CSE 503

Introduction to Computer Science for Non-Majors

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Day 32

MergeSort and Recursion

Announcements

- Autolab for Lab #5 is not up yet but I will try to have it up by tonight

Recap

- Two different search algorithms: **`LinearSearch`** and **`BinarySearch`**
    - **`LinearSearch`** on list of size ***N*** requires ***N*** comparisons in the worst case
    - **`BinarySearch`** on a **sorted** list of size ***N*** requires **log(*N*)** comparisons in the worst case
    - As we try larger and larger inputs, ***N*** grows much faster than **log(*N*)**
- **`SelectionSort`** is the first sorting algorithm we've seen
    - Select the smallest item from input and add it to the end of output
    - Requires (roughly) N^2 steps to sort a list of size ***N***

Recursion

Factorial

factorial(n) = n * (n-1) * (n-2) * … * 2 * 1

Factorial

$$\text{factorial}(n) = n * \underbrace{(n-1) * (n-2) * \ldots * 2 * 1}_{\text{factorial}(n-1)}$$

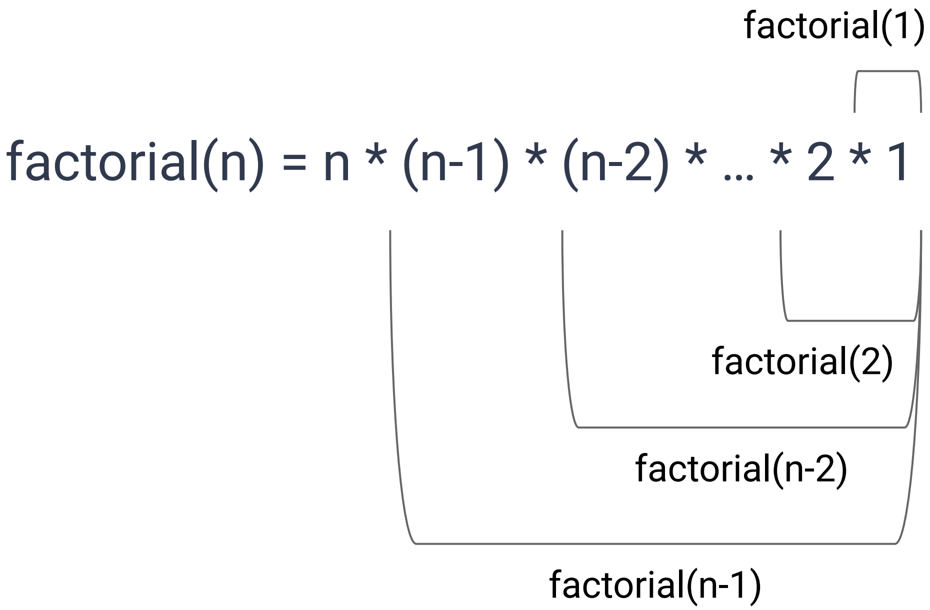
Factorial

$$\text{factorial}(n) = n * \underbrace{(n-1) * \underbrace{(n-2) * \ldots * 2 * 1}_{\text{factorial}(n-2)}}_{\text{factorial}(n-1)}$$

Factorial

$$\text{factorial}(n) = n * \underbrace{(n-1) * \underbrace{(n-2) * \ldots * \underbrace{2 * 1}_{\text{factorial}(2)}}_{\text{factorial}(n-2)}}_{\text{factorial}(n-1)}$$

Factorial



factorial(1)

factorial(n) = n * (n-1) * (n-2) * … * 2 * 1

factorial(2)

factorial(n-2)

factorial(n-1)

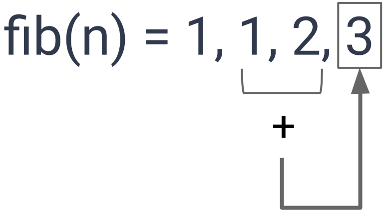
Fibonacci

fib(n) = 1, 1

Fibonacci

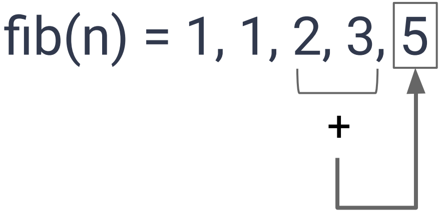
$$\mathrm{fib}(n) = \underbrace{1, 1}_{+}, \boxed{2}$$

Fibonacci



$$\text{fib}(n) = 1, \underbrace{1, 2,}_{+} \boxed{3}$$

Fibonacci



fib(n) = 1, 1, 2, 3, 5

+

Fibonacci

fib(n) = 1, 1, 2, 3, 5, 8, 13, 21, 34, …

Fibonacci

fib(n) = 1, 1, 2, 3, 5, 8, 13, 21, 34, …

fib(n) = fib(n-1) + fib(n-2)

Towers of Hanoi

Live Demo!

Recursion

Recursion (in CS) is the when we define a function using itself

Recursion

Recursion (in CS) is the when we define a function using itself

There is a **base case**, where the result can be directly computed

- ie: factorial(1) = 1, fib(1) = 1, fib(2) = 1, the smallest nesting doll, Towers of Hanoi with one disc

Recursion

Recursion (in CS) is the when we define a function using itself

There is a **base case**, where the result can be directly computed

- ie: factorial(1) = 1, fib(1) = 1, fib(2) = 1, the smallest nesting doll, Towers of Hanoi with one disc

There is a **recursive case**, where the result is computed by running the function on a smaller input/problem

- ie: factorial(10) = 10 * factorial(9), fib(26) = fib(25) + fib(24), etc

MergeSort

MergeSort is a recursive sorting algorithm.

It is an example of a **Divide and Conquer** approach to solving a problem:

MergeSort

MergeSort is a recursive sorting algorithm.

It is an example of a **Divide and Conquer** approach to solving a problem:

1. **Divide** the problem into smaller pieces

MergeSort

MergeSort is a recursive sorting algorithm.

It is an example of a **Divide and Conquer** approach to solving a problem:

1. **Divide** the problem into smaller pieces
2. **Conquer (solve)** the smaller problems

MergeSort

MergeSort is a recursive sorting algorithm.

It is an example of a **Divide and Conquer** approach to solving a problem:

1. **Divide** the problem into smaller pieces
2. **Conquer (solve)** the smaller problems
3. **Combine** the smaller solutions into a larger solution

MergeSort

Input: An array with elements in an unknown order.

Output: An array with elements in sorted order.

MergeSort - Questions

Divide (break the list into smaller lists)
What's the smallest list we could try to sort?

MergeSort - Questions

Divide (break the list into smaller lists)

What's the smallest list we could try to sort? ***N* = 1**

MergeSort - Questions

Divide (break the list into smaller lists)
What's the smallest list we could try to sort? ***N* = 1**

Conquer (sort the smaller lists)
How do we sort it?

MergeSort - Questions

Divide (break the list into smaller lists)
What's the smallest list we could try to sort? ***N*** **= 1**

Conquer (sort the smaller lists)
How do we sort it? If ***N*** **= 1**, it's already sorted!!!

MergeSort - Questions

Divide (break the list into smaller lists)
What's the smallest list we could try to sort? ***N* = 1**

Conquer (sort the smaller lists)
How do we sort it? If ***N* = 1**, it's already sorted!!!

Combine (combine the sorted lists into a bigger sorted list)
How can we do this, and how long does it take?

