MEMORIA PRACTICA 4

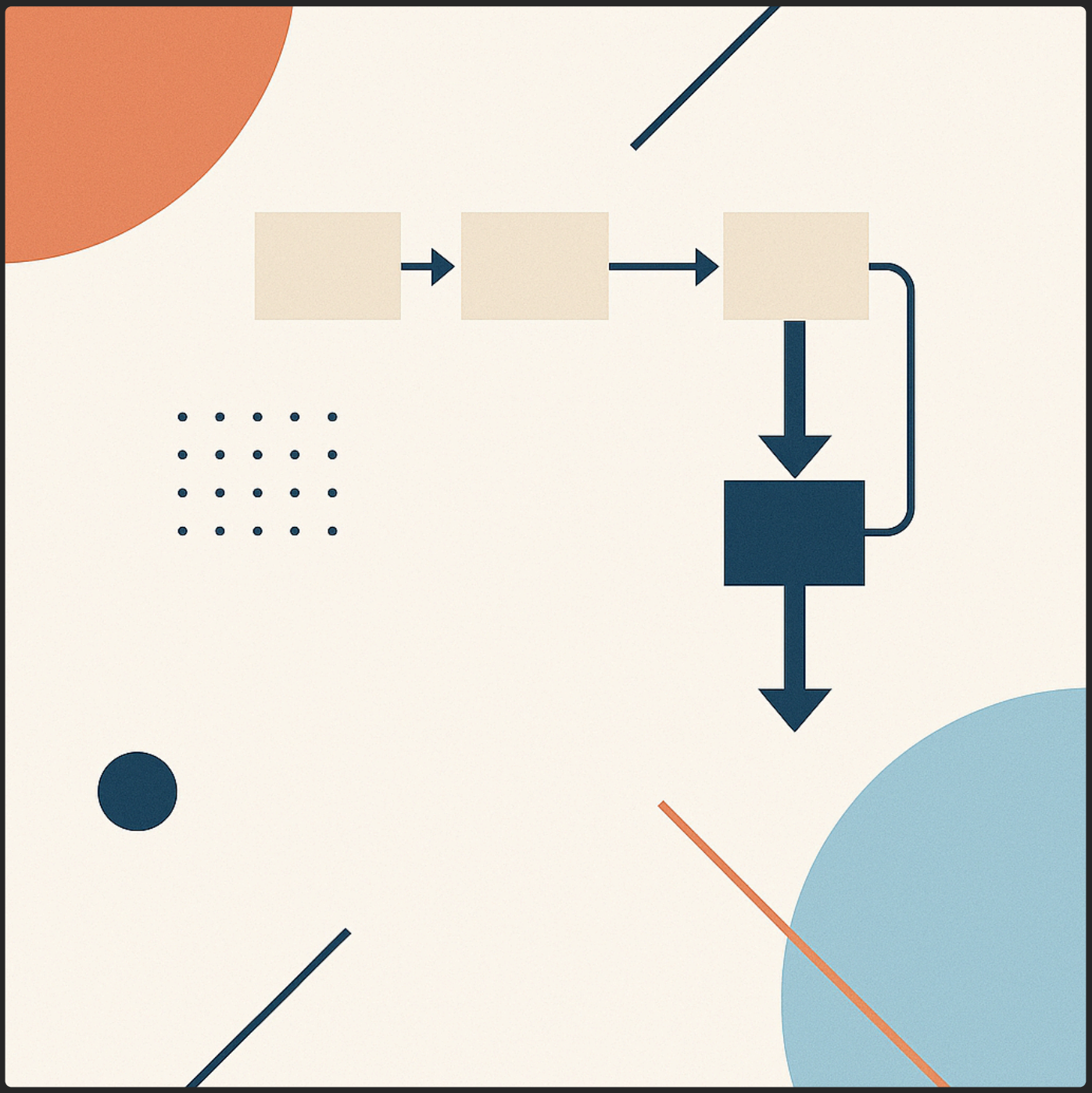

Tabla de contenido

Análisis teórico

En esta práctica, bastante más completa que las anteriores a mi parecer, vamos a estudiar de manera tanto teórica como empírica la complejidad de varios algoritmos usando distintas técnicas.
Vamos a empezar por el conteo de operaciones elementales y de ahí sacaremos la función

