CSCI 241

Lecture 3:
Recursion, Mergesort

Announcements

- First programming assignment (A1) out tonight or tomorrow
    - We'll cover the rest of the sorting algorithms you need for A1 it this week.
- Quiz 0 is graded. You'll get an email from Gradescope to set up your account, log in, and see your graded work.

Happenings

- Tuesday, 1/15: CS/SMATE Faculty Candidate, **Caroline Hardin**, Research Talk: Connection Reset by Peer: Who Learns at Hackathons?, 4-5PM, CF 316
- Wednesday, 1/6: CS/SMATE Faculty Candidate, **Caroline Hardin**, Teaching Talk: When the ‘Ifs’ are Stiff and ‘Nots’ are Knots: Debugging Techniques through E-textiles, 4-5PM, CF 316
- Wednesday 1/16: WWU’s MLK Jr event: “We are not the makers of history. We are made by history”, 7PM, PAC
- Winter Career Fair featuring STEM, 2/7: get your resume ready!

Roadmap

- Last week:
  - selection and insertion sorts
  - Some intuition on runtime analysis
- This week:
  - Recursive sorting algorithms (merge, quick)
  - Radix sort
- Next week: data structures

Goals for today:

- Understand how recursive methods are **executed**.
- Be able to **understand** and **develop** recursive methods *without* getting confused by the details of how they are executed.
- Gain intuition for how merge sort works

Why are we talking about recursion, I thought we were learning about sorting?

```
mergeSort(A, start, end):
  if (A.length < 2):
    return
  mid = (end-start)/2
  mergeSort(A,start,mid)
  mergeSort(A,mid, end)
  merge(A, start, mid, end)
```

How do we execute recursive methods?

How do we execute non-recursive methods?

```
x = max(1,3)
    => 3
```

How do we execute non-recursive methods?

```
x = max(1,3)
          3
```

How do we execute recursive methods?

```
/** return n!; pre: n >= 0 */
fact(n):
  if n == 0:
      return 1
   return n * fact(n - 1)
```

```
fact(3)
=> 3 * fact(2)
       => 2 * fact(1)
              => 1 * fact(0)
                     => 1
```

How do we execute recursive methods?

```
/** return n!; pre: n >= 0 */
fact(n):
  if n == 0:
      return 1
   return n * fact(n - 1)
```

```
fact(3)
=> 3 * fact(2)
       => 2 * fact(1)
              => 1 * fact(0)
                     1
```

How do we execute recursive methods?

```
/** return n!; pre: n >= 0 */
fact(n):
  if n == 0:
      return 1
   return n * fact(n - 1)
```

```
fact(3)
=> 3 * fact(2)
       => 2 * fact(1)
              => 1 * 1
```

How do we execute recursive methods?

```
/** return n!; pre: n >= 0 */
fact(n):
  if n == 0:
      return 1
   return n * fact(n - 1)
```

```
fact(3)
=> 3 * fact(2)
        => 2 * fact(1)
                    1
```

How do we execute recursive methods?

```
/** return n!; pre: n >= 0 */
fact(n):
  if n == 0:
      return 1
   return n * fact(n - 1)
```

```
fact(3)
=> 3 * fact(2)
              2
```

How do we execute recursive methods?

```
/** return n!; pre: n >= 0 */
fact(n):
  if n == 0:
      return 1
   return n * fact(n - 1)
```

```
fact(3)
=> 6
```

Your turn

Fibonacci:

n:	0	1	2	3	4	5	6	7	8
fib(n):	0	1	1	2	3	5	8	13	21

```
/** return the nth fibonacci number
  * precondition: n >= 0 */
fib(n):
  if n <= 1:
    return n
  return fib(n-1) + fib(n-2)
```

Problem 1: If I call `fib(3)`,

A. How many times is `fib` called? (show your work)
B. What value is returned?

Your turn

Fibonacci:

n:	0	1	2	3	4	5	6	7	8
fib(n):	0	1	1	2	3	5	8	13	21

```
/** return the nth fibonacci number
  * precondition: n >= 0 */
fib(n):
  if n <= 1:
    return n
  return fib(n-1) + fib(n-2)
```

1A - ABCD:

A. 3
B. 4
C. 5
D. 6

Your turn

Fibonacci:

n:	0	1	2	3	4	5	6	7	8
fib(n):	0	1	1	2	3	5	8	13	21

```
/** return the nth fibonacci number
  * precondition: n >= 0 */
fib(n):
  if n <= 1:
    return n
  return fib(n-1) + fib(n-2)
```

1A - ABCD:

A. 3
B. 4
C. 5
D. 6

Problem 2: If I call `fib(4)`,

A. How many times is `fib` called? (show your work)
B. What value is returned?

How do we understand recursive methods?

1. Make sure it has a **precise specification**.
2. Make sure it works in the **base case**.
3. Ensure that each recursive call makes **progress** towards the base case.
4. Replace each **recursive call** with the **spec** and verify overall behavior is correct.

How do we understand recursive methods?

```
def count_e(s):
    """ returns # of 'e' in string s
    """
    if len(s) == 0:
        return 0
    first = 0
    if s[0] == 'e':
        first = 1

    return first + count_e(s[1..end])
```

1. **spec**
2. **base case**
3. **progress**
4. **recursive call —> spec**

1. Spec
2. Base case
3. Progress
4. Recursive call <=> spec

Got it?

This code has **at least one** bug:

```
dup(String s):
   if s.length == 0:
      return s

   return s[0] + s[0] + dup(s)
```

Got it?

1. Spec
2. Base case
3. Progress
4. Recursive call
   <=> spec

```
/** return a copy of s with each
  * character repeated */
dup(String s):
   if s.length == 0:
      return s

   return s[0] + s[0] + dup(s)
```

1. **spec!**

Got it?

1. Spec
2. Base case
3. Progress
4. Recursive call
   <=> spec

```
/** return a copy of s with each
  * character repeated */
dup(String s):
  if s.length == 0:
     return s

  return s[0] + s[0] + dup(s)
```

3. **progress!**

Got it?

1. Spec
2. Base case
3. Progress
4. Recursive call
   <=> spec

```
/** return a copy of s with each
  * character repeated */
dup(String s):
  if s.length == 0:
     return s

  return s[0] + s[0] + dup(s[1..s.length])
```

3. **progress!**

How do we develop recursive methods?

1. Write a **precise specification**.

2. Write a **base case** without using recursion.

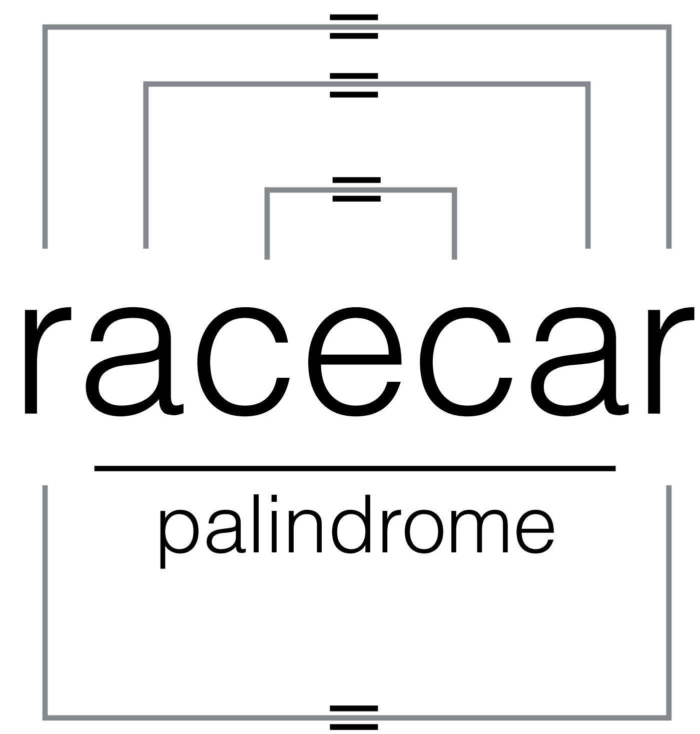
3. Define all other cases in terms of **subproblems** of the same kind.

4. Implement these definitions using the **recursive call** to compute solutions to the subproblems.

Palindromes

Examples:

- civic
- radar
- deed
- racecar



Recursive definition: A string s is a palindrome if

- `s.length < 2`, OR
- `s[0] == s[end-1]` AND s[1..end-2] is a palindrome

racecar

palindrome

=

Recursive definition: A string so is a palindrome if

- `s.length < 2`, OR
- `s[0] == s[end-1]` AND s[1..end-2] is a palindrome

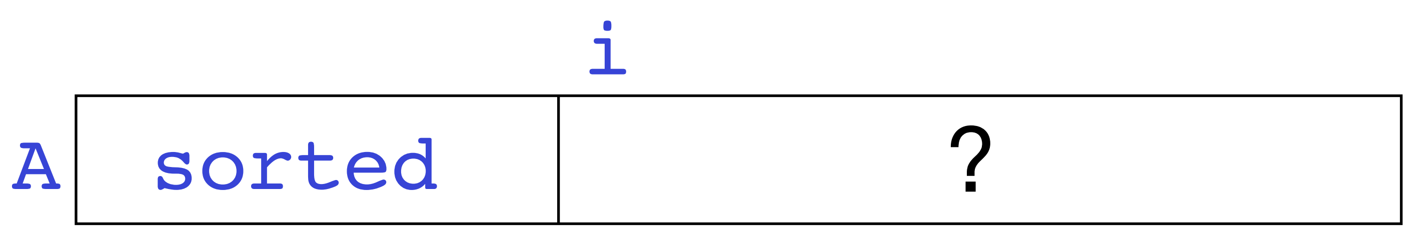
Problem 3: Write a recursive palindrome checker:

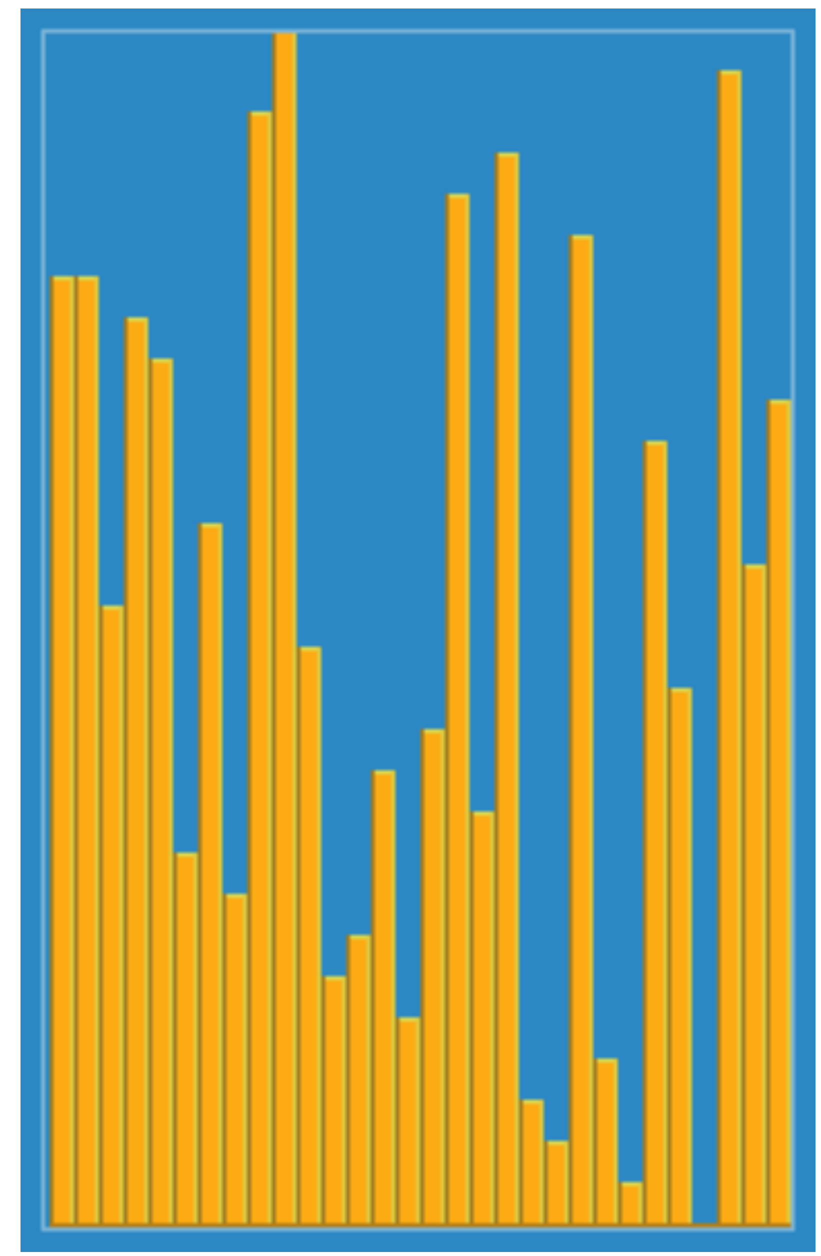
```
/** return true iff s[start..end]
  * is a palindrome */
public boolean isPal(s, start, end) {
  // your code here
}
```

Incremental Algorithms

solve a problem a little bit at a time.

