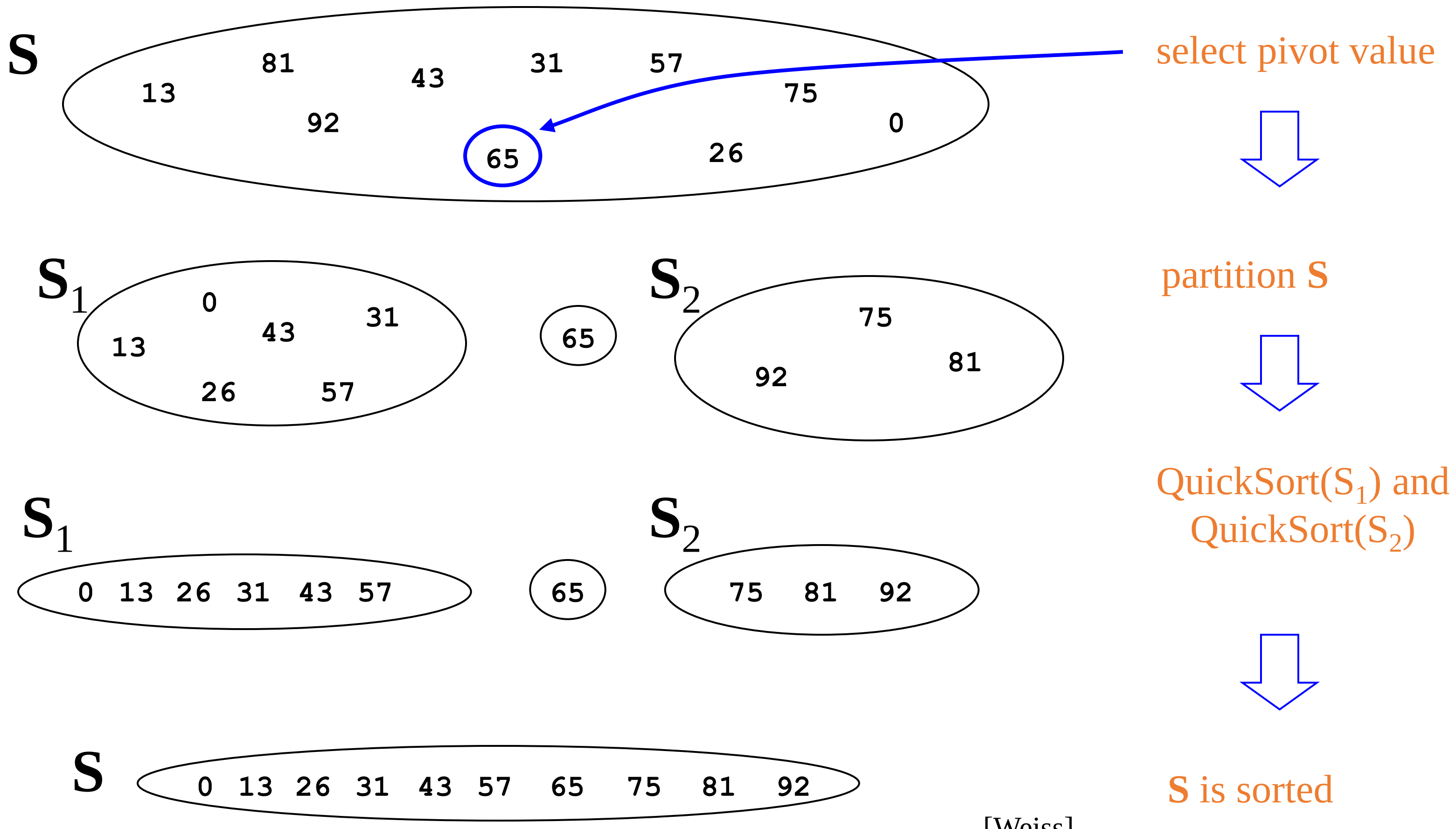
Quicksort

- Quicksort uses a divide and conquer strategy, but does not require the O(N) extra space that MergeSort does
  - Partition array into left and right sub-arrays
    - Choose an element of the array, called pivot
    - the elements in left sub-array are all less than pivot
    - elements in right sub-array are all greater than pivot
  - Recursively sort left and right sub-arrays
  - Concatenate left and right sub-arrays in O(1) time

"Four easy steps"

- To sort an array **S**
  1. If the number of elements in **S** is 0 or 1, then return. The array is sorted.
  2. Pick an element *v* in **S**. This is the *pivot* value.
  3. Partition **S**-{*v*} into two disjoint subsets, $\mathbf{S}_1$ = {all values $x \leq v$}, and $\mathbf{S}_2$ = {all values $x \geq v$}.
  4. Return QuickSort($\mathbf{S}_1$), *v*, QuickSort($\mathbf{S}_2$)

