Curso de nivelación de algoritmos

Complejidad, búsqueda y ordenamiento

Ejemplo 1.1

¿Qué computa el siguiente algoritmo?

```
RV ← 0
i ← 0
while (i < |A|) {
        j ← 0
        sumaAnteriores ← 0
        while (j < i) {
                sumaAnteriores ← sumaAnteriores + A[j]
                j ← j + 1
        }
        if (sumaAnteriores = A[i]) {
           RV ← RV + 1
        }
        i ← i + 1
}
return RV
```

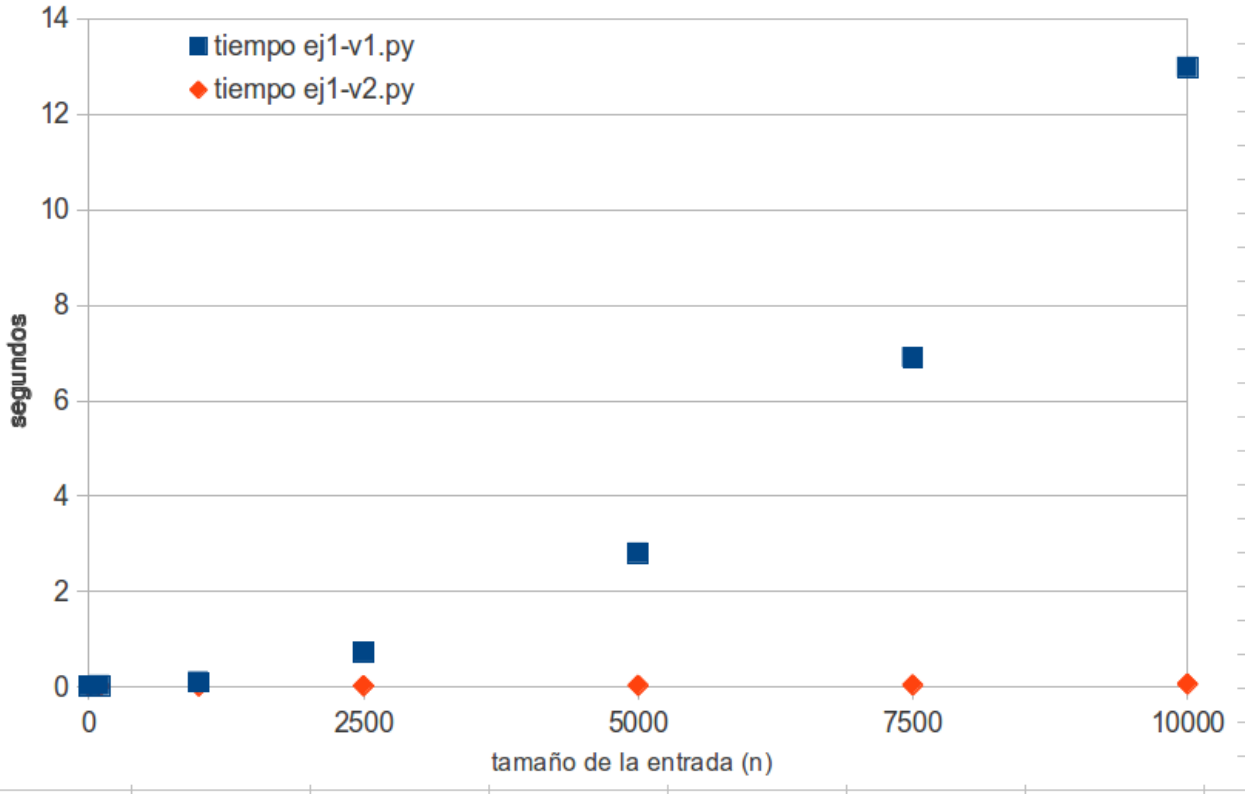
Ejemplo 1.2

¿Y este otro?

$RV \leftarrow 0$
$i \leftarrow 0$
$sumaAnteriores \leftarrow 0$
while $(i < |A|)$ {
    if $(sumaAnteriores = A[i])$ {
        $RV \leftarrow RV + 1$
    }
    $sumaAnteriores \leftarrow sumaAnteriores + A[i]$
    $i \leftarrow i + 1$
}
return RV

Tiempo de ejecución

n	tiempo ej1-v1.py	tiempo ej1-v2.py
1	0.027	0.028
10	0.029	0.029
100	0.034	0.024
1000	0.107	0.027
2500	0.736	0.031
5000	2.808	0.039
7500	6.901	0.048
10000	12.98	0.07



Comparación de los tiempos de ejecución de implementaciones en Python de los algoritmos de los ejemplos 1.1 y 1.2.

Tiempo de ejecución (otro enfoque)

El tiempo de ejecución de un programa se mide en función del tamaño de la entrada.

- Ejemplo: longitud de la lista de entrada.

Tiempo de ejecución (otro enfoque)

El tiempo de ejecución de un programa se mide en función del tamaño de la entrada.

- Ejemplo: longitud de la lista de entrada.

Notación: $T(n)$: tiempo de ejecución de un programa con una entrada de tamaño n.

