Lecture 10
Sorting 1

EECS-214

The 1900 US Census

- Collect information about **every person** in the united states
  - Name
  - Age
  - Sex
  - Address
  - Race
  - Etc.
- Write it all down

- Generate **statistics**
  - How many people live in each state?
  - How many elderly are there in a given neighborhood?

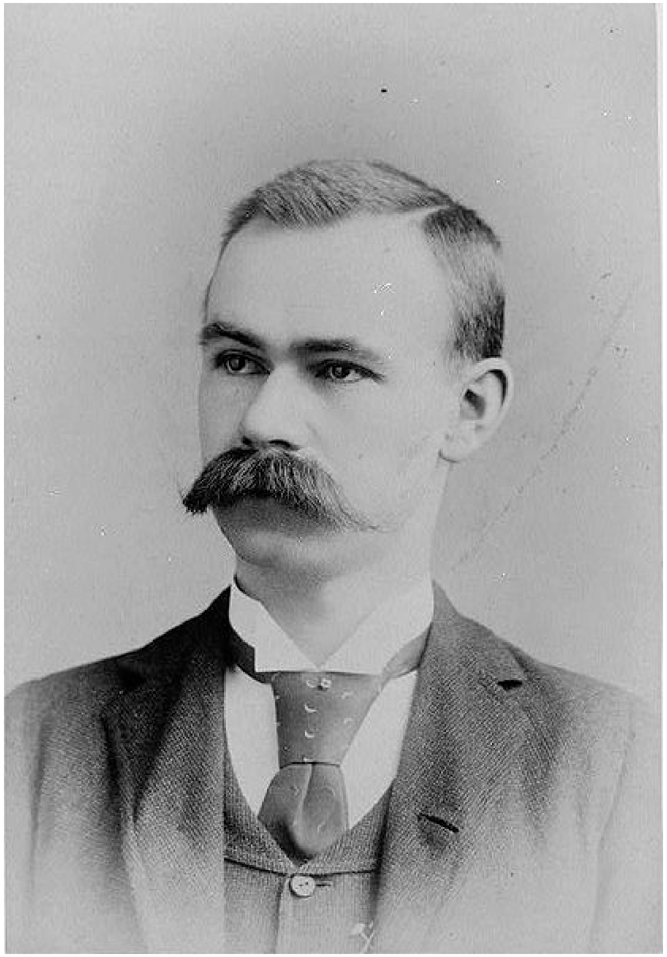
- How do you tally that sort of data **automatically**?

  (no computers in 1900)

Herman Hollerith

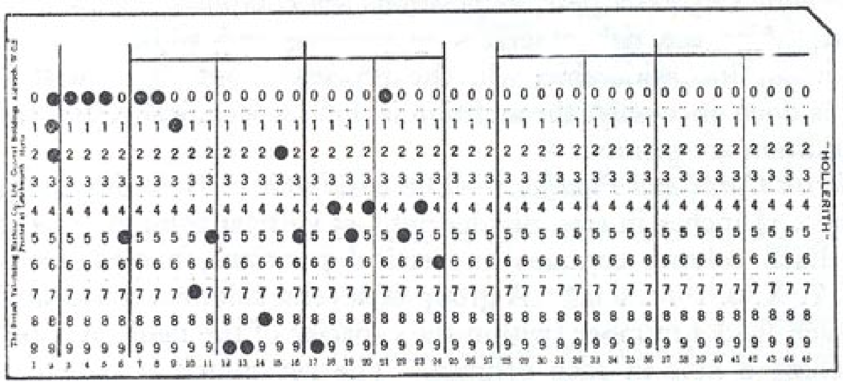
- Statistician and engineer working for the census bureau
  - Invented **punch cards**
  - And the technology for **processing** them **electronically**

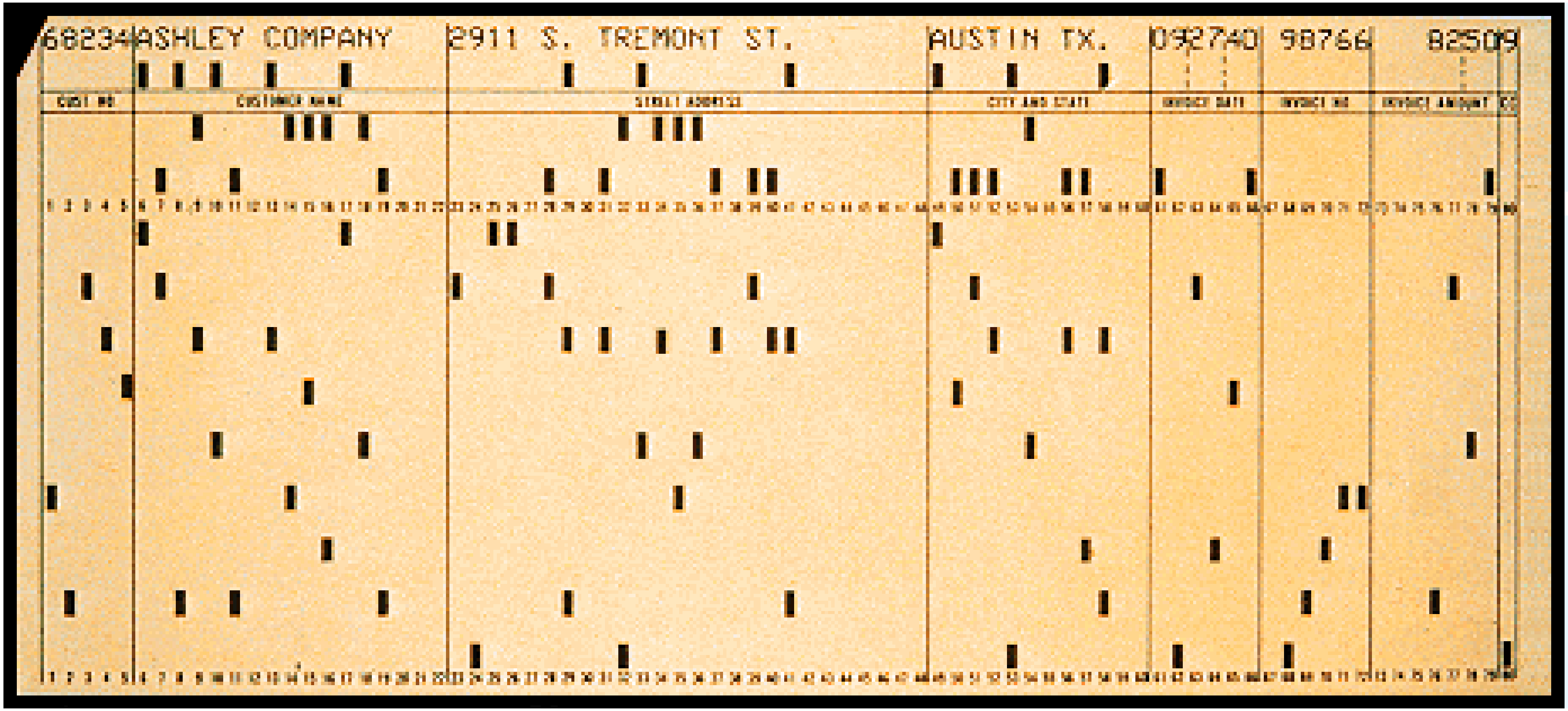
- So he **started a company** to make **punch card machines** for the government



