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.
Insertion

one insert process

sorted

2 3 6 9 8

Insert 8 into the sorted segment

2 3 6 8 9

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

Compare adjacent components: swap 9 & 4
Insertion

Compare adjacent components: swap 8 & 4
Insertion

Compare adjacent components: swap 6 & 4
Insertion

one insert process

Compare adjacent components:
DONE! No more swaps.

See Insert.m for the insert process
Sort vector $\mathbf{x}$ using the Insertion Sort algorithm

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

Length 1 subvector is “sorted”

Insert $\mathbf{x}(2)$: $[\mathbf{x}(1:2), \mathbf{C}, \mathbf{S}] = \text{Insert}(\mathbf{x}(1:2))$

Insert $\mathbf{x}(3)$: $[\mathbf{x}(1:3), \mathbf{C}, \mathbf{S}] = \text{Insert}(\mathbf{x}(1:3))$

Insert $\mathbf{x}(4)$: $[\mathbf{x}(1:4), \mathbf{C}, \mathbf{S}] = \text{Insert}(\mathbf{x}(1:4))$

Insert $\mathbf{x}(5)$: $[\mathbf{x}(1:5), \mathbf{C}, \mathbf{S}] = \text{Insert}(\mathbf{x}(1:5))$

Insert $\mathbf{x}(6)$: $[\mathbf{x}(1:6), \mathbf{C}, \mathbf{S}] = \text{Insert}(\mathbf{x}(1:6))$

InsertionSort.m
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
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

The only change is in how we do the comparison!

See InsertionSortIntervals.m

Lecture 27
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
What are some issues and potential problems with the “table look-up” strategy?

- **Accuracy**—need a “dense grid” to get high accuracy \( \rightarrow \) significant memory usage
- If an exact x-value is not found, need some kind of approximation
- Incur searching cost if the x-values are not simple indices
- Feasible in high dimensions (multiple dependent variables)?

<table>
<thead>
<tr>
<th>x</th>
<th>f(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.01</td>
</tr>
<tr>
<td>2</td>
<td>2.67</td>
</tr>
<tr>
<td>3</td>
<td>5.71</td>
</tr>
<tr>
<td>4</td>
<td>9.12</td>
</tr>
<tr>
<td>5</td>
<td>7.98</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Pre-calculate and store these values (e.g., in a vector H)

To be continued in this week’s lab.
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
% 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

% 12 35 33 15 42 45
x 31