CSCI 241

Lecture 5

Recursive Sorting:

Mergesort and Quicksort

Announcements

- Quiz 0 scores are out.
- I will post a video going over it (Q1.mp4) later today.
-

Goals:

- Be able to **understand** and **develop** recursive methods *without* thinking about the details of how they are executed.
- Know the generic steps of a divide-and-conquer algorithm.
- Thoroughly understand the mechanism of mergesort and quicksort.
- Be prepared to implement **merge** and **partition** helper methods.

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 understand recursive methods?

How do we understand recursive methods?

1. Make sure it has a **precise specification**.

How do we understand recursive methods?

1. Make sure it has a **precise specification**.

2. Make sure it works in the **base case**.

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.

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?

```
/** returns # of ‘e’ in string s */
def count_e(s):
  if len(s) == 0:
     return 0
  first = 0
  if s[0] == ‘e’:
     first = 1

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

How do we understand recursive methods?

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

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

1. **spec** ✓

How do we understand recursive methods?

```
/** returns # of 'e' in string s */
def count_e(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**

How do we understand recursive methods?

```
/** returns # of 'e' in string s */
def count_e(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**

How do we understand recursive methods?

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

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!**

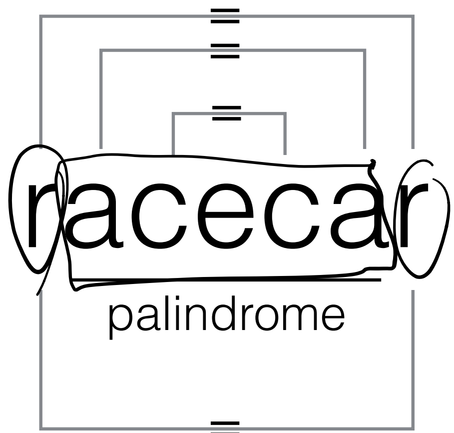
How do we develop recursive methods?

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.

Examples:

Palindromes

- 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

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:

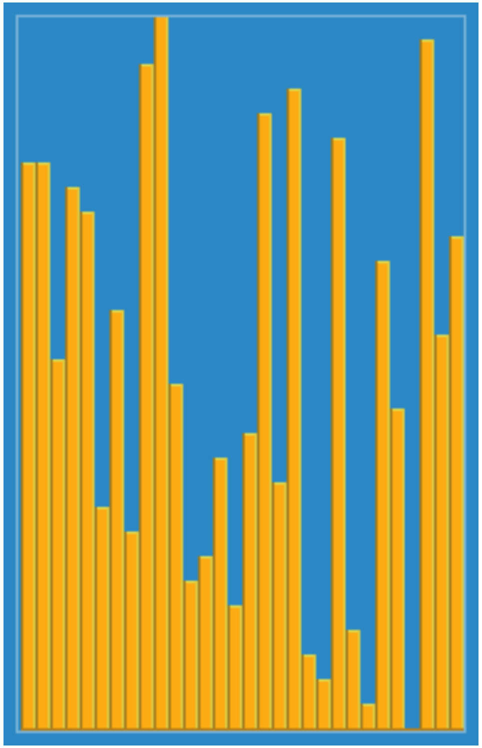
```
/** return true iff s[start..end]
  * is a palindrome */
public boolean isPal(s, start, end) {
  // your code here
}
```

Incremental Algorithms

solve a problem a little bit at a time.

Incremental Algorithms

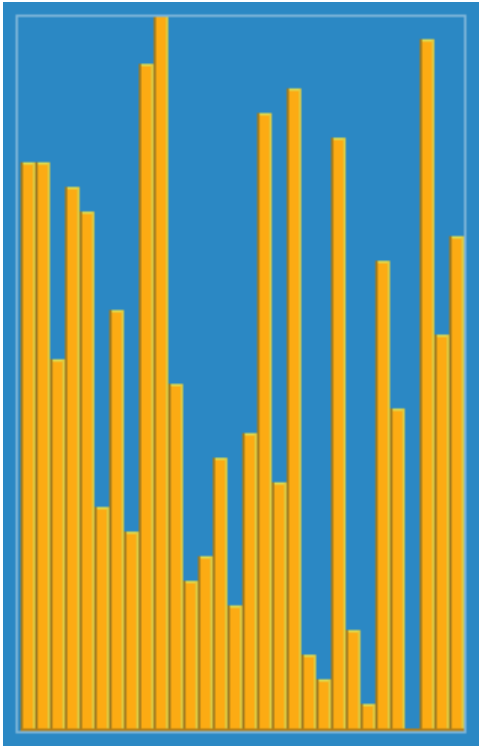
solve a problem a little bit at a time.



Incremental Algorithms

solve a problem a little bit at a time.

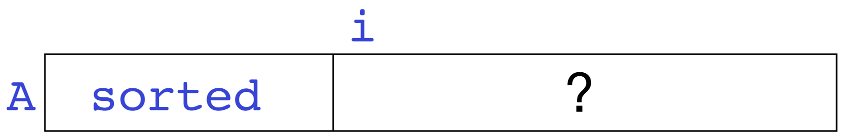
Natural programming mechanism: loops

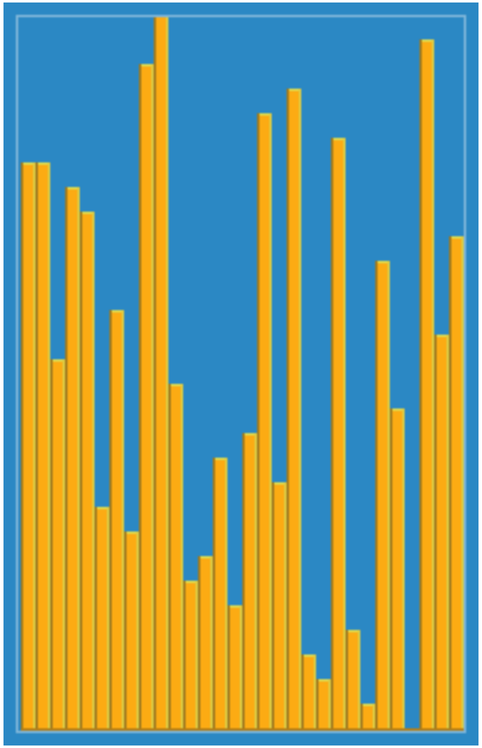


Incremental Algorithms

solve a problem a little bit at a time.

Natural programming mechanism: loops

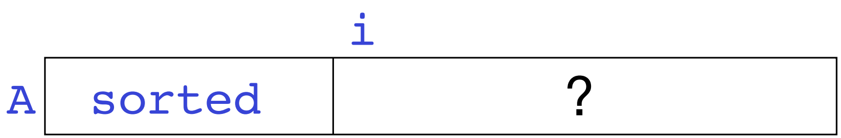


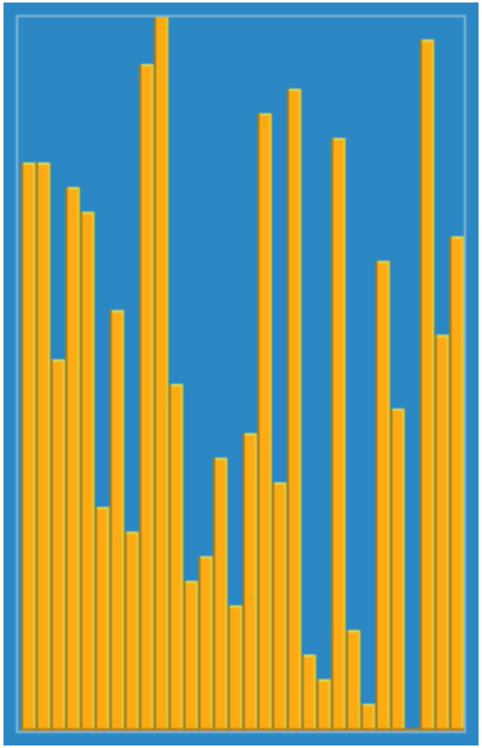


Incremental Algorithms

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.

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

Divide-and-Conquer

Algorithms

solve a problem by breaking it into smaller problems.

(easier!)

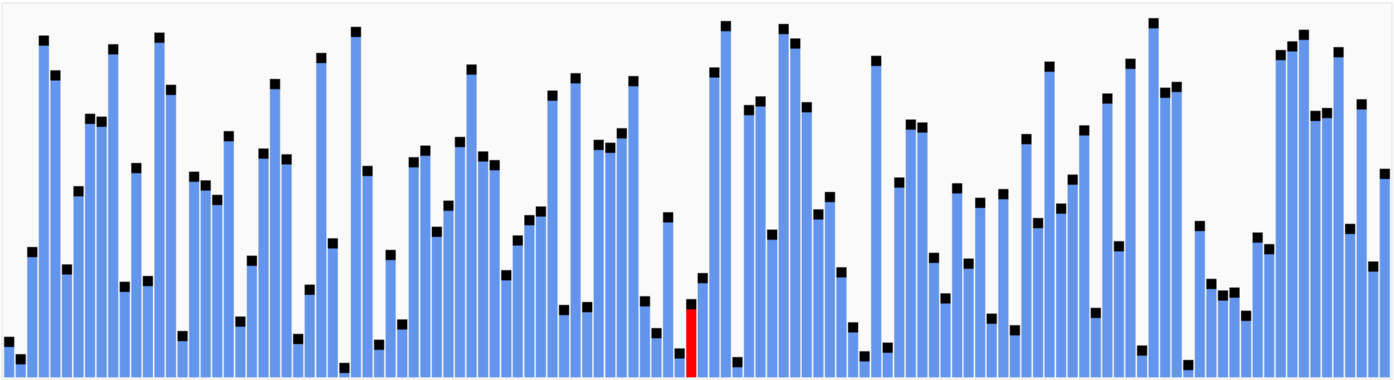
https://upload.wikimedia.org/wikipedia/commons/f/fe/Quicksort.gif

Divide-and-Conquer

Algorithms

solve a problem by breaking it into smaller problems.

(easier!)



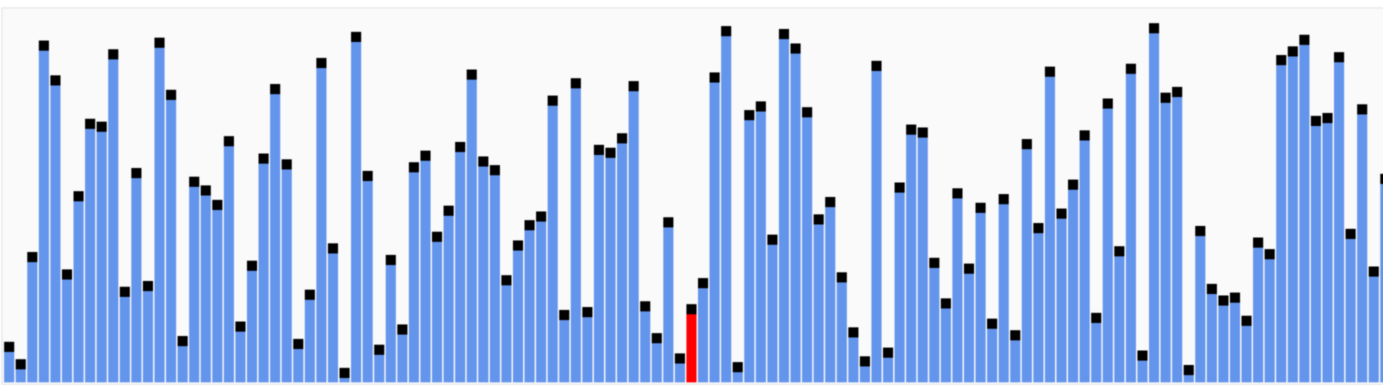
https://upload.wikimedia.org/wikipedia/commons/f/fe/Quicksort.gif

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

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)
```