- Unidad: cantidad de instrucciones.
- Ejemplo: $T(n) = c \cdot n^2$, donde c es una constante.

Tiempo de ejecución

Podemos considerar tres casos del tiempo de ejecución:

- peor caso: tiempo máximo de ejecución para alguna entrada;
- mejor caso: tiempo mínimo de ejecución para alguna entrada;
- caso promedio: tiempo de ejecución para la *entrada promedio*.

Tiempo de ejecución

Podemos considerar tres casos del tiempo de ejecución:

- peor caso: tiempo máximo de ejecución para alguna entrada;
- mejor caso: tiempo mínimo de ejecución para alguna entrada;
- caso promedio: tiempo de ejecución para la *entrada promedio*.

Vamos a considerar sólo el peor caso: $T(n)$ es una cota superior del tiempo de ejecución para entradas arbitrarias de tamaño n.

Ejemplo 1.1

$RV \leftarrow 0$
$i \leftarrow 0$
while $(i < |A|)$ {
    $j \leftarrow 0$
    $sumaAnteriores \leftarrow 0$
    while $(j < i)$ {
        $sumaAnteriores \leftarrow sumaAnteriores + A[j]$
        $j \leftarrow j + 1$
    }
    if $(sumaAnteriores = A[i])$ {
        $RV \leftarrow RV + 1$
    }
    $i \leftarrow i + 1$
}

Ejemplo 1.1

$RV \leftarrow 0$ (1)
$i \leftarrow 0$ (1)
while $(i < |A|)$ { (3)
  $j \leftarrow 0$ (1)
  $sumaAnteriores \leftarrow 0$ (1)
  while $(j < i)$ { (3)
    $sumaAnteriores \leftarrow sumaAnteriores + A[j]$ (5)
    $j \leftarrow j + 1$ (3)
  }
  if $(sumaAnteriores = A[i])$ { (4)
    $RV \leftarrow RV + 1$ (3)
  }
  $i \leftarrow i + 1$ (3)
}

$T(|A|) = 5 + \frac{25}{2}|A| + \frac{11}{2}|A|^2 \in O(|A|^2)$ (orden cuadrático)

Ejemplo 1.2

$RV \leftarrow 0$
$i \leftarrow 0$
$sumaAnteriores \leftarrow 0$
while $(i < |A|)$ {
    if $(sumaAnteriores = A[i])$ {
        $RV \leftarrow RV + 1$
    }
    $sumaAnteriores \leftarrow sumaAnteriores + A[i]$
    $i \leftarrow i + 1$
}

