Análisis de complejidad

Quick Sort

Análisis de complejidad QuickSort

- El tiempo de ejecución del ordenamiento rápido depende de si la partición está equilibrada o desequilibrada, que a su vez depende de qué elementos se utilizan para la partición.

- Si la partición está equilibrada, el algoritmo se ejecuta asintóticamente tan rápido como el ordenamiento por mezcla.

- Sin embargo, si la partición está desequilibrada, puede ejecutarse asintóticamente con la misma lentitud como el algoritmo de inserción.

Partición en el peor de los casos

- El peor comportamiento de caso para la ordenación rápida ocurre cuando la rutina de partición produce un subproblema con **n – 1** elementos y otro con **0** elementos.
- Supongamos que esta partición desequilibrada surge en cada llamada recursiva.
- La partición toma un tiempo **Θ(n)**. Desde la llamada recursiva en un arreglo de tamaño 0 solo devuelve, **T(0) = Θ(1)** y la recurrencia del tiempo de ejecución es:

$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \ . \end{aligned}$$

- Intuitivamente, si sumamos los costos incurridos en cada nivel de la recursividad, obtenemos una serie aritmética, que se evalúa como $\Theta(n^2)$.

$$\sum_{k=1}^{n} k = \frac{1}{2}n(n+1) = \Theta(n^2) .$$

- De hecho, es sencillo de utilizar el método de sustitución para demostrar que la recurrencia:

  $T(n) = T(n-1) + \Theta(n)$ tiene la solución $T(n) = \Theta(n^2)$.

- Por lo tanto, si la partición está desequilibrada al máximo en cada nivel recursivo del algoritmo, el tiempo de ejecución es $\Theta(n^2)$.
- Por lo tanto, el peor tiempo de ejecución de Quicksort no es mejor que el de inserción.
- Además, el tiempo de ejecución $\Theta(n^2)$ ocurre también cuando el arreglo de entrada ya está completamente ordenado, una situación común en la cual inserción se ejecuta en tiempo $O(n)$.

Partición en el mejor de los casos

- En la división más par posible, PARTICIÓN produce dos subproblemas, cada uno de los cuales tiene tamaño no más de n/2, ya que uno es de tamaño $\lfloor n/2 \rfloor$ y otro de tamaño $\lceil n/2 \rceil - 1$.
- En este caso, quicksort se ejecuta mucho más rápido. La recurrencia del tiempo de ejecución es entonces:

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

- Donde toleramos el descuido de ignorar el redondeo de piso y de techo y de restar 1.
- Por el teorema maestro decimos que esta recurrencia tiene la solución:

$$T(n) = \Theta(n \lg n)$$

- Equilibrando los dos lados de la partición en cada nivel de la recursividad, obtenemos un algoritmo asintóticamente más rápido.

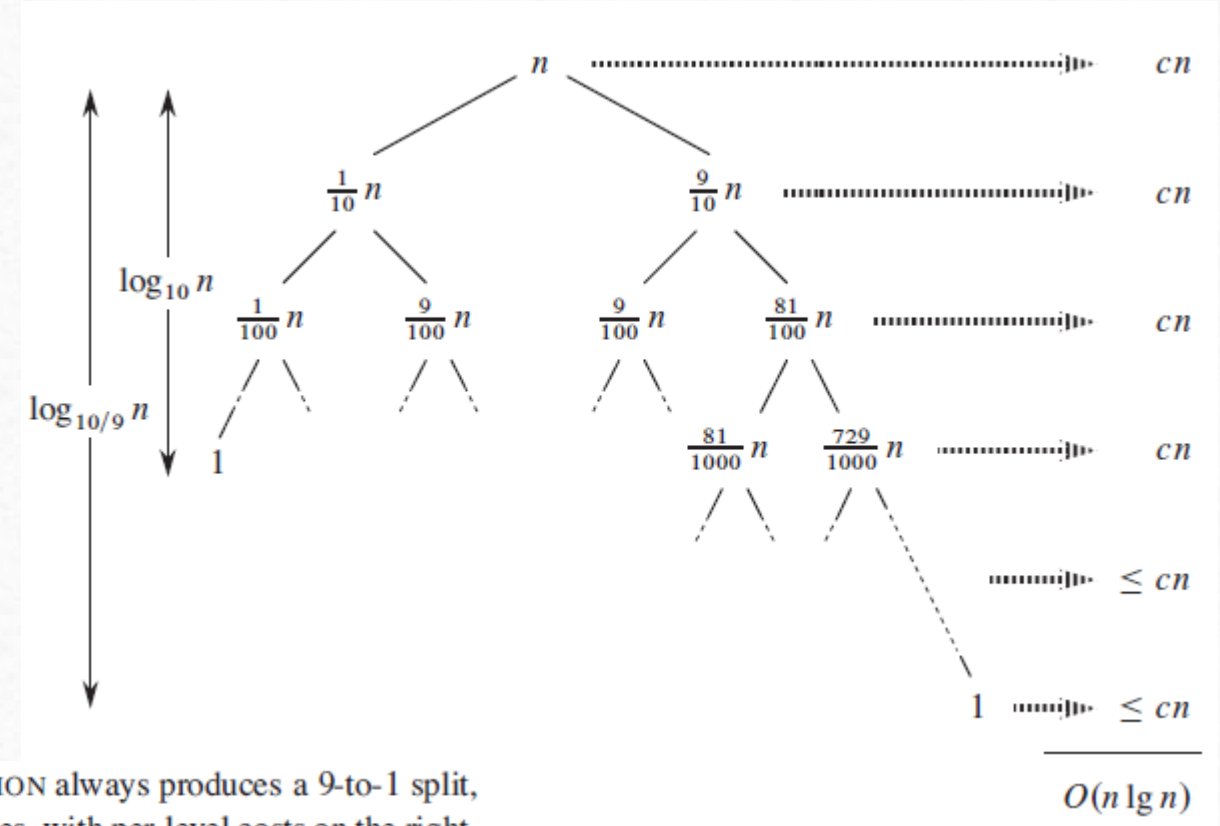
Partición equilibrada

- El tiempo de ejecución promedio de quickSort está mucho más cerca del mejor caso que de el peor de los casos.
- La clave es entender cómo el equilibrio de la partición se refleja en el recurrencia que describe el tiempo de ejecución.
- Suponga, por ejemplo, que el algoritmo de particionamiento siempre produce de 9 a 1 divisiones proporcionales, que a primera vista parecen bastante desequilibradas. Entonces obtenemos la recurrencia:

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

- en el tiempo de ejecución de quicksort, donde hemos incluido explícitamente la constante **c** oculta en el término **Θ(n)**.

- De hecho, cualquier división de proporcionalidad constante produce un árbol de recursividad de profundidad **Θ(ln n)**, donde el costo en cada nivel es **O(n).**
- Por lo tanto, el tiempo de ejecución es **O(n lg n)** siempre que la división tenga proporcionalidad.



A recursion tree for QUICKSORT in which PARTITION always produces a 9-to-1 split, yielding a running time of $O(n \lg n)$. Nodes show subproblem sizes, with per-level costs on the right. The per-level costs include the constant c implicit in the $\Theta(n)$ term.

- La figura anterior muestra el árbol de recursividad para esta recurrencia.
- Observe que cada nivel del árbol tiene un costo **cn**, hasta que la recursividad alcanza una condición de frontera en la profundidad $\log_{10} n = \Theta(\lg n)$, y luego los niveles tienen un costo como máximo **cn**.
- La recursividad termina en la profundidad $\log_{10/9} n = \Theta(\lg n)$.
- El costo total de la clasificación rápida es por tanto $O(n \lg n)$.
- Por lo tanto, con una división proporcional de 9 a 1 en cada nivel de recursividad, que intuitivamente parece bastante desequilibrada, quicksort se ejecuta en $O(n \lg n)$ tiempo (asintóticamente lo mismo que si la división estuviera justo por la mitad).
- En efecto, incluso una división de 99 a 1 produce un tiempo de ejecución de $O(n \lg n)$.
- De hecho, cualquier división por una constante proporcional produce un árbol de recursividad de profundidad $\Theta(\lg n)$, donde el costo en cada nivel es $O(n)$.
- Por lo tanto, el tiempo de ejecución es $O(n \lg n)$ siempre que la división tenga una constante proporcional.

Intuición para el caso promedio

- Para desarrollar una noción clara del comportamiento aleatorio de Quicksort, debemos hacer una suposición sobre la frecuencia con la que esperamos encontrar las diversas entradas.
- El comportamiento de QuickSort depende del orden relativo de los valores de los elementos en el arreglo de entrada, y no de los valores particulares del arreglo.
- Asumiremos por ahora que todas las permutaciones de los números de entrada son igualmente probables.
- Cuando ejecutamos Quicksort en un arreglo con entradas aleatoria, la partición es muy poco probable que suceda de la misma manera en todos los niveles, como ha supuesto nuestro análisis informal.

- Se espera entonces que algunas de las divisiones estén razonablemente bien equilibradas y que algunas otras estarán bastante desequilibradas.
- En el caso medio, la PARTICIÓN produce una mezcla de divisiones "buenas" y "malas".
- En un árbol de recursividad para una ejecución del caso promedio de PARTICIÓN, las divisiones buenas y malas se distribuyen aleatoriamente por todo el árbol.
- Supongamos, que las divisiones buenas y malas se dividen en niveles alternos en el árbol, y que las divisiones buenas son divisiones en el mejor de los casos y las divisiones malas son divisiones en el peor de los casos.

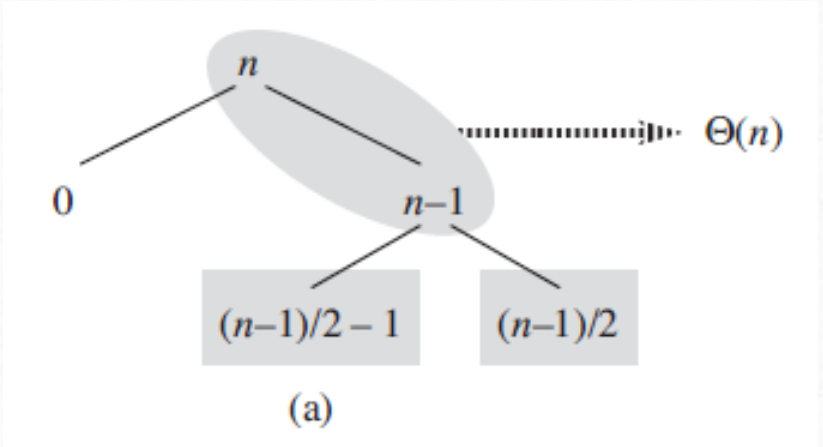
- La figura (a) muestra las divisiones en dos niveles consecutivos en el árbol de recursividad. En la raíz del árbol, el costo de la partición es ***n***, y los subarreglos producidos tienen tamaños ***n – 1*** y ***0***: en el peor caso.
- En el siguiente nivel, el subarreglo de tamaño ***n – 1*** sufre el mejor de los casos dividiéndose en subarreglos de tamaño ***(n – 1)/2 – 1*** y ***(n – 1)/2***.
- Supongamos que el costo de la condición de frontera es **1** para el subarreglo de tamaño **0**.

