Sorting (contd.)

Reference: "Data Structures and Algorithm Analysis", C. Shaffer, pp. 223–244; skip sections 7.2.2 and 7.3.

QuickSort

- Due to Hoare
- Fast in-memory sorting algorithm on the average
- Idea is to choose a "pivot" element "e" in S and divide the original sequence S into

  $S1 = \{a \in S \mid a < e\}$

  $S2 = \{a \in S \mid a = e\}$

  $S3 = \{a \in S \mid a > e\}$

  Recursively sort S1 and S3 and the concatenated sorted subsequences S1, S2 and S3
- If we have an array for S, we can do all the above steps in-place unlike merge sort.

```
static <E extends Comparable<E>>
     void qsort(E[] A, int i, int j) { // Quicksort
  int pivotindex = findpivot(A, i, j); // Pick a pivot
  swap(A, pivotindex, j); // Stick pivot at end
  // k will be the first position in the right subarray
  int k = partition(A, i - 1, j, A[j]);
  swap(A, k, j); // Put pivot in place
  if ((k - i) > 1)
    qsort(A, i, k - 1); // Sort left partition
  if ((j - k) > 1)
    qsort(A, k + 1, j); // Sort right partition
}
```

static <E extends Comparable<E>> findPivot(E [] A, int *i*, int j) {
    return *i*; // if choosing some other pivot swap it with first position i
}

Source : Dr. Calvin & textbook

Quicksort Partition

1. Choose a pivot element (example: rightmost element).
2. Initialize pointers to start and end of array.
3. Move left pointer right until encounter element $\geq$ pivot.
4. Move right pointer left until encounter element $<$ pivot.
5. Swap elements pointed to.
6. Repeat until left, right pointers cross.

The cost to partition array of length n is $O(n)$.

Source : Dr. Calvin & textbook

Algorithm for Partitioning

If the array is randomly ordered, it does not matter which element is the pivot.

For simplicity we pick the element with subscript `first`

Trace of Partitioning (cont.)

If the array is randomly ordered, it does not matter which element is the pivot.

For simplicity we pick the element with subscript `first`

Trace of Partitioning (cont.)

For visualization purposes, items less than or equal to the pivot will be colored blue; items greater than the pivot will be colored white

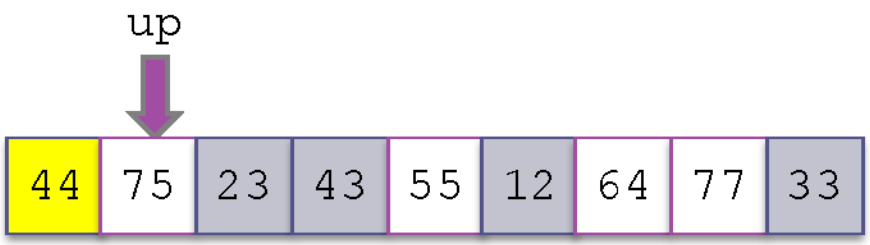
Trace of Partitioning (cont.)

For visualization purposes, items less than or equal to the pivot will be colored blue; items greater than the pivot will be colored white

Trace of Partitioning (cont.)

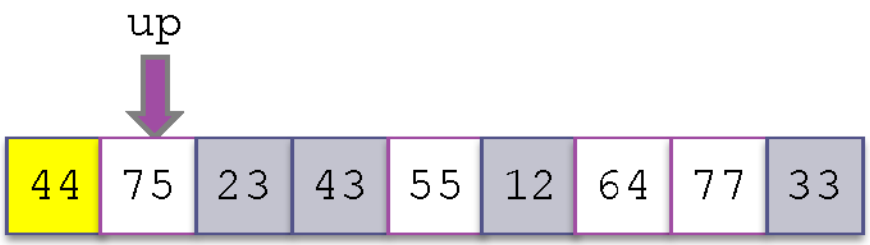
Search for the first value at the left end of the array that is greater than the pivot value

Trace of Partitioning (cont.)