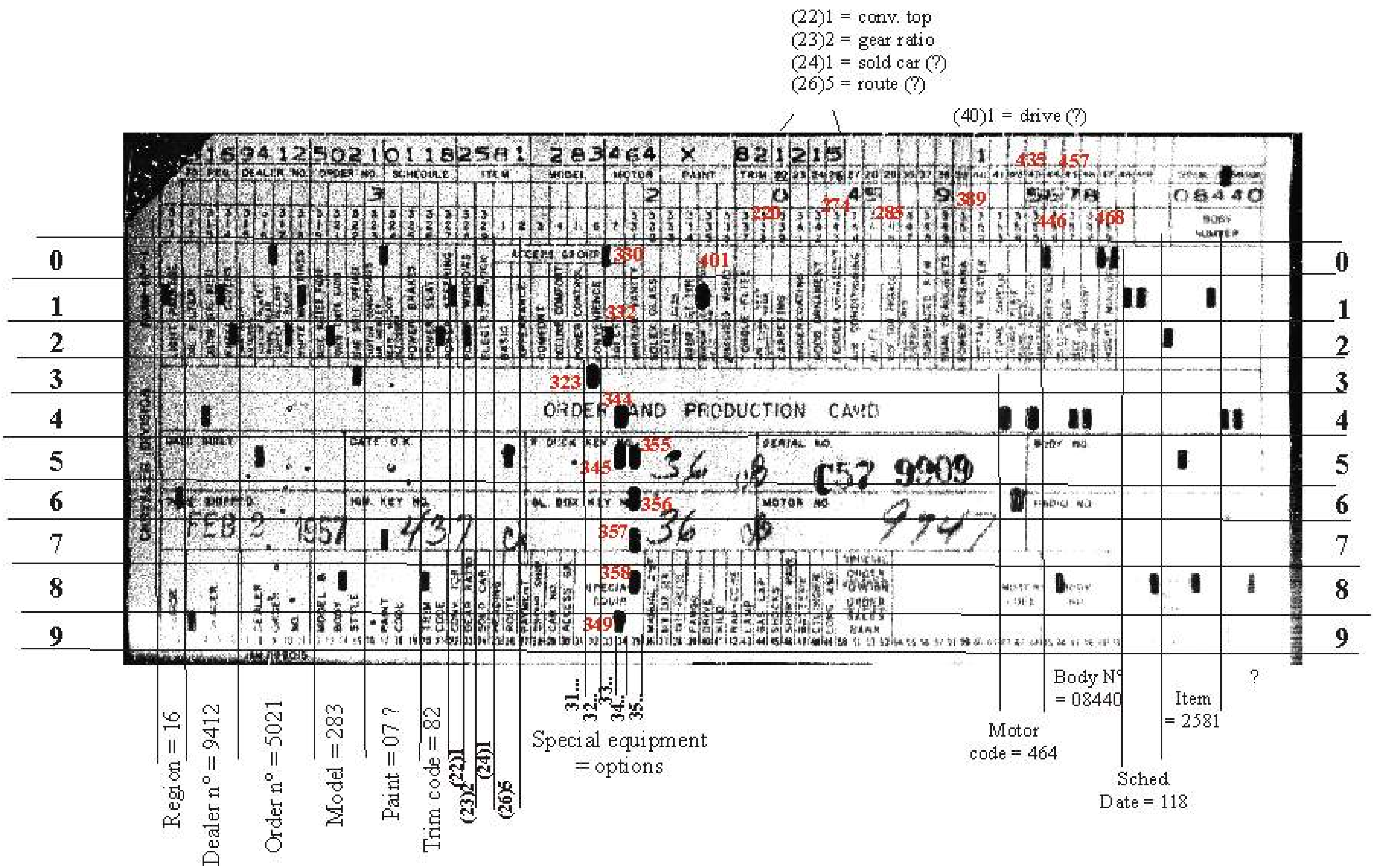
Hollerith cards (later form)

- One **card per record** in the database
- 80 columns per card
- Each column can be punched in one of 12 places
  - But originally only one hole per column





