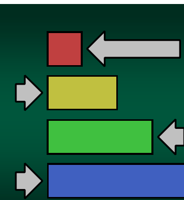
O(n Log n) Sorting

Section 2.2 – 2.3

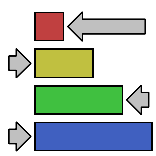


Merging Arrays

Quite easy... and quite common

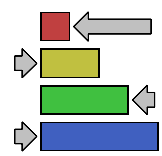
Merging Arrays

- 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



Merging Arrays

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



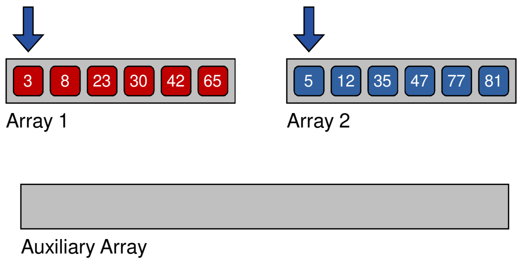
Merging Arrays

- 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)

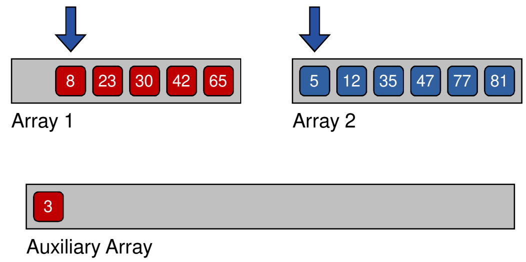
Merge Algorithm

- 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

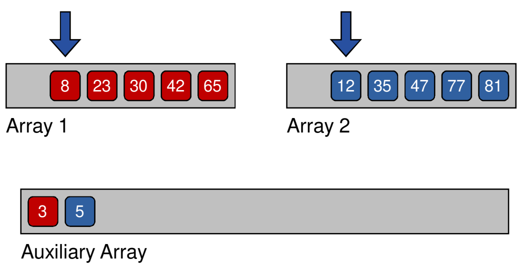
Sorted Array Merge Example



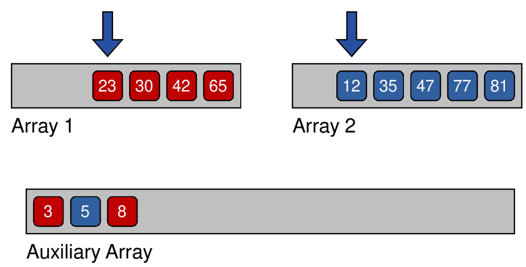
Sorted Array Merge Example



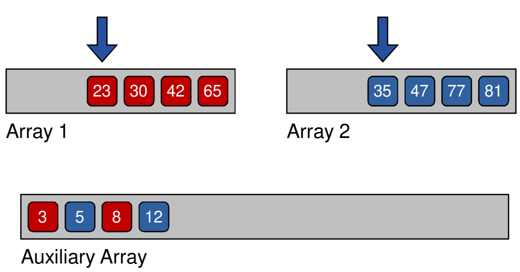
Sorted Array Merge Example



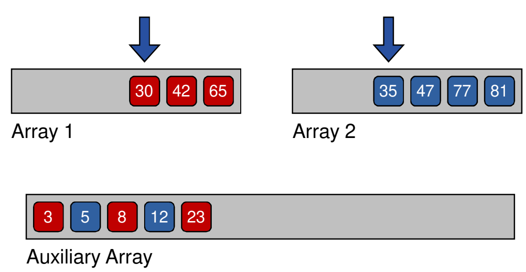
Sorted Array Merge Example



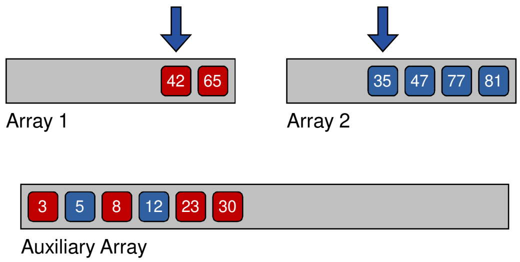
Sorted Array Merge Example



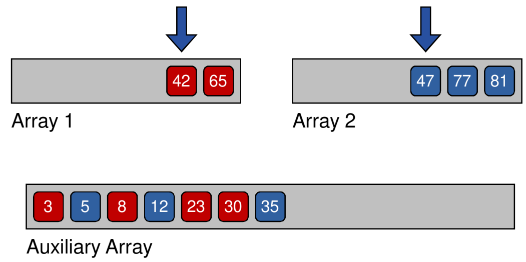
Sorted Array Merge Example



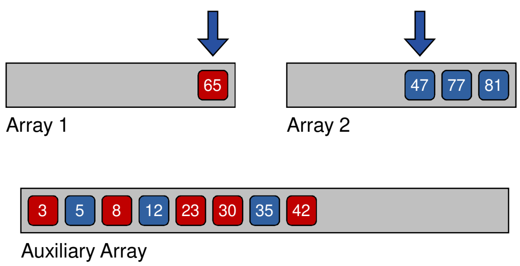
Sorted Array Merge Example



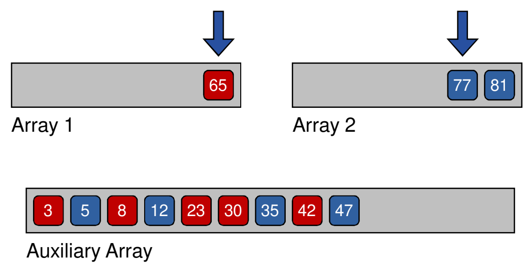
Sorted Array Merge Example



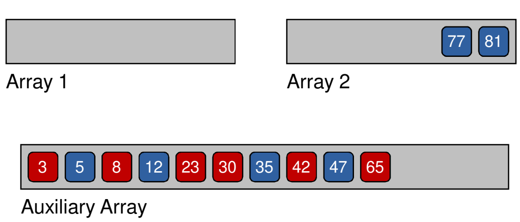
Sorted Array Merge Example



