

Intermediate Sorting Algorithms

Don't be scared!

Objectives

- Understand the limitations of the sorting algorithms we've learned so far
- Implement merge sort
- Implement quick sort
- Implement radix sort

WHY LEARN THIS?

- The sorting algorithms we've learned so far don't scale well
- Try out bubble sort on an array of 100000 elements, it will take quite some time!
- We need to be able to sort large arrays more quickly

FASTER SORTS

- There is a family of sorting algorithms that can improve time complexity from $O(n^2)$ to $O(n \log n)$
- There's a tradeoff between efficiency and simplicity
- The more efficient algorithms are much less simple, and generally take longer to understand
- Let's dive in!

Merge Sort

- It's a combination of two things - merging and sorting!
- Exploits the fact that arrays of 0 or 1 element are always sorted
- Works by decomposing an array into smaller arrays of 0 or 1 elements, then building up a newly sorted array

How does it work?

[8, 3, 5, 4, 7, 6, 1, 2]

[8, 3, 5, 4]

[7, 6, 1, 2]

[8, 3]

[5, 4]

[7, 6]

[1, 2]

[8]

[3]

[5]

[4]

[7]

[6]

[1]

[2]

[3, 8]

[4, 5]

[6, 7]

[1, 2]

[3, 4, 5, 8]

[1, 2, 6, 7]

[1, 2, 3, 4, 5, 6, 7, 8]

Let's visualize this!

Merging Arrays

- In order to implement merge sort, it's useful to first implement a function responsible for merging two sorted arrays
- Given two arrays which are sorted, this helper function should create a new array which is also sorted, and consists of all of the elements in the two input arrays
- This function should run in **$O(n + m)$** time and **$O(n + m)$** space and **should not** modify the parameters passed to it.

Merging Arrays Pseudocode

- Create an empty array, take a look at the smallest values in each input array
- While there are still values we haven't looked at...
 - If the value in the first array is smaller than the value in the second array, push the value in the first array into our results and move on to the next value in the first array
 - If the value in the first array is larger than the value in the second array, push the value in the second array into our results and move on to the next value in the second array
 - Once we exhaust one array, push in all remaining values from the other array

YOUR

TURN

mergeSort Pseudocode

- Break up the array into halves until you have arrays that are empty or have one element
- Once you have smaller sorted arrays, merge those arrays with other sorted arrays until you are back at the full length of the array
- Once the array has been merged back together, return the merged (and sorted!) array

[10,24,73,76]

mergeSort([10,24,76,73])

[10,24]

merge

[73,76]

mergeSort([10,24])

mergeSort([76,73])

[10]

merge

[24]

[76]

merge

[73]

mergeSort([10])

mergeSort([24])

mergeSort([76])

mergeSort([73])

YOUR

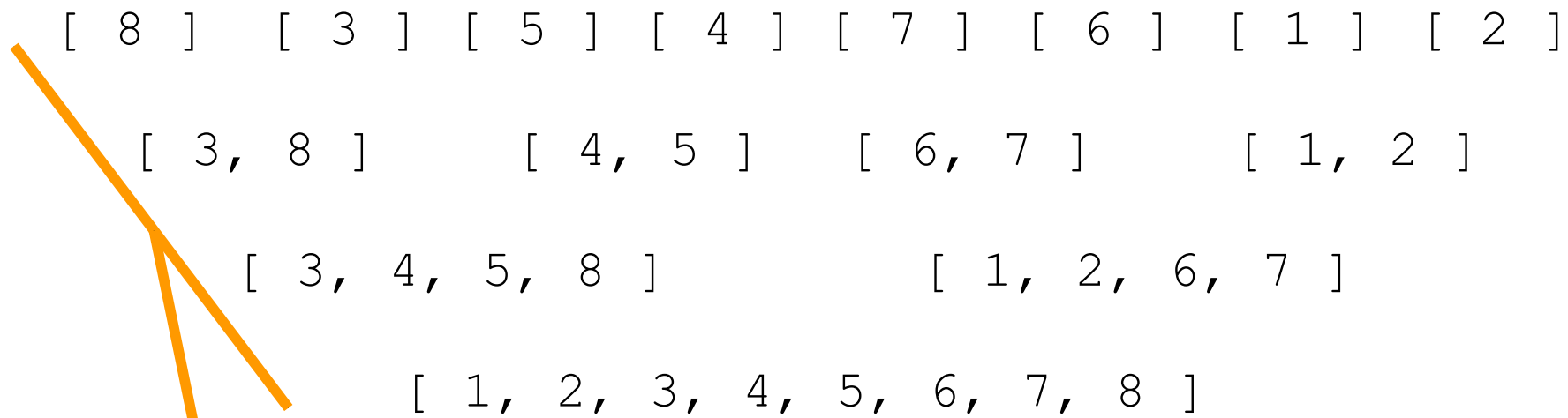
TURN

Big O of mergeSort

Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity
$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

Big O of mergeSort

Why???