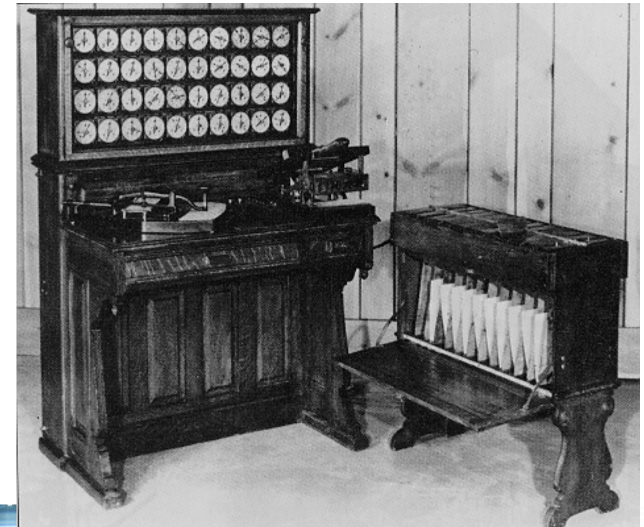
ID# Name Street Address City Date Invoice# $


(22)1 = conv. top
(23)2 = gear ratio
(24)1 = sold car (?)
(26)5 = route (?)
(40)1 = drive (?)
ORDER AND PRODUCTION CARD
DATE SHIPPED
FEB 2 1957
SERIAL NO.
C57 9909
MOTOR NO.
BODY NO.
RADIO NO.
BODY NUMBER
08440
SPECIAL EQUIP
Region = 16
Dealer n° = 9412
Order n° = 5021
Model = 283
Paint = 07 ?
Trim code = 82
Special equipment = options
Motor code = 464
Body N° = 08440
Sched Date = 118
Item = 2581
?

Punch card sorters

Hollerith developed a **tabulating machine** that would

- **Count** the number of cards with a given pattern of holds in them
- **Sort** the cards into bins based on which hole was punched in a given column



Multi-digit sorting

- What the #$% can you do with a machine that **sorts on just one column**?
- What if you have a bunch of cards with **3 digit numbers** punched on them and you want to sort them into order?

Unsorted numbers:

- 345
- 085
- 024
- 978
- 432
- 074
- 001
- 247
- 082

Radix sort:
The (arguably) first cool algorithm

- First sort them by their **last digit**
  - This is the basic operation of the card sorter

Unsorted numbers:

- 345
- 085
- 024
- 978
- 432
- 074
- 001
- 247
- 082

Radix sort:
The (arguably) first cool algorithm

- First sort them by their last digit
  - Great. They're sorted by the **last digit**. Now what?

Sorted numbers:

- **001**
- **432**
- **082**
- **024**
- **074**
- **345**
- **085**
- **247**
- **978**

Radix sort:
The (arguably) first cool algorithm

- First sort them by their last digit
- Now take **all the cards** from all the bins, **in order**

Sorted numbers:

- 001
- 432
- 082
- 024
- 074
- 345
- 085
- 247
- 978

Radix sort:
The (arguably) first cool algorithm

- First sort them by their last digit
- Now take all the cards from all the bins, in order
- And sort them by their **middle digit**
  - The card sorter preserves the order of cards with the same middle digit

Sorted numbers:

- 001
- 432
- 082
- 024
- 074
- 345
- 085
- 247
- 978

Radix sort:
The (arguably) first cool algorithm

- First sort them by their last digit
- Now take all the cards from all the bins, in order
- And sort them by their middle digit
  - Presto! The cards are now sorted by the **last two** digits

Sorted numbers:

- **001**
- **024**
- **432**
- **345**
- **247**
- **074**
- **978**
- **082**
- **085**

Radix sort:
The (arguably) first cool algorithm

- First sort them by their last digit
- Now take all the cards from all the bins, in order
- And sort them by their middle digit
- And, finally, resort by the **first digit**

Sorted numbers:

- 001
- 024
- 432
- 345
- 247
- 074
- 978
- 082
- 085

Radix sort:
The (arguably) first cool algorithm

- First sort them by their last digit
- Now take all the cards from all the bins, in order
- And sort them by their middle digit
- And, finally, resort by the first digit

Sorted numbers:

- **001**
- **024**
- **074**
- **082**
- **085**
- **247**
- **345**
- **432**
- **978**

Radix sort:
The (arguably) first cool algorithm

- First sort them by their last digit
- Now take all the cards from all the bins, in order
- And sort them by their middle digit
- And, finally, resort by the first digit
- They're now **properly sorted** by numerical value

Sorted numbers:

- 001
- 024
- 074
- 082
- 085
- 247
- 345
- 432
- 978

Analysis

- **How long** does Hollerith's algorithm take?
  - For n cards
  - With m digit numbers

- The loop **runs $\boldsymbol{m}$ times**
  - Each iteration **processes $\boldsymbol{n}$ cards**
    - Processing each card takes a constant amount of time
- So the total time is is $\boldsymbol{O(mn)}$
  - For small values of m, this is really great

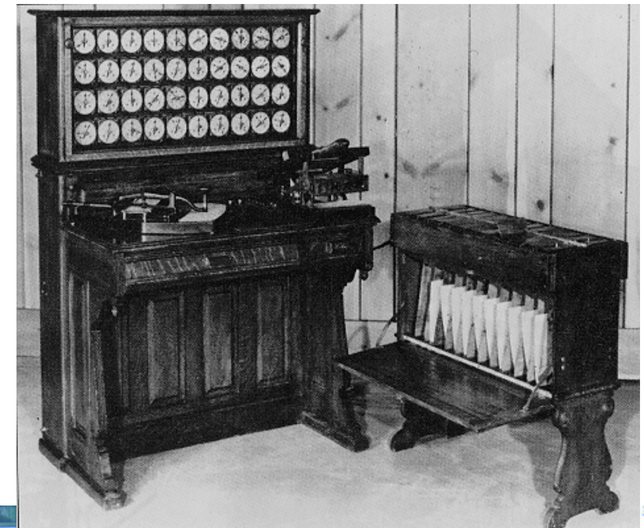
For each digit (in reverse order):

