CSE 332: Data Structures and Parallelism

Spring 2022

Richard Anderson

Lecture 14: Sorting II

Announcements

- Midterm, Friday, November 4
  - In class
  - No notes, no calculators
  - Coverage: up to, and including Sorting
  - Review session, Tuesday, Nov 1, CSE2 G01
- Lecture on Wed, Nov 2 will end at 1:10 pm

Sorting: *The Big Picture*

Simple algorithms: $O(n^2)$

- Insertion sort
- Selection sort
- …

Fancier algorithms: $O(n \log n)$

- Heap sort
- Merge sort
- Quick sort (avg)
- …

Comparison lower bound: $\Omega(n \log n)$

Specialized algorithms: $O(n)$

- Bucket sort
- Radix sort

"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 in half, *recursively* sort left and right halves, then *merge* two halves
  → known as Mergesort

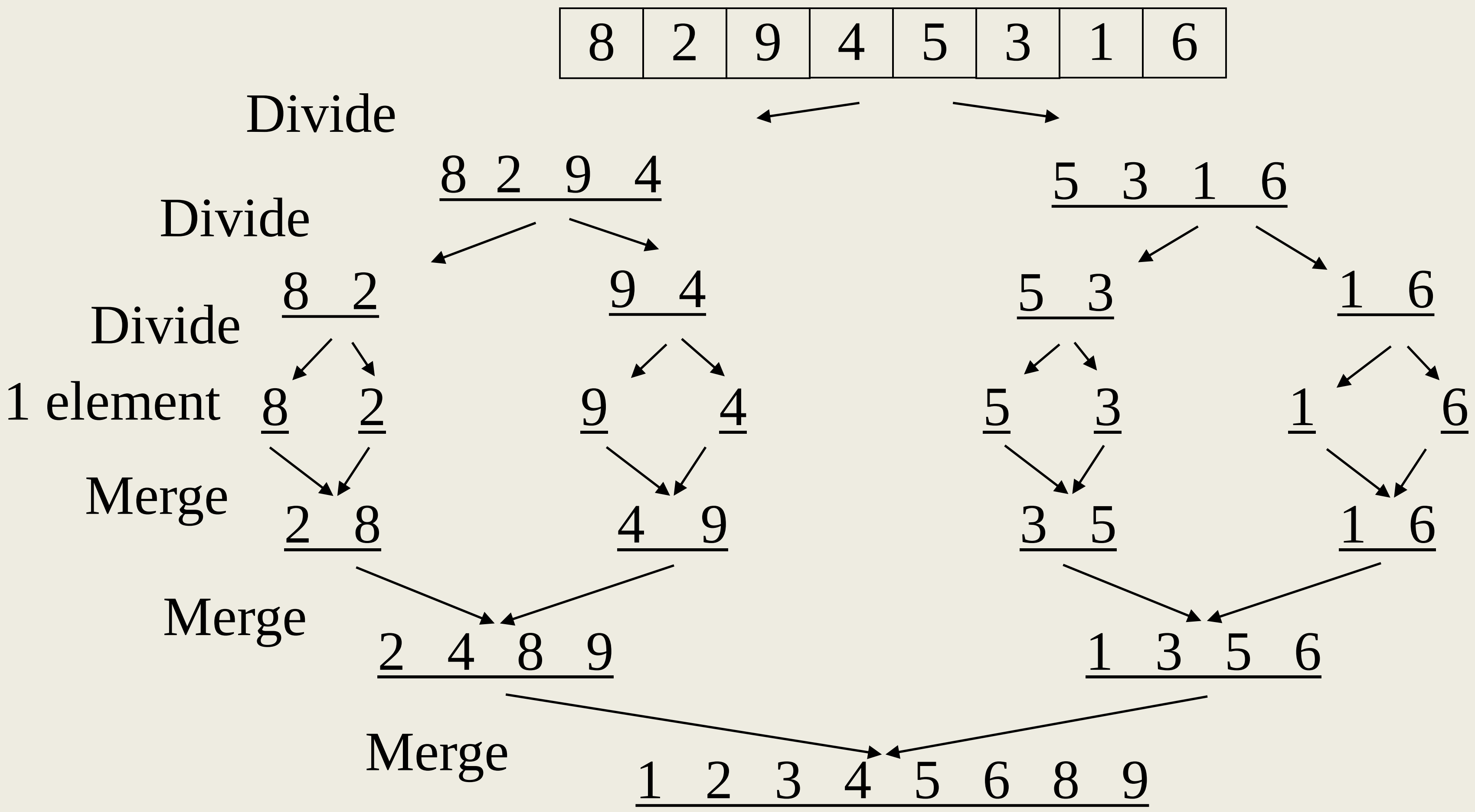
- **Idea 2 :** Partition array into small items and large items, then recursively sort the two sets
  → known as Quicksort

Mergesort

8	2	9	4	5	3	1	6

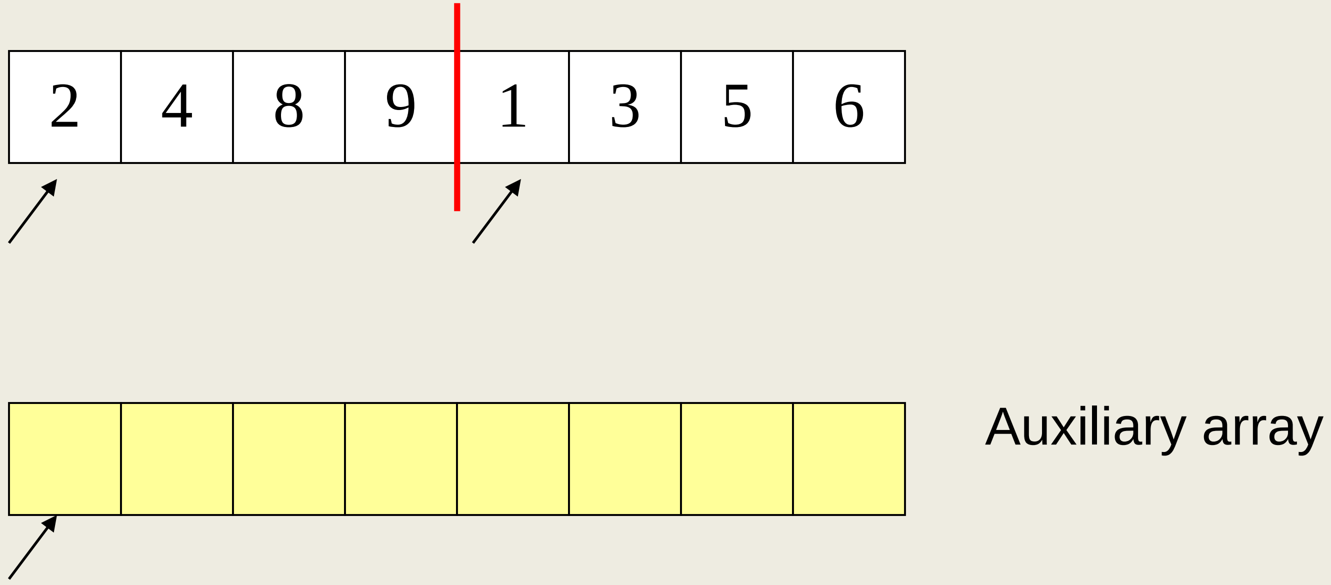
- Divide it in two at the midpoint
- Sort each half (recursively)
- Merge two halves together

Mergesort Example



Merging: Two Pointer Method

Merge using an auxiliary array



Merging

```
Merge(A[], Temp[], left, mid, right)  {
  int i, j, k, l, target
  i = left
  j = mid + 1
  target = left
  while (i ≤ mid && j ≤ right) {
    if (A[i] ≤ A[j])
      Temp[target] = A[i++]
    else
      Temp[target] = A[j++]
    target++
  }
  if (i > mid) //left completed
    for (k = left to target-1)
      A[k] = Temp[k];
  if (j > right) //right completed
    k = mid
    l = right
    while (k ≥ i)
      A[l--] = A[k--]
    for (k = left to target-1)
      A[k] = Temp[k]
}
```

Recursive Mergesort

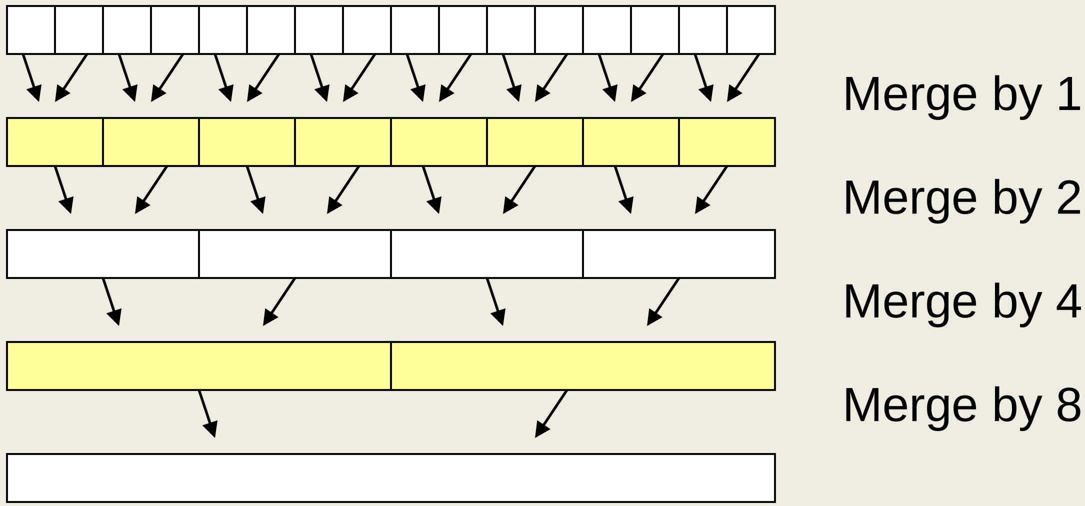
```
MainMergesort(A[1..n], n) {
  Array Temp[1..n]
  Mergesort[A, Temp, 1, n]
}

Mergesort(A[], Temp[], left, right) {
  if (left < right) {
    mid = (left + right)/2
    Mergesort(A, Temp, left, mid)
    Mergesort(A, Temp, mid+1, right)
    Merge(A, Temp, left, mid, right)
  }
}
```

What is the recurrence relation?

Mergesort: Complexity

Iterative Mergesort



Properties of Mergesort

- In-place?
- Sorted list complexity?
- Nicely extends to handle linked lists.
- Multi-way merge is basis of big data sorting.
- Java uses Mergesort on Collections and on Arrays of Objects.

Recurrences

General form:

$T(N) = S(N) + \sum_i a_i T(f_i(N))$; $T(1) = c$;

Important recurrences

$T(N) = T(N-1) + f(N)$

$T(N) = T(aN) + cN$, $a < 1$

$T(N) = aT(N/b) + Nc$

$$T(N) = T(N-1) + N^2; \quad T(1) = 0$$

$$T(N) = T(N/2) + N; \quad T(1) = 1$$

$$T(N) = 4\,T(N/4) + N;\quad T(1) = 1$$

Quicksort

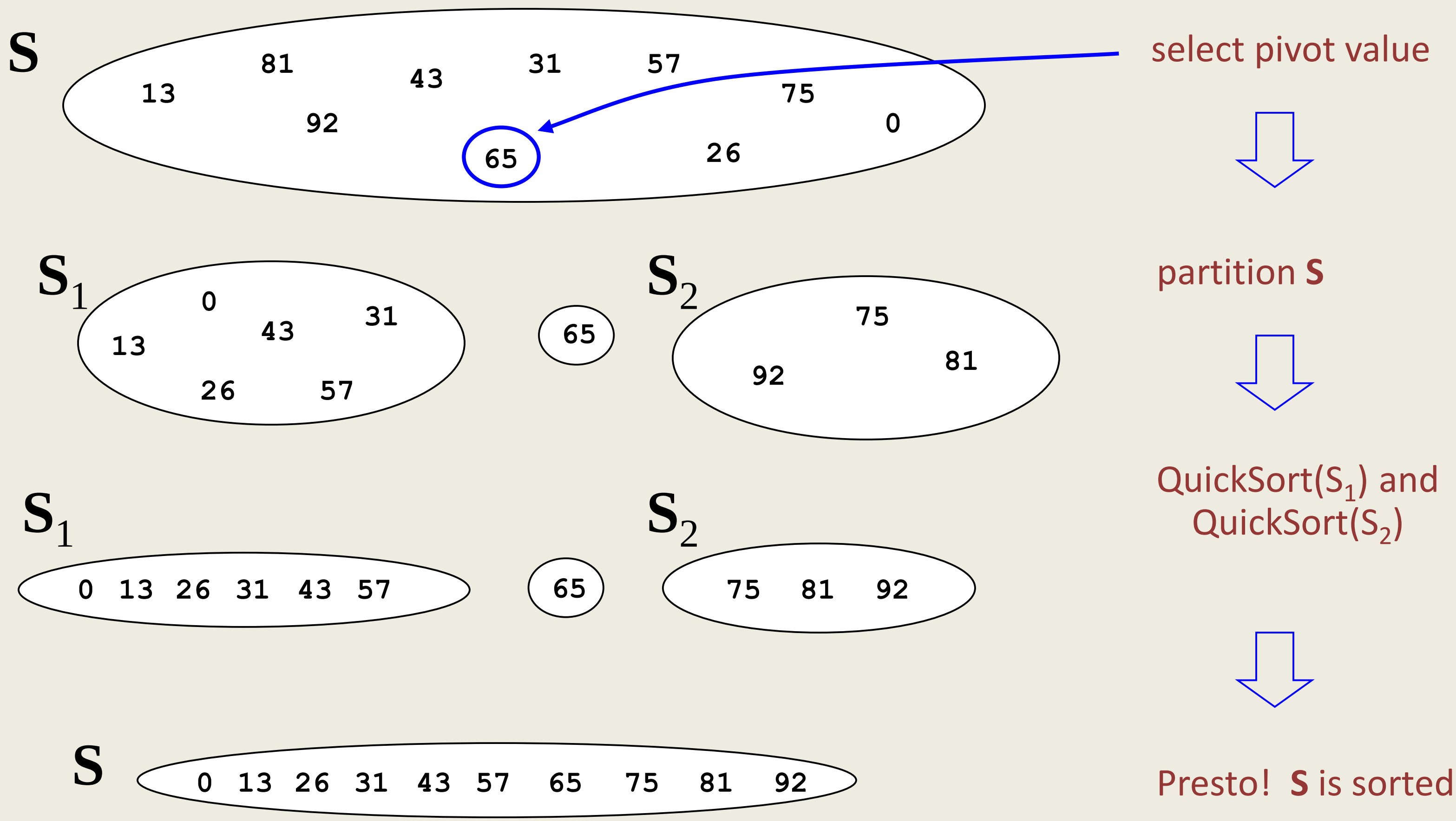
Quicksort uses a divide and conquer strategy, but does not require the O(N) extra space that MergeSort does.

Here's the idea for sorting array **S**:

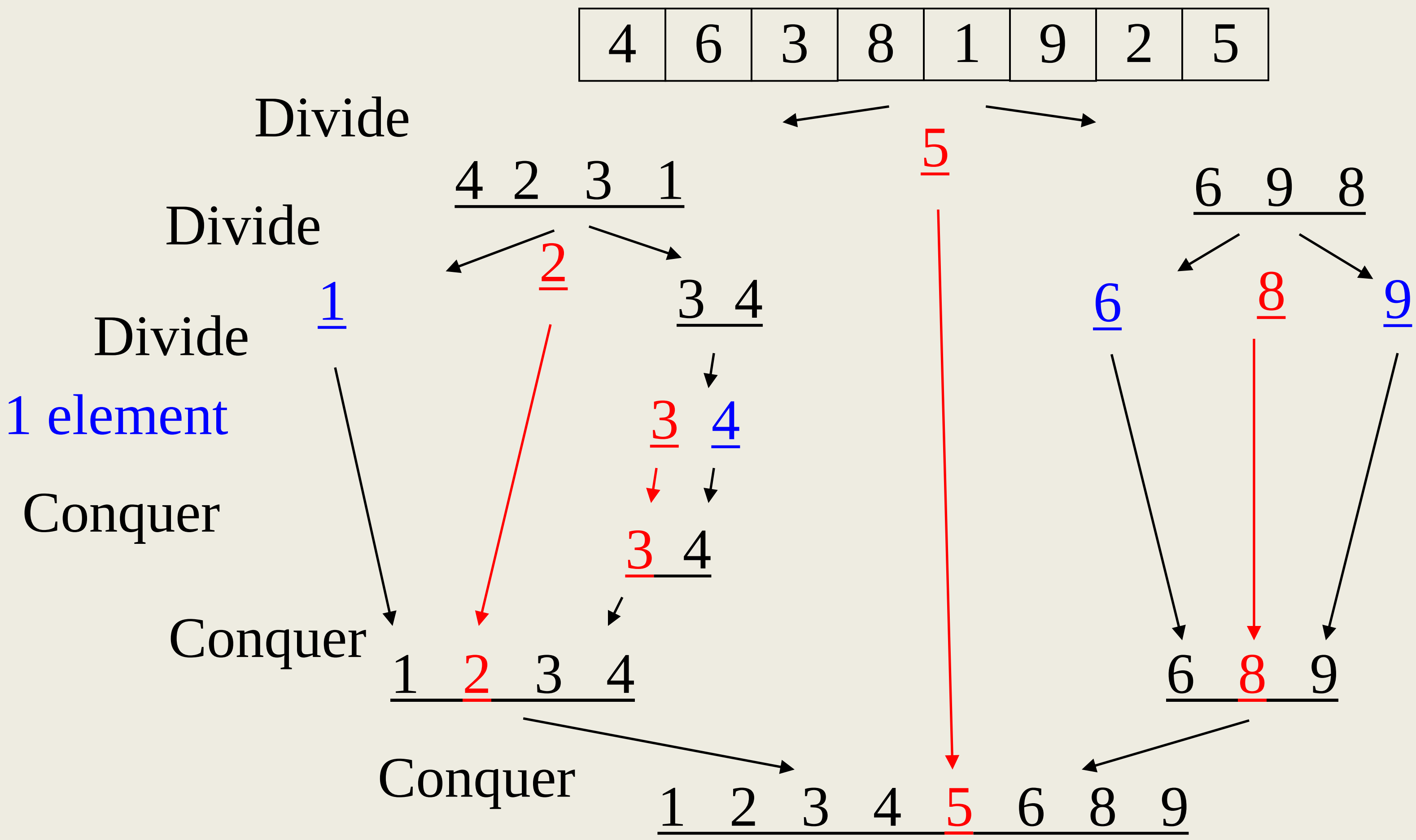
1. Pick an element *v* in **S**. This is the ***pivot*** value.
2. Partition **S**-{*v*} into two disjoint subsets, $\mathbf{S}_1$ and $\mathbf{S}_2$ such that:
   - elements in $\mathbf{S}_1$ are all $\leq v$
   - elements in $\mathbf{S}_2$ are all $\geq v$
3. Return concatenation of QuickSort($\mathbf{S}_1$), *v*, QuickSort($\mathbf{S}_2$)

Recursion ends if Quicksort( ) receives an array of length 0 or 1.

The steps of Quicksort



Quicksort Example



Pivot Picking and Partitioning

The tricky parts are:

- **Picking the pivot**
  - Goal: pick a pivot value so that $|S_1|$ and $|S_2|$ are roughly equal in size.
- **Partitioning**
  - Preferably in-place
  - Dealing with duplicates

Picking the pivot

- Choose the first element in the subarray
- Choose a value that might be close to the middle
  - Median of three
- Choose a random element

Quicksort Partitioning

- Partition the array into left and right sub-arrays such that:
  - elements in left sub-array are $\leq$ pivot
  - elements in right sub-array are $\geq$ pivot
- Can be done in-place with another “two pointer method”
  - Sounds like mergesort, but here we are *partitioning*, not sorting…
  - …and we can do it in-place.
- Lots of work has been invested in engineering quicksort

Quicksort Pseudocode

Putting the pieces together:

```
Quicksort(A[], left, right) {
  if (left < right) {
    medianOf3Pivot(A, left, right);
    pivotIndex = Partition(A, left+1, right-1);

    Quicksort(A, left, pivotIndex – 1);
    Quicksort(A, pivotIndex + 1, right);
  }
}
```

Important Tweak

Insertion sort is actually better than quicksort on small arrays. Thus, a better version of quicksort:

```
Quicksort(A[], left, right) {
  if (right - left ≥ CUTOFF) {
    medianOf3Pivot(A, left, right);
    pivotIndex = Partition(A, left+1, right-1);

    Quicksort(A, left, pivotIndex - 1);
    Quicksort(A, pivotIndex + 1, right);

  } else {
    InsertionSort(A, left, right);
  }
}
```

CUTOFF = 16 is reasonable.

Quicksort run time

- What is the best case behavior?

Worst case run time

- What is the bad case for partitioning?
- Design a bad case input (assume first element is chosen as pivot)

Average case performance

- Assume all permutations of the data are equally likely
  - Or equivalently, a random pivot is chosen

- The math gets messy, but doable

Properties of Quicksort

- O(N^2) worst case performance, but
  O(N log N) average case performance.
- Pure quicksort not good for small arrays.
- No iterative version (without using a stack).
- "In-place," but uses auxiliary storage because of recursive calls.
- Used by Java for sorting arrays of primitive types.