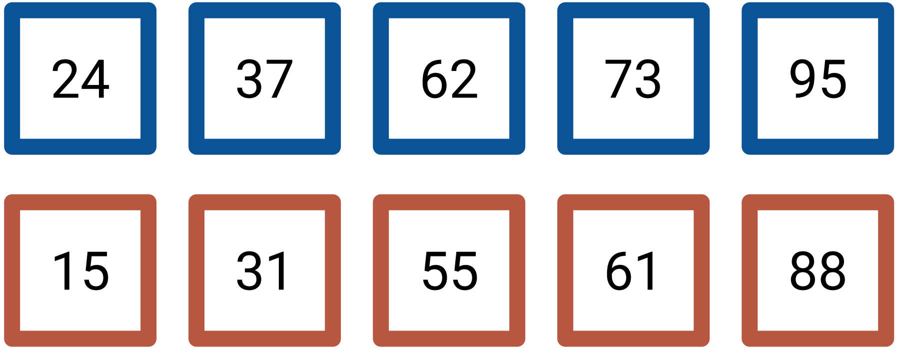
MergeSort - Questions

Divide (break the list into smaller lists)
What's the smallest list we could try to sort? ***N* = 1**

Conquer (sort the smaller lists)
How do we sort it? If ***N* = 1**, it's already sorted!!!

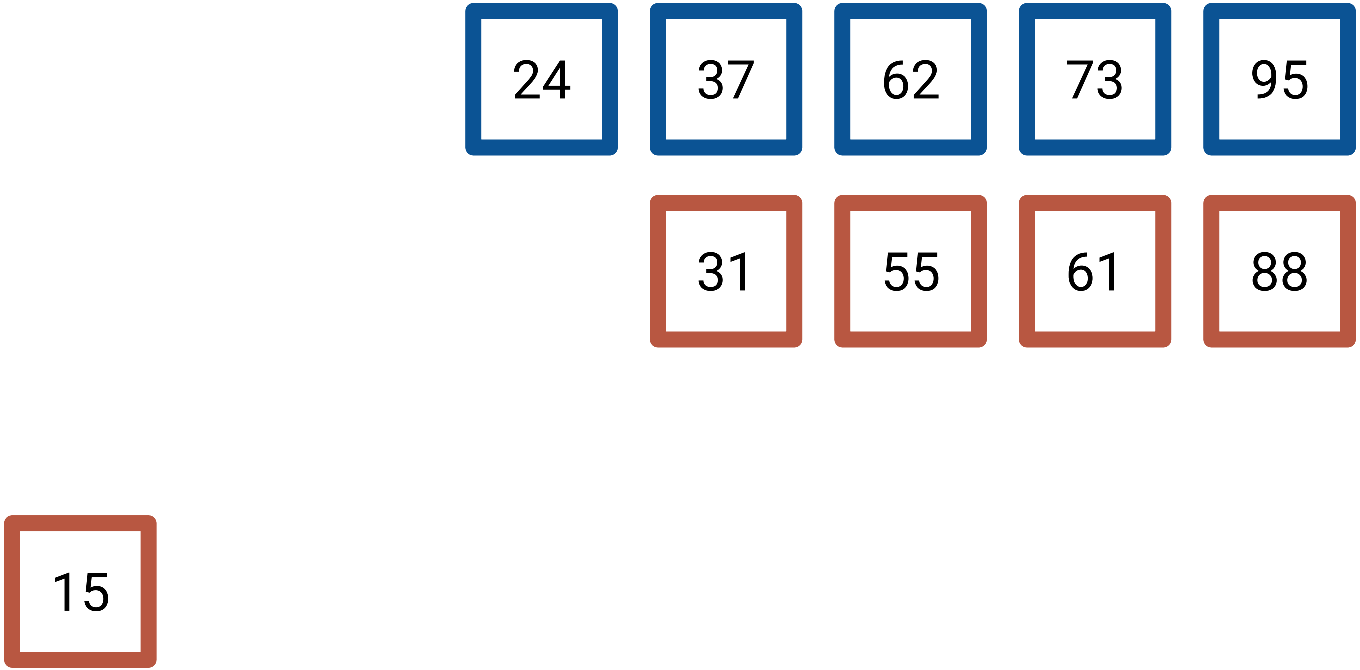
Combine (combine the sorted lists into a bigger sorted list)
How can we do this, and how long does it take? Merge…

How do we Merge Two Sorted Arrays?



24	37	62	73	95
15	31	55	61	88

How do we Merge Two Sorted Arrays?

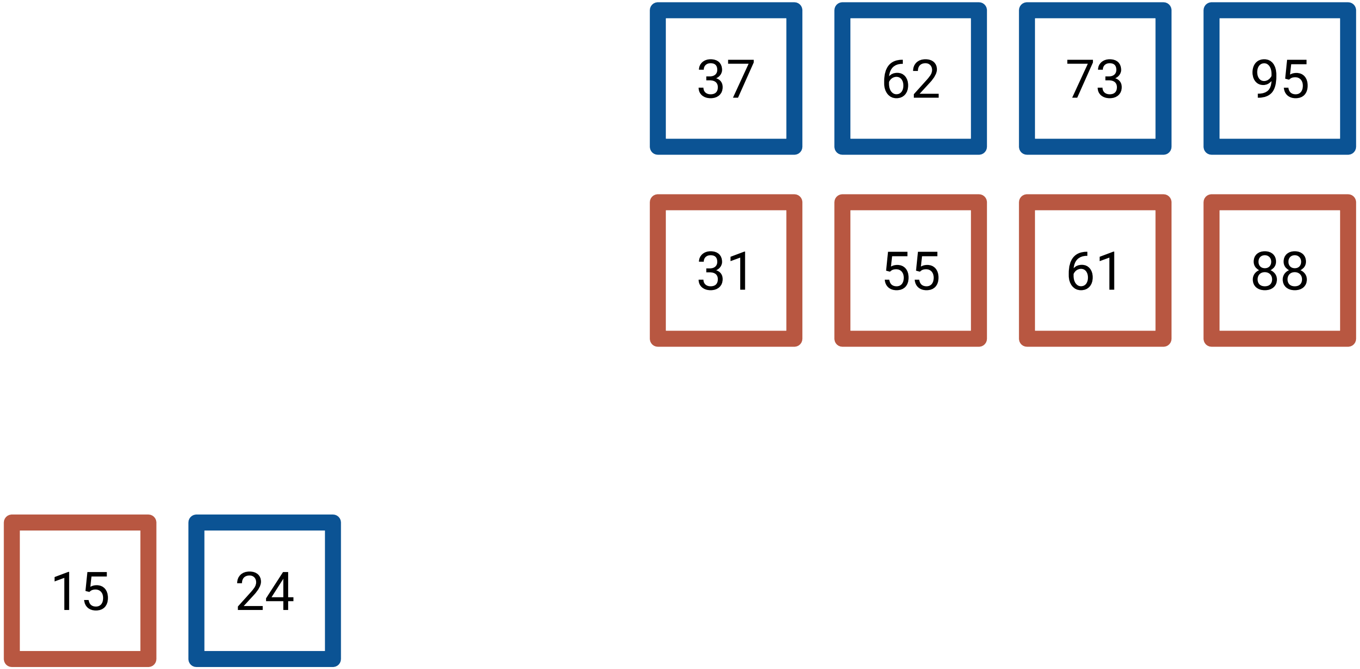


24 37 62 73 95

31 55 61 88

15

How do we Merge Two Sorted Arrays?

