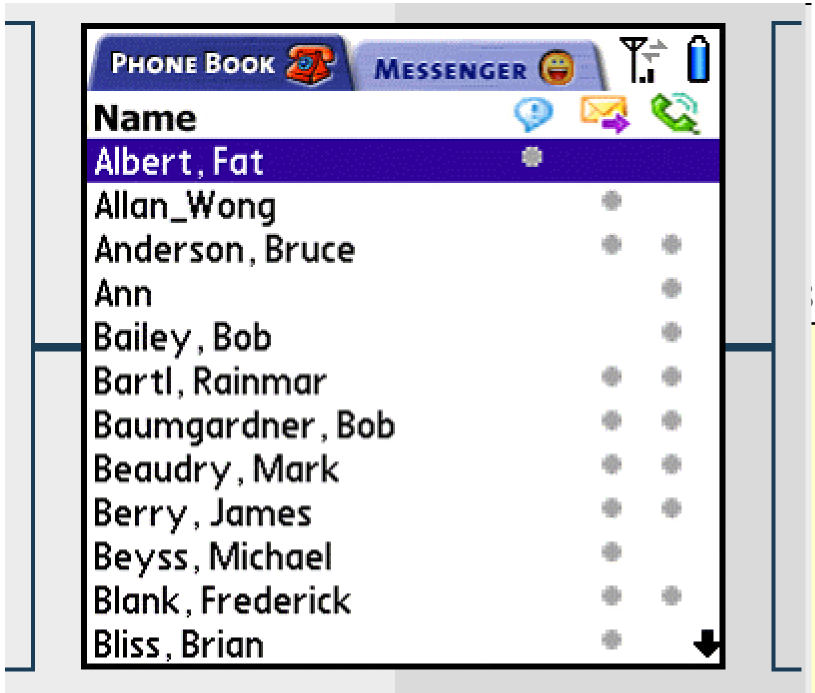
- Previous Lecture:
  - Recursion (Ch. 14)

- Today, Lecture 27:
  - Algorithms for sorting and efficiency analysis (Ch. 8)
    - Insertion Sort algorithm
    - See *Insight* §8.2 for the Bubble Sort algorithm
  - Algorithms for searching and analysis (Ch. 9)
    - Linear search (review)
    - Binary search

- Announcements:
  - Test 2B submissions due today, 4:30pm EDT
  - Since Tues 5/12 is the last day of classes, the Tues discussion sections will be converted to open office hrs. All students are welcome (Zoom links will be posted to Canvas).
  - Project 6 due Tues 11pm EDT. Remember academic integrity!
  - Regular office/consulting hours end on Tues. See Canvas and course website for Study period office/consulting hours
  - Final exam: “2hr” take-home, 48hr submission window. Mon, 5/18, 9am
  - Please complete course evaluations – worth extra point on Final

Sorting data allows us to search more easily



Boston Marathon Top Women Finishers

			Official Time	State	Country	Ctz
			2:25:25		ETH	
			2:25:27		RUS	
			2:26:34		KEN	
			2:28:12		LAT	
			2:29:48		ETH	
			2:30:52		ITA	
7	F12	Olaru, Nuta	2:33:56		ROM	
8	F6	Guta, Robe Tola	2:34:37		ETH	
9	F1	Grigoryeva, Lidiya	2:35:37		RUS	
	F35	Hood, Stephanie A.	2:44:44	IL	USA	CAN
	F14	Robson, Denise C.	2:45:54	NS	CAN	
	F11	Chemjor, Magdaline	2:46:25		KEN	
	F101	Sultanova-Zhdanova, Firaya	2:47:17	FL	USA	RUS
	F15	Mayger, Eliza M.	2:47:36		AUS	
	F24	Anklam, Ashley A.	2:48:43	MN	USA	

Name	Score	Grade
Jorge	92.1	
Ahn	91.5	
Oluban	90.6	
Chi	88.9	
Minale	88.1	
Bell	87.3	

