Previous Lecture:
- OOP: Overriding methods
- Introduction to recursion

Today’s Lecture:
- Recursion examples: remove all occurrences of a character in a string, a mesh of triangles
- Sort algorithm: Insertion Sort
- See Insight §8.2 for the Bubble Sort algorithm

Announcements:
- Last call to notify us of final exam conflict
- Project 6 parts A and B due 5/2, Thurs, at 11pm
Recursion

- The Fibonacci sequence is defined recursively:
  \[ F(1) = 1, \quad F(2) = 1, \]
  \[ F(3) = F(1) + F(2) = 2, \quad F(4) = F(2) + F(3) = 3 \]
  \[ F(k) = F(k-2) + F(k-1) \]
  It is defined in terms of itself; its definition invokes itself.

- Algorithms and functions can be recursive as well. I.e., a function can call itself.

- Example: remove all occurrences of a character from a string
  \[
  'gc aatc gga c' \rightarrow 'gcaatcggaac'
  \]
Example: removing all occurrences of a character

- Can solve using iteration—check one character (one component of the vector) at a time

\[ S \]

\[
\begin{array}{cccc}
    \text{1} & \text{2} & \ldots & k & \ldots \\
    'c' & 's' & ' ' & '1' & '1' & '1' & '2' \\
\end{array}
\]

Subproblem 1:
Keep or discard \( s(1) \)

Subproblem 2:
Keep or discard \( s(2) \)

Subproblem \( k \):
Keep or discard \( s(k) \)

Iteration:
Divide problem into sequence of equal-sized, identical subproblems

See RemoveChar_loop.m
Example: removing all occurrences of a character

- Can solve using **recursion**
  - Original problem: remove all the blanks in string s
  - Decompose into two parts: 1. remove blank in s(1)
    2. remove blanks in s(2:length(s))

Original problem: 

Decompose into 2 parts: 

Decompose
function s = removeChar(c, s)
% Return string s with character c removed

if length(s) == 0  % Base case: nothing to do
    return
else

end
function s = removeChar(c, s)
% Return string s with character c removed

if length(s)==0  % Base case: nothing to do
    return
else
    if s(1)~=c
        else
            end
        end
    else
        end
end
function s = removeChar(c, s)
% Return string s with character c removed

if length(s) == 0  % Base case: nothing to do
    return
else
    if s(1) ~= c
        % return string is
        % s(1) and remaining s with char c removed

    else

    end
end
function s = removeChar(c, s)
% Return string s with character c removed

if length(s)==0  % Base case: nothing to do
    return
else
    if s(1)~=c
        % return string is
        % s(1) and remaining s with char c removed
    else
        % return string is just
        % the remaining s with char c removed
    end
end
end
function s = removeChar(c, s)
% Return string s with character c removed

if length(s) == 0 % Base case: nothing to do
    return
else
    if s(1) ~= c
        % return string is
        % s(1) and remaining s with char c removed
        s = [s(1) removeChar(c, s(2:length(s)));
    else
        % return string is just
        % the remaining s with char c removed
    end
end
function s = removeChar(c, s)
% Return string s with character c removed

if length(s)==0  % Base case: nothing to do
    return
else
    if s(1)==c
        % return string is
        % s(1) and remaining s with char c removed
        s= [s(1) removeCSub(c, s(2:length(s)))];
    else
        % return string is just
        % the remaining s with char c removed
        s= removeCSub(c, s(2:length(s)));
    end
end
function s = removeChar(c, s)
% Return string s with character c removed

if length(s)==0  % Base case: nothing to do
    return
else
    if s(1)~=c
        % return string is
        % s(1) and remaining s with char c removed
        s= [s(1) removeCSub(c, s(2:length(s)))];
    else
        % return string is just
        % the remaining s with char c removed
        s= removeCSub(c, s(2:length(s)));
    end
end
end
function s = removeChar(c, s)
% Return string s with character c removed

if length(s) == 0  % Base case: nothing to do
    return
else
    if s(1) ~= c
        % return string is
        % s(1) and remaining s with char c removed
        s = [s(1) removeChar(c, s(2:length(s)))];
    else
        % return string is just
        % the remaining s with char c removed
        s = removeChar(c, s(2:length(s)));
    end
end
function s = removeChar(c, s)
    if length(s)==0
        return
    else
        if s(1)~=c
            s= [s(1) removeChar(c, s(2:length(s)))];
        else
            s= removeChar(c, s(2:length(s)));
        end
    end
end

removeChar - 1st call

\[
\begin{array}{c}
c \_ \\
s \_ \_ o \_ \_ g
\end{array}
\]
function s = removeChar(c, s)
if length(s)==0
    return
else
    if s(1)~=c
        s= [s(1) removeChar(c, s(2:length(s)))];
    else
        s= removeChar(c, s(2:length(s)));
    end
end

