

Sorting (Part II: Divide and Conquer)

CSE 373

Data Structures

Lecture 14

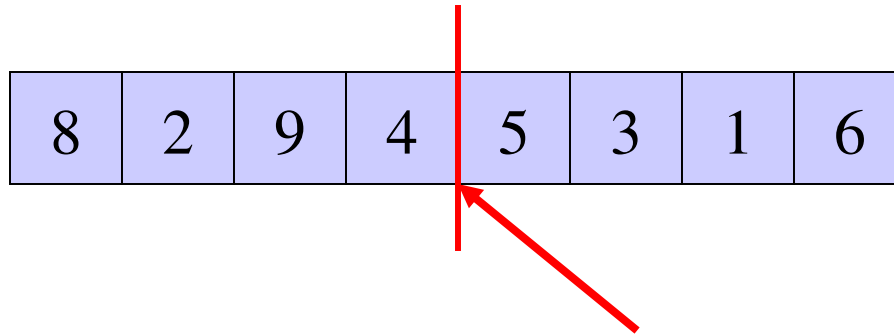
Readings

- Reading
 - › Section 7.6, Mergesort
 - › Section 7.7, Quicksort

“Divide and Conquer”

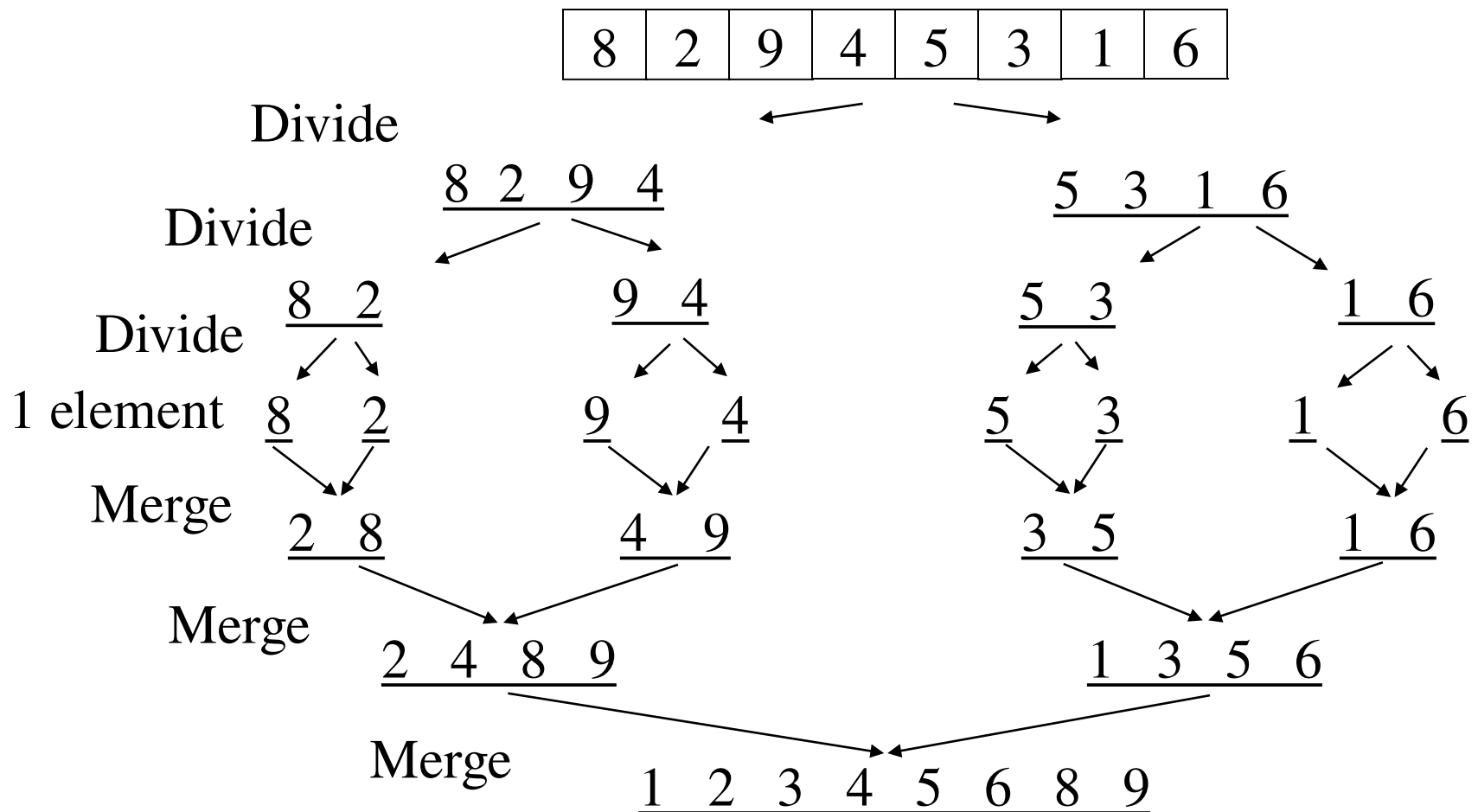
- Very important strategy in computer science:
 - › Divide problem into smaller parts
 - › Independently solve the parts
 - › Combine these solutions to get overall solution
- **Idea 1**: Divide array into two halves, *recursively* sort left and right halves, then *merge* two halves → **Mergesort**
- **Idea 2** : Partition array into items that are “small” and items that are “large”, then recursively sort the two sets → **Quicksort**

