Analysis of Algorithms 1 (Fall 2013) Istanbul Technical University Computer Eng. Dept.

Chapter 7: Quicksort

Course slides from Susan Bridges @MS State have been used in preparation of these slides.

Last updated: October 25, 2011

Purpose

- Learn about quicksort algorithm
- Learn about important subroutine used by quicksort for partitioning
- Analyze randomized quicksort

Outline

- Description of Quicksort and Partition
- Intuitive Discussion of Performance of Quicksort
- Version of Quicksort that Uses Random Sampling
- Analysis of Randomized Quicksort

Why Quicksort?

- Popular algorithm for sorting large input arrays
- Worst-case running time $\Theta(n^2)$ on input array of n numbers
- Slow worst-case running time, but often best practical choice for sorting because remarkably efficient on average
  - expected running time is $\Theta(n \lg n)$, and constant factors hidden in $\Theta(n \lg n)$ notation are quite small
- Advantage of sorting in place
- Works well even in virtual memory environments

Description of Quicksort

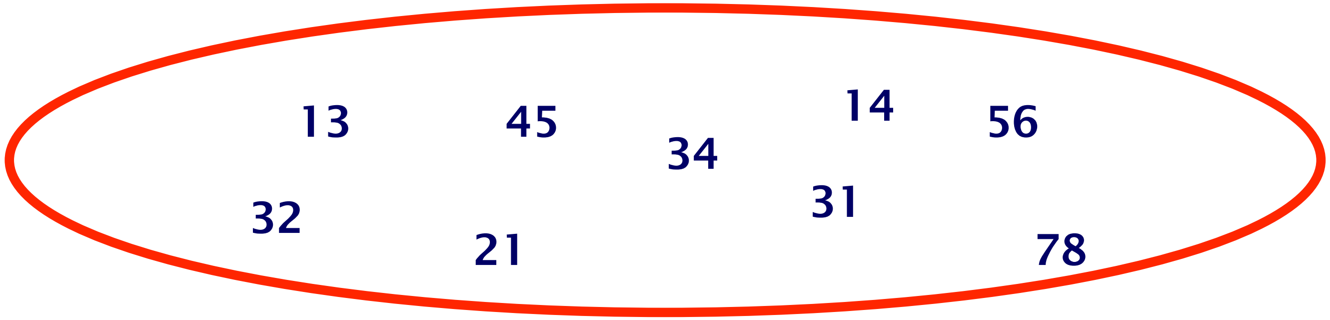
- Like merge sort, based on divide-and-conquer paradigm
- **Divide:**
  - Partition A[p..r] into two (possibly empty) subarrays A[p..q-1] and A[q+1 .. r] such each element of A[p..q-1] ≤ A[q] and A[q] ≤ each element of A[q+1..r] (A[q] called pivot)
  - Compute the index q as part of this partitioning procedure
- **Conquer:**
  - Sort the two subarrays by recursive calls to quicksort
- **Combine:**
  - Since the subarrays are sorted in place, no work is needed to combine them: A[p..r] is now sorted

Quicksort Algorithm

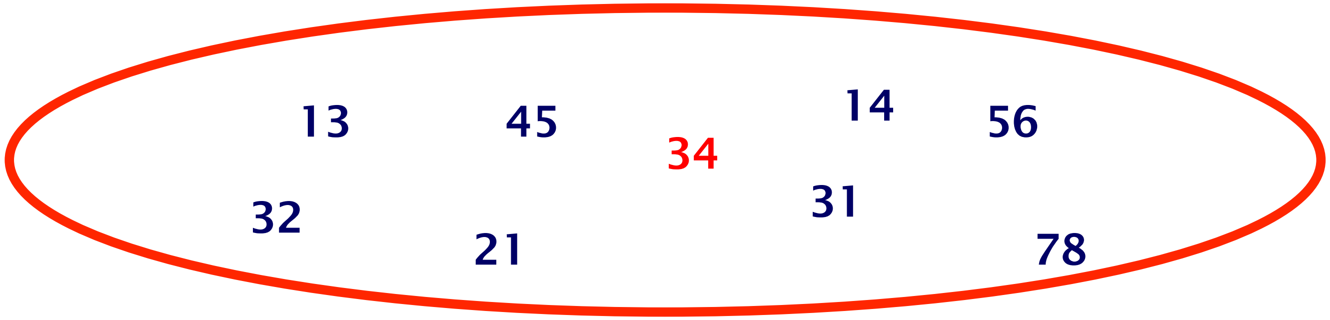
```
QUICKSORT(A,p,r)
if p < r
   then q ← PARTITION(A,p,r)
        QUICKSORT(A,p,q-1)
        QUICKSORT(A,q+1,r)
```

Initial call: QUICKSORT(A,1, length[A])