removeChar - 1st call

<table>
<thead>
<tr>
<th>s</th>
<th>d_ o_ g</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>_____</td>
</tr>
</tbody>
</table>
function s = removeChar(c, s)
    if length(s)==0
        return
    else
        if s(1)~=c
            s = [s(1) removeChar(c, s(2:length(s)))];
        else
            s = removeChar(c, s(2:length(s)));
        end
    end
end
function s = removeChar(c, s)
    if length(s)==0
        return
    else
        if s(1)~=c
            s= [s(1) removeChar(c, s(2:length(s)))];
        else
            s= removeChar(c, s(2:length(s)));
        end
    end
end
```matlab
function s = removeChar(c, s)
    if length(s) == 0
        return
    else
        if s(1) ~= c
            s = [s(1) removeChar(c, s(2:length(s)))];
        else
            s = removeChar(c, s(2:length(s)));
        end
    end
end
```
function s = removeChar(c, s)
    if length(s)==0
        return
    else
        if s(1)==c
            s = [s(1) removeChar(c, s(2:length(s)))];
        else
            s = removeChar(c, s(2:length(s)));
        end
    end
end
function s = removeChar(c, s)
    if length(s)==0
        return
    else
        if s(1)~=c
            s = [s(1) removeChar(c, s(2:length(s)))];
        else
            s = removeChar(c, s(2:length(s)));
        end
    end
end
function s = removeChar(c, s)
if length(s)==0
    return
else
    if s(1)~=c
        s = [s(1) removeChar(c, s(2:length(s)))];
    else
        s = removeChar(c, s(2:length(s)));  \[       \]
    end
end

removeChar – 1st call
\[
\begin{array}{|c|}
\hline
\text{c} \\
\text{s} | d \quad o \quad o \quad g \\
\text{[d ______]} \\
\hline
\end{array}
\]

removeChar – 2nd call
\[
\begin{array}{|c|}
\hline
\text{c} \\
\text{s} | o \quad o \quad g \\
\text{[ ______]} \\
\hline
\end{array}
\]

removeChar – 3rd call
\[
\begin{array}{|c|}
\hline
\text{c} \\
\text{s} | o \quad g \\
\text{[o ______]} \\
\hline
\end{array}
\]

removeChar – 4th call
\[
\begin{array}{|c|}
\hline
\text{c} \\
\text{s} | g \\
\text{[ ______]} \\
\hline
\end{array}
\]

removeChar – 5th call
\[
\begin{array}{|c|}
\hline
\text{c} \\
\text{s} | g \\
\text{[g ______]} \\
\hline
\end{array}
\]

removeChar – 6th call
\[
\begin{array}{|c|}
\hline
\text{c} \\
\text{s} | ' \\
\text{[ ______]} \\
\hline
\end{array}
\]
function s = removeChar(c, s)
    if length(s)==0
        return
    else
        if s(1)~=c
            s = [s(1) removeChar(c, s(2:length(s)))];
        else
            s = removeChar(c, s(2:length(s)));
        end
    end
end
function s = removeChar(c, s)
    if length(s)==0
        return
    else
        if s(1)==c
            s = [s(1) removeChar(c, s(2:length(s)))];
        else
            s = removeChar(c, s(2:length(s)));
        end
    end
end
function s = removeChar(c, s)
    if length(s)==0
        return
    else
        if s(1)~=c
            s= [s(1) removeChar(c, s(2:length(s)))];
        else
            s= removeChar(c, s(2:length(s)));
        end
    end
end
function s = removeChar(c, s)
if length(s)==0
    return
else
    if s(1)==c
        s= [s(1) removeChar(c, s(2:length(s)))];
    else
        s= removeChar(c, s(2:length(s)));
    end
end
Key to recursion

- Must identify (at least) one base case, the “trivially simple” case
  - no recursion is done in this case
- The recursive case(s) must reflect progress towards the base case
  - E.g., give a shorter vector as the argument to the recursive call – see removeChar
Divide-and-conquer methods, such as recursion, is useful in geometric situations

Chop a region up into triangles with smaller triangles in “areas of interest”

Recursive mesh generation
Mesh Generation

Step one in simulating flow around an airfoil is to generate a mesh and (say) estimate velocity at each mesh point.
Mesh Generation in 3D
Why is mesh generation a divide-&-conquer process?