$O(\log n)$ decompositions

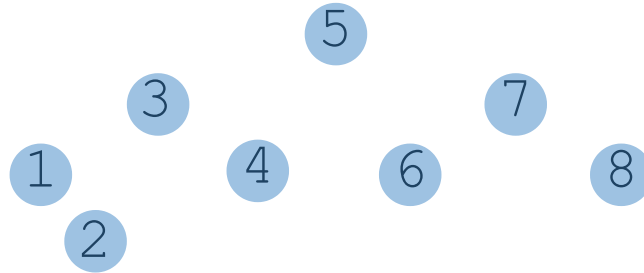
$O(n)$ comparisons per decomposition

Quick Sort

- Like merge sort, exploits the fact that arrays of 0 or 1 element are always sorted
- Works by selecting one element (called the "pivot") and finding the index where the pivot should end up in the sorted array
- Once the pivot is positioned appropriately, quick sort can be applied on either side of the pivot

How does it work?

[5, 2, 1, 8, 4, 7, 6, 3]



Let's visualize this!

Pivot Helper

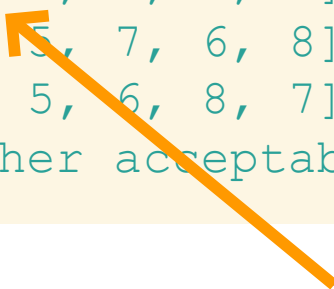
- In order to implement merge sort, it's useful to first implement a function responsible arranging elements in an array on either side of a pivot
- Given an array, this helper function should designate an element as the pivot
- It should then rearrange elements in the array so that all values less than the pivot are moved to the left of the pivot, and all values greater than the pivot are moved to the right of the pivot
- The order of elements on either side of the pivot doesn't matter!
- The helper should do this **in place**, that is, it should not create a new array
- When complete, the helper should return the index of the pivot

Picking a pivot

- The runtime of quick sort depends in part on how one selects the pivot
- Ideally, the pivot should be chosen so that it's roughly the median value in the data set you're sorting
- For simplicity, we'll always choose the pivot to be the first element (we'll talk about consequences of this later)

Pivot Helper Example

```
let arr = [ 5, 2, 1, 8, 4, 7, 6, 3 ]  
  
pivot(arr); // 4;  
  
arr;  
// any one of these is an acceptable mutation:  
// [2, 1, 4, 3, 5, 8, 7, 6]  
// [1, 4, 3, 2, 5, 7, 6, 8]  
// [3, 2, 1, 4, 5, 7, 6, 8]  
// [4, 1, 2, 3, 5, 6, 8, 7]  
// there are other acceptable mutations too!
```



All that matters is for 5 to be at index 4, for smaller values to be to the left, and for larger values to be to the right

Pivot Pseudocode

- It will help to accept three arguments: an array, a start index, and an end index (these can default to 0 and the array length minus 1, respectively)
- Grab the pivot from the start of the array
- Store the current pivot index in a variable (this will keep track of where the pivot should end up)
- Loop through the array from the start until the end
 - If the pivot is greater than the current element, increment the pivot index variable and then swap the current element with the element at the pivot index
- Swap the starting element (i.e. the pivot) with the pivot index
- Return the pivot index

YOUR

TURN

Quicksort Pseudocode

- Call the pivot helper on the array
- When the helper returns to you the updated pivot index, recursively call the pivot helper on the subarray to the left of that index, and the subarray to the right of that index
- Your base case occurs when you consider a subarray with less than 2 elements