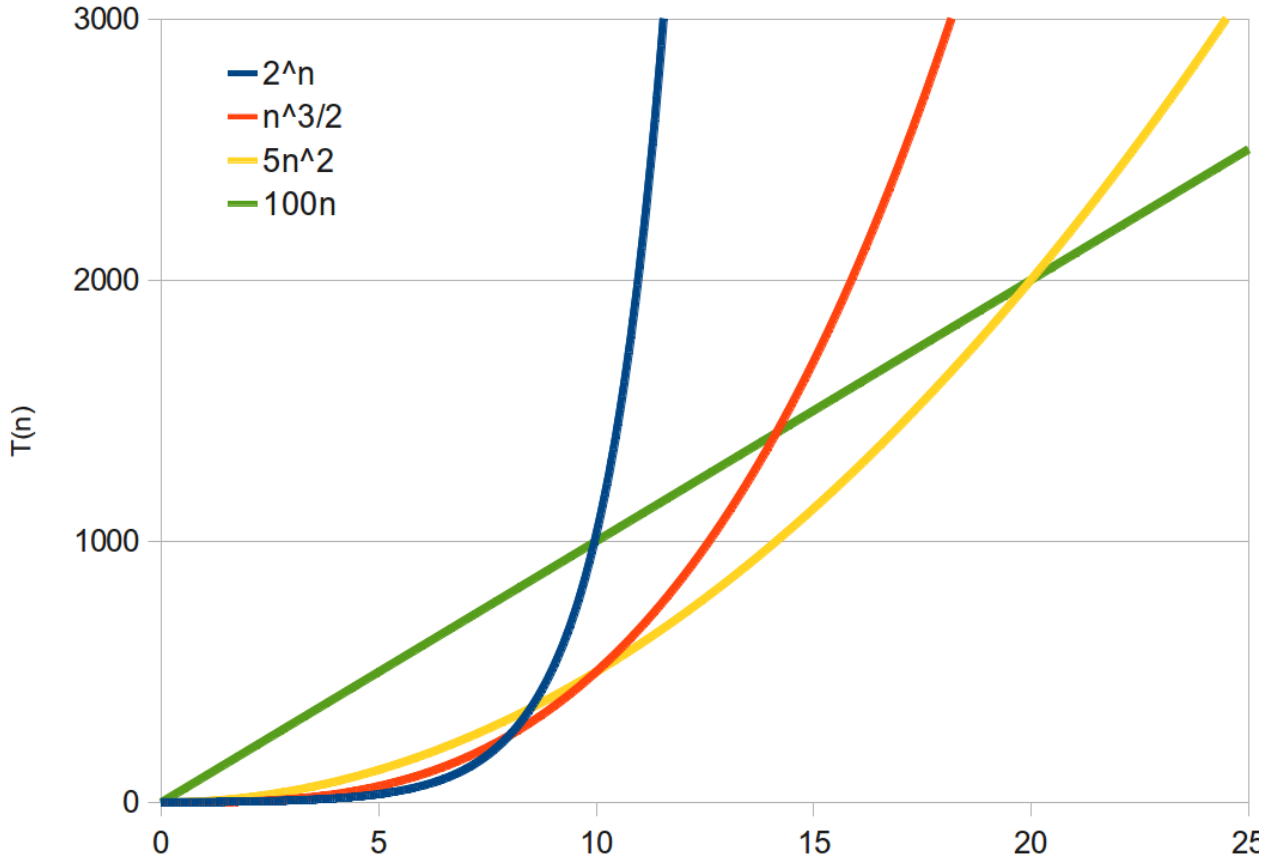
Ejemplo 1.2

$RV \leftarrow 0$ (1)
$i \leftarrow 0$ (1)
$sumaAnteriores \leftarrow 0$ (1)
while $(i < |A|)$ { (3)
    if $(sumaAnteriores = A[i])$ { (4)
        $RV \leftarrow RV + 1$ (3)
    }
    $sumaAnteriores \leftarrow sumaAnteriores + A[i]$ (5)
    $i \leftarrow i + 1$ (3)
}

$T(|A|) = 6 + 18\,|A| \in O(|A|)$ (orden lineal)

Entonces, ¿cuál programa usamos?

Atención #1: Tamaño de la entrada



Si el tamaño de la entrada está acotado, quizá conviene usar un programa de mayor orden pero con constantes bajas.

Entonces, ¿cuál programa usamos?

Atención #2: Objetivos contrapuestos

Para resolver un problema, queremos un programa...

1. que sea fácil de programar (que escribirlo nos demande poco tiempo, que sea simple y fácil de entender);
2. que consuma pocos recursos: tiempo y espacio (memoria, disco rígido).

Entonces, ¿cuál programa usamos?

Atención #2: Objetivos contrapuestos

Para resolver un problema, queremos un programa...

1. que sea fácil de programar (que escribirlo nos demande poco tiempo, que sea simple y fácil de entender);
2. que consuma pocos recursos: tiempo y espacio (memoria, disco rígido).

En general priorizamos un objetivo sobre el otro:

- para programas que correrán pocas veces, priorizamos el objetivo **??**;
- para programas que correrán muchas veces, priorizamos el objetivo **??**.

Problema de búsqueda

$Buscar\ (x, A) \rightarrow (está, pos)$

Problema de búsqueda

$Buscar\ (x, A) \rightarrow (está, pos)$

$está \leftarrow false$
$pos \leftarrow -1$
$j \leftarrow 0$
while $(j < |A|)$ {
    if $(A[j] = x)$ {
        $está \leftarrow$ true
        $pos \leftarrow j$
    }
    $j \leftarrow j + 1$
}

Problema de búsqueda

Buscar $(x, A) \rightarrow (está, pos)$

$está \leftarrow false$
$pos \leftarrow -1$
$j \leftarrow 0$
while $(j < |A|)$ {
    if $(A[j] = x)$ {
        $está \leftarrow$ true
        $pos \leftarrow j$
    }
    $j \leftarrow j + 1$
}

¿Cuál es el orden de complejidad?

Problema de búsqueda

Buscar $(x, A) \to (está, pos)$

$está \leftarrow false$ $O(1)$
$pos \leftarrow -1$ $O(1)$ ¿Cuál es el orden de complejidad?
$j \leftarrow 0$ $O(1)$
while $(j < |A|)$ { $O(1)$
    if $(A[j] = x)$ { $O(1)$
        $está \leftarrow$ true $O(1)$
        $pos \leftarrow j$ $O(1)$
    }
    $j \leftarrow j + 1$ $O(1)$
} while: $O(|A|)$ iteraciones

Problema de búsqueda

Buscar $(x, A) \rightarrow (está, pos)$

$está \leftarrow false$ $O(1)$
$pos \leftarrow -1$ $O(1)$
$j \leftarrow 0$ $O(1)$
while $(j < |A|)$ { $O(1)$
    if $(A[j] = x)$ { $O(1)$
        $está \leftarrow$ true $O(1)$
        $pos \leftarrow j$ $O(1)$
    }
    $j \leftarrow j + 1$ $O(1)$
} while: $O(|A|)$ iteraciones

¿Cuál es el orden de complejidad?

Búsqueda lineal $\in O(|A|)$

Problema de búsqueda

Buscar $(x, A) \to (\textit{está}, \textit{pos})$

```
está ← false    O(1)
pos ← −1    O(1)
j ← 0    O(1)
while (j < |A|) {    O(1)
        if (A[j] = x) {    O(1)
            está ← true    O(1)
            pos ← j    O(1)
        }
        j ← j + 1    O(1)
}    while: O(|A|) iteraciones
```

¿Cuál es el orden de complejidad?

