

Estructura de Datos y Algoritmos

ITBA 2024-Q2

Buscando apariciones...

Cuando se busca una/múltiples apariciones de un elemento en un conjunto, se puede proceder de diferentes formas:

- a) Dejar la colección como está e ingeniárselas para navegar en ella. Ese fue el caso de los algoritmos Naïve y de KMP. En ese caso, el “texto” es una colección de caracteres y no se modifica para que no se pierda su semántica.
- b) Generar una estructura auxiliar, llamada *índice*, que facilite la búsqueda. Ese fue el caso del *archivo invertido*. Los documentos no se modifican para que no pierda su semántica, pero se crea un *índice*. Se busca sobre dicho *índice*. Si la búsqueda resulta exitosa en él, entonces se llega al documento.

Algoritmos sobre índices

Siendo el índice la estructura auxiliar que se utiliza para encontrar un elemento, entonces la **búsqueda sobre la mismo debe ser muy eficiente.**

Índice (definición)

Estructura de datos que facilita la búsqueda (lookup).



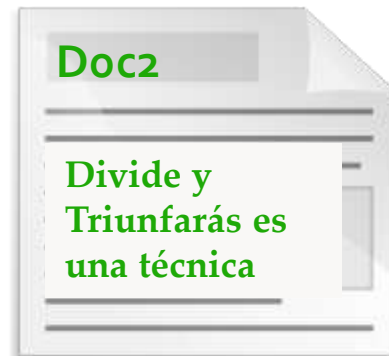
Un índice está compuesto por elementos que representan la información que indizan.

Caso de uso 1: documentos para search engine. La colección de documentos contiene



¿Qué tiene el índice (archivo invertido)?

Caso de uso 1: documentos para search engine. La colección de documentos contiene



**una
técnica**

...

EDA

Clave de búsqueda
(key)

<doc1.txt, doc2.txt, doc3.txt>

<doc1.txt, doc2.txt>

...

<doc3.txt>

Información asociada. Puede estar en RAM o (si es mucha) indicará cómo llegar a la información en disco

Caso de uso 2: La colección contiene “alumnos” (opaco)



¿Qué tiene el índice si quiero buscar por “legajo”?

Caso de uso 2: La colección contiene “alumnos”



58622
58333
...
45382

Clave de búsqueda
(key)

<58622, Ana Garcia, 20, agarcia@gmail.com>
<58333, Pablo Conte, 19, pconte@gmail.com>
...
<45382, Leo Nilo, 20, lnilo@gmail.com>

Información asociada. Puede estar en RAM o (si es mucha) indicará
cómo llegar a la información en disco

Caso de uso 2: La colección contiene “alumnos” (opaco)



¿Qué tiene el índice si quiero buscar por “**edad**”?

Clave repetida
manejada sin
compactar

Caso de uso 2: La colección compacta (compacto)



20
19
...
20

Clave de búsqueda
(key)

<58622, Ana Garcia, 20, agarcia@gmail.com>
<58333, Pablo Conte, 19, pconte@gmail.com>
...
<45382, Leo Nilo, 20, lnilo@gmail.com>

Información asociada. Puede estar en RAM o (si es mucha) indicará
cómo llegar a la información en disco

Clave repetida
manejada con
compactación

Caso de uso 2: La colección con claves repetidas (opaco)



20

19

...

Clave de búsqueda
(key)

<58622, Ana Garcia, 20, agarcia@gmail.com>, ..., <45382,
Leo Nilo, 20, lnilo@gmail.com>
<58333, Pablo Conte, 19, pconte@gmail.com>, ...

...

Información asociada. Puede estar en RAM o (si es mucha) indicará cómo
llegar a la información en disco

Características de Índices

- La clave de búsqueda puede o no tener repetidos. Si es única no tiene sentido hablar de compactación. Si puede repetirse, podremos tener la info asociada compactada o no.
- La clave de búsqueda debe permitir buscar rápidamente la info adicional.

Ahora bien, el índice no sólo se utiliza para “buscar”. Hay otras operaciones **necesarias** sobre él. ¿Cuáles?

- Búsqueda =>obvio, para ello se lo construye.
- Inserción=>el índice debe reflejar los datos de la colección. O sea, si inserto un documento en la colección, preciso que el índice lo refleje.
- Borrado=> el índice debe reflejar los datos de la colección. O sea, si borro un documento en la colección, preciso que el índice lo refleje
- Y más también.

¿Qué estructura de las que conocen podría ser buena para representar un índice? **Supongamos que los repetidos no se compactan y que hay espacio prealocado suficiente para las inserciones.**

Rta

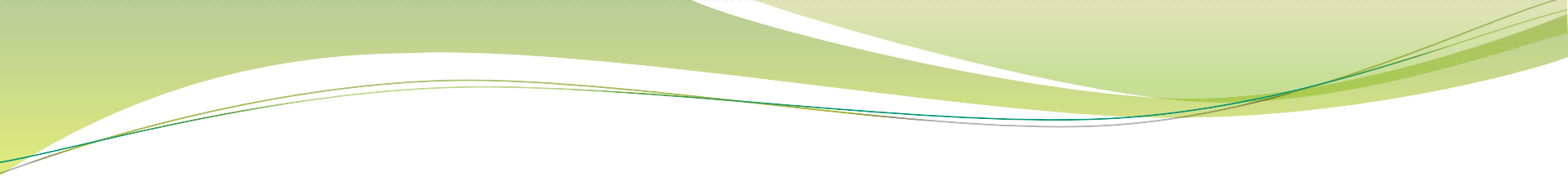
	Búsqueda	Inserción	Borrado
Arreglo (cualquiera, desordenado)	✗	✓ (los agrego al final)	✗
Arreglo ordenado por clave de búsqueda	✓	✗	✗
Hashing	?	?	?

Formalicemos esa clasificación... Calculemos complejidad temporal en cada caso (asumimos **que hay espacio prealocado suficiente para las inserciones**)

	Búsqueda	Inserción	Borrado
Arreglo (cualquiera, desordenado)			
Arreglo ordenado por clave de búsqueda			
Hashing			

El problema del arreglo es que tiene que garantizar contigüidad de sus elementos.
El problema del hashing es si tiene que resolver colisiones. Si no tiene colisiones es $O(1)$, pero es ideal...

Los 3 casos se penalizan si “se acaba el espacio prealocado”



Pareciera que hashing es un escenario propicio para un índice.

Pero, ¿Cómo se comportaría si también precisamos las siguientes típicas operaciones de un índice:

Búsqueda por rangos.

Devolver el máximo/mínimo elemento

Ej: buscar los alumnos que tienen legajos entre 1000 y 2000.

Ej: cuál es la máxima edad?



No hay estructuras de datos perfectas. Dependen de los casos de uso necesarios...

En general los objetivos se contraponen y hay que buscar un trade-off.

Como nada es perfecto, vamos a comenzar analizando el comportamiento de los **arreglos ordenados por clave de búsqueda**...

Arreglos ordenados

¿Por qué dijimos que la búsqueda puntual es $O(\log_2 n)$?

Veamos el algoritmo que toma ventaja de estar ordenado.

Se llama “búsqueda binaria”.

Lo mostramos para claves numéricas enteras de tipo long.

Busco el 34

2	8	10	15	17	21	28	30	34	42	50	60	62	70
---	---	----	----	----	----	----	----	----	----	----	----	----	----

30	34	42	50	60	62	70
----	----	----	----	----	----	----

30	34	42
----	----	----



¿Cómo calcular complejidades en algoritmos
recursivos?



En Pgm imperativa y POO han usado la técnica de programación **Divide y Triunfarás (Divide and Conquer)**:

La solución de un problema de tamaño de entrada N se **divide en problemas de tamaño menor** hasta que la solución es trivial. Finalmente, **se combinan los resultados parciales** para dar solución al problema original.

Típicamente, puede plantearse con un algoritmo recursivo.

¿Ejemplos?

Los números de Fibonacci para $N \geq 0$

$$\text{Fibo}(N) = \begin{cases} N & \text{si } N \leq 1 \\ \text{Fibo}(N-1) + \text{Fibo}(N-2) & \text{si } N > 1 \end{cases}$$

Teorema Maestro

Si una fórmula recurrente puede expresarse genéricamente así:

$$T(N) = a + bT\left(\frac{N}{c}\right) + d$$

Donde:

Invocación recursiva que divide en subproblemas

Combinación de soluciones parciales

N es el tamaño de entrada del problema

$a \in N_{\geq 1}$ (¿cuántas invocaciones recursivas realiza ese paso?)

$b \in N_{>1}$ (mide tasa en que se reduce el tamaño del input)

$c \in R_{>0}$

$d \in R_{\geq 0}$

Si una fórmula recurrente puede expresarse genéricamente así:

$$T(N) = a * T\left(\frac{N}{b}\right) + c * N^d$$

Entonces la complejidad O grande está dada por los siguientes 3 casos (c no cuenta):

- Si $a < b^d$ entonces el algoritmo es $O(N^d)$
- Si $a = b^d$ entonces el algoritmo es $O(N^d * \log N)$
- Si $a > b^d$ entonces el algoritmo es $O(N^{\log_b a})$



El Teorema Maestro es una herramienta muy útil para resolver recurrencias.

