Randomized Quick Sort

Outlines

- ✓ Introduction
- ✓ Algorithm
- ✓ Analysis

Randomizing Quicksort

Randomly permute the elements of the input array before sorting.

OR ... modify the PARTITION procedure

At each step of the algorithm we exchange element $A[p]$ with an element chosen at random from $A[p \ldots r]$.

The pivot element $x = A[p]$ is equally likely to be any one of the $r - p + 1$ elements of the subarray.

Randomized Algorithms

No input can produce worst case behavior.

Worst case occurs only if we get “unlucky” numbers from the random number generator.

Randomization can NOT eliminate the worst-case but it can make it less likely!

Randomized PARTITION

Alg: RANDOMIZED-PARTITION(A, p, r)

$i \leftarrow$ RANDOM(p, r)

Exchange $A[p] \leftrightarrow A[i]$

return PARTITION(A, p, r)

Randomized Quicksort

Alg : RANDOMIZED-QUICKSORT(A, p, r)

if (p < r) **then**

$q \leftarrow$ RANDOMIZED-PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, q)

RANDOMIZED-QUICKSORT(A, q + 1, r)

Formal Worst-Case Analysis of Quicksort

$T(n)$ = Worst-case running time

$$T(n) = \max_{1 \le q \le n-1} (T(q) + T(n-q)) + \Theta(n)$$

Use Substitution method to show that the running time of Quicksort is $O(n^2)$.

Guess $T(n) = O(n^2)$

Induction goal: $T(n) \le cn^2$

Induction hypothesis: $T(k) \le ck^2$ for any $k < n$

Worst-Case Analysis of Quicksort

Proof of induction goal:

$$T(n) \le \max_{1 \le q \le n-1} (cq^2 + c(n-q)^2) + \Theta(n) = c \cdot \max_{1 \le q \le n-1} (q^2 + (n-q)^2) + \Theta(n)$$

The expression $q^2 + (n-q)^2$ achieves a maximum over the range $1 \le q \le n-1$ at one of the endpoints.

$$\max_{1 \le q \le n-1} (q^2 + (n - q)^2) = 1^2 + (n - 1)^2 = n^2 - 2(n - 1)$$

$$T(n) \le cn^2 - 2c(n - 1) + \Theta(n)$$

$$\le cn^2$$

Revisit Partitioning

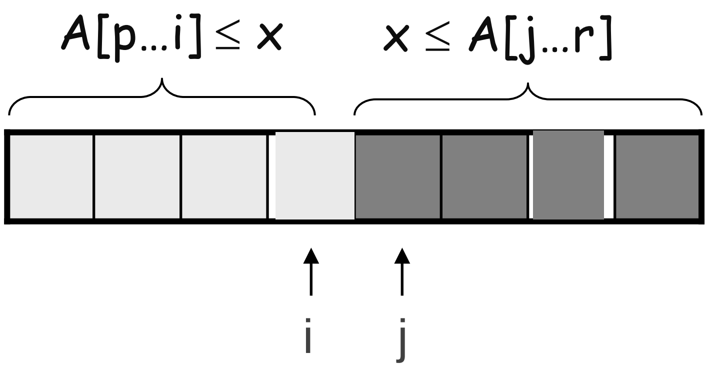
Hoare's Partition:

Select a pivot element x around which to partition.

Grows two regions

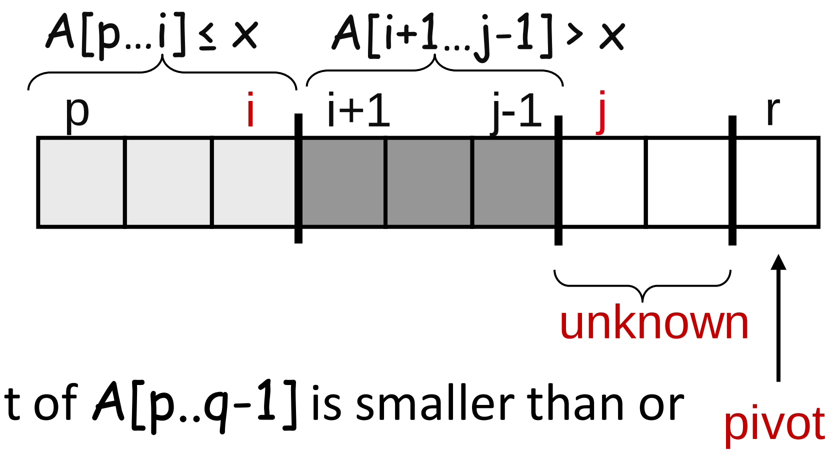
$A[p \ldots i] \leq x$

$x \leq A[j \ldots r]$



Another Way to PARTITION (Lomuto's Partition)

Given an array A, partition the array into the following subarrays:



- A pivot element $x = A[q]$
- Subarray $A[p..q-1]$ such that each element of $A[p..q-1]$ is smaller than or equal to x (the pivot).
- Subarray $A[q+1..r]$, such that each element of $A[p..q+1]$ is <u>strictly</u> greater than x (the pivot).