Mergesort



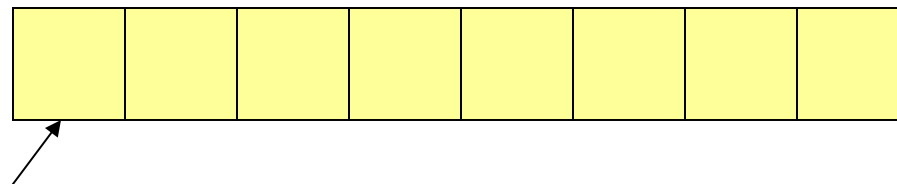
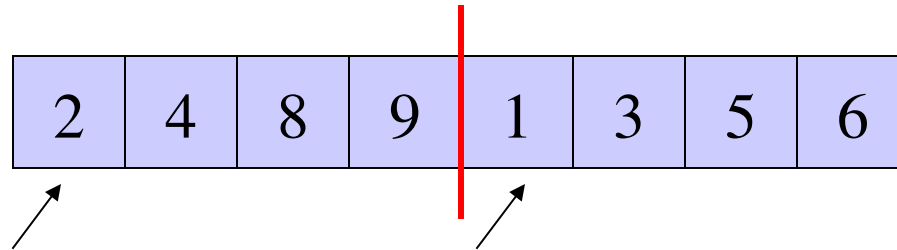
- Divide it in two at the midpoint
- Conquer each side in turn (by recursively sorting)
- Merge two halves together

Mergesort Example



Auxiliary Array

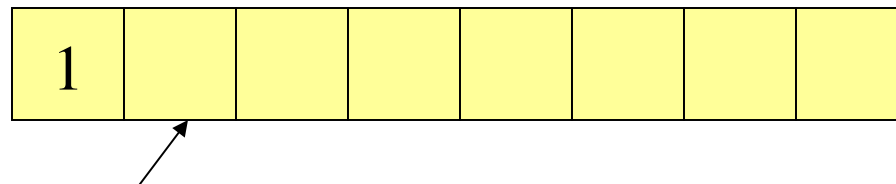
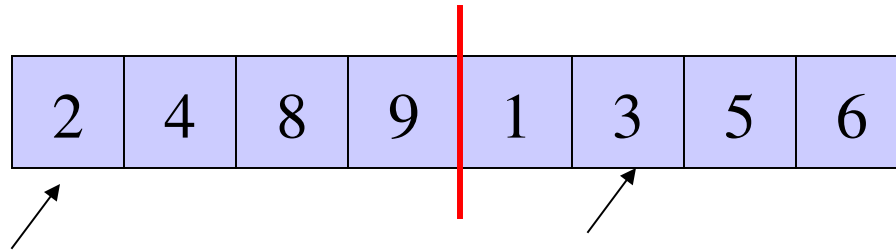
- The merging requires an auxiliary array.