Natural programming mechanism: loops





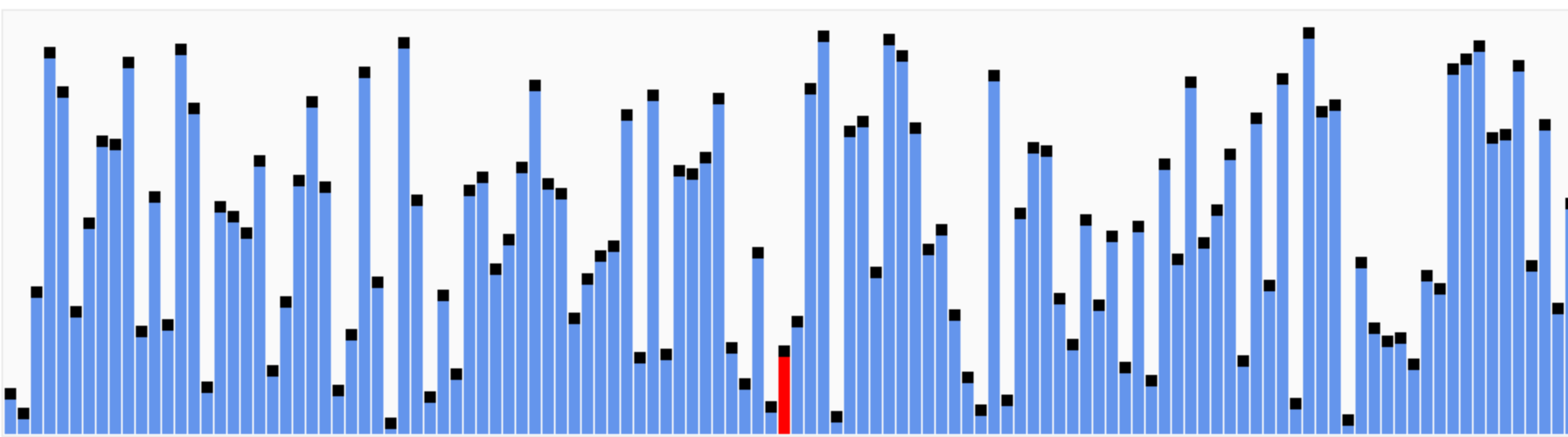
insertion sort

Divide-and-Conquer Algorithms

solve a problem by breaking it into smaller problems.

(easier!)

Natural programming mechanism: recursion



https://upload.wikimedia.org/wikipedia/commons/f/fe/Quicksort.gif

Divide-and-Conquer Algorithms

solve a problem by breaking it into smaller problems.

Natural programming mechanism: recursion

Three generic steps:

1. **Divide (into sub-problems)**
2. **Conquer (the sub-problems)**
3. **Combine (into a solution to the original problem)**

Divide-and-Conquer Algorithms

solve a problem by breaking it into smaller problems.

Natural programming mechanism: recursion

Three generic steps:

1. **Divide (into sub-problems)**
2. **Conquer (the sub-problems)**
3. **Combine (into a solution to the original problem)**

Why are we talking about divide-and-conquer, I thought we were learning how to sort things?

An example of Divide-and-Conquer

```
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
  if (A.length < 2):
    return
  mid = (end-start)/2

  mergeSort(A,start,mid)
  mergeSort(A,mid, end)

  merge(A, start, mid, end)
```