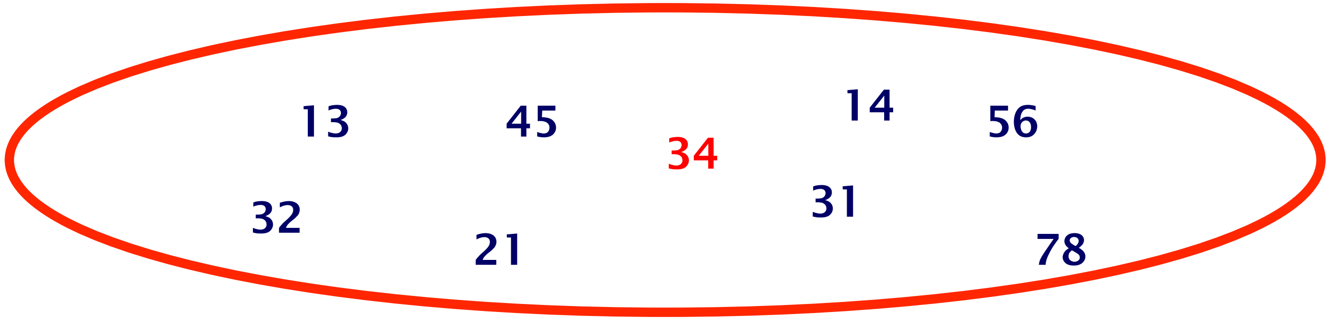
Quicksort Example



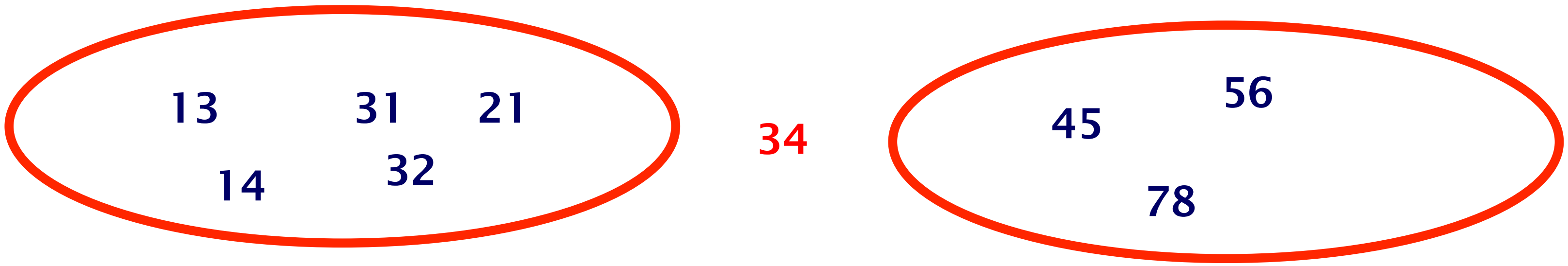
Select Pivot



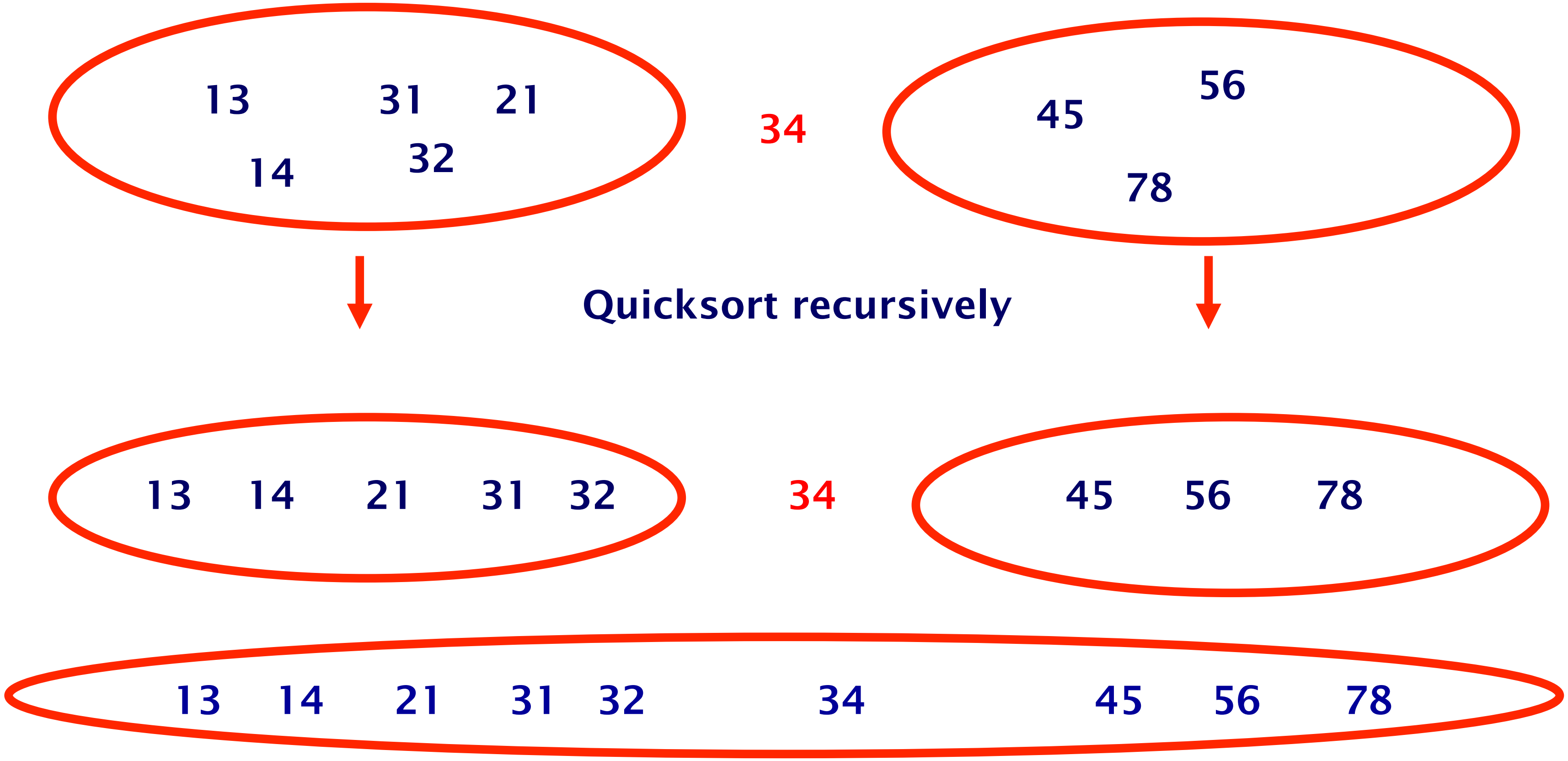
Quicksort Example



Partition around Pivot



Quicksort Example

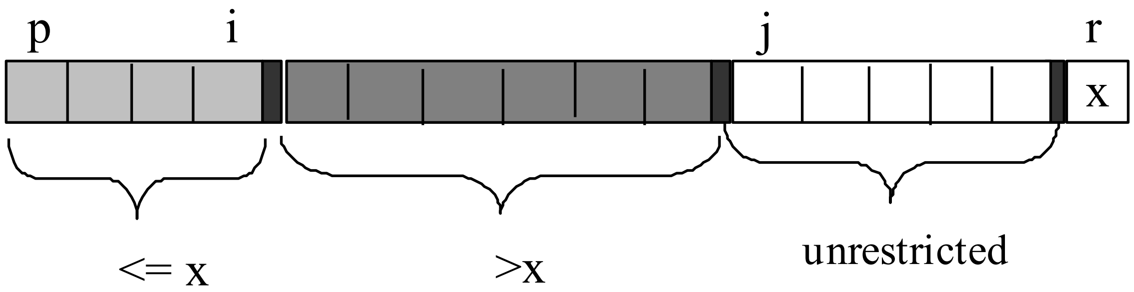


Partition Algorithm

- Rearranges subarray A[p..r] in place

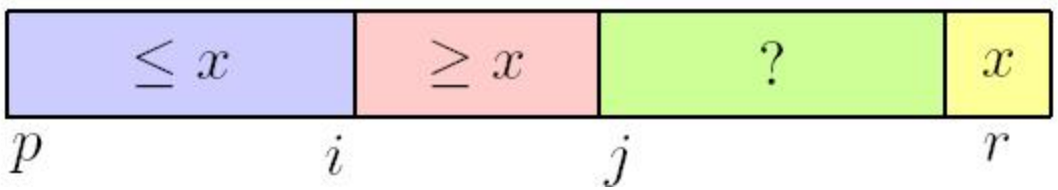
```
PARTITION(A,p,r)
x ← A[r]
i ← p - 1
for j ← p to r-1
     do if A[j] ≤ x
        then i ← i + 1
             exchange A[i] ↔ A[j]
exchange A[i+1] ↔ A[r]
return i+1
```