Auxiliary array

Auxiliary Array

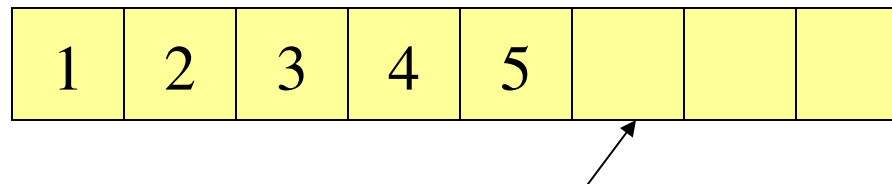
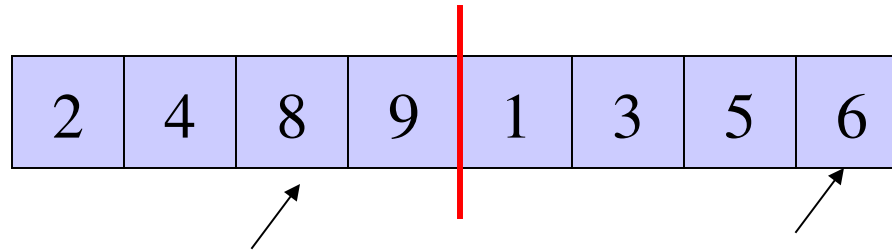
- The merging requires an auxiliary array.



Auxiliary array

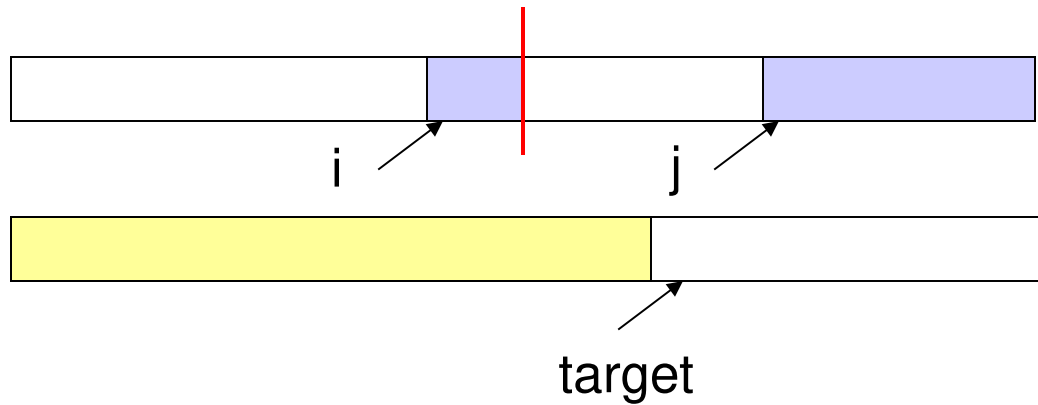
Auxiliary Array

- The merging requires an auxiliary array.

