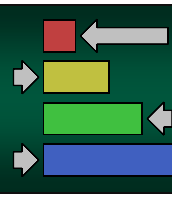
Recursive Sorting

Part 6

1



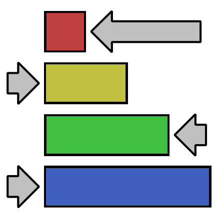
Merging Arrays

Quite easy... and quite common

2

Merging Arrays

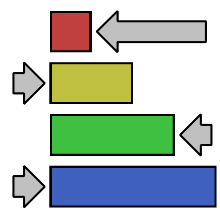
- It is a common task in Computer Science to combine two different arrays into one
- If both arrays are unsorted…
  - the task is fairly simple O(n)
  - just add one onto the end of the other



3

Merging Arrays

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



4

Merging Arrays

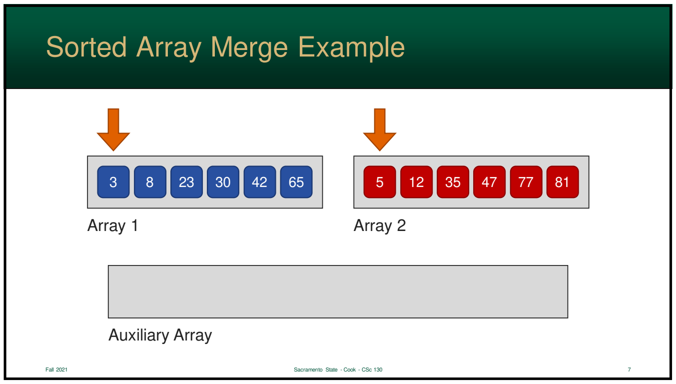
- The algorithm for merging two sorted arrays is very simple
- The resulting time complexity is O(n)
- However, it requires auxiliary storage of O(n)

5

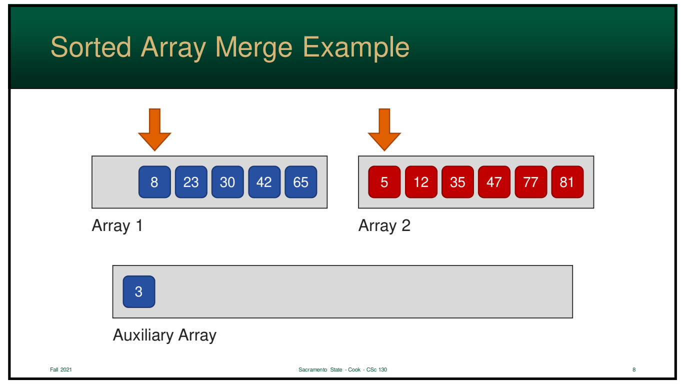
Merge Algorithm

- 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

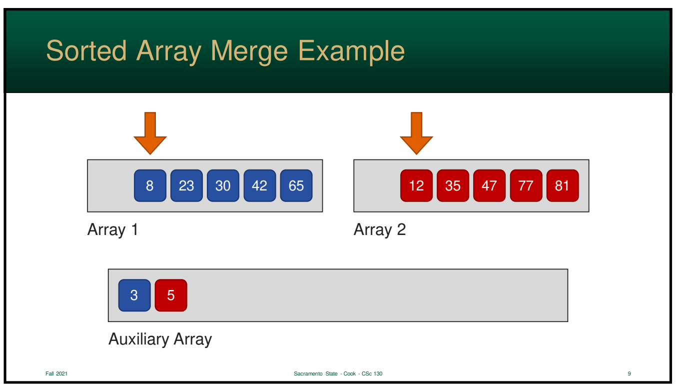
6


Sorted Array Merge Example
3 8 23 30 42 65
Array 1
5 12 35 47 77 81
Array 2
Auxiliary Array

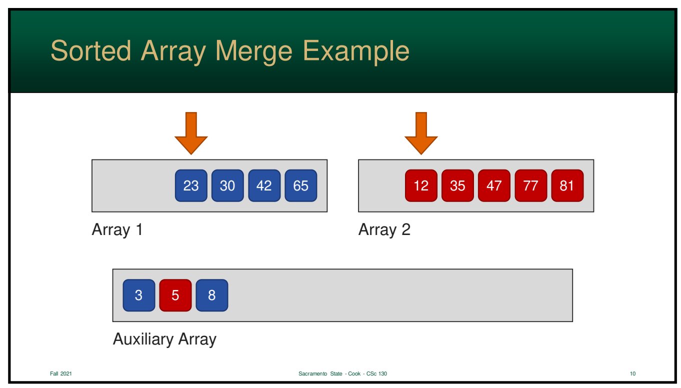
7


Sorted Array Merge Example
8 23 30 42 65
Array 1
5 12 35 47 77 81
Array 2
3
Auxiliary Array

