Estructura de Datos y Algoritmos

ITBA 2024-Q2

En búsqueda binaria hablamos del cálculo de complejidad temporal para algoritmos recurrentes.

Ahora bien, la búsqueda binaria puede implementarse en forma recursiva (la que vimos) o iterativa.

Es decir, indexOf() de 4 parámetros puede ser recursiva o iterativa

```
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);
}
```

```java
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 encontre
        return mid;

    if (elemento < arreglo[mid]  )
        return indexOf( arreglo, izq, mid-1, elemento);

    return indexOf( arreglo, mid+1, der, elemento);

}
```

Versión recursiva

Versión iterativa

```java
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 encontre
        return mid;

    if (elemento < arreglo[mid]  )
        return indexOf( arreglo, izq, mid-1, elemento);

    return indexOf( arreglo, mid+1, der, elemento);

}
```

Versión recursiva

```java
static private int indexOf(int[] arreglo, int izq, int der, int elemento) {
    while (izq <= der) {
        // hay intervalo [izq, der]
        int mid= (der + izq) / 2;

        if (elemento == arreglo[mid]  )  // lo encontre
            return mid;

        if (elemento < arreglo[mid]  )
            der= mid-1;
        else
            izq=mid+1;
    }
    return -1;
}
```

Versión iterativa

¿Cuál es la complejidad temporal de la versión iterativa de búsqueda binaria?

```java
static private int indexOf(int[] arreglo, int izq, int der, int elemento) {
    while (izq <= der) {
        // hay intervalo [izq, der]
        int mid= (der + izq) / 2;

        if (elemento == arreglo[mid]  )  // lo encontre
            return mid;

        if (elemento < arreglo[mid]  )
            der= mid-1;
        else
            izq=mid+1;
    }
    return -1;
}
```

Rta: ???

¿Cuál es la complejidad temporal de la versión iterativa de búsqueda binaria?

```
static private int indexOf(int[] arreglo, int izq, int der, int elemento) {
    while (izq <= der) {
        // hay intervalo [izq, der]
        int mid= (der + izq) / 2;

        if (elemento == arreglo[mid]  )  // lo encontre
            return mid;

        if (elemento < arreglo[mid]  )
            der= mid-1;
        else
            izq=mid+1;
    }
    return -1;
}
```

Rta: La misma que en recursión. El ciclo se ejecuta s veces ($s = \log_2 N$) y se hacen 6 operaciones.

Si la versión iterativa y la recursiva tiene la misma complejidad temporal, habrá alguna ventaja de una frente a otra?

Rta.

???

Si la versión iterativa y la recursiva tiene la misma complejidad temporal, habrá alguna ventaja de una frente a otra?

Rta.

Sí. En la complejidad espacial.

Calculemos.

¿Cuál es la complejidad espacial de la versión iterativa de búsqueda binaria?

```
static private int indexOf(int[] arreglo, int izq, int der, int elemento) {
    while (izq <= der) {
        // hay intervalo [izq, der]
        int mid= (der + izq) / 2;

        if (elemento == arreglo[mid]  )  // lo encontre
            return mid;

        if (elemento < arreglo[mid]  )
            der= mid-1;
        else
            izq=mid+1;
    }
    return -1;
}
```

Rta: ???

¿Cuál es la complejidad espacial de la versión iterativa de búsqueda binaria?

```
static private int indexOf(int[] arreglo, int izq, int der, int elemento) {
    while (izq <= der) {
        // hay intervalo [izq, der]
        int mid= (der + izq) / 2;

        if (elemento == arreglo[mid]  )  // lo encontre
            return mid;

        if (elemento < arreglo[mid]  )
            der= mid-1;
        else
            izq=mid+1;
    }
    return -1;
}
```

Rta: O(1)

¿Cuál es la complejidad espacial de la versión recursiva de búsqueda binaria? ¿Cuántos stack frames se generan? ¿Cuánto espacio se reserva dentro?

```
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 encontre
        return mid;

    if (elemento < arreglo[mid]  )
        return indexOf( arreglo, izq, mid-1, elemento);

    return indexOf( arreglo, mid+1, der, elemento);

}
```

Rta: ???

¿Cuál es la complejidad espacial de la versión recursiva de búsqueda binaria? ¿Cuántos stack frames se generan? ¿Cuánto espacio se reserva dentro?

```
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 encontre
        return mid;

    if (elemento < arreglo[mid]  )
        return indexOf( arreglo, izq, mid-1, elemento);

    return indexOf( arreglo, mid+1, der, elemento);

}
```

Rta: Se realizan $\log_2 N$ *steps o sea* $O(\log_2 N)$

Discusión

En nuestra implementación del índice precisamos de un arreglo ordenado. Invocamos el Arrays.sort() de java.

Analicemos qué métodos hay para ordenar arreglos.

Ordenación de Arreglos

Método "Quicksort"

Opera *in-place.*

Aplica la técnica Divide & Conquer.

Puede implementarse recursivamente o iterativamente.

Particiona en sub arreglos.

- En cada sub-arreglo elige un pivot y ordena para que todos los elementos a la izquierda del pivot sean menores que él y los de la derecha sean mayores que él => el pivot está en la posición correcta.
- Si un sub-arreglo tiene 0 o 1 elemento, está ya ordenado (no continua) => fin de la recurrencia

- Ejemplo: tomando como pivote primer elemento

Pivot “34”

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

- Ejemplo: tomando como pivote primer elemento

Pivot “34”

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

Particiona la lista <= 34 y >34 (o bien <34 y >=34)

- Ejemplo: tomando como pivote primer elemento

Pivot "34"

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

Particiona la lista <= 34 y >34 (o bien <34 y >=34)

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

Pos 8

El único que está seguro en el lugar es el 34!

Ahora hacer lo mismo con las 2 sub-arreglos por separado

- Para la lista derecha: pivot 60

10	8	21	17	28	30	2	15	34	60	70	50	62	40

Particiona la lista derecha<= 60 y >60 (o bien <60 y >=60)

- Para la lista derecha: pivot 60

10	8	21	17	28	30	2	15	34	60	70	50	62	40

Particiona la lista derecha<= 60 y >60 (o bien <60 y >=60)

10	8	21	17	28	30	2	15	34	50	40	60	62	70

Ahora también el 60 está en su lugar.

Ahora hacer lo mismo con las 2 sub-arreglos por separado.

10	8	21	17	28	30	2	15	34	50	40	60	62	70

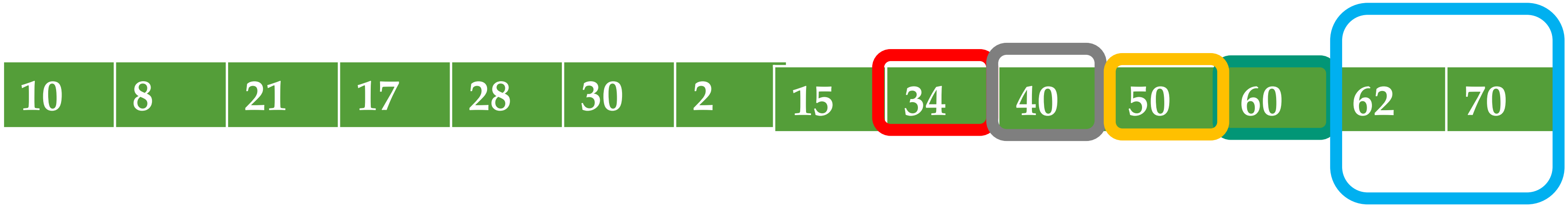
10	8	21	17	28	30	2	15	34	50	40	60	62	70

10	8	21	17	28	30	2	15	34	40	50	60	62	70

10	8	21	17	28	30	2	15	34	40	50	60	62	70

10	8	21	17	28	30	2	15	34	40	50	60	62	70

10	8	21	17	28	30	2	15	34	40	50	60	62	70



Así siguiendo. Finalmente todos quedan ordenados.

- Implementar **version recursiva** “quicksort”, para la version int[].
- Chequear su correctitud.
- Calcular la complejidad espacial y temporal para el peor y mejor caso.

Posible implementación.

```java
public static void quicksort(int[] unsorted) {
    quicksort (unsorted, unsorted.length-1);
}


public static void quicksort(int[] unsorted, int cantElements) {
    quicksortHelper (unsorted, 0, cantElements);
}
```

```java
static private void swap(int[] unsorted, int pos1, int pos2) {
    int auxi= unsorted[pos1];
    unsorted[pos1]= unsorted[pos2];
    unsorted[pos2]= auxi;
}
```

```java
private static void quicksortHelper (int[] unsorted, int leftPos, int rightPos) {
    if (rightPos <= leftPos )
        return;
```

```java
}
```

```java
static private void swap(int[] unsorted, int pos1, int pos2) {
    int auxi= unsorted[pos1];
    unsorted[pos1]= unsorted[pos2];
    unsorted[pos2]= auxi;
}
```

```java
private static void quicksortHelper (int[] unsorted, int leftPos, int rightPos) {
    if (rightPos <= leftPos )
        return;

    // tomamos como pivot el primero. Podria ser otro elemento
    int pivotValue= unsorted[leftPos];

    // excluimos el pivot del cjto.
    swap(unsorted, leftPos, rightPos);
```

```java
}
```

```java
static private void swap(int[] unsorted, int pos1, int pos2) {
    int auxi= unsorted[pos1];
    unsorted[pos1]= unsorted[pos2];
    unsorted[pos2]= auxi;
}
```

```java
private static void quicksortHelper (int[] unsorted, int leftPos, int rightPos) {
    if (rightPos <= leftPos )
        return;

    // tomamos como pivot el primero. Podria ser otro elemento
    int pivotValue= unsorted[leftPos];

    // excluimos el pivot del cjto.
    swap(unsorted, leftPos, rightPos);

    // particionar el cjto sin el pivot
    int pivotPosCalculated= partition(unsorted, leftPos, rightPos-1, pivotValue);

}
```

```java
static private void swap(int[] unsorted, int pos1, int pos2) {
    int auxi= unsorted[pos1];
    unsorted[pos1]= unsorted[pos2];
    unsorted[pos2]= auxi;
}
```

```java
private static void quicksortHelper (int[] unsorted, int leftPos, int rightPos) {
    if (rightPos <= leftPos )
        return;

    // tomamos como pivot el primero. Podria ser otro elemento
    int pivotValue= unsorted[leftPos];

    // excluimos el pivot del cjto.
    swap(unsorted, leftPos, rightPos);

    // particionar el cjto sin el pivot
    int pivotPosCalculated= partition(unsorted, leftPos, rightPos-1, pivotValue);


    // el pivot en el lugar correcto
    swap(unsorted, pivotPosCalculated, rightPos);

}
```

```java
static private void swap(int[] unsorted, int pos1, int pos2) {
    int auxi= unsorted[pos1];
    unsorted[pos1]= unsorted[pos2];
    unsorted[pos2]= auxi;
}
```

```java
private static void quicksortHelper (int[] unsorted, int leftPos, int rightPos) {
    if (rightPos <= leftPos )
        return;

    // tomamos como pivot el primero. Podria ser otro elemento
    int pivotValue= unsorted[leftPos];

    // excluimos el pivot del cjto.
    swap(unsorted, leftPos, rightPos);

    // particionar el cjto sin el pivot
    int pivotPosCalculated= partition(unsorted, leftPos, rightPos-1, pivotValue);


    // el pivot en el lugar correcto
    swap(unsorted, pivotPosCalculated, rightPos);


    // salvo unsorted[middle] todo puede estar mal
    // pero cada particion es autonoma
    quicksortHelper(unsorted, leftPos, pivotPosCalculated - 1);
    quicksortHelper(unsorted, pivotPosCalculated + 1, rightPos );

}
```

Implementar método Partition. Tiene que resolverse con complejidad espacial O(1)

Es decir, en ese arreglo el pivot 34.

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

Lo mandamos al fondo y lo excluimos hasta saber a dónde va. Invocamos a Partition sin el último.

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

Al volver de la invocación del 0..7 los <=34, del 8..12 los >34

													34

Implementar método Partition. Tiene que resolverse con complejidad espacial O(1)

Es decir, en ese arreglo el pivot 34.

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

Lo mandamos al fondo y lo excluimos hasta saber a dónde va. Invocamos a Partition sin el último.

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

Al volver de la invocación del 0..7 los <=34, del 8..12 los >34

													34

En pos=8 tendría que estar el 34

Implementarlo !

Posible solución

```java
static private int partition(int[] unsorted, int leftPos, int rightPos, int pivotValue) {
    while (leftPos <= rightPos) {

        while (leftPos <= rightPos && unsorted[leftPos] < pivotValue)
            leftPos++;

        while (leftPos <= rightPos && unsorted[rightPos] > pivotValue)
            rightPos--;

        if (leftPos <= rightPos)
            swap(unsorted, leftPos++, rightPos--);
    }
    return leftPos;
}
```

Calculando Complejidad Temporal de Quicksort

¿Cuál es el peor caso?

Rta ???

¿Se puede aplicar el Master Theorem para el peor caso?
Rta ???

Calculando Complejidad Temporal de Quicksort

¿Cuál es el peor caso?

Rta Que esté todo ordenado!!!

¿Se puede aplicar el Master Theorem para el peor caso?
Rta
No. Hay 2 invocaciones recursivas pero no en partes iguales.

Usemos otra forma de calcular complejidad temporal para el peor caso

Times(N) =

= N + Times(N-1)

….

Usemos otra forma de calcular complejidad temporal para el peor caso

Times(N) =

= N + Times(N-1)

= N + (N-1) + Times(N-2)

….

Usemos otra forma de calcular complejidad temporal para el peor caso

Times(N) =
= N + Times(N-1)
= N + (N-1) + Times(N-2)
= N + (N-1) + (N-2) + Times(N-3)
….

Usemos otra forma de calcular complejidad temporal para el peor caso

$$
\begin{aligned}
\text{Times}(N) &= \\
&= N + \text{Times}(N-1) \\
&= N + (N-1) + \text{Times}(N-2) \\
&= N + (N-1) + (N-2) + \text{Times}(N-3) \\
&\;\ldots. \\
&= N + (N-1) + (N-2) + \ldots. + 3 + \text{Times}(2) \\
&= N + (N-1) + (N-2) + \ldots. + 3 + 2 + \text{Times}(1) \\
&= N + (N-1) + (N-2) + \ldots. + 3 + 2 + 1
\end{aligned}
$$

Usemos otra forma de calcular complejidad temporal para el peor caso

Times(N) =

= N + Times(N-1)

= N + (N-1) + Times(N-2)

= N + (N-1) + (N-2) + Times(N-3)

….

= N + (N-1) + (N-2) + …. + 3 + Times(2)

= N + (N-1) + (N-2) + …. + 3 + 2 + Times(1)

= N + (N-1) + (N-2) + …. + 3 + 2 + 1

Rta: Times(N)= $\sum_{i=1}^{N} i$ o sea O(N^2)

Quicksort está diseñado para, en el mejor de los casos, tener listas de tamaño mitad en cada iteración. Si eso se lograra, entonces

La complejidad del algoritmo recursivo para mejor caso :

Times(N)

$\cancel{\text{Times(N)}}$ + N

Times(N/2) Times(N/2)

~~Times(N)~~ + N

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

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

Times(N/4) Times(N/4)

Times(N/4) Times(N/4)

~~Times(N)~~ + N

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

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

Times(N/8) Times(N/8) | Times(N/8) Times(N/8) | Times(N/8) Times(N/8) | Times(N/8) Times(N/8)

¿Cuántas veces?

~~Times(N)~~ + N

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

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

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

……… ……………… ……………… ……………………… ……………… ……… ……… ………..

Times(1) Times(1) ……………. Times(1) Times(1) ………. ………….. …. … Times(1) Times(1)

¿Cuántas veces?

Rta: termina en el paso s
donde Times(1) es Times($N/2^s$)
o sea $1 = N/2^s$
La cantidad de steps realizados s es
$\log_2 N$

~~Times(N)~~ + N

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

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

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

……… ……………… ……………… ………………………… ………………… …………… ………

Times(1) Times(1) ……………. Times(1) Times(1) ………. …………. …. … Times(1) Times(1)

¿Qué pinta tiene lo que se hace en cada paso?

~~Times(N)~~ + N

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

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

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

……… …………….. …………… ………… ……… ………….. ………. ……. ……….

Times(1) Times(1) ……………. Times(1) Times(1) ………. ………….. …. … Times(1) Times(1)

¿Qué pinta tiene lo que se hace en cada paso?

Rta: paso 0 $2^0 * N/2^0 = N$
paso 1 $2^1 * N/2^1 = N$
paso 2 $2^2 * N/2^2 = N$
paso 3 $2^3 * N/2^3 = N$
…

~~Times(N)~~ + N

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

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

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

……… ……………… ………… ………………… ………… ……… ……… ……….

Times(1) Times(1) ……………. Times(1) Times(1) ……… ………… …. … Times(1) Times(1)

Times(N) = $\sum_{i=1}^{log2N} N$

Times(N)= N * $log2N$

El algoritmo es O(N log2 N)

Se puede Aplicar Master Theorem para mejor caso?
Como sería?

Se puede Aplicar Master Theorem para mejor caso?
Como sería?

Times(N) = 2 * Times(N/2) + O(N)

Se puede Aplicar Master Theorem para mejor caso?
Como sería?

Times(N) = 2 * Times(N/2) + O(N)

O sea, a= 2, b=2 y d = 1

Finalmente, es el caso dos, o sea $O(N^{d} * \log N)$
O sea $O(N * \log N)$

¿Cómo mejorar a quicksort para que cuando venga casi ordenado no de tan mal?

Rta

????

¿Cómo mejorar a quicksort para que cuando venga casi ordenado no de tan mal?

Rta

Cambiar el Pivot. Ej: tomar el elemento del medio, un elemento random, tomar la mediana de 3 elementos candidatos predeterminados, etc.

Complejidad espacial?

Rta

???

Complejidad espacial?

Rta

En el peor caso, debido a los stackframes tenemos

O(N)

Complejidad espacial?

Rta

En el peor caso, debido a los stackframes tenemos O(N)

En el mejor caso, debido a los stackframes tenemos O(log2 N)

Tarea:

Buscar algoritmo Mergesort, estudiarlo por cuenta de uds. e implementarlo (puede ser el que no opera in situ)
Analizar complejidad espacial y temporal.

Tarea:

Usa Java alguno de esos métodos?
¿Qué complejidad tiene?

TP 3A- Ejer 3

Por cuenta de Uds, terminar el ejer 3.