Previous Lecture:
- OOP: Access modifiers & inheritance

Today, Lecture 25:
- Recursion

Announcements:
- Project 6A code is available
  - Description still being refined, but draft is available
  - More reading than writing
- Final exam May 24
  - Conflict survey coming this weekend; reply ASAP
- Course evaluation survey next week
  - Anonymous responses, but credit for submitting it
Recursion

A method of problem solving by breaking a problem into smaller and smaller instances of the same problem until an instance is so small that it’s trivial to solve
Fibonacci sequence

Sequence

\[ f_1 = 1, \quad f_2 = 1 \]
\[ f_n = f_{n-1} + f_{n-2} \]

\( f(1) = 1; \quad f(2) = 1 \)
for \( k = 3:n \)
\[
    f(k) = f(k-1) + f(k-2);
\]
end

Function

\[ f(n) = \begin{cases} 
1, & n < 3 \\
(f(n-1) + f(n-2)), & n \geq 3
\end{cases} \]

function y = f(n)
    if n < 3
        y = 1;
    else
        y = f(n-1) + f(n-2);
    end
end
Recursion

- The Fibonacci sequence is defined recursively:
  \[ F(1) = 1, \ F(2) = 1, \]
  \[ F(3) = F(1) + F(2) = 2, \]
  \[ F(4) = F(2) + F(3) = 3 \]

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' → 'gcaatcggac'
Example: removing all occurrences of a character

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

\[
\begin{array}{cccccccc}
1 & 2 & \cdots & k & \cdots \\
\text{s} & \text{‘c’} & \text{‘s’} & \text{‘’} & \text{‘1’} & \text{‘1’} & \text{‘1’} & \text{‘2’}
\end{array}
\]

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))
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
        % 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
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)
% 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

\[ \text{removeChar('\_', 'd_o_g')} \]
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('_', 'd_o_g')

removeChar - 1\textsuperscript{st} call
\[
\begin{array}{c}
\text{c}_- \\
\text{s}\_\_d_o_g \\
[\text{d} \_\_\_\_]
\end{array}
\]

removeChar - 2\textsuperscript{nd} call
\[
\begin{array}{c}
\text{c}_- \\
\text{s}\_\_\_o_g \\
[\_\_\_\_\_\_]
\end{array}
\]

removeChar - 3\textsuperscript{rd} call
\[
\begin{array}{c}
\text{c}_- \\
\text{s}\_\_o_g \\
[\_\_\_\_\_\_]
\end{array}
\]

removeChar - 4\textsuperscript{th} call
\[
\begin{array}{c}
\text{c}_- \\
\text{s}\_g \\
[\_\_\_\_\_\_]
\end{array}
\]

removeChar - 5\textsuperscript{th} call
\[
\begin{array}{c}
\text{c}_- \\
\text{s}g \\
[g \_\_\_\_\_]
\end{array}
\]
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
end

removeChar('_', 'd_o_g')
```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

removeChar(‘_’, 'd_o_g')
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`
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

How many call frames are opened (used) in executing each of the following statements?

>> st = removeChar('t', 'Matlab');
>> sx = removeChar('x', 'Matlab');

A 3, 0  B 4, 1  C 3, 6  D 6, 6  E 7, 7
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”

3D Graphics: Level of Detail

Recursive mesh generation
When physics is too complicated for one big region, divide it into two smaller regions.

- **Subproblem:** solve physics inside one region
- **Division:** split region in half
- **Base case:** solution looks smooth in entire region

Nilsson, Gerritsen, Younis 2004
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:
    left, right, top;
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...
    MeshTriangle([x(1) a(1) a(3)],[y(1) b(1) b(3)],L-1)
    MeshTriangle([a(1) x(2) a(2)],[b(1) y(2) b(2)],L-1)
    MeshTriangle([a(3) a(2) x(3)],[b(3) b(2) y(3)],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., do a **lower level of subdivision** in the recursive call – see `MeshTriangle`
Recursion can be useful in different settings.

Mesh generation

Computer graphics

Search "tree" structures