YOUR

TURN

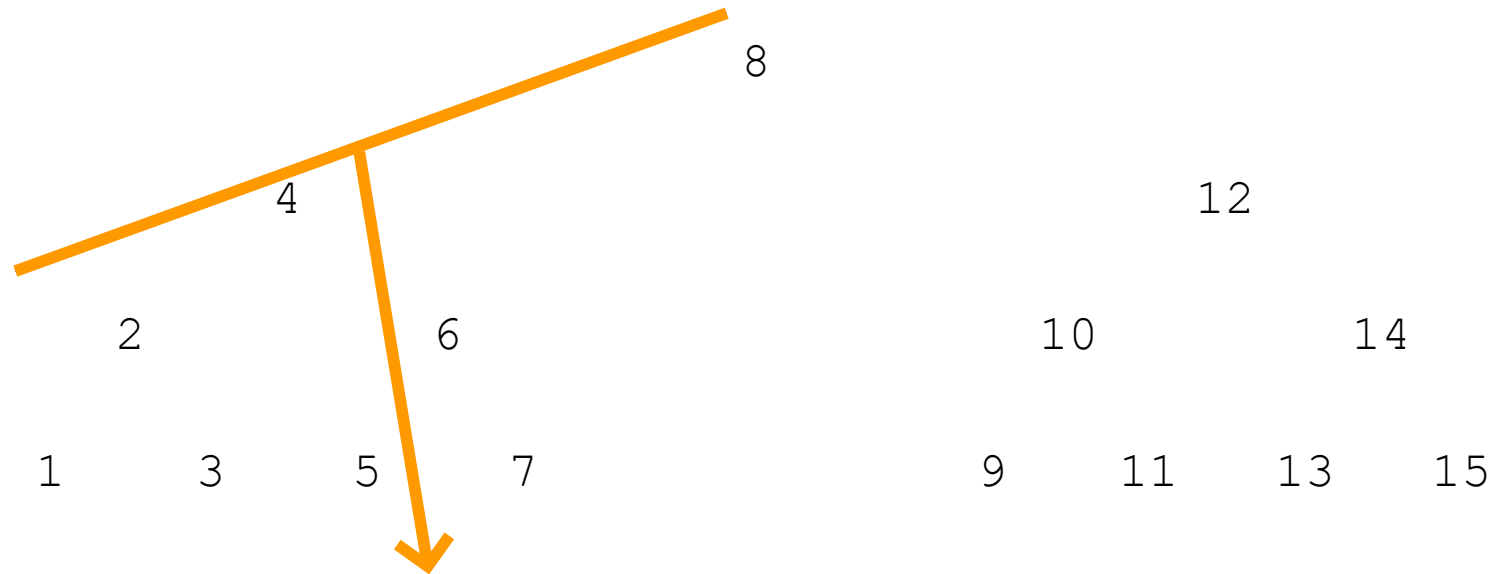
Big O of Quicksort

Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity
$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$

Big O of Quicksort

Why???