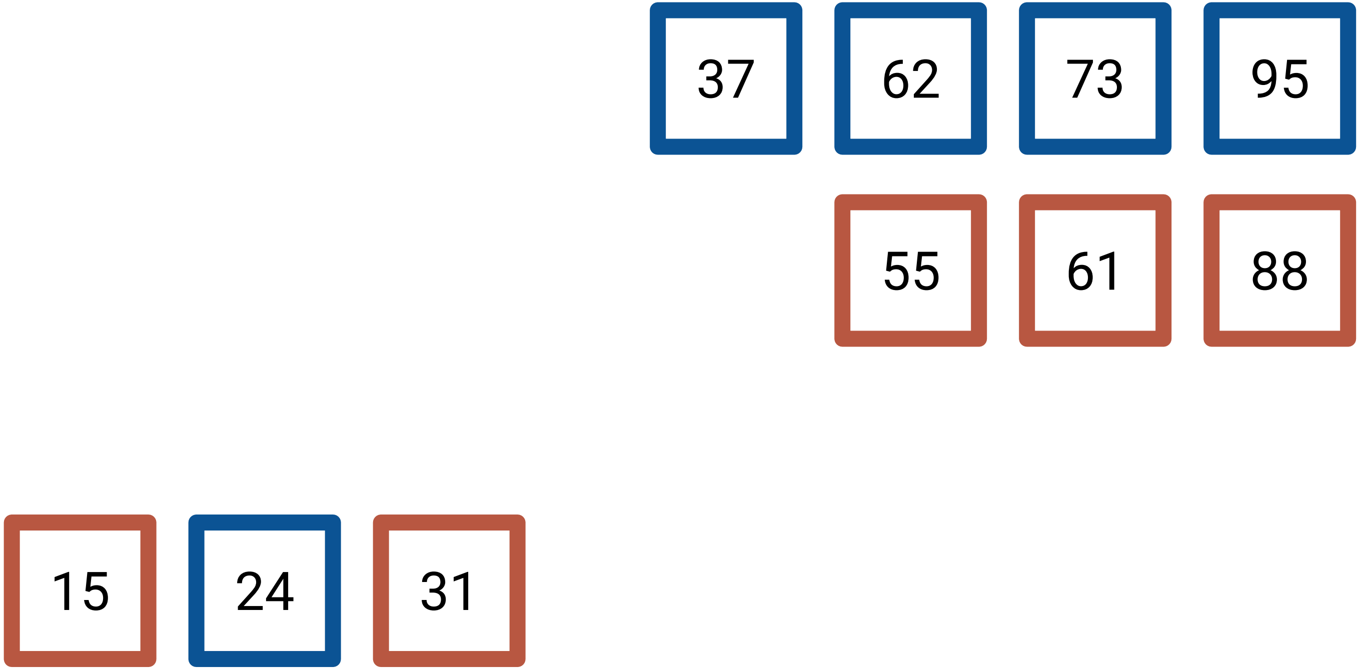


37 62 73 95

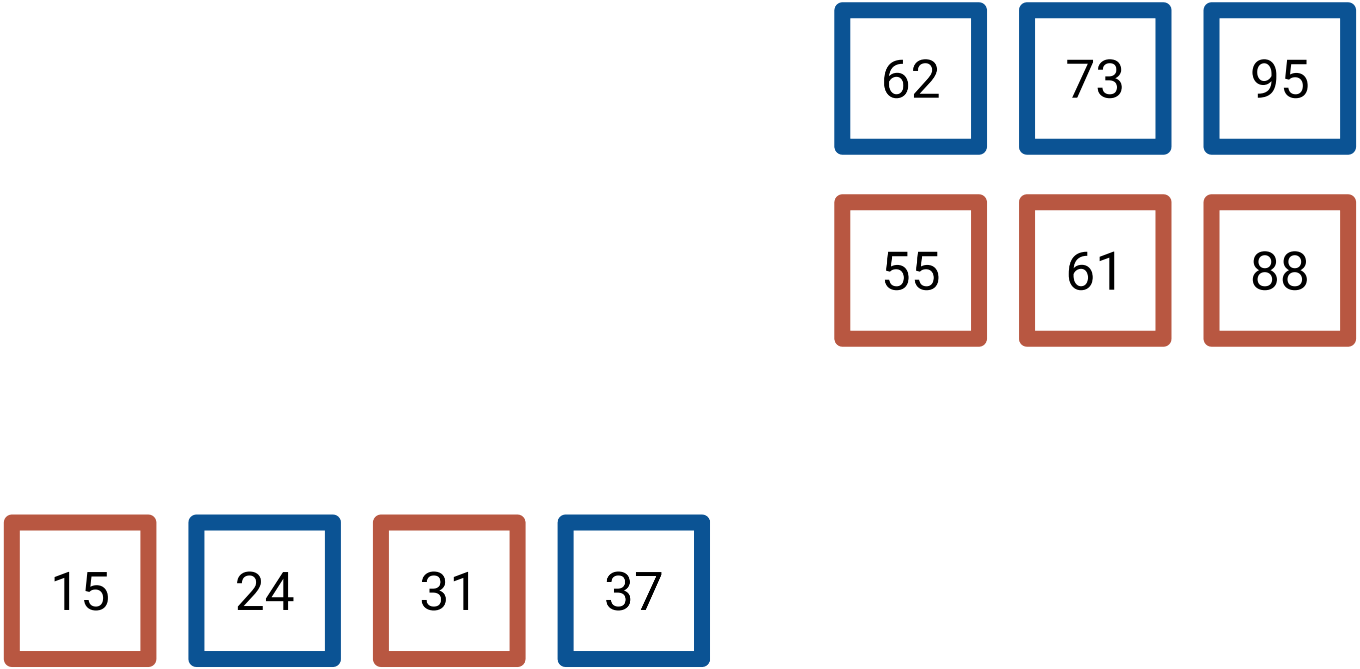
31 55 61 88

15 24

How do we Merge Two Sorted Arrays?



How do we Merge Two Sorted Arrays?

