CSCI 241

Lecture 6
Quicksort
Stability; Non-Comparison Sorts
Radix Sort

Announcements

- Quiz 1 is out on Gradescope - take it between 10am and 10pm today.

Goals:

- Thoroughly understand the mechanism of mergesort and quicksort.
- Be prepared to implement **merge** and **partition** helper methods.
- Know what it means for a sorting algorithm to be **stable**
- Understand the distinction between comparison and non-comparison sorts.
- Be prepared to implement radix sort.

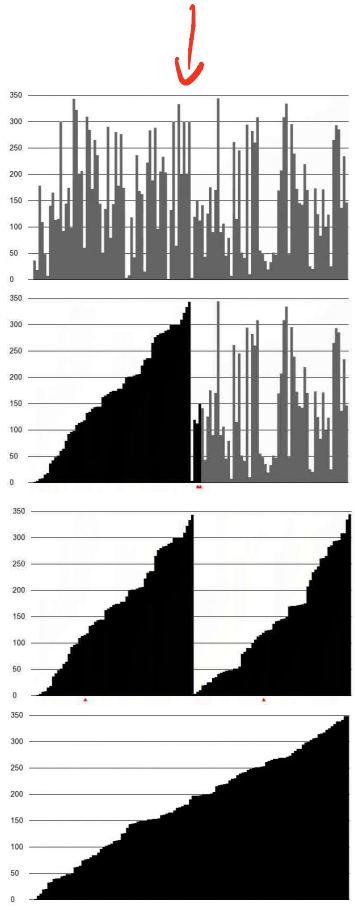
```
/** sort A[start..end] using mergesort */
mergeSort(A, start, end):
   if (end-start < 2):
      return
   mid = (end+start)/2

   mergeSort(A,start,mid)

   mergeSort(A,mid, end)

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

Divide

Conquer (left)

Conquer (right)

Combine



https://visualgo.net/bn/sorting

Mergesort vs Quicksort

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

  mid = (end+st)/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+1,end)
```

Mergesort vs Quicksort

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

  mid = (end+st)/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+1,end)
```

Mergesort vs Quicksort

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

  mid = (end+st)/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+1,end)
```

Mergesort vs Quicksort

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

  mid = (end+st)/2

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

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

Divide

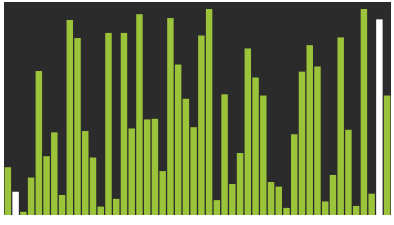
Conquer

Combine

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

  mid = partition(A,st,end)

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

Quicksort

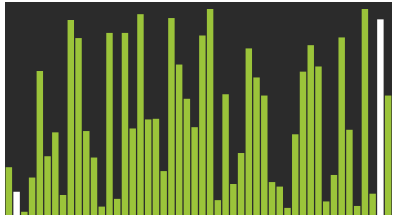
Unsorted:



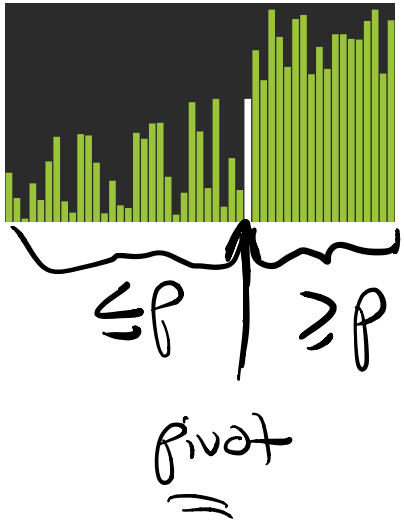
```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return

  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
```

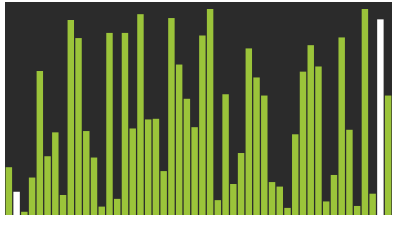
```
mid = partition(A,st,end)
```

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

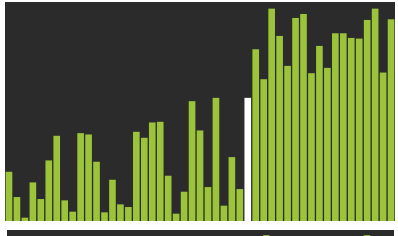
```
quickSort(A,mid+1, end)
```

Quicksort

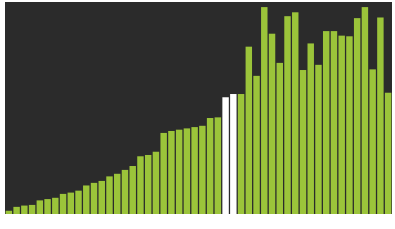
Unsorted:



Small things left
big things right:



Sort left things:



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

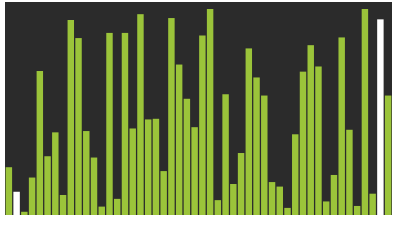
```
mid = partition(A,st,end)
```

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

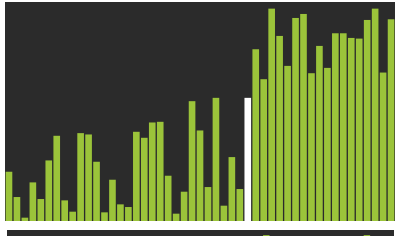
```
quickSort(A,mid+1, end)
```

Quicksort

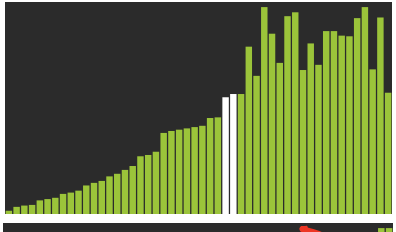
Unsorted:



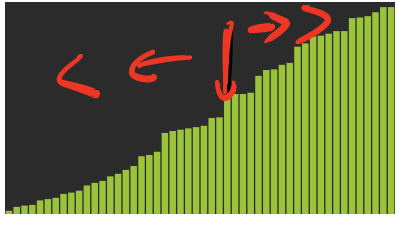
Small things left big things right:



Sort left things:



Sort right things:



```
/** 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

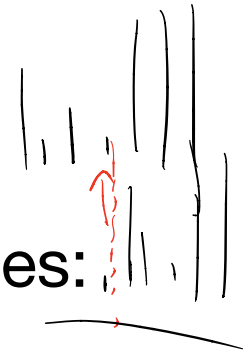
```
/** quicksort A[st..end]*/
quickSort(A, st, end):
  if (small):
    return

  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)
```

index of the pivot

```
  quickSort(A,st,mid)

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

Quicksort

Key issues:

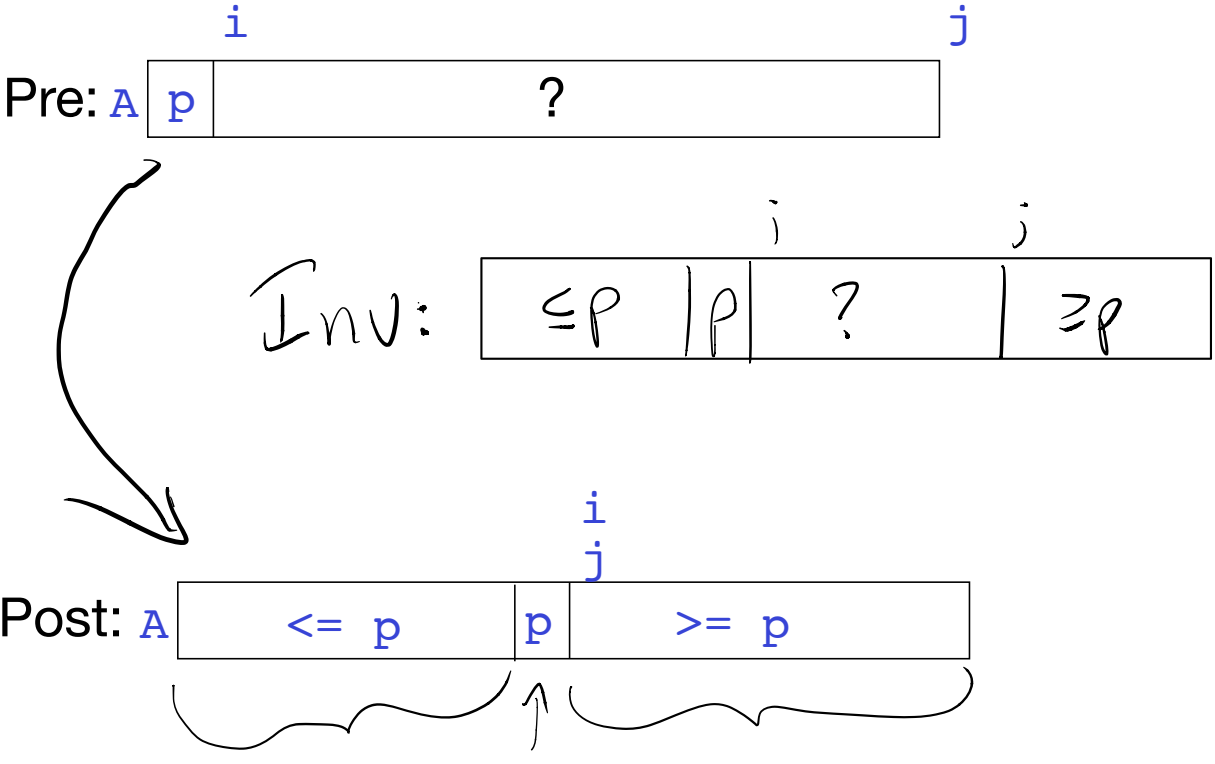
1. Picking the pivot

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

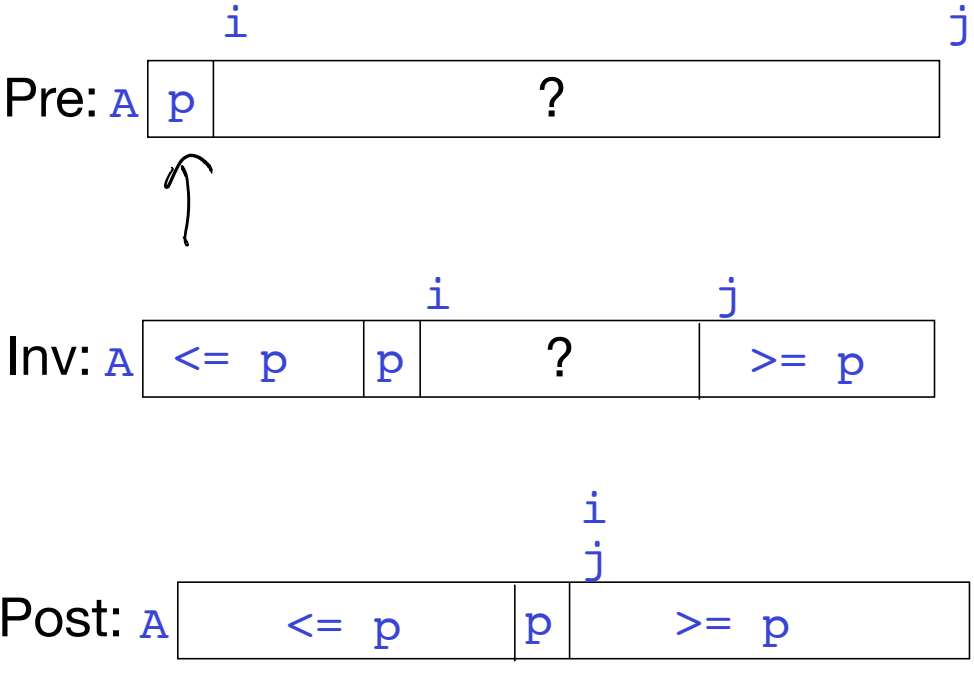
2. Implementing `partition`

```
/** 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)
```

