Análisis y Diseño de Algoritmos.

Sesión 4. 9 de septiembre de 2015.

Maestría en Sistemas Computacionales.

Por: Hugo Iván Piza Dávila.

Divide y Vencerás

- Familia de algoritmos que dividen un problema en problemas más pequeños.
- Se espera que la complejidad para resolver los problemas pequeños sea menor que la del problema original.
- La descomposición en sub-problemas se realiza mediante la recursión.
- Esta técnica es utilizada en algoritmos de ordenamiento y búsqueda sofisticados, como:
  - ***Quicksort, Mergesort, búsqueda binaria interpolada***

Divide y Vencerás

- Hay dos razones para utilizar esta técnica:

1. El algoritmo es más intuitivo que la versión original (iterativa) y no agrega costo computacional.
2. El algoritmo es más rápido que la versión original: reduce la complejidad temporal.

- No todo algoritmo recursivo es Divide y Vencerás:

```
return Fibonacci(N - 1) + Fibonacci(N - 2)
```

1. Agrega costo computacional a la versión iterativa
2. ¿Divide y Perderás?

Divide y Vencerás

- Forma general 1:

```
void Método(espacio de búsqueda)
```

1. Realizar operación(es) con el espacio de búsqueda
2. ¿Podemos terminar con éxito?

   Encontramos lo que estábamos buscando
3. ¿Podemos terminar con fracaso?

   El espacio de búsqueda llegó al mínimo
4. ¿Aún no podemos terminar?
   a. Si es necesario, invocar Método(sub-espacio-1) …
   b. Si es necesario, invocar Método(sub-espacio-***n***)

Divide y Vencerás

- Forma general 1:

  `type Método(espacio de búsqueda)`

  1. Realizar operación(es) con el espacio de búsqueda
  2. ¿Podemos terminar con éxito?

     Devolvemos el valor que estábamos buscando
  3. ¿Podemos terminar con fracaso?

     Devolvemos un valor de error de tipo `type`
  4. ¿Aún no podemos terminar?
     a. Si es necesario, devolver Método(sub-espacio-1) …
     b. Si es necesario, devolver Método(sub-espacio-***n***)

Búsqueda binaria

- Restricción: el arreglo debe estar ordenado.
- Idea:
  1. Buscar el valor en el punto medio de la lista.
  2. Si no está ahí, debe estar:
     a) O en la mitad izquierda, si el valor a buscar fue menor.
     b) O en la mitad derecha, si el valor a buscar fue mayor.
  3. Buscar el valor en el punto medio de la mitad elegida.
  4. Si no está ahí, debe estar:
     a) O en la mitad izquierda, si el valor a buscar fue menor.
     b) ¿Le seguimos?

Búsqueda binaria

- ¿O mejor con un ejemplo?

1. Valor a buscar = **26**
2. Espacio de búsqueda = [5, 8, 10, 13, 19, 26, 35, 47]
3. Punto medio = 19
4. Espacio de búsqueda = [5, 8, 10, 13, 19, 26, 35, 47]
5. Punto medio = 35
6. Espacio de búsqueda = [5, 8, 10, 13, 19, 26, 35, 47]
7. Punto medio = 26 … ☺

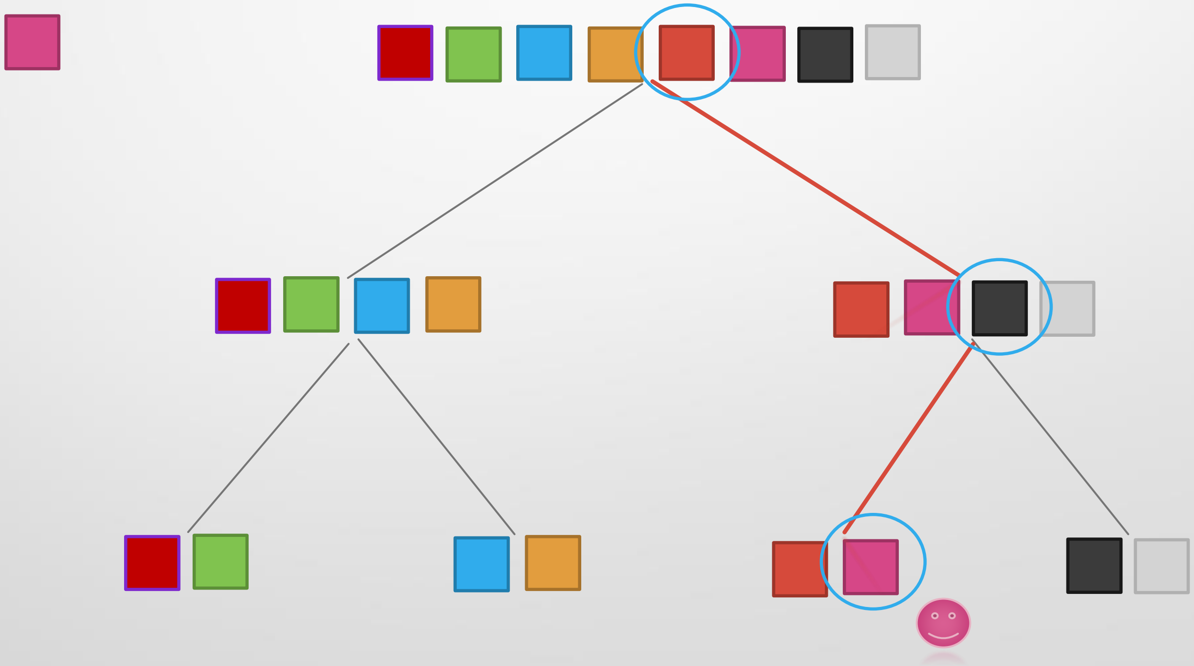
¿Cuántas comparaciones se hicieron?

¿Cómo fue cambiando el espacio de búsqueda?

Búsqueda binaria

- ¿O mejor con un dibujito?

A buscar :



Mediana

- Restricción: no existen elementos repetidos.

número Mediana(L: lista de N elementos, K: posición esperada)

1. $P \leftarrow$ el número de elementos más pequeños que $L_{N/2}$
2. Si $P = K$, *devolver* $L_{N/2}$
3. Si $P > K$ *en la mitad hay un número grande*
   a) Crear una lista L1 con los elementos más pequeños que $L_{N/2}$
   b) Devolver Mediana(L1, K)
4. Si $P < K$ *en la mitad hay un número pequeño*
   a) Crear una lista L2 con los elementos más grandes que $L_{N/2}$
   b) Devolver Mediana(L2, K – P – 1)

Mediana

- ¿O mejor con un ejemplito? *El K inicial será N/2*

1. Lista = [19, 10, 47, 5, 13, 26, 8, 35], K = 4

   $L_{N/2} = 13$, $P = 3 < K \;\therefore\; K = K - P - 1 = 0$

2. Lista = [19, 47, 26, 35], K = 0

   $L_{N/2} = 26$, $P = 1 > K$

3. Lista = [19], K = 0

   $L_{N/2} = 19$, $P = 0 = K \;\therefore$ Mediana = 19

¿Cuántas comparaciones se efectuaron?

Quicksort

- Inventado en 1960 por C.A.R. Hoare, británico.
- Ejecuta en promedio N log N operaciones para ordenar N elementos.
- Uno de los algoritmos de ordenamiento eficientes más populares: no es difícil de implementar.
- Desventajas:
  - Es recursivo en su forma original (se puede arreglar).
  - Ejecuta N^2 operaciones en el peor caso.
  - Es frágil: un error pequeño en la implementación puede ocasionar que no funcione en varios casos (¿suena conocido?)

Quicksort

- Realiza *particiones* del espacio de búsqueda.

1. Elige un elemento de la lista: pivote (típicamente el último).
2. Determina la posición definitiva para pivote.
3. Coloca los elementos menores a pivote de su lado izquierdo y a los mayores de si lado derecho.
    - Nótese que los dos sub-arreglos formados se pueden ordenar de forma independiente.
4. Repite todos los pasos con la mitad izquierda y derecha de pivote hasta que el tamaño de cada sub-arreglo lo permita ordenar de forma manual ($N \leq 2$).

Quicksort

- Los sub-arreglos se van a gestionar mediante índices izquierdo y derecho (no mediante nuevos arreglos).
- Quicksort(array, left, right)
  1. Si el arreglo delimitado por left y right es lo suficientemente pequeño, ordenar (si es necesario) y terminar.
  2. Sea p = partition (array, left, right)
     - p es la posición final del elemento elegido como pivote.
     - La implementación de partition() varía pero es crucial.
  3. Quicksort(array, left, p – 1)
  4. Quicksort(array, p + 1, right)

¿Cómo realizar una partición?