The pivot element is <u>not included</u> in any of the two subarrays.

Another Way to PARTITION (cont'd)

Alg.: PARTITION(*A, p, r)*

$x \leftarrow A[r]$

$i \leftarrow p - 1$

for $j \leftarrow p$ **to** $r - 1$

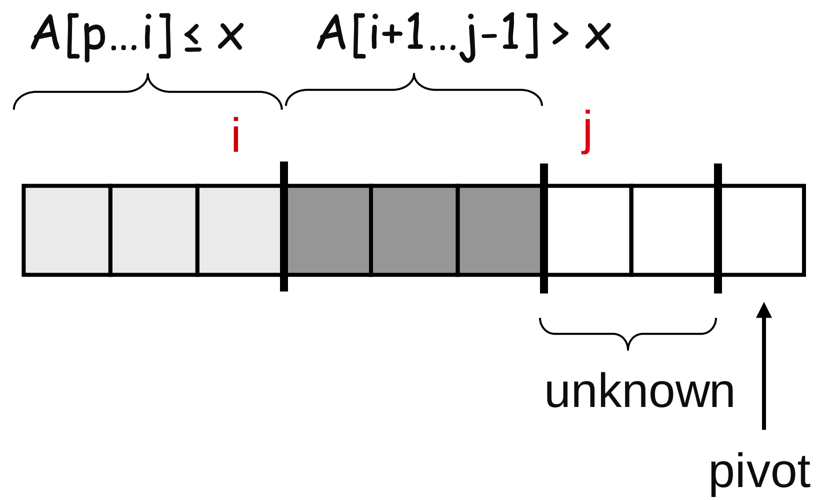
  do if $A[\, j\,] \le x$

    then $i \leftarrow i + 1$

      exchange $A[i] \leftrightarrow A[j]$

exchange $A[i + 1] \leftrightarrow A[r]$

return $i + 1$



Chooses the last element of the array as a pivot
Grows a subarray [p..i] of elements ≤ x
Grows a subarray [i+1..j-1] of elements >x
Running Time: $\Theta(n)$, where $n=r-p+1$

Randomized Quicksort (using Lomuto's partition)

Alg. : RANDOMIZED-QUICKSORT(A, p, r)

if p < r **then**

$q \leftarrow$ RANDOMIZED-PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, $q - 1$)

RANDOMIZED-QUICKSORT(A, $q + 1$, r)

The pivot is <u>no longer</u> included in any of the subarrays!!

Analysis of Randomized Quicksort

Alg. : RANDOMIZED-QUICKSORT(A, p, r)

The running time of Quicksort is dominated by PARTITION !!

if p < r

then $q \leftarrow$ RANDOMIZED-PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, q - 1)

RANDOMIZED-QUICKSORT(A, q + 1, r)

PARTITION is called at most n times

(at each call a pivot is selected and never again included in future calls)

PARTITION

Alg.: PARTITION*(A, p, r)*

$x \leftarrow A[r]$

$i \leftarrow p - 1$

for $j \leftarrow p$ **to** $r - 1$

do if $A[\,j\,] \le x$

then $i \leftarrow i + 1$

exchange $A[i] \leftrightarrow A[j]$

exchange $A[i + 1] \leftrightarrow A[r]$

return $i + 1$

O(1) - constant

# of comparisons: X_k between the pivot and the other elements

Amount of work at call k: $c + X_k$

Average-Case Analysis of Quicksort

Let X = **total number of comparisons performed in all calls to PARTITION:**

$$X = \sum_k X_k$$

The total work done over the **entire** execution of Quicksort is

O(nc+X)=O(n+X)

Need to estimate E(X)

Problem

Analyze the complexity of the following function:

```
F(i) {
 if i=0
   then return 1
 return (2*F(i-1)) }
```

Recurrence: $T(n)=T(n-1)+c$

Use iteration to solve it .... $T(n)=\Theta(n)$