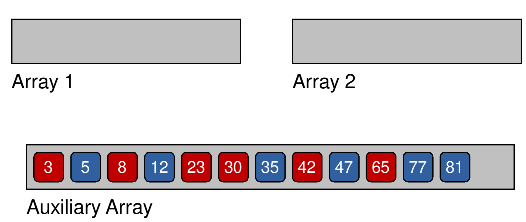
Sorted Array Merge Example

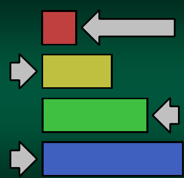


Dump the Rest of the Array



Dump the Rest of the Array



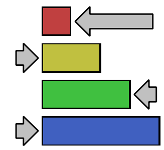


Merge Sort

Divide and conquer!

Merge Sort

- *Merge Sort* is a divide-and-conquer approach to sorting arrays
- It is based on the idea of cutting the array into smaller and smaller sublists until sorting them is arbitrary
- This approach ultimately results in O(n log n)



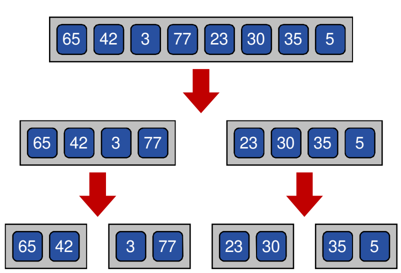
Merge Sort

- 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 lists into 2 until the list is either 1 element – which is, well, obviously sorted

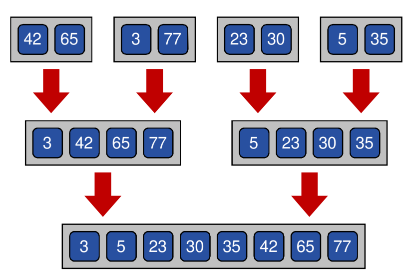
Merge Sort

- As the recursion bubbles up, each sub list is merged using the algorithm we just discussed
- Since an auxiliary array is required for the merge process, Merge-Sort, while fast, has O(n) auxiliary storage requirements

Merge Sort Example: Recurse down

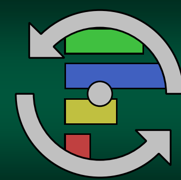


Merge Sort Example: Merge Up



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

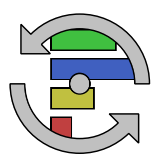


Quick Sort

Oh, I am getting dizzy....