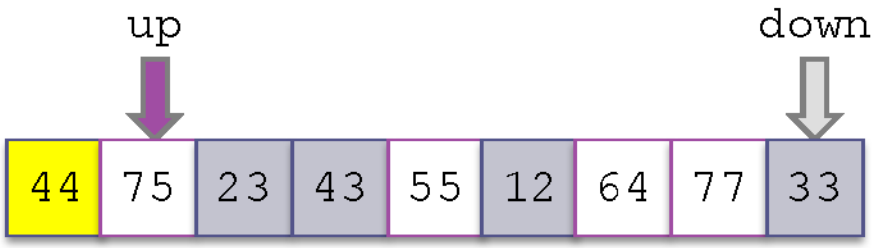
Search for the first value at the left end of the array that is greater than the pivot value

Trace of Partitioning (cont.)



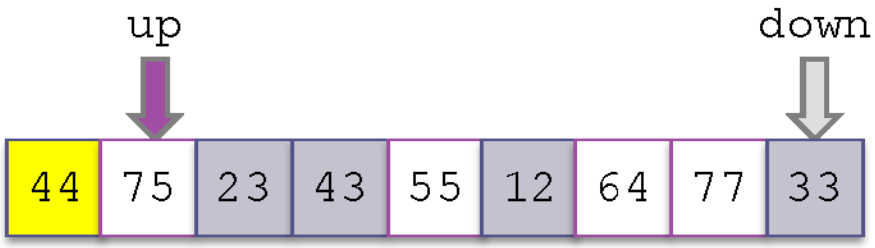
Then search for the first value at the right end of the array that is less than or equal to the pivot value

Trace of Partitioning (cont.)



Then search for the first value at the right end of the array that is less than or equal to the pivot value

Trace of Partitioning (cont.)



Exchange these values

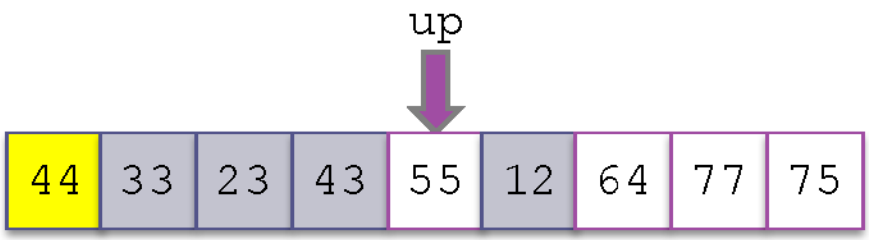
Trace of Partitioning (cont.)

Exchange these values

Trace of Partitioning (cont.)

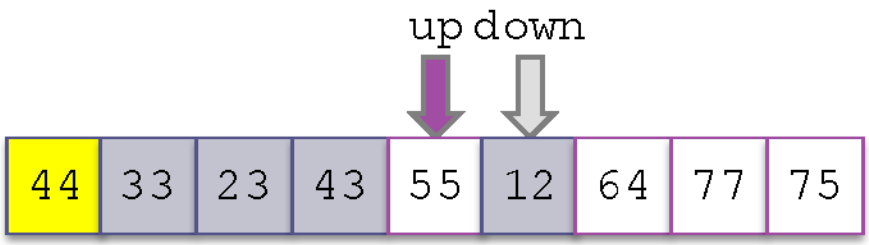
Repeat

Trace of Partitioning (cont.)



Find first value at left end greater than pivot

Trace of Partitioning (cont.)



Find first value at right end less than or equal to pivot

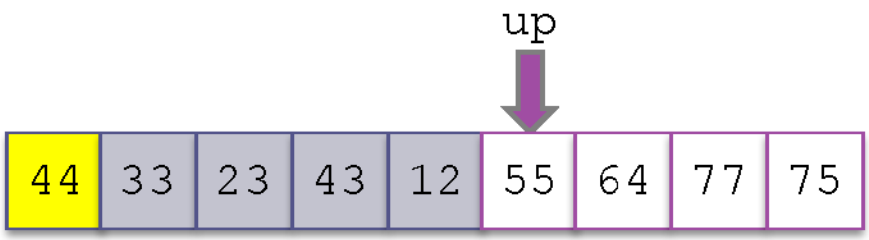
Trace of Partitioning (cont.)