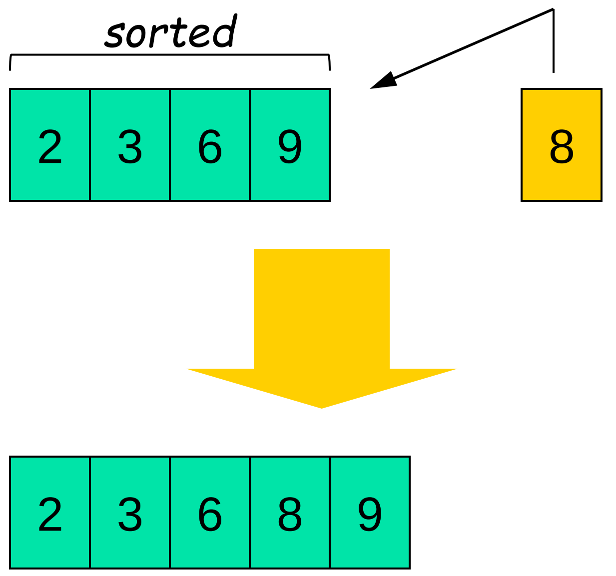
There are many algorithms for sorting

- Insertion Sort (to be discussed today)
- Bubble Sort (read *Insight* §8.2)
- Merge Sort (to be discussed next lecture)
- Quick Sort (a variant used by Matlab's built-in `sort` function)

- Each has advantages and disadvantages. Some algorithms are faster (time-efficient) while others are memory-efficient
- *Great opportunity for learning how to analyze programs and algorithms!*