Let’s draw this graphic
Start with a triangle
A “level-1” partition of the triangle

(obtained by connecting the midpoints of the sides of the original triangle)

Now do the same partitioning (connecting midpts) on each corner (white) triangle to obtain the “level-2” partitioning
The “level-2” partition of the triangle
The “level-3” partition of the triangle
The “level-4” partition of the triangle
The “level-4” partition of the triangle
The basic operation at each level

if the triangle is small
Don’t subdivide and just color it yellow.

else
Subdivide:
Connect the side midpoints;
color the interior triangle magenta;
apply same process to each outer triangle.
end
Draw a level-4 partition of the triangle with these vertices
At the start...
Recur: apply the same process on the lower left triangle
Recur again
... and again

The next lower left corner triangle (white) is small—no more subdivision and just color it yellow.
Now lower left corner triangle of the “level-4” partition is done. Continue with another corner triangle.
... and continue
Now the lower left corner triangle of the “level-3” partition is done. Continue with another corner triangle…
We’re “climbing our way out” of the deepest level of partitioning
Eventually climb all the way out to get the final result
The basic operation at each level

if the triangle is small
Don’t subdivide and just color it yellow.
else
Subdivide:
Connect the side midpoints;
color the interior triangle magenta;
apply same process to each outer triangle.
end
function MeshTriangle(x,y,L)
% x,y are 3-vectors that define the vertices of a triangle.
% Draw level-L partitioning. Assume hold is on.

if L==0
  % Recursion limit reached; no more subdivision required.
  fill(x,y,'y') % Color this triangle yellow
else
  % Need to subdivide: determine the side midpoints; connect
  % midpts to get “interior triangle”; color it magenta.

  % Apply the process to the three "corner" triangles...
end
function MeshTriangle(x,y,L)
% x,y are 3-vectors that define the vertices of a triangle.
% Draw level-L partitioning. Assume hold is on.

if L==0
  % Recursion limit reached; no more subdivision required.
  fill(x,y,'y')  % Color this triangle yellow
else
  % Need to subdivide: determine the side midpoints; connect
  % midpts to get "interior triangle"; color it magenta.
  a = [(x(1)+x(2))/2 (x(2)+x(3))/2 (x(3)+x(1))/2];
  b = [(y(1)+y(2))/2 (y(2)+y(3))/2 (y(3)+y(1))/2];
  fill(a,b,'m')
  % Apply the process to the three "corner" triangles...
end
function MeshTriangle(x,y,L)
% x,y are 3-vectors that define the vertices of a triangle.
% Draw level-L partitioning. Assume hold is on.

if L==0
  % Recursion limit reached; no more subdivision required.
  fill(x,y,'y')  % Color this triangle yellow
else
  % Need to subdivide: determine the side midpoints; connect
  % midpts to get "interior triangle"; color it magenta.
  a = [(x(1)+x(2))/2 (x(2)+x(3))/2 (x(3)+x(1))/2];
  b = [(y(1)+y(2))/2 (y(2)+y(3))/2 (y(3)+y(1))/2];
  fill(a,b,'m')

  % Apply the process to the three "corner" triangles...
  MeshTriangle([x(1) a(1) a(3)],[y(1) b(1) b(3)],L-1)
  MeshTriangle([x(2) a(2) a(1)],[y(2) b(2) b(1)],L-1)
  MeshTriangle([x(3) a(3) a(2)],[y(3) b(3) b(2)],L-1)
end
Key to recursion

- Must identify (at least) one base case, the “trivially simple” case
  - No recursion is done in this case
- The recursive case(s) must reflect progress towards the base case
  - E.g., give a shorter vector as the argument to the recursive call – see removeChar
  - E.g., ask for a lower level of subdivision in the recursive call – see MeshTriangle
Searching for an item in a collection

Is the collection organized?
What is the organizing scheme?
Sorting data allows us to search more easily

<table>
<thead>
<tr>
<th>Rank</th>
<th>Name</th>
<th>Time</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NED Ranomi Kromowidjojo</td>
<td>53.00</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>BLR Aliaksandra Herasimenia</td>
<td>53.38</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>CHN Tang Yi</td>
<td>53.44</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>AUS Melanie Schlanger</td>
<td>53.47</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>USA Missy Franklin</td>
<td>53.64</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>GBR Francesca Halsall</td>
<td>53.66</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>DEN Jeanette Ottesen Gray</td>
<td>53.75</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>USA Jessica Hardy</td>
<td>54.02</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Score</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jorge</td>
<td>92.1</td>
<td></td>
</tr>
<tr>
<td>Ahn</td>
<td>91.5</td>
<td></td>
</tr>
<tr>
<td>Oluban</td>
<td>90.6</td>
<td></td>
</tr>
<tr>
<td>Chi</td>
<td>88.9</td>
<td></td>
</tr>
<tr>
<td>Minale</td>
<td>88.1</td>
<td></td>
</tr>
<tr>
<td>Bell</td>
<td>87.3</td>
<td></td>
</tr>
</tbody>
</table>
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

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

Insert 8 into the sorted segment

one insert process

Just swap 8 & 9
Insertion

Insert 4 into the sorted segment

2 3 6 9 8

2 3 6 8 9

sorted

2 3 6 8 9 4
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 $\mathbf{x}(2): [\mathbf{x}(1:2), C, S] = \text{Insert}(\mathbf{x}(1:2))$

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

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

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

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

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