Lec 16: Sorting

Sam McCauley

March 18, 2026

Admin

- Lab today; remember that you need to do the lab in-person during t
- Quiz Friday: Sorting and big-*O* notation
  - I will ask you to prove the big-*O* bound for an algorithm
  - Be sure to know the definition, and how to apply it
- Anything else?

Sorting

Sorting

- Goal: sequence of steps

Sorting

- Goal: sequence of steps
- Guarantee that the cards are sorted at the end

Sorting

- Goal: sequence of steps
- Guarantee that the cards are sorted at the end
- We want to be able to:

Sorting

- Goal: sequence of steps
- Guarantee that the cards are sorted at the end
- We want to be able to:
    - Code it up in Java

Sorting

- Goal: sequence of steps
- Guarantee that the cards are sorted at the end
- We want to be able to:
  - Code it up in Java
  - Analyze the running time

Specifics

- Have an array of numbers

Specifics

- Have an array of numbers
- Want to sort them *in place* (without copying to a new array)

Specifics

- Have an array of numbers
- Want to sort them *in place* (without copying to a new array)
    - In other words: sort them using $O(1)$ extra space.

Specifics

- Have an array of numbers
- Want to sort them *in place* (without copying to a new array)
    - In other words: sort them using $O(1)$ extra space.

Where to Start?

10	21	-3	40	17	13	11	-4

- Is there any number we can put directly in the correct place?

Where to Start?

10	21	-3	40	17	13	11	-4

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot

Where to Start?

10	21	-3	40	17	13	11	-4

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number

Where to Start?

10	21	-3	40	17	13	11	-4

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?

Where to Start?

10	21	-3	40	17	13	11	-4

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?
  - $O(n)$

Where to Start?

10	21	-3	40	17	13	11	-4

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?
  - $O(n)$
- *Swap* that number with the last number

10	21	-3	40	17	13	11	-4

Maximum so far:

Where to Start?

10	21	-3	40	17	13	11	-4

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?
  - $O(n)$
- *Swap* that number with the last number

10	21	-3	40	17	13	11	-4

Maximum so far: 10 at pos 0

Where to Start?

10	21	-3	40	17	13	11	-4

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?
  - $O(n)$
- *Swap* that number with the last number

10	21	-3	40	17	13	11	-4

Maximum so far: 21 at pos 1

Where to Start?

10	21	-3	40	17	13	11	-4

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?
  - $O(n)$
- *Swap* that number with the last number

10	21	-3	40	17	13	11	-4

Maximum so far: 21 at pos 1

Where to Start?

10	21	-3	40	17	13	11	-4

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?
  - $O(n)$
- *Swap* that number with the last number

10	21	-3	40	17	13	11	-4

Maximum so far: 40 at pos 3

Where to Start?

10	21	-3	40	17	13	11	-4

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?
  - $O(n)$
- *Swap* that number with the last number

10	21	-3	40	17	13	11	-4

Maximum so far: 40 at pos 3

Where to Start?

10	21	-3	40	17	13	11	-4

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?
  - $O(n)$
- *Swap* that number with the last number

10	21	-3	40	17	13	11	-4

Maximum so far: 40 at pos 3

Where to Start?

10	21	-3	40	17	13	11	-4

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?
  - $O(n)$
- *Swap* that number with the last number

10	21	-3	40	17	13	11	-4

Maximum so far: 40 at pos 3

Where to Start?

10	21	-3	40	17	13	11	-4

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?
  - $O(n)$
- *Swap* that number with the last number

10	21	-3	40	17	13	11	-4

Maximum so far: 40 at pos 3

Where to Start?

10	21	-3	40	17	13	11	-4

- Is there any number we can put directly in the correct place?
- We can put the largest number in the last slot
- Scan through the array to find the maximum number
  - Time?
  - $O(n)$
- *Swap* that number with the last number

10	21	-3	-4	17	13	11	40

Now what?

- Do it again! But now on all but the last element of the array

Now what?

- Do it again! But now on all but the last element of the array

- (This is essentially a recursive algorithm)

Selection Sort

Maximum so far: 10 at pos 0

Selection Sort

Maximum so far: 21 at pos 1

Selection Sort

Maximum so far: 21 at pos 1

Selection Sort

Maximum so far: 21 at pos 1

Selection Sort

Maximum so far: 21 at pos 1

Selection Sort

Maximum so far: 21 at pos 1

Selection Sort

Maximum so far: 21 at pos 1

Selection Sort

Maximum so far: 10 at pos 0

Selection Sort

Maximum so far: 11 at pos 1

Selection Sort

Maximum so far: 11 at pos 1

Selection Sort

Maximum so far: 11 at pos 1

Selection Sort

Maximum so far: 17 at pos 4

Selection Sort

Maximum so far: 17 at pos 4

Selection Sort

Maximum so far: 17 at pos 4

Selection Sort

Let’s look at the code

- We’ll do loops, not recursion

Let’s look at the code

- We’ll do loops, not recursion
- Let’s assume we have a `swap(int[], int, int)` method that swaps two indices of an array

```
1  public void swap(int[] arr, int index1, int index2) {
2      int temp = arr[index1];
3      arr[index1] = arr[index2];
4      arr[index2] = temp;
5  }
```

Selection Sort Code

```
1  public static void selectionSort(int data[]) {
2      int numUnsorted = data.length;
3      int index; // general index
4      int max; // index of largest value
5      while (numUnsorted > 0) {
6          // determine maximum value in array
7          max = 0;
8          for (index = 1; index < numUnsorted; index++) {
9              if (data[max] < data[index]) {
10                 max = index;
11             }
12         }
13         swap(data, max, numUnsorted-1);
14         numUnsorted--;
15     }
16 }
```

Selection Sort Code: Method Version

```
1  public static int findMax(int data[], int maxIndex) {
2      int max = 0;
3      for (int index = 1; index < maxIndex; index++) {
4          if (data[max] < data[index]) {
5              max = index;
6          }
7      }
8      return max;
9  }
10
11 public static void selectionSort(int data[]) {
12     int numUnsorted = data.length;
13     while (numUnsorted > 0) {
14         int max = findMax(data, numUnsorted);
15         swap(data, max, numUnsorted-1);
16         numUnsorted--;
17     }
18 }
```

Why Does This Work?

- Idea: after the loop iterates *i* times,
  - The last *i* slots of the array contain the *i* largest elements of the array in sorted order

Why Does This Work?

- Idea: after the loop iterates i times,
  - The last i slots of the array contain the i largest elements of the array in sorted order
- When $i = n$ we are done

Using Generics for Selection Sort

Generalizing our Sort

- We wrote selection sort code that only works on an array of `ints`

Generalizing our Sort

- We wrote selection sort code that only works on an array of `ints`
- Can we use the power of interfaces to write code that will work for a wide variety of objects?

Generalizing our Sort

- We wrote selection sort code that only works on an array of `ints`
- Can we use the power of interfaces to write code that will work for a wide variety of objects?
- What method(s) do we need the objects to have to use selection sort?

