Fundamentos de análisis de algoritmos

Práctica 4

Análisis teórico

Búsqueda Binaria 1

```cpp
int AlgoritmosBusqueda::busquedaBinaria1(int A[], int izq, int der, int x)
{

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

}
```

Mejor caso

El mejor caso se da cuando el elemento se encuentra en el punto medio de la primera comparación.

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.

Mejor caso → Θ(1)

Peor caso y caso medio

Hacemos un ejemplo para sacar un patrón común. Empezamos con n elementos:

T(n)=T(n/2)+c

Para resolver T(n/2), hacemos lo mismo: T(n/2)=T(n/4)+c.

Sustituyendo: T(n)=(T(n/4)+c)+c=T(n/4)+2c

Y otra vez: T(n/4)=T(n/8)+c.

Sustituyendo: T(n)=(T(n/8)+c)+2c=T(n/8)+3c

Vemos un patrón: después de k pasos, $T(n) = T\left(\frac{n}{2^k}\right) + k \cdot c$

Por recursividad, el problema se repite hasta que el tamaño es muy pequeño, digamos 1.

Por definición de logaritmo, necesitamos $\log_2 n$ pasos para que el tamaño sea 1:

Para que $\frac{n}{2^k} = 1 \rightarrow k = \log_2 n$

Sustituyendo de nuevo vemos como

$$T(n) = T(1) + (\log_2 n) \cdot c$$

Al ser T(1) constante, podemos ver como el coste crece logarítmicamente con n

Por tanto: $\Theta(\log n)$

Búsqueda Binaria 2

```
1  {
2
3      if (izq > der) { // 1 OE comparación
4          return -1;
5      } else {
6          int medio = (izq + der) / 2; // 3 OE → Asignación y 2 operaciones aritméticas
7          if (A[medio] == x && (medio == izq || A[medio - 1] != x)) {
8              // 2 OE → 2 accesos al vector, 3 comparaciones, 2 op lógicas, 1 op aritmética
9              return medio;
10         } else if (A[medio] >= x) { // 2 OE → acceso al vector y comparación
11             return busquedaBinaria2(A, izq, medio - 1, x); // 2 OE → llamada a función y operación aritmética + Recursividad
12         } else {
13             return busquedaBinaria2(A, medio + 1, der, x); // 2 OE → llamada a función y operación aritmética + Recursividad
14         }
15
16     }
17     // 1 OE de cualquier return en cualquier caso. Arriba no las tuve en cuenta para hacerlo aquí
18
19 }
```

Esta búsqueda es prácticamente idéntica a la anterior, la única diferencia radica en la primera comparación, que ahora también debe cumplir que el punto medio sea igual al inicio o que el elemento anterior al punto medio sea distinto al elemento a buscar.

Mejor caso

Al igual que el algoritmo anterior, el mejor caso se da cuando se cumple la condición del if de la línea 8 en la primera llamada al método. Se realizarían 13 OE y el coste sería también constante.

Mejor caso → $\Theta(1)$

Peor caso y caso medio

El estudio sería exactamente el mismo que el algoritmo anterior.

Por tanto: $\Theta(\log n)$

Búsqueda por interpolación

```
1  int AlgoritmosBusqueda::busquedaBinariaInterpolacion(int A[], int izq, int der, int x)
2  {
3
4      if (izq > der || x < A[izq] || x > A[der]) {
5          return -1;
6      } else {
7          int pos = izq + ((x - A[izq]) * (der - izq)) / (A[der] - A[izq]);
8          if (pos < izq || pos > der) {
9              pos = (izq + der) / 2;
10         }
11         if (A[pos] == x) {
12             return pos;
13         } else if (A[pos] > x) {
14             return busquedaBinariaInterpolacion(A, izq, pos - 1, x);
15         } else {
16             return busquedaBinariaInterpolacion(A, pos + 1, der, x);
17         }
18     }
19 }
```

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.

Mejor caso

El mejor caso se da cuando la estimación del algoritmo es correcta en la primera llamada al método y cumple el if de la línea 11. En este caso, el coste sería constante.

Por tanto: $\Theta(1)$

Peor caso

Ocurre cuando la distribución no es nada uniforme. Por ejemplo, cuando es exponencial. La estimación de pos es consistentemente incorrecta, lo que lleva a una reducción lineal del espacio de búsqueda en cada paso. En este caso, el algoritmo puede degenerar a una búsqueda lineal. $\mathrm{T(n)} = \Theta(n)$

Caso Medio

Para datos uniformemente distribuidos, el número esperado de comparaciones es mucho menor que en la búsqueda binaria estándar. $T(n) = \Theta(\log(\log n))$