Sort cards into bins
(based only on that digit)

Merge cards into one stack

A company built on card sorting

- Hollerith's company was very successful



A company built on card sorting

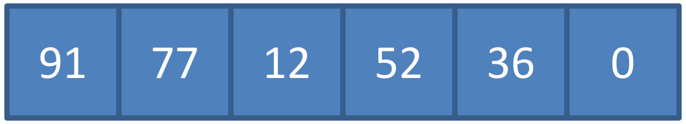
- Hollerith’s company was very successful

- It eventually merged with some other companies and renamed itself “**International Business Machines**” sorting is kind of a big deal in computing …

Sort algorithms

- Most of the time, radix sort isn't appropriate
  - Don't know number of digits in advance
  - Sorting things that aren't numbers to begin with

- Sorting algorithms have been **extensively studied** in computer science
  - And now **you** get to extensively study them too!



Sort algorithms

Most sort algorithms

- Take an **array** as input
- And **rewrite** the **array in place**
- To produce a sorted array with the same data

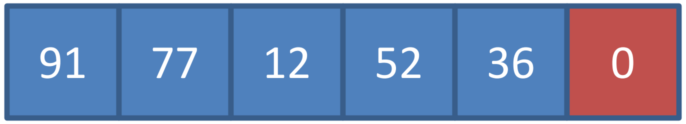
91	77	12	52	36	0

The trick is to design an algorithm that **scales** well as the **size of the array** increases

Selection sort

The **worst** possible sort algorithm

- Find the smallest element



Selection sort

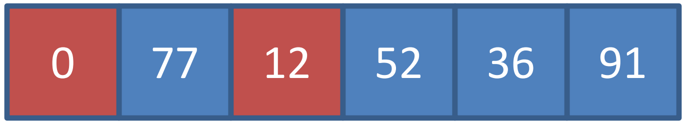
The worst possible sort algorithm

0	77	12	52	36	91

- Find the **smallest** element
- **Swap** it with the first element

Selection sort

The worst possible sort algorithm



- Find the **smallest** element
- **Swap** it with the first element
- Find the **next smallest** element

Selection sort

The worst possible sort algorithm

0	12	77	52	36	91

- Find the **smallest** element
- **Swap** it with the first element
- Find the **next smallest** element
- **Swap** it with the second element

Selection sort

The worst possible sort algorithm

0	12	77	52	36	91

- **Repeat** for the rest of the elements

Selection sort

The worst possible sort algorithm

0	12	36	52	77	91

- **Repeat** for the rest of the elements

Why does selection sort suck?

For an array with n elements

- The loop in selection sort runs n iterations
- So it **calls FindMin n times**
- FindMin does an exhaustive search of the remainder of the array

```
SelectionSort(a)
  for (i=0; i<a.Length; i++)
    index = FindMin(a, i)
    swap a[i] and a[index]

FindMin(a, start)
  minIndex = start
  minValue = a[start]
  for (i=start+1; i<a.Length; i++)
    if a[i]<minValue
      minValue = a[i]
      minIndex = I
  return minIndex
```

Why does selection sort suck?

- The first time FindMin is called, its loop runs for n **iterations**
- The second time for $n-1$ iterations
- The third, for $n-2$ iterations, etc.
- And so on …
- Until on the last call, it runs just once

```
SelectionSort(a)
  for (i=0; i<a.Length; i++)
    index = FindMin(a, i)
    swap a[i] and a[index]

FindMin(a, start)
  minIndex = start
  minValue = a[start]
  for (i=start+1; i<a.Length; i++)
    if a[i]<minValue
      minValue = a[i]
      minIndex = I
  return minIndex
```

Why does selection sort suck?

- The **total time** that loop runs to sort an n element array is therefore

$$1 + 2 + 3 + \ \ldots + n = \sum_{i=1}^{n} i$$

$$= \frac{n(n-1)}{2} = \frac{n^2 - n}{2}$$

$$= \boldsymbol{O}(\boldsymbol{n}^{\mathbf{2}})$$

Why does selection sort suck?

- Selection sort is an $O(n^2)$ algorithm
- In fact
  - It's not only $O(n^2)$ in the **worst case**
  - It's $O(n^2)$ in the **best case**
- Now that's one **crappy** sort algorithm!

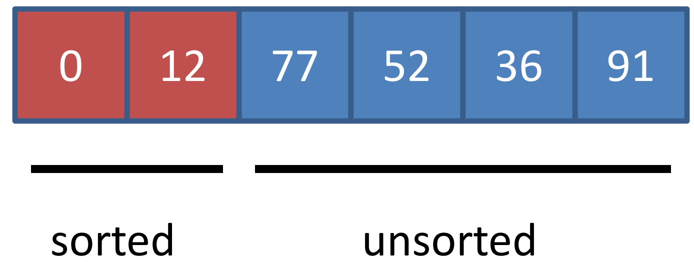
```
SelectionSort(a)
  for (i=0; i<a.Length; i++)
    index = FindMin(a, i)
    swap a[i] and a[index]

FindMin(a, start)
  minIndex = start
  minValue = a[start]
  for (i=start+1; i<a.Length; i++)
    if a[i]<minValue
      minValue = a[i]
      minIndex = I
  return minIndex
```

Insertion sort:
A somewhat less crappy algorithm

Basic idea

- **Divide** the array into two parts
  - The beginning of the array is **sorted**
  - The end of the array is **unsorted**

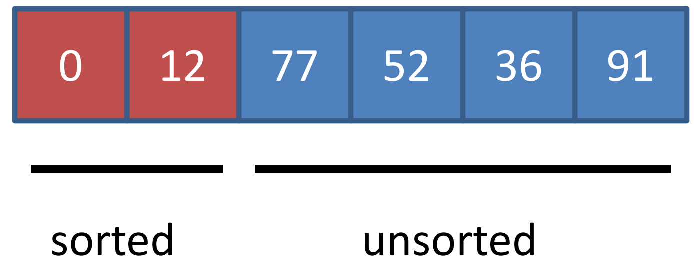


- Declare the first element of the array to be a one-element sorted section

- Repeat until done
  - **Move one element**
  - From the unsorted part
  - To the sorted part
  - Placing it in the **correct location**

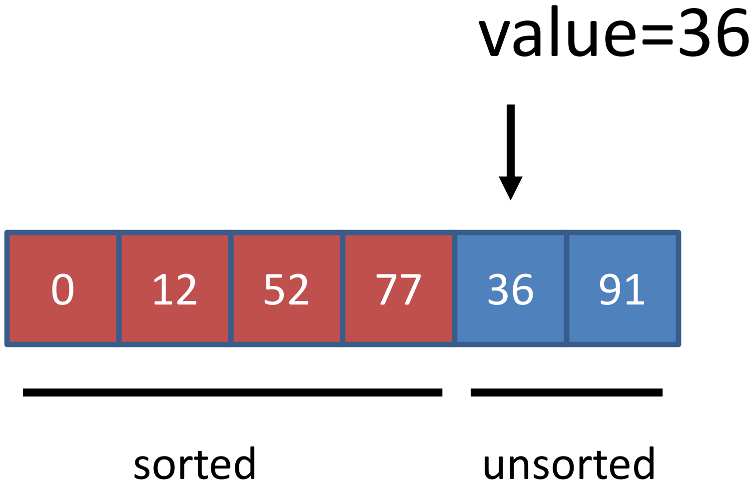
Insertion sort:
A somewhat less crappy algorithm

```
InsertionSort(a)
  for (i=1; i<a.length; i++)
     Insert(a, i, a[i])
```




```
Insert(a, position, value)
  // Copy elements forward
  // until you find one less than value
  for (i=pos-1; i≥0 && a[i]>value; i--)
    a[i+1] = a[i]
  // Put value in the hole you created
  a[i+1] = value
```

Inserting an element

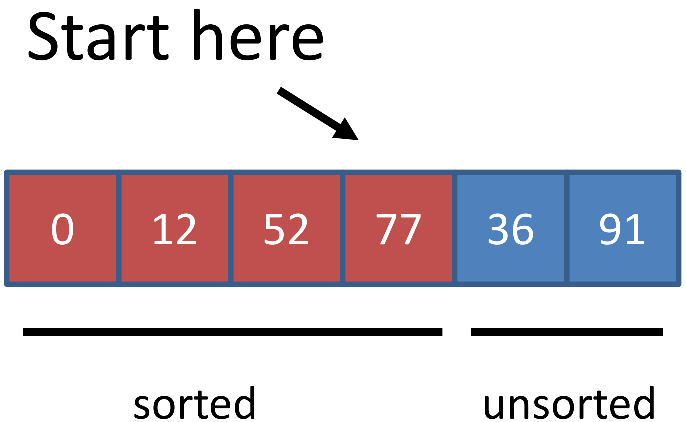



```
InsertionSort(a)
  for (i=1; i<a.length; i++)
    Insert(a, i, a[i])

Insert(a, position, value)
  // Copy elements forward
  // until you find one less than value
  for (i=pos-1; i≥0 && a[i]>value; i--)
    a[i+1] = a[i]
  // Put value in the hole you created
  a[i+1] = value
```

Inserting an element

value=36

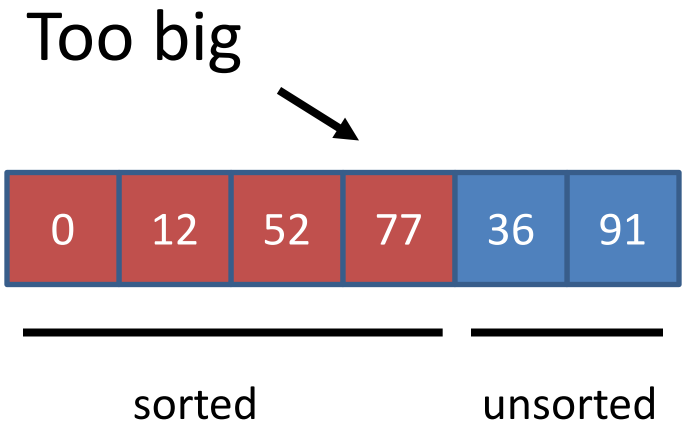



```
InsertionSort(a)
  for (i=1; i<a.length; i++)
     Insert(a, i, a[i])

Insert(a, position, value)
  // Copy elements forward
  // until you find one less than value
  for (i=pos-1; i≥0 && a[i]>value; i--)
    a[i+1] = a[i]
  // Put value in the hole you created
  a[i+1] = value
```

Inserting an element

value=36

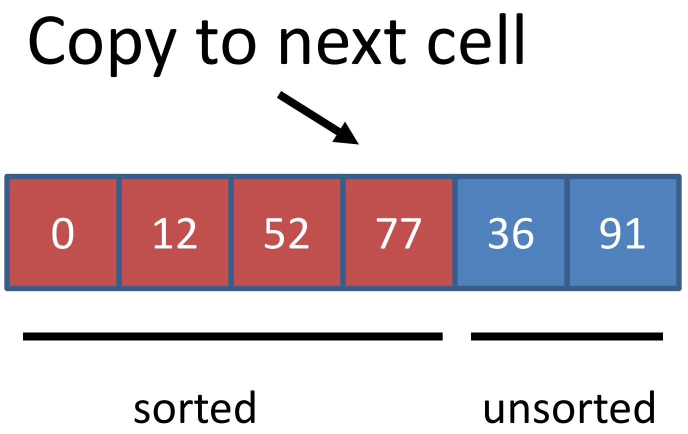



```
InsertionSort(a)
  for (i=1; i<a.length; i++)
     Insert(a, i, a[i])

Insert(a, position, value)
  // Copy elements forward
  // until you find one less than value
  for (i=pos-1; i≥0 && a[i]>value; i--)
    a[i+1] = a[i]
  // Put value in the hole you created
  a[i+1] = value
```

Inserting an element

value=36



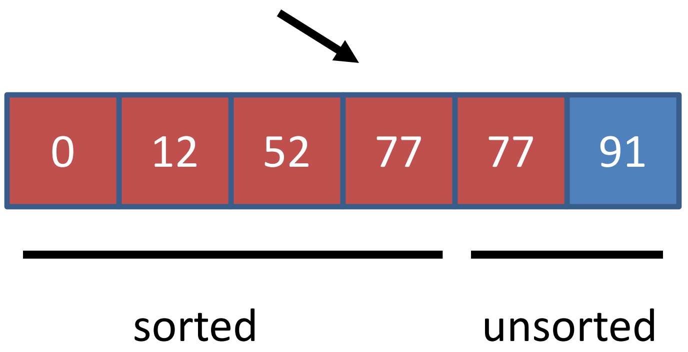

```
InsertionSort(a)
  for (i=1; i<a.length; i++)
    Insert(a, i, a[i])

Insert(a, position, value)
  // Copy elements forward
  // until you find one less than value
  for (i=pos-1; i≥0 && a[i]>value; i--)
    a[i+1] = a[i]
  // Put value in the hole you created
  a[i+1] = value
```

Inserting an element

value=36

Copy to next cell



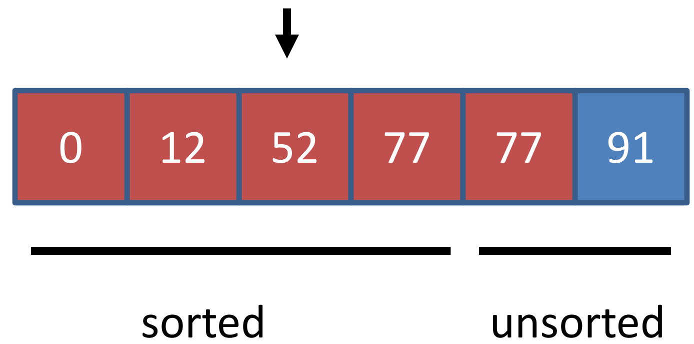

```
InsertionSort(a)
  for (i=1; i<a.length; i++)
    Insert(a, i, a[i])

Insert(a, position, value)
  // Copy elements forward
  // until you find one less than value
  for (i=pos-1; i≥0 && a[i]>value; i--)
    a[i+1] = a[i]
  // Put value in the hole you created
  a[i+1] = value
```

Inserting an element

value=36

Try the previous cell

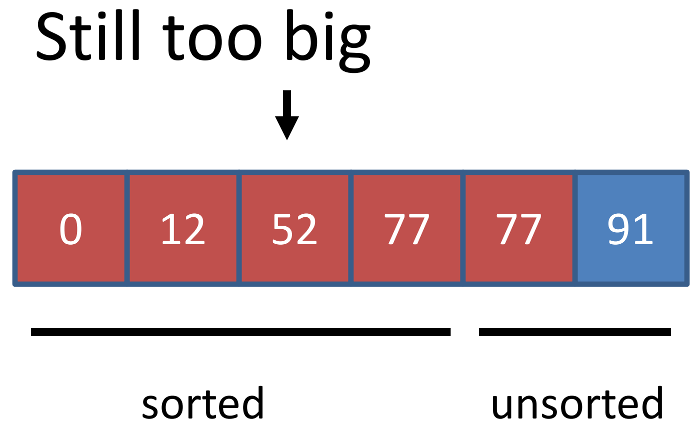



```
InsertionSort(a)
  for (i=1; i<a.length; i++)
    Insert(a, i, a[i])

Insert(a, position, value)
  // Copy elements forward
  // until you find one less than value
  for (i=pos-1; i≥0 && a[i]>value; i--)
    a[i+1] = a[i]
  // Put value in the hole you created
  a[i+1] = value
```

Inserting an element

value=36



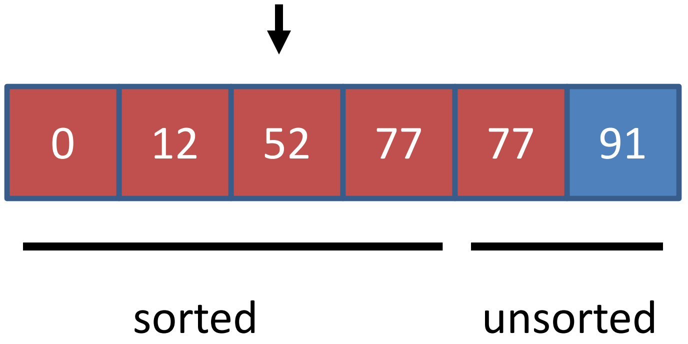

```
InsertionSort(a)
  for (i=1; i<a.length; i++)
     Insert(a, i, a[i])

Insert(a, position, value)
  // Copy elements forward
  // until you find one less than value
  for (i=pos-1; i≥0 && a[i]>value; i--)
    a[i+1] = a[i]
  // Put value in the hole you created
  a[i+1] = value
```

Inserting an element

value=36

Copy

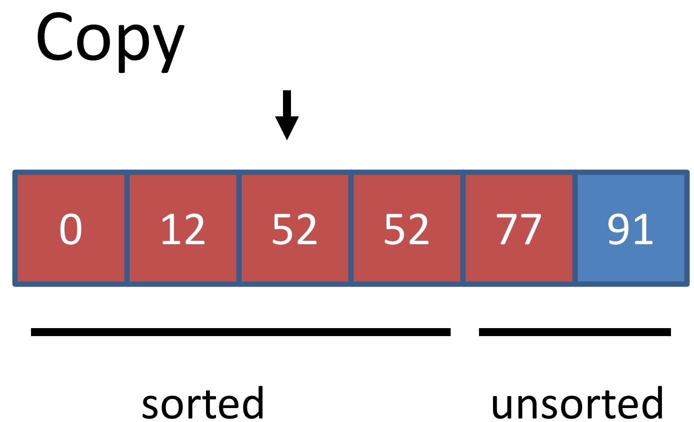



```
InsertionSort(a)
  for (i=1; i<a.length; i++)
    Insert(a, i, a[i])

Insert(a, position, value)
  // Copy elements forward
  // until you find one less than value
  for (i=pos-1; i≥0 && a[i]>value; i--)
    a[i+1] = a[i]
  // Put value in the hole you created
  a[i+1] = value
```

Inserting an element

value=36



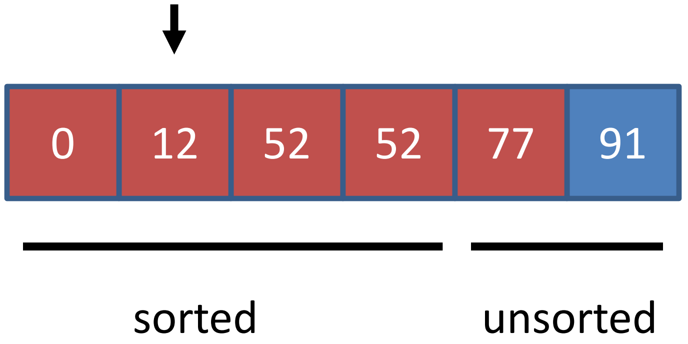

```
InsertionSort(a)
  for (i=1; i<a.length; i++)
     Insert(a, i, a[i])
```

```
Insert(a, position, value)
  // Copy elements forward
  // until you find one less than value
  for (i=pos-1; i≥0 && a[i]>value; i--)
    a[i+1] = a[i]
  // Put value in the hole you created
  a[i+1] = value
```

Inserting an element

value=36

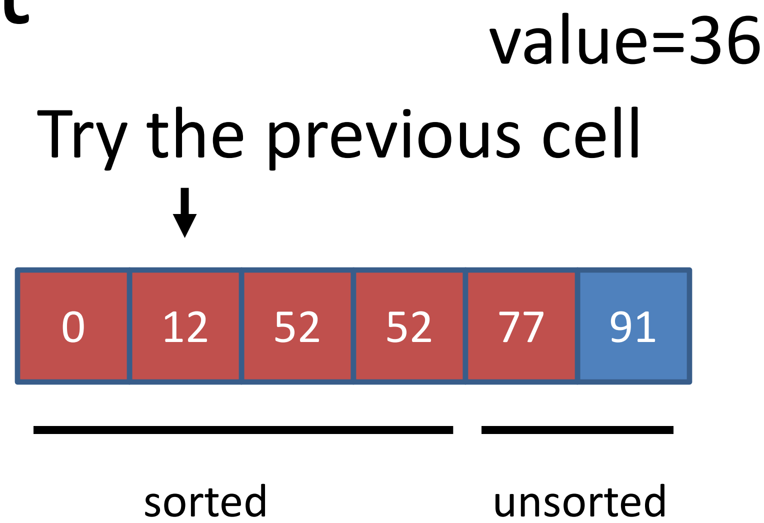
Try the previous cell




```
InsertionSort(a)
  for (i=1; i<a.length; i++)
     Insert(a, i, a[i])

Insert(a, position, value)
  // Copy elements forward
  // until you find one less than value
  for (i=pos-1; i≥0 && a[i]>value; i--)
    a[i+1] = a[i]
  // Put value in the hole you created
  a[i+1] = value
```

Inserting an element

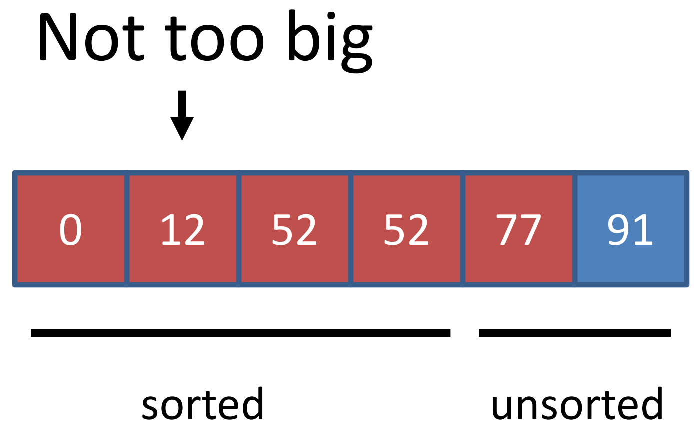



```
InsertionSort(a)
  for (i=1; i<a.length; i++)
     Insert(a, i, a[i])

Insert(a, position, value)
  // Copy elements forward
  // until you find one less than value
  for (i=pos-1; i≥0 && a[i]>value; i--)
    a[i+1] = a[i]
  // Put value in the hole you created
  a[i+1] = value
```