Selection Sort Code

```
1  public static void selectionSort(int data[]) {
2      int numUnsorted = data.length;
3      int index; // general index
4      int max; // index of largest value
5      while (numUnsorted > 0) {
6          // determine maximum value in array
7          max = 0;
8          for (index = 1; index < numUnsorted; index++) {
9              if (data[max] < data[index]) {
10                 max = index;
11             }
12         }
13         swap(data, max, numUnsorted-1);
14         numUnsorted--;
15     }
16 }
```

All we do with `data` is see if `data[max] < data[index]`.

`Comparable<T>` Interface

- This is a Java interface (Built-in; don’t need to write a file, or even import anything.)

`Comparable<T>` Interface

- This is a Java interface (Built-in; don't need to write a file, or even import anything.)
- `Comparable<T>` has only one method:
  `public int compareTo(T other)`
  - Returns a negative integer if this object is smaller than the argument; `0` if equal; a positive integer if larger

`Comparable<T>` Interface

- This is a Java interface (Built-in; don't need to write a file, or even import anything.)
- `Comparable<T>` has only one method:
  `public int compareTo(T other)`
  - Returns a negative integer if this object is smaller than the argument; `0` if equal; a positive integer if larger
- Let's look at how the selection sort code changes to use a comparable object

`Comparable<T>` Interface

- This is a Java interface (Built-in; don't need to write a file, or even import anything.)
- `Comparable<T>` has only one method:
  `public int compareTo(T other)`
  - Returns a negative integer if this object is smaller than the argument; `0` if equal; a positive integer if larger
- Let's look at how the selection sort code changes to use a comparable object
- `Integer` already implements `Comparable<Integer>` so we can already sort `Integers`; same with `Strings`

Creating a generic sorting method

- We could make the `SortTest` class generic.

Creating a generic sorting method

- We could make the `SortTest` class generic.
- Really: want to make one method generic. Can we do this in Java?

Creating a generic sorting method

- We could make the `SortTest` class generic.
- Really: want to make one method generic. Can we do this in Java?
- Yes! Looks something like this:
  - `public static <E> void SelectionSort(ArrayList<E> list)`

Creating a generic sorting method

- We could make the `SortTest` class generic.
- Really: want to make one method generic. Can we do this in Java?
- Yes! Looks something like this:
  - `public static <E> void SelectionSort(ArrayList<E> list)`
- Problem: can't sort objects of *any* class E. Needs to be comparable with other objects of class E

Generic Upper Bounds

- Way to tell Java that a generic type needs to meet certain requirements

Generic Upper Bounds

- Way to tell Java that a generic type needs to meet certain requirements
- That way, at compile time, Java can make sure our types match up

Generic Upper Bounds

- Way to tell Java that a generic type needs to meet certain requirements
- That way, at compile time, Java can make sure our types match up
- These are called *upper bounds*

Generic Upper Bounds

- Way to tell Java that a generic type needs to meet certain requirements
- That way, at compile time, Java can make sure our types match up
- These are called *upper bounds*
- Let's say we only want to accept objects that meet the requirements of the `List` interface. Rather than `<E>`, we write something like `<E extends List>`
  - (Yes, it's `extends` and not `implements`. There are some good back-end reasons for this.)

Generic Upper Bounds

- Way to tell Java that a generic type needs to meet certain requirements
- That way, at compile time, Java can make sure our types match up
- These are called *upper bounds*
- Let's say we only want to accept objects that meet the requirements of the `List` interface. Rather than `<E>`, we write something like `<E extends List>`
  - (Yes, it's `extends` and not `implements`. There are some good back-end reasons for this.)
- What do we want for our `selectionSort` method?
  - Want `<E extends Comparable<E>>`
  - That is to say: we want a type `E` that implements `Comparable<E>`. That is to say: need that objects of type `E` have a `compareTo` method that takes objects of type `E` as argument

Generic Selection Sort Code

```
1  public static <E extends Comparable<E> > void selectionSort(
       ArrayList<E> data) {
2      int numUnsorted = data.size();
3      int index; // general index
4      int max; // index of largest value
5      while (numUnsorted > 0) {
6          // determine maximum value in array
7          max = 0;
8          for (index = 1; index < numUnsorted; index++) {
9              if (data.get(max).compareTo(data.get(index)) < 0)
                   {
10                 max = index;
11             }
12         }
13         swap(data, max, numUnsorted-1);
14         numUnsorted--;
15     }
16 }
```

Where we are

- Can sort any object so long as it implements `Comparable<E>`

Where we are

- Can sort any object so long as it implements `Comparable<E>`
- What are the downsides of this?

Where we are

- Can sort any object so long as it implements `Comparable<E>`
- What are the downsides of this?
    - What if we want to sort objects that aren't already comparable and we don't want to modify the class?

Where we are

- Can sort any object so long as it implements `Comparable<E>`
- What are the downsides of this?
    - What if we want to sort objects that aren't already comparable and we don't want to modify the class?
    - Can only sort objects one way. (What if we want to sort `Employees` by age rather than name? Would need to rewrite the `Employee` class!)

Sorting with Comparators

- Way to sort objects without changing the class.

Sorting with Comparators

- Way to sort objects without changing the class.
- Let's try to sort employees by name, *without changing* `Employee`. How can we do that?

Sorting with Comparators

- Way to sort objects without changing the class.
- Let's try to sort employees by name, *without changing* `Employee`. How can we do that?
- Idea: use an object whose job it is to compare employees

Sorting with Comparators

- Way to sort objects without changing the class.
- Let's try to sort employees by name, *without changing* `Employee`. How can we do that?
- Idea: use an object whose job it is to compare employees
  - This object will not store any data

Sorting with Comparators

- Way to sort objects without changing the class.
- Let's try to sort employees by name, *without changing* `Employee`. How can we do that?
- Idea: use an object whose job it is to compare employees
  - This object will not store any data
  - Will just have a `int compare(Employee, Employee)` method to compare two employees

Sorting with Comparators

- Way to sort objects without changing the class.
- Let's try to sort employees by name, *without changing* `Employee`. How can we do that?
- Idea: use an object whose job it is to compare employees
  - This object will not store any data
  - Will just have a `int compare(Employee, Employee)` method to compare two employees
- In general: `Comparator<T>` has a `int compare(T, T)` method

Sorting with Comparators

- Way to sort objects without changing the class.
- Let's try to sort employees by name, *without changing* `Employee`. How can we do that?
- Idea: use an object whose job it is to compare employees
  - This object will not store any data
  - Will just have a `int compare(Employee, Employee)` method to compare two employees
- In general: `Comparator<T>` has a `int compare(T, T)` method
  - `compare` returns < 0 if first is smaller; 0 if equal; > 0 if second is smaller

Sorting with Comparators

- Way to sort objects without changing the class.
- Let's try to sort employees by name, *without changing* `Employee`. How can we do that?
- Idea: use an object whose job it is to compare employees
  - This object will not store any data
  - Will just have a `int compare(Employee, Employee)` method to compare two employees
- In general: `Comparator<T>` has a `int compare(T, T)` method
  - `compare` returns < 0 if first is smaller; 0 if equal; > 0 if second is smaller
  - Only job: compare objects of type `T`

Sorting with Comparators

- Way to sort objects without changing the class.
- Let's try to sort employees by name, *without changing* `Employee`. How can we do that?
- Idea: use an object whose job it is to compare employees
  - This object will not store any data
  - Will just have a `int compare(Employee, Employee)` method to compare two employees
- In general: `Comparator<T>` has a `int compare(T, T)` method
  - `compare` returns < 0 if first is smaller; 0 if equal; > 0 if second is smaller
  - Only job: compare objects of type `T`
  - Need to `import java.util.Comparator`

Writing a `Comparator<Employee>`

- This is a class that implements `Comparator<Employee>`

Writing a `Comparator<Employee>`

- This is a class that implements `Comparator<Employee>`
- Goal: sort employees by *age*

Writing a `Comparator<Employee>`

- This is a class that implements `Comparator<Employee>`
- Goal: sort employees by *age*
- Goal 2: sort employees by *name*

Using a Comparator

- Let’s say we want a sort method that uses a comparator, rather than a comparable object

Using a Comparator

- Let’s say we want a sort method that uses a comparator, rather than a comparable object
- (Our sort should work for *any* comparator)

Using a Comparator

- Let’s say we want a sort method that uses a comparator, rather than a comparable object
- (Our sort should work for *any* comparator)
- Idea: take a comparator object as an argument. Then we can use its compare method to compare the objects!

Using a Comparator

- Let’s say we want a sort method that uses a comparator, rather than a comparable object
- (Our sort should work for *any* comparator)
- Idea: take a comparator object as an argument. Then we can use its compare method to compare the objects!
- Let’s write a selection sort method that uses a comparator. And, let’s write two comparators for employees: one that compares by name; another that compares by age.

Comparable vs Comparator

- Comparable:

Comparable vs Comparator

- Comparable:
    - Built in to the class (doesn’t need a new class for comparsion)

Comparable vs Comparator

- Comparable:
  - Built in to the class (doesn’t need a new class for comparsion)
  - Can only compare objects of a class one way

Comparable vs Comparator

- Comparable:
  - Built in to the class (doesn’t need a new class for comparsion)
  - Can only compare objects of a class one way
- Comparator:

Comparable vs Comparator

- Comparable:
  - Built in to the class (doesn’t need a new class for comparsion)
  - Can only compare objects of a class one way
- Comparator:
  - Need to write a new class to compare two objects

Comparable vs Comparator

- Comparable:
  - Built in to the class (doesn’t need a new class for comparsion)
  - Can only compare objects of a class one way
- Comparator:
  - Need to write a new class to compare two objects
  - But: don’t need to modify the original class

Comparable vs Comparator

- Comparable:
  - Built in to the class (doesn’t need a new class for comparsion)
  - Can only compare objects of a class one way
- Comparator:
  - Need to write a new class to compare two objects
  - But: don’t need to modify the original class
  - Can be used to compare objects of a class multiple different ways

Merge Sort

Merge Sort

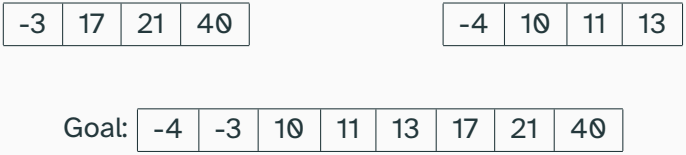
- We can sort faster than $O(n^2)$!
    - Can sort in $O(n \log n)$ time
- Merge sort is a classic way to do that
- Very fast in practice; used as the basis of many state of the art sorting algorithms

Sorting Recursively?

- Can we use recursion to sort?
- Base case?
  - One-element array is always sorted.
- How can we create a smaller subproblem?
  - Could do one element smaller, but that gives us $O(n^2)$ (like in selection sort)
  - Can we divide the size by 2 like in Binary Search?

Merge Sort Hint

- Let's say I have two sorted arrays
- How fast can I sort the concatenation of these arrays?



- Hint: where does the *smallest* element between the two arrays reside?
- Repeatedly take the smaller of the first remaining element in the two arrays. Takes $O(n)$ time

Merge Sort

- If array has size 1, just return

Merge Sort

- If array has size 1, just return
- Split array into two halves

Merge Sort

- If array has size 1, just return
- Split array into two halves
- Sort each half

Merge Sort

- If array has size 1, just return
- Split array into two halves
- Sort each half
    - How?

Merge Sort

- If array has size 1, just return
- Split array into two halves
- Sort each half
  - How?
  - Using Merge Sort!

Merge Sort

- If array has size 1, just return
- Split array into two halves
- Sort each half
  - How?
  - Using Merge Sort!
- Then merge the resulting arrays together in $O(n)$ time by walking through them

Merge Sort PseudoCode

```
1  mergeSortRecursive(list):
2      if list has length 1
3          return
4
5      list firstHalf =  first half of list
6      mergeSortRecursive(firstHalf);
7
8      list secondHalf = second half of list
9      mergeSortRecursive(secondHalf);
10
11     merge firstHalf and secondHalf into list
```

Merge PseudoCode

```
merge(list, firstHalf, secondHalf):
    firstIndex = 0
    secondIndex = 0
    for each slot in list:
        if firstHalf[firstIndex] < secondHalf[secondIndex]:
            list[slot] = firstHalf[firstIndex]
            firstIndex++
        else:
            list[slot] = secondHalf[secondIndex]
            secondIndex++
```

This pseudocode has an out-of-bounds error. Let's fix it!

Merge PseudoCode: Option 1

```
merge(list, firstHalf, secondHalf):
    firstIndex = 0
    secondIndex = 0
    for each slot in list:
        if secondIndex >= secondHalf.length or (firstIndex <
            firstHalf.length and firstHalf[firstIndex] <
           secondHalf[secondIndex]):
            list[slot] = firstHalf[firstIndex]
            firstIndex++
        else:
            list[slot] = secondHalf[secondIndex]
            secondIndex++
```

The extra checks, with short-circuit evaluation, mean that we no longer have out-of-bounds issues.

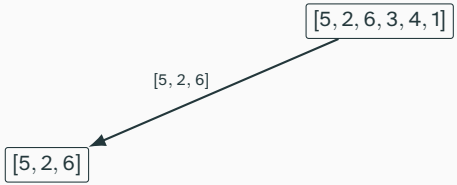
Merge PseudoCode: Option 2

```
merge(list, firstHalf, secondHalf):
    firstIndex = 0
    secondIndex = 0
    for each slot in list:
        if secondIndex >= secondHalf.length:
            list[slot] = firstHalf[firstIndex]
            firstIndex++
        else if firstIndex >= firstHalf.length:
            list[slot] = secondHalf[secondIndex]
            secondIndex++
        else if firstHalf[firstIndex] < secondHalf[
            secondIndex]):
            list[slot] = firstHalf[firstIndex]
            firstIndex++
        else:
            list[slot] = secondHalf[secondIndex]
            secondIndex++
```

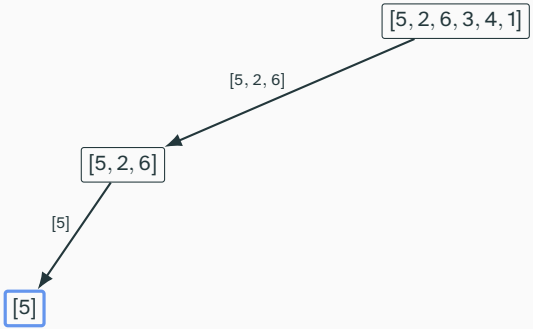
This strategy has a little redundancy, but it's easier to read how the out-of-bounds