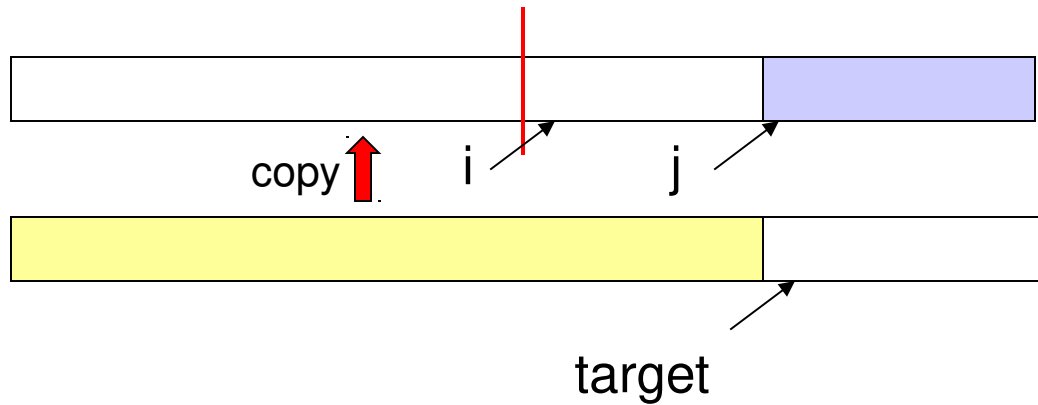


Auxiliary array

Merging

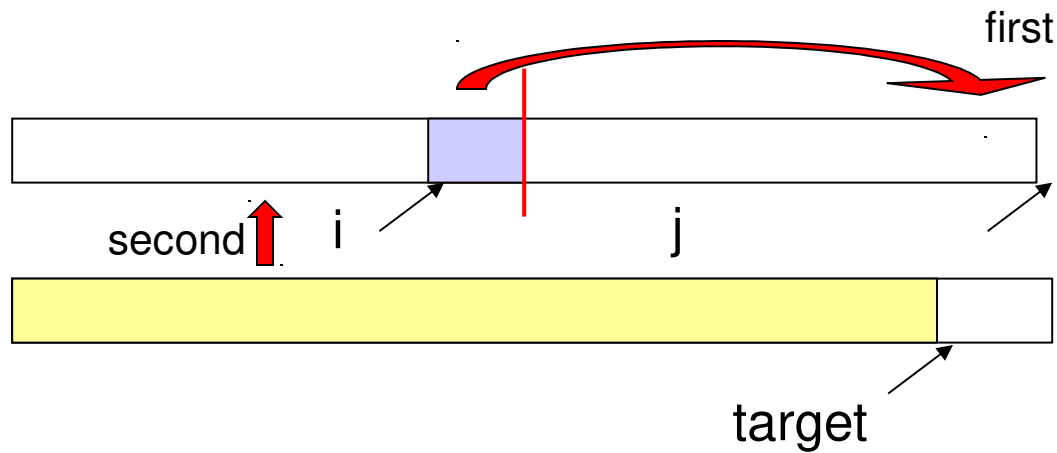


normal



Left completed first

Merging



Right completed
first

Merging

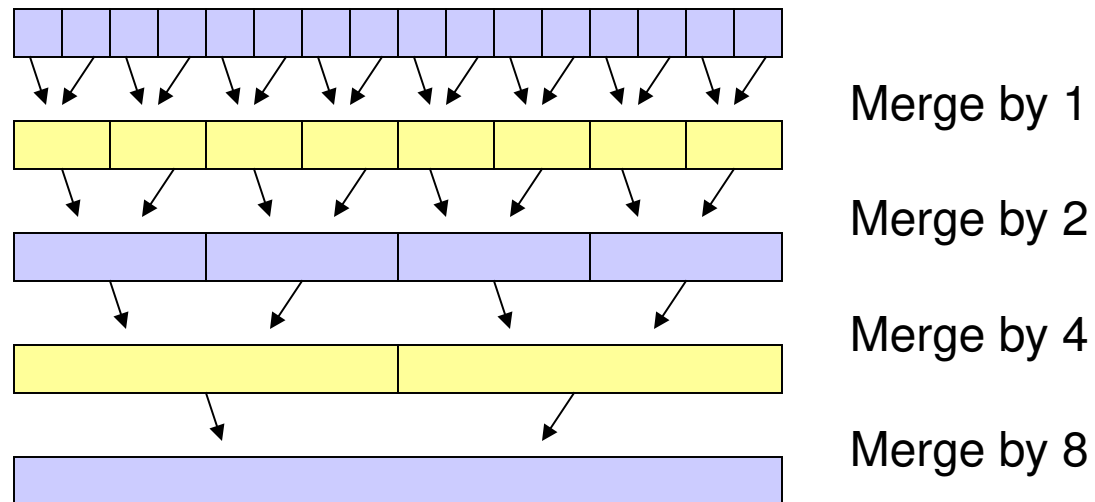
```
Merge(A[], T[] : integer array, left, right : integer) : {  
  mid, i, j, k, l, target : integer;  
  mid := (right + left)/2;  
  i := left; j := mid + 1; target := left;  
  while i ≤ mid and j ≤ right do  
    if A[i] ≤ A[j] then T[target] := A[i] ; i:= i + 1;  
    else T[target] := A[j]; j := j + 1;  
    target := target + 1;  
  if i > mid then //left completed//  
    for k := left to target-1 do A[k] := T[k];  
  if j > right then //right completed//  
    k := mid; l := right;  
    while k ≥ i do A[l] := A[k]; k := k-1; l := l-1;  
    for k := left to target-1 do A[k] := T[k];  
}
```