Ejemplo 1: ¿Se podrá aplicar a Fibonacci?

$$\text{Fibo}(N) = \begin{cases} N & \text{si } N \leq 1 \\ \text{Fibo}(N-1) + \text{Fibo}(N-2) & \text{si } N > 1 \end{cases}$$

```
static public int fibo(int N) {  
    if (N <= 1)  
        return N;  
  
    return fibo(N-1) + fibo(N-2);  
}
```

O(1)

Rta. Times(N) = Times(N-1) + Times(N-2) + 4

$O(1)$

$$\text{Times}(N) = \text{Times}(N-1) + \text{Times}(N-2) + 4$$

¿Cuáles son las constantes a, b, c, y d? ¿Qué caso aplica? ¿Cuál es la complejidad O grande?

???

En general, la **Técnica Divide y Triunfarás** procede de la siguiente forma:

- Divide el problema en subproblemas de un **mismo tamaño**.
- Resuelve cada subproblema en forma independiente, por recursión
- Combina los resultados parciales para dar solución final.

Cuando esto ocurre, puede aplicarse el **Teorema Maestro**.

Ejemplo 2: ¿Se podrá aplicar a **búsqueda binaria** en arreglo ordenado?

Podríamos comenzar garantizando que todo llega bien al algoritmo de búsqueda binaria, pero el cálculo lo tenemos que aplicar al algoritmo recurrente (no acá):

```
static public int indexOf(int[] arreglo, int cantElementos, int elemento) {  
    if (cantElementos <= 0)  
        throw new IllegalArgumentException("cantidad de elementos debe ser positiva");  
  
    // chequear si esta ordenado y sino ordenarlo  
    // bla bla bla  
  
    return indexOf(arreglo, 0, cantElementos-1, elemento);  
}
```

La parte recurrente podría programarse así:

```
static private int indexOf(int[] arreglo, int izq, int der, int elemento) {  
    if (izq > der)  
        return -1;    // no estaba  
  
    // hay intervalo [izq, der]  
    int mid= (der + izq) / 2;  
  
    if (elemento == arreglo[mid] ) // lo encuentre  
        return mid;  
}
```

2 recursiones excluyentes.
El tamaño se divide a la mitad

}

$O(1)$

O sea, $\text{Times}(N) = \text{Times}(N/2) + 6$

¿Cuáles son las constantes a, b, c, y d? ¿Qué caso aplica? ¿Cuál es la complejidad O grande?

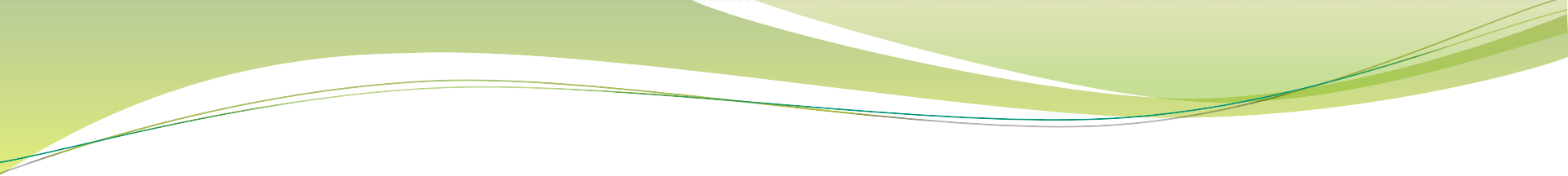
Entonces la complejidad O grande está dada por los siguientes 3 casos (c no cuenta):

- Si $a < b^d$ entonces el algoritmo es $O(N^d)$
- Si $a = b^d$ entonces el algoritmo es $O(N^d * \log N)$
- Si $a > b^d$ entonces el algoritmo es $O(N^{\log_b a})$

En búsqueda binaria ($a=1$, $b=2$, $d=0$), ¿Cuál de ellos aplica? ¿Cuál es la complejidad O grande?



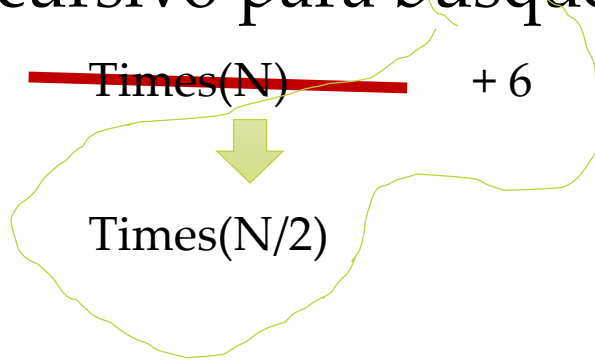
Hay otra alternativa al Master Theorem: el Algoritmo de Expansión recursiva.