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
```

1. Divide

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

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

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**

An example of Divide-and-Conquer

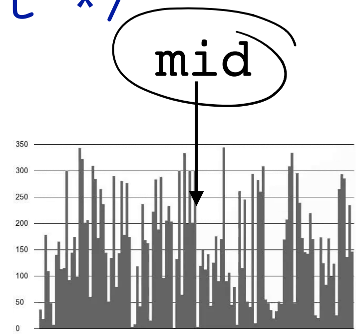
```
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
  if (A.length < 2):
    return
  mid = (end+start)/2

  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)
```

```
/** 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)
```

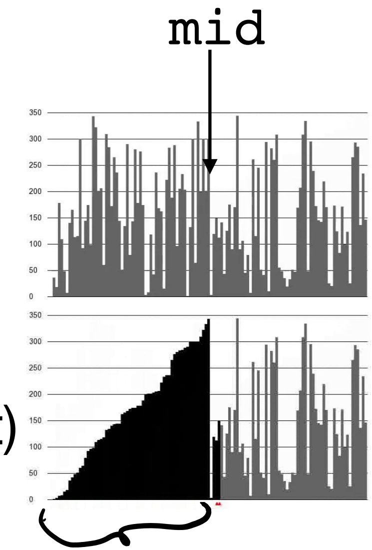
Divide



```
/** 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)