Merge Sort

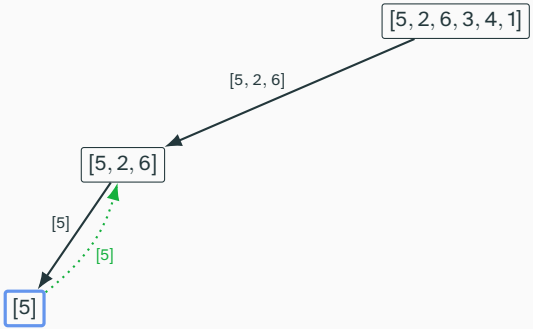
Merge Sort



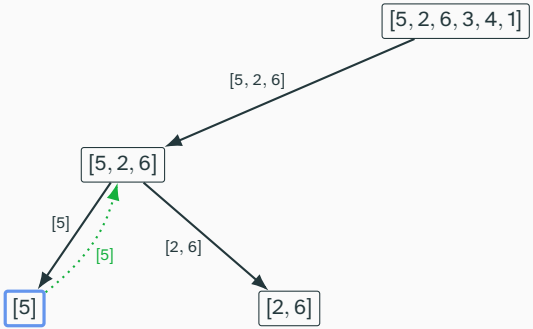
Merge Sort



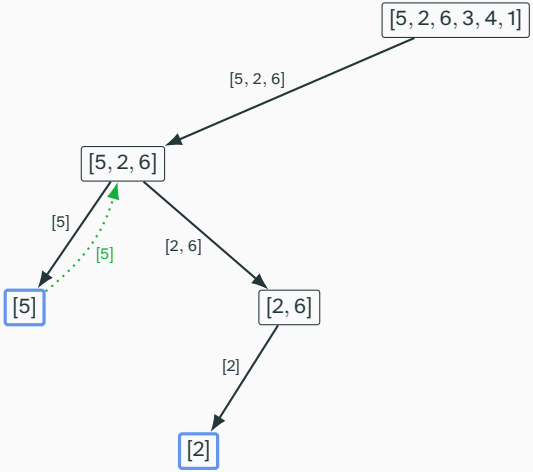
Merge Sort



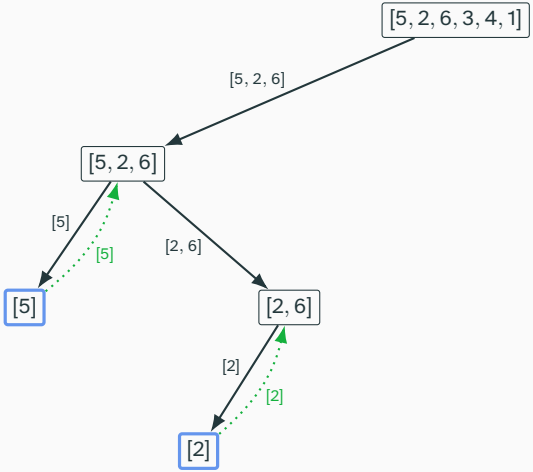
Merge Sort



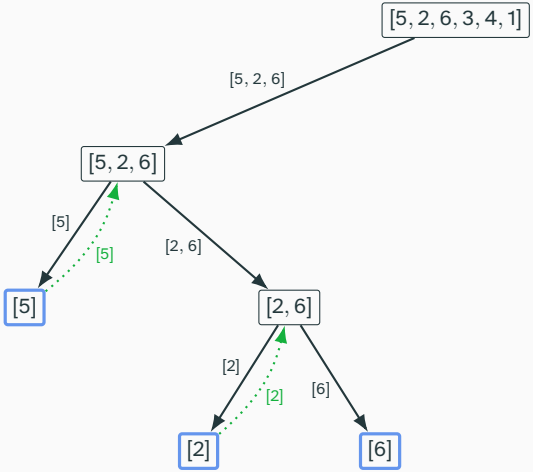
Merge Sort



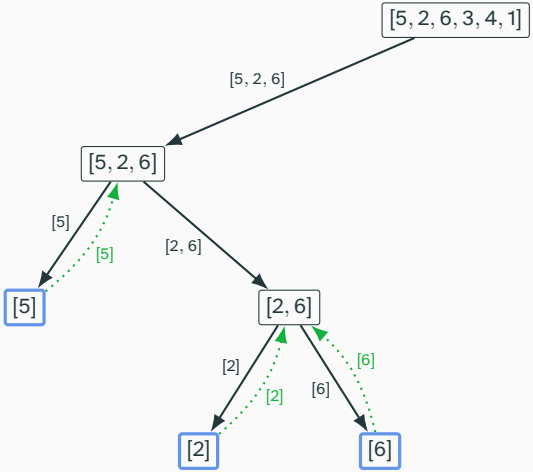
Merge Sort



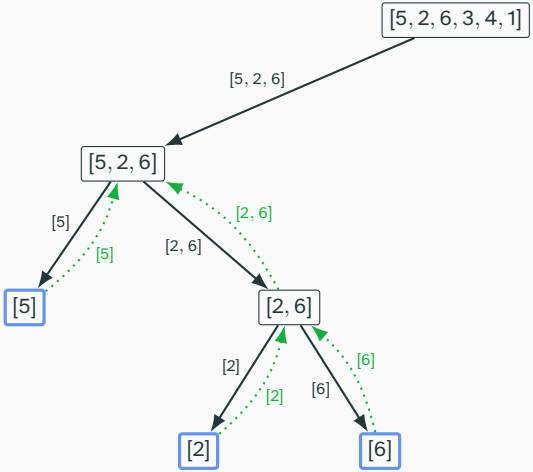
Merge Sort



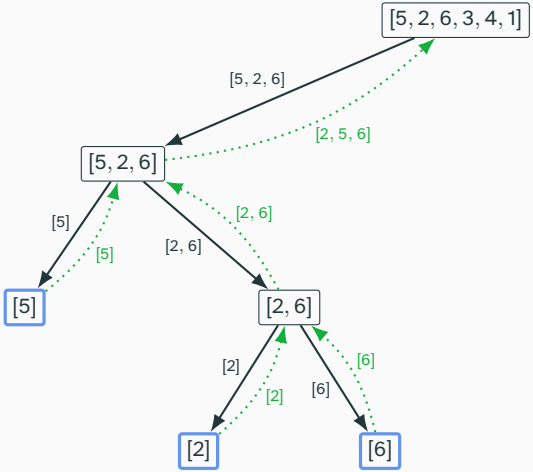
Merge Sort



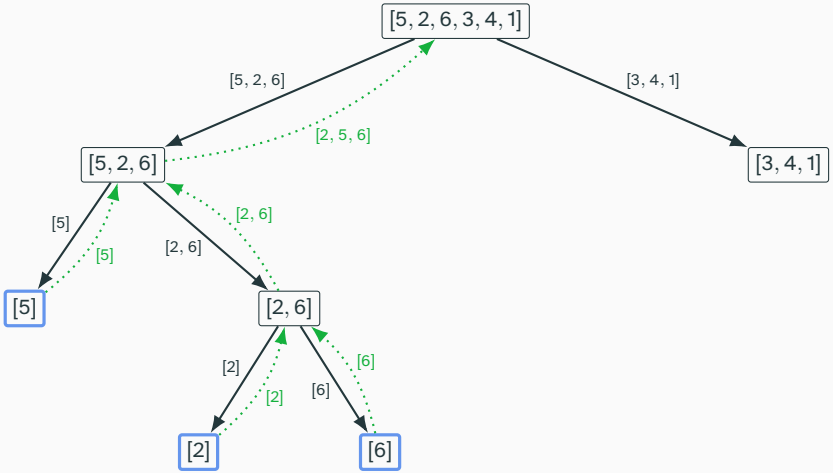
Merge Sort



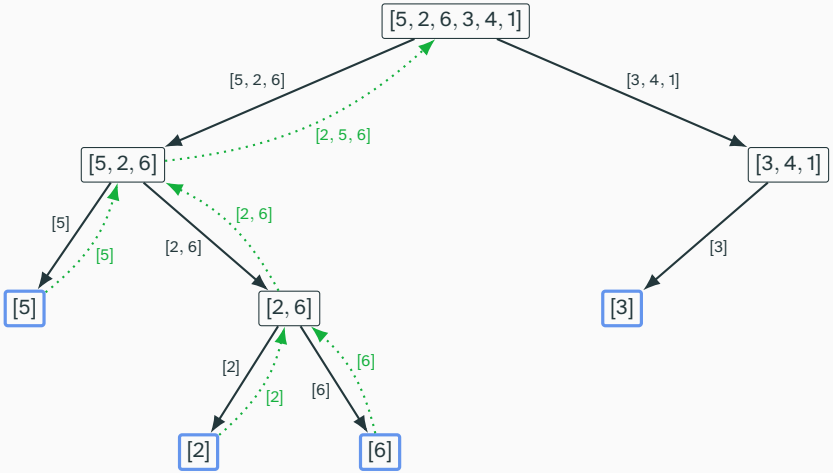
Merge Sort



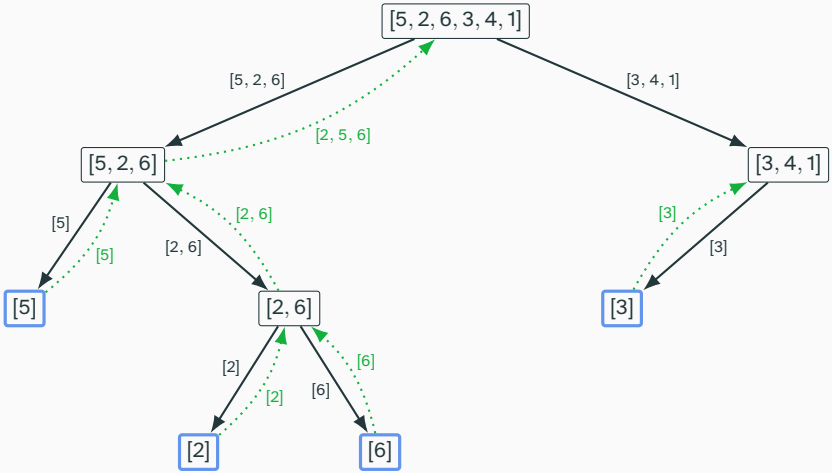
Merge Sort



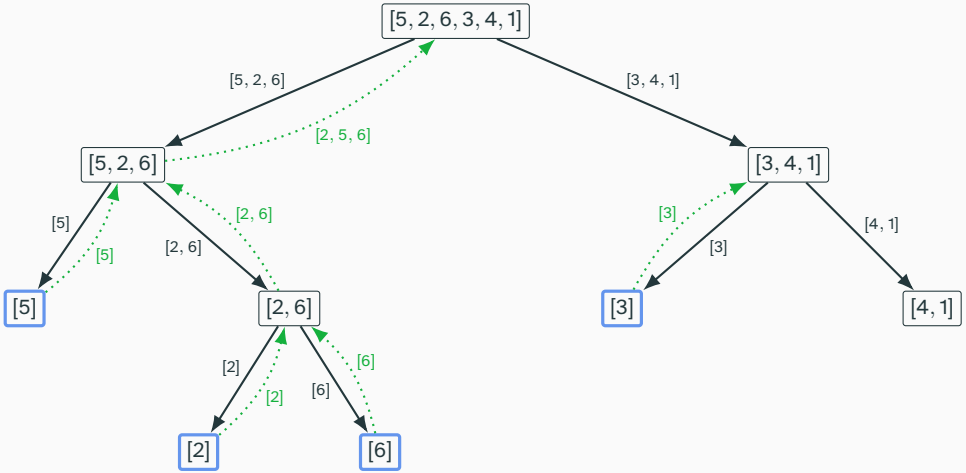
Merge Sort



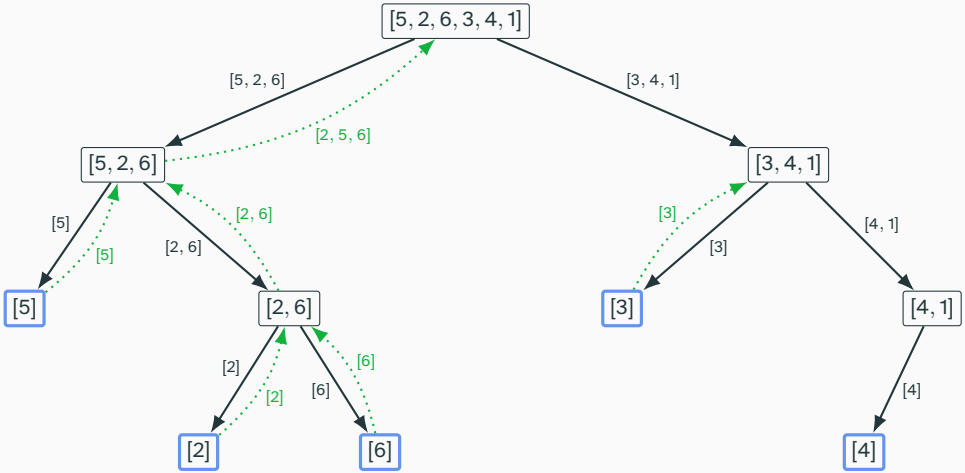
Merge Sort



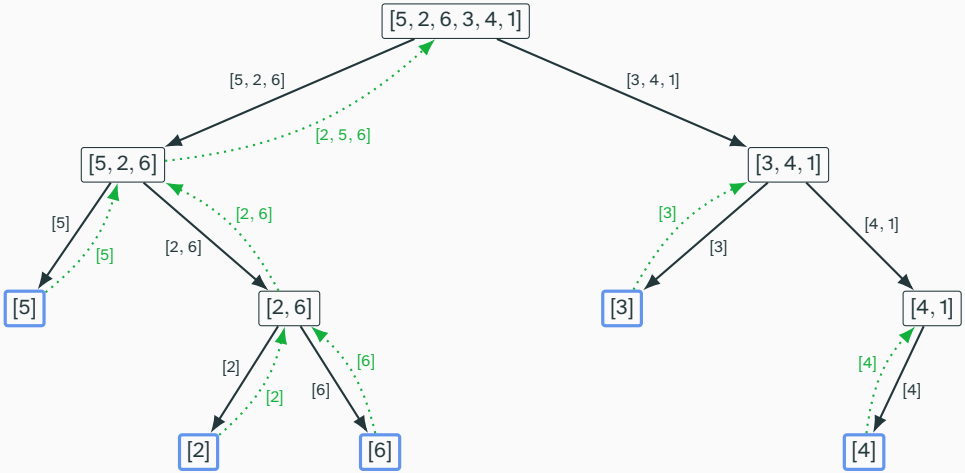
Merge Sort



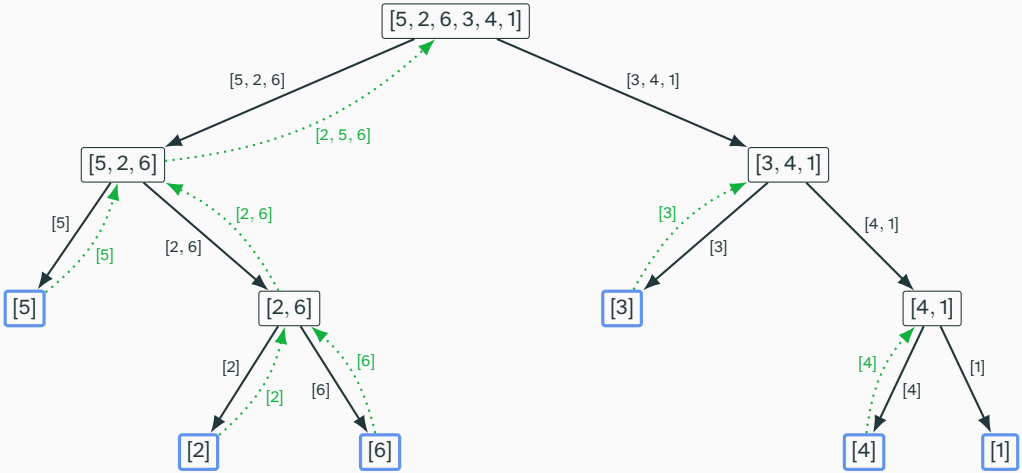
Merge Sort



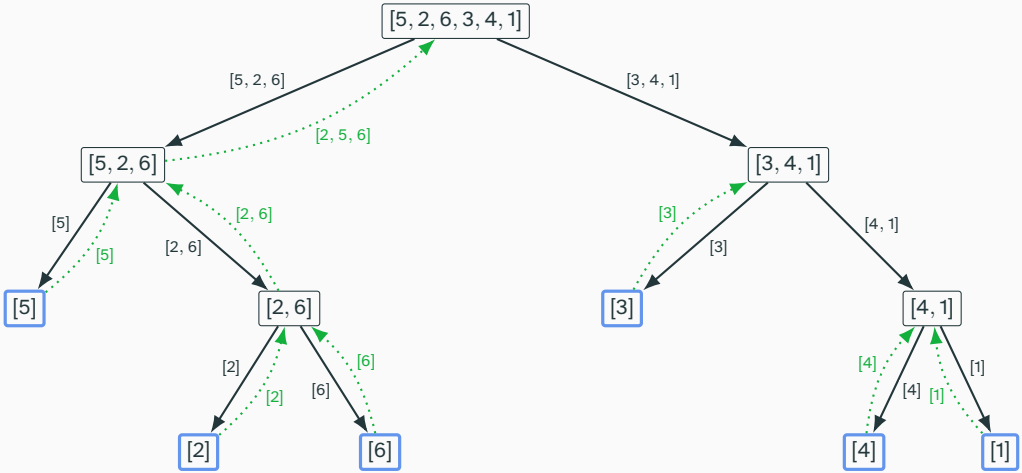
Merge Sort



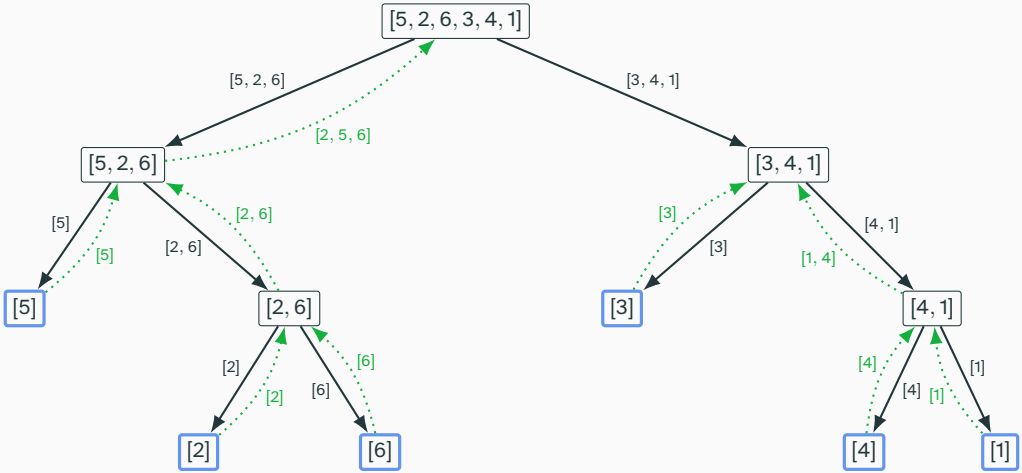
Merge Sort



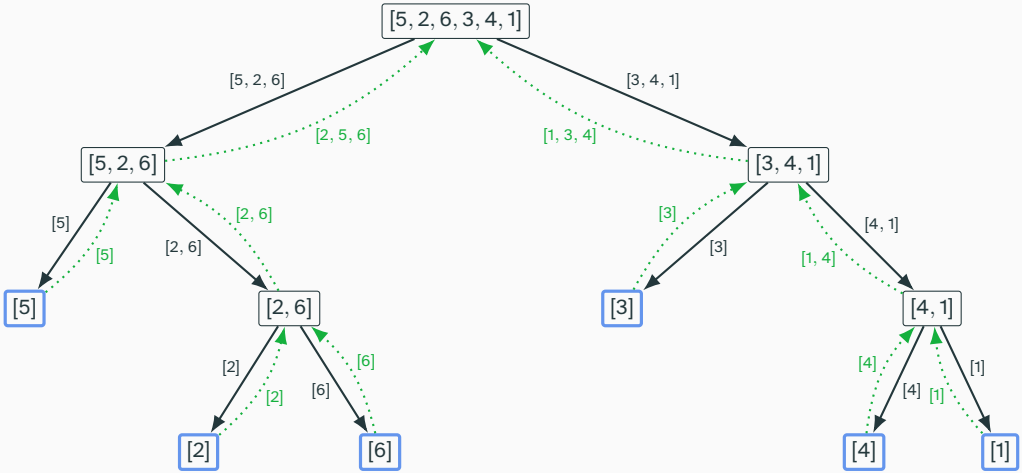
Merge Sort



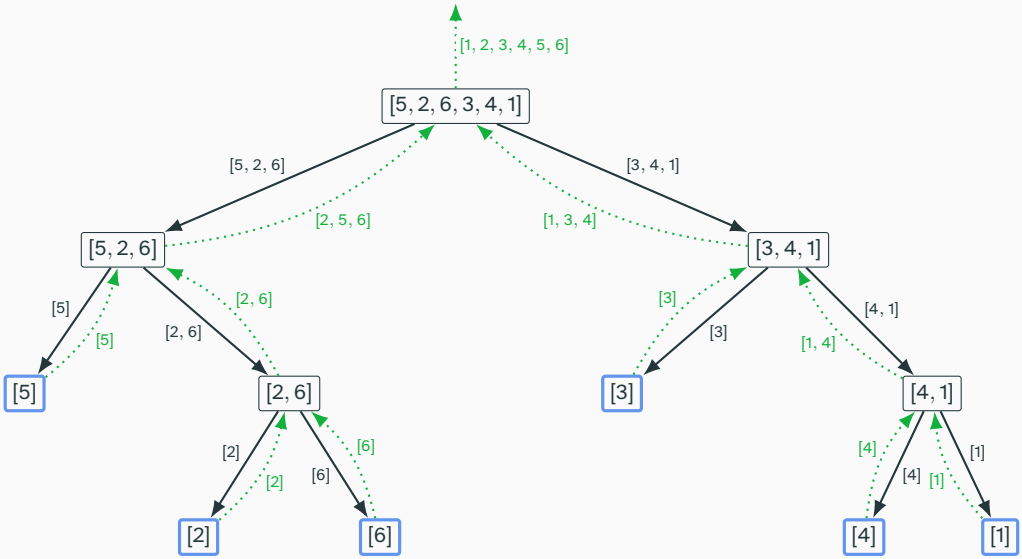
Merge Sort


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

Merge Sort


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

Merge Sort


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

Merge Sort Discussion

- *All work* is done during copying: either creating `firstHalf` and `secondHalf`, or merging the two halves into `list`

Merge Sort Discussion

- *All work* is done during copying: either creating `firstHalf` and `secondHalf`, or merging the two halves into `list`

- “Splitting” part of the algorithm just recurses down to arrays of size 1

Merge Sort Running Time

```
1  merge(list, firstHalf, secondHalf):
2      firstHalfIndex = 0
3      secondHalfIndex = 0
4      for each slot in list:
5          if secondIndex >= secondHalf.length or (firstIndex <
               firstHalf.length and firstHalf[firstIndex] <
              secondHalf[secondIndex]):
6              list[slot] = firstHalf[firstIndex]
7              firstIndex++
8          else:
9              list[slot] = secondHalf[secondIndex]
10             secondIndex++
```

How much time does a merge take for a `list` of length n?

Merge Sort Running Time

```
1  merge(list, firstHalf, secondHalf):
2      firstHalfIndex = 0
3      secondHalfIndex = 0
4      for each slot in list:
5          if secondIndex >= secondHalf.length or (firstIndex <
               firstHalf.length and firstHalf[firstIndex] <
              secondHalf[secondIndex]):
6              list[slot] = firstHalf[firstIndex]
7              firstIndex++
8          else:
9              list[slot] = secondHalf[secondIndex]
10             secondIndex++
```

How much time does a merge take for a `list` of length n? $O(n)$

Merge Sort Running Time

```
1  merge(list, firstHalf, secondHalf):
2      firstHalfIndex = 0
3      secondHalfIndex = 0
4      for each slot in list:
5           if firstHalf[firstHalfIndex] < secondHalf[
               secondHalfIndex]:
6                list[slot] = firstHalf[firstHalfIndex]
7                firstHalfIndex++
8           else:
9                list[slot] = secondHalf[secondHalfIndex]
10               secondHalfIndex++
```

How much time do we take other than the recursive calls?

Merge Sort Running Time

```
1  merge(list, firstHalf, secondHalf):
2      firstHalfIndex = 0
3      secondHalfIndex = 0
4      for each slot in list:
5           if firstHalf[firstHalfIndex] < secondHalf[
               secondHalfIndex]:
6                list[slot] = firstHalf[firstHalfIndex]
7                firstHalfIndex++
8           else:
9                list[slot] = secondHalf[secondHalfIndex]
10               secondHalfIndex++
```

How much time do we take other than the recursive calls?

- $O(n)$ time outside of the merge.

Merge Sort Running Time

```
1  merge(list, firstHalf, secondHalf):
2      firstHalfIndex = 0
3      secondHalfIndex = 0
4      for each slot in list:
5           if firstHalf[firstHalfIndex] < secondHalf[
               secondHalfIndex]:
6                list[slot] = firstHalf[firstHalfIndex]
7                firstHalfIndex++
8           else:
9                list[slot] = secondHalf[secondHalfIndex]
10               secondHalfIndex++
```

How much time do we take other than the recursive calls?

- $O(n)$ time outside of the merge.
- $O(n)$ time for the merge.

Merge Sort Running Time

```
1  merge(list, firstHalf, secondHalf):
2      firstHalfIndex = 0
3      secondHalfIndex = 0
4      for each slot in list:
5           if firstHalf[firstHalfIndex] < secondHalf[
               secondHalfIndex]:
6                list[slot] = firstHalf[firstHalfIndex]
7                firstHalfIndex++
8           else:
9                list[slot] = secondHalf[secondHalfIndex]
10               secondHalfIndex++
```

How much time do we take other than the recursive calls?

- $O(n)$ time outside of the merge.
- $O(n)$ time for the merge.
- Total is $O(n) + O(n) = O(2n) = O(n)$

Finishing Merge Sort Analysis

- Each time we call merge sort, it takes $O(n)$ time.

Finishing Merge Sort Analysis

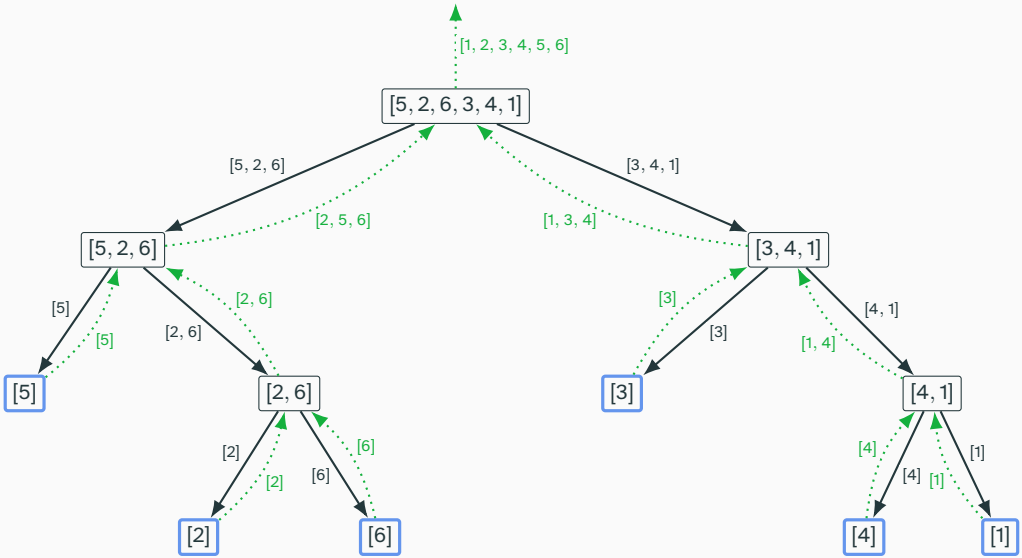
- Each time we call merge sort, it takes $O(n)$ time.