Búsqueda lineal $\in O(|A|)$

¿Y si agregamos "$\wedge \neg \textit{está}$" a la guarda del while?

Problema de búsqueda

Buscar $(x, A) \to (\textit{está}, \textit{pos})$

```
está ← false    O(1)
pos ← −1    O(1)
j ← 0    O(1)
while (j < |A|) {    O(1)
        if (A[j] = x) {    O(1)
            está ← true    O(1)
            pos ← j    O(1)
        }
        j ← j + 1    O(1)
}    while: O(|A|) iteraciones
```

¿Cuál es el orden de complejidad?

Búsqueda lineal $\in O(|A|)$

¿Y si agregamos "$\wedge \neg \textit{está}$" a la guarda del while?

En este algoritmo, cortar antes la ejecución puede ahorrar tiempo, pero no cambia el **orden en el peor caso**.

Problema de búsqueda

Buscar $(x, A) \to (está, pos)$

$está \leftarrow false$ $O(1)$
$pos \leftarrow -1$ $O(1)$
$j \leftarrow 0$ $O(1)$
while $(j < |A|)$ { $O(1)$
  if $(A[j] = x)$ { $O(1)$
    $está \leftarrow$ true $O(1)$
    $pos \leftarrow j$ $O(1)$
  }
  $j \leftarrow j + 1$ $O(1)$
} while: $O(|A|)$ iteraciones

¿Cuál es el orden de complejidad?

Búsqueda lineal $\in O(|A|)$

¿Y si agregamos "$\wedge \neg está$" a la guarda del while?

En este algoritmo, cortar antes la ejecución puede ahorrar tiempo, pero no cambia el **orden en el peor caso**.

¿Cuán eficientes son estos algoritmos si A está ordenado?

Veamos un algoritmo de búsqueda para listas ordenadas.

4	7	23	41	44	59	97	134	165	187	210	212	249	280	314
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Buscamos el número **97**...

Veamos un algoritmo de búsqueda para listas ordenadas.

4	7	23	41	44	59	97	134	165	187	210	212	249	280	314
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Buscamos el número **97**...

4	7	23	41	44	59	97	134
0	1	2	3	4	5	6	7

165	187	210	212	249	280	314
8	9	10	11	12	13	14

Veamos un algoritmo de búsqueda para listas ordenadas.

4	7	23	41	44	59	97	134	165	187	210	212	249	280	314
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Buscamos el número **97**...

4	7	23	41	44	59	97	134
0	1	2	3	4	5	6	7

165	187	210	212	249	280	314
8	9	10	11	12	13	14

4	7	23	41
0	1	2	3

44	59	97	134
4	5	6	7

Veamos un algoritmo de búsqueda para listas ordenadas.

4	7	23	41	44	59	97	134	165	187	210	212	249	280	314
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Buscamos el número **97**...

4	7	23	41	44	59	97	134
0	1	2	3	4	5	6	7

165	187	210	212	249	280	314
8	9	10	11	12	13	14

4	7	23	41
0	1	2	3

44	59	97	134
4	5	6	7

44	59
4	5

97	134
6	7

Veamos un algoritmo de búsqueda para listas ordenadas.

4	7	23	41	44	59	97	134	165	187	210	212	249	280	314
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Buscamos el número **97**...

4	7	23	41	44	59	97	134
0	1	2	3	4	5	6	7

165	187	210	212	249	280	314
8	9	10	11	12	13	14

4	7	23	41
0	1	2	3

44	59	97	134
4	5	6	7

44	59
4	5

97	134
6	7

97	134
6	7

Veamos un algoritmo de búsqueda para listas ordenadas.

4	7	23	41	44	59	97	134	165	187	210	212	249	280	314
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Buscamos el número **97**...

4	7	23	41	44	59	97	134
0	1	2	3	4	5	6	7

165	187	210	212	249	280	314
8	9	10	11	12	13	14

4	7	23	41
0	1	2	3

44	59	97	134
4	5	6	7

44	59
4	5

97	134
6	7

97
6

134
7

97	✓
6	

Búsqueda binaria

¿Cuál es el comportamiento detrás de este algoritmo?

Si la lista está ordenada, entonces en cada paso puedo partir la lista en:

a) la mitad que puede contener el elemento y
b) la mitad que no puede contenerlo.

Indefectiblemente, se llega a un punto en que la lista ya no puede ser dividida (tiene un solo elemento) y, o bien el elemento es el buscado o no.

Búsqueda binaria

Buscar $(x, A) \to (\mathit{está}, \mathit{pos})$ (supone que la lista está ordenada)

Búsqueda binaria

Buscar $(x, A) \to (está, pos)$ (supone que la lista está ordenada)

$(está, pos) \leftarrow (false, -1)$
$(izq, der) \leftarrow (0, |A| - 1)$
while $(izq < der)$ {
    $med \leftarrow (izq + der)$ div 2
    if $(A[med] < x)$ {
        $izq \leftarrow med + 1$
    } else {
        $der \leftarrow med$
    }
}
if $(x = A[izq])$ {
    $(está, pos) \leftarrow (true, izq)$
}

Búsqueda binaria

Buscar $(x, A) \to (está, pos)$ (supone que la lista está ordenada)

$(está, pos) \leftarrow (false, -1)$
$(izq, der) \leftarrow (0, |A| - 1)$
while $(izq < der)$ {
    $med \leftarrow (izq + der)$ div 2
    if $(A[med] < x)$ {
        $izq \leftarrow med + 1$
    } else {
        $der \leftarrow med$
    }
}
if $(x = A[izq])$ {
    $(está, pos) \leftarrow (true, izq)$
}

¿Cuál es el orden de complejidad?