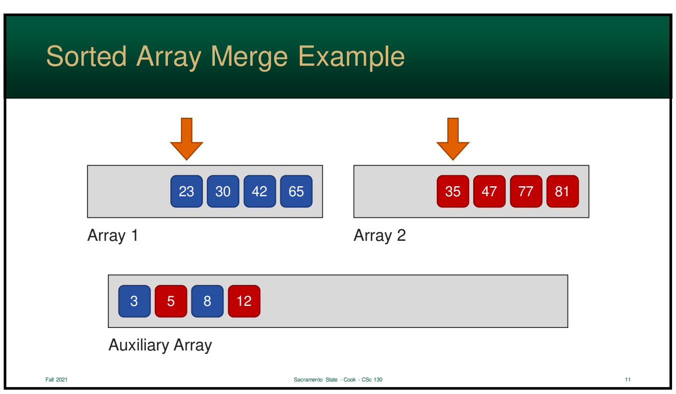
8


Sorted Array Merge Example
8 23 30 42 65
Array 1
12 35 47 77 81
Array 2
3 5
Auxiliary Array

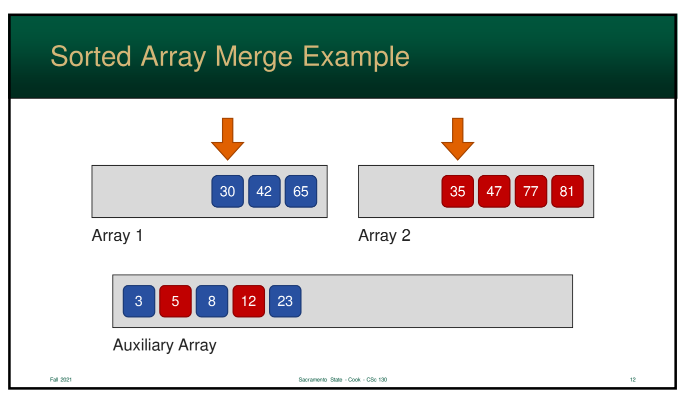
9


Sorted Array Merge Example
23 30 42 65
Array 1
12 35 47 77 81
Array 2
3 5 8
Auxiliary Array

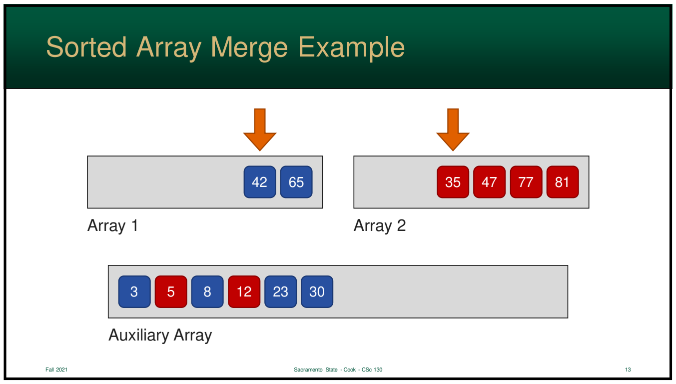
10


Sorted Array Merge Example
23 30 42 65
Array 1
35 47 77 81
Array 2
3 5 8 12
Auxiliary Array

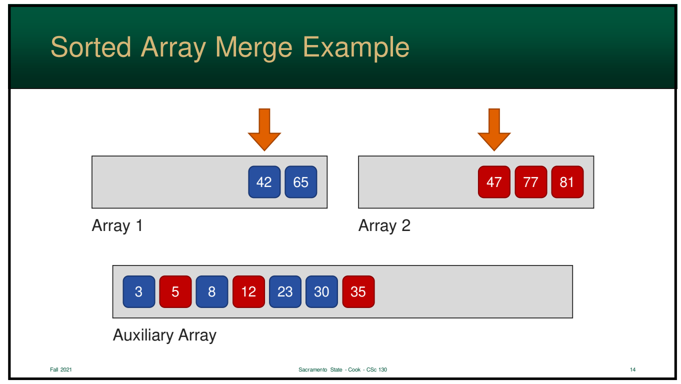
11


Sorted Array Merge Example
30 42 65
Array 1
35 47 77 81
Array 2
3 5 8 12 23
Auxiliary Array