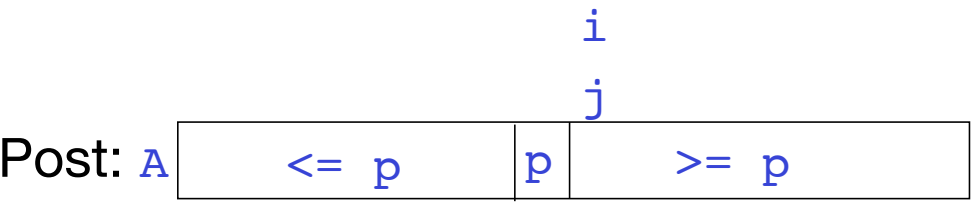
Implementing Partition



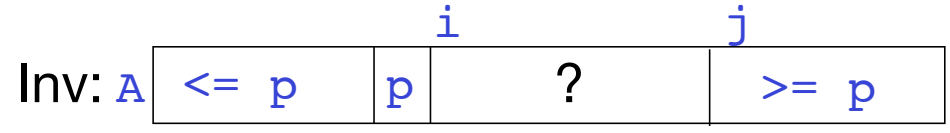
Implementing Partition



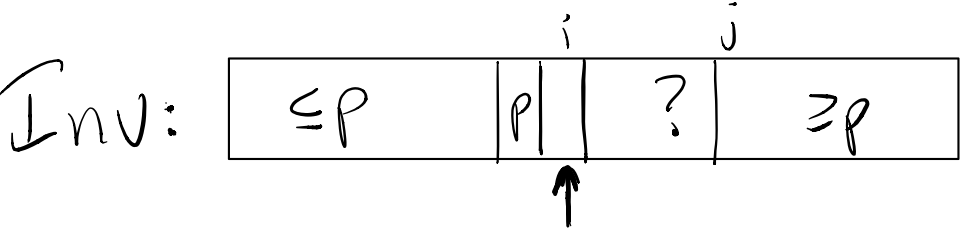
Implementing Partition

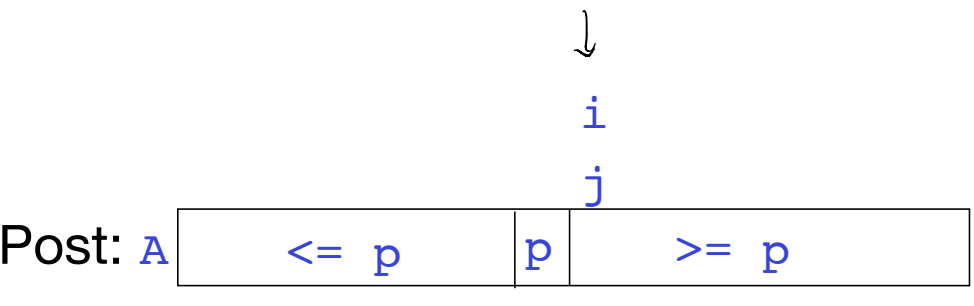


Implementing Partition




```
while i < j:
    if A[i] ≤ p:
        swap(A, i, i-1)
        i++;
    else:
        swap(A, i, j-1)
        j--
```





Questions?

Stability

Objects can be sorted on **keys** - **different** objects may have the same value.

A **stable** sort maintains the order of distinct elements with the same key.

- Example: sort a list of Student objects by first name only
- Example: sorting numbers on 10's place only

[6^{*} 2^{*} 6^{+} 2^{+} 3 4]

Stability

Objects can be sorted on **keys** - **different** objects may have the same value.

A **stable** sort maintains the order of distinct elements with the same key.

- Example: sort a list of Student objects by first name only
- Example: sorting numbers on 10's place only

$$[\ 6^{*} \quad 2^{*} \quad 6^{+} \quad 2^{+} \quad 3 \quad 4\ ]$$

$$[\ 61 \quad 21 \quad 63 \quad 23 \quad 35 \quad 48\ ]$$

Stability

A **stable** sort maintains the order of elements with the same value.

$$[\ 6^{*}\ \ 2^{*}\ \ 6^{+}\ \ 2^{+}\ \ 3\ \ 4\ ]$$

Stably sorted: $[\ 2^{*}\ \ 2^{+}\ \ 3\ \ 4\ \ 6^{*}\ \ 6^{+}]$

Unstably sorted: $[\ 2^{+}\ \ 2^{*}\ \ 3\ \ 4\ \ 6^{*}\ \ 6^{+}]$

Stability

A **stable** sort maintains the order of elements with the same value.