Comportamiento variantes

Búsqueda Binaria 1

```
int AlgoritmosBusqueda::busquedaBinaria1(int A[], int izq, int der, int x)
{
    // Excepción. Realmente no forma parte de la búsqueda como tal
    if (izq > der) // 1 OE comparación
        return -1;

    int medio = (izq + der) / 2; // 3 OE -> Asignación y 2 operaciones aritméticas
    if (A[medio] == x){ // 2 OE -> acceso al vector y comparación
        return medio;
    }

    else if (A[medio] > x){ // 2 OE -> acceso al vector y comparación
        return busquedaBinaria1(A, izq, medio - 1, x); // 2 OE -> llamada a
función y operación aritmética + Recursividad
    }

    else{
        return busquedaBinaria1(A, medio + 1, der, x); // 2 OE -> llamada a
función y operación aritmética + Recursividad
    }

    // 1 OE de cualquier return en cualquier caso. Arriba no las tuve en cuenta
para hacerlo aquí
}
```

Conteo Operaciones Elementales

Con el conteo de operaciones elementales obtenemos lo siguiente:

Primer caso base

Este caso se puede dar en 3 situaciones:

- Actúa de early return en caso de que el usuario pase valores incorrectos en la primera llamada al algoritmo

- Si llegamos a un punto de recursividad en el que el elemento no ha sido encontrado aún y se sigue llamando al algoritmo. Este último caso sería cuando el elemento no se encuentra en el array.
- Si el array es de tamaño 1. Este caso puede darse en la situación anterior o en la primera llamada a la función.
  La primera situación la obviaremos para el estudio del coste, ya que no forma parte del algoritmo como tal, más bien es un método de prevención de errores.

```
if (izq > der) // 1 OE comparación
        return -1; // 1 OE salto
```

En este caso base se realizan `2 OE`

- `1 OE` Op. lógica
- `1 OE` Salto
  Por tanto en este caso tenemos coste constante $\rightarrow c_{noEnc}$

Segundo caso base

```
if (izq > der) // 1 OE comparación
...

int medio = (izq + der) / 2; // 3 OE -> Asignación y 2 operaciones aritméticas
if (A[medio] == x){ // 2 OE -> acceso al vector y comparación
    return medio; // 1 OE salto
}
```

Este caso base se da cuando el elemento se encuentra en la posición media del array actual (ya sea en la primera llamada a la función o en alguna de las llamadas recursivas).
En este caso base se realizan `7 OE`

- `2 OE` Op. lógicas
- `1 OE` Acceso al vector
- `2 OE` Op. aritméticas
- `1 OE` Salto
- `1 OE` Asignación
  Por tanto en este caso tenemos coste constante $\rightarrow c_{enc}$

Tercer caso

```
if (izq > der) // 1 OE comparación
...
int medio = (izq + der) / 2; // 3 OE -> Asignación y 2 operaciones aritméticas
if (A[medio] == x) // 2 OE -> acceso al vector y comparación
...
else if (A[medio] > x){ // 2 OE -> acceso al vector y comparación
    return busquedaBinaria1(A, izq, medio - 1, x); // 3 OE -> llamada a función,
operación aritmética y salto +  coste Recursividad
}
```

Este caso se da cuando el elemento es mayor al elemento medio del array actual (Si el elemento se encuentra en el array, está en la parte derecha).
En este caso base se realizan `11 OE`

- `3 OE` Op. lógicas
- `2 OE` Acceso al vector
- `3 OE` Op. aritméticas
- `2 OE` Salto -> return y llamada a función
- `1 OE` Asignación
  Por tanto en este caso tenemos coste constante + Coste de la función llamada
  $\rightarrow c_{dcha} + T(\frac{n}{2})$

Cuarto caso

```
if (izq > der) // 1 OE comparación
...
int medio = (izq + der) / 2; // 3 OE -> Asignación y 2 operaciones aritméticas
if (A[medio] == x){ // 2 OE -> acceso al vector y comparación
...
else if (A[medio] > x){ // 2 OE -> acceso al vector y comparación
...
else{
    return busquedaBinaria1(A, medio + 1, der, x); // 3 OE -> llamada a función,
operación aritmética y salto +  coste Recursividad
}
```