Búsqueda binaria

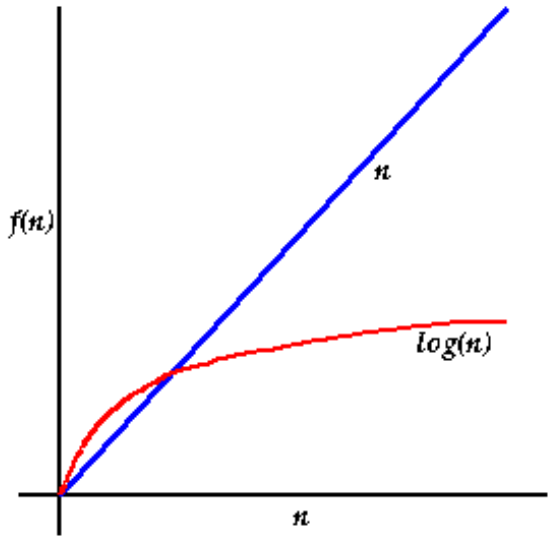
Buscar $(x, A) \to (está, pos)$ (supone que la lista está ordenada)

$(está, pos) \leftarrow (false, -1)$
$(izq, der) \leftarrow (0, |A| - 1)$
while $(izq < der)$ {
    $med \leftarrow (izq + der)$ div 2
    if $(A[med] < x)$ {
        $izq \leftarrow med + 1$
    } else {
        $der \leftarrow med$
    }
}
if $(x = A[izq])$ {
    $(está, pos) \leftarrow (true, izq)$
}

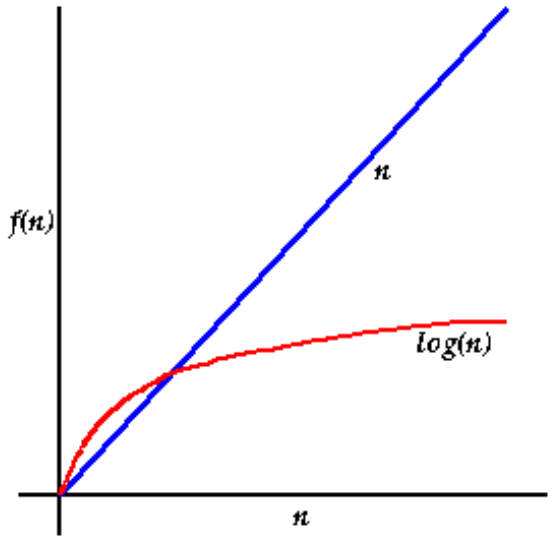
¿Cuál es el orden de complejidad?

$O(\log |A|)$, orden logarítmico

Búsqueda lineal vs. binaria

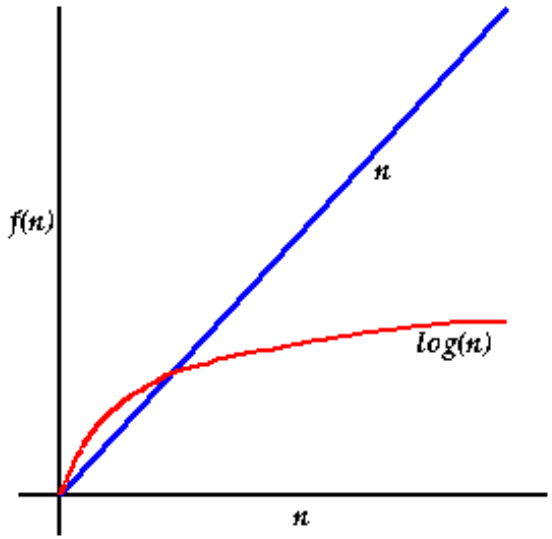


Búsqueda lineal vs. binaria



¿Cuán importante es la diferencia entre $O(\log n)$ y $O(n)$?

Búsqueda lineal vs. binaria

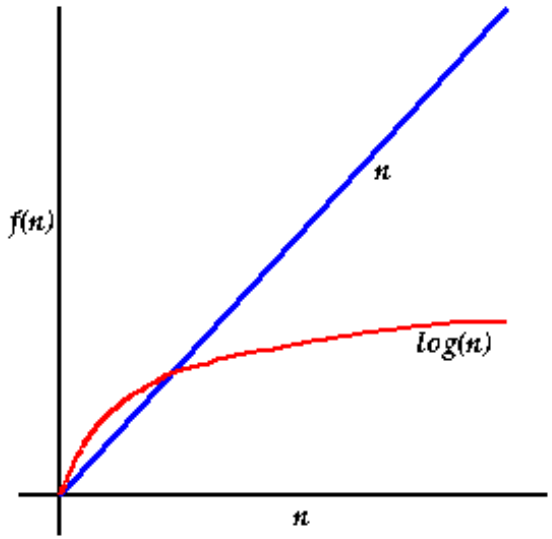


¿Cuán importante es la diferencia entre $O(\log n)$ y $O(n)$?