Recursive Mergesort

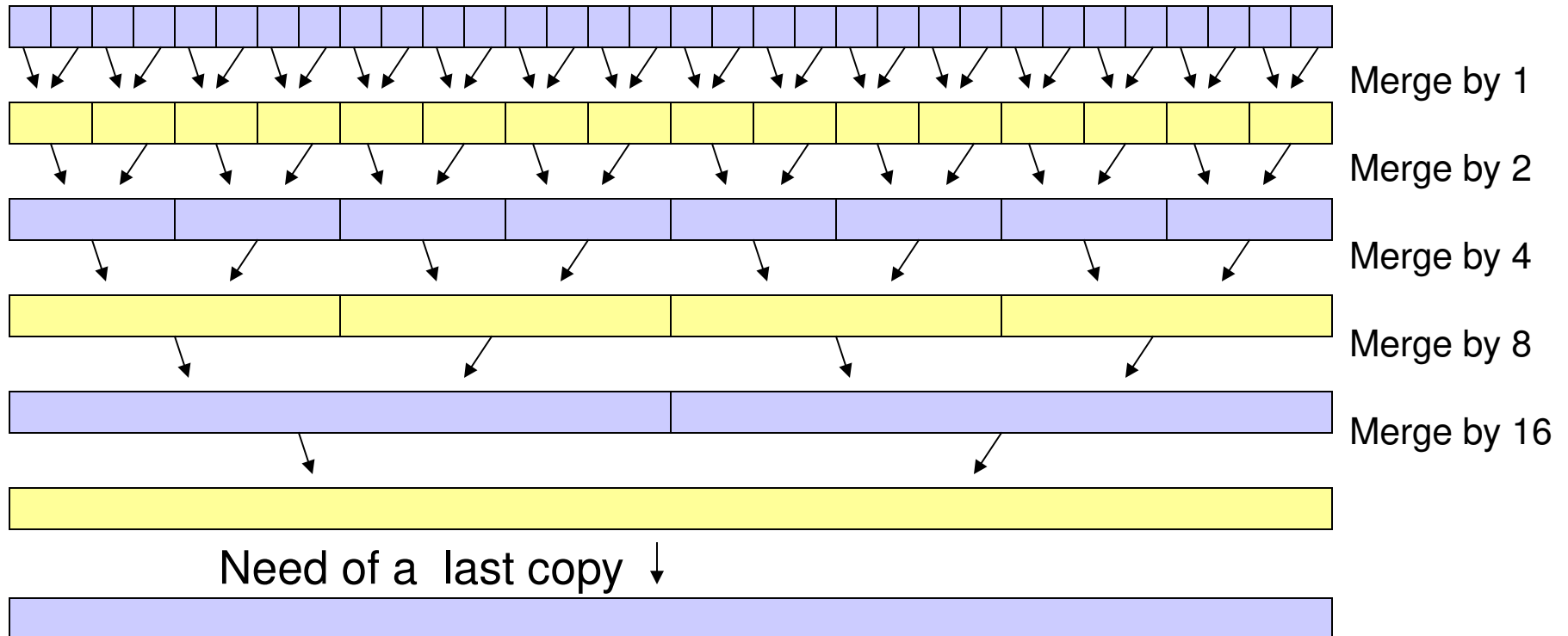
```
Mergesort(A[], T[] : integer array, left, right : integer) : {  
  if left < right then  
    mid := (left + right)/2;  
    Mergesort(A,T,left,mid);  
    Mergesort(A,T,mid+1,right);  
    Merge(A,T,left,right);  
}
```

```
MainMergesort(A[1..n]: integer array, n : integer) : {  
  T[1..n]: integer array;  
  Mergesort[A,T,1,n];  
}
```

Iterative Mergesort



Iterative Mergesort



Iterative Mergesort

```
IterativeMergesort(A[1..n]: integer array, n : integer) : {  
  //precondition: n is a power of 2//  
  i, m, parity : integer;  
  T[1..n]: integer array;  
  m := 2; parity := 0;  
  while m ≤ n do  
    for i = 1 to n - m + 1 by m do  
      if parity = 0 then Merge(A,T,i,i+m-1);  
      else Merge(T,A,i,i+m-1);  
    parity := 1 - parity;  
    m := 2*m;  
  if parity = 1 then  
    for i = 1 to n do A[i] := T[i];  
}
```

How do you handle non-powers of 2?