Este caso se da cuando el elemento es mayor al elemento medio del array actual (Si el elemento se encuentra en el array, está en la parte derecha).
En este caso base se realizan `11 OE`

- `3 OE` Op. lógicas
- `2 OE` Acceso al vector
- `3 OE` Op. aritméticas
- `2 OE` Salto -> return y llamada a función
- `1 OE` Asignación
  Por tanto en este caso tenemos coste constante + Coste de la función llamada
  $\rightarrow c_{izda} + T(\frac{n}{2})$

Conclusión

Todos los costes constantes podemos agruparlos/simplificarlos, obteniendo C
Como conclusión de los cuatro casos tenemos lo siguiente:

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + C & \text{si } n > 1 \\ c_{base} & \text{si } n = 1 \text{ (tamaño base)} \end{cases}$$

Mejor caso

Como podemos observar, el mejor caso se da cuando el tamaño del array inicial es 1, tanto si el elemento a buscar coincide como si no, el algoritmo devuelve `-1` y su coste sería constante, de únicamente `2 OE`.
$\mathrm{T(n)} = c_{noEnc}$

Si no contempláramos la posibilidad de que el array pasado pueda ser de un elemento, el mejor caso pasaría a ser el Segundo caso base.
Al completarse en la primera llamada al método, se realizarían solo `7 OE` y en este caso la complejidad sería constante, dando igual la talla del vector.
$\mathrm{T(n)} = c_{enc}$

Contemplemos (o no) esa opción, el coste del algoritmo en el mejor caso seguiría siendo constante en ambos supuestos.

Solución

$T(n) = c_{base}$

$\text{Mejor caso } \Theta \rightarrow (1)$

Peor caso

Por lo que hemos visto más arriba, podemos decir que el peor caso se daría cuando la recursividad tiene que repetirse hasta llegar al tamaño base y aún así no se encuentra el elemento.
Como en esta recursividad, en cada paso, el tamaño del array se reduce a la mitad.
Por tanto, tenemos:

$$T(n) = T\left(\frac{n}{2}\right) + C_{peor}$$

Aplicando reducción por división:

Reducción por división

$$T(n) = \begin{cases} c & \text{Si } 1 \leq n < b \\ a \cdot T\left(\frac{n}{b}\right) + f(n) & \text{Si } n \geq b \end{cases}$$

- c es el coste en el caso directo
- a es el número de llamadas recursivas
- b factor de disminución del tamaño de los datos
- $f(n) = C \cdot n^k$ es el coste de preparación de las llamadas y de combinación de los resultados.

donde se tiene:

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k \log n) & \text{si } a = b^k \\ \Theta(n^{\log_b a}) & \text{si } a > b^k \end{cases}$$

en nuestro caso:

$$\begin{cases} a = 1 \\ b = 2 \\ f(n) = C \cdot n^k \rightarrow n^k = 1 \rightarrow k = 0 \end{cases}$$

Solución

$$T(n) \in \Theta(n^k \cdot \log n) = \Theta(n^0 \cdot \log n) =$$

$$\text{Peor caso } \rightarrow \Theta(\log n)$$

Caso medio

Para la esta búsqueda binaria, el caso medio tiene un comportamiento muy similar al peor caso. En promedio, también se divide el problema por la mitad en cada paso. El número de pasos para encontrar un elemento (o determinar que no está) es, en promedio, también proporcional al logaritmo del tamaño del array.
Por tanto, tenemos:

$$T(n) = T\left(\frac{n}{2}\right) + C_{medio}$$

Al igual que en peor caso, la solución sería:

> **Solución**
>
> $$\text{Caso medio} \rightarrow \Theta(\log n)$$

Búsqueda Binaria 2

Esta búsqueda es prácticamente idéntica a la anterior, la principal diferencia radica en el segundo caso base, que ahora también debe cumplir que si el punto medio es el elemento a buscar debe ser la primera ocurrencia de este.

```
int AlgoritmosBusqueda::busquedaBinaria2(int A[], int izq, int der, int x){

    if (izq > der){ // 1 OE comparación
        return -1;
    }

    int medio = (izq + der) / 2; // 3 OE -> Asignación y 2 operaciones aritméticas
    if (A[medio] == x && (medio == izq || A[medio - 1] != x)){
        //  8 OE -> 2 accesos al vector, 5 op lógicas, 1 op aritmética
        return medio;
    }
    else if (A[medio] >= x){ // 2 OE -> acceso al vector y op. lógica
        return busquedaBinaria2(A, izq, medio - 1, x); // 2 OE -> llamada a
función y operación aritmética + Recursividad
    }
    else{
        return busquedaBinaria2(A, medio + 1, der, x); // 2 OE -> llamada a
función y operación aritmética + Recursividad
    }
}
    // 1 OE de cualquier return en cualquier caso. Arriba no las tuve en cuenta
para hacerlo aquí
}
```

Conteo Operaciones Elementales

Primer caso base

Exactamente igual que en Búsqueda binaria 1.

Primer caso base

Este caso se puede dar en 3 situaciones:

- Actúa de early return en caso de que el usuario pase valores incorrectos en la primera llamada al algoritmo
- Si llegamos a un punto de recursividad en el que el elemento no ha sido encontrado aún y se sigue llamando al algoritmo. Este último caso sería cuando el elemento no se encuentra en el array.
- Si el array es de tamaño 1. Este caso puede darse en la situación anterior o en la primera llamada a la función.
  La primera situación la obviaremos para el estudio del coste, ya que no forma parte del algoritmo como tal, más bien es un método de prevención de errores.

```
if (izq > der) // 1 OE comparación
        return -1; // 1 OE salto
```

En este caso base se realizan `2 OE`

- `1 OE` Op. lógica
- `1 OE` Salto
  Por tanto en este caso tenemos coste constante $\rightarrow c_{noEnc}$

Segundo caso base

Exactamente igual que en Búsqueda binaria 1 pero con la diferencia de que el elemento central del array actual debe ser la primera ocurrencia.
Esta pequeña diferencia incrementa las operaciones elementales pero mantiene el coste de este caso constante. Por tanto $\rightarrow c_{enc}$

```
if (izq > der){ // 1 OE op. lógica
...
int medio = (izq + der) / 2; // 3 OE -> Asignación y 2 operaciones aritméticas
if (A[medio] == x && (medio == izq || A[medio - 1] != x)){
    // 8 OE -> 2 accesos al vector, 5 op lógicas, 1 op aritmética
    return medio; // 1 OE salto
}
```

En este caso base se realizan `13 OE`

- `6 OE` Op. lógicas
- `1 OE` Asignación
- `1 OE` Salto
- `2 OE` Acceso al vector
- `3 OE` Op. aritméticas

Tercer caso

Exactamente igual que en Búsqueda binaria 1, aunque el código difiera en el op. lógico, el resto es igual.

```
if (izq > der){ // 1 OE op. logica
...
int medio = (izq + der) / 2; // 3 OE -> Asignación y 2 operaciones aritméticas
if (A[medio] == x && (medio == izq || A[medio - 1] != x)){
    //  8 OE -> 2 accesos al vector, 5 op lógicas, 1 op aritmética
...
else if (A[medio] >= x){ // 2 OE -> acceso al vector y op. lógica
    return busquedaBinaria2(A, izq, medio - 1, x); // 3 OE -> llamada a función,
operación aritmética y salto(return) + Recursividad
}
```

En este caso se realizan `17 OE`

- `7 OE` Op. lógicas
- `1 OE` Asignación
- `2 OE` Salto
- `3 OE` Acceso al vector
- `4 OE` Op. aritméticas

Por tanto en este caso tenemos coste constante + Coste de la función llamada
$\rightarrow c_{izda} + T(\frac{n}{2})$

Cuarto caso

Exactamente igual que en Búsqueda binaria 1, aunque el código difiera en el op. lógico, el resto es igual.

```
if (izq > der){ // 1 OE op. logica
...
int medio = (izq + der) / 2; // 3 OE -> Asignación y 2 operaciones aritméticas
if (A[medio] == x && (medio == izq || A[medio - 1] != x)){
    //  8 OE -> 2 accesos al vector, 5 op lógicas, 1 op aritmética
...
else if (A[medio] >= x){ // 2 OE -> acceso al vector y op. lógica
...
else{
    return busquedaBinaria2(A, medio + 1, der, x); // 3 OE -> llamada a función,
operación aritmética y salto + Recursividad
}
```