Best Case



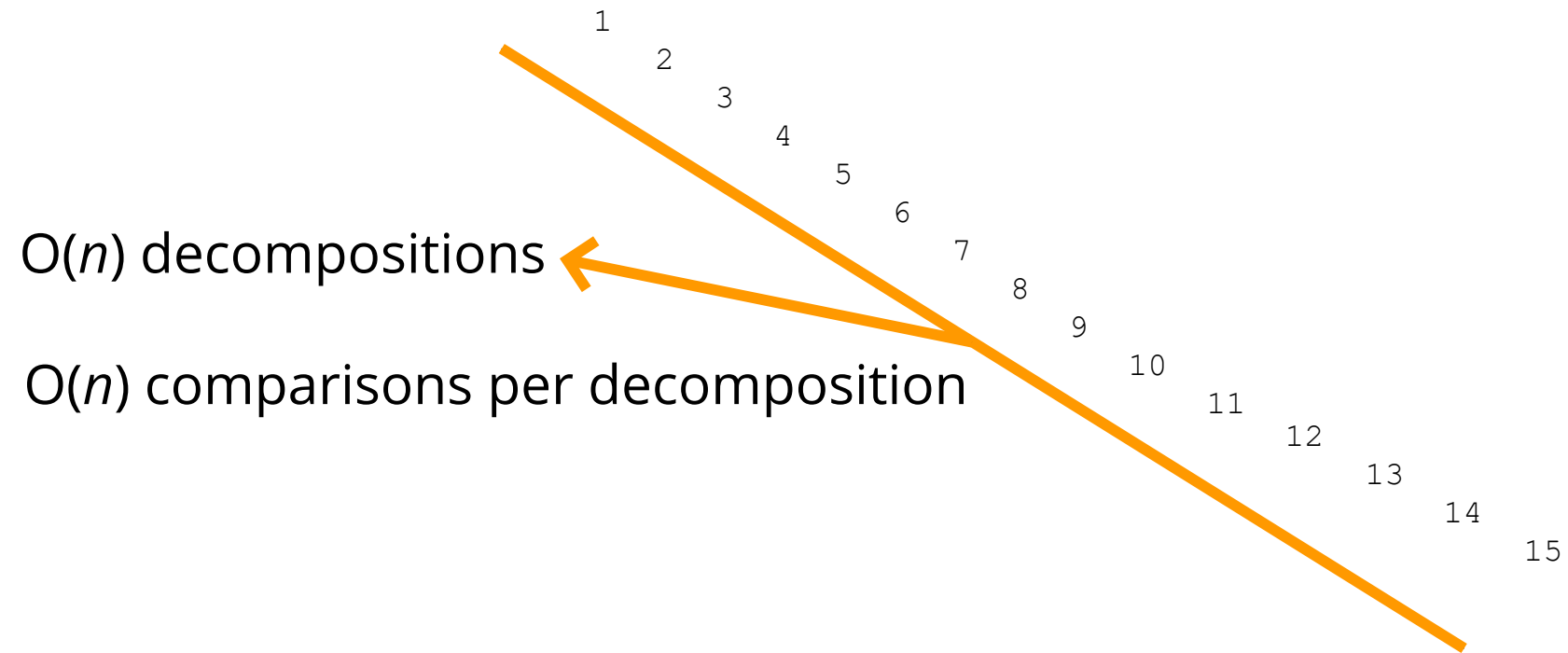
$O(\log n)$ decompositions

$O(n)$ comparisons per decomposition

Big O of Quicksort

Why???

Worst Case



COMPARISON SORTS

Average Time Complexity

- Bubble Sort - $O(n^2)$
- Insertion Sort - $O(n^2)$
- Selection Sort - $O(n^2)$
- Quick Sort - $O(n \log(n))$
- Merge Sort - $O(n \log(n))$

Can we do better?

CAN WE DO
BETTER?

YES,

BUT NOT BY MAKING
COMPARISONS

RADIX

SORT

RADIX SORT

Radix sort is a special sorting algorithm that works on lists of numbers

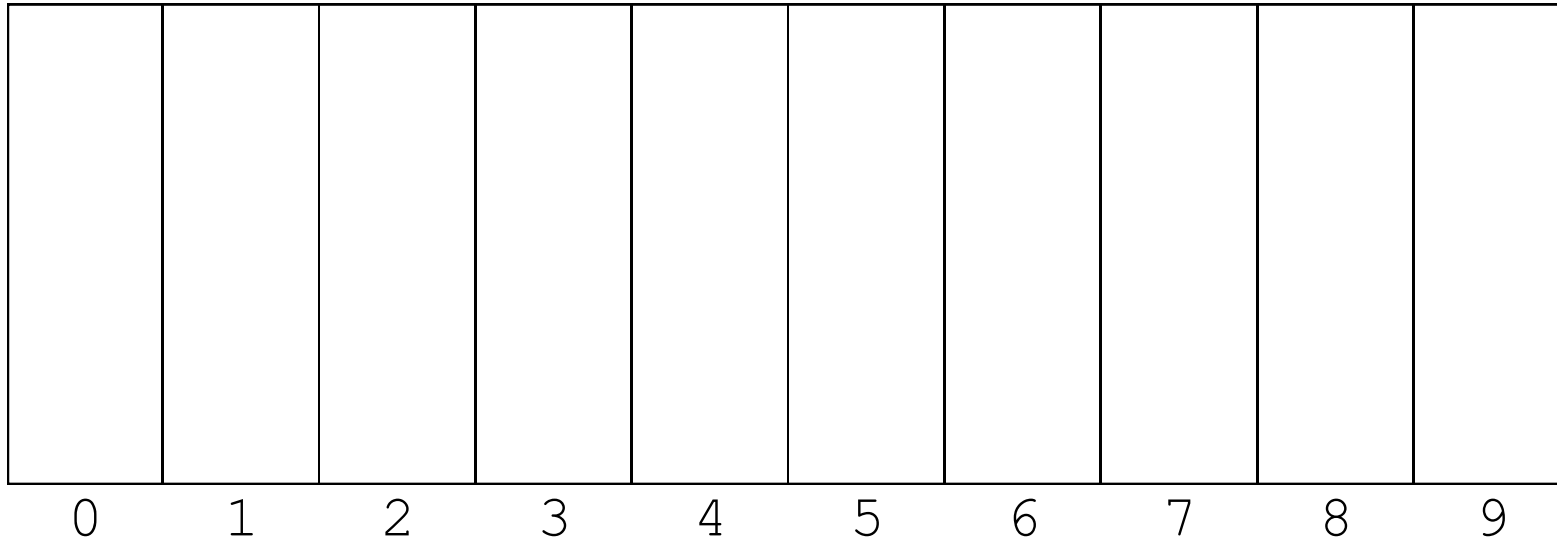
It never makes comparisons between elements!