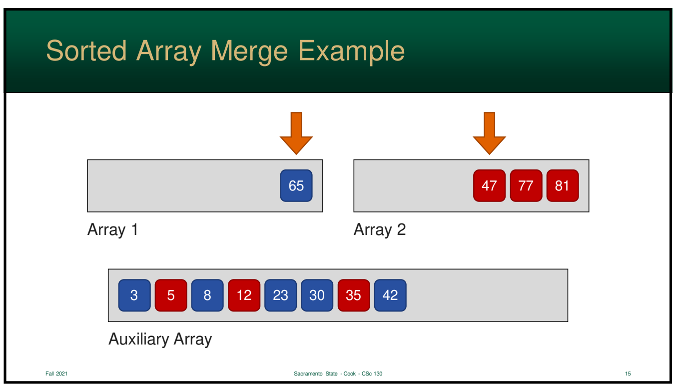
12



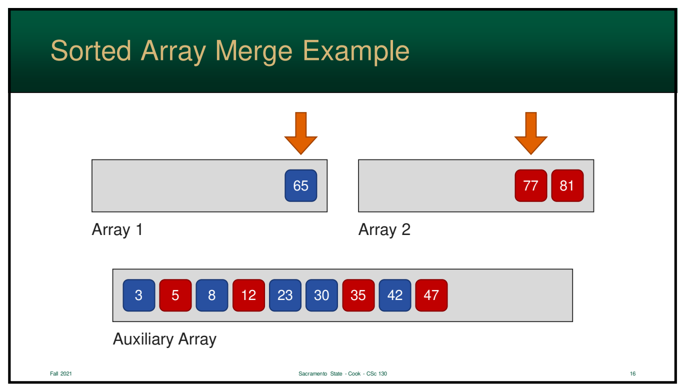
13



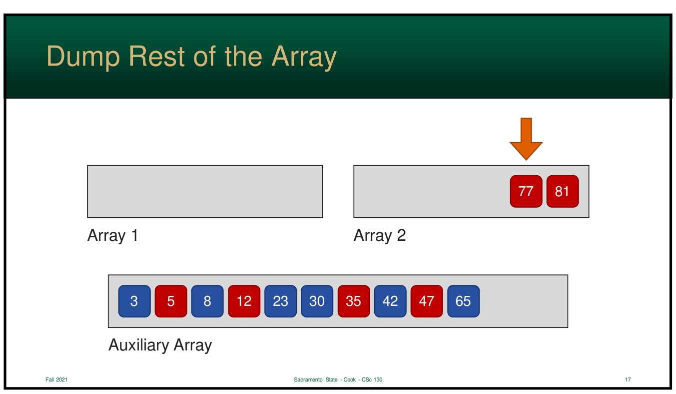
14



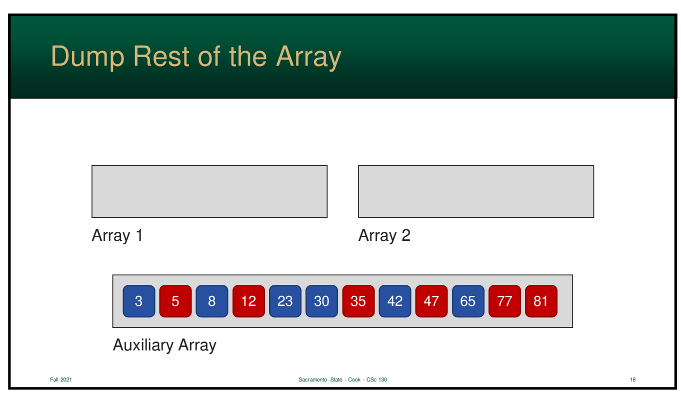
15



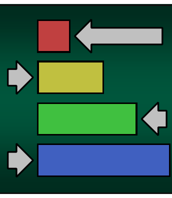
16



17



18



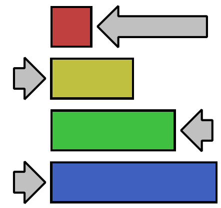
Merge Sort

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



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

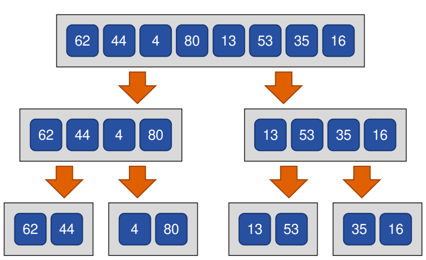
21

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

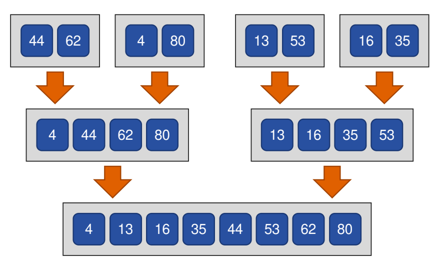
22

Merge Sort Example: Recurse down



23

Sort Merge Sort Example: Merge Up

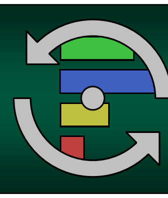


24

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

25



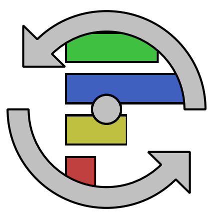
Quick Sort

Oh, I am getting dizzy....

26

Quick Sort

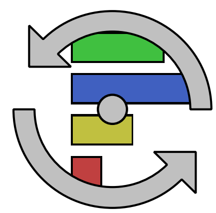
- *Quick-Sort* is a divide-and-conquer algorithm that rotates values around a *pivot*