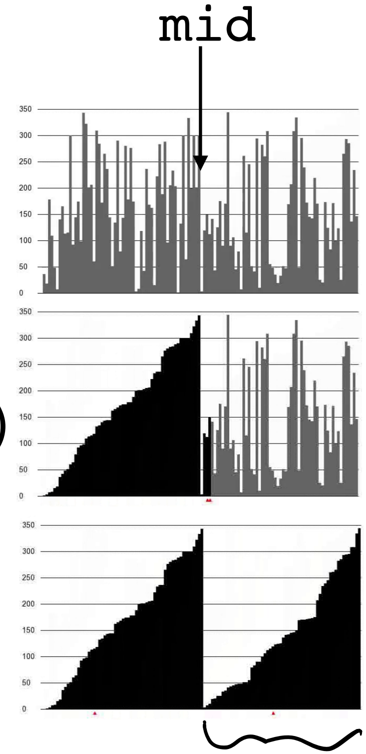


```
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
  if (A.length < 2):
    return
  mid = (end+start)/2

  mergeSort(A,start,mid)

  mergeSort(A,mid, end)

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

Divide

Conquer (left)

Conquer (right)



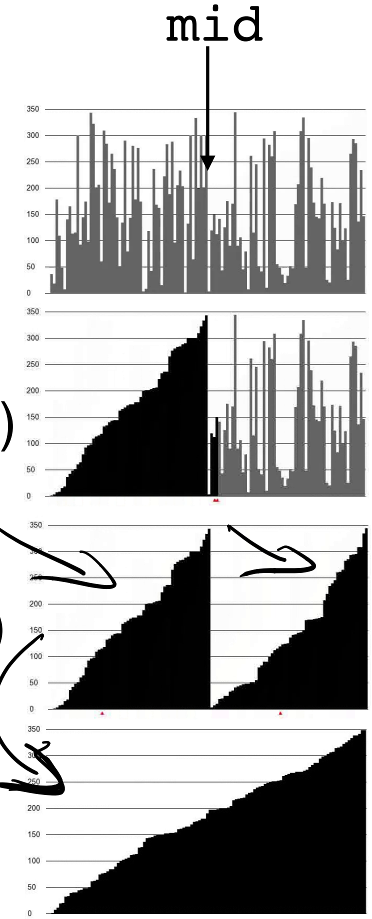
```
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
   if (A.length < 2):
      return
   mid = (end+start)/2

   mergeSort(A,start,mid)

   mergeSort(A,mid, end)

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

Divide

Conquer (left)

Conquer (right)

Combine



Merge Step

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

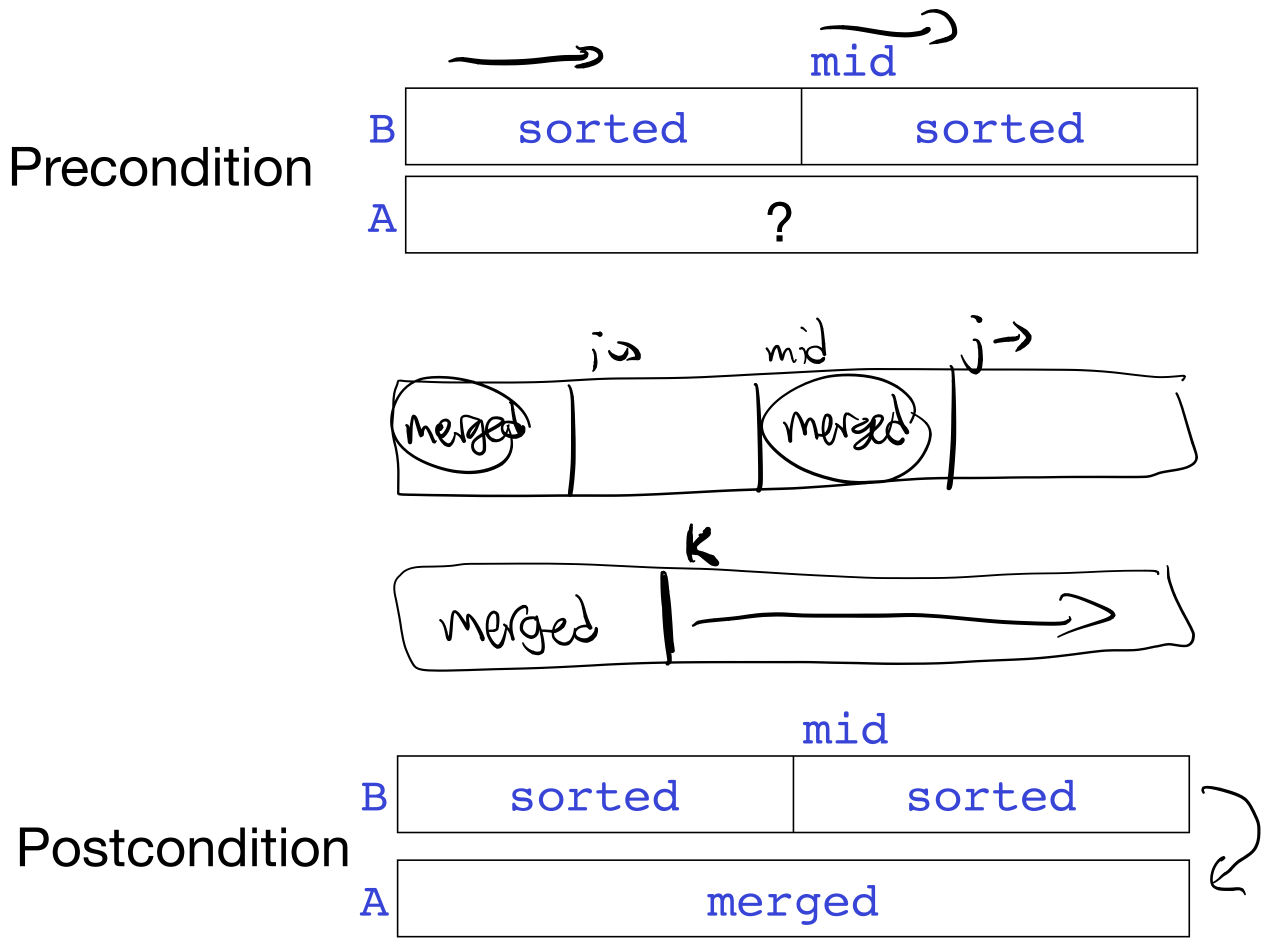
https://facultyweb.cs.wwu.edu/~wehrwes/courses/csci241_18f/img/merge.gif