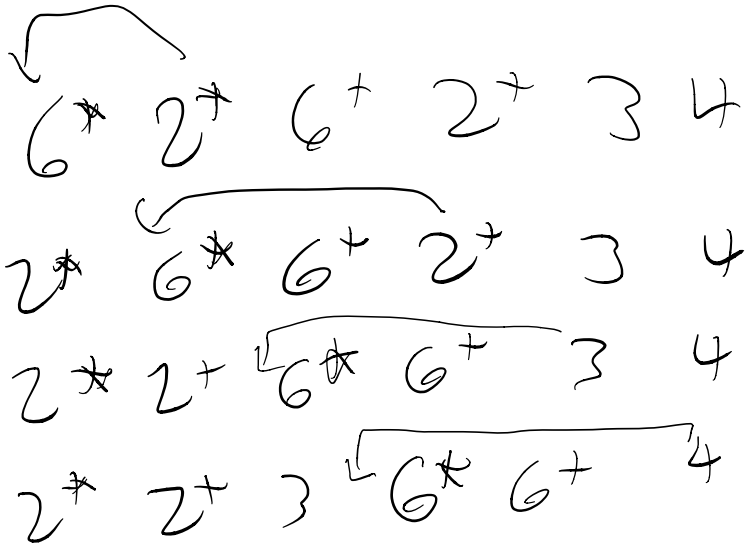
In groups: Sort this list using insertion and selection sort. Is either algorithm stable?

$$[\,6^{*}\;\; 2^{*}\;\; 6^{+}\;\; 2^{+}\;\; 3\;\; 4\,]$$

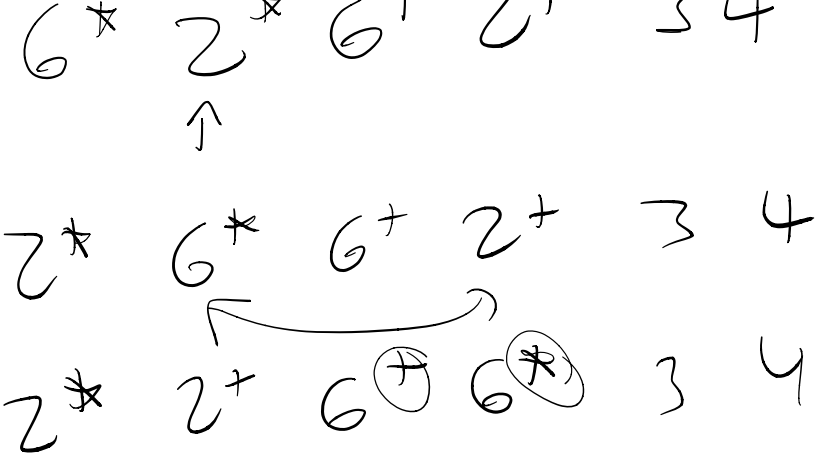
```
insertionSort(A):
 i = 0;
 while i < A.length:
   // push A[i] to
   //  its sorted position
   //  in A[0..i]
   // increment i
```

```
selectionSort(A):
  i = 0;
  while i < A.length:
    // find min of A[i..A.length]
    // swap it with A[i]
    // increment i
```

Ins



Sel



Comparison sorts operate by comparing pairs of elements.

Examples: all four sorts we've seen so far!

Comparison sorts operate by comparing pairs of elements.

Examples: all four sorts we've seen so far!

...is there any other way to do it?

Comparison sorts operate by comparing pairs of elements.

Examples: all four sorts we've seen so far!

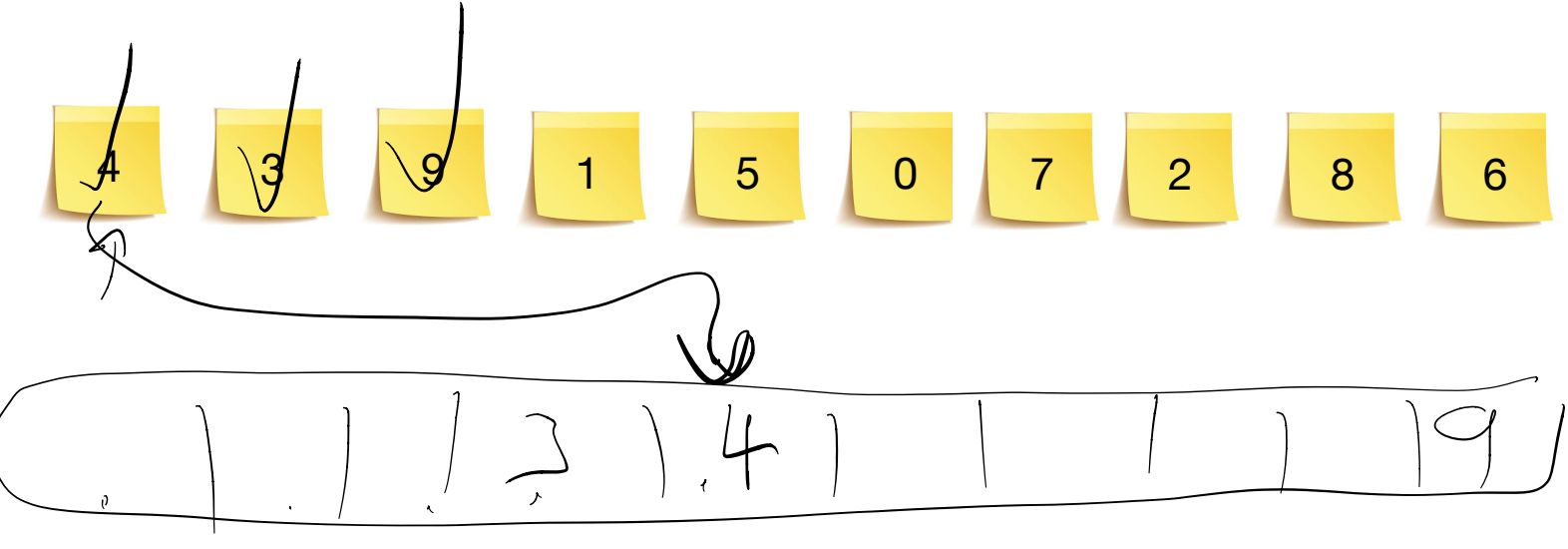
...is there any other way to do it?