Depende de nuestro contexto...

- ¿Cuál es el tamaño del listado en el cual haremos la búsqueda? (FCEyN vs. ANSES)

Búsqueda lineal vs. binaria

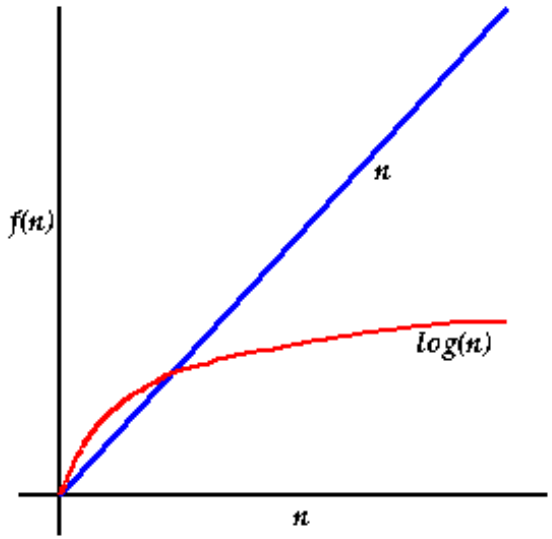


¿Cuán importante es la diferencia entre $O(\log n)$ y $O(n)$?

Depende de nuestro contexto...

- ¿Cuál es el tamaño del listado en el cual haremos la búsqueda? (FCEyN vs. ANSES)
- ¿Cuánto cuesta cada consulta individual? (Lectura en memoria vs. consulta por Internet)

Búsqueda lineal vs. binaria

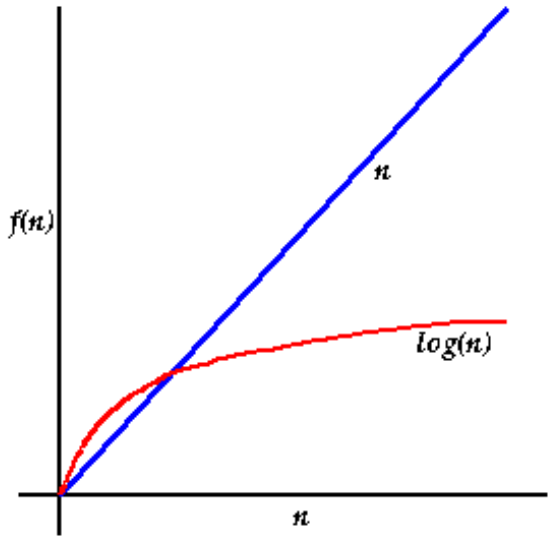


¿Cuán importante es la diferencia entre $O(\log n)$ y $O(n)$?

Depende de nuestro contexto...

- ¿Cuál es el tamaño del listado en el cual haremos la búsqueda? (FCEyN vs. ANSES)
- ¿Cuánto cuesta cada consulta individual? (Lectura en memoria vs. consulta por Internet)
- ¿Cuántas veces vamos a necesitar hacer esta búsqueda? (una vez por mes vs. millones de veces por día)

Búsqueda lineal vs. binaria



¿Cuán importante es la diferencia entre $O(\log n)$ y $O(n)$?

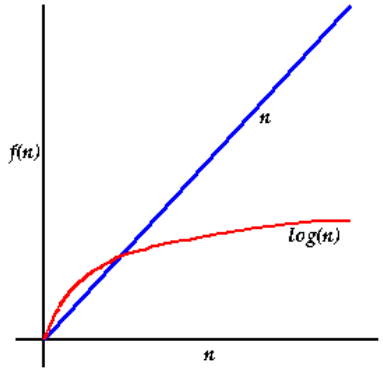
Depende de nuestro contexto...

- ¿Cuál es el tamaño del listado en el cual haremos la búsqueda? (FCEyN vs. ANSES)
- ¿Cuánto cuesta cada consulta individual? (Lectura en memoria vs. consulta por Internet)
- ¿Cuántas veces vamos a necesitar hacer esta búsqueda? (una vez por mes vs. millones de veces por día)
- ¿El contenido de la lista es estable o se modifica muy a menudo?

Problema de ordenamiento

Problema de búsqueda:

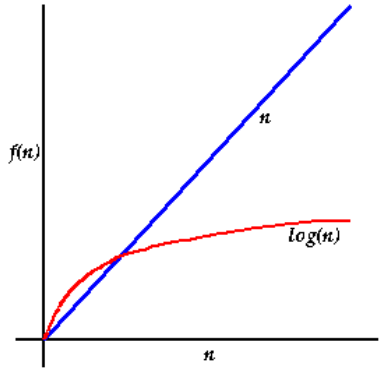
- $O(n)$ para listas arbitrarias.
- $O(\log n)$ para listas ordenadas.



Problema de ordenamiento

Problema de búsqueda:

- $O(n)$ para listas arbitrarias.
- $O(\log n)$ para listas ordenadas.



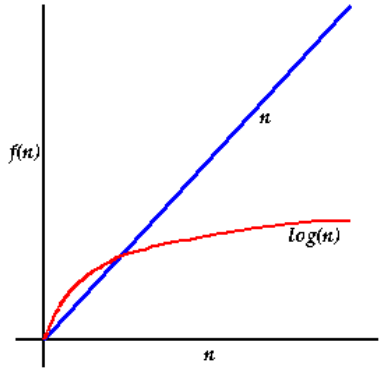
Ordenar una lista es un problema de mucha importancia en la práctica.

- Ej: ¿De qué sirve una guía de teléfonos desordenada?

Problema de ordenamiento

Problema de búsqueda:

- $O(n)$ para listas arbitrarias.
- $O(\log n)$ para listas ordenadas.



Ordenar una lista es un problema de mucha importancia en la práctica.

- Ej: ¿De qué sirve una guía de teléfonos desordenada?

¿Ideas para resolver este problema?

Problema de ordenamiento

59	7	388	41	2	280	50	123

Selection sort

Para cada i entre 0 y $|A| - 1$, buscar el menor elemento en $A[i \,..\, |A| - 1]$ e intercambiarlo con $A[i]$.

Selection sort

Para cada i entre 0 y $|A| - 1$, buscar el menor elemento en $A[i \,..\, |A| - 1]$ e intercambiarlo con $A[i]$.

```
i ← 0
while (i < |A|) {
        posmin ← i
        j ← i + 1
        while (j < |A|) {
                if (A[j] < A[posmin]) {
                   posmin ← j
                }
                j ← j + 1
        }
        swap(A, i, posmin)
        i ← i + 1
}
```

Selection sort

Para cada i entre 0 y $|A| - 1$, buscar el menor elemento en $A[i \,..\, |A| - 1]$ e intercambiarlo con $A[i]$.

$i \leftarrow 0$ $\quad O(1)$
while $(i < |A|)$ { $\quad O(1)$
    $posmin \leftarrow i$ $\quad O(1)$
    $j \leftarrow i + 1$ $\quad O(1)$
    while $(j < |A|)$ { $\quad O(1)$
        if $(A[j] < A[posmin])$ { $\quad O(1)$
            $posmin \leftarrow j$ $\quad O(1)$
        }
        $j \leftarrow j + 1$ $\quad O(1)$
    }
    swap$(A, i, posmin)$ $\quad O(1)$
    $i \leftarrow i + 1$ $\quad O(1)$
}

Selection sort

Para cada i entre 0 y $|A| - 1$, buscar el menor elemento en $A[i \,..\, |A| - 1]$ e intercambiarlo con $A[i]$.

```
i ← 0                                                    O(1)
while (i < |A|) {                                        O(1)        while: O(|A|) iteraciones
    posmin ← i                                           O(1)
    j ← i + 1                                            O(1)
    while (j < |A|) {                                    O(1)        while: O(|A|) iteraciones
        if (A[j] < A[posmin]) {                          O(1)
            posmin ← j                                   O(1)
        }
        j ← j + 1                                        O(1)
    }
    swap(A, i, posmin)                                   O(1)
    i ← i + 1                                            O(1)
}
```

Selection sort

Para cada i entre 0 y $|A| - 1$, buscar el menor elemento en $A[i \,..\, |A| - 1]$ e intercambiarlo con $A[i]$.

$i \leftarrow 0$ $\quad O(1)$
while $(i < |A|)$ { $\quad O(1)$ $\quad$ while: $O(|A|)$ iteraciones
  $posmin \leftarrow i$ $\quad O(1)$
  $j \leftarrow i + 1$ $\quad O(1)$
  while $(j < |A|)$ { $\quad O(1)$ $\quad$ while: $O(|A|)$ iteraciones
    if $(A[j] < A[posmin])$ { $\quad O(1)$
      $posmin \leftarrow j$ $\quad O(1)$
    }
    $j \leftarrow j + 1$ $\quad O(1)$
  }
  swap$(A, i, posmin)$ $\quad O(1)$
  $i \leftarrow i + 1$ $\quad O(1)$
}

Complejidad temporal: $O(|A|^2)$

Insertion sort

Para cada i entre 0 y $|A| - 1$, mover el elemento $A[i]$ a su posición correcta en $A[0\,..\,i]$ (así, $A[0\,..\,i]$ queda ordenado).

Insertion sort

Para cada i entre 0 y $|A| - 1$, mover el elemento $A[i]$ a su posición correcta en $A[0\,..\,i]$ (así, $A[0\,..\,i]$ queda ordenado).

```
i ← 0
while (i < |A|) {
        j ← i
        while (j > 0 ∧ A[j − 1] > A[j]) {
                swap(A, j − 1, j)
                j ← j − 1
        }
        i ← i + 1
}
```

Insertion sort

Para cada i entre 0 y $|A| - 1$, mover el elemento $A[i]$ a su posición correcta en $A[0\,..\,i]$ (así, $A[0\,..\,i]$ queda ordenado).