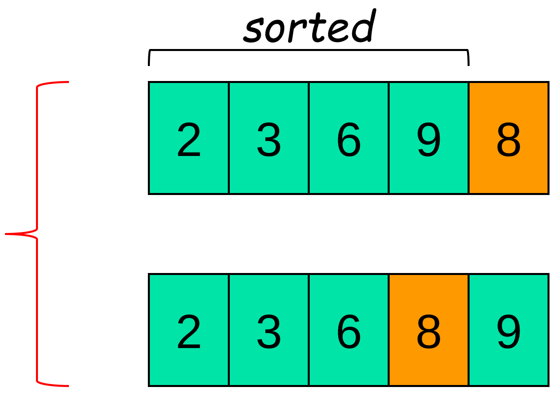
The Insertion Process

- Given a sorted array x, insert a number y such that the result is sorted



Insertion

one insert process



Insert 8 into the sorted segment

Just swap 8 & 9

Insertion

2	3	6	9	8

2	3	6	8	9

sorted

2	3	6	8	9	4

Insert 4 into the sorted segment

Insertion

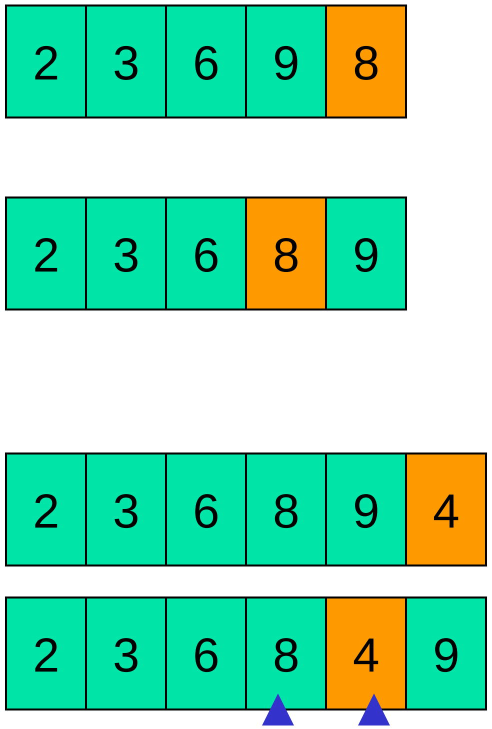
2	3	6	9	8

2	3	6	8	9

2	3	6	8	9	4

Compare adjacent components:
swap 9 & 4

Insertion



Compare adjacent components:
swap 8 & 4

Insertion

2	3	6	9	8

2	3	6	8	9

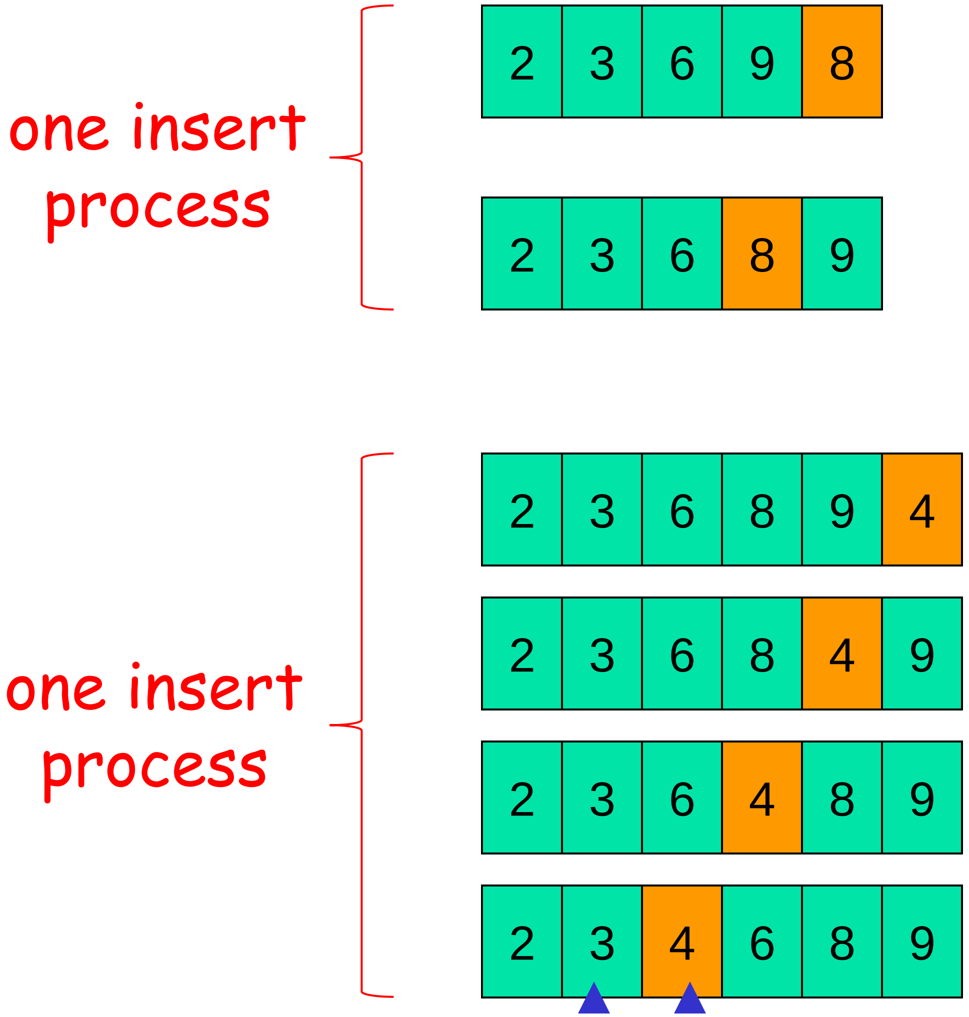
2	3	6	8	9	4

2	3	6	8	4	9

2	3	6	4	8	9

Compare adjacent components:
swap 6 & 4

Insertion



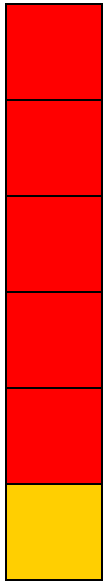
Compare adjacent components:
DONE! No more swaps.

See function **`Insert`** for the insert process

Sort vector **`x`** using the Insertion Sort algorithm

Need to start with a *sorted* subvector. How do you find one?

`x`



Length 1 subvector is "sorted"

Insert **`x(2)`**: **`x(1:2) = Insert(x(1:2))`**

Insert **`x(3)`**: **`x(1:3) = Insert(x(1:3))`**

Insert **`x(4)`**: **`x(1:4) = Insert(x(1:4))`**

Insert **`x(5)`**: **`x(1:5) = Insert(x(1:5))`**

Insert **`x(6)`**: **`x(1:6) = Insert(x(1:6))`**

`insertionSortSimple.m`

Contract between Insert and InsertionSort

Insert

- Assumes all but the last element of x is already sorted
- Returns a fully-sorted array (one more element sorted than given)

therefore

InsertionSort (driver)

- Must only call Insert() on a subarray with a pre-sorted prefix
- Has a bigger pre-sorted subarray to pass to Insert() next time – progress is made each iteration

Size of sorted prefix grows each time.
When it equals the size of the original array, the task is done

How much “work” is insertion sort?

- In the worst case, make k comparisons to insert an element in a sorted array of k elements.

Insertion

one insert process

2	3	6	9	8

2	3	6	8	9

Insert into sorted array of length 4

one insert process

2	3	6	8	9	4

2	3	6	8	4	9

2	3	6	4	8	9

2	3	4	6	8	9

Insert into sorted array of length 5

How much "work" is insertion sort?

- In the worst case, make k comparisons to insert an element in a sorted array of k elements. For an array of length N:

$$1 + 2 + \ldots + (N\text{-}1) = \frac{N(N-1)}{2}\text{, say } N^2 \text{ for big } N$$

`InsertionSort.m`

Checkpoint question: N^2 performance

Suppose it takes 5ms to sort an array with 100 elements using Insertion Sort. How long would you expect sorting 1000 elements to take?

A. 25ms

B. 50ms

C. 500ms

D. 5000ms

E. 1e6 ms

Efficiency considerations

- Worst case, best case, average case
- Use of subfunction incurs an "overhead"
- Memory use and access

- Example: Rather than directing the *insert* process to a subfunction, have it done "in-line."
- Also, Insertion sort can be done "in-place," i.e., using "only" the memory space of the original vector.

```
function x = InsertionSortInplace(x)
% Sort vector x in ascending order with insertion sort

n = length(x);
for i= 1:n-1
      % Sort x(1:i+1) given that x(1:i) is sorted















end
```

```
function x = InsertionSortInplace(x)
% Sort vector x in ascending order with insertion sort

n = length(x);
for i= 1:n-1
      % Sort x(1:i+1) given that x(1:i) is sorted
      j= i;

      while

          % swap x(j+1) and x(j)




          j= j-1;

      end
end
```

Do this for review later!

A note on optimization

- "Inlining" multiple pieces of an algorithm should *not* be your go-to strategy
  - It's easier to understand (and verify) small pieces that do a simple task than monolithic code that does a complicated task
    - Better communication, less buggy
  - Hard to predict when it will actually be faster
    - Large code has a performance cost in addition to a maintenance cost
    - Measuring performance not as easy as it sounds
  - Compilers can do this automatically
    - Auto-inlining will reveal opportunities for in-place array edits

Sort an array of objects

- Given x, a 1-d array of Interval references, sort x according to the widths of the Intervals from narrowest to widest
- Use the insertion sort algorithm
- How much of our code needs to be changed?

A. No change

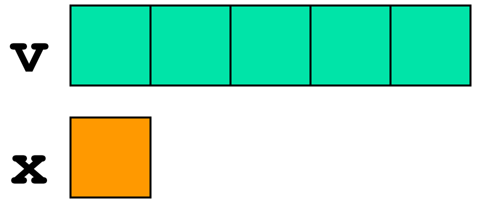
B. One statement

C. About half the code

D. Most of the code

Searching for an item in an unorganized collection?

- May need to look through the whole collection to find the target item
- E.g., find value x in vector v



- Linear search

```
% Linear Search
% f is index of first occurrence
%    of value x in vector v.
% f is -1 if x not found.
k= 1;
while  k<=length(v) && v(k)~=x
    k= k + 1;
end
if  k>length(v)
    f= -1; % signal for x not found
else
    f= k;
end
```

v	12	35	33	15	42	45
x	31					

```
% Linear Search
% f is index of first occurrence of value x in vector v.
% f is -1 if x not found.
k= 1;
while  k<=length(v) && v(k)~=x
    k= k + 1;
end
if  k>length(v)
    f= -1; % signal for x not found
else
    f= k;
end
```

`n` comparisons against the target are needed in worst case, `n=length(v)`.

```
% Linear Search
% f is index of first occurrence
%   of value x in vector v.
% f is -1 if x not found.
k= 1;
while  k<=length(v) && v(k)~=x
    k= k + 1;
end
if  k>length(v)
    f= -1; % signal for x no
else
    f= k;
end
```

Searching in a sorted list should require less work

v	12	15	33	35	42	45
x	31					

What if **v** is sorted?

An ordered (sorted) list

The Manhattan phone book has 1,000,000+ entries.

How is it possible to locate a name by examining just a <u>tiny</u>, <u>tiny</u> fraction of those entries?

Key idea of "phone book search": repeated halving

To find the page containing Pat Reef's number…

```
while  (Phone book is longer than 1 page)
       Open to the middle page.
       if  "Reef" comes before the first entry,
              Rip and throw away the 2nd half.
       else
              Rip and throw away the 1st half.
       end
end
```

What happens to the phone book length?

```
Original:      3000 pages
After 1 rip:   1500 pages
After 2 rips:   750 pages
After 3 rips:   375 pages
After 4 rips:   188 pages
After 5 rips:    94 pages
      :
After 12 rips:    1 page
```

Binary Search

Repeatedly halving the size of the "search space" is the main idea behind the method of binary search.