Exchange

Trace of Partitioning (cont.)

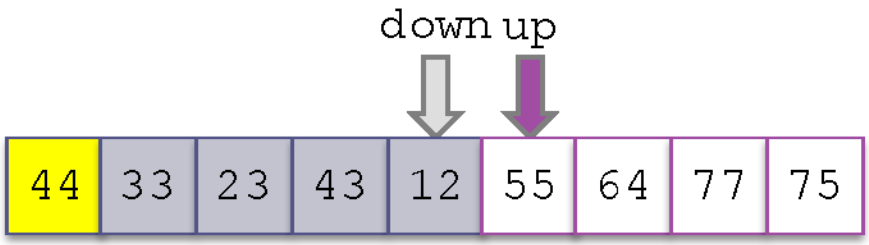
Repeat

Trace of Partitioning (cont.)



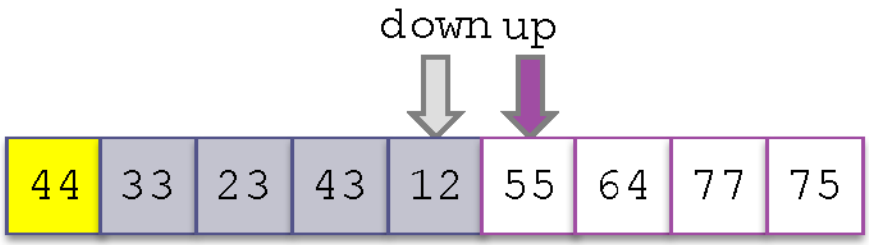
Find first element at left end greater than pivot

Trace of Partitioning (cont.)



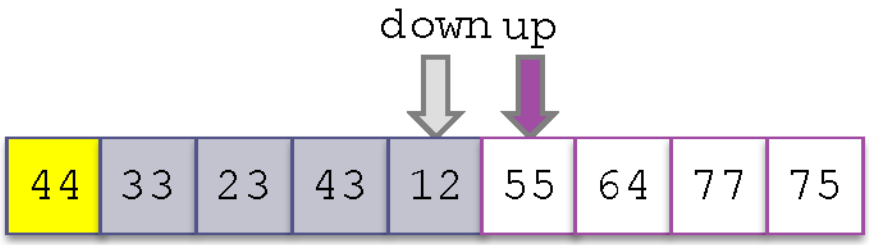
Find first element at right end less than or equal to pivot

Trace of Partitioning (cont.)



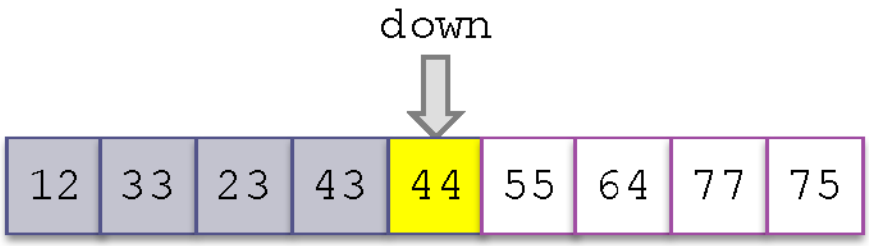
Since down has "passed" up, do not exchange

Trace of Partitioning (cont.)



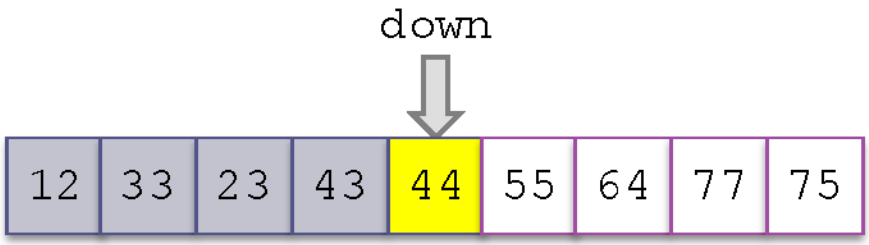
Exchange the pivot value with the value at down

Trace of Partitioning (cont.)