How can the final copy be avoided?

Mergesort Analysis

- Let $T(N)$ be the running time for an array of N elements
- Mergesort divides array in half and calls itself on the two halves. After returning, it merges both halves using a temporary array
- Each recursive call takes $T(N/2)$ and merging takes $O(N)$

Mergesort Recurrence Relation

- The recurrence relation for $T(N)$ is:
 - › $T(1) \leq a$
 - base case: 1 element array \rightarrow constant time
 - › $T(N) \leq 2T(N/2) + bN$
 - Sorting N elements takes
 - the time to sort the left half
 - plus the time to sort the right half
 - plus an $O(N)$ time to merge the two halves
- $T(N) = O(n \log n)$ (see Lecture 5 Slide 17)

Properties of Mergesort

- Not in-place
 - › Requires an auxiliary array ($O(n)$ extra space)
- Stable
 - › Make sure that **left** is sent to target on equal values.
- Iterative Mergesort reduces copying.

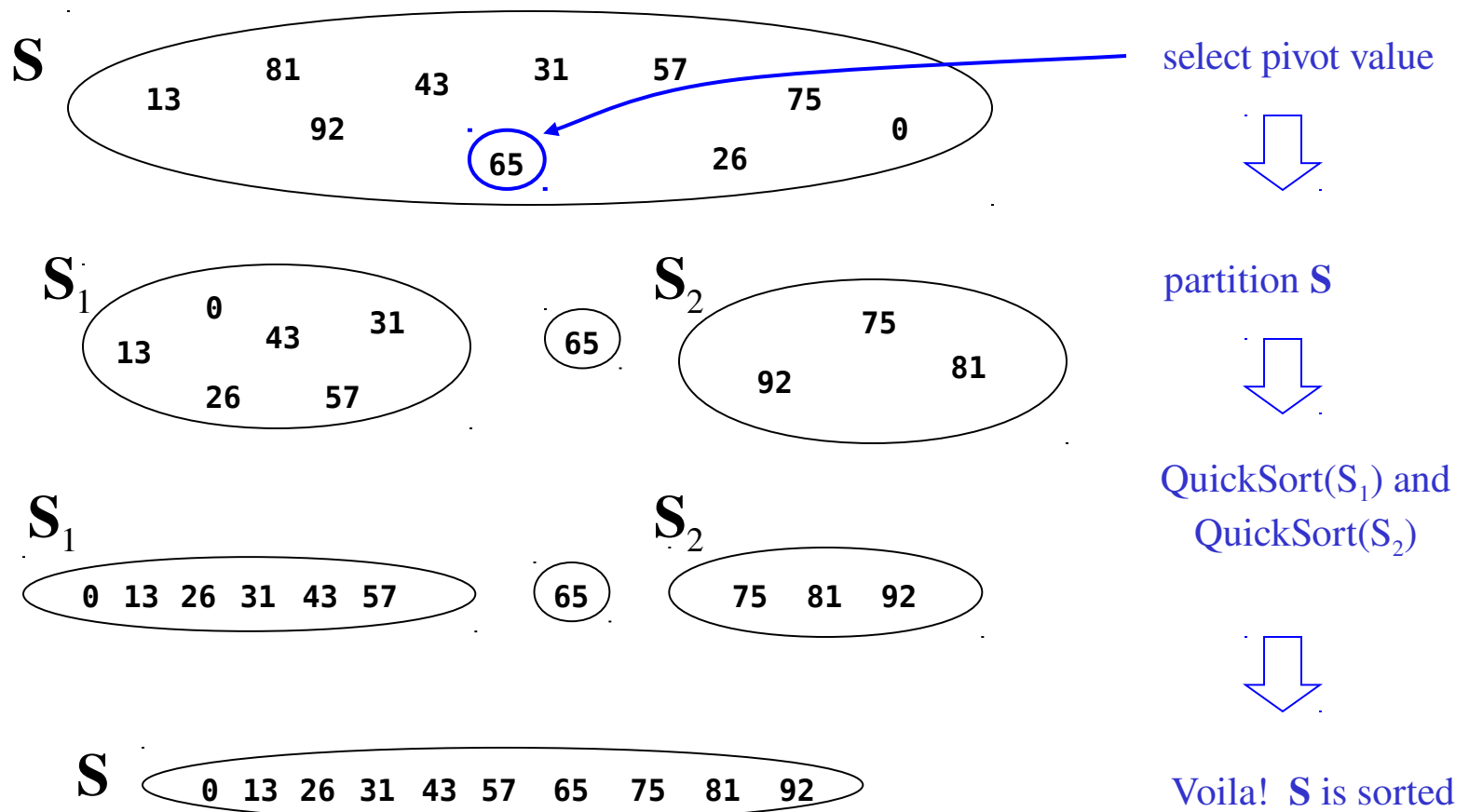
Quicksort

- 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, **S**₁ = {all values $x \leq v$ }, and **S**₂ = {all values $x \geq v$ }.
 4. Return QuickSort(**S**₁), *v*, QuickSort(**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: 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 $A[i] > \text{pivot}$
- › Decrement j until you hit elmt $A[j] < \text{pivot}$
- › 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
---	---	---	---	---	---	---	---	---	---

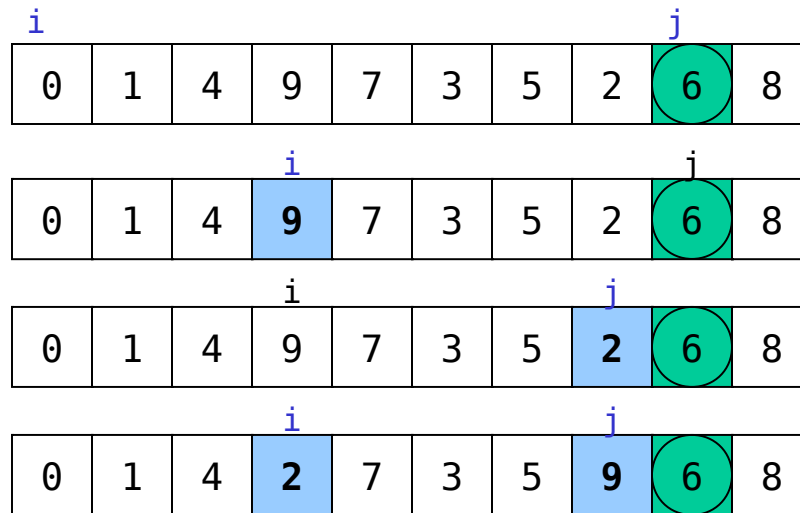
i

