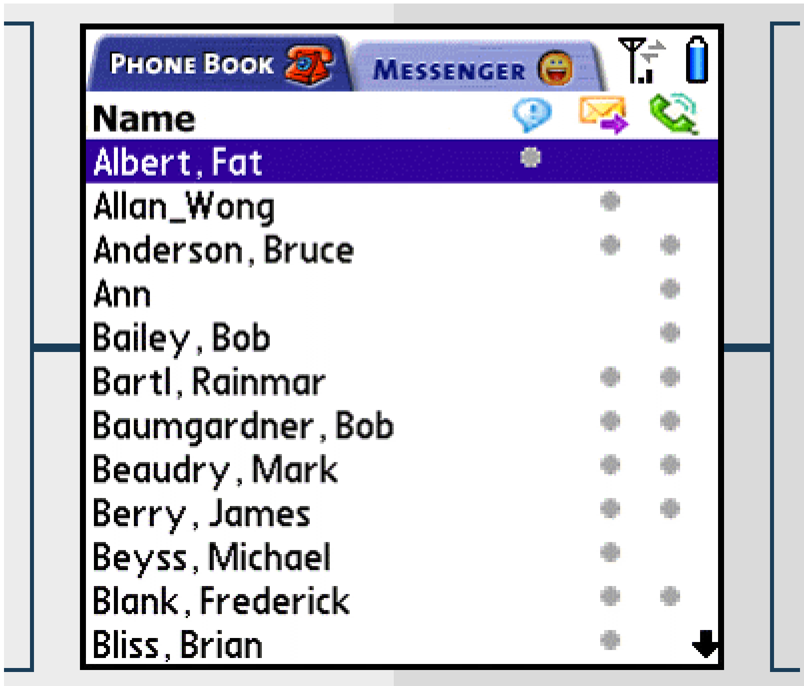
- Previous Lecture:
  - Recursion

- Today's Lecture:
  - Sorting and searching
    - *Insertion sort*, linear search
    - Read about *Bubble Sort* in Insight
  - "Divide and conquer" strategies
    - Binary search

- Announcements
  - Discussion in computer lab this week
  - P6 due Thursday at 11pm
  - Final exam: Dec 7^{th} 2pm for both Lec1 and Lec 2