Exchange the pivot value with the value at down

Trace of Partitioning (cont.)



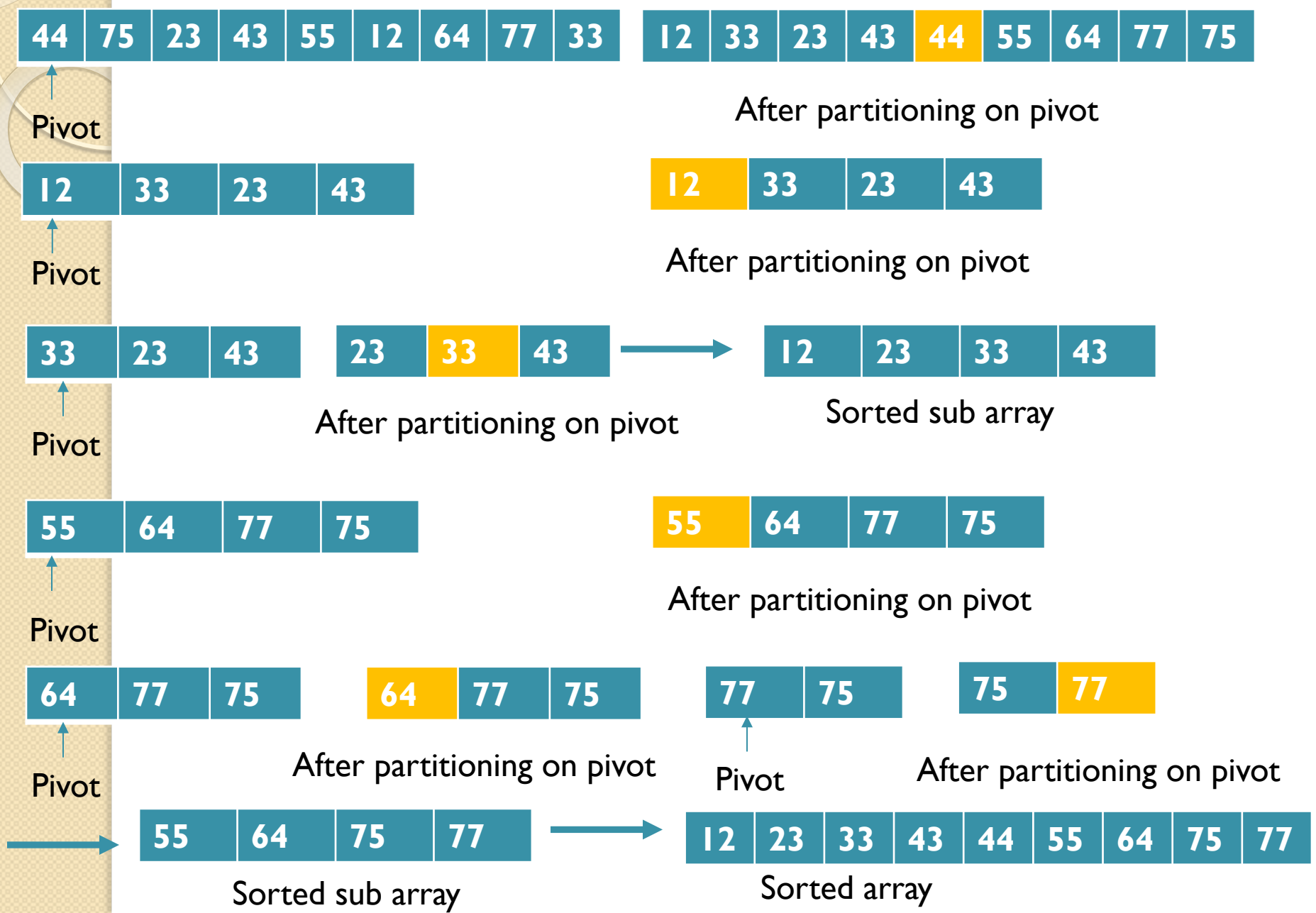
The pivot value is in the correct position; return the value of `down` and assign it to the pivot index `pivIndex`

```
static <E extends Comparable<E>>
     int partition(E[] A, int l, int r, E pivot) {

  do { // Move bounds inward until they meet
    while (A[++l].compareTo(pivot) < 0)
      ;
    while ((r != 0) && (A[--r].compareTo(pivot) >= 0))
      ;
    swap(A, l, r); // Swap out-of-place values
  } while (l < r); // Stop when they cross
  swap(A, l, r); // Reverse last, wasted swap
  return l; // Return first position in right partition
}
```