En este caso se realizan `17 OE`

- `7 OE` Op. lógicas
- `1 OE` Asignación
- `2 OE` Salto
- `3 OE` Acceso al vector
- `4 OE` Op. aritméticas

Por tanto en este caso tenemos coste constante + Coste de la función llamada
$\rightarrow c_{dcha} + T(\frac{n}{2})$

Conclusión

Todos los costes constantes podemos agruparlos/simplificarlos, obteniendo C
Como conclusión de los cuatro casos tenemos lo siguiente:

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + C & \text{si } n > 1 \\ c_{base} & \text{si } n = 1 \text{ (tamaño base)} \end{cases}$$

Mejor caso

Al igual que el algoritmo anterior, el mejor caso se da cuando se cumple la condición de uno de los dos `if` primeros en la primera llamada a la función. Básicamente, cuando el tamaño del array es 1 o cuando el elemento se encuentra en la posición media del array inicial y, además, es la primera ocurrencia. Ambos son de coste constante, por tanto:

Solución

$T(n) = c_{base}$

$\text{Mejor caso} \rightarrow \Theta(1)$

Peor caso

El peor caso es exactamente igual al de Búsqueda binaria 1.
Por tanto:

Solución

$$T(n) \in \Theta(n^k \cdot \log n) = \Theta(n^0 \cdot \log n) =$$

$$\text{Peor caso } \rightarrow \Theta(\log n)$$

Caso medio

El caso medio también es exactamente igual al de Búsqueda binaria 1.
Por tanto:

Solución

$$\text{Caso medio } \rightarrow \Theta(\log n)$$

Búsqueda por interpolación

A diferencia de la búsqueda binaria estándar que siempre comprueba el medio, la búsqueda por interpolación estima la posición de x basándose en su valor relativo a los límites, asumiendo una distribución algo uniforme.

```
int AlgoritmosBusqueda::busquedaBinariaInterpolacion(int A[], int izq, int der, int x){

    if (izq > der || x < A[izq] || x > A[der]){ //7 OE -> 5 Op. lógicas y 2 accesos a vector
        return -1;
    }

    int pos = izq + ((x - A[izq]) * (der - izq)) / (A[der] - A[izq]); //10 OE -> 1 Asignacion, 6 Op. aritméticas, 3 Accesos vector
    if (pos < izq || pos > der) // 3 OE -> 3 Op. lógicas
        pos = (izq + der) / 2; // 3 OE -> 1 Asignación, 2 Operación aritmética

    if (A[pos] == x){ // 2 OE -> 1 Acceso vector y 1 Op. lógica
        return pos;
    }
    else if (A[pos] > x){ // 2 OE -> 1 Acceso vector y 1 Op. lógica
        return busquedaBinariaInterpolacion(A, izq, pos - 1, x); // 2 OE -> 1 salto y 1 op. aritmética
    }
    else{
        return busquedaBinariaInterpolacion(A, pos + 1, der, x); // 2 OE -> 1 salto y 1 op. aritmética
    }
    // 1 OE de cualquier return en cualquier caso. Arriba no las tuve en cuenta para hacerlo aquí

}
```

Conteo Operaciones Elementales

Primer caso base

Prácticamente igual a los anteriores, tiene más consideraciones pero sigue siendo coste constante. Por tanto $\rightarrow c_{noEnc}$

```
if (izq > der || x < A[izq] || x > A[der]){ //7 OE
    return -1; // 1 OE
}
```

En este caso base se realizan `8 OE`

- `5 OE` Op. lógicas
- `1 OE` Salto
- `2 OE` Acceso al vector

Segundo caso base

Prácticamente igual a las búsquedas binarias anteriores. Por tanto $\rightarrow c_{enc}$

```
if (izq > der || x < A[izq] || x > A[der]){ //7 OE
...
int pos = izq + ((x - A[izq]) * (der - izq)) / (A[der] - A[izq]); //10 OE

if (pos < izq || pos > der) // 3 OE
    pos = (izq + der) / 2; // 3 OE

if (A[pos] == x){ // 2 OE
    return pos; // 1 OE
}
```