The steps of QuickSort



[Weiss]

Details, details

- Implementing the actual partitioning
- Picking the pivot
  - want a value that will cause $|S_1|$ and $|S_2|$ to be non-zero, and close to equal in size if possible
- Dealing with cases where the element equals the pivot

Quicksort Partitioning

- Need to partition the array into left and right sub-arrays
  - the elements in left sub-array are $\leq$ pivot
  - elements in right sub-array are $\geq$ pivot
- How do the elements get to the correct partition?
  - Choose an element from the array as the pivot
  - Make one pass through the rest of the array and swap as needed to put elements in partitions

Partitioning Algorithm Illustrated

	i								j	pivot	
	$\boxed{8}$	1	4	9	0	3	5	2	$\boxed{7}$	6	
	i							j		pivot	
Move	$\boxed{8}$	1	4	9	0	3	5	$\boxed{2}$	7	6	
	i							j		pivot	
swap	$\boxed{2}$	1	4	9	0	3	5	$\boxed{8}$	7	6	
				i			j			pivot	
move	2	1	4	$\boxed{9}$	0	3	$\boxed{5}$	8	7	6	
				i			j			pivot	
swap	2	1	4	$\boxed{5}$	0	3	$\boxed{9}$	8	7	6	
						j	i			pivot	i and j
move	2	1	4	5	0	$\boxed{3}$	$\boxed{9}$	8	7	6	have crossed
Swap S[i] with pivot	2	1	4	5	0	$\boxed{3}$	$\boxed{6}$	8	7	9	
						j	i pivot				

Partitioning:Choosing the pivot

- One implementation (there are others)
  - median3 finds pivot and sorts left, center, right
    - Median3 takes the median of leftmost, middle, and rightmost elements
    - An alternative is to choose the pivot randomly (need a random number generator; “expensive”)
    - Another alternative is to choose the first element (but can be very bad. Why?)
  - Swap pivot with next to last element

Partitioning in-place

- Set pointers i and j to start and end of array
- Increment i until you hit element A[i] > pivot
- Decrement j until you hit elmt A[j] < pivot
- Swap A[i] and A[j]
- Repeat until i and j cross
- Swap pivot (at A[N-2]) with A[i]

Example

Choose the pivot as the median of three

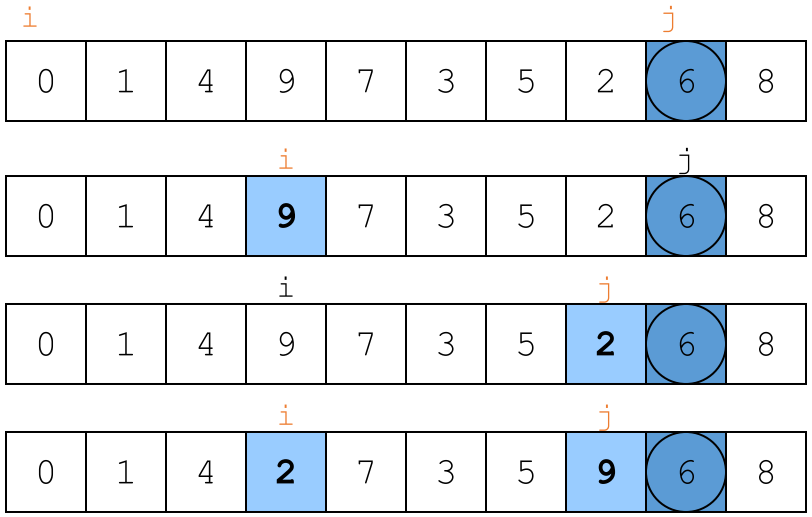
0	1	2	3	4	5	6	7	8	9
8	1	4	9	**0**	3	5	2	7	**6**