Source : Dr. Calvin & textbook

Quicksort example



Quicksort time complexity

Best case: Always partition in half : $O(n \lg n)$.

Worst case: Bad partition : $\Theta\left(n^2\right)$.

Average case: $\Theta\left(n \lg n\right)$ if each input order equally likely.

Randomized Quicksort: choose the pivot element at random, each element of the subarray with the same probability. Then average time is $\Theta\left(n \lg n\right)$ (regardless of input).

Source : Dr. Calvin & textbook

Selection problem

- How long does it take find maximum or minimum in a set of n elements ?

  O(n) time even in the best-case as we need to make $n - 1$ comparisons

- How do we find the k-th smallest element (i.e smallest element larger than at least k elements) ?

- These are called "order statistic" problems. Finding "median" is same as finding the [n/2]-th smallest element in a set of n-elements.

Finding k-th smallest element

- Of course by sorting we can find all these elements. *This takes O(n log n) time using best sorting algorithm.*
- Another approach is to build a min-heap and then do extractMin() operations k times.

This takes O(n+k log n) time, slight improvement over sorting when k is small.

- Better approach is to use divide-and-conquer similar to Quicksort.

Choose a pivot e and divide S into 3 sets $S1, S2\ and\ S3$.

(a) if $k \leq |S1|$, recursively find k-th smallest element in S1

(b) if $|S1| < k \leq |S1| + |S2|$, 'e' is k-th smallest element

(c) Otherwise recursively $find\ k - (|S1| + |S2|)$ −th smallest element in $S3$.

Selection example

- S has 12 elements using pivot e

$$|S1| = 5, |S2| = 3 \; |S3| = 4$$

3rd smallest element must be in $S1$

6th smallest element = pivot element

10th smallest element -- 2nd smallest element in $S3$

```
static <E extends Comparable<E>>
     int randomized-partition(E[] A, int l, int r) {

  //rand is an object of class Random
  int k = rand.nextInt(r-l+1)+l; // k is random from l to
  E pivot = A[k];
  do { // Move bounds inward until they meet
    while (A[++l].compareTo(pivot) < 0)
      ;
    while ((r != 0) && (A[--r].compareTo(pivot) >= 0))
      ;
    swap(A, l, r); // Swap out-of-place values
  } while (l < r); // Stop when they cross
  swap(A, l, r); // Reverse last, wasted swap
  return l; // Return first position in right partition
}
```

Source : Dr. Calvin & textbook

// Find i-th smallest element in the subarray A[p….r]

RandomizedSelect(A,p,r,i)

if $p = r$ then return $A[p]$
q ← **Randomized-Partition**(A,p,r)
$k \leftarrow q - p + 1$ // k = |$S1$| + |$S2$|
if $i = k$ then return A[q] // pivot is i-th smallest
else if $i < k$ then
    return **Randomized-Select**(A,p,q-1,i)
else return **Randomized-Select**(A,q+1,r,i-k)

The worst-case running time of Randomized-Select is $\Theta(n^2)$, but on average it is $O(n)$.

There is a deterministic selection algorithm that runs in $O(n)$ time in the worst case.

This works by choosing median of medians of 5-element groups as the pivot during each recursive invocation.

Source : Dr. Calvin & textbook