- Long story short: $\log n$ recursive calls until the base case

Finishing Merge Sort Analysis

- Each time we call merge sort, it takes $O(n)$ time.
- Long story short: $\log n$ recursive calls until the base case
- Total: $O(n \log n)$ time

Merge Sort Analysis



Insertion Sort

Insertion Sort

- Similar to Selection Sort

Insertion Sort

- Similar to Selection Sort
- Still $O(n^2)$, but significantly more efficient in practice (we'll come back to this)

Insertion Sort

- Similar to Selection Sort
- Still $O(n^2)$, but significantly more efficient in practice (we'll come back to this)
- This time we'll start with why it works, and derive the algorithm

Insertion Sort

- A different approach to sorting

Insertion Sort

- A different approach to sorting
- After the *k*th loop, the first *k* items in the array are sorted
  - The first *k* items may not be the smallest—but they are in sorted order

Insertion Sort

- A different approach to sorting
- After the kth loop, the first k items in the array are sorted
  - The first k items may not be the smallest—but they are in sorted order
- How can we guarantee this for $k = 1$?

Insertion Sort

- A different approach to sorting
- After the kth loop, the first k items in the array are sorted
  - The first k items may not be the smallest—but they are in sorted order
- How can we guarantee this for $k = 1$?
  - Don’t need to do anything

Insertion Sort

- A different approach to sorting
- After the kth loop, the first k items in the array are sorted
  - The first k items may not be the smallest—but they are in sorted order
- How can we guarantee this for $k = 1$?
  - Don't need to do anything
- Let's say it works for k. What does the $k + 1$st loop need to accomplish to maintain the invariant?

-3	10	21	40	17	13	11	-4

Insertion Sort

- A different approach to sorting
- After the kth loop, the first k items in the array are sorted
  - The first k items may not be the smallest—but they are in sorted order
- How can we guarantee this for $k = 1$?
  - Don't need to do anything
- Let's say it works for k. What does the $k + 1$st loop need to accomplish to maintain the invariant?

  - Needs to insert the $k + 1$st item among the first k items in sorted order.

A Beautiful Way to Accomplish This

- Want to take the new item and move it into sorted position

A Beautiful Way to Accomplish This

- Want to take the new item and move it into sorted position
- Idea: need to move it down until the previous element is smaller

A Beautiful Way to Accomplish This

- Want to take the new item and move it into sorted position
- Idea: need to move it down until the previous element is smaller
- Inner loop: store element we are trying to insert. Shift elements down while it is smaller.

Insertion Sort Code

```java
public static void insertionSort(int data[]) {
    int numSorted = 1; // number of values in place
    int index; // general index
    while (numSorted < data.length) {
        int temp = data[numSorted]; // first unsorted value
        for (index = numSorted; index > 0; index--) {
            if (temp < data[index-1]) {
                data[index] = data[index-1];
            } else {
                break;
            }
        }
        data[index] = temp; // reinsert value
        numSorted++;
    }
}
```

Insertion Sort Code

```
1  public static void insertionSort(int data[]) {
2      int numSorted = 1; // number of values in place
3      int index; // general index
4      while (numSorted < data.length) {
5          int temp = data[numSorted]; // first unsorted value
6          fo                                              {
7
8
9              } else {
10                 break;
11             }
12         }
13         data[index] = temp; // reinsert value
14         numSorted++;
15     }
16 }
```

Can we get rid of the `break` command in this code?

Insertion Sort Code # 2

```
1  public static void insertionSort(int data[]) {
2      int numSorted = 1; // number of values in place
3      while (numSorted < data.length) {
4          int temp = data[numSorted]; // first unsorted value
5          int index = numSorted;
6          while(index > 0 && temp < data[index - 1]) {
7              data[index] = data[index-1];
8              index--;
9          }
10         data[index] = temp; // reinsert value
11         numSorted++;
12     }
13 }
```

Tradeoff with Selection Sort

- No swap method needed

Tradeoff with Selection Sort

- No swap method needed
- Code is a little shorter

Tradeoff with Selection Sort

- No swap method needed
- Code is a little shorter
- Efficiency?
  - Both take n iterations of the outer loop. What about the inner loop?
  - Selection sort *always* iterates through $n - i$ elements on the ith iteration
  - Insertion sort may stop early! Can lead to better performance in practice (and is never worse)

Tradeoff with Selection Sort

- No swap method needed
- Code is a little shorter
- Efficiency?
  - Both take n iterations of the outer loop. What about the inner loop?
  - Selection sort *always* iterates through $n - i$ elements on the ith iteration
  - Insertion sort may stop early! Can lead to better performance in practice (and is never worse)
- To be clear: both are still $O(n^2)$ in terms of worst-case performance. Insertion sort just has better constants, and better best-case performance

Tradeoff with Merge Sort (Just for fun)

- Merge Sort is better than Insertion Sort for large n

Tradeoff with Merge Sort (Just for fun)

- Merge Sort is better than Insertion Sort for large n
- Less clear for small n!

Tradeoff with Merge Sort (Just for fun)

- Merge Sort is better than Insertion Sort for large n
- Less clear for small n!
    - Insertion sort: $n^2/2$ worst case; $n^2/4$ “on average”

Tradeoff with Merge Sort (Just for fun)

- Merge Sort is better than Insertion Sort for large n
- Less clear for small n!
  - Insertion sort: $n^2/2$ worst case; $n^2/4$ “on average”
  - Merge sort: $2n \log_2 n$

Tradeoff with Merge Sort (Just for fun)

- Merge Sort is better than Insertion Sort for large n
- Less clear for small n!
  - Insertion sort: $n^2/2$ worst case; $n^2/4$ “on average”
  - Merge sort: $2n \log_2 n$
  - Tradeoff point n somewhere in the range 32–256

Tradeoff with Merge Sort (Just for fun)

- Merge Sort is better than Insertion Sort for large n
- Less clear for small n!
  - Insertion sort: $n^2/2$ worst case; $n^2/4$ “on average”
  - Merge sort: $2n \log_2 n$
  - Tradeoff point n somewhere in the range 32–256
- Bears out in practice: insertion sort is better on small lists

Tradeoff with Merge Sort (Just for fun)

- Merge Sort is better than Insertion Sort for large n
- Less clear for small n!
  - Insertion sort: $n^2/2$ worst case; $n^2/4$ “on average”
  - Merge sort: $2n \log_2 n$
  - Tradeoff point n somewhere in the range 32–256
- Bears out in practice: insertion sort is better on small lists
- A well-implemented Merge Sort switches to Insertion Sort as a base case