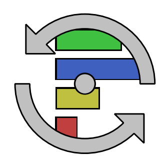
Quick Sort

- *Quick-Sort* is a divide-and-conquer algorithm that rotates values around a *pivot*
- Even faster than both Merge Sort and Heap Sort
- ... but does have its weaknesses



How it Works

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

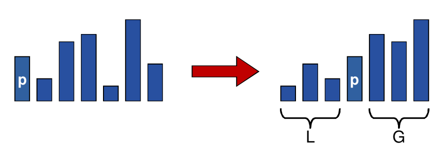


Choosing a Pivot

- The pivot can be any element in the sub-list – we just need one real value to compare
- This value is used to *partition* the values
- Different versions use different pivots
  - first item in the sub-list
  - end item in the sub-list
  - the midpoint of the sub-list
  - random value in the sub-list

Partitioning the Values

- After the pivot is selected, all elements are moved
- Two loops move through the elements swapping elements less than/greater than the pivot

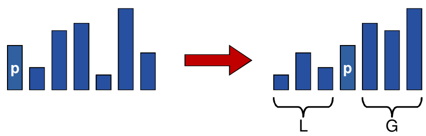


Partitioning the Values

The result is...

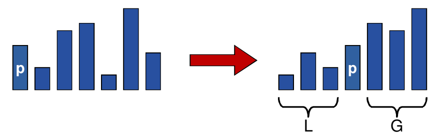
- sub-list L that contains items less than P and
- sub-list G that contains items greater than P

- The sub-lists are stored in the original data array



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



Partition Algorithm

- 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 little
- Each moves independently

Partition Algorithm

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

Example Partition

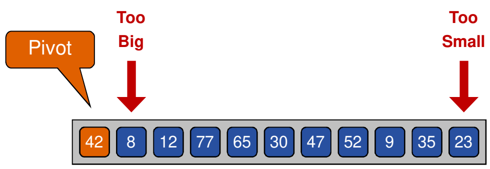
- *In this example*, we pivot at the start of the array
- Any value can be used...
  - but it will have to be swapped to the start before the algorithm runs
  - this "saves" the pivot for later

42	8	12	77	65	35	47	52	5	30	23

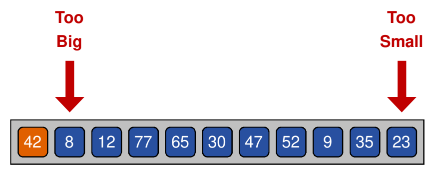
Quick Sort Algorithm

```
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
```

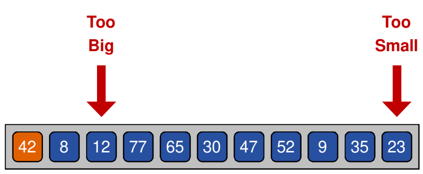
Example: Pivot is First



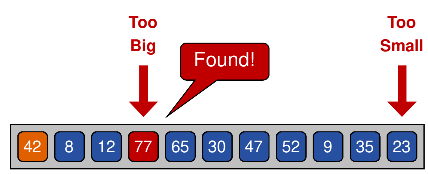
Move Too Big



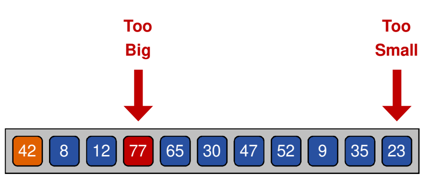
Move Too Big



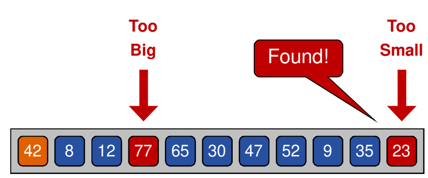
Move Too Big: Found!