1. Elegir al último elemento como pivote (right)
2. Determinar la posición p_1 del primer elemento que no sea <u>menor</u> que el pivote (que no deba estar ahí).
   - La búsqueda va de izquierda a derecha del sub-arreglo.
3. Determinar la posición p_2 del primer elemento que no sea <u>mayor</u> que el pivote (que no deba estar ahí).
   - La búsqueda va de derecha – 1 a izquierda del sub-arreglo.
4. ¿Se cruzaron p_1 y p_2?
   1. Intercambiar los elementos en p_1 y pivote (right).
   2. La posición de la partición es p_1.
5. ¿No se cruzaron?
   1. Intercambiar los elementos en p_1 y p_2.
   2. Regresar al paso 2 con los p_1, p_2 siguientes.

Ejemplo

1. Lista a ordenar = {5, 4, -8, 2, -1, 9, 0, -3, 7, 6}
2. Izquierda = 0, Derecha = 9, Pivote = 6
3. Primera partición:
   1. $p_1 = 5$ {$lista_5 = 9$}
   2. $p_2 = 7$ {$lista_7 = -3$}
   3. No se cruzan
   4. Swap(5, 7)
   5. Lista = {5, 4, -8, 2, -1, -3, 0, 9, 7, 6}
   6. $p_1 = 7$ {$lista_7 = 9$}
   7. $p_2 = 6$ {$lista_6 = 0$}
   8. Sí se cruzan
   9. Swap(7, 9)
   10. Lista = {**5, 4, -8, 2, -1, -3, 0**, 6, **7, 9**}
   11. Partición = 7

Mergesort

- Inventado en 1945 por John von Neumann, Húngaro.
- Su complejidad temporal es N log N en los casos mejor, peor y promedio.
- Mezcla el contenido de dos arreglos ordenados en un arreglo más grande $\Rightarrow$ complejidad lineal.
- Desventajas:
  - Es recursivo en su forma original (se puede arreglar).
  - La necesidad de estar creando nuevos arreglos en cada llamada recursiva.

Mergesort

- Realiza *particiones* del espacio de búsqueda.

1. Si el arreglo tiene longitud mínima, ordenarlo de manera manual y devolverlo.
2. Crea dos sub-arreglos del arreglo original, uno con la mitad izquierda y otro con la mitad derecha.
   - Es posible que un arreglo sea más grande que el otro.
3. Ordena cada sub-arreglo mediante dos llamadas a este mismo método.
4. Devuelve el resultado de <u>mezclar</u> los dos sub-arreglos previamente ordenados.

¿Cómo hacer la mezcla?

- Arreglo1 = {3, 5, 7, 8, 10}
- Arreglo2 = {2, 4, 5, 6, 10, 12}
- Arreglo3 = {2, 3, 4, 5, 5, 6, 7, 8, 10, 10, 12}
- Se requiere un contador por cada arreglo
  - Arreglo3[0] = Arreglo2[0]
  - Arreglo3[1] = Arreglo1[0]
  - Arreglo3[2] = Arreglo2[1]
  - Arreglo3[3] = Arreglo1[1]
  - Arreglo3[4] = Arreglo2[2]
  - Arreglo3[5] = Arreglo2[3] …
  - Arreglo3[10] = Arreglo2[5]

Ejemplo

1. Lista a ordenar = {5, 4, -8, 2, -1, 9, 0, -3, 7, 6}
2. **Izquierdo** = {5, 4, -8, 2, -1}
   a) Izquierdo' = {5, 4}
      i. Izquierdo'' = {5}
      ii. Derecho'' = {4}
      iii. Mezcla'' = {4, 5}

      Ordenar izquierdo'

   b) Derecho' = {-8, 2, -1}
      i. Izquierdo'' = {-8}
      ii. Derecho'' = {2, -1}
      iii. Mezcla'' = {-8, -1, 2}

      Ordenar derecho'

   c) Mezcla' = {-8, -1, 2, 4, 5}

   Ordenar izquierdo

3. **Derecho** = {9, 0, -3, 7, 6}
   a) Izquierdo' = {9, 0}
   b) Derecho' = {-3, 7, 6}
   c) Mezcla' = {-3, 0, 6, 7, 9}

   Ordenar derecho

Análisis de algoritmos recursivos

- C_N = Complejidad con N elementos o número de instrucciones ejecutadas con N elementos
- En análisis se realizará de forma inductiva
- Caso base: normalmente C_1
- Caso inductivo: C_N estará en función de $C_{M<N}$

Análisis de algoritmos recursivos

- Caso 1:
  - En cada llamada recursiva el espacio de búsqueda se reduce en uno.
  - Se efectúa una operación con cada elemento.
- Fórmula general: $C_N = C_{N-1} + N$, con: $N \geq 2$, $C_1 = 1$.
  - La complejidad con N elementos es igual a N más la complejidad con un elemento menos.
- $C_N = C_{N-1} + N$. Observe que: $C_{N-1} = C_{N-2} + (N - 1)$.

$$
\begin{aligned}
C_N &= C_{N-1} + N \\
&= C_{N-2} + N - 1 + N \\
&= C_{N-3} + N - 2 + N - 1 + N \ldots \\
&= C_1 + 2 + 3 + \ldots + N - 2 + N - 1 + N = \tfrac{1}{2}N(N + 1) \in O(N^2).
\end{aligned}
$$

Análisis de algoritmos recursivos

- Caso 2:
  - En cada llamada recursiva el espacio de búsqueda se reduce a la mitad:
  - Se efectúa sólo una operación.
- Fórmula general: $C_N = C_{N/2} + 1$, con: $N = 2^M \geq 2$, $C_1 = 0$
  - La complejidad con N elementos es igual a uno más la complejidad con la mitad de los elementos.
- $C_N \quad = C_{N/2} + 1$. Observe que: ${}^{N}/_{2} = 2^{M-1}$.

$$= C_{2^{M-1}} + 1$$

$$= C_{2^{M-2}} + 1 + 1$$

$$= C_{2^{0}} + 1 + \ldots + 1 + 1 \;=\; M = \mathbf{log_2 N} \in O(\lg N).$$

Análisis de algoritmos recursivos

- Caso 3:
  - En cada llamada recursiva el espacio de búsqueda se reduce a la mitad.
  - Se efectúa una operación por cada elemento.
- Fórmula general: $C_N = C_{N/2} + N$, con: $N = 2^M \geq 2$, $C_1 = 0$.
  - La complejidad con N elementos es igual a N más la complejidad con la mitad de los elementos.
- $C_N \quad = C_{N/2} + N$. Observe que: ${}^N/_2 = 2^{M-1}$.

$= C_{2^{M-1}} + N$

$= C_{2^{M-2}} + \frac{1}{2}N + N$

$= C_{2^0} + \mathbf{1} + 2 + 4 \ldots + \frac{1}{4}N + \frac{1}{2}N + N \;=\; \mathbf{2N - 1} \in O(N)$.

Análisis de algoritmos recursivos

- Caso 4:
  - Se realizan dos llamadas recursivas, cada una procesa una mitad del espacio de búsqueda actual.
  - Se efectúa una operación con cada elemento.
- Fórmula general: $C_N = 2C_{N/2} + N$, con: $N = 2^M \geq 2$, $C_1 = 0$.
  - La complejidad con N elementos es igual a N más el doble de la complejidad con la mitad de los elementos (porque son dos llamadas recursivas).
- $C_N = 2C_{N/2} + N$. Observe que: ${}^{N}/_{2} = 2^{M-1}$.

$$= 2C_{2^{M-1}} + N$$

$$= 4C_{2^{M-2}} + N + N$$

$$= NC_{2^0} + N + N + \ldots + N = MN = \mathbf{N \log_2 N} \in O(N \lg N).$$

Tarea

- Para Radix, Mediana, Quicksort y Mergesort:
  1. Terminar la implementación (de Mediana no).
  2. Realizar un análisis *a priori* de la complejidad temporal y espacial en los casos que lo permitan.
  3. Usando varios arreglos desordenados con diferente N, comparar los tiempos de Quicksort y Mergesort.
  4. Lo mismo pero con arreglos casi-ordenados/invertidos.
  5. Incluir código fuente nuevo y tablas comparativas.