- Previous Lecture:
  - Inheritance in OOP
  - Overriding methods
- Today's Lecture:
  - Recursion
    - Remove all occurrences of a character in a string
    - A mesh of triangles
- Announcements:
  - Discussion in the lab this week. Attendance is optional but be sure to do the posted exercise.
  - Project 6 due Thurs Dec 4 at 11pm. Remember academic integrity!
  - Office/consulting hours end Tuesday (tonight) for Thanksgiving Break and resume Monday

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

Subproblem 1:
Keep or discard s(1)

Subproblem 2:
Keep or discard s(2)

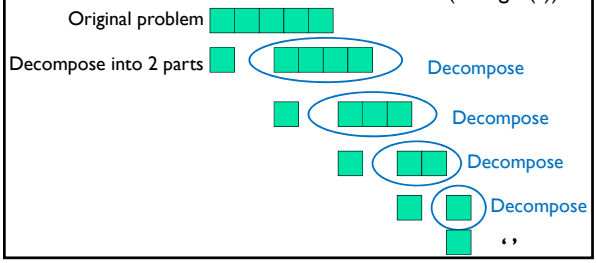
Subproblem k:
Keep or discard s(k)

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
  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)
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
```

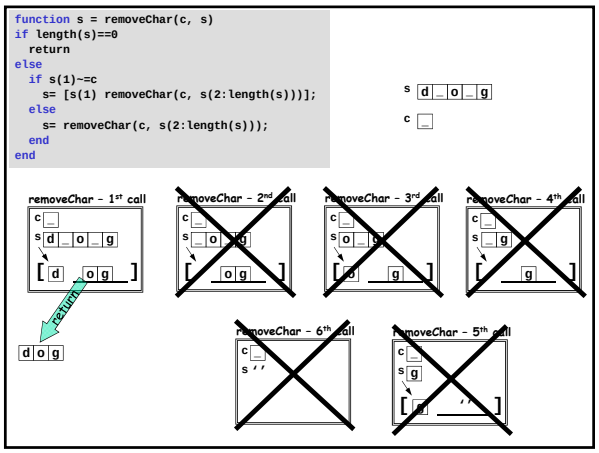
s d _ o _ g

c _

removeChar - 1st call

c _

s d _ o _ g

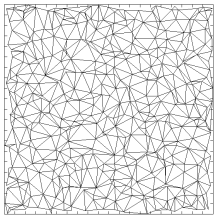


Key to recursion

- 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`

Divide-and-conquer methods, such as recursion, is useful in geometric situations

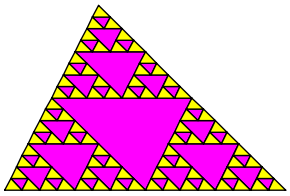
Chop a region up into triangles with smaller triangles in "areas of interest"



Recursive mesh generation

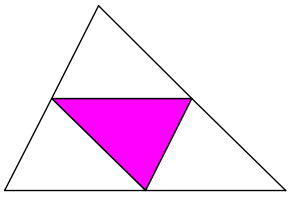
Why is mesh generation a divide-&-conquer process?

Let's draw this graphic



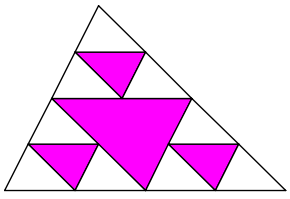
A "level-1" partition of the triangle

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

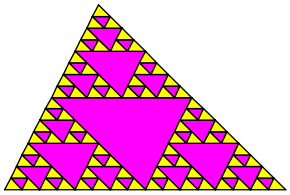


Now do the same partitioning (connecting midpts) on each corner (white) triangle to obtain the "level-2" partitioning

The "level-2" partition of the triangle



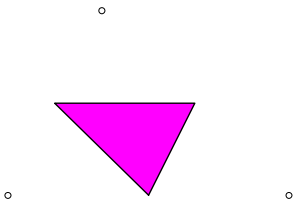
The "level-4" partition of the triangle



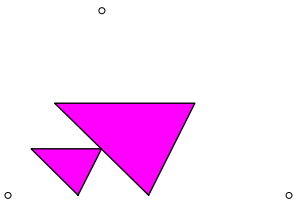
The basic operation at each level

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

At the start…



Recur: apply the same process on the lower left triangle

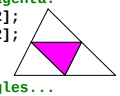



```
function MeshTriangle(x,y,L)
% x,y are 3-vectors that define the vertices of a triangle.
% Draw level-L partitioning.  Assume hold is on.

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`**