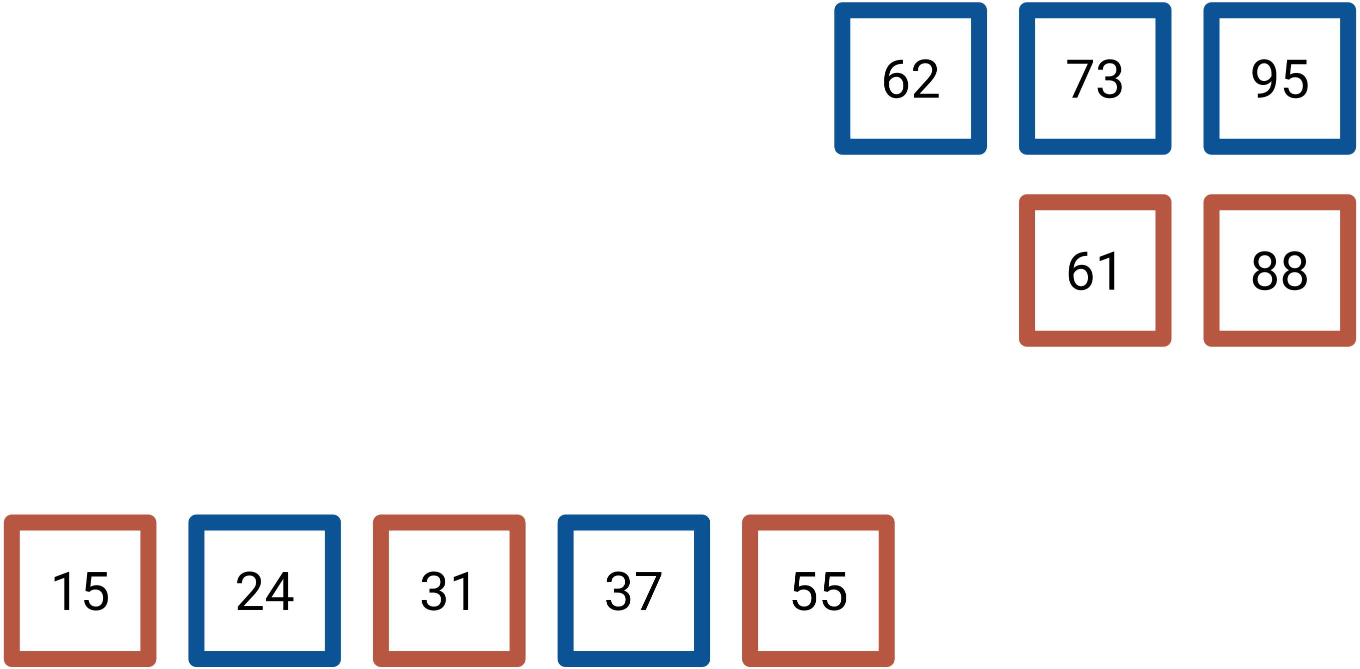


62 73 95

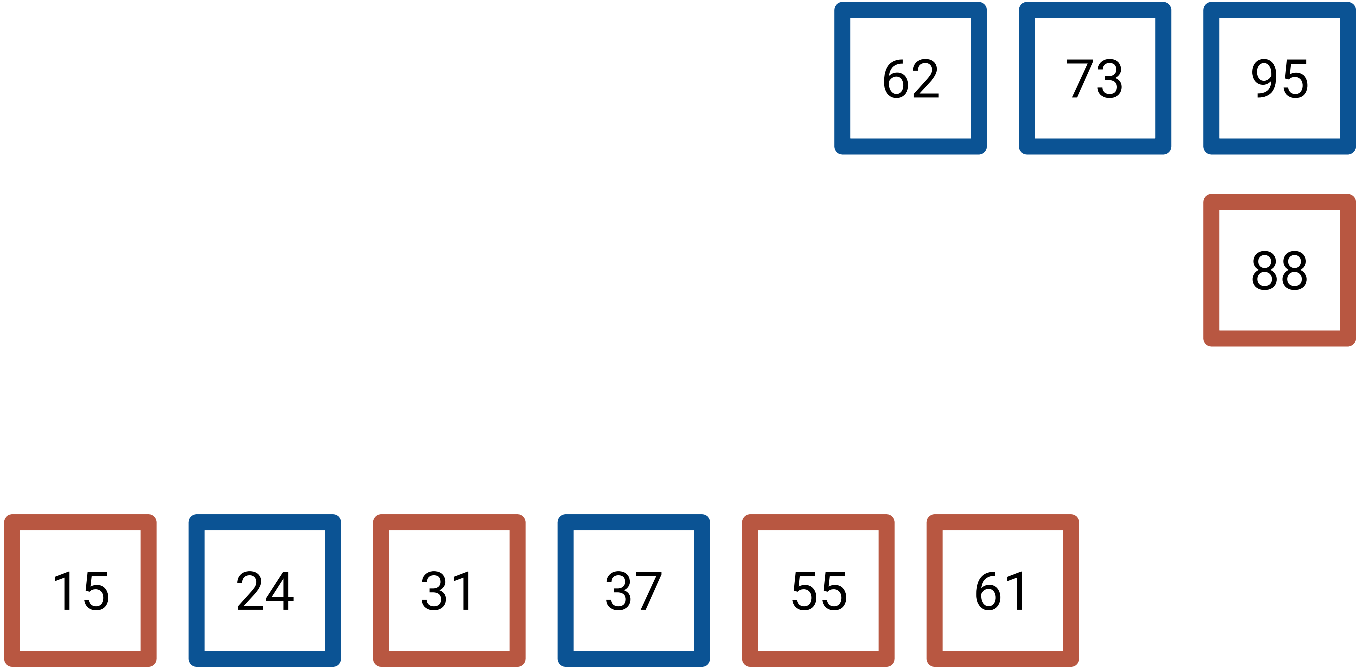
55 61 88

15 24 31 37

How do we Merge Two Sorted Arrays?



How do we Merge Two Sorted Arrays?

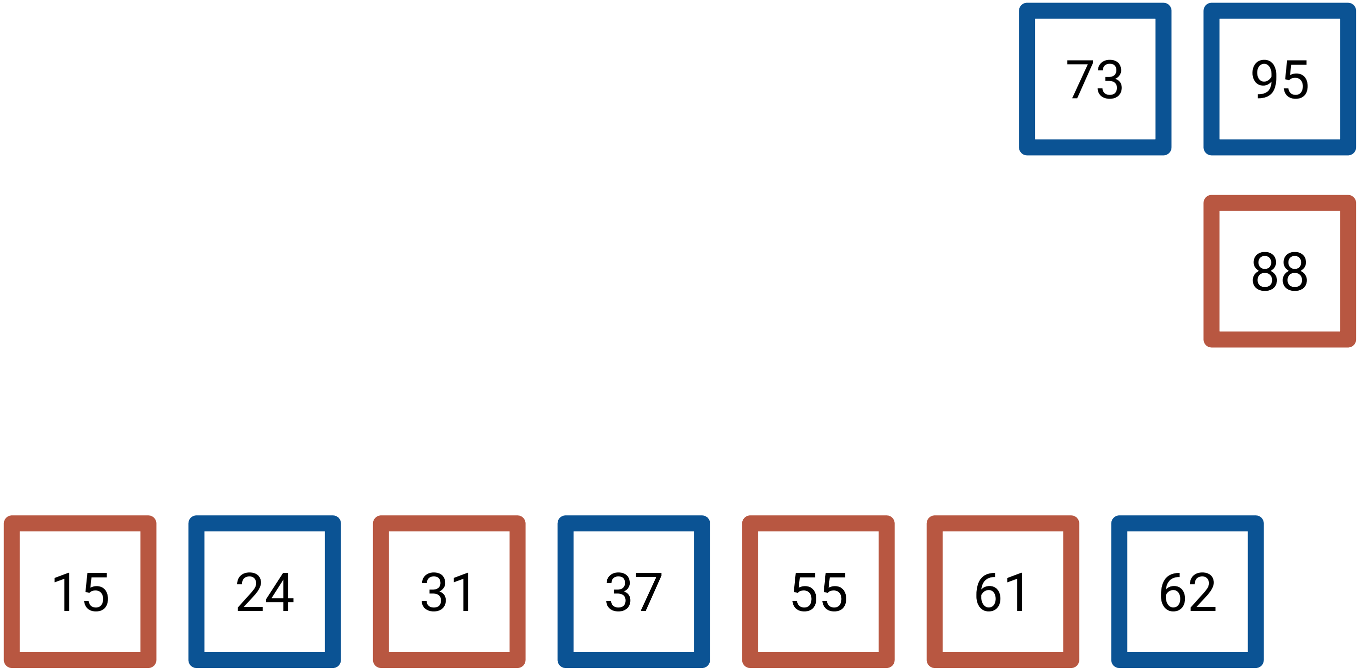


62 73 95

88

15 24 31 37 55 61

How do we Merge Two Sorted Arrays?