How do you sort without comparing elements?

How do you sort things without comparing them?

Suppose I gave you 10 sticky notes with the digits 0 through 9.

What algorithm would you use to sort them?



How do you sort things without comparing them?

Suppose I gave you 10 sticky notes with the digits 0 through 9.

What algorithm would you use to sort them?

How many times did you need to look at each sticky note?

How do you sort things without comparing them?

Suppose I gave you 10 sticky notes with the digits 0 through 9.

What algorithm would you use to sort them?

How many times did you need to look at each sticky note?

What if there are duplicates?

LSD Radix Sort

```
/** least significant digit radix sort A */
LSDRadixSort(A):
max_digits = max # digits in any element of A
for d in 0..max_digits:
  do a stable sort of A on the dth least
  significant digit

// A is now sorted(!)
```

LSD Radix Sort

```
/** least significant digit radix sort A */
LSDRadixSort(A):
max_digits = max # digits in any element of A
for d in 0..max_digits:
  do a stable sort of A on the dth least
  significant digit

// A is now sorted(!)
```

LSD Radix Sort

```
/** least significant digit radix sort A */
LSDRadixSort(A):
max_digits = max # digits in any element of A
for d in 0..max_digits:
  do a stable sort of A on the dth least
  significant digit

// A is now sorted(!)
```

Don’t believe me? https://visualgo.net/en/sorting

LSD Radix Sort using queue buckets

Pseudocode from visualgo.net:

```
LSDRadixSort(A):
 create 10 buckets (queues) for each digit (0 to 9)
 for each digit (least- to most-significant):
   for each element in A:
     move element into its bucket based on digit
   for each bucket, starting from smallest digit
     while bucket is non-empty
       restore element to list
```

LSD Radix Sort using queue buckets

```
Pseudocode from visualgo.net:

LSDRadixSort(A):
 create 10 buckets (queues) for each digit (0 to 9)
 for each digit (least- to most-significant):
   for each element in A:
     move element into its bucket based on digit
   for each bucket, starting from smallest digit
     while bucket is non-empty
       restore element to list
```

LSD Intuition: sort on most-significant digit **last**; if tied, yield to the next most significant digit, and so on.
Only works because **stability** preserves orderings from less significant digits (previously sorted).

Exercise: Radix sort this

[7, 19, 21, 11, 14, 54, 1, 8]

Hint: [07, 19, 21, 11, 14, 54, 01, 08]

```
LSDRadixSort(A):
 create 10 buckets (queues) for each digit (0 to 9)
 for each digit (least- to most-significant):
   for each element in A:
     move element into its bucket based on digit
   for each bucket, starting from smallest digit
     while bucket is non-empty
       restore element to list
```

Exercise: Radix sort this

[07, 19, 61, 11, 14, 54, 01, 08]

Buckets on 1's place:

0	1	2	3	4	5	6	7	8	9

Sorted on 1's place:

Buckets on 10's place:

0	1	2	3	4	5	6	7	8	9

Sorted on 10's place: