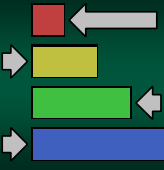


Recursive Sorting

Part 6

1



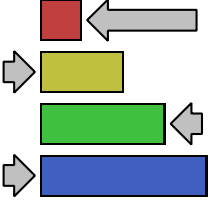
Merging Arrays

Quite easy... and quite common

2

Merging Arrays

- It is a common task in Computer Science to combine two different arrays into one
- If both arrays are unsorted...
 - the task is fairly simple $O(n)$
 - just add one onto the end of the other

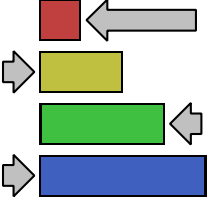


Fall 2024 Sacramento State - CSIS - CSIS 130 3

3

Merging Arrays

- However, often two sorted arrays are combined
- ...and the resulting array must be sorted



Fall 2024 Sacramento State - CSIS - CSIS 130 4

4

Merging Arrays

- The algorithm for merging two sorted arrays is very simple
- The resulting time complexity is $O(n)$
- However, it requires auxiliary storage of $O(n)$

Fall 2024 Sacramento State - CSIS - CSIS 130 5

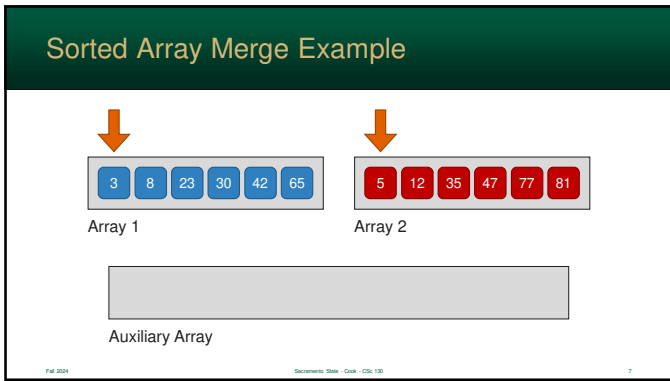
5

Merge Algorithm

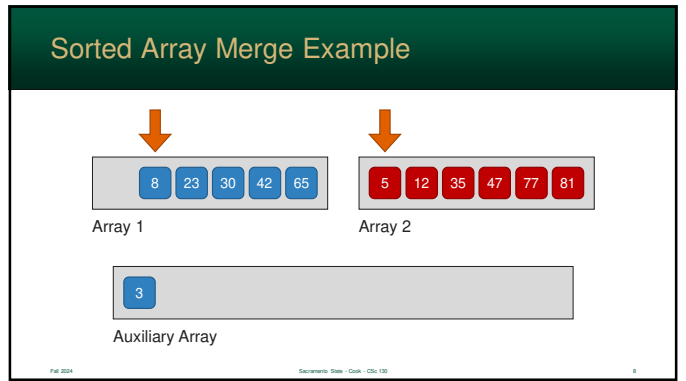
- Keep two counters – one for each array
- Loop while both arrays have data
 - take the smaller element and put it in the auxiliary array
 - increment the array's counter (which just lost an element)
- After the loop
 - one array will still have elements
 - append them to the auxiliary array

Fall 2024 Sacramento State - CSIS - CSIS 130 6

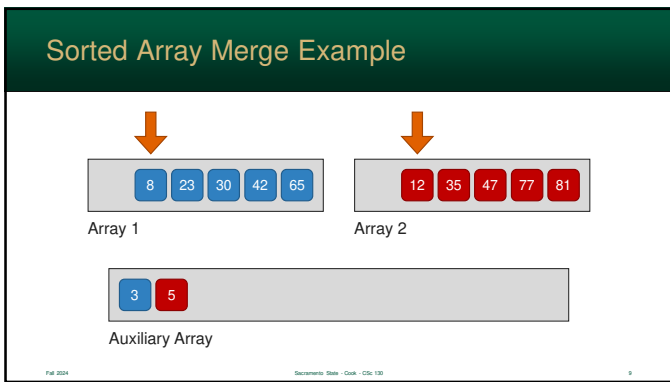
6



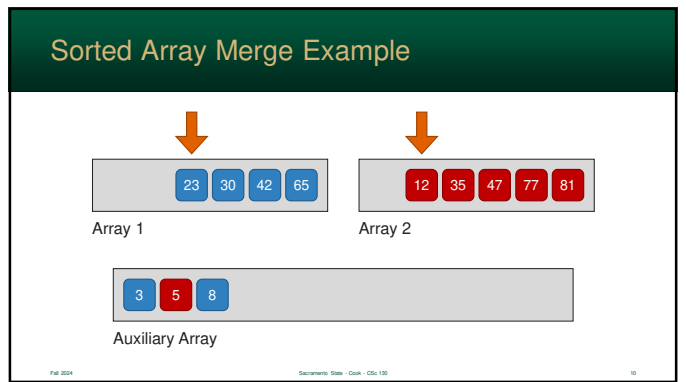
7



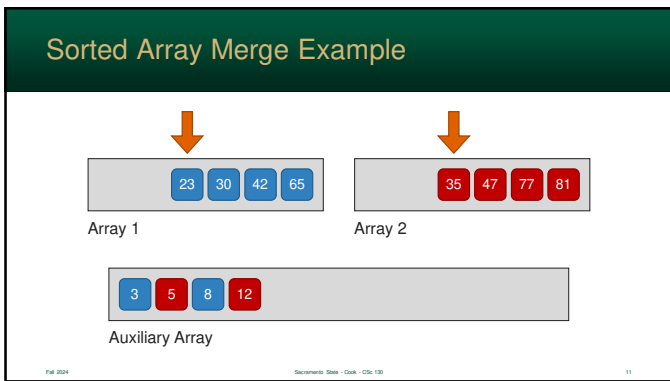
8



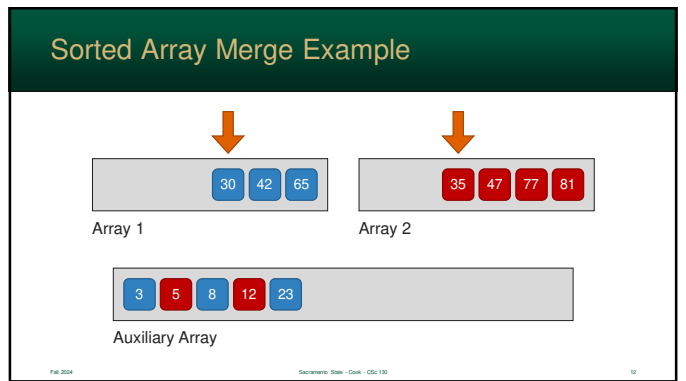
9



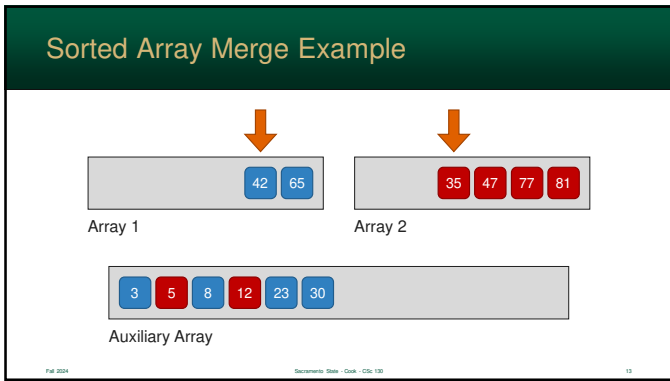
10



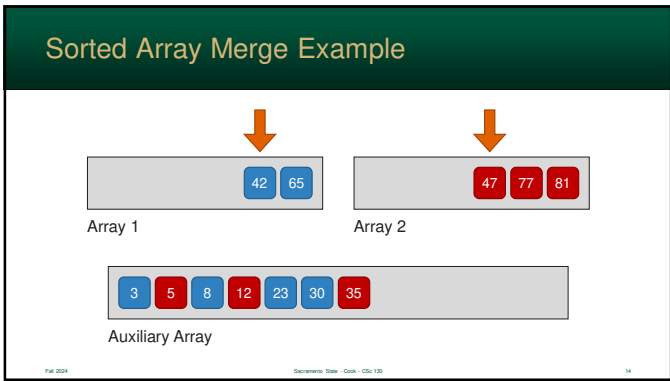
11



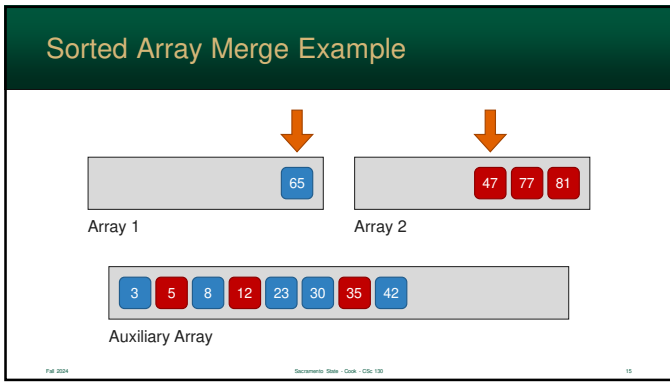
12



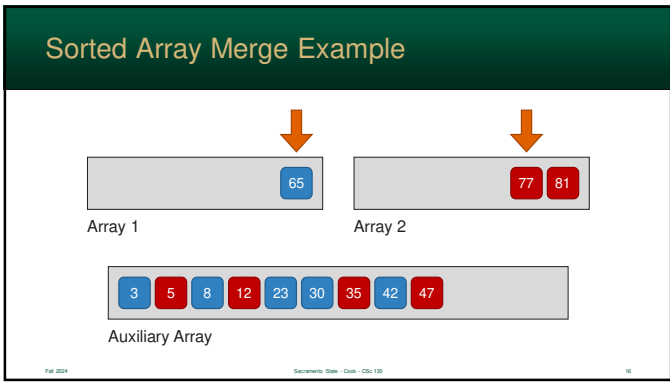
13



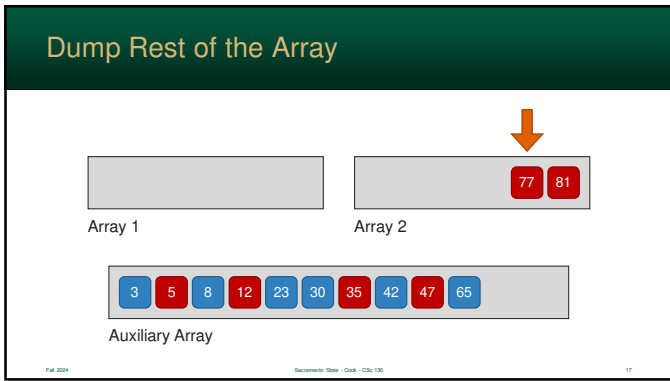
14



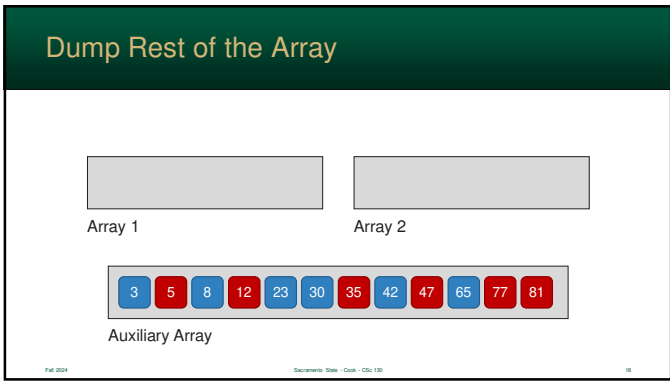
15



16



17



18

Merge Sort

Divide and conquer!

19

Merge Sort

- *Merge Sort* is a divide-and-conquer algorithm that cuts an array into smaller and smaller sublists until sorting them is arbitrary
- Invented by *John von Neumann* in 1945 (19 BBW)

Merge Sort

20

Merge Sort

- Because Merge-Sort defines a dividing the list into a list into smaller instances of itself, it naturally is solved using recursion
- Each recursive step cuts the list into two sublists until...
 - the list has 2 elements – arbitrary swap
 - the list has 1 element – which is, well, sorted

21

Merge Sort

- As the recursion bubbles up, each sub list is **merged** using the algorithm we just discussed
- Divide-and-conquer algorithms ultimately result in $O(n \log n)$
- Since an auxiliary array is required for the merge process, Merge-Sort, while fast, has $O(n)$ auxiliary storage requirements

22

Merge Sort Example: Recurse down

62 44 4 80 13 53 35 16

62 44 4 80 13 53 35 16

62 44 4 80 13 53 35 16

23

Sort Merge Sort Example: Merge Up

44 62 4 80 13 53 16 35

4 44 62 80 13 16 35 53

4 13 16 35 44 53 62 80

24

Merge Sort Summary

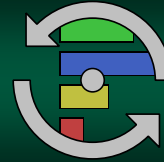
Merge Sort	
Time Average	$O(n \log n)$
Time Best	$O(n \log n)$
Time Worst	$O(n \log n)$
Auxiliary space	$O(n)$
Stable	Yes – Equal element order preserved
Online?	Yes – New data → new sublist

Fall 2024

Sacramento State - Comp - CS110

25

25



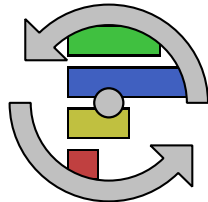
Quick Sort

Oh, I am getting dizzy....

26

Quick Sort

- *Quick-Sort* is a divide-and-conquer algorithm that rotates values around a *pivot*
- Invented by *C. A. R. Hoare* in 1959 (*5 BBW*)
- Even faster than both Merge Sort and Heap Sort
- ... but has a weaknesses



Fall 2024

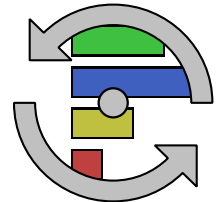
Sacramento State - Comp - CS110

27

27

How it Works

- Like Merge-Sort, the array is broken down into smaller and smaller sub-lists
- However, before recursion
 - a value p is chosen in the sub-list as the *pivot* value
 - smaller items are moved before it
 - larger items are moved after it



Fall 2024

Sacramento State - Comp - CS110

28

28

Choosing a Pivot

- Pivot can be any element in the sub-array
- ...we need one actual value to compare
- This *pivot* is used to *partition* the values
- Different versions use different pivots
 - first item in the sub-array
 - end item in the sub-array
 - the midpoint of the sub-array
 - random value in the sub-array

Fall 2024

Sacramento State - Comp - CS110

29

29

Partitioning the Values

- After the pivot p is selected, all elements are moved
- Two, separate, loops move through the elements and swaps elements less than/greater than the pivot
- The result is...
 - sub-array **L** contains items less than p
 - sub-array **G** contains items greater than p

Fall 2024

Sacramento State - Comp - CS110

30

30

Partitioning (pivot is the first item)

Fall 2024 Sacramento State - CS&E - CS110 31

31

Partitioning the Values

- **Note:** neither **L** or **G** is sorted yet
- These will be called **recursively** by Quick-Sort
- Moving the elements, in-place, can look a tad ugly code-wise, but the logic is straight forward

Fall 2024 Sacramento State - CS&E - CS110 32

32

Partition Algorithm

- The sub-lists are stored in the **original** array – so there's **no** auxiliary storage
- The algorithm maintains two pointers
 - first moves left to right and keeps track of the values that are **too big**
 - second moves right to left and keeps track of the values that are **too small**
- Each moves independently

Fall 2024 Sacramento State - CS&E - CS110 33

33

Partition Algorithm

- First move the **Too Big** pointer until a value is found that is **bigger** than the pivot
- Then move the **Too Small** pointer until a value is found that is **smaller** than Pivot
- Then, these values are swapped
- When the two pointers collide, we are done

Fall 2024 Sacramento State - CS&E - CS110 34

34

Example Partition

- *In this example,* we pivot at the **start** of the array
- **Any** value can be used...
 - but it will have to be swapped to the start before the algorithm runs
 - this "saves" the pivot for later

Fall 2024 Sacramento State - CS&E - CS110 35

35

Quick Sort Algorithm

```

while (tooBig < tooSmall)
{
    while (array[tooBig] <= array[pivot])
    {
        tooBig++;
    }

    while (array[tooSmall] > array[pivot])
    {
        tooSmall--;
    }

    if (tooBig < tooSmall)
    {
        //swap array[tooBig] and array[tooSmall]
    }
}

//swap array[tooSmall] and array[pivot]
//Recursive QuickSort on both L and G
    
```

Fall 2024 Sacramento State - CS&E - CS110 36

36

Example: Pivot is First

Slide 37 shows an array of 11 numbers: 42, 8, 12, 77, 65, 30, 47, 52, 9, 35, 23. The first element, 42, is highlighted in red and labeled 'Pivot'. A green arrow labeled 'Too Big' points to the element 77. A blue arrow labeled 'Too Small' points to the element 23.

37

Example: Move Too Big

Slide 38 shows the same array as slide 37. A green arrow labeled 'Too Big' points to the element 77. A blue arrow labeled 'Too Small' points to the element 23.

38

Example: Move Too Big

Slide 39 shows the same array as slide 38. A green arrow labeled 'Too Big' points to the element 77. A blue arrow labeled 'Too Small' points to the element 23.

39

Example: Move Too Big

Slide 40 shows the same array as slide 39. A green arrow labeled 'Found!' points to the element 77. A blue arrow labeled 'Too Small' points to the element 23.

40

Example: Now, Move Too Small

Slide 41 shows the same array as slide 40. A green arrow labeled 'Too Big' points to the element 77. A blue arrow labeled 'Too Small' points to the element 23.

41

Example: Found (Immediately)

Slide 42 shows the same array as slide 41. A green arrow labeled 'Too Big' points to the element 77. A blue arrow labeled 'Too Small' points to the element 23. A blue callout box labeled 'Found!' points to the element 77.

42

Example: Swap Values

Slide 43 shows an array of 11 numbers: 42, 8, 12, 77, 65, 30, 47, 52, 9, 35, 23. The number 77 is highlighted in green and labeled "Too Big" with a green arrow pointing down. The number 23 is highlighted in blue and labeled "Too Small" with a blue arrow pointing up.

43

Example: Keep going... Move Too Big

Slide 44 shows the same array as slide 43, but the values 77 and 23 have been swapped. The array is now 42, 8, 12, 23, 65, 30, 47, 52, 9, 35, 77. The number 77 is highlighted in green and labeled "Too Big" with a green arrow pointing down. The number 23 is highlighted in blue and labeled "Too Small" with a blue arrow pointing up.

44

Example: Too Big Found

Slide 45 shows the array from slide 44. The number 65 is highlighted in green and labeled "Too Big" with a green arrow pointing down. The number 77 is highlighted in green and labeled "Found!" with a green callout box.

45

Example: Move Too Small

Slide 46 shows the array from slide 44. The number 65 is highlighted in green and labeled "Too Big" with a green arrow pointing down. The number 23 is highlighted in blue and labeled "Found!" with a blue callout box.

46

Example: Too Small Found

Slide 47 shows the array from slide 44. The number 65 is highlighted in green and labeled "Too Big" with a green arrow pointing down. The number 35 is highlighted in blue and labeled "Too Small" with a blue arrow pointing up.

47

Example: Swap Values

Slide 48 shows the array from slide 44. The number 65 is highlighted in green and labeled "Too Big" with a green arrow pointing down. The number 35 is highlighted in blue and labeled "Too Small" with a blue arrow pointing up.

48

Example: Keep going... Move Too Big

Too Big

Too Small

42 8 12 23 35 30 47 52 9 65 77

Fall 2024 Sacramento State - CS&E - CS130 49

49

Example: Keep going... Move Too Big

Too Big

Too Small

42 8 12 23 35 30 47 52 9 65 77

Fall 2024 Sacramento State - CS&E - CS130 50

50

Example: Too Big Found

Found!

Too Big

Too Small

42 8 12 23 35 30 47 52 9 65 77

Fall 2024 Sacramento State - CS&E - CS130 51

51

Example: Move Too Small

Too Big

Too Small

42 8 12 23 35 30 47 52 9 65 77

Fall 2024 Sacramento State - CS&E - CS130 52

52

Example: Too Small Found

Too Big

Too Small

Found!

42 8 12 23 35 30 47 52 9 65 77

Fall 2024 Sacramento State - CS&E - CS130 53

53

Example: Swap Values

Too Big

Too Small

42 8 12 23 35 30 47 52 9 65 77

Fall 2024 Sacramento State - CS&E - CS130 54

54

Example: Keep going... Move Too Big

Diagram illustrating an array [42, 8, 12, 23, 35, 30, 9, 52, 47, 65, 77]. A green arrow points to the element 52, labeled "Too Big". A blue arrow points to the element 9, labeled "Too Small".

55

Example: Too Big Found

Diagram illustrating an array [42, 8, 12, 23, 35, 30, 9, 52, 47, 65, 77]. A green arrow points to the element 52, labeled "Too Big". A blue arrow points to the element 9, labeled "Too Small". A speech bubble labeled "Found!" points to the element 52.

56

Example: Move Too Small

Diagram illustrating an array [42, 8, 12, 23, 35, 30, 9, 52, 47, 65, 77]. A green arrow points to the element 52, labeled "Too Big". A blue arrow points to the element 9, labeled "Too Small".

57

Example: Move Too Small

Diagram illustrating an array [42, 8, 12, 23, 35, 30, 9, 52, 47, 65, 77]. A green arrow points to the element 52, labeled "Too Big". A blue arrow points to the element 9, labeled "Too Small".

58

Example: Pointers Passed Each Other

Diagram illustrating an array [42, 8, 12, 23, 35, 30, 9, 52, 47, 65, 77]. A green arrow points to the element 52, labeled "Too Big". A blue arrow points to the element 9, labeled "Too Small".

59

Example: Swap Pivot & Too Small

Diagram illustrating an array [42, 8, 12, 23, 35, 30, 9, 52, 47, 65, 77]. A green arrow points to the element 52, labeled "Too Big". A blue arrow points to the element 9, labeled "Too Small".

60

Example: Done (with this pass)

9 8 12 23 35 30 42 52 47 65 77

Fall 2024 Sacramento State - CS&E - CS130 61

61

Recursion Time!

- Notice: all the items **before** the pivot are **smaller** and all the items **after** are a **larger**
- Now, we can recurse both sides
- The result is a sorted array

9 8 12 23 35 30 42 52 47 65 77

Fall 2024 Sacramento State - CS&E - CS130 62

62

Quick Sort Example

44 62 4 80 13 53 35 16 77

16 35 13 4 44 77 53 80 62

Fall 2024 Sacramento State - CS&E - CS130 63

63

Quick Sort Example

44 62 4 80 13 53 35 16 77

16 35 13 4 44 77 53 80 62

4 13 16 35 62 53 77 80

Fall 2024 Sacramento State - CS&E - CS130 64

64

Quick Sort Example

4 13 16 35 44 53 62 77 80

Sorted on base case

In the main array from the first partition

Fall 2024 Sacramento State - CS&E - CS130 65

65

Quick Sort: Worst Case

- Assume we get array that is already sorted
- This can cause huge problems!
- Shockingly, the efficiency of this sort can degenerate if we are not careful

Fall 2024 Sacramento State - CS&E - CS130 66

66

Quick Sort: Worst Case

- If the first item is the pivot
 - a sorted array will cause both the pointers will pass simply pass each other
 - one sub-array will be empty, the second will contains **ALL** the elements - 1
- If the last item is the pivot
 - reverse sorted array will have the same effect

Fall 2024 Sacramento State - CS&E - CS130 67

67

Quick Sort: Worst Case

Fall 2024 Sacramento State - CS&E - CS130 68

68

Worst Case: Move Too Big

Fall 2024 Sacramento State - CS&E - CS130 69

69

Worst Case: Now, Move Too Small

Fall 2024 Sacramento State - CS&E - CS130 70

70

Worst Case: Pointers Passed

Fall 2024 Sacramento State - CS&E - CS130 71

71

Worst Case: Recurse on n-1

Fall 2024 Sacramento State - CS&E - CS130 72

72

Quick Sort Analysis

- So, in the worst case, Quick Sort is $O(n^2)$
- ... and, given all the work it has to do with the pointers, it gets beat by Bubble Sort



Fall 2024

Srinivasan Sivasubramanian - CS510

73

73

How Can We Avoid This?

- If you don't know if the array is randomized, *manually randomize the values*
- $O(n)$ – run i from first to last element and swap $array[i]$ and $array[random]$



Fall 2024

Srinivasan Sivasubramanian - CS510

74

74

Quick Sort Summary

Quick Sort	
Time Average	$O(n \log n)$
Time Best	$O(n \log n)$
Time Worst	$O(n^2)$
Auxiliary space	$O(1)$
Stable	No – Equal element order not preserved
Online?	No

Fall 2024

Srinivasan Sivasubramanian - CS510

75

75