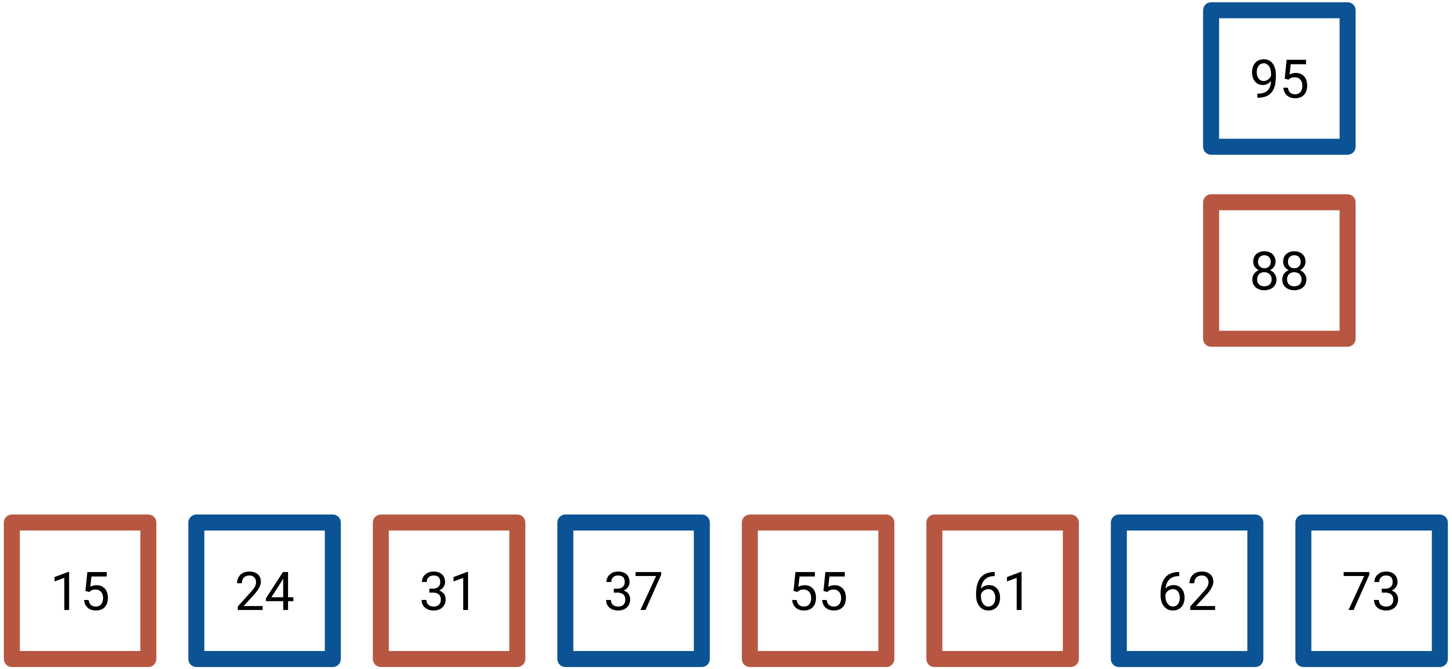


73 95

88

15 24 31 37 55 61 62

How do we Merge Two Sorted Arrays?

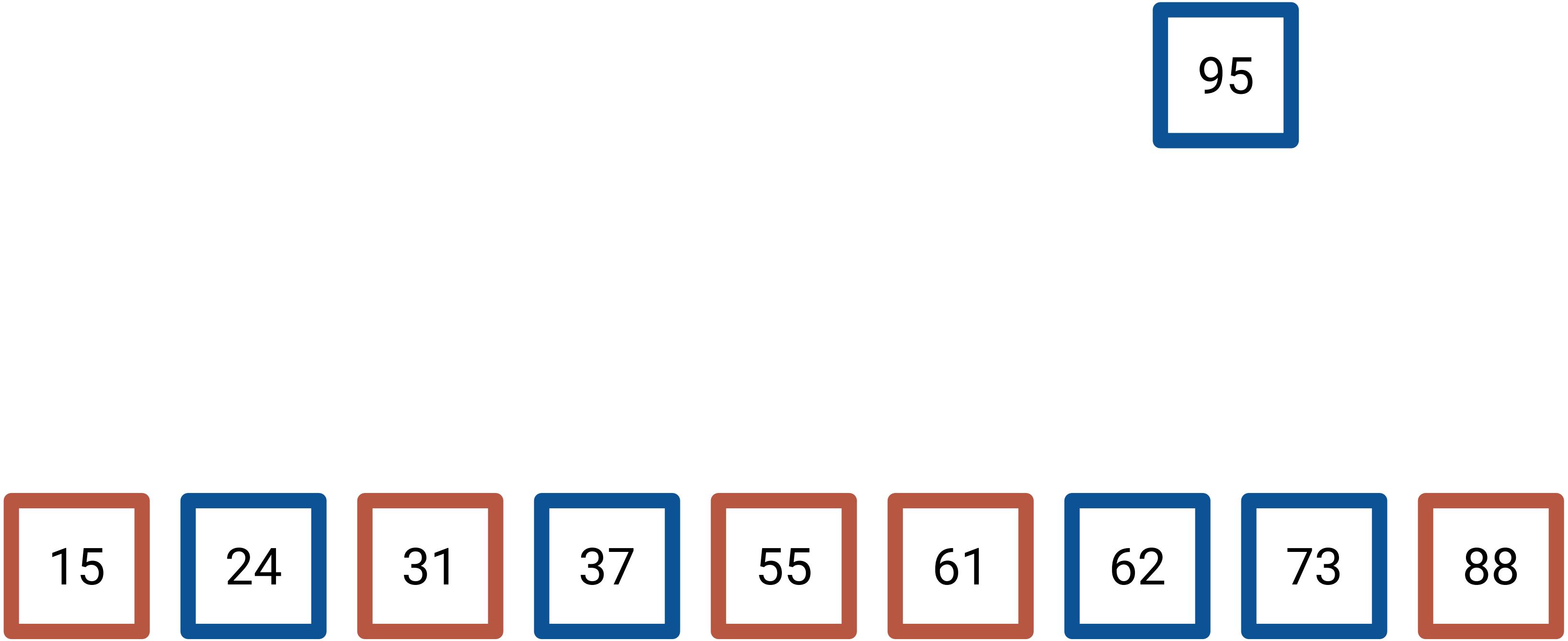


95

88

15 24 31 37 55 61 62 73

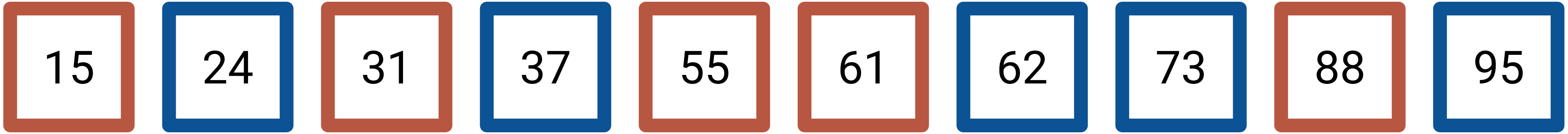
How do we Merge Two Sorted Arrays?



95

15 24 31 37 55 61 62 73 88

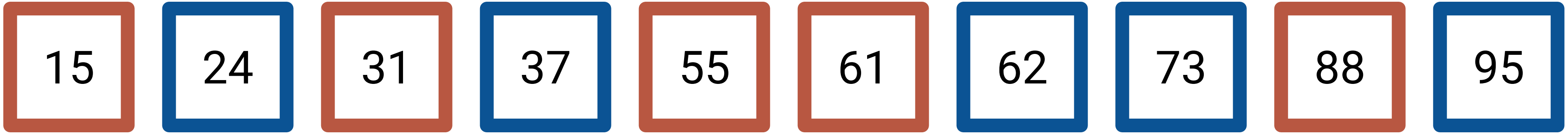
How do we Merge Two Sorted Arrays?



15	24	31	37	55	61	62	73	88	95

How do we Merge Two Sorted Arrays?

How many comparisons does this require?

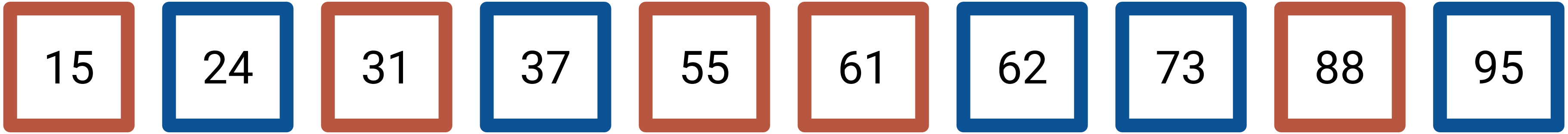


15 24 31 37 55 61 62 73 88 95

How do we Merge Two Sorted Arrays?

How many comparisons does this require?

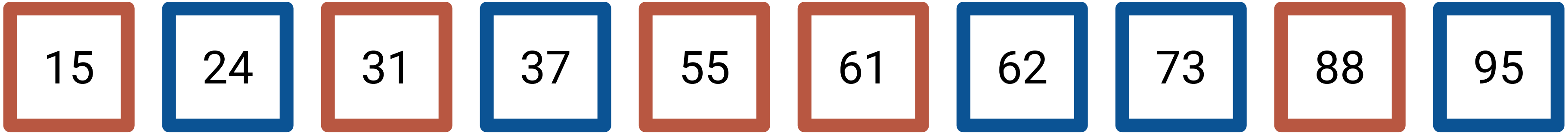
For ***N*** total items, we need ***N*** comparisons



How do we Merge Two Sorted Arrays?