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

$S_1 < \text{pivot}$

pivot

$S_2 > \text{pivot}$

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.

Quicksort Best Case Performance

- Algorithm always chooses best pivot and splits sub-arrays in half at each recursion
 - › $T(0) = T(1) = O(1)$
 - constant time if 0 or 1 element
 - › For $N > 1$, 2 recursive calls plus linear time for partitioning
 - › $T(N) = 2T(N/2) + O(N)$
 - Same recurrence relation as Mergesort
 - › $T(N) = \underline{O(N \log N)}$

Quicksort Worst Case Performance

- Algorithm always chooses the worst pivot – one sub-array is empty at each recursion
 - › $T(N) \leq a$ for $N \leq C$
 - › $T(N) \leq T(N-1) + bN$
 - › $\leq T(N-2) + b(N-1) + bN$
 - › $\leq T(C) + b(C+1) + \dots + bN$
 - › $\leq a + b(C + (C+1) + (C+2) + \dots + N)$
 - › $T(N) = O(N^2)$
- Fortunately, *average case performance* is $O(N \log N)$ (see text for proof)

Properties of Quicksort

- Not stable because of long distance swapping.
- No iterative version (without using a stack).
- Pure quicksort not good for small arrays.
- “In-place”, but uses auxiliary storage because of recursive call ($O(\log n)$ space).
- $O(n \log n)$ average case performance, but $O(n^2)$ worst case performance.

Folklore

- “Quicksort is the best in-memory sorting algorithm.”
- Truth
 - › Quicksort uses very few comparisons on average.
 - › Quicksort does have good performance in the memory hierarchy.
 - Small footprint
 - Good locality