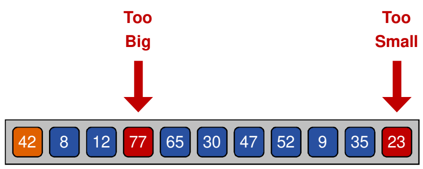
Now, Move Too Small



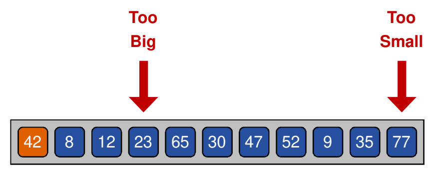
Move Too Small: Found!



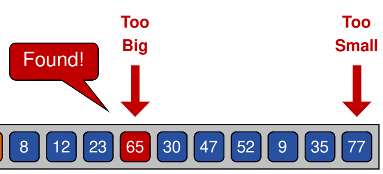
Swap these two



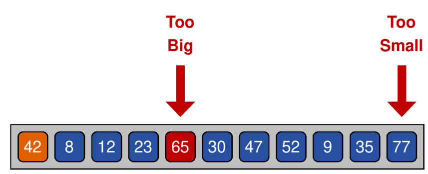
Keep going… Move Too Big



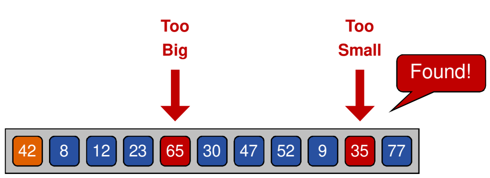
Move Too Big: Found



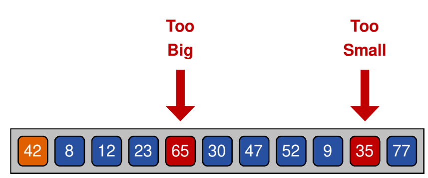
Move Too Small



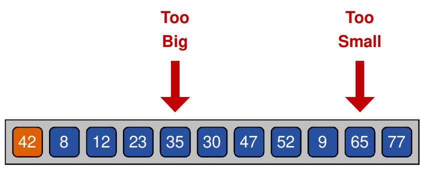
Move Too Small: Found



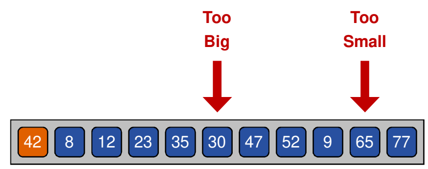
Swap



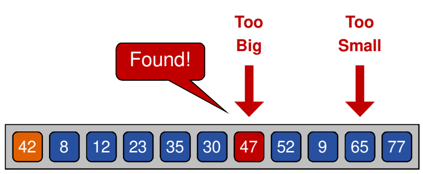
Still going… Move Too Big



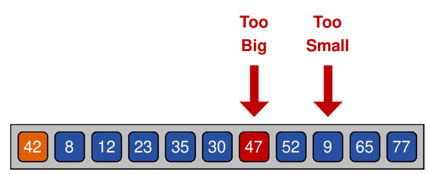
Move Too Big



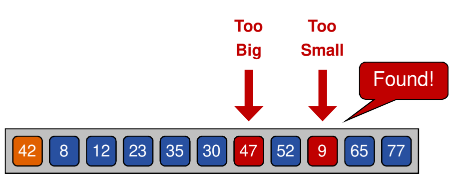
Move Too Big: Found



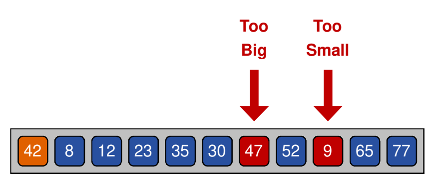
Move Too Small



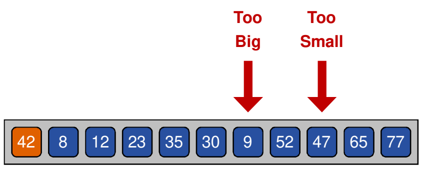
Move Too Small: Found



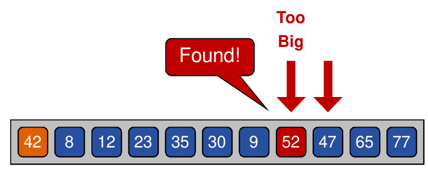
Swap



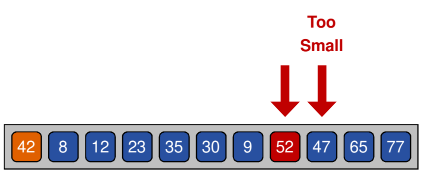
And again… Move Too Big



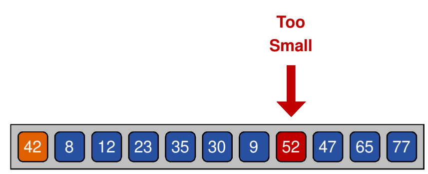
Move Too Big: Found



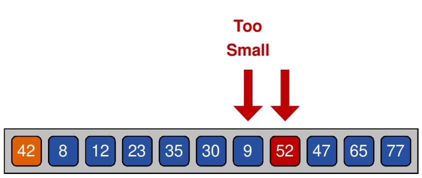