- Invented by *C. A. R. Hoare* in 1959
- Even faster than both Merge Sort and Heap Sort
- ... but has a weaknesses



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



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

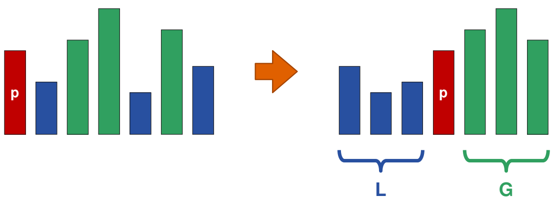
29

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*

30

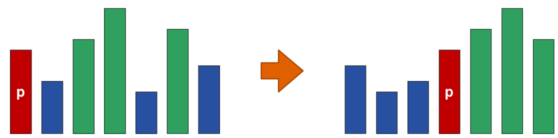
Partitioning (pivot is the first item)



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



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

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

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

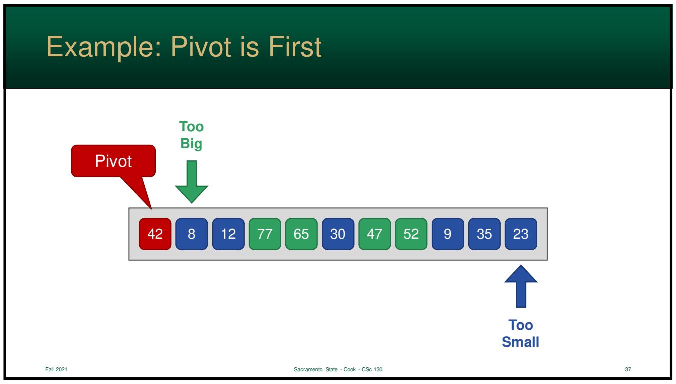
42	8	12	77	65	30	47	52	9	35	23

35

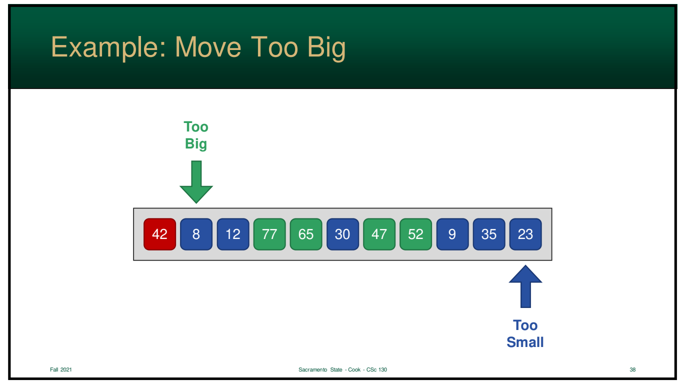
Quick Sort Algorithm

```
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]
//Recurse QuickSort on both L and G
```

