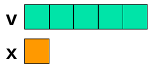
- Previous Lecture:
  - Acoustic data: frequency computation
  - Touchtone phone
- Today's Lecture:
  - Search: Linear Search
  - Sort: Bubble Sort and Insertion Sort
  - Efficiency Analysis
- Announcements:
  - Prelim 3 scores will be posted on Sunday; paper will be returned next Tues

---

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
```

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

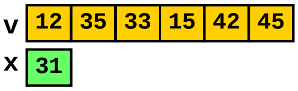
A. squared

B. doubled

C. the same

D. halved

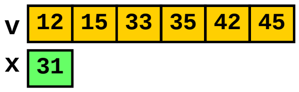
```
% Linear Search
% f is index of first occurrence
%   of value x in vector v.
% f is -1 if x not found.
k= 1;
while  k<=length(v) && v(k)~=x
    k= k + 1;
end
if  k>length(v)
    f= -1; % signal for x not found
else
    f= k;
end
```


v 12 35 33 15 42 45

x 31

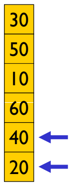
```
% Linear Search
% f is index of first occurrence of value x in vector v.
% f is -1 if x not found.
k= 1;
while  k<=length(v) && v(k)~=x
    k= k + 1;
end
if  k>length(v)
    f= -1; % signal for x not found
else
    f= k;
end
```

Searching in a sorted list should require less work


v 12 15 33 35 42 45

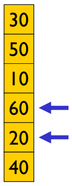
x 31

The "bubble" process



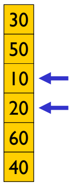
Compare adjacent values.
Swap if "out of order."

The “bubble” process



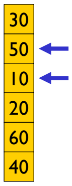
Compare adjacent values.
Swap if “out of order.”

The “bubble” process



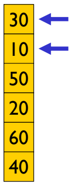
Compare adjacent values.
Swap if “out of order.”

The “bubble” process



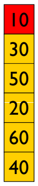
Compare adjacent values.
Swap if “out of order.”

The “bubble” process



Compare adjacent values.
Swap if “out of order.”

The “bubble” process

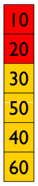


The smallest (lightest) value “bubbles” to the top

Done in one pass through the vector

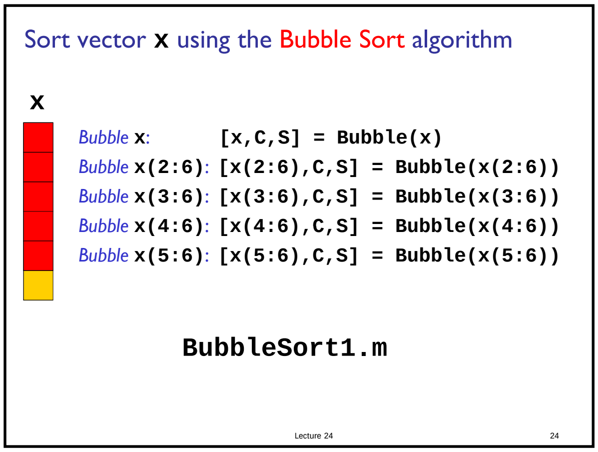
`Bubble.m`

The second “bubble” process



After two bubble processes, the first two components are sorted.