An item in a sorted array of length n can be located with just $\log_2 n$ comparisons.

"Savings" is significant!

n	log2(n)
100	7
1000	10
10000	13

What is true of the half we keep?

- Let L be the leftmost page we keep (may be 0, aka front cover)
- Let R be the page after the last one we keep (might be `length(v)+1`, aka back cover)
- Then the name we are looking for is >= the first name on page L, and < the first name on page R
- When only one page left (R = L+1),
  - If name is in book, it will be on page L
  - If name is not in book, it should be inserted after some names already on page L

Binary search: target x = 70

	1	2	3	4	5	6	7	8	9	10	11	12
v	12	15	33	35	42	45	51	62	73	75	86	98

L: 0

Mid: 6

R: 13

`v(Mid) <= x`

So throw away the left half...

Binary search: target x = 70

	1	2	3	4	5	6	7	8	9	10	11	12
v	12	15	33	35	42	45	51	62	73	75	86	98

L: 6

Mid: 9

R: 13

`x < v(Mid)`

So throw away the right half…

Binary search: target x = 70

	1	2	3	4	5	6	7	8	9	10	11	12
v	12	15	33	35	42	45	51	62	73	75	86	98

L: 6

Mid: 7

R: 9

`v(Mid) <= x`

So throw away the left half…

Binary search: target x = 70

	1	2	3	4	5	6	7	8	9	10	11	12
v	12	15	33	35	42	45	51	62	73	75	86	98

L: 7

Mid: 8

R: 9

`v(Mid) <= x`

So throw away the left half...

Binary search: target x = 70

	1	2	3	4	5	6	7	8	9	10	11	12
v	12	15	33	35	42	45	51	62	73	75	86	98

L: 8

Mid: 8

R: 9

Done because

R-L = 1

```
function L = binarySearch(x, v)
% Find position after which to insert x. v(1)<…<v(end).
% L is the index such that v(L) <= x < v(L+1);
% L=0 if x<v(1).  If x>v(end), L=length(v) but x~=v(L).

% Maintain a search window [L..R] such that v(L)<=x<v(R).
% Since x may not be in v, initially set ...
L=0;  R=length(v)+1;

% Keep halving [L..R] until R-L is 1,
%   always keeping  v(L) <= x < v(R)
while  R ~= L+1
    m= floor((L+R)/2);  % middle of search window
    if


    else


    end
end
```

```
function L = binarySearch(x, v)
% Find position after which to insert x. v(1)<…<v(end).
% L is the index such that v(L) <= x < v(L+1);
% L=0 if x<v(1).  If x>v(end), L=length(v) but x~=v(L).

% Maintain a search window [L..R] such that v(L)<=x<v(R).
% Since x may not be in v, initially set ...
L=0;  R=length(v)+1;

% Keep halving [L..R] until R-L is 1,
%   always keeping  v(L) <= x < v(R)
while  R ~= L+1
    m= floor((L+R)/2);  % middle of search window
    if  v(m) <= x

        L= m;
    else

        R= m;
    end
end
```

This version is different from that in *Insight*

```
function L = binarySearch(x, v)
% Find position after which to insert x. v(1)<…<v(end).
% L is the index such that v(L) <= x < v(L+1);
% L=0 if x<v(1).  If x>v(end), L=length(v) but x~=v(L).

% Maintain a search window [L..R] such that v(L)<=x<v(R).
% Since x may not be in v, initially set ...
L=0;  R=length(v)+1;

% Keep halving [L..R] until R-L is 1,
%   always keeping  v(L) <= x < v(R)
while  R ~= L+1
    m= floor((L+R)/2);  % middle of search window
    if  v(m) <= x

        L= m;
    else

        R= m;
    end
end
```

20	30	40	46	50	52	68	70

0 1 2 3 4 5 6 7 8 9

Play with **showBinarySearch.m**

What happens if the values in the sorted vector are not unique? Say, the target value is in the vector and that value appears in the vector multiple times…

A. The first occurrence is identified

B. The last occurrence is identified

C. Any one of the occurrences may be identified

D. Binary search doesn't work

Binary search is efficient, but we need to sort the vector in the first place so that we can use binary search

- Many different algorithms out there...
- We saw insertion sort (and read about bubble sort)
- Let's look at **merge sort**
- An example of the "divide and conquer" approach using recursion