$i \leftarrow 0$ $\quad O(1)$
while $(i < |A|)$ { $\quad O(1)$
    $j \leftarrow i$ $\quad O(1)$
    while $(j > 0 \wedge A[j-1] > A[j])$ { $\quad O(1)$
        swap$(A, j-1, j)$ $\quad O(1)$
        $j \leftarrow j - 1$ $\quad O(1)$
    }
    $i \leftarrow i + 1$ $\quad O(1)$
}

Insertion sort

Para cada i entre 0 y $|A| - 1$, mover el elemento $A[i]$ a su posición correcta en $A[0\,..\,i]$ (así, $A[0\,..\,i]$ queda ordenado).

```
i ← 0                                   O(1)
while (i < |A|) {                       O(1)        while: O(|A|) iteraciones
    j ← i                               O(1)
    while (j > 0 ∧ A[j − 1] > A[j]) {   O(1)        while: O(|A|) iters
        swap(A, j − 1, j)               O(1)
        j ← j − 1                       O(1)
    }
    i ← i + 1                           O(1)
}
```

Insertion sort

Para cada i entre 0 y $|A| - 1$, mover el elemento $A[i]$ a su posición correcta en $A[0\,..\,i]$ (así, $A[0\,..\,i]$ queda ordenado).

$i \leftarrow 0$ $\quad O(1)$
while $(i < |A|)$ { $\quad$ O(1) $\quad$ while: $O(|A|)$ iteraciones
  $j \leftarrow i$ $\quad O(1)$
  while $(j > 0 \wedge A[j-1] > A[j])$ { $\quad$ O(1) $\quad$ while: $O(|A|)$ iters
    swap$(A, j-1, j)$ $\quad O(1)$
    $j \leftarrow j - 1$ $\quad O(1)$
  }
  $i \leftarrow i + 1$ $\quad O(1)$
}

Complejidad temporal: $O(|A|^2)$

Bubble sort

Comparar cada par de elementos adyacentes en A, e invertirlos si están en orden incorrecto. Repetir $|A|$ veces.

Bubble sort

Comparar cada par de elementos adyacentes en A, e invertirlos si están en orden incorrecto. Repetir $|A|$ veces.

```
i ← 0
while (i < |A|) {
        j ← 0
        while (j < |A| − 1) {
                if (A[j] > A[j + 1]) {
                   swap(A, j, j + 1)
                }
                j ← j + 1
        }
        i ← i + 1
}
```

Bubble sort

Comparar cada par de elementos adyacentes en A, e invertirlos si están en orden incorrecto. Repetir $|A|$ veces.

$i \leftarrow 0$ $O(1)$
while $(i < |A|)$ { $O(1)$
  $j \leftarrow 0$ $O(1)$
  while $(j < |A| - 1)$ { $O(1)$
    if $(A[j] > A[j+1])$ { $O(1)$
      swap$(A, j, j+1)$ $O(1)$
    }
    $j \leftarrow j + 1$ $O(1)$
  }
  $i \leftarrow i + 1$ $O(1)$
}

Bubble sort

Comparar cada par de elementos adyacentes en A, e invertirlos si están en orden incorrecto. Repetir $|A|$ veces.

```
i ← 0                                   O(1)
while (i < |A|) {                       O(1)        while: O(|A|) iteraciones
    j ← 0                               O(1)
    while (j < |A| − 1) {               O(1)        while: O(|A|) iteraciones
        if (A[j] > A[j + 1]) {          O(1)
            swap(A, j, j + 1)           O(1)
        }
        j ← j + 1                       O(1)
    }
    i ← i + 1                           O(1)
}
```

Bubble sort

Comparar cada par de elementos adyacentes en A, e invertirlos si están en orden incorrecto. Repetir $|A|$ veces.

```
i ← 0                                   O(1)
while (i < |A|) {                       O(1)        while: O(|A|) iteraciones
    j ← 0                               O(1)
    while (j < |A| − 1) {               O(1)        while: O(|A|) iteraciones
        if (A[j] > A[j + 1]) {          O(1)
            swap(A, j, j + 1)           O(1)
        }
        j ← j + 1                       O(1)
    }
    i ← i + 1                           O(1)
}
```

Complejidad temporal: $O(|A|^2)$

Demos de ordenamiento

- https://www.toptal.com/developers/sorting-algorithms/
- https://youtu.be/Ns4TPTC8whw
- https://youtu.be/ROaIU379l3U
- https://youtu.be/semGJAJ7i74

Repaso de la clase de hoy

- Tiempo de ejecución: en segundos y en cantidad de operaciones.
- Peor caso, mejor caso y caso promedio.
- Búsqueda lineal y binaria.
- Algoritmos de ordenamiento de listas:
  - Selection sort. $O(n^2)$
  - Insertion sort. $O(n^2)$
  - Bubble sort. $O(n^2)$

Próximos temas

- Recursión algorítmica.
- Divide and Conquer.
- Merge sort.