

  No es sencillo calcularlo por la dificultad de establecer todos los órdenes iniciales posibles.

  Aplicando la fórmula de permutaciones sin repetición, para *n* elementos tenemos *n!* órdenes distintos.

  Por ejemplo, para nada más que 10 elementos, tendríamos que estimar el tiempo de los 3628800 casos distintos.

Algoritmos de Ordenamiento: Selección

- El peor caso. Comparaciones:

Iteración	Comparaciones
1	*n-1*
2	*n-2*
3	*n-3*
...	...
n-1	1

Algoritmos de Ordenamiento: Selección

- El peor caso. Comparaciones:

$$
\begin{array}{rl}
 & comp = (n-1) + (n-2) + (n-3) + ... + \quad 2 \quad + \quad 1 \\
+ & comp = \quad 1 \quad + \quad 2 \quad + \quad 3 \quad + ... + (n-2) + (n-1) \\
\hline
 & 2 \cdot comp = \quad n \quad + \quad n \quad + \quad n \quad + ... + \quad n \quad + \quad n
\end{array}
$$

$$comp = \frac{n \cdot (n-1)}{2}$$

Algoritmos de Ordenamiento: Selección

- El peor caso. Intercambios:

  Se producen *n-1* intercambios. Como cada intercambio conlleva 3 asignaciones, tenemos

$$asignaciones = 3 * (n-1)$$

Algoritmos de Ordenamiento: Selección

Observando comparaciones y asignaciones podemos concluir que el algoritmo es de complejidad cuadrática -> $O(n^2)$.

Algoritmos de Ordenamiento: Burbuja

```
Procedure Ordenar(var v: tvector; n:indice);
Var i, j, p:indice;
    item: tipoelem;
Begin
    for i:= n downto 2 do
        begin
            for j:=1 to i-1 do
                if v[j]<v[j+1] then
                    begin
                        item:= v[j];
                        v[j]:= v[j+1];
                        v[i+1]:= item;
                    end;
        end;
End;
```

Algoritmos de Ordenamiento: Burbuja

- El peor caso. Comparaciones:

  Al igual que las del método de selección, la cantidad de comparaciones es igual a:

$$comp = \frac{n \cdot (n-1)}{2}$$

Algoritmos de Ordenamiento: Burbuja

- El peor caso. Intercambios:

  Por cada comparación se produce un intercambio; teniendo en cuenta que cada intercambio se compone de tres asignaciones:

$$asig = \frac{3 \cdot n \cdot (n-1)}{2}$$

Algoritmos de Ordenamiento: Burbuja

- Tanto por la cantidad de comparaciones como por la de asignaciones, la complejidad del método es cuadrática -> $O(n^2)$.

Algoritmos de Ordenamiento: Merge-sort

- Utiliza la estrategia “divide y vencerás”.
- Se divide el vector en dos mitades.
- Luego se ordena cada una de ellas.
- Finalmente se mezclan los dos subvectores (merging).

Algoritmos de Ordenamiento: Merge-sort

```
Procedure Ordenar(var v: tvector; ini, fin:indice);
Var medio:indice;
Begin
    if ini < fin then
        begin
            medio:= (ini + fin) div 2;
            Ordenar(v, ini, medio);
            Ordenar(v, medio+1, fin);
            merge(v, ini, medio, fin);
        end;
End;
```

Algoritmos de Ordenamiento: Merge-sort

```
Procedure Merge(var a: tvector; inicio, medio, fin:indice);
Var i, k:indice;
    v: tvector;
Begin
    j:= inicio;  k:= medio + 1;        i:=0;
    while (j<= medio) and (k<=fin)do
    begin
        i:=i+1;
        if a[j] < a[k] then
        begin
            v[i]:= a[j]; j:= j + 1;
        end
        else
        begin
            v[i]:= a[k];     k:= k +1;
        end;
    end;
```

Algoritmos de Ordenamiento: Merge-sort

```
{Volcar los elementos que queden en el primer o segundo subvector}
while (j<= medio) do
    begin
        i:=i+1; v[i]:= a[j];        j:= j + 1;
    end;
while (k<= fin) do
    begin
        i:=i+1; v[i]:= a[k];        k:= k + 1;
    end;
{Asignar v al vector de salida a}
a := v;
End;
```

Algoritmos de Ordenamiento: Merge-sort

- La mezcla de los dos subvectores es de complejidad lineal -> O(n).
- Si el ordenamiento tiene complejidad cuadrática el costo de ordenar la mitad del vector bajará a la cuarta parte.

Algoritmos de Ordenamiento: Merge-sort

- Al igual que en el caso de la búsqueda binaria, el vector se irá dividiendo a la mitad hasta quedar de longitud uno (1). Esto ocurre luego de $\log_2(n)$ divisiones.

Algoritmos de Ordenamiento: Merge-sort

- Se repetirá $log_2(n)$ veces código que es $O(n)$, con lo cual la complejidad de merge-sort es:

$$O(n \log n)$$

Algoritmos de Ordenamiento: Merge-sort

- Ventaja adicional: el ordenamiento de cada subvector lo pueden realizar simultáneamente dos procesadores distintos.
- Costo adicional: la mezcla de los dos subvectores requiere del uso de un vector auxiliar, con el consiguiente uso de memoria.
- El método quick-sort emplea una estrategia similar pero al no requerir la mezcla de vectores optimiza el uso de memoria.