Move Too Small



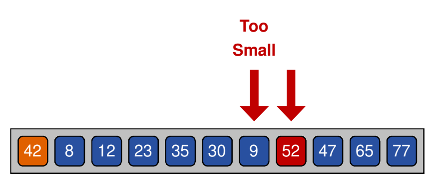
Move Too Small



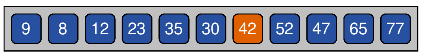
Pointers Passed Each Other!



Finally, Swap Pivot and Too Small

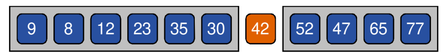


Done (with this pass)

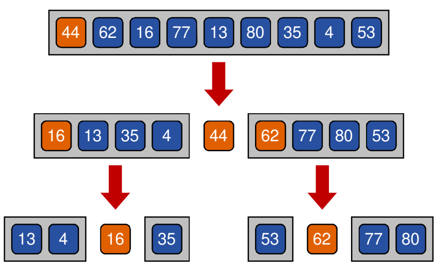


Recursion Time!

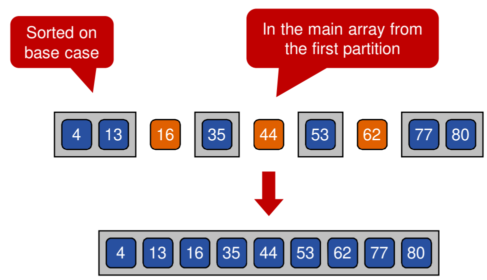
- Notice: all the items before the pivot are smaller and all the items after are a larger
- Now, we can recurse both sides
- The end result is a sorted array



Quick Sort Example



Quick Sort Example



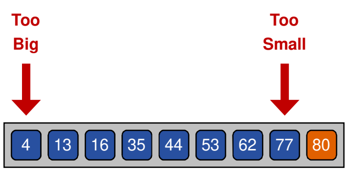
Quick Sort: Worst Case

- 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

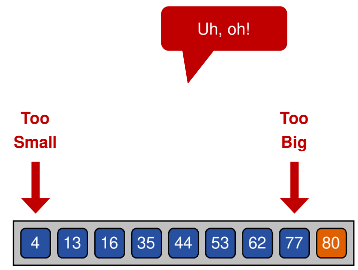
Quick Sort: Worst Case

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

Quick Sort: Worst Case



Quick Sort: Worst Case



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

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]

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