Merge Step

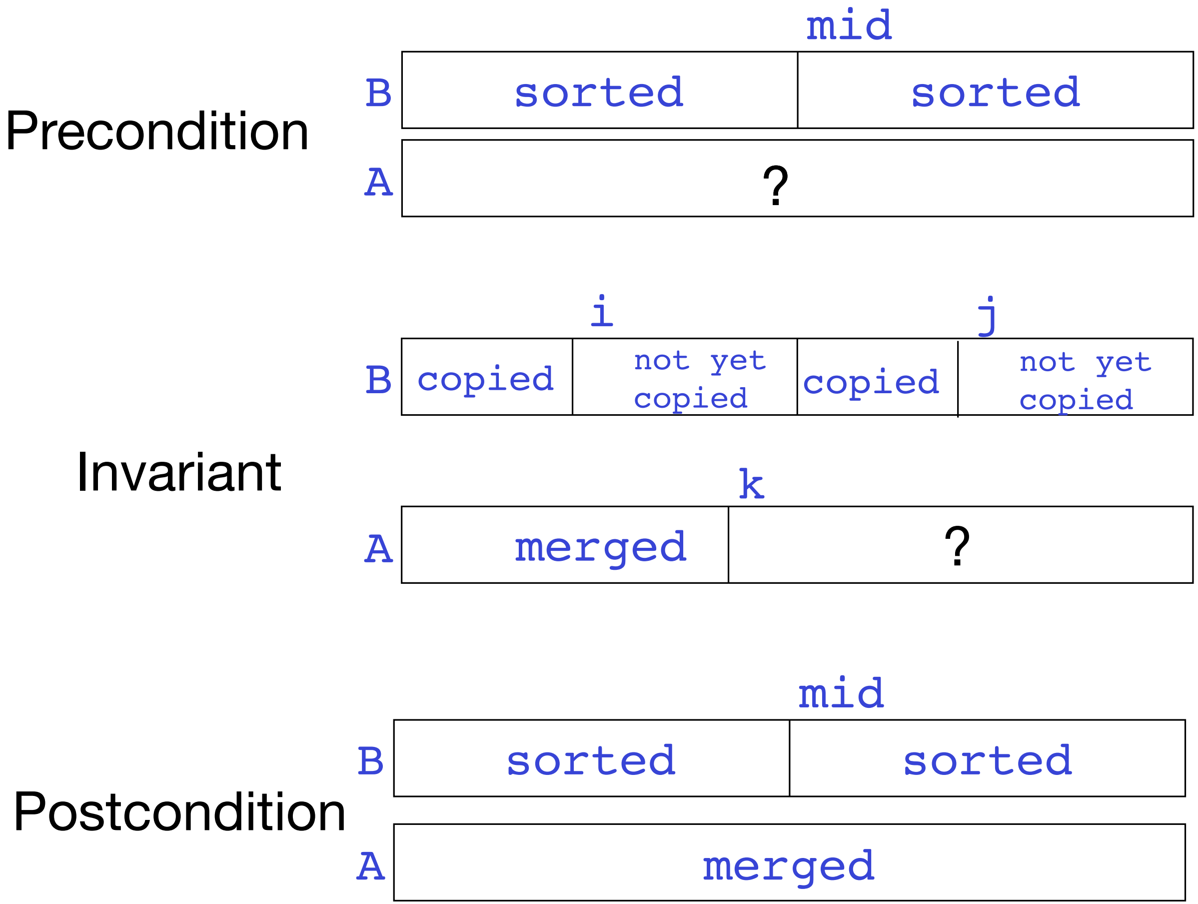
- Merge two halves, each of which is **sorted**.

