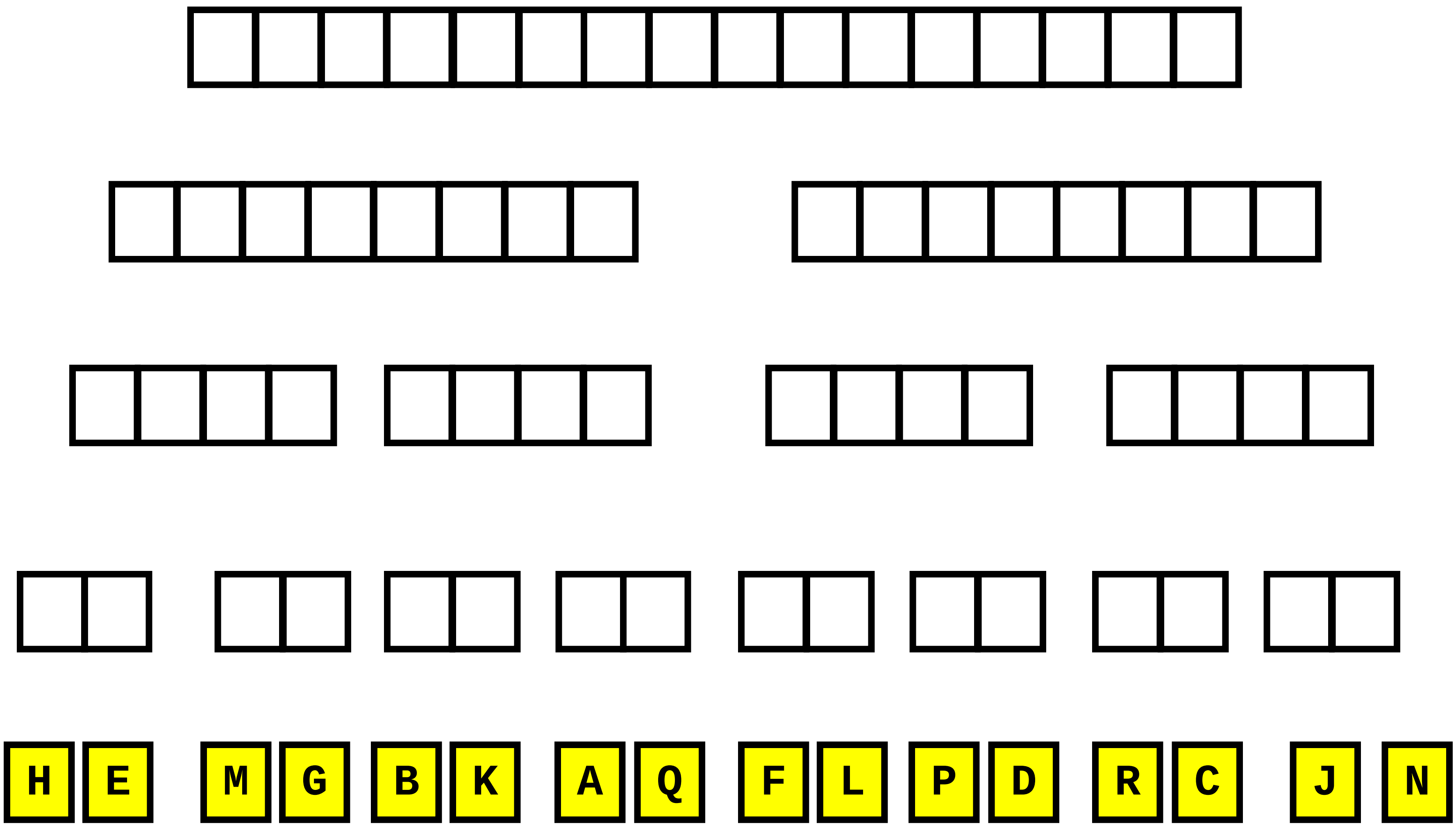
- Previous Lecture:
  - "Divide and conquer" strategies
    - Binary search
    - Merge sort

- Today's Lecture:
  - Some efficiency considerations
  - "Divide and conquer" strategies (cont'd)—recursion
    - Merge sort
    - Removing a character (e.g., the blank) from a string
    - Tiling (subdividing) a triangle, e.g., Sierpinski Triangle

- Announcements
  - Discussion this week in the computer lab (UP B7)
  - Project 7 due Thursday at 11pm
  - CS1112 final will be 12/14 (Mon) 7pm in Barton West

Merge sort is a "divide-and-conquer" strategy



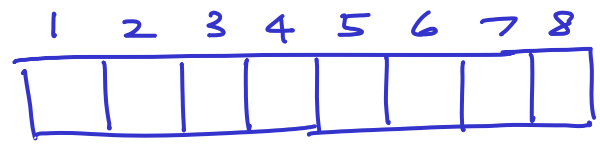
```
function y = mergeSort(x)
% x is a vector.  y is a vector
% consisting of the values in x
% sorted from smallest to largest.

n = length(x);
if n==1
      y = x;
else
      m  = floor(n/2);
      yL = mergeSort(x(1:m));
      yR = mergeSort(x(m+1:n));
      y  = merge(yL,yR);
end
```

```
function y=mergeSort(x)
n=length(x);
if n==1
      y=x;
else
      m=floor(n/2);
      yL=mergeSort(x(1:m));
      yR=mergeSort(x(m+1:n));
      y=merge(yL,yR);
end
```

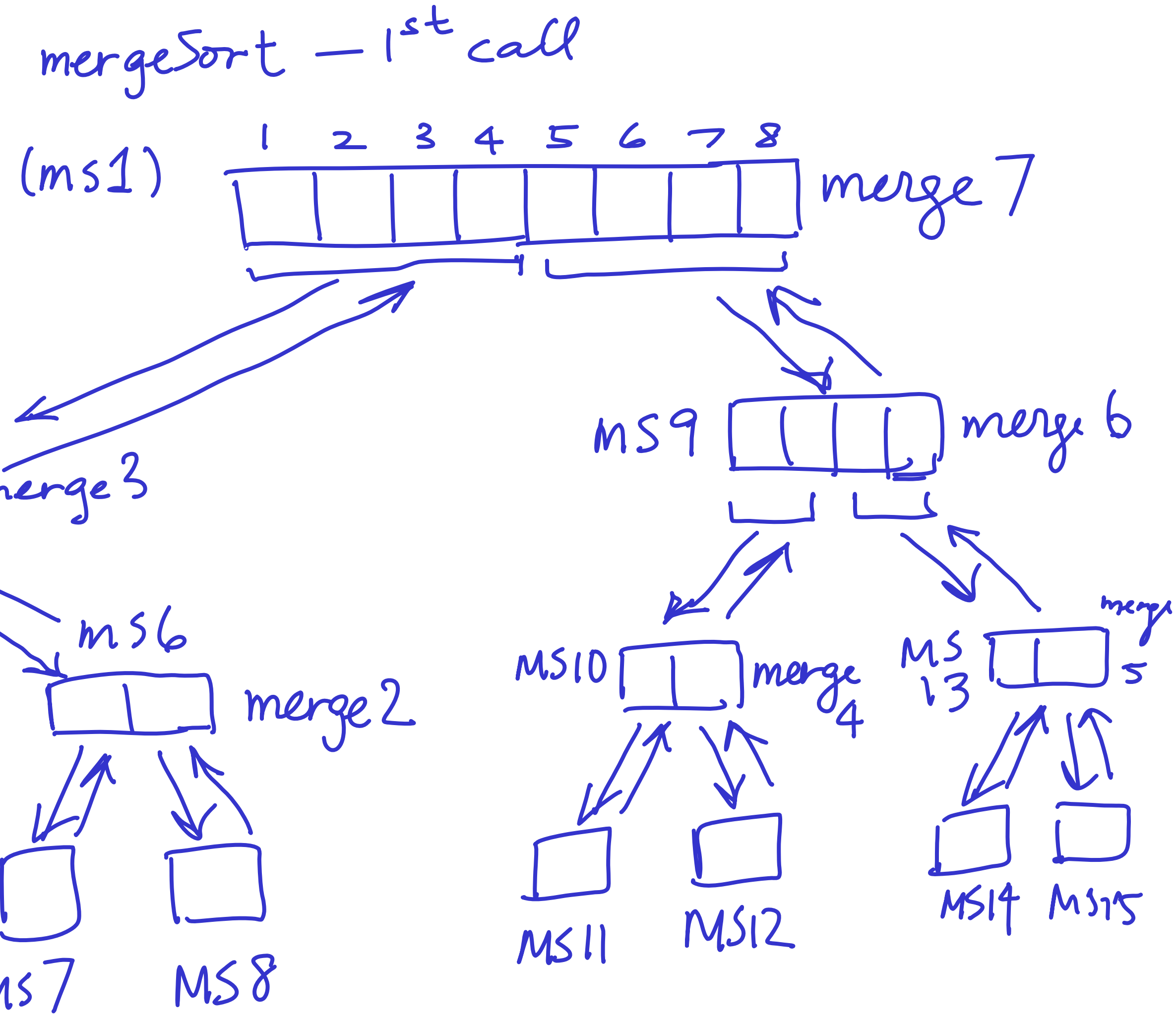
```
function y=mergeSort(x)
n=length(x);
if n==1
      y=x;
else
      m=floor(n/2);
      yL=mergeSort(x(1:m));
      yR=mergeSort(x(m+1:n));
      y=merge(yL,yR);
end
```

mergeSort — 1st call

(ms1)



```
function y=mergeSort(x)
n=length(x);
if n==1
      y=x;
else
      m=floor(n/2);
      yL=mergeSort(x(1:m));
      yR=mergeSort(x(m+1:n));
      y=merge(yL,yR);
end
```



How do merge sort, insertion sort, and bubble sort compare?

- Insertion sort and bubble sort are similar
  - Both involve a series of comparisons and swaps
  - Both involve nested loops
- Merge sort uses recursion

```
function x = insertSort(x)
% Sort vector x in ascending order with insertion sort

n = length(x);
for i= 1:n-1
   % Sort x(1:i+1) given that x(1:i) is sorted
   j= i;
   need2swap= x(j+1) < x(j);
   while  need2swap

      % swap x(j+1) and x(j)
      temp= x(j);
      x(j)= x(j+1);
      x(j+1)= temp;

      j= j-1;
      need2swap= j>0 && x(j+1)<x(j);
   end
end
```

Insertion sort is more efficient than bubble sort on average—fewer comparisons (Lecture 24)

How do merge sort and insertion sort compare?

- Insertion sort: (worst case) makes i comparisons to insert an element in a sorted array of i elements. For an array of length N:

  $1+2+\ldots+(N-1) = N(N-1)/2$, say N^2 for big N

- Merge sort:

