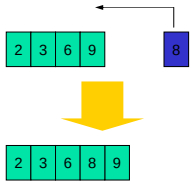
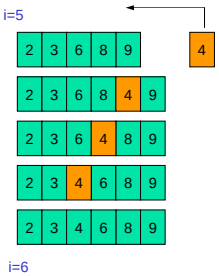
- Previous Lecture:
  - "Divide and conquer" strategies—recursion
    - Merge sort
    - Sierpinski Triangle, revisited
- Today's Lecture:
  - Insertion sort vs. merge sort
  - Timing with `tic toc`
  - Time efficiency vs. memory efficiency
- Announcements
  - Project 6 has been posted. Due 5/1, 6pm.
  - CS100M final will be 5/8 (Thurs) 9am. Tell us now if you have a final exam conflict. Email Kelly Patwell with your complete exam schedule (course #s and times)

Insertion Sort

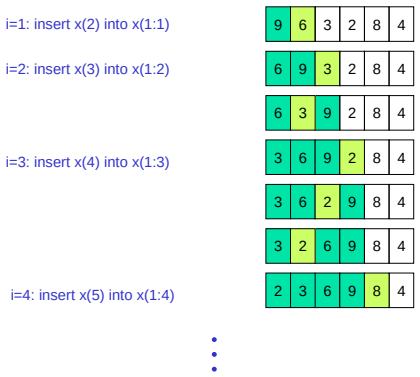
- Given a sorted array x, insert a number y such that the result is sorted



Develop the insertion sort algorithm



- The sorted segment grows one element at a time—need to keep track of the length of the sorted segment, say, index i
- What is the simpliest (shortest) case? A list of length 1, so start with i=1
- Inserting the (i+1)th element requires a series of swaps: swap until the element to be inserted is at the correct place
  - ➡ a while-loop




```
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
```


Use tic toc to perform timing operation

```
x= rand(1000,1);
% Time InsertSort
  tic
  y= insertSort(x);
  t= toc;  % #seconds since tic
```

How do merge sort and insertion sort compare?

- Merge sort: _____
- Insertion sort: (worst case) takes j operations to insert an element in a sorted array of j elements. In total
  _____ for big N
- Insertion sort is done *in-place*; merge sort requires much more memory

How to choose??

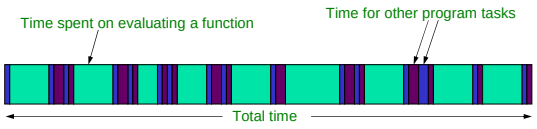
- Depends on application
- Merge sort is especially good for sorting large data set (but watch out for memory usage)
- Insertion sort is "order N^2" at worst case, but what about an average case? If the application requires that you maintain a sorted array, insertion sort may be a good choice

Why not just use Matlab's sort function?

- Flexibility
- E.g., to maintain a sorted list, just write the code for insertion sort
- E.g., sort strings or other complicated structures
- Sort according to some criterion set out in a function file
  - Observe that we have the comparison `x(j+1)<x(j)`
  - The comparison can be a function that returns a boolean value

Expensive function evaluations

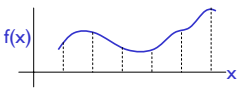
- Consider the execution of a program that is dominated by multiple calls to an expensive-to-evaluate function (e.g., climate simulation models)



- Can try to improve efficiency by dealing with the expensive function evaluations

Dealing with expensive function evaluations

- Can the function code be improved?
- Can we do fewer function evaluations?
- Can we pre-compute and store specific function values so that during the main program execution the program can just look up the values?
  - Consider function f(x). If there are many function calls and few distinct values of x, can get substantial speedup
  - Only speeds up main program execution—it still takes time to do the pre-computation



What are some issues and potential problems with the "table look-up" strategy?