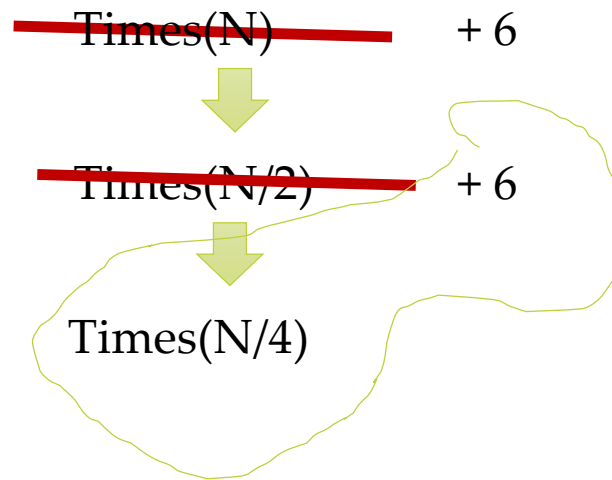
Otra forma de encontrar la complejidad del algoritmo
recursivo para búsqueda binaria:

Times(N)

Otra forma de encontrar la complejidad del algoritmo recursivo para búsqueda binaria:



Otra forma de encontrar la complejidad del algoritmo recursivo para búsqueda binaria:



Otra forma de encontrar la complejidad del algoritmo recursivo para búsqueda binaria:

~~Times(N)~~ + 6



~~Times(N/2)~~ + 6



~~Times(N/4)~~ + 6



Times(N/8)

Otra forma de encontrar la complejidad del algoritmo recursivo para búsqueda binaria:

~~Times(N)~~ + 6



~~Times(N/2)~~ + 6

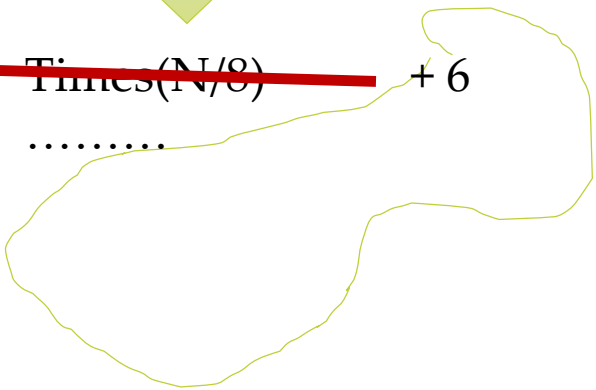


~~Times(N/4)~~ + 6



~~Times(N/8)~~ + 6

.....



Otra forma de encontrar la complejidad del algoritmo recursivo para búsqueda binaria:

~~Times(N)~~ + 6



~~Times(N/2)~~ + 6



~~Times(N/4)~~ + 6



~~Times(N/8)~~ + 6



Cuántas veces?.....



Times(1)

Otra forma de encontrar la complejidad del algoritmo recursivo para búsqueda binaria:

~~Times(N)~~ + 6



~~Times(N/2)~~ + 6



~~Times(N/4)~~ + 6



~~Times(N/8)~~ + 6



Cuántas veces?



Times(1)

Rta:

Step 0: Times($\frac{N}{2^0}$)

Step 1: Times($\frac{N}{2^1}$)

Step 2: Times($\frac{N}{2^2}$)

Step 3: Times($\frac{N}{2^3}$)

Ultimo step s: Times($\frac{N}{2^s}$) y eso es Times(1)

Entonces, como $\frac{N}{2^s} = 1$

Entonces, $N = 2^s$

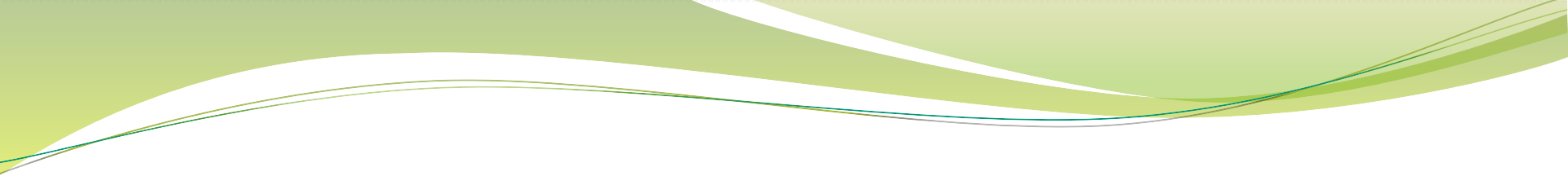
La cantidad de steps realizados es $\log_2 N$

$$\text{Times}(N) = \sum_{i=1}^{\log_2 N} 6$$

$$\text{Times}(N) = 6 * \log_2 N$$

..