La combinación de la división incorrecta seguida de la división correcta produce tres subarreglos de tamaños:

- **0**
- ***(n – 1)/2 – 1***
- ***(n – 1)/2***

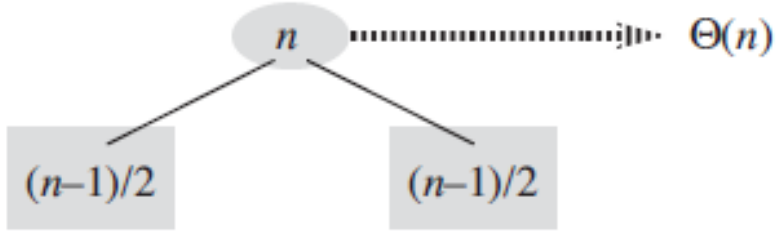
a un costo de partición combinado de

Θ(n) + Θ(n – 1) = Θ(n).



(a) Two levels of a recursion tree for quicksort. The partitioning at the root costs n and produces a "bad" split: two subarrays of sizes 0 and $n - 1$. The partitioning of the subarray of size $n - 1$ costs $n - 1$ and produces a "good" split: subarrays of size $(n - 1)/2 - 1$ and $(n - 1)/2$.

- La figura (b) es de un solo nivel de particionamiento, que produce dos subarreglos de tamaño **(n – 1)/2**, a un costo de **Θ(n)**.
- Intuitivamente, el costo **Θ(n – 1)** de la mala división se puede absorber en el costo **Θ(n)** de la buena división, y la división resultante es buena.
- Por lo tanto, el tiempo de ejecución de clasificación rápida, cuando los niveles alternan entre divisiones buenas y malas, es como el tiempo de ejecución de solo divisiones buenas: todavía **O(n lg n)**, pero con una constante ligeramente mayor oculta por la notación **O**.



(b)

(**b**) A single level of a recursion tree that is very well balanced. In both parts, the partitioning cost for the subproblems shown with elliptical shading is $\Theta(n)$. Yet the subproblems remaining to be solved in (a), shown with square shading, are no larger than the corresponding subproblems remaining to be solved in (b).

Una versión aleatoria de QuickSort

- Al explorar el comportamiento de caso promedio de quicksort, se ha hecho una suposición que todas las permutaciones de los números de entrada son igualmente probables. Sin embargo, no siempre podemos esperar que esta suposición se mantenga.
- A veces podemos agregar aleatorización a un algoritmo para obtener un buen rendimiento esperado en todas las entradas.
- Se considera la versión aleatoria de quicksort como el algoritmo de clasificación de elección para entradas lo suficientemente grandes.

- Para ello se va a usar una técnica de aleatorización en QuickSort llamada muestreo aleatorio.
- En lugar de usar siempre ***A[r]*** como pivote, seleccionaremos un elemento elegido al azar del subarreglo ***A[p..r]***.
- Lo hacemos intercambiando primero el elemento ***A[r]*** con un elemento elegido al azar de ***A[p..r]***.
- Muestreando aleatoriamente el rango ***p,….,r*** nos aseguramos de que el pivote ***x = A[r]*** es igualmente probable que sea cualquiera de los elementos ***r – p + 1*** en el subarreglo.
- Debido a que se elige aleatoriamente el elemento pivote, se espera que la división del arreglo de entrada deba estar en promedio razonablemente bien equilibrado.

- Los cambios en las funciones PARTITION y QUICKSORT son pequeños.
- En el procedimiento de la nueva partición, simplemente se implementa el intercambio antes de particionar la secuencia.

```
RANDOMIZED-PARTITION(A, p, r)
1  i = RANDOM(p, r)
2  exchange A[r] with A[i]
3  return PARTITION(A, p, r)
```

- El procedimiento nuevo QuickSort llama a RANDOMIZED-PARTITION en lugar de PARTITION.

```
RANDOMIZED-QUICKSORT(A, p, r)
1  if p < r
2      q = RANDOMIZED-PARTITION(A, p, r)
3      RANDOMIZED-QUICKSORT(A, p, q − 1)
4      RANDOMIZED-QUICKSORT(A, q + 1, r)
```