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	

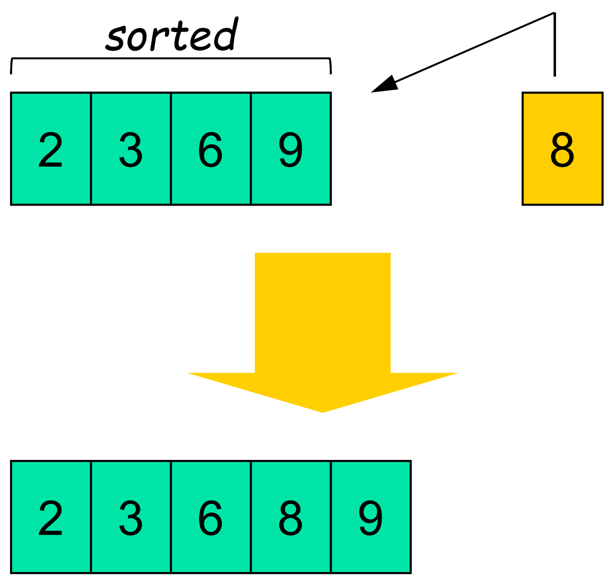
There are many algorithms for sorting

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

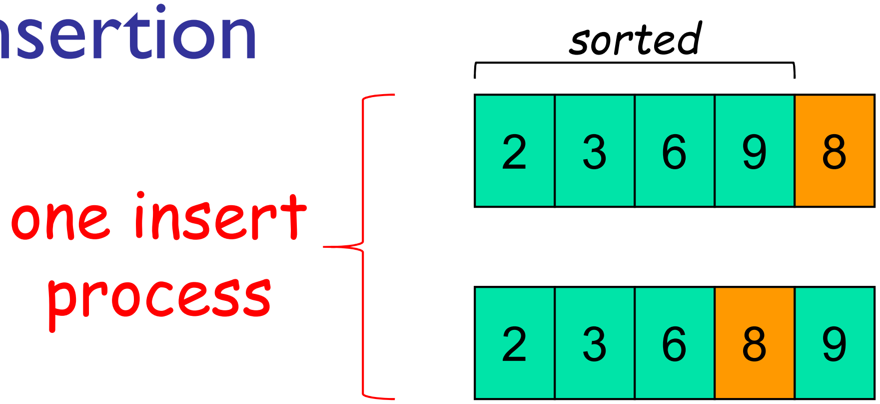
- Each has advantages and disadvantages. Some algorithms are faster (time-efficient) while others are memory-efficient
- *Great opportunity for learning how to analyze programs and algorithms!*

The Insertion Process

- <u>Given a sorted array</u> x, insert a number y such that the result is sorted



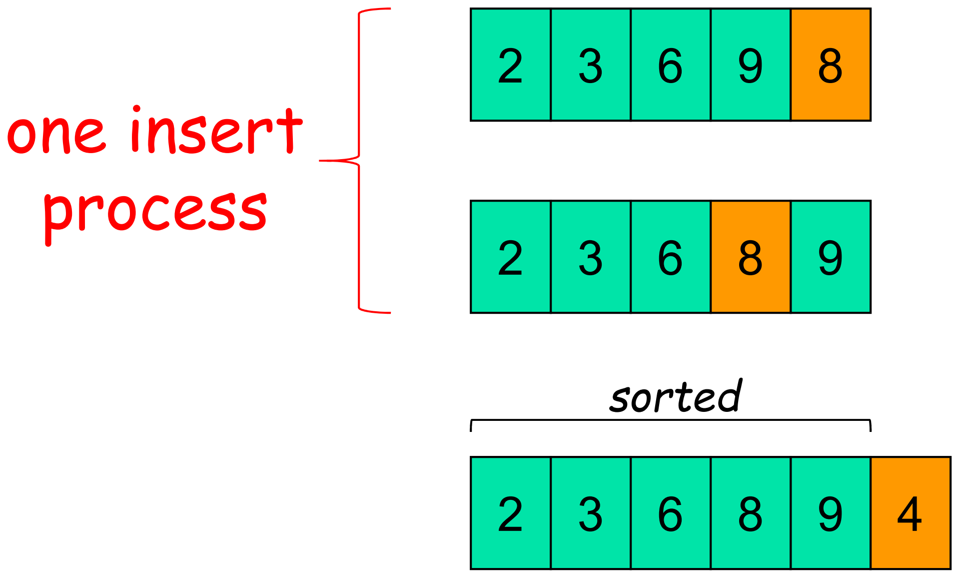
Insertion



Insert 8 into the sorted segment

Just swap 8 & 9

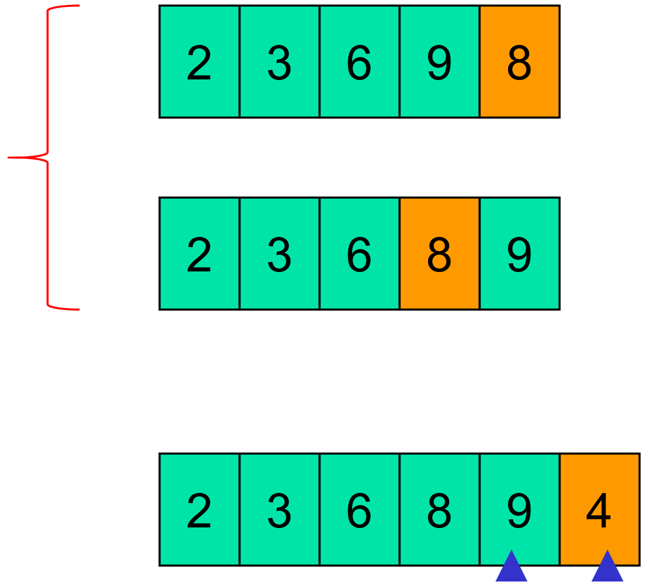
Insertion



Insert 4 into the sorted segment

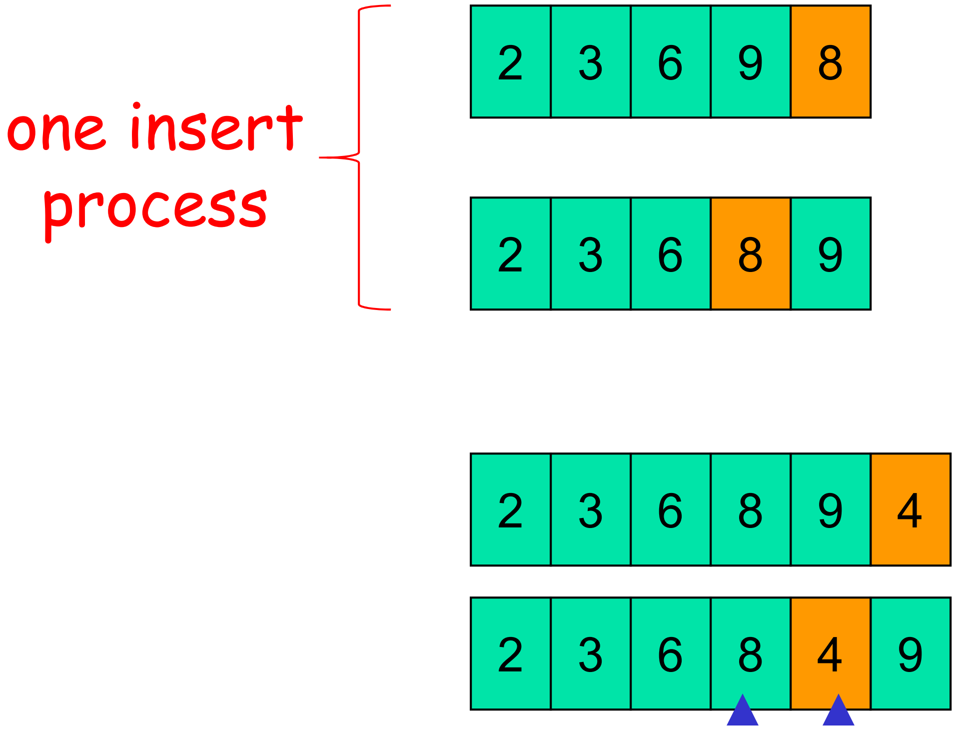
Insertion

one insert process



Compare adjacent components: swap 9 & 4

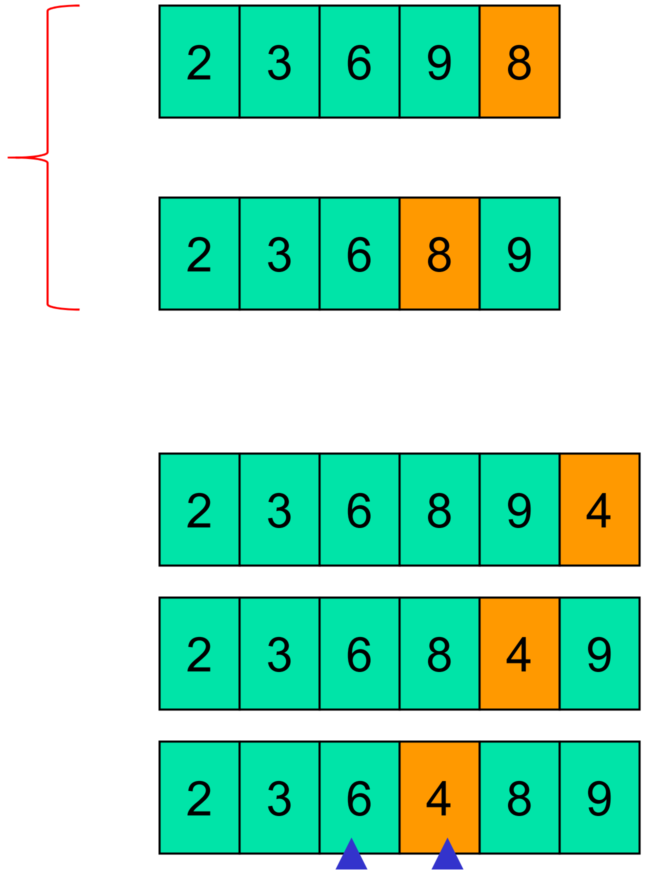
Insertion



Compare adjacent components:
swap 8 & 4

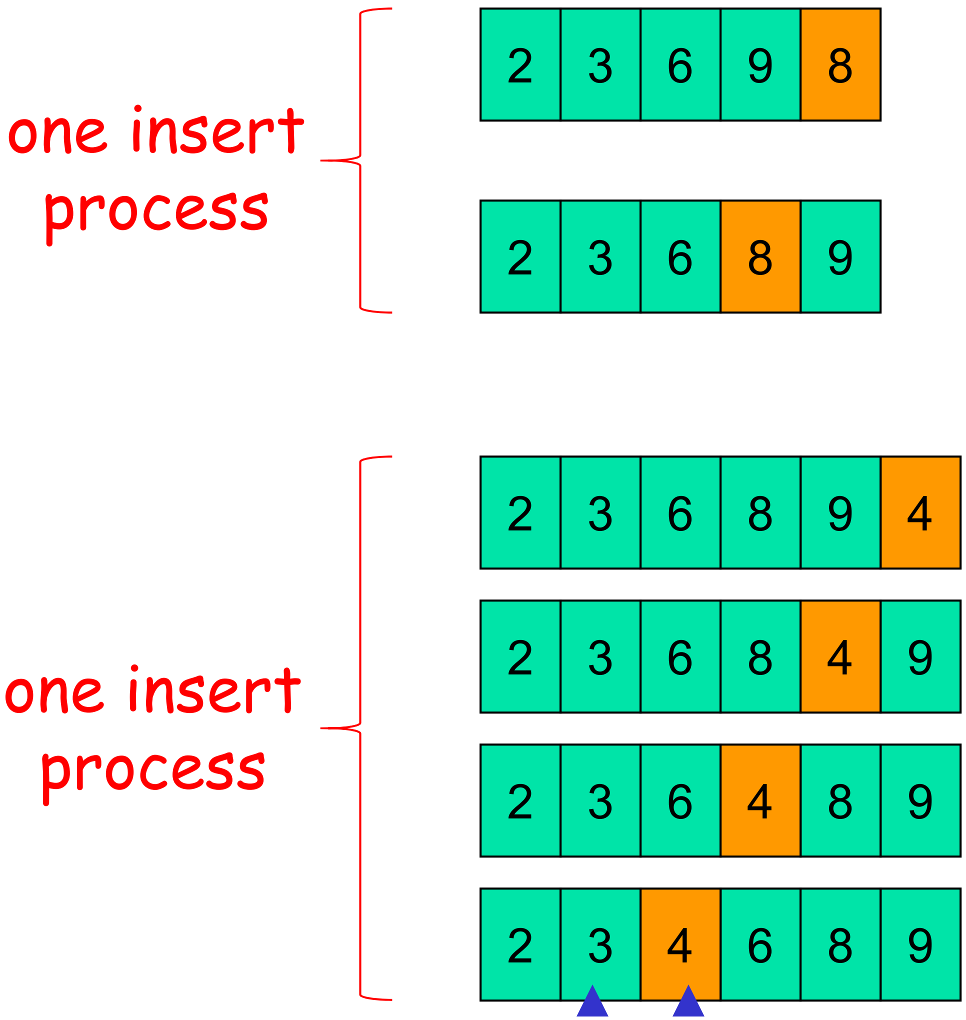
Insertion

one insert process



Compare adjacent components: swap 6 & 4

Insertion



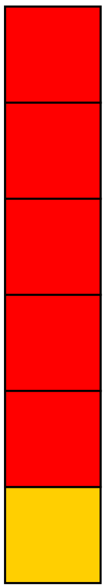
Compare adjacent components:
DONE! No more swaps.