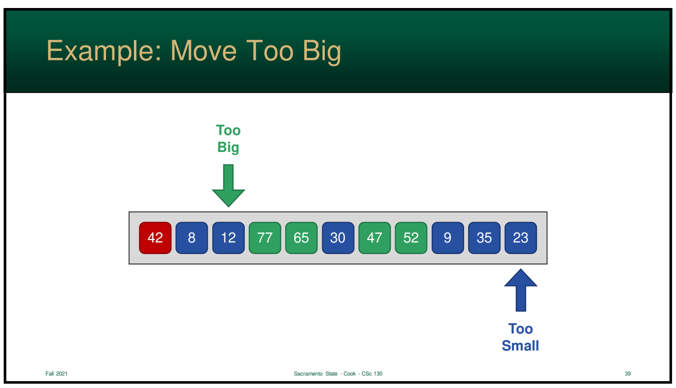

36



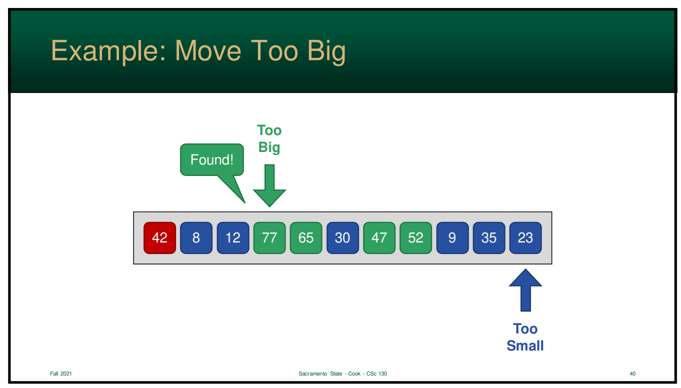
37



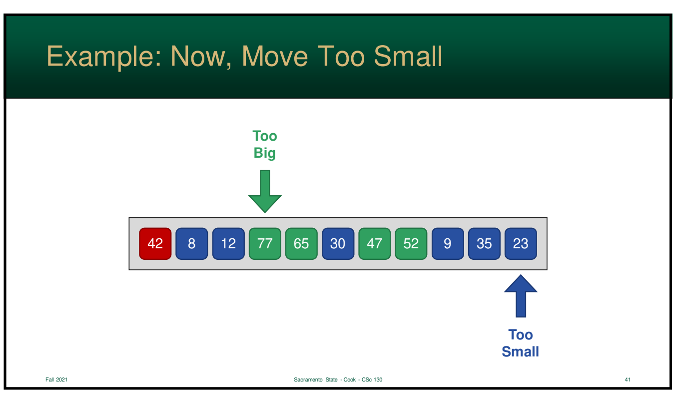
38



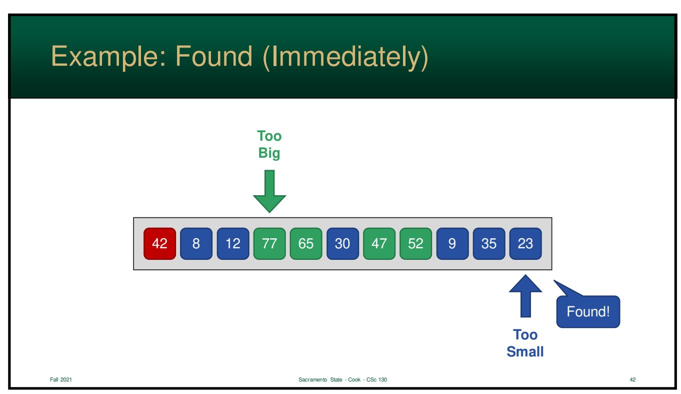
39



40

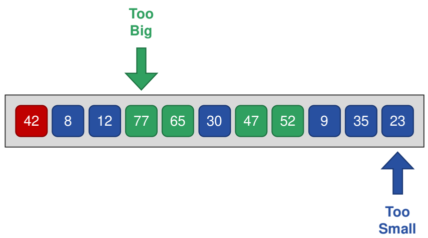


41



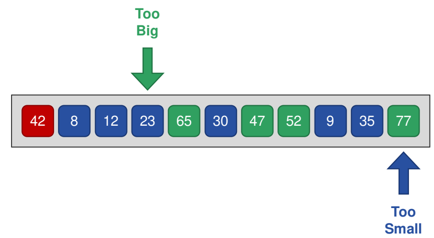
42

Example: Swap Values



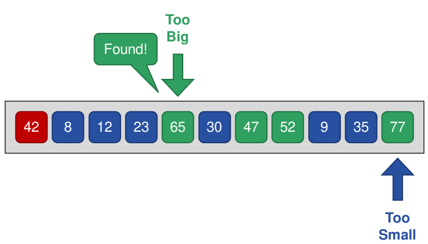
43

Example: Keep going… Move Too Big



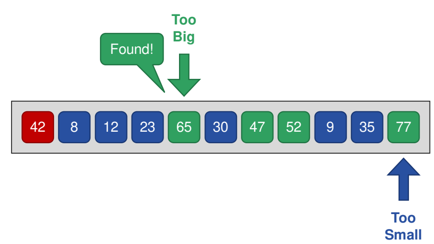
44

Example: Too Big Found



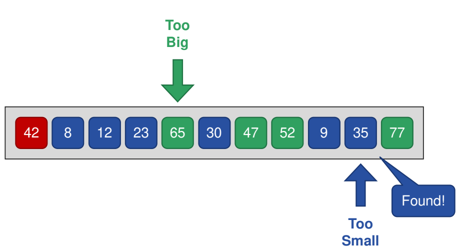
45

Example: Move Too Small



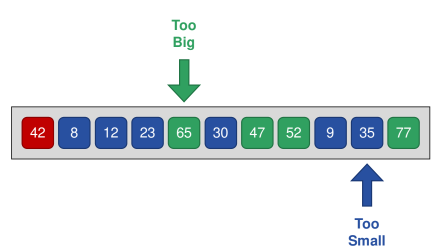
46

Example: Too Small Found

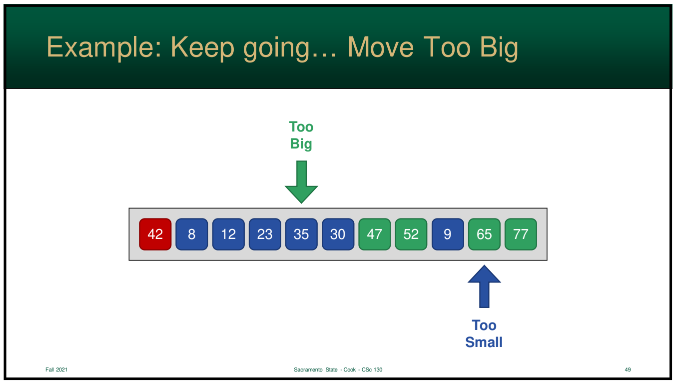


47

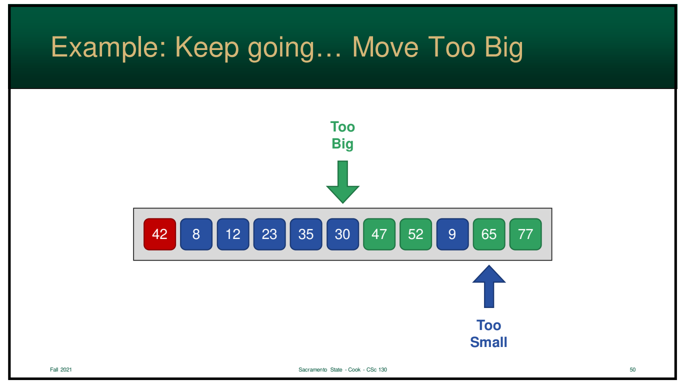