It exploits the fact that information about the size of a number is encoded in the number of digits.

More digits means a bigger number!

How does it work?

[4, 7, 29, 86, 408, 593, 902, 1556, 3556, 4386, 8157, 9637]



Let's visualize this!

RADIX SORT HELPERS

In order to implement radix sort, it's helpful to build a few helper functions first:

`getDigit(num, place)` - returns the digit in *num* at the given *place* value

```
getDigit(12345, 0); // 5
getDigit(12345, 1); // 4
getDigit(12345, 2); // 3
getDigit(12345, 3); // 2
getDigit(12345, 4); // 1
getDigit(12345, 5); // 0
```


RADIX SORT HELPERS

In order to implement radix sort, it's helpful to build a few helper functions first:

`getDigit(num, place)` - returns the digit in *num* at the given *place* value

```
function getDigit(num, i) {  
  return Math.floor(Math.abs(num) / Math.pow(10, i)) % 10;  
}
```

RADIX SORT HELPERS

In order to implement radix sort, it's helpful to build a few helper functions first:

`digitCount(num)` - returns the number of digits in *num*

```
digitCount(1); // 1  
digitCount(25); // 2  
digitCount(314); // 3
```

RADIX SORT HELPERS

In order to implement radix sort, it's helpful to build a few helper functions first:

`digitCount(num)` - returns the number of digits in *num*

```
function digitCount(num) {  
  if (num === 0) return 1;  
  return Math.floor(Math.log10(Math.abs(num))) + 1;  
}
```

RADIX SORT HELPERS

In order to implement radix sort, it's helpful to build a few helper functions first:

`mostDigits(nums)` - Given an array of numbers, returns the number of digits in the largest numbers in the list

```
mostDigits([1234, 56, 7]); // 4
mostDigits([1, 1, 11111, 1]); // 5
mostDigits([12, 34, 56, 78]); // 2
```

RADIX SORT HELPERS

In order to implement radix sort, it's helpful to build a few helper functions first:

`mostDigits(nums)` - Given an array of numbers, returns the number of digits in the largest numbers in the list

```
function mostDigits(nums) {  
  let maxDigits = 0;  
  for (let i = 0; i < nums.length; i++) {  
    maxDigits = Math.max(maxDigits, digitCount(nums[i]));  
  }  
  return maxDigits;  
}
```

YOUR

TURN

RADIX SORT PSEUDOCODE

- Define a function that accepts list of numbers
- Figure out how many digits the largest number has
- Loop from $k = 0$ up to this largest number of digits
- For each iteration of the loop:
 - Create buckets for each digit (0 to 9)
 - place each number in the corresponding bucket based on its k th digit
- Replace our existing array with values in our buckets, starting with 0 and going up to 9
- return list at the end!

YOUR

TURN

RADIX SORT BIG O

Time Complexity (Best)	Time Complexity (Average)	Time Complexity (Worst)	Space Complexity
$O(nk)$	$O(nk)$	$O(nk)$	$O(n + k)$

- n - length of array
- k - number of digits(average)

Recap

- Merge sort and quick sort are standard efficient sorting algorithms
- Quick sort can be slow in the worst case, but is comparable to merge sort on average
- Merge sort takes up more memory because it creates a new array (in-place merge sorts exist, but they are really complex!)
- Radix sort is a fast sorting algorithm for numbers
- Radix sort exploits place value to sort numbers in linear time (for a fixed number of digits)