En este caso base se realizan `26 OE`

- `9 OE` Op. lógicas
- `1 OE` Salto
- `6 OE` Acceso al vector
- `2 OE` Asignación
- `8 OE` Op. aritméticas

Tercer caso

Prácticamente igual a las búsquedas binarias anteriores.

```
if (izq > der || x < A[izq] || x > A[der]){ //7 OE
...

int pos = izq + ((x - A[izq]) * (der - izq)) / (A[der] - A[izq]); //10 OE
if (pos < izq || pos > der) // 3 OE
    pos = (izq + der) / 2; // 3 OE

if (A[pos] == x){ // 2 OE
...

else if (A[pos] > x){ // 2 OE
    return busquedaBinariaInterpolacion(A, izq, pos - 1, x); // 3 OE +
recursividad
}
```

En este caso se realizan `30 OE`

- `10 OE` Op. lógicas
- `2 OE` Salto
- `7 OE` Acceso al vector
- `2 OE` Asignación
- `9 OE` Op. aritméticas

Por tanto en este caso tenemos coste constante + Coste de la función llamada
$\rightarrow c_{izda} + T(\frac{n}{2})$

Cuarto caso

Prácticamente igual a las búsquedas binarias anteriores.

```
if (izq > der || x < A[izq] || x > A[der]){ //7 OE
...

int pos = izq + ((x - A[izq]) * (der - izq)) / (A[der] - A[izq]); //10 OE
if (pos < izq || pos > der) // 3 OE
    pos = (izq + der) / 2; // 3 OE

if (A[pos] == x){ // 2 OE
...

else if (A[pos] > x){ // 2 OE
...

else{
    return busquedaBinariaInterpolacion(A, pos + 1, der, x); // 3 OE +
recursividad
}
```

En este caso se realizan `30 OE`

- `10 OE` Op. lógicas
- `2 OE` Salto
- `7 OE` Acceso al vector
- `2 OE` Asignación
- `9 OE` Op. aritméticas

Por tanto en este caso tenemos coste constante + Coste de la función llamada
$\rightarrow c_{dcha} + T(\frac{n}{2})$

Conclusión

Todos los costes constantes podemos agruparlos/simplificarlos, obteniendo C
Como conclusión de los cuatro casos tenemos lo siguiente:

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + C & \text{si } n > 1 \\ c_{base} & \text{si } n = 1 \text{ (tamaño base)} \end{cases}$$

Mejor caso

El mejor caso se da cuando la estimación del algoritmo es correcta en la primera llamada al método y cumple `if (A[pos] == x)`. En este caso, el coste sería constante.

Al igual que con los anteriores, dependiendo de cómo contemplemos el primer if, también podría considerarse éste como mejor caso si:

- El array es tamaño n =1
- El elemento a buscar es menor que el primer elemento del array (no se encontraría en el array)
- El elemento a buscar es mayor que el último elemento del array (no se encontraría en el array)

No supone mucha importancia considerar uno como mejor caso, o incluso ambos, ya que los dos tienen coste constante y el mejor caso sería el mismo. Por tanto:

> **Solución**
>
> $T(n) = c_{base}$
>
> $\text{Mejor caso } \Theta \rightarrow (1)$

Peor caso

Ocurre cuando la distribución no es nada uniforme. Por ejemplo, cuando es exponencial. La estimación de pos es consistentemente incorrecta, lo que haría que la estimación haga que el algoritmo se comporte de forma similar a una búsqueda lineal.

Como ya hemos visto, en cada paso recursivo, el algoritmo realiza un número constante de operaciones y luego realiza una llamada recursiva sobre un sub-array.
El tamaño de este sub-array depende de la calidad de la estimación. En el peor caso, si la estimación siempre cae en un extremo, el problema se reduce en una cantidad constante, la relación de recurrencia quedaría algo así:

$$T(n) = T(n-c) + C_{peor}$$

Aplicando reducción por sustracción:

Reducción por sustracción

$$T(n) = \begin{cases} c & \text{Si } 0 \leq n < b \\ a \cdot T\,(n-b) + p(n) & \text{Si } n \geq b \end{cases}$$

- c es el coste en el caso directo
- a es el número de llamadas recursivas
- b disminución del tamaño de los datos
- $p(n) = C \cdot n^k$ es el coste de preparación de las llamadas y de combinación de los resultados.

donde se tiene:

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{\frac{n}{a}}) & \text{si } a > 1 \end{cases}$$

en nuestro caso:

$$\begin{cases} a = 1 \\ b = \ (\text{Constante. En cualquier caso, sería menor que n}) \\ p(n) = C \cdot n^k \rightarrow n^k = 1 \rightarrow k = 0 \end{cases}$$

Solución

$$T(n) \in \Theta(n^{k+1}) = \Theta(n^{0+1})$$

$$\text{Peor caso } \rightarrow \Theta(n)$$

Caso Medio

Cuando los datos están distribuidos de manera que la fórmula de interpolación hace buenas estimaciones en promedio, se reduce el rango de búsqueda mucho más rápido que dividiendo por la mitad.

Como esta estimación no va siempre al medio y tiende a aproximarse mejor al elemento correcto si está bien distribuido,

Este es el caso donde la Búsqueda por Interpolación puede ser significativamente más eficiente que la Búsqueda Binaria estándar. Si los datos en el array están distribuidos de manera **uniforme**, la fórmula de interpolación

$$pos = izq + \frac{(x - A[izq]) \cdot (der - izq)}{(A[der] - A[izq])}$$

tiende a calcular una posición `pos` que está, en promedio, mucho más cerca del elemento buscado de lo que estaría el simple punto medio. Esto significa que el tamaño del sub-array restante es, en promedio, considerablemente menor que la mitad del tamaño original.

La reducción del tamaño del problema en cada paso no es una división por una constante fija, sino que depende de la calidad de la estimación, que a su vez depende de la distribución de los datos. Para una distribución uniforme, esta reducción es, en promedio, muy efectiva.

En cada paso recursivo, el algoritmo realiza un número constante de operaciones no recursivas y luego hace una llamada recursiva a un subproblema cuyo tamaño se ha reducido de forma más drástica que en la búsqueda binaria estándar.

Debido a esto, no podemos resolverlo con las relaciones de recurrencia simples de los apuntes.

Buscando por internet se puede ver como, haciendo un análisis matemático más avanzado (cosa que yo no puedo hacer) que modela el comportamiento promedio, la complejidad sería:

Solución

$\text{Caso medio} \rightarrow \Theta(\log(\log n))$

Esta solución crece significativamente más lento que $\Theta(\log n)$

Comparación Búsqueda Binaria 2 y Búsqueda por interpolación

¿En qué situaciones BÚSQUEDABINARIAINTERPOLACIÓN sería significativamente más eficiente?

La búsqueda binaria por interpolación es más eficiente con respecto a la búsqueda binaria con índice de primera ocurrencia en los casos en los que la distribución de los elementos del array sea uniforme. Siempre y cuando obviemos la posibilidad de que se dé el mejor caso de la búsqueda con índice de ocurrencia.

¿En qué situaciones podría tener un rendimiento peor?

Cuando la distribución no sea nada uniforme. En este caso, la complejidad de búsqueda por interpolación sería de $\Theta(n)$ mientras que la búsqueda con índice de primera ocurrencia sería $\Theta(\log n)$. Siempre y cuando obviemos la posibilidad de que se dé el mejor caso de la búsqueda por interpolación.

¿Qué consideraciones numéricas hay que tener en cuenta en BÚSQUEDABINARIAINTERPOLACIÓN?

Hay que tener en cuenta que las estimaciones de la posición pueden dar un índice no válido. Por eso después de hacer la estimación, hay que comprobar que esté en rango y, si no lo está, asignar la posición media.

Mientras realizaba el estudio empírico, me di cuenta que en el cálculo de la estimación puede darse que el denominador de la división sea 0. Esto hay que controlarlo.

```
// Si la resta de los elementos extremos es 0. Crashea -> Hay que controlarlo
    int denominador = (A[der] - A[izq]);
    if( denominador == 0)
        denominador =1;
int pos = izq + ((x - A[izq]) * (der - izq)) / denominador; //10 OE -> 1
Asignacion, 6 Op. aritméticas, 3 Accesos vector
```

Debido a esto las OE cambian pero los casos constantes y recursivos siguen siendo iguales. Tiene 3 OE más.