Example: Swap Values



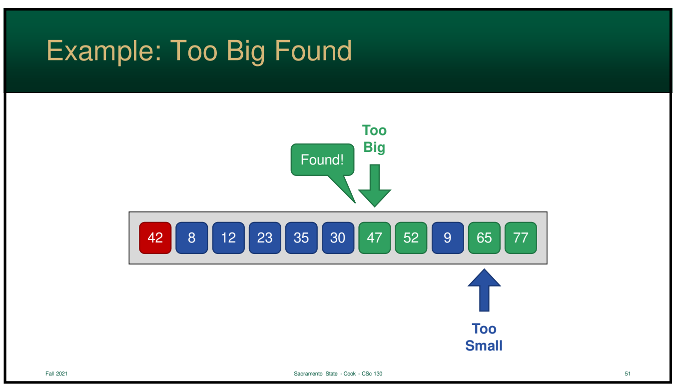
48



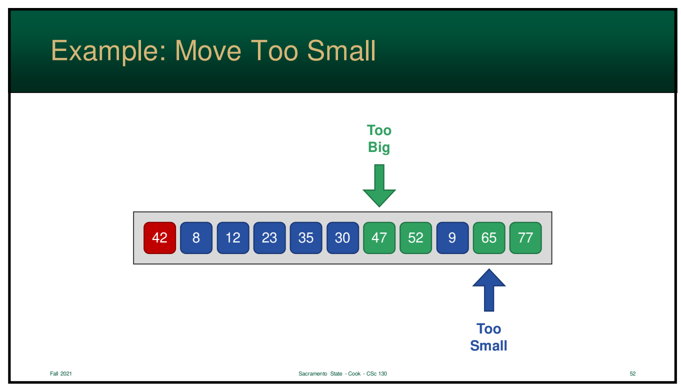
49



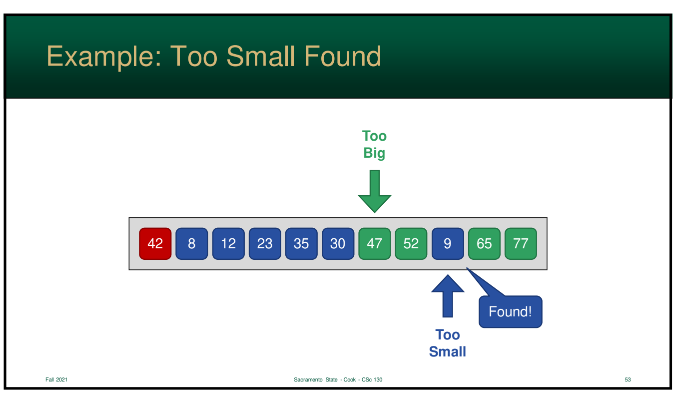
50



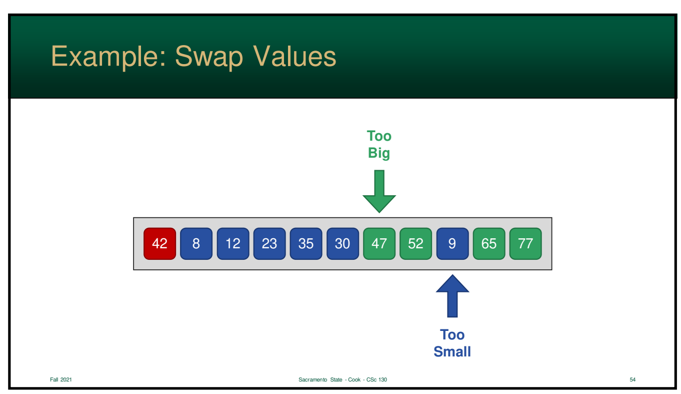
51



52

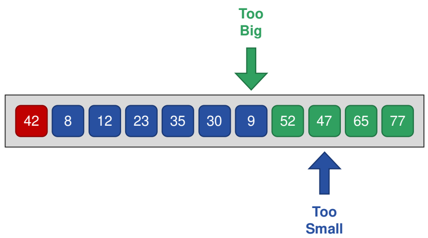


53



54

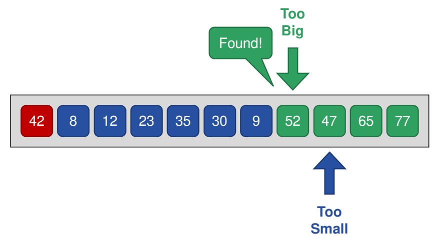
Example: Keep going… Move Too Big



Fall 2021 Sacramento State - Cook - CSc 130 55

55

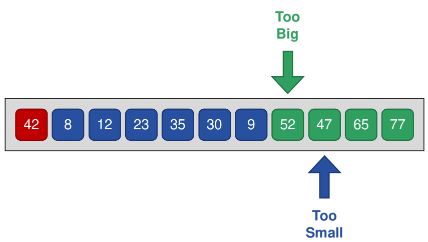
Example: Too Big Found



Fall 2021 Sacramento State - Cook - CSc 130 56

56

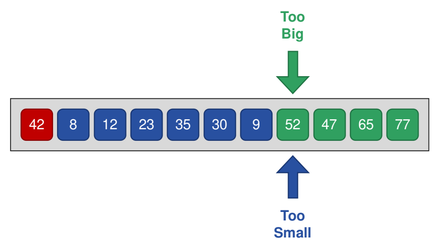
Example: Move Too Small



Fall 2021 Sacramento State - Cook - CSc 130 57

57

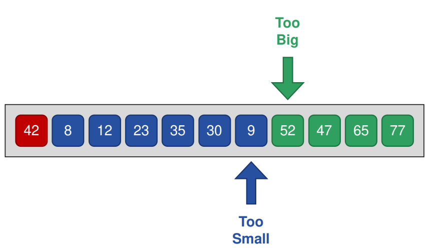
