CI2612: Algoritmos y Estructuras de Datos II

Blai Bonet

Universidad Simón Bolívar, Caracas, Venezuela

Quicksort

© 2016 Blai Bonet

Objetivos

- Algoritmo quicksort y su análisis de tiempo: peor caso, mejor caso y caso promedio
- Algoritmo randomizado de quicksort y su análisis
- Discutir la optimalidad asintótica de ambos algoritmos con respecto a tiempo promedio y tiempo esperado

© 2016 Blai Bonet

Introducción a quicksort

El algoritmo de quicksort tiene un tiempo de corrida de $\Theta(n^2)$ en el peor caso y $\Theta(n \log n)$ en el **caso promedio**

Sin embargo, la constante escondida en $\Theta(n \log n)$ para quicksort es mucho menor que la constante escondida en $\Theta(n \log n)$ para heapsort

Al igual que heapsort, quicksort es un algoritmo "in place": requiere memoria adicional constante

La versión randomizada de quicksort es tal vez el algoritmo más utilizado para ordenar grandes volúmenes de datos

Quicksort fue inventado por C. A. R. Hoare en 1962

© 2016 Blai Bonet

Dividir y conquistar

Como mergesort, quicksort es un **algoritmo recursivo** del tipo dividir y conquistar:

– **Divide** el arreglo de entrada $A[p \dots r]$ en dos subarreglos $A[p \dots q-1]$ y $A[q+1 \dots r]$ con $p \leq q \leq r$ tal que $A[i] \leq A[j]$ para todo i y j tal que $1 \leq i \leq q \leq j \leq r$

– **Conquista** cada subarreglo al ordenarlos de forma recursiva

– **Combina** los subarreglos ordenados de forma trivial ya que al hacer el ordenamiento "in place" de cada subproblema, no hace falta trabajo adicional para combinar los resultados

© 2016 Blai Bonet

Quicksort

Input: arreglo $A[p \dots r]$ con $r-p+1$ elementos e índices p y r

Output: arreglo $A[p \dots r]$ reordenado de menor a mayor

```
1 Quicksort(array A, int p, int r)
2     if p < r
3         q = Partition(A, p, r)
4         Quicksort(A, p, q-1)
5         Quicksort(A, q+1, r)
```

La llamada inicial para ordenar $A[1 \dots n]$ es `Quicksort(A,1,n)`

`Partition(A,p,r)` es la **rutina clave** que retorna un índice q y particiona el arreglo en dos subarreglos $A[p \dots q-1]$ y $A[q+1 \dots r]$ tal que $A[i] \leq A[q] \leq A[j]$ para todo $p \leq i < q$ y todo $q < j \leq r$

© 2016 Blai Bonet

Partition en acción

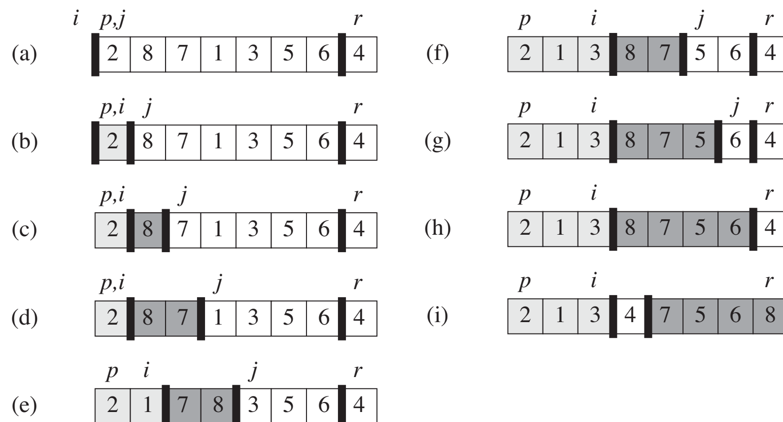

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2016 Blai Bonet

Partition

Input: arreglo $A[p \dots r]$ con $r-p+1$ elementos e índices p y r

Output: índice q y arreglo $A[p \dots r]$ reordenado tal que $A[i] \leq A[q] \leq A[j]$ para todo $p \leq i < q < j \leq r$

```
1 Partition(array A, int p, int r)
2     x = A[r]                                  % elemento pivote
3     i = p - 1
4     for j = p to r - 1
5         if A[j] <= x
6             i = i + 1
7             Intercambiar A[i] con A[j]
8     Intercambiar A[i+1] con A[r]
9     return i + 1
```

El tiempo de corrida de `Partition(A,p,r)` es $\Theta(n)$ donde $n = r-p+1$ ya que realiza n iteraciones, cada una requiriendo tiempo constante

© 2016 Blai Bonet

Correctitud de Quicksort

La correctitud de `Quicksort` se reduce a la correctitud de `Partition`: si `Partition` es correcto, `Quicksort` es también correcto

Para establecer la correctitud de `Partition`, usaremos los siguientes invariantes para el inicio de cada iteración del lazo 4–7:

1. si $p \leq k \leq i$, entonces $A[k] \leq x$
2. si $i < k < j$, entonces $A[k] > x$
3. si $k = r$, entonces $A[k] = x$

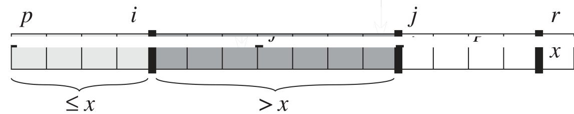

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2016 Blai Bonet

Correctitud de Partition (1 de 3)

Inicialización: al comienzo de la primera iteración, $i = p - 1$ y $j = p$. Como no existe k tal que $p \leq k \leq i$ ó $i < k < j$, los invariantes 1 y 2 son ciertos. El invariante 3 es cierto por la asignación $x \leftarrow A[r]$ en la línea 2 de `Partition`

© 2016 Blai Bonet

Correctitud de Partition (2 de 3)

Mantenimiento: consideramos dos casos para una iteración dada: (a) $A[j] > x$ y (b) $A[j] \leq x$

Caso (a): la iteración no cambia el arreglo y j se incrementa. Al inicio de la nueva iteración tenemos un nuevo $k = j$ para el cual verificar el invariante 2, el cual es cierto porque $A[j] > x$

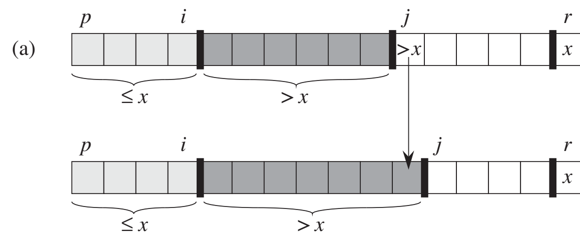

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2016 Blai Bonet

Correctitud de Partition (2 de 3)

Mantenimiento: consideramos dos casos para una iteración dada: (a) $A[j] > x$ y (b) $A[j] \leq x$

Caso (b): $A[i+1]$ se intercambia con $A[j]$ e i se incrementa. No es difícil ver que el invariante se cumple al inicio de la próxima iteración

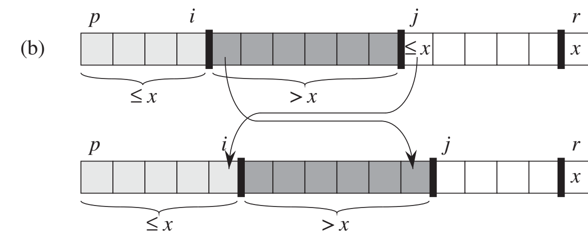

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2016 Blai Bonet

Correctitud de Partition (3 de 3)

Terminación: al finalizar el lazo, $j = r$ y todos los elementos del arreglo pertenecen a uno de los tres grupos:

- los $A[k]$ para $p \leq k \leq i$ son menores o iguales a x
- los $A[k]$ para $i < k < r$ son mayores a x
- $A[r] = x$

El último paso de `Partition(A,p,r)` es intercambiar $A[i+1]$ con $A[r]$ y retornar $i+1$

Si `Partition(A,p,r)` retorna q, es claro que $A[i] \leq A[q] < A[j]$ para $p \leq i < q < j \leq r$ □

© 2016 Blai Bonet

Discusión del desempeño de Quicksort (1 de 5)

El desempeño de `Quicksort` depende de como es el "balance" de la partición en cada nivel de la recursión:

- si la partición es balanceada, `Quicksort` corre en tiempo $O(n \log n)$
- si la partición no es balanceada, `Quicksort` corre en tiempo $O(n^2)$

El balance de la partición depende del pivote que se selecciona para hacer la partición

© 2016 Blai Bonet

Discusión del desempeño de Quicksort (2 de 5)

Peor caso: el peor caso ocurre cuando la partición tiene $n-1$ elementos de un lado y 0 elementos del otro lado

Si esto sucede en cada llamada recursiva, podemos expresar el tiempo $T(n)$ de `Quicksort` para n elementos por:

$$T(n) \;=\; T(n-1) + \Theta(n)$$

ya que calcular la partición toma tiempo $\Theta(n)$

En este caso, $T(n) = \Theta(n^2)$ lo cual puede verificarse por sustitución

© 2016 Blai Bonet

Discusión del desempeño de Quicksort (3 de 5)

Mejor caso: en el mejor caso la partición es completamente balanceada: un lado contiene $\lfloor n/2 \rfloor$ elementos y el otro $\lceil n/2 \rceil - 1$

Si esto sucede en cada llamada recursiva, el tiempo $T(n)$ para `Quicksort` satisface:

$$T(n) \;=\; 2T(n/2) + \Theta(n)$$

Por el 2do caso del Teorema Maestro, $T(n) = \Theta(n \log n)$

© 2016 Blai Bonet

Discusión del desempeño de Quicksort (4 de 5)

Caso promedio: en el caso promedio `Quicksort` se comporta como en el mejor caso ya que para incurrir en $O(n^2)$ tiempo, muchas particiones deben ser muy desbalanceadas

De hecho, si en cada llamada recursiva el "split" es de 9 a 1 (i.e. un lado de la partición tienen 9 veces más elementos que el otro), el tiempo $T(n)$ puede expresarse por:

$$T(n) \;=\; T(9n/10) + T(n/10) + cn$$

donde hemos sustituido el término $\Theta(n)$ por cn

Aún en este caso, `Quicksort` incurre en tiempo $O(n \log n)$ como se puede ver en la siguiente figura (y lo mismo ocurre cuando el split tiene una **proporcionalidad constante** en cada nivel)

© 2016 Blai Bonet

Análisis de Quicksort para particiones 9 a 1

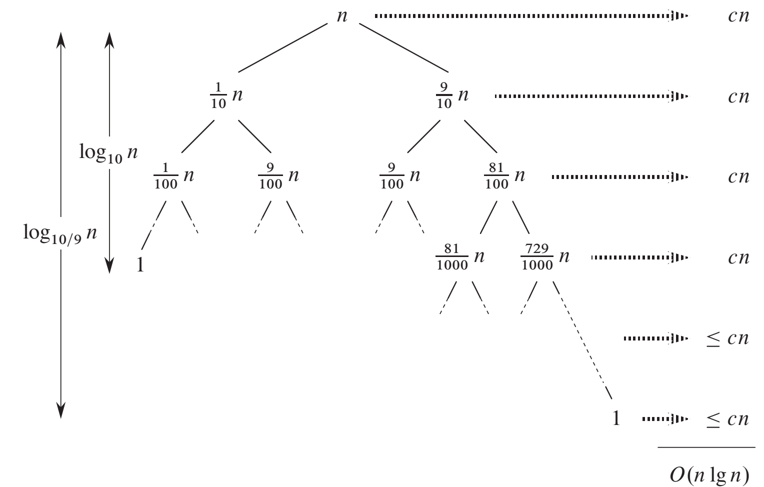

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

© 2016 Blai Bonet

Discusión del desempeño de Quicksort (5 de 5)

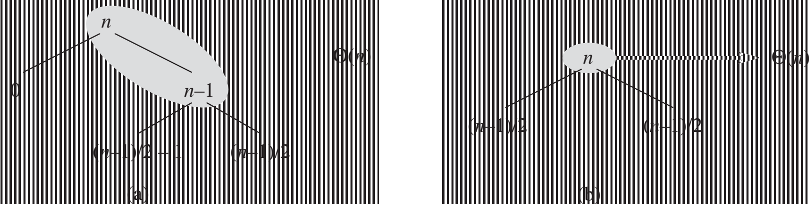

Imagen de Cormen et al. Intro. to Algorithms. MIT Press

Aún si suponemos que los splits se alternan en el árbol de recursión entre splits mejor caso y splits de peor caso, `Quicksort` corre en tiempo $O(n \log n)$

© 2016 Blai Bonet

Quicksort randomizado

El desempeño de `Quicksort` depende de obtener buenos splits para el arreglo A. En `Randomized-Quicksort` escogemos el pivote al azar para evitar con gran probabilidad los splits malos

La versión randomizada de quicksort, `Randomized-Quicksort`, se considera el **algoritmo de elección** cuando se quieren ordenar por **comparación** grandes volúmenes de datos

Una manera de randomizar el algoritmo es permutar la entrada de forma aleatoria antes de ejecutar `Quicksort`. Sin embargo, el análisis es más fácil si en cada llamada a `Partition` escogemos un **pivote al azar**

© 2016 Blai Bonet

Quicksort randomizado

Input: arreglo $A[p \ldots r]$ con $r-p+1$ elementos e índices p y r

Output: arreglo $A[p \ldots r]$ reordenado de menor a mayor

```
1  Randomized-Quicksort(array A, int p, int r)
2      if p < r
3          q = Randomized-Partition(A, p, r)
4          Randomized-Quicksort(A, p, q-1)
5          Randomized-Quicksort(A, q+1, r)
6
7  Randomized-Partition(array A, int p, int r)
8      i = Random(p, r)
9      Intercambiar A[i] con A[r]
10     return Partition(A, p, r)
```

© 2016 Blai Bonet

Correctitud de Randomized-Quicksort

La correctitud de `Randomized-Quicksort` se desprende automáticamente de la correctitud de `Quicksort`

La correctitud de `Quicksort` no depende del elemento pivote escogido en cada llamada recursiva. Sólo depende de la correctitud de `Partition` en calcular la partición de A dado el pivote

© 2016 Blai Bonet

Análisis del peor caso de Quicksort

Sea $T(n)$ el tiempo en el peor caso para quicksort de n elementos:

$$T(n) = \max_{0 \leq q < n} [T(q) + T(n-q-1)] + \Theta(n)$$

donde q representa el índice devuelto por `Partition(A,p,r)`

Utilizamos sustitución con el **"guess"** $T(n) \leq cn^2$ para alguna c (ver abajo):

$$\begin{aligned} T(n) &= \max_{0 \leq q < n} [T(q) + T(n-q-1)] + \Theta(n) \\ &\leq \max_{0 \leq q < n} [cq^2 + c(n-q-1)^2] + \Theta(n) \\ &\leq c \max_{0 \leq q < n} [q^2 + (n-q-1)^2] + \Theta(n) \end{aligned}$$

La función $q^2 + (n-q-1)^2$ obtiene su máximo en $q = 0$ ó $q = n-1$

© 2016 Blai Bonet

Análisis del peor caso de Quicksort

Sea $T(n)$ el tiempo en el peor caso para quicksort de n elementos:

$$T(n) = \max_{0 \leq q < n} [T(q) + T(n-q-1)] + \Theta(n)$$

donde q representa el índice devuelto por `Partition(A,p,r)`

Entonces,

$$\begin{aligned} T(n) &\leq c \max_{0 \leq q < n} [q^2 + (n-q-1)^2] + \Theta(n) \\ &= c(n-1)^2 + \Theta(n) \\ &\leq cn^2 \end{aligned}$$

para c suficientemente grande tal que $cn^2 - c(n-1)^2 = c(2n-1) \geq \Theta(n)$

© 2016 Blai Bonet

Análisis del peor caso de Quicksort

Sea $T(n)$ el tiempo en el peor caso para quicksort de n elementos:

$$T(n) = \max_{0 \leq q < n} [T(q) + T(n-q-1)] + \Theta(n)$$

donde q representa el índice devuelto por `Partition(A,p,r)`

Hemos probado $T(n) = O(n^2)$

Por otro lado, si la partición es siempre **desbalanceada**, $T(n) = \Omega(n^2)$ (ejercicios)

Entonces, $T(n) = \Theta(n^2)$

© 2016 Blai Bonet

Análisis del caso promedio de Quicksort

Haremos el análisis del caso promedio suponiendo que todos los elementos del arreglo son **distintos**. Si existen elementos iguales, `Randomized-Quicksort` puede tomar tiempo esperado $\Theta(n^2)$ (ejercicios)

Lo primero que tenemos que hacer es considerar una distribución de probabilidad sobre todas las $n!$ posibles entradas de tamaño n al algoritmo de `Quicksort`

A falta de información específica sobre la distribución, asumimos que todas las entradas son **equiprobables** (igualmente probables)

Primero analizamos el tiempo esperado de `Randomized-Quicksort` y luego analizamos el tiempo promedio de `Quicksort`

© 2016 Blai Bonet

Análisis de Randomized-Quicksort (1 de 7)

Primero veamos que `Randomized-Partition` se llama a lo sumo n veces donde $n = r - p + 1$ es el número de elementos en A

`Randomized-Partition` selecciona un elemento pivote en A, el elemento $A[q]$, y dicho elemento no vuelve a considerarse durante la recursión

Como existen n elementos y `Randomized-Partition` "saca" un elemento en cada llamada, `Randomized-Partition` se llama a lo sumo n veces

© 2016 Blai Bonet

Análisis de Randomized-Quicksort (2 de 7)

Ahora calculamos el tiempo tomado por **todas las llamadas** a `Randomized-Partition` (cf. análisis agregado de tiempo)

El tiempo de una llamada es proporcional al número de comparaciones que se realizan en la línea 5 de `Partition`
(recuerde que `Randomized-Partition` llama a `Partition`)

El **tiempo total** de `Randomized-Quicksort` es $O(n + X)$ donde n acota el número de llamadas hechas a `Randomized-Partition` y X es el número total de comparaciones que `Partition` realiza en todas sus invocaciones

© 2016 Blai Bonet

Partition

Input: arreglo $A[p \dots r]$ con $r - p + 1$ elementos e índices p y r

Output: índice q y arreglo $A[p \dots r]$ reordenado tal que $A[i] \leq A[q] \leq A[j]$ para todo $p \leq i < q < j \leq r$

```
1 Partition(array A, int p, int r)
2     x = A[r]                                  % elemento pivote
3     i = p - 1
4     for j = p to r - 1
5         if A[j] <= x
6             i = i + 1
7             Intercambiar A[i] con A[j]
8     Intercambiar A[i+1] con A[r]
9     return i + 1
```

El tiempo de corrida de `Partition(A,p,r)` es $\Theta(n)$ donde $n = r - p + 1$ ya que realiza n iteraciones, cada una requiriendo tiempo constante

© 2016 Blai Bonet

Análisis de Randomized-Quicksort (3 de 7)

Número de comparaciones $A[j] \leq x$ realizadas por `Partition`:

Comenzamos **renombrando** los elementos de A como $z_1, z_2, \dots, z_n$ tal que $z_1 \leq z_2 \leq \cdots \leq z_n$

Sea $Z_{ij} = \{z_i, z_{i+1}, \dots, z_j\}$ el conjunto de elementos entre z_i y z_j inclusive

¿Cuántas comparaciones del tipo z_i con z_j hace `Partition`?

Observe que un par de elementos z_i y z_j se comparan a lo sumo una vez: si se comparan, uno de ellos es el pivote, el cual nunca vuelve a considerarse

Definimos la **v.a. indicadora** $X_{ij} = \mathbb{I}\{z_i \text{ se compara con } z_j\}$

© 2016 Blai Bonet

Análisis de Randomized-Quicksort (4 de 7)

Número de comparaciones $A[j] \leq x$ realizadas por `Partition`:

El número total de comparaciones realizadas es $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$

Calculemos el valor esperado de X:

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{E}[X_{ij}]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{P}(z_i \text{ se compara con } z_j)$$

© 2016 Blai Bonet

Análisis de Randomized-Quicksort (5 de 7)

Número de comparaciones $A[j] \leq x$ realizadas por `Partition`:

Para calcular $\mathbb{P}(z_i \text{ se compara con } z_j)$, considere la **primera llamada** a `Partition` que retorna un pivote $z_\ell \in Z_{ij}$

– Antes de dicha llamada, todos los elementos Z_{ij} **aparecen juntos** en los subarreglos A de todas las activaciones de `Randomized-Quicksort`; i.e. si algún $z \in Z_{ij}$ está en A, todos los otros también están en A (¿por qué?)

– Después de dicha llamada, Z_{ij} se particiona y sus elementos no vuelven a aparecer en el mismo arreglo; i.e. z_i y z_j se "separan"

– Si el pivote z_ℓ es z_i ó z_j, `Partition` hace la comparacion z_i con z_j

– Si el pivote z_ℓ no es z_i ni z_j, el elemento z_i **nunca se comparara** con z_j

© 2016 Blai Bonet

Análisis de Randomized-Quicksort (6 de 7)

Número de comparaciones $A[j] \leq x$ realizadas por `Partition`**:**

$$
\begin{aligned}
&\mathbb{P}(z_i \text{ se compara con } z_j) \\
&\quad = \mathbb{P}(\text{el primer pivote escogido en } Z_{ij} \text{ es } z_i \text{ ó } z_j) \\
&\quad \leq \mathbb{P}(\text{el primer pivote escogido en } Z_{ij} \text{ es } z_i) + \\
&\qquad \mathbb{P}(\text{el primer pivote escogido en } Z_{ij} \text{ es } z_j) \\
&\quad = \frac{1}{j-i+1} + \frac{1}{j-i+1} \\
&\quad = \frac{2}{j-i+1}
\end{aligned}
$$

© 2016 Blai Bonet

Análisis de Randomized-Quicksort (7 de 7)

Número de comparaciones $A[j] \leq x$ realizadas por `Partition`**:**

$$
\begin{aligned}
\mathbb{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbb{P}(z_i \text{ se compara con } z_j) \\
&\leq \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&\leq \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k+1} = \sum_{i=1}^{n-1} O(\log n) \\
&= O(n \log n)
\end{aligned}
$$

El tiempo esperado para `Randomized-Quicksort` sobre n elementos distintos es $O(n + n\log n) = O(n \log n)$ (igual al número esperado de comparaciones)

© 2016 Blai Bonet

Análisis del caso promedio de Quicksort

El tiempo esperado de `Randomized-Quicksort` es igual al tiempo esperado de un algoritmo que **permuta la entrada de forma aleatoria** y luego llama a `Quicksort` sobre la entrada permutada

Por otro lado, como todas las permutaciones de entradas las asumimos equiprobables, entonces el tiempo esperado del algoritmo que primero permuta la entrada y luego llama a `Quicksort` es igual al tiempo promedio de `Quicksort` sobre todas las entradas

Más adelante veremos que `Randomized-Quicksort` y `Quicksort` son **asintóticamente óptimos** con respecto a tiempo esperado y tiempo promedio (i.e. no existe algoritmo de ordenamiento por comparaciones que corra en tiempo promedio $o(n \log n)$ o algoritmo que realize un número $o(n \log n)$ de comparaciones esperadas)

© 2016 Blai Bonet

Resumen

- `Quicksort` es un algoritmo eficiente que corre en tiempo $O(n \log n)$ y realiza $O(n \log n)$ comparaciones en promedio (sobre todas las entradas)
- En el peor caso `Quicksort` realiza $\Theta(n^2)$ comparaciones y corre en $\Theta(n^2)$ tiempo
- `Randomized-Quicksort` evita el peor caso con gran probabilidad escogiendo el pivote a utilizar en cada llamada a `Partition` de forma aleatoria
- El tiempo esperado de `Randomized-Quicksort` para cualquier entrada es $O(n \log n)$, igual al número de comparaciones esperadas
- En muchos casos, `Randomized-Quicksort` es el algoritmo estándar para ordenar grandes volúmenes de datos

© 2016 Blai Bonet

Ejercicios (1 de 3)

1. (7.1-1) ¿Cuál es el resultado de `Partition` sobre el arreglo $\langle 13, 19, 9, 5, 12, 8, 7, 4, 21, 2, 6, 11\rangle$?

2. (7.1-2) ¿Qué retorna `Partition` cuando todos los elementos del arreglo son iguales? Modifique `Partition` para que retorne $q = \lfloor (p+q)/2 \rfloor$ cuando todos los elementos sean iguales. El nuevo `Partition` debe correr en tiempo lineal

3. (7.1-4) Modifique `Quicksort` para que ordene de forma decreciente

4. (7.2-1) Use el método de sustitución para probar que la recurrencia $T(n) = T(n-1) + \Theta(n)$ tiene solución $T(n) = \Theta(n^2)$

© 2016 Blai Bonet

Ejercicios (2 de 3)

5. (7.2-2) ¿Cuál es el tiempo de corrida de `Quicksort` cuando todos los elementos del arreglo son iguales?

6. (7.2-3) Muestre que el tiempo de corrida de `Quicksort` es $\Theta(n^2)$ cuando todos los elementos del arreglo son distintos y están ordenados de forma creciente

7. Muestre que `Quicksort` y `Randomized-Quicksort` no son algoritmos estables de ordenamiento

8. Obtenga una versión estable de `Quicksort` que corra en $O(n \log n)$ tiempo en promedio. Su algoritmo no tienen que ser "in place"

© 2016 Blai Bonet

Ejercicios (3 de 3)

9. (7.3-2) ¿Cuántas llamadas hace `Randomized-Quicksort` al generador de números aleatorios `Random` en el mejor caso y peor caso?

10. (7.4-3) Mostrar que la funcion $f(q) = q^2 + (n-q-1)^2$ obtiene su máximo sobre $q = 0, 1 \ldots, n-1$ cuando $q = 0$ ó $q = n-1$

11. (7.4-4) Mostrar que `Randomized-Quicksort` corre en tiempo esperado $\Omega(n \log n)$

© 2016 Blai Bonet