Estudio empírico

Funcionamiento general del main

Reutilizando el código de las prácticas anteriores, he creado un main que:

1. Crea un archivo lógico al que asociamos un documento excel con extensión `csv`.
2. En un bucle que itera tantas veces como se indique, creamos un vector aleatorio de la talla inicial indicada. En bucle, la talla incrementa en cada iteración. Este incremento también puede indicarse. Dentro del bucle:
    1. Creamos un vector aleatorio
    2. Con este vector creado. Entramos en otro bucle para buscar tantos números aleatorios como se indique. En cada iteración se genera un número a buscar aleatorio.
    3. Se mide el tiempo de cada algoritmo buscando el número aleatorio.
    4. Se escribe en el archivo lógico los datos obtenidos.
3. Cerramos el archivo lógico.

Resultados empíricos

Vectores aleatorios

Con los resultados obtenidos, podemos comprobar que, de manera general, tanto el Algoritmo de Búsqueda Binaria como Algoritmo de Búsqueda Binaria con índice de primera ocurrencia suelen seguir la misma línea en cuanto a costes. Por otro lado, se puede ver como el algoritmo de Búsqueda por interpolación es bastante mejor en la mayoría de casos.

Elementos inexistentes

Este estudio no tiene una implementación concreta ya que se da de manera natural en la mayoría de estudios. Los elementos de búsqueda generados son, en mayor parte aleatorios, así que ya se estudian.

Verificación de que siempre devuelve la primera ocurrencia

En este estudio, generamos un vector aleatorio con elementos repetidos, igual que el estudio anterior de elementos repetidos.
La única diferencia es que nos quedamos con el índice del primer elemento que se repita en el vector, buscamos ese valor y comparamos si el índice devuelto es el mismo que el que habíamos guardado.
Esto lo hemos conseguido de la siguiente manera:
Cuando generamos el vector aleatorio, la primera vez que se genera un número repetido, nos quedamos con ese valor, después, una vez se haya generado el vector completo, lo ordenamos.
Una vez ordenado el vector, con el algoritmo de búsqueda secuencial (iterar desde el inicio del vector hasta encontrar el elemento) buscamos el valor que habíamos guardado y nos quedamos con su índice.
Cuando comprobamos el algoritmo de búsqueda binaria con índice de primera ocurrencia, comprobamos que la posición devuelta coincida con la posición guardada anteriormente.

Código de la generación del vector de manera resumida

```
vector<int> generarVectorAleatorioConRepes(int tallaVec,int
&indicePrimerElementoRepetido)
{
    vector<int> vec(n);
    bool repetido = false; // para guardar el primer elemento repetido
    for (int i = 0; i < tallaVec; ++i)
    {


        if(generarDuplicado){ // Condicion para generar un aleatorio nuevo o
repetir
            vec[i] = vec[i-1]; // Repetimos el valor anterior
            // Guardo el indice del primer elemento repetido
            if(!repetido){
                PrimElemRepe = vec[i]; // Guardamos el valor actual
                repetido = true;
            }
        }else{
            vec[i] = generaAleatorio();
        }
    }

    // ordenamos el vector
    // Método que hace uso de quicksort para ordenar vector
    sort(vec.begin(), vec.end());

    // Buscamos primera ocurrencia del vector con busqueda secuencial
    // Reemplazamos en la variable el valor del elemento por el primer
    // indice que lo contenga
    PrimElemRepe = busquedaSecuencial(vec, PrimElemRepe);

    return vec; // Devolvemos el indice
}
```

Actividad 5

El problema que se nos plantea es bastante sencillo, sabiendo que el coste del algoritmo de ordenación Mergesort es $\Theta(\log n)$, podemos simplemente ordenarlo y, una vez ordenado, recorrer el array 1 vez, guardando en un vector auxiliar los elementos únicos. Esto nos daría un

coste que sería igual al coste de Mergesort más el coste de recorrer el array de talla N una vez ($\Theta(n)$).

Por tanto:

$\Theta(n \log n)$

Actividad 5.4

La complejidad se mantendría. Sólo hay que añadir un contador para guardarlo en caso de que coincida con el número de veces requerido.