Regions of Subarray Maintained by PARTITION

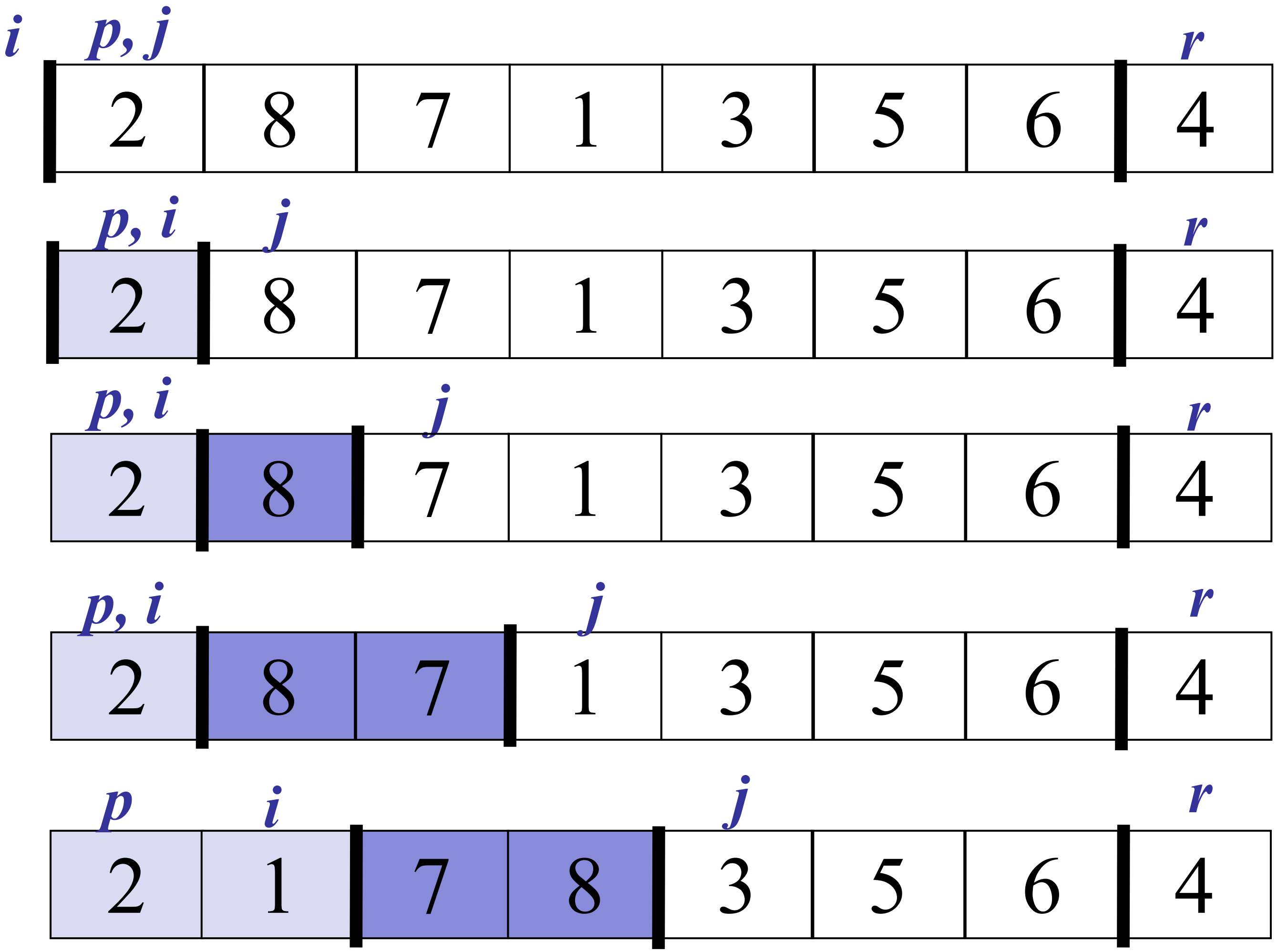


1. Each value in A[p..i] ≤ x
2. Each value in A[i+1..j-1] > x
3. A[r] = x
4. A[j..r-1] can take on any values

Invariant:

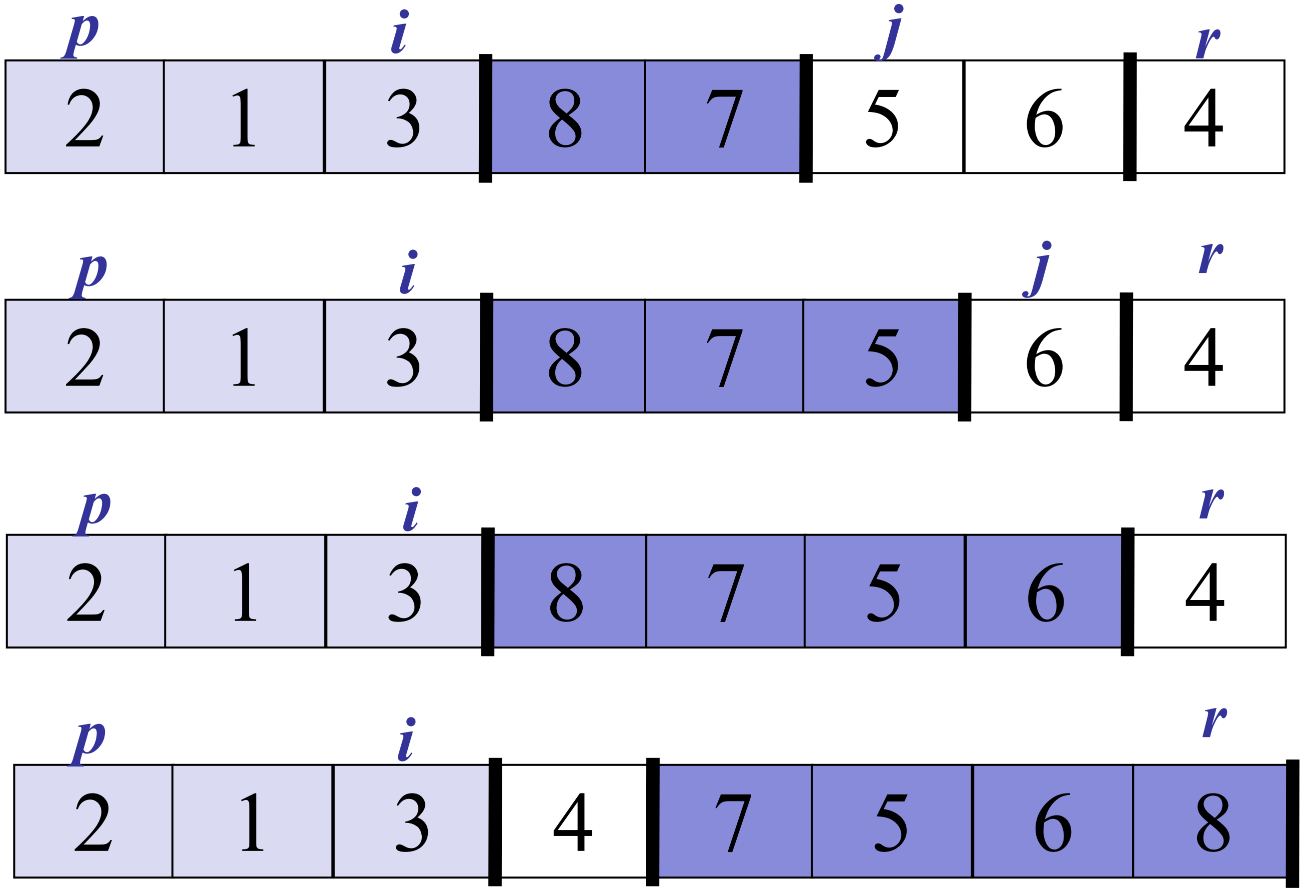


Example of Partition



cont.->

Example of Partition (cont.)



Loop Invariant for Partition

- At beginning of each iteration of loop in lines 3-6, for any array index *k:*
  1. If $p \leq k \leq i$, then $A[k] \leq x$
  2. If $i + 1 \leq k \leq j-1$, then $A[k] > x$
  3. If $k = r$, then $A[k] = x$

```
PARTITION(A,p,r)
1  x ← A[r]
2  i ← p - 1
3  for j ← p to r-1
4       do if A[j] ≤ x
5          then i ← i + 1
6                exchange A[i] ↔ A[j]
7   exchange A[i+1] ↔ A[r]
8   return i+1
```

Loop Invariant Correctness

- We need to show that
  - this loop invariant is true prior to first iteration
  - each iteration of loop maintains invariant
  - invariant provides a useful property to show correctness when loop terminates

Loop Invariant Correctness: Initialization

- Prior to first iteration of loop, $i = p - 1$, and $j = p$
- There are no values between p and i, and no values between $i + 1$ and $j - 1$, so first two conditions of loop invariant are trivially satisfied
- Assignment in line 1 satisfies third condition

Loop Invariant Correctness: Maintenance

- Two cases to consider depending on outcome of test in line 4
- When A[j]>x
  - Only action in loop is to increment j
  - After j is incremented, condition 2 holds for all A[j-1] and all other entries remain unchanged
- When A[j] ≤ x
  - i is incremented, A[i] and A[j] are swapped, and then j is incremented
  - Because of the swap, we now have that A[i] ≤ x, and condition 1 is satisfied
  - Similarly, we also have that A[j-1]>x, since item that was swapped into A[j-1] is, by loop invariant, greater than x

Loop Invariant Correctness: Termination

- At termination, $j = r$
- Therefore, every entry in the array is in one of the three sets described by the invariant, and we have partitioned the values in the array into three sets:
  - those less than or equal to x
  - those greater than x
  - singleton set containing x

Partition

- Final two lines move pivot element into its place in middle of array by swapping it with leftmost element greater than x
- Output now satisfies specifications given for the divide step
- Running time on A[p..r] is $\Theta(n)$, where $n = r - p + 1$

```
PARTITION(A,p,r)
x ← A[r]
 i ← p - 1
 for j ← p to r-1
        do if A[j] ≤ x
           then i ← i + 1
                  exchange A[i] ↔ A[j]
  exchange A[i+1] ↔ A[r]
  return i+1
```

Performance of Quicksort

- Depends on whether partitioning is balanced or unbalanced:
  - **Worst case:** Each time partitioning is done, one subarray contains *n* -1 of *n* elements from previous call and the other is empty
  - **Best case:** Each time partitioning is done, each subarray contains n/2 of elements from previous call

Worst Case

Cost of Partition: $\Theta(n)$

Recurrence for Quicksort:

$$T(n) = T(n-1) + T(0) + \Theta(n)$$
$$= T(n-1) + \Theta(n)$$

Worst Case (cont.)

Solving recurrence by iteration:

$$
\begin{aligned}
T(n) &= \Theta(n) + T(n-1) \\
&= \Theta(n) + \Theta(n-1) + \Theta(n-2) + \ldots \ + \Theta(1) \\
&= \sum_{k=1}^{n} \Theta(k) \\
&= \Theta\left(\sum_{k=1}^{n} k\right) \\
&= \Theta(n^2)
\end{aligned}
$$

Best Case

Recurrence for Quicksort:

$$T(n) \le 2T(n/2) + \Theta(n)$$

Solving recurrence by Master Method case 2:

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

Average Case

- Suppose split is always 9-to-1
- Recurrence:

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

- Solving recurrence by recursion tree:

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

Randomized Version of Quicksort

- When an algorithm has average case performance and worst case performance that are very different, we can try to minimize odds of encountering worst case
- For quicksort: Randomly choose pivot element in A[p..r]

Randomized PARTITION

RANDOMIZED-PARTITION (A, p, r)

1 $i \leftarrow$ RANDOM (p, r)

2 exchange $A[r] \leftrightarrow A[i]$

3 **return** PARTITION (A, p, r)

Randomized Quicksort

```
RANDOMIZED-QUICKSORT (A, p, r)
    if p < r
      then q ← RANDOMIZED-PARTITION (A, p, r)
           RANDOMIZED-QUICKSORT (A, p, q-1)
           RANDOMIZED-QUICKSORT (A, q+1, r)
```

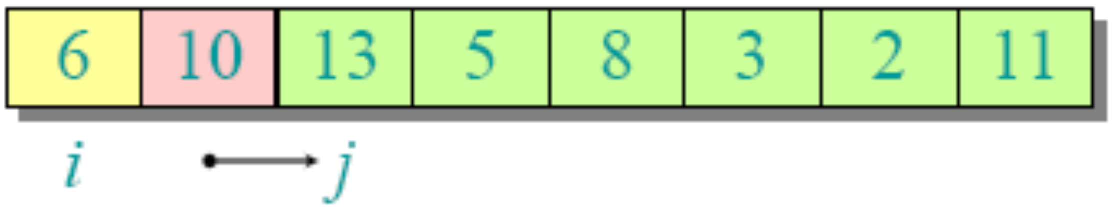
Summary

- Description of Quicksort
- Intuitive Discussion of Performance of Quicksort
- Version of Quicksort that Uses Random Sampling
- Analysis of Randomized Quicksort