See **`Insert.m`** 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),C,S] = Insert(x(1:2))**

Insert **x(3)**: **[x(1:3),C,S] = Insert(x(1:3))**

Insert **x(4)**: **[x(1:4),C,S] = Insert(x(1:4))**

Insert **x(5)**: **[x(1:5),C,S] = Insert(x(1:5))**

Insert **x(6)**: **[x(1:6),C,S] = Insert(x(1:6))**

InsertionSort.m

Insertion Sort vs. Bubble Sort

- Read about Bubble Sort in *Insight* §8.2
- Both algorithms involve the repeated comparison of adjacent values and swaps
- Find out which algorithm is more efficient on average

Other 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
```

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 or two statements

C. About half the code

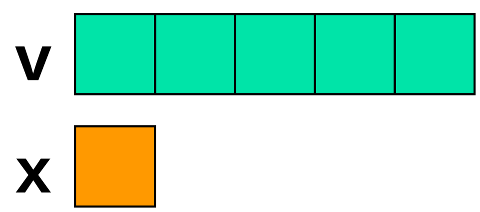
D. Most of the code

The only change is in how we do the **comparison!**

See **`InsertionSortIntervals.m`**

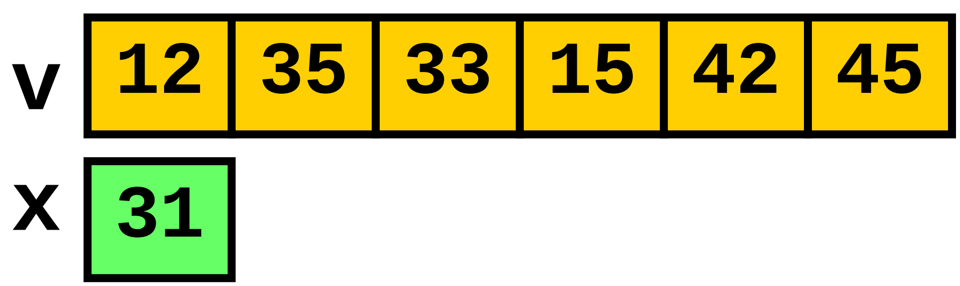
Searching for an item in an unorganized collection?

- 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
```



```
% 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
```

A. squared

B. doubled

C. the same

D. halved

Suppose another vector is twice as long as v. The expected “effort” required to do a linear search is …

```
% 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 Reed's number…

```
while  (Phone book is longer than 1 page)
        Open to the middle page.
        if  "Reed" 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.

```
% 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)`.

Binary Search

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

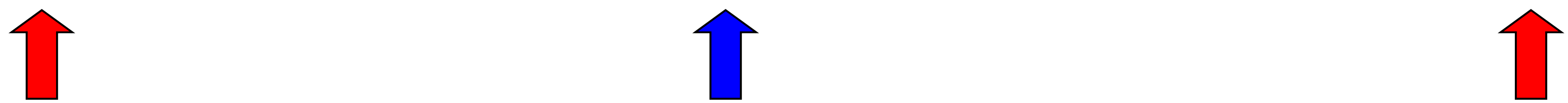
An item in a sorted array of length n can be located with just $\log_2 n$ comparisons.

"Savings" is significant!

n	log2(n)
100	7
1000	10
10000	13

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: 1

Mid: 6

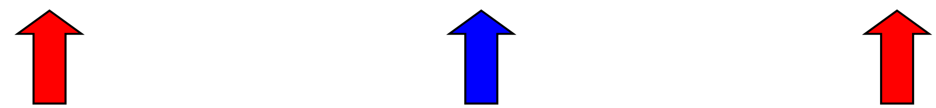
R: 12

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: 12

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

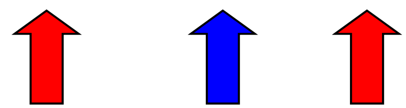
R: 9

`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**