El algoritmo es $O(\log_2 N)$



Ahora bien, ese cálculo lo hemos realizado partiendo de Times(N).

O sea, que lo importante es saber calcular Times(N) a partir de código.

Para el calculo de Times(N):

Cuando el código es **no-recursive** miramos las invocaciones, ciclos (paralelos vs anidados), etc.

Si el código es **recursive** hay que considerar la cantidad de invocaciones realizadas también.

Ejemplo de un código no recursivo (aca a N lo llamé dim)

```
static public int surprise(int[] arreglo, int dim) {  
    int vble= 0;  
    for (int rec = 0; rec < dim; rec++) {  
        for (int j = rec+1; j < dim; j++) {  
            if (arreglo[rec] * arreglo[j] == 0)  
                vble++;  
        }  
    }  
    return vble;  
}
```

$$\text{Times}(\text{dim}) = \sum_{\text{rec}=0}^{\text{dim}-1} (3 + \sum_{j=\text{rec}+1}^{\text{dim}-1} 5)$$

$$\text{Times}(\text{dim}) = \sum_{\text{rec}=0}^{\text{dim}-1} (3 + 5 (\text{dim} - 1 - (\text{rec} + 1) + 1))$$

$$\text{Times}(\text{dim}) = \sum_{\text{rec}=0}^{\text{dim}-1} (3 + 5 (\text{dim} - \text{rec} - 1))$$

$$\text{Times}(\text{dim}) = \sum_{rec=0}^{dim-1} (3 + 5 (\text{dim} - rec - 1))$$

$$\text{Times}(\text{dim}) = \sum_{rec=0}^{dim-1} (5 \text{ dim} - 5 rec - 2)$$

$$\text{Times}(\text{dim}) = \sum_{rec=0}^{dim-1} (5 \text{ dim} - 2) - \sum_{rec=0}^{dim-1} 5 rec$$

$$\text{Times}(\text{dim}) = (5 \text{ dim} - 2) \text{ dim} + 5 \sum_{rec=0}^{dim-1} rec$$

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

$$\text{Times}(\text{dim}) = (5 \text{ dim} - 2) \text{ dim} + 5 \frac{(\text{dim}-1) (\text{dim}-1+1)}{2}$$

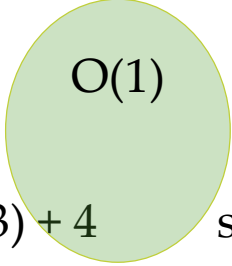
El algoritmo es $O(\text{dim}^2)$

Ejercicio

Sea el siguiente código

```
public static int surprise(int N) {  
    if (N < 4)  
        return 16;  
  
    int auxi1= surprise( N / 3);  
    int auxi2= surprise( N / 3);  
  
    return auxi1 + auxi2;  
}
```

Times(N) =


$$\text{Times}(N) = \begin{cases} 2 * \text{Times}(N/3) + 4 & \text{si } N \geq 4 \\ 1 & \text{si } N < 4 \end{cases}$$

Cuál es la O grande? Puedo aplicar el Teorema Maestro?

Ejercicio (variante)

Sea el siguiente código

```
public static int surprise(int N) {  
    if (N < 4)  
        return 16;  
  
    for (int i = 0; i < N; i++) {  
        System.out.println(i);  
    }  
  
    int auxi1= surprise( N / 3);  
    int auxi2= surprise( N / 3);  
  
    return auxi1 + auxi2;  
}
```

Times(N) =

$$\text{Times}(N) = \begin{cases} 2 * \text{Times}(N/3) + O(N) & \text{si } N \geq 4 \\ 1 & \text{si } N < 4 \end{cases}$$

Cuál es la O grande? Puedo aplicar el Teorema Maestro?

Resumiendo

Existen diferentes formas de calcular complejidad.
Hay que calcular correctamente $T(n)$.

Para el caso concreto de las recurrentes:

- Si aplican las condiciones, podemos aplicar Teorema Maestro
- Se puede expandir el árbol de invocaciones.
- Etc.