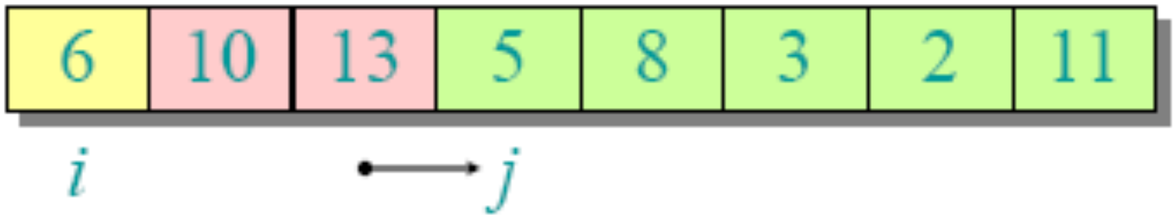
Example of Partitioning

6	10	13	5	8	3	2	11
i	j						

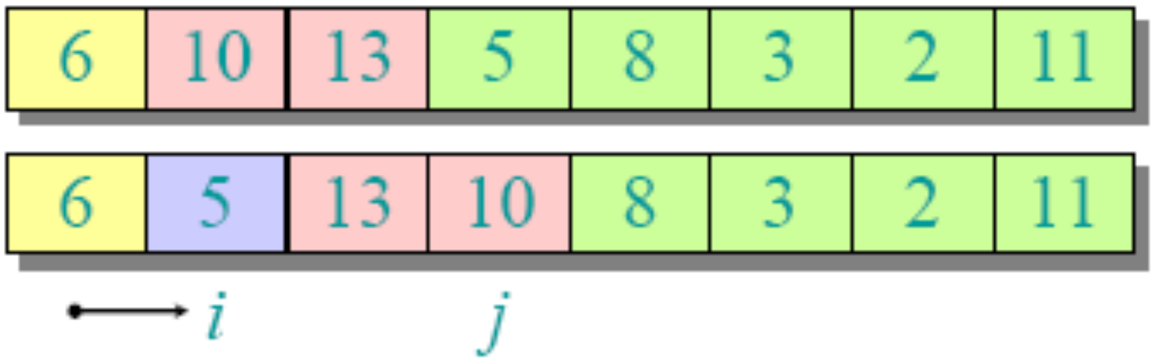
Example of Partitioning



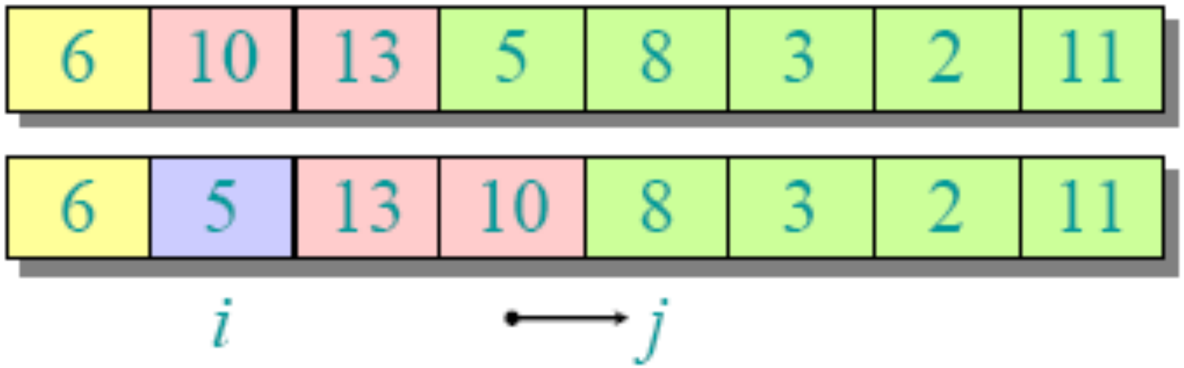
Example of Partitioning