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

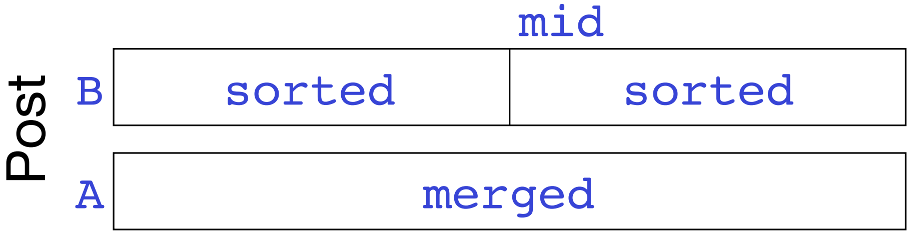
Merge step: Loop Invariant



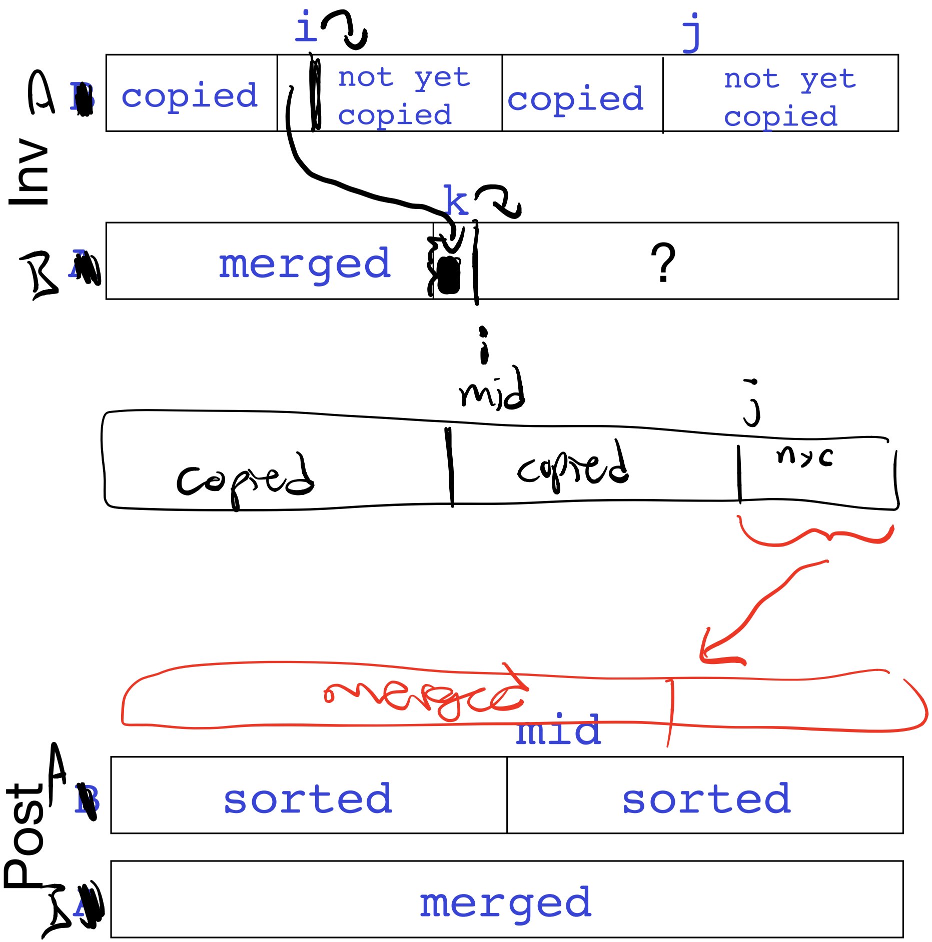
Merge step: Loop Invariant



Merge step: Loop Invariant



Merge step: Loop Invariant



Set i, j

Make new array A

while neither uncopied section is empty:

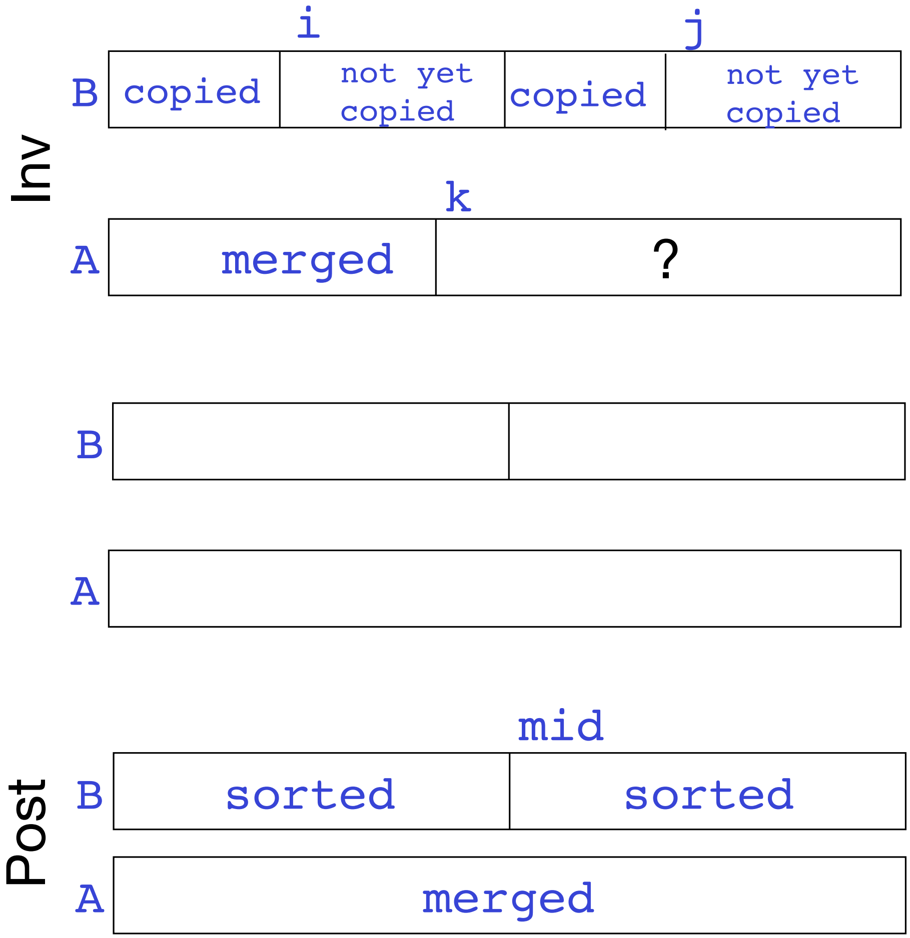
copy smaller of A[i], A[j] into B[k]

increment i, k

or j

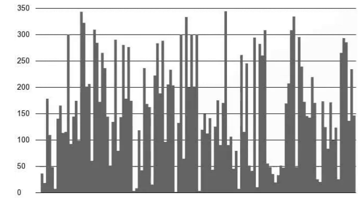
copy remaining values from left or right half

Merge step: Loop Invariant

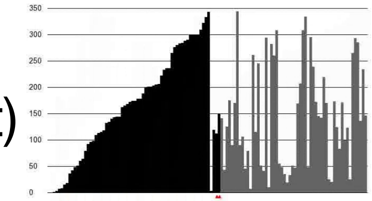


```
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
  if (A.length < 2):
    return
  mid = (end-start)/2

  mergeSort(A,start,mid)

  mergeSort(A,mid, end)

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

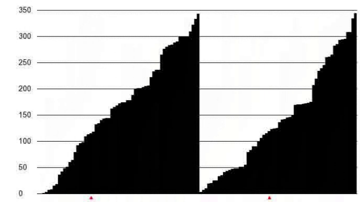
Divide



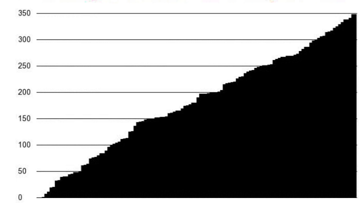
Conquer (left)



Conquer (right)



Combine



https://visualgo.net/bn/sorting

Quicksort

```
/** mergesort A[st..end]*/
mergeSort(A, st, end):
  if (small):
    return

  mid = (end-start)/2

  mergeSort(A,st,mid)
  mergeSort(A,mid, end)

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

```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return

  mid = partition(A,st,end)

  quickSort(A,st,mid)
  quickSort(A,mid, end)
```

Quicksort

```
/** mergesort A[st..end]*/
mergeSort(A, st, end):
  if (small):
    return

  mid = (end-start)/2

  mergeSort(A,st,mid)
  mergeSort(A,mid, end)

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

Divide

```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return

  mid = partition(A,st,end)

  quickSort(A,st,mid)
  quickSort(A,mid, end)
```

Quicksort

```
/** mergesort A[st..end]*/
mergeSort(A, st, end):
  if (small):
    return

  mid = (end-start)/2

  mergeSort(A,st,mid)
  mergeSort(A,mid, end)

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

Divide

Conquer

```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return

  mid = partition(A,st,end)

  quickSort(A,st,mid)
  quickSort(A,mid, end)
```

Quicksort

```
/** mergesort A[st..end]*/
mergeSort(A, st, end):
  if (small):
    return

  mid = (end-start)/2

  mergeSort(A,st,mid)
  mergeSort(A,mid, end)

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

Divide

Conquer

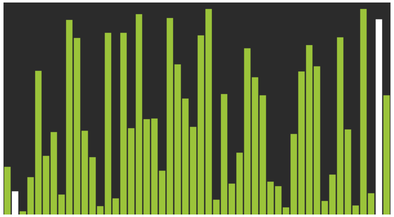
Combine

