Previous Lecture:
- Recursion examples: remove all occurrences of a character in a string, a mesh of triangles
- Sorting algorithms: Insertion Sort (and Bubble Sort in Insight §8.2)

Today’s Lecture:
- Analyze Insertion Sort and Bubble Sort
- Sorting an array of objects
- Writing efficient code

Announcements:
- Discussion this week in Upson B7 computer lab
- Project 6 parts A and B due 5/2, Thurs, at 11pm

The Insertion Process
- Given a sorted array \( x \), insert a number \( y \) such that the result is sorted

\[
\begin{align*}
2 & \quad 3 & \quad 6 & \quad 9 & \quad 8 \\
2 & \quad 3 & \quad 6 & \quad 9 & \quad 4 \\
2 & \quad 3 & \quad 6 & \quad 4 & \quad 9 \\
2 & \quad 3 & \quad 4 & \quad 6 & \quad 9 \\
\end{align*}
\]

Sort vector \( x \) using the Insertion Sort algorithm

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

\[
\begin{align*}
x(2) & : [x(1:2),C,S] = \text{Insert}(x(1:2)) \\
x(3) & : [x(1:3),C,S] = \text{Insert}(x(1:3)) \\
x(4) & : [x(1:4),C,S] = \text{Insert}(x(1:4)) \\
x(5) & : [x(1:5),C,S] = \text{Insert}(x(1:5)) \\
x(6) & : [x(1:6),C,S] = \text{Insert}(x(1:6)) \\
\end{align*}
\]

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

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
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.

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

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 ...

% 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

What if v is sorted?