Inserting an element

value=36



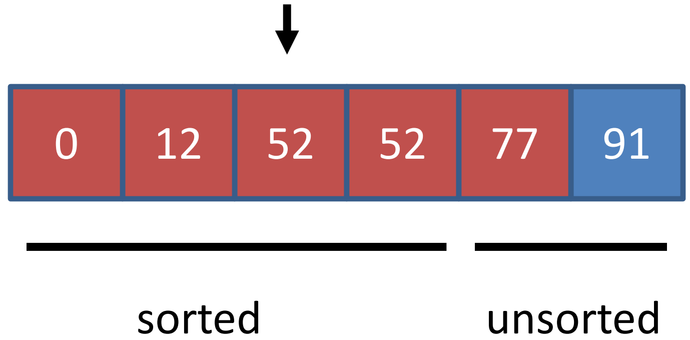

```
InsertionSort(a)
  for (i=1; i<a.length; i++)
    Insert(a, i, a[i])
```

```
Insert(a, position, value)
  // Copy elements forward
  // until you find one less than value
  for (i=pos-1; i≥0 && a[i]>value; i--)
    a[i+1] = a[i]
  // Put value in the hole you created
  a[i+1] = value
```

Inserting an element

value=36

Place value in **next** cell



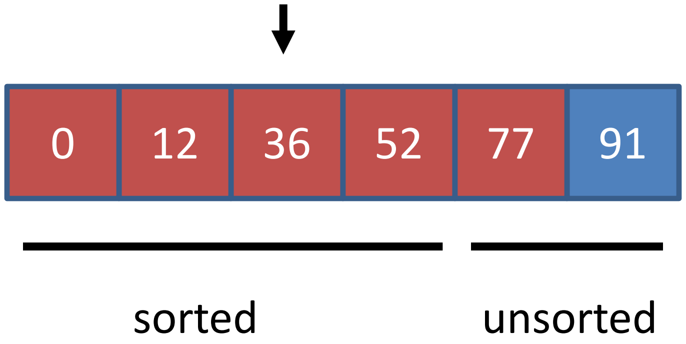

```
InsertionSort(a)
  for (i=1; i<a.length; i++)
     Insert(a, i, a[i])

Insert(a, position, value)
  // Copy elements forward
  // until you find one less than value
  for (i=pos-1; i≥0 && a[i]>value; i--)
    a[i+1] = a[i]
  // Put value in the hole you created
  a[i+1] = value
```

Inserting an element

value=36

Finished insertion!




```
InsertionSort(a)
  for (i=1; i<a.length; i++)
    Insert(a, i, a[i])
```

```
Insert(a, position, value)
  // Copy elements forward
  // until you find one less than value
  for (i=pos-1; i≥0 && a[i]>value; i--)
    a[i+1] = a[i]
  // Put value in the hole you created
  a[i+1] = value
```

Worst-case analysis

- The **worst case** is when the array is in **reverse order**
  - Last element should be first
  - First element should be last
- Insert's loop then runs
  - 1 iteration the first time
  - 2 iterations the second time
  - 3 iterations the third time
  - Etc.
- Execution is **quadratic time**
  - I.e. $O(n^2)$

```
InsertionSort(a)
  for (i=1; i<a.length; i++)
     Insert(a, i, a[i])

Insert(a, position, value)
  for (i=pos-1; i≥0 && a[i]>value; i--)
    a[i+1] = a[i]
  a[i+1] = value
```

Best-case analysis

- The **best case** is when the array is **already sorted**
  - Each element is already in its proper place
  - Insert’s loop runs for 0 iterations each time
- Execution is **linear time**
  - i.e. $O(n)$
  - Yay!

```
InsertionSort(a)
  for (i=1; i<a.length; i++)
    Insert(a, i, a[i])

Insert(a, position, value)
  for (i=pos-1; i≥0 && a[i]>value; i--)
    a[i+1] = a[i]
  a[i+1] = value
```

Average-case analysis

- Sadly, the average-case performance is **still quadratic** – $O(n^2)$
  – Although we won't prove it in class

```
InsertionSort(a)
  for (i=1; i<a.length; i++)
    Insert(a, i, a[i])

Insert(a, position, value)
  for (i=pos-1; i≥0 && a[i]>value; i--)
    a[i+1] = a[i]
  a[i+1] = value
```

Performance

- Insertion sort is usually a bad idea
- However, its has its applications
  - If n **is small**
    - Easy to write
    - Pretty fast
  - If you know the data will already be **mostly sorted**
    - Will run close to linear time

```
InsertionSort(a)
  for (i=1; i<a.length; i++)
    Insert(a, i, a[i])

Insert(a, position, value)
  for (i=pos-1; i≥0 && a[i]>value; i--)
    a[i+1] = a[i]
  a[i+1] = value
```

Bubble sort (aka "bogosort"):

When you don't care enough to send the very best

- Algorithm:
  - **Scan** the array, **comparing adjacent** elements
  - If a pair is **out of order swap them**
  - **Repeat** until you make a whole scan of the array **without swapping**
- Performance
  - Bad: $\boldsymbol{O(n^2)}$ **worst case** and **average case**
    - $O(n)$ best case
    - But generally still slower than insertion sort in practice
- Only saving grace is that it's easy to implement
  - So sometimes used for small values of n in performance non-critical situations

```
BubbleSort(a)
 swapped = true
 while swapped
   swapped = false
   for (i=0; i<a.Length-1; i++)
     if (a[i]>a[i+1])
       swap a[i] and a[i+1]
       swapped = true
```

next: sorts that don’t suck