```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return

  mid = partition(A,st,end)

  quickSort(A,st,mid)
  quickSort(A,mid, end)
```

no op

Quicksort

Unsorted:

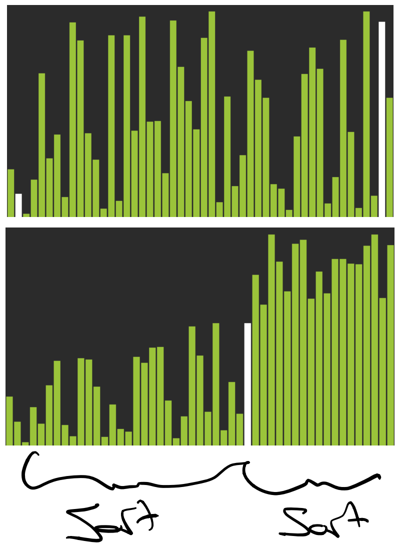


```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return

  mid = partition(A,st,end)

  quickSort(A,st,mid)

  quickSort(A,mid+1, end)
```

Quicksort

Unsorted:

Small things left big things right:

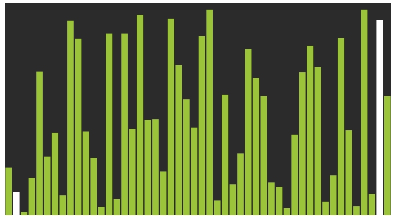



```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return

  mid = partition(A,st,end)

  quickSort(A,st,mid)

  quickSort(A,mid+1, end)
```

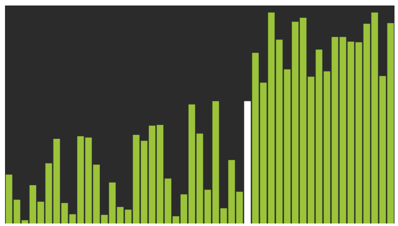
Quicksort

```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return
```

Unsorted:

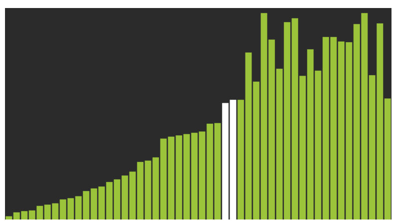


Small things left big things right:



← `mid = partition(A,st,end)`

Sort left things:



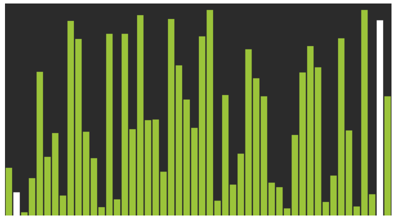
← `quickSort(A,st,mid)`

`quickSort(A,mid+1, end)`

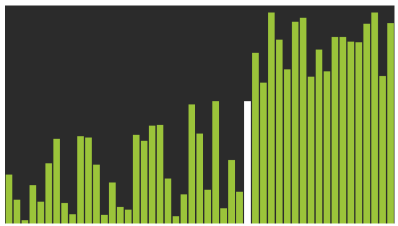
Quicksort

```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return
```

Unsorted:

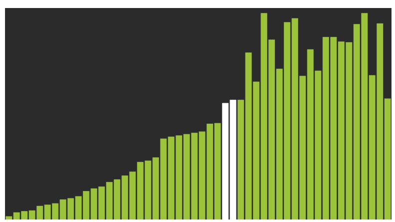


Small things left big things right:



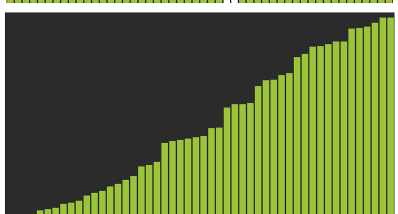
```
← mid = partition(A,st,end)
```

Sort left things:



```
← quickSort(A,st,mid)
```

Sort right things:



```
← quickSort(A,mid+1, end)
```

Quicksort

```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return


  mid = partition(A,st,end)



  quickSort(A,st,mid)



  quickSort(A,mid+1, end)
```

Quicksort

Key issues:

```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return

  mid = partition(A,st,end)

  quickSort(A,st,mid)

  quickSort(A,mid+1, end)
```

Quicksort

Key issues:

1. Picking the pivot

- First, middle, or last
- Median of first, middle, or last

```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return

  mid = partition(A,st,end)

  quickSort(A,st,mid)

  quickSort(A,mid+1, end)
```

Quicksort

Key issues:

1. Picking the pivot

- First, middle, or last
- Median of first, middle, or last

2. Implementing `partition`

```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return


  mid = partition(A,st,end)


  quickSort(A,st,mid)


  quickSort(A,mid+1, end)
```