How many comparisons does this require?

For ***N*** total items, we need ***N*** comparisons
(because we only ever need to compare the first element of each list)



15 24 31 37 55 61 62 73 88 95

Divide

- We know how to combine sorted arrays
- We know that the base case of ***N*** **= 1** is already sorted
- How do we divide our problem to get there?

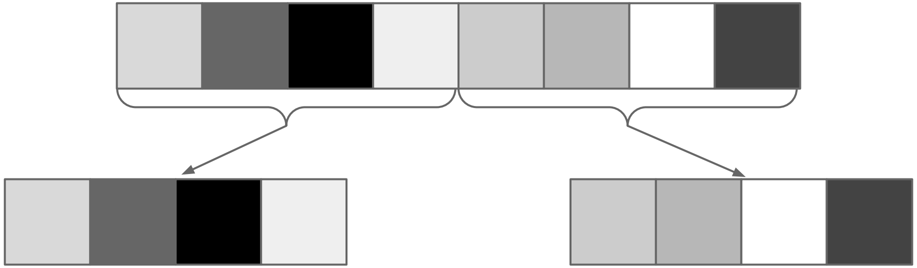
Divide

- We know how to combine sorted arrays
- We know that the base case of ***N* = 1** is already sorted
- How do we divide our problem to get there?

Let's divide our array in half (recursively)!

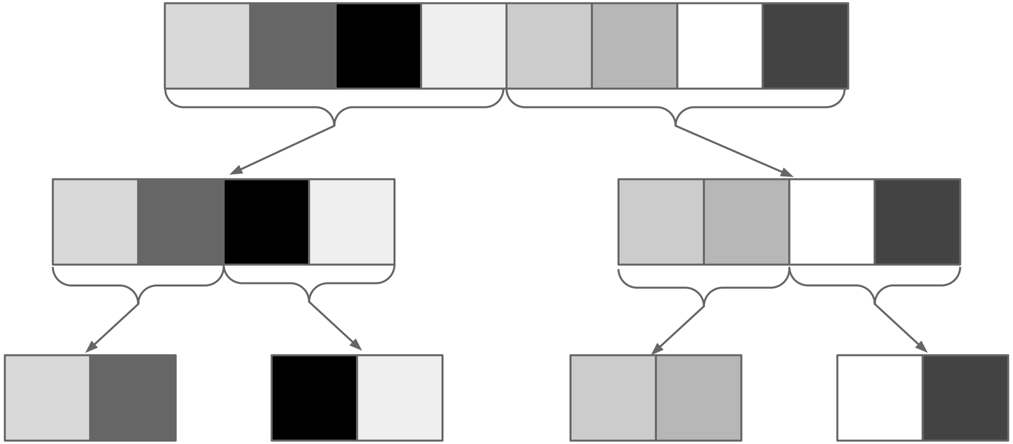
Visualization - Divide

Visualization - Divide



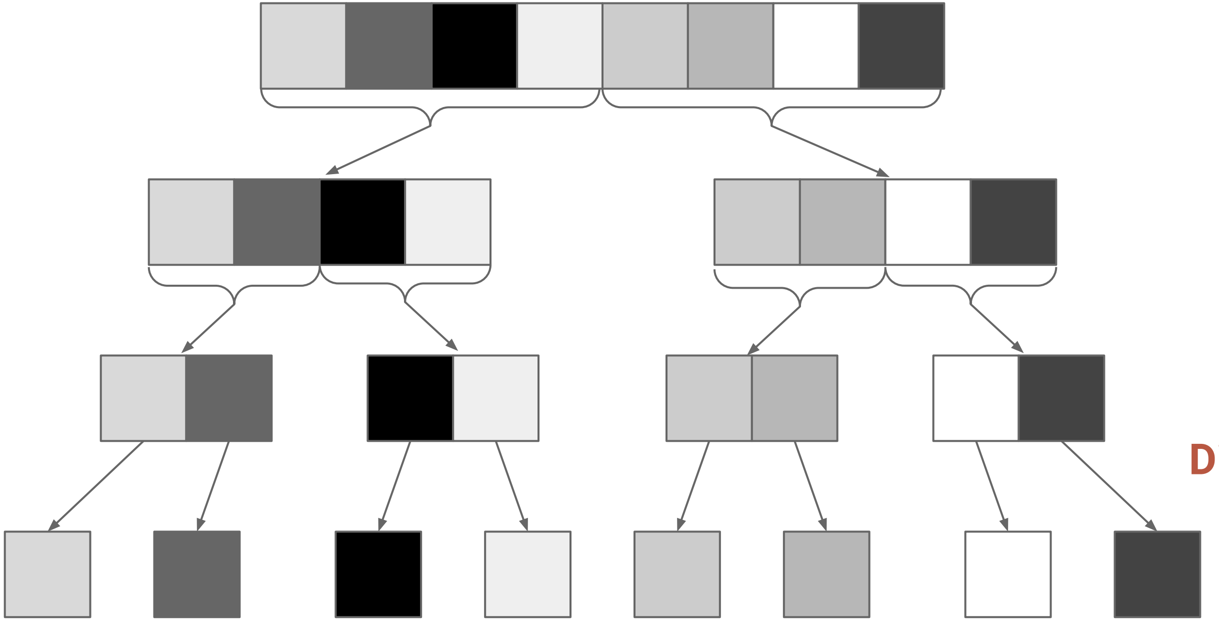
Divide the input in half

Visualization - Divide



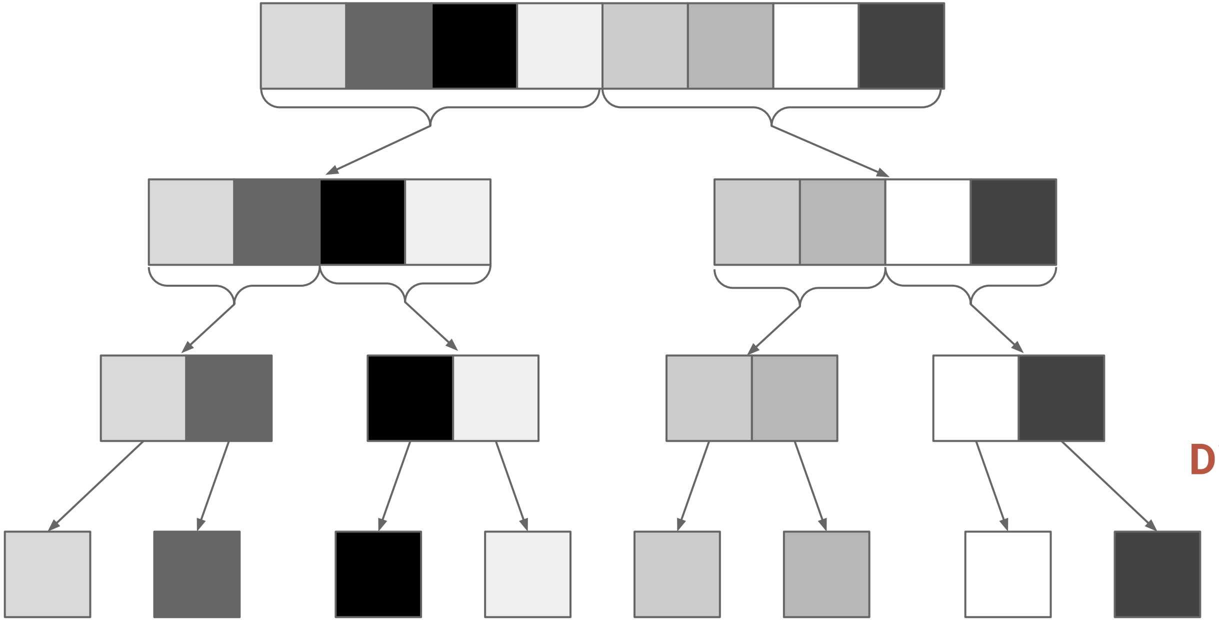
Divide each half in half

Visualization - Divide



Divide each half in half again...

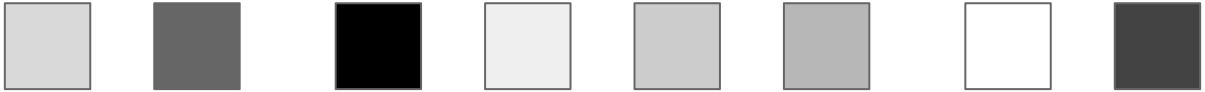
Visualization - Conquer



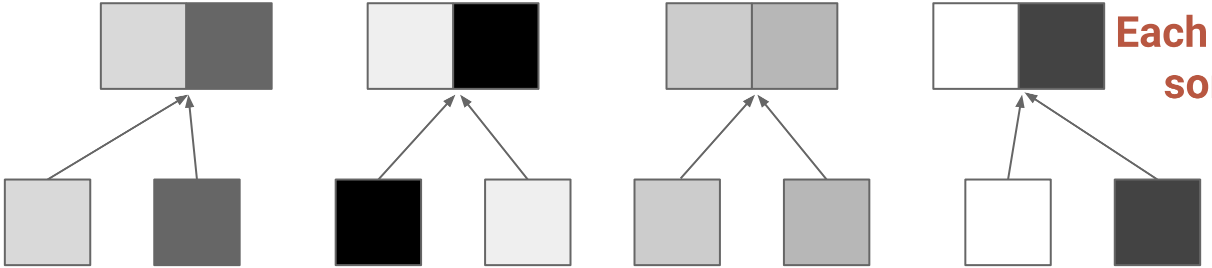
Divide each half in half again...

We can't divide in half anymore (base case)

Visualization - Combine

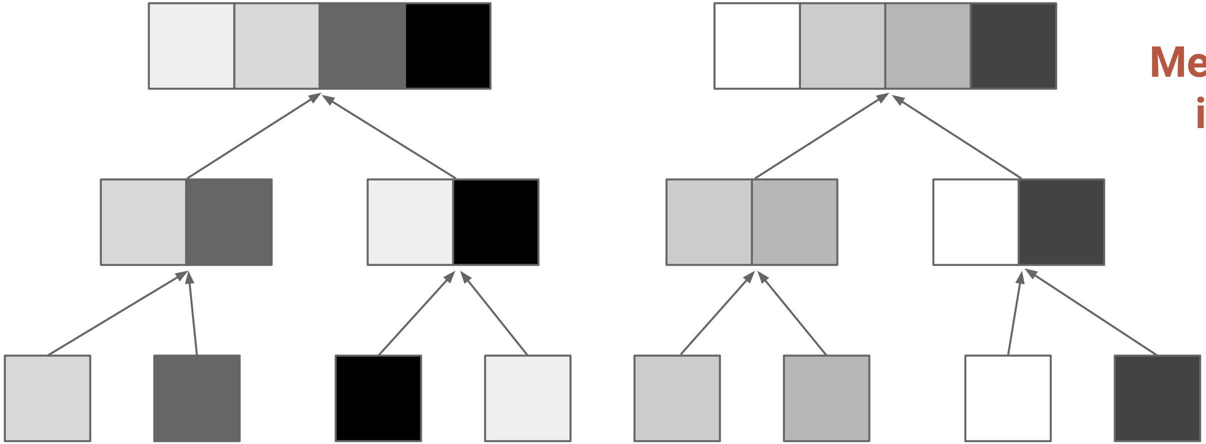


Visualization - Combine



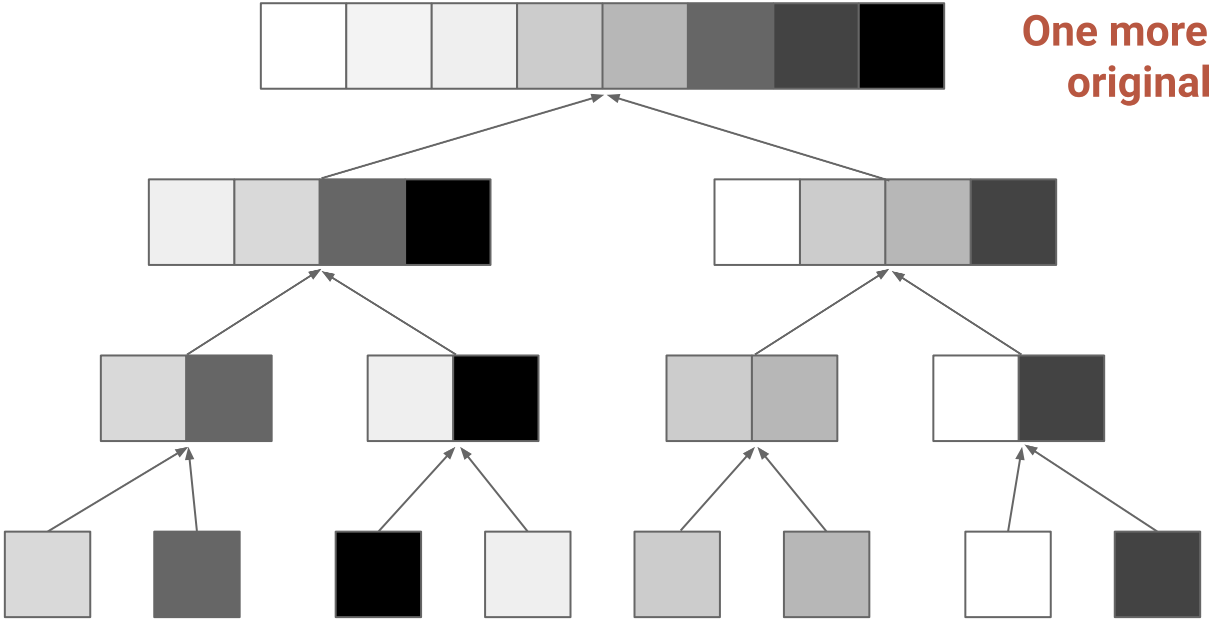
Each single item list is sorted…merge each pair into a bigger sorted list

Visualization - Combine



Merge each pair of 2 into sorted lists of size 4

Visualization - Combine



One more merge gets our original list fully sorted

mergeSort and mergeSortHelper

```
def mergeSort(X):
  mergeSortHelper(X, 0, len(X))
  return X

def mergeSortHelper(X, left, right):
  if (right - left) > 1:
    mid = (left + right) // 2
    mergeSortHelper(X, left, mid)
    mergeSortHelper(X, mid, right)
    merge(X, left, mid, right)
```

`mergeSort` and `mergeSortHelper`

```
def mergeSort(X):
  mergeSortHelper(X, 0, len(X))
  return X

def mergeSortHelper(X, left, right):
  if (right - left) > 1:
    mid = (left + right) // 2
    mergeSortHelper(X, left, mid)
    mergeSortHelper(X, mid, right)
    merge(X, left, mid, right)
```

The **`mergeSortHelper`** function performs merge sort on a region of the list.

In this case, the whole list.

`mergeSort` and `mergeSortHelper`

```
def mergeSort(X):
  mergeSortHelper(X, 0, len(X))
  return X

def mergeSortHelper(X, left, right):
  if (right - left) > 1:
    mid = (left + right) // 2
    mergeSortHelper(X, left, mid)
    mergeSortHelper(X, mid, right)
    merge(X, left, mid, right)
```

`mergeSortHelper` is a recursive function…it will call itself on a smaller input.

mergeSort and mergeSortHelper

```
def mergeSort(X):
  mergeSortHelper(X, 0, len(X))
  return X

def mergeSortHelper(X, left, right):
  if (right - left) > 1:
    mid = (left + right) // 2
    mergeSortHelper(X, left, mid)
    mergeSortHelper(X, mid, right)
    merge(X, left, mid, right)
```

We only do something if the region passed has more than one element.