Repeatedly apply the bubble process to sort the whole array

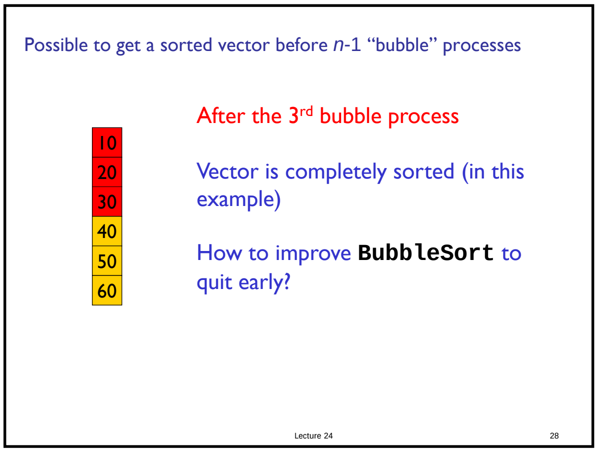


Sort vector **x** using the Bubble Sort algorithm

x

Bubble **x**: `[x,C,S] = Bubble(x)`
Bubble **x(2:6)**: `[x(2:6),C,S] = Bubble(x(2:6))`
Bubble **x(3:6)**: `[x(3:6),C,S] = Bubble(x(3:6))`
Bubble **x(4:6)**: `[x(4:6),C,S] = Bubble(x(4:6))`
Bubble **x(5:6)**: `[x(5:6),C,S] = Bubble(x(5:6))`

BubbleSort1.m

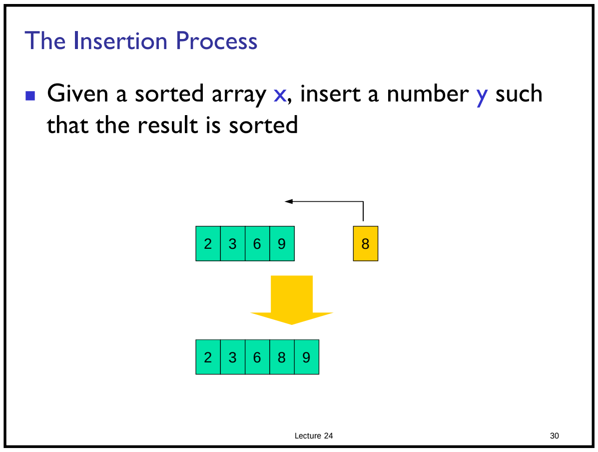


Possible to get a sorted vector before *n*-1 "bubble" processes

After the 3rd bubble process

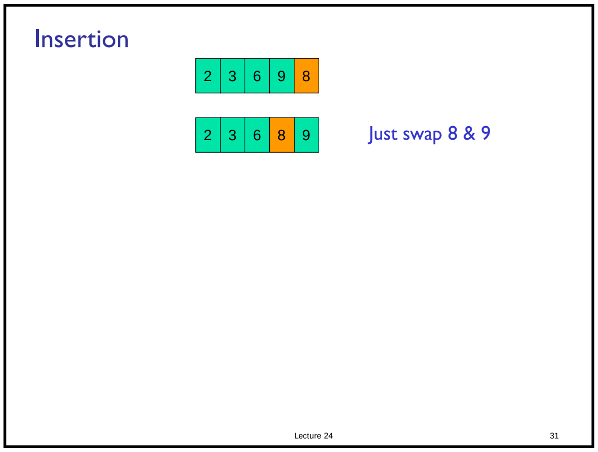
Vector is completely sorted (in this example)

How to improve **BubbleSort** to quit early?



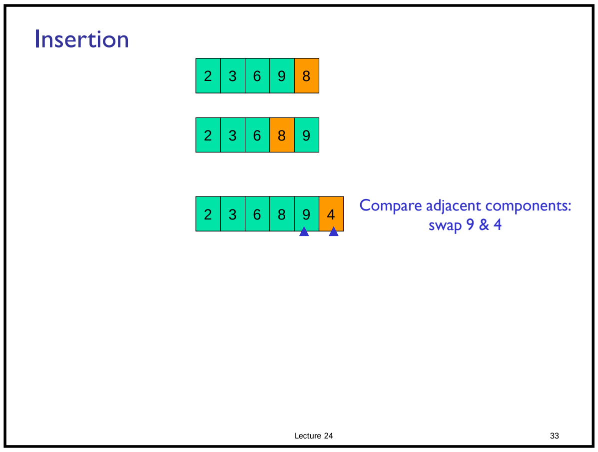
The Insertion Process

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



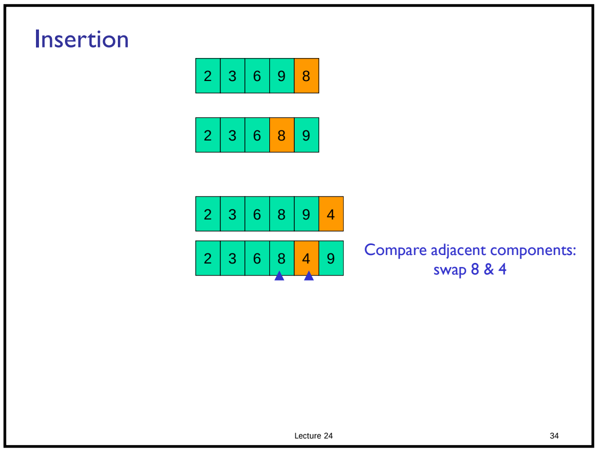
Insertion

Just swap 8 & 9



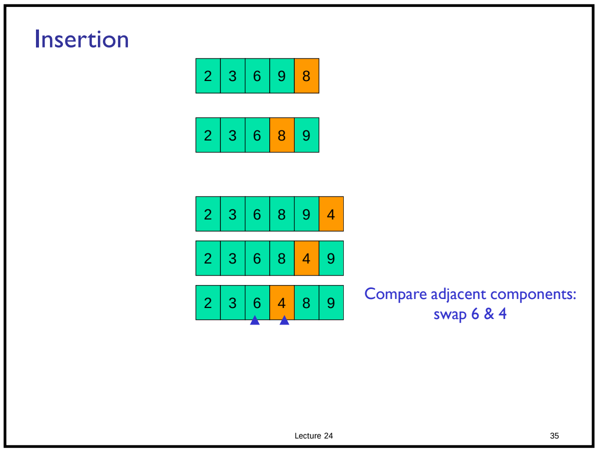
Insertion

Compare adjacent components: swap 9 & 4



Insertion

Compare adjacent components: swap 8 & 4



Insertion

2 3 6 9 8

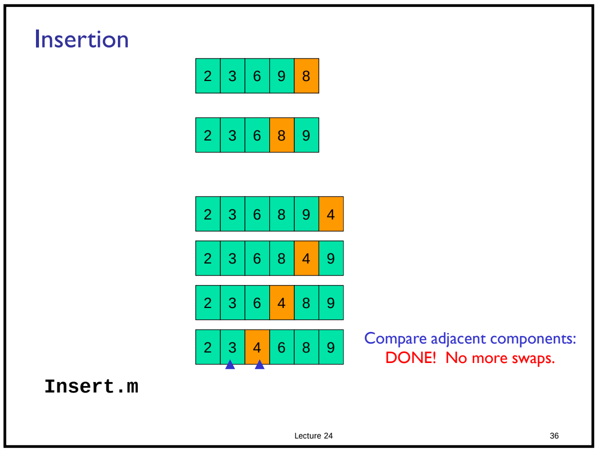
2 3 6 8 9

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

2 3 6 9 8

2 3 6 8 9

2 3 6 8 9 4

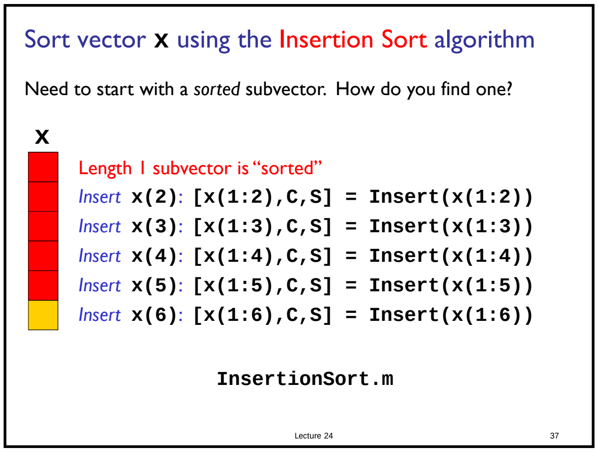
2 3 6 8 4 9

2 3 6 4 8 9

2 3 4 6 8 9

Compare adjacent components:
DONE! No more swaps.

`Insert.m`



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))`

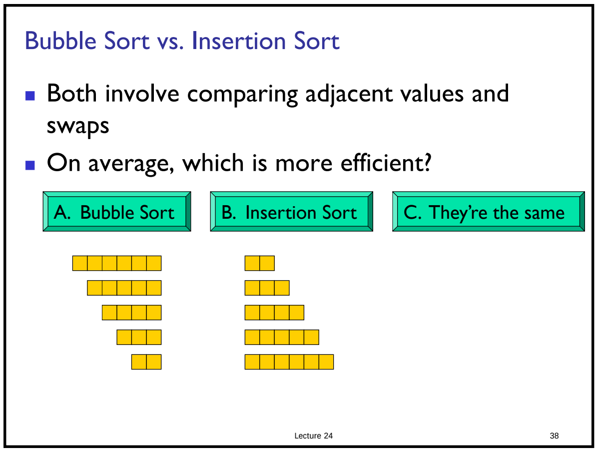
Insert `x(3)`: `[x(1:3),C,S] = Insert(x(1:3))`

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`



Bubble Sort vs. Insertion Sort

- Both involve comparing adjacent values and swaps
- On average, which is more efficient?

A. Bubble Sort

B. Insertion Sort

C. They're the same

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 = insertSort(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
```