Median of 0, 6, 8 is 6. Pivot is 6

0	1	4	9	7	3	5	2	6	8
i								j	

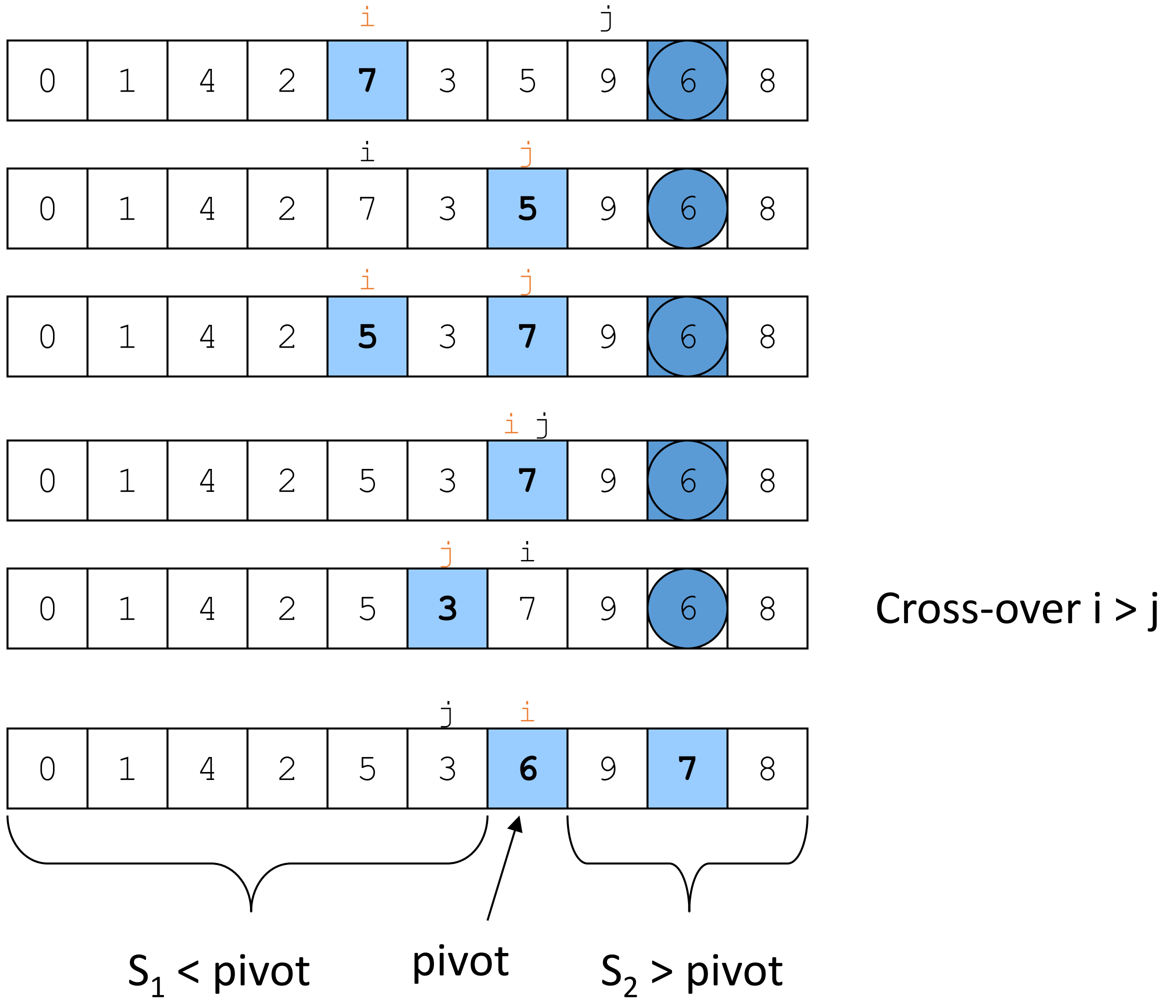
Place the largest at the right
and the smallest at the left.
Swap pivot with next to last element.

Example



Move i to the right up to A[i] larger than pivot.
Move j to the left up to A[j] smaller than pivot.
Swap

Example



i j
0 1 4 2 7 3 5 9 6 8
i j
0 1 4 2 7 3 5 9 6 8
i j
0 1 4 2 5 3 7 9 6 8
i j
0 1 4 2 5 3 7 9 6 8
j i
0 1 4 2 5 3 7 9 6 8
Cross-over i > j
j i
0 1 4 2 5 3 6 9 7 8
S_1 < pivot
pivot
S_2 > pivot

Recursive Quicksort

```
Quicksort(A[]: integer array, left,right : integer): {
pivotindex : integer;
if left + CUTOFF ≤ right then
  pivot := median3(A,left,right);
  pivotindex := Partition(A,left,right-1,pivot);
  Quicksort(A, left, pivotindex – 1);
  Quicksort(A, pivotindex + 1, right);
else
  Insertionsort(A,left,right);
}
```

Don't use quicksort for small arrays.
CUTOFF = 10 is reasonable.

Properties of Quicksort

- Not stable because of long distance swapping.
- No iterative version
- Pure quicksort not good for small arrays.
- "In-place", but uses auxiliary storage because of recursive call (O(logn) space).
- O(n log n) average case performance, but $O(n^2)$ worst case performance.

Time complexity of Sorting

- Several sorting algorithms have been discussed and the best ones, so far:
  - Heap sort and Merge sort: O(n log n)
  - Quick sort (best one in practice): O(n log n) on average, O(n^2) worst case
- Can we do better than O(n log n)?
  - No.
  - It can be proven that any comparison-based sorting algorithm will need to carry out at least O(n log n) operations