With just one element (the base case), our list is already sorted so do nothing.

`mergeSort` and `mergeSortHelper`

```
def mergeSort(X):
  mergeSortHelper(X, 0, len(X))
  return X

def mergeSortHelper(X, left, right):
  if (right - left) > 1:
    mid = (left + right) // 2
    mergeSortHelper(X, left, mid)
    mergeSortHelper(X, mid, right)
    merge(X, left, mid, right)
```

If there is more than one element in our region, then compute the midpoint of the region.

`mergeSort` and `mergeSortHelper`

```
def mergeSort(X):
  mergeSortHelper(X, 0, len(X))
  return X

def mergeSortHelper(X, left, right):
  if (right - left) > 1:
    mid = (left + right) // 2
    mergeSortHelper(X, left, mid)
    mergeSortHelper(X, mid, right)
    merge(X, left, mid, right)
```

If there is more than one element in our region, then compute the midpoint of the region.

Then call **`mergeSortHelper`** on the left and right halves.

`mergeSort` and `mergeSortHelper`

```
def mergeSort(X):
  mergeSortHelper(X, 0, len(X))
  return X

def mergeSortHelper(X, left, right):
  if (right - left) > 1:
    mid = (left + right) // 2
    mergeSortHelper(X, left, mid)
    mergeSortHelper(X, mid, right)
    merge(X, left, mid, right)
```

If there is more than one element in our region, then compute the midpoint of the region.

Then call **`mergeSortHelper`** on the left and right halves.

Finally, merge the partial results.

merge

```
def merge(X, left, mid, right):
  temp = []
  left_idx = left
  right_idx = mid
  while left_idx < mid and right_idx < right:
    if X[left_idx] < X[right_idx]:
      temp.append(X[left_idx])
      left_idx = left_idx + 1
    else:
      temp.append(X[right_idx])
      right_idx = right_idx + 1
  while left_idx < mid:
    temp.append(X[left_idx])
    left_idx = left_idx + 1
  while right_idx < right:
    temp.append(X[right_idx])
    right_idx = right_idx + 1
  for i in range(left, right):
    X[i] = temp[i-left]
```

merge

Set the left_idx to the first index of the left half, and the right_idx to the first index of the right half.

```
def merge(X, left, mid, right):
  temp = []
  left_idx = left
  right_idx = mid
  while left_idx < mid and right_idx < right:
    if X[left_idx] < X[right_idx]:
      temp.append(X[left_idx])
      left_idx = left_idx + 1
    else:
      temp.append(X[right_idx])
      right_idx = right_idx + 1
  while left_idx < mid:
    temp.append(X[left_idx])
    left_idx = left_idx + 1
  while right_idx < right:
    temp.append(X[right_idx])
    right_idx = right_idx + 1
  for i in range(left, right):
    X[i] = temp[i-left]
```

merge

Keep going as long as there are more elements we haven't merged in both halves.

```
def merge(X, left, mid, right):
  temp = []
  left_idx = left
  right_idx = mid
  while left_idx < mid and right_idx < right:
    if X[left_idx] < X[right_idx]:
      temp.append(X[left_idx])
      left_idx = left_idx + 1
    else:
      temp.append(X[right_idx])
      right_idx = right_idx + 1
  while left_idx < mid:
    temp.append(X[left_idx])
    left_idx = left_idx + 1
  while right_idx < right:
    temp.append(X[right_idx])
    right_idx = right_idx + 1
  for i in range(left, right):
    X[i] = temp[i-left]
```

`merge`

If the front of the left half is smaller than the front of the right half, add it to our result and update the value of left_idx

```
def merge(X, left, mid, right):
  temp = []
  left_idx = left
  right_idx = mid
  while left_idx < mid and right_idx < right:
    if X[left_idx] < X[right_idx]:
      temp.append(X[left_idx])
      left_idx = left_idx + 1
    else:
      temp.append(X[right_idx])
      right_idx = right_idx + 1
  while left_idx < mid:
    temp.append(X[left_idx])
    left_idx = left_idx + 1
  while right_idx < right:
    temp.append(X[right_idx])
    right_idx = right_idx + 1
  for i in range(left, right):
    X[i] = temp[i-left]
```

`merge`

Do the opposite if the front of the right half was the smaller of the two

```
def merge(X, left, mid, right):
  temp = []
  left_idx = left
  right_idx = mid
  while left_idx < mid and right_idx < right:
    if X[left_idx] < X[right_idx]:
      temp.append(X[left_idx])
      left_idx = left_idx + 1
    else:
      temp.append(X[right_idx])
      right_idx = right_idx + 1
  while left_idx < mid:
    temp.append(X[left_idx])
    left_idx = left_idx + 1
  while right_idx < right:
    temp.append(X[right_idx])
    right_idx = right_idx + 1
  for i in range(left, right):
    X[i] = temp[i-left]
```

`merge`

After one of the halves runs out, make sure to just append the rest of the half that still has leftover elements

```
def merge(X, left, mid, right):
  temp = []
  left_idx = left
  right_idx = mid
  while left_idx < mid and right_idx < right:
    if X[left_idx] < X[right_idx]:
      temp.append(X[left_idx])
      left_idx = left_idx + 1
    else:
      temp.append(X[right_idx])
      right_idx = right_idx + 1
  while left_idx < mid:
    temp.append(X[left_idx])
    left_idx = left_idx + 1
  while right_idx < right:
    temp.append(X[right_idx])
    right_idx = right_idx + 1
  for i in range(left, right):
    X[i] = temp[i-left]
```

`merge`

Copy the result back into the original list

```python
def merge(X, left, mid, right):
  temp = []
  left_idx = left
  right_idx = mid
  while left_idx < mid and right_idx < right:
    if X[left_idx] < X[right_idx]:
      temp.append(X[left_idx])
      left_idx = left_idx + 1
    else:
      temp.append(X[right_idx])
      right_idx = right_idx + 1
  while left_idx < mid:
    temp.append(X[left_idx])
    left_idx = left_idx + 1
  while right_idx < right:
    temp.append(X[right_idx])
    right_idx = right_idx + 1
  for i in range(left, right):
    X[i] = temp[i-left]
```

Runtime

How many steps does it take to sort a list with ***N*** *items?*

Runtime

How many steps does it take to sort a list with ***N*** *items?*

How many steps does it take to merge the ***N*** *items?*

Runtime

*How many steps does it take to sort a list with **N** items?*

*How many steps does it take to merge the **N** items? **N** steps*

Runtime

*How many steps does it take to sort a list with **N** items?*

*How many steps does it take to merge the **N** items? **N** steps*

How many times do we have to merge?

Runtime

How many steps does it take to sort a list with ***N*** *items?*

How many steps does it take to merge the ***N*** *items?* ***N*** *steps*

How many times do we have to merge? **log(*N*)**

Runtime

How many steps does it take to sort a list with ***N*** *items?*

How many steps does it take to merge the ***N*** *items?* ***N*** *steps*

How many times do we have to merge? **log(*N*)**

Total number of steps: *N* log(*N*)

SelectionSort vs MergeSort

SelectionSort requires roughly N^2 steps to sort a list of size ***N***

N^2 *grows pretty fast…*

*If we **double** the size of our list we **quadruple** the number of steps*

SelectionSort vs MergeSort

SelectionSort requires roughly N^2 steps to sort a list of size N

N^2 grows pretty fast…

*If we **double** the size of our list we **quadruple** the number of steps*

MergeSort requires roughly N **log(**N**)** steps to sort a list of size N

N **log(**N**)** grows much slower

*If we **double** the size of our list, we only increase the number of steps by a **little more than double***