1. **Divide**
2. **Conquer**
3. **Combine**

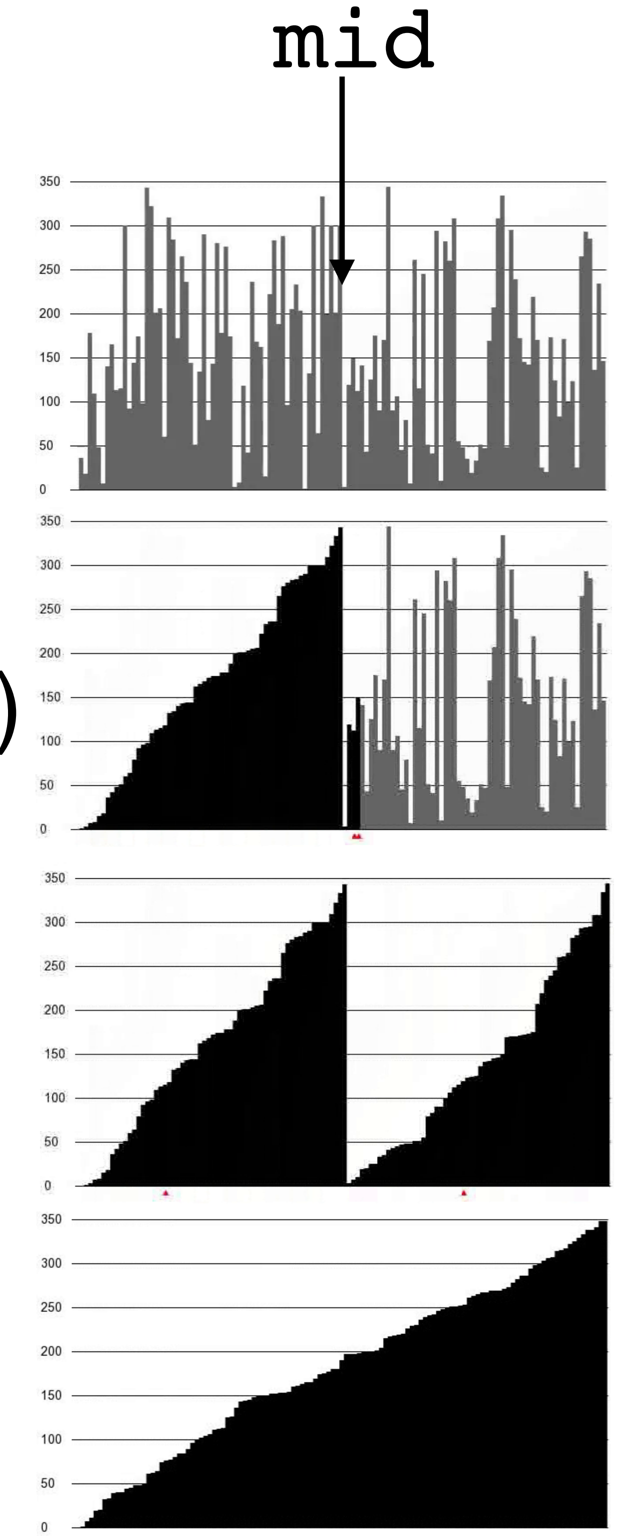
```
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
  if (A.length < 2):
    return
  mid = (end-start)/2

  mergeSort(A,start,mid)

  mergeSort(A,mid, end)

  merge(A, start, mid, end)
```

Divide

Conquer (left)

Conquer (right)

Combine



1. Spec

```
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
  if (A.length < 2):
    return
  mid = (end-start)/2
```

2. Base case

3. Progress

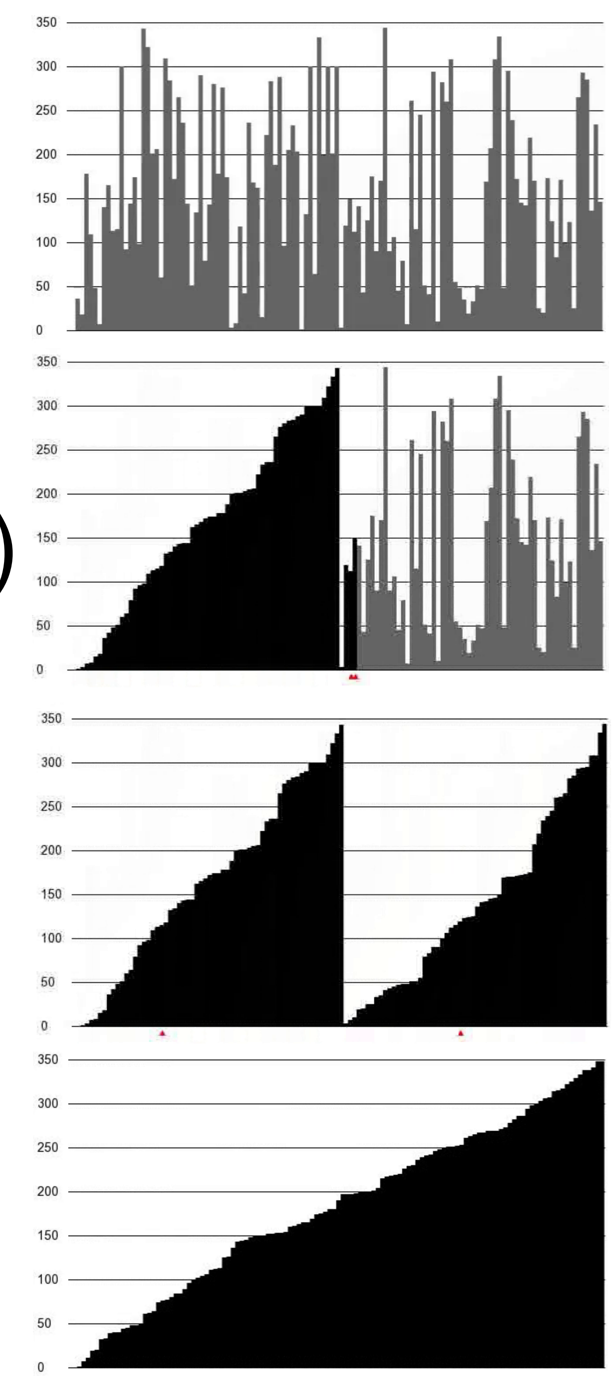
```
  mergeSort(A,start,mid)
  mergeSort(A,mid, end)
  merge(A, start, mid, end)
```

Divide

Conquer (left)

Conquer (right)

Combine



1. Spec

```
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
  if (A.length < 2):
    return
  mid = (end-start)/2
```

2. Base case

3. Progress

```
  sort A[start..mid]
```

4. Replace recursive calls with spec

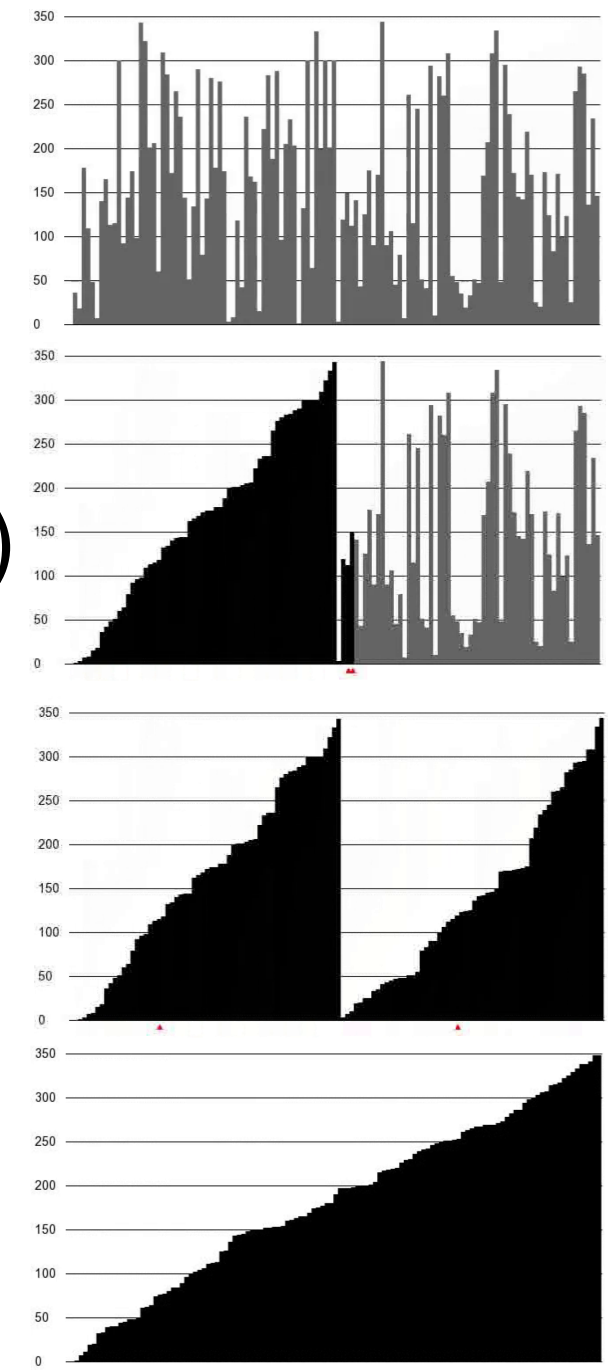
```
  sort A[mid..end]

  merge(A, start, mid, end)
```

Divide

Conquer (left)

Conquer (right)

Combine



Merge Step

- Merge two halves, each of which is **sorted**.

https://facultyweb.cs.wwu.edu/~wehrwes/courses/csci241_18f/img/merge.gif