Example: Move Too Small



Fall 2021 Sacramento State - Cook - CSc 130 58

58

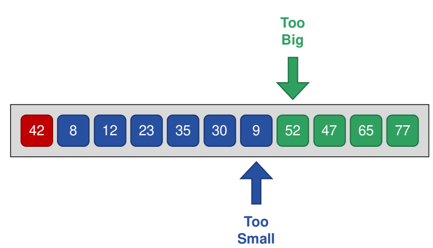
Example: Pointers Passed Each Other



Fall 2021 Sacramento State - Cook - CSc 130 59

59

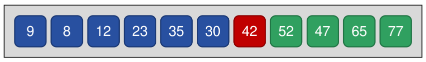
Example: Swap Pivot & Too Small



Fall 2021 Sacramento State - Cook - CSc 130 60

60

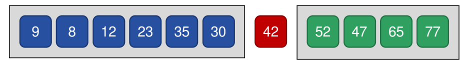
Example: Done (with this pass)



61

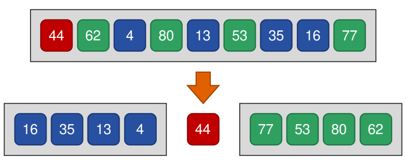
Recursion Time!

- 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



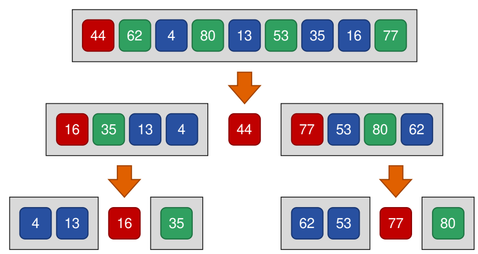
62

Quick Sort Example



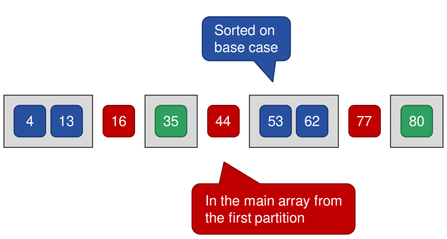
63

Quick Sort Example



64

Quick Sort Example



65

Quick Sort: Worst Case

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

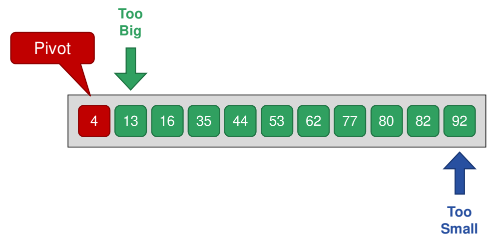
66

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

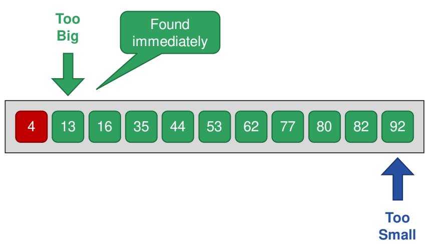
67

Quick Sort: Worst Case



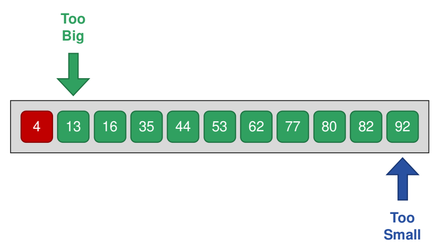
68

Worst Case: Move Too Big



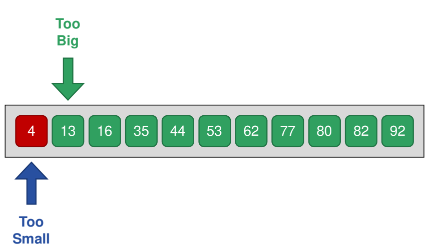
69

Worst Case: Now, Move Too Small



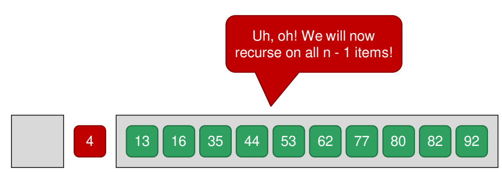
70

Worst Case: Pointers Passed



71

Worst Case: Recurse on n-1



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

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]

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

75