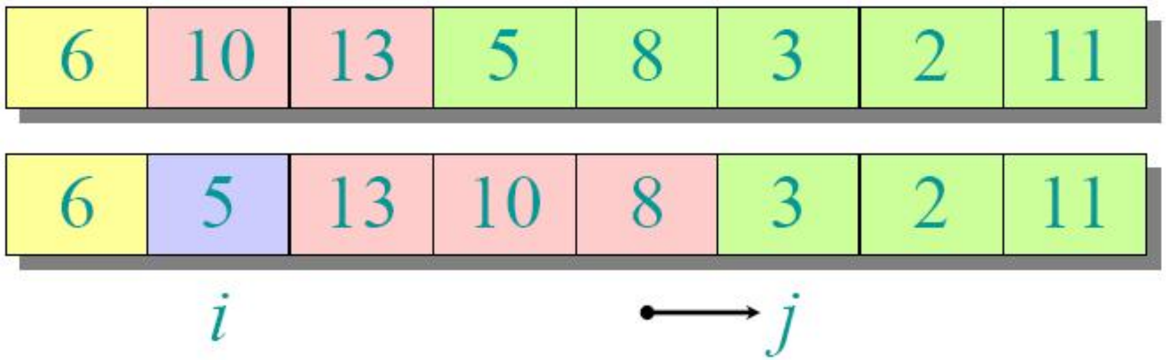
Example of Partitioning



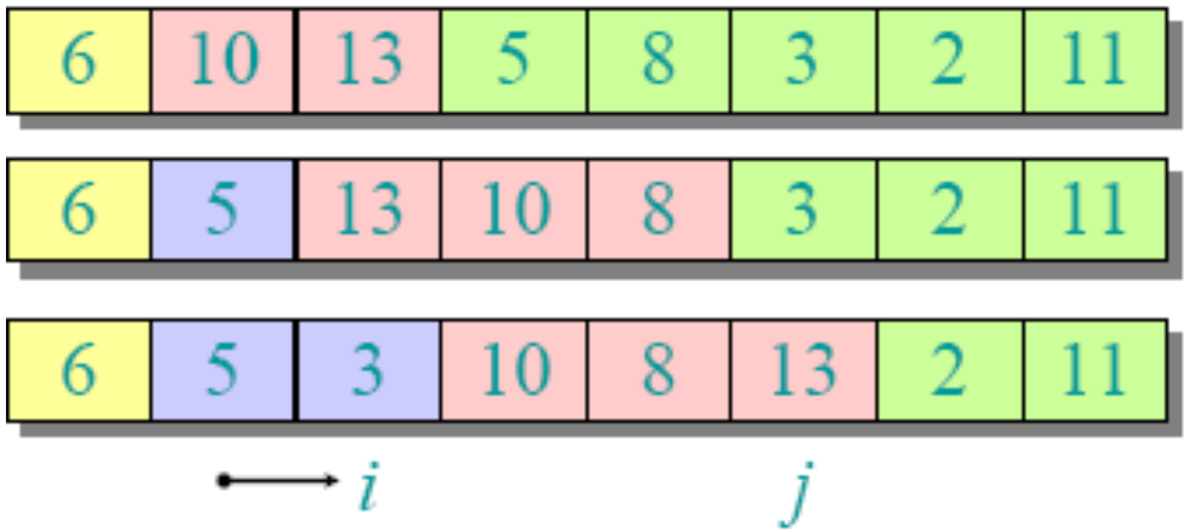
Example of Partitioning



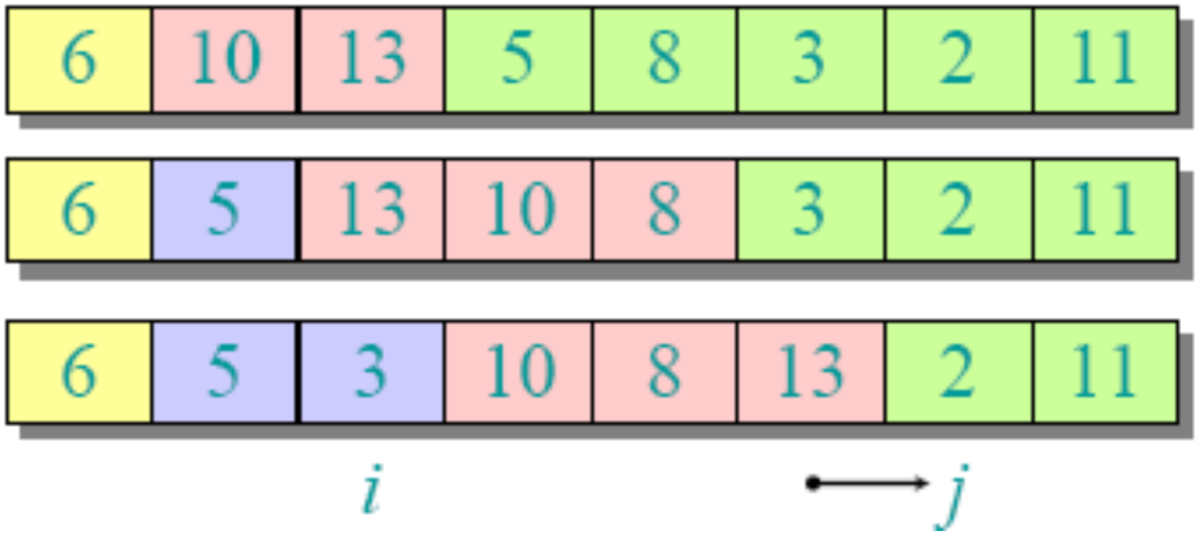
Example of Partitioning



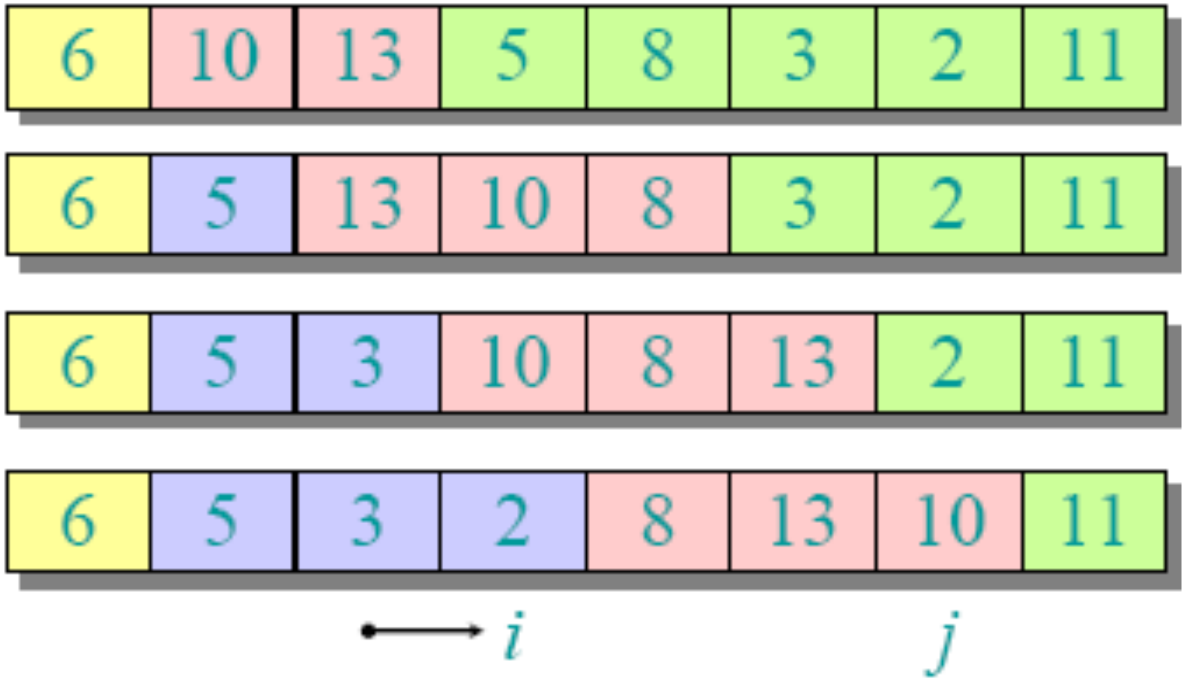
Example of Partitioning



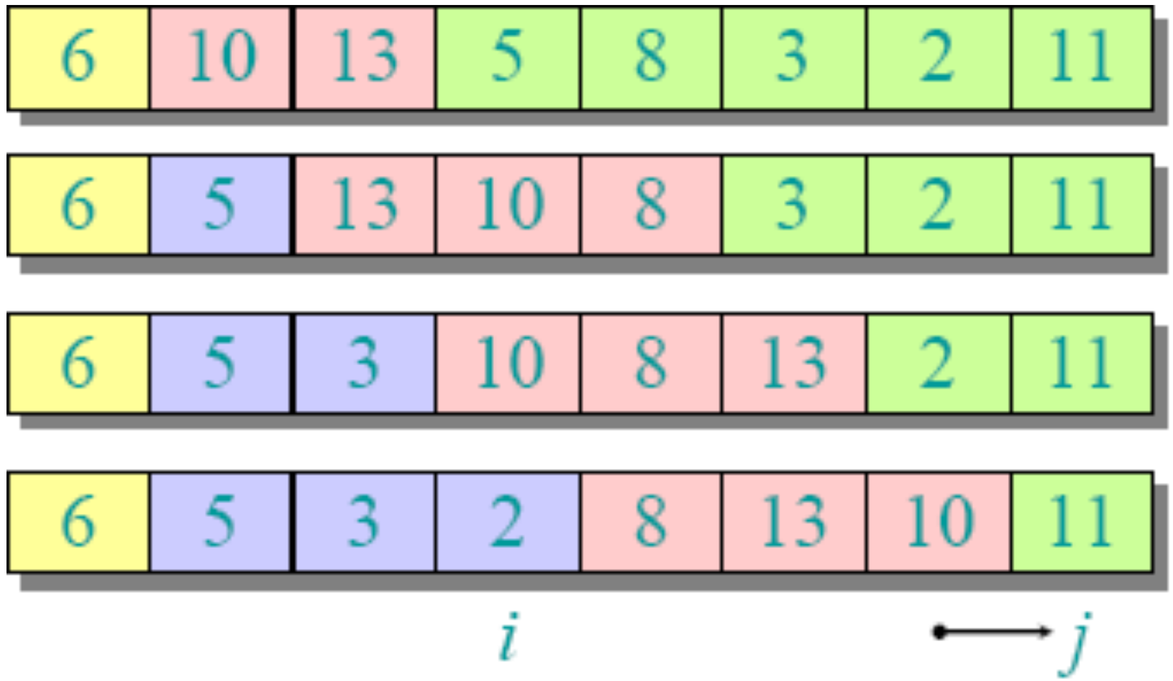
Example of Partitioning



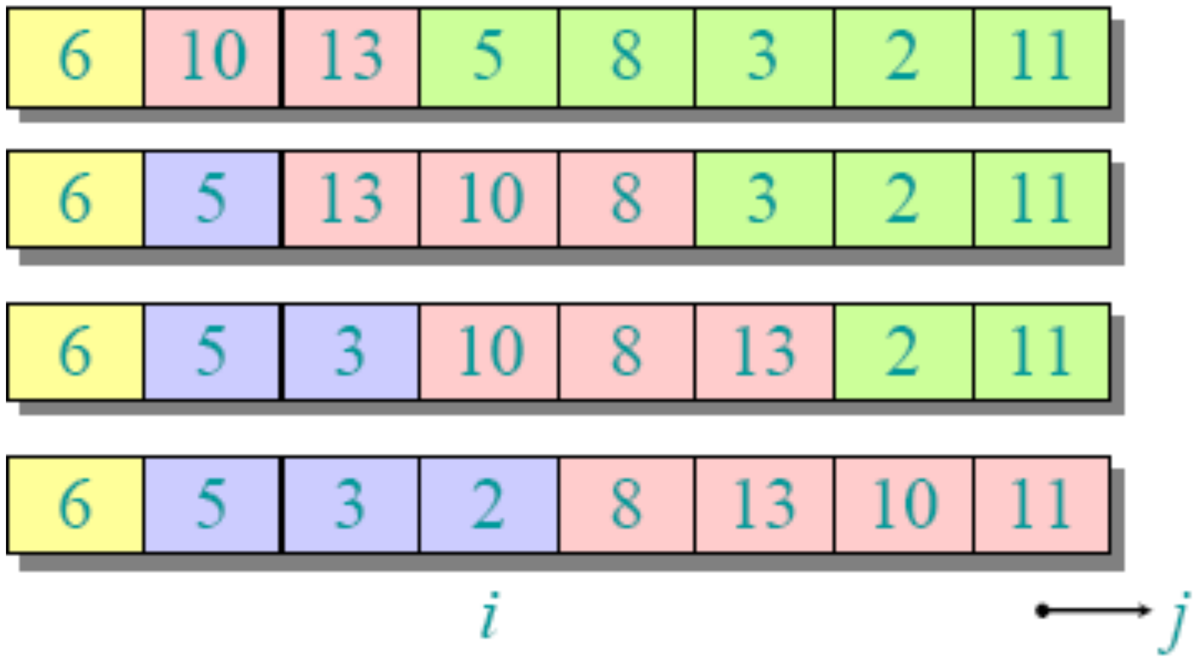
Example of Partitioning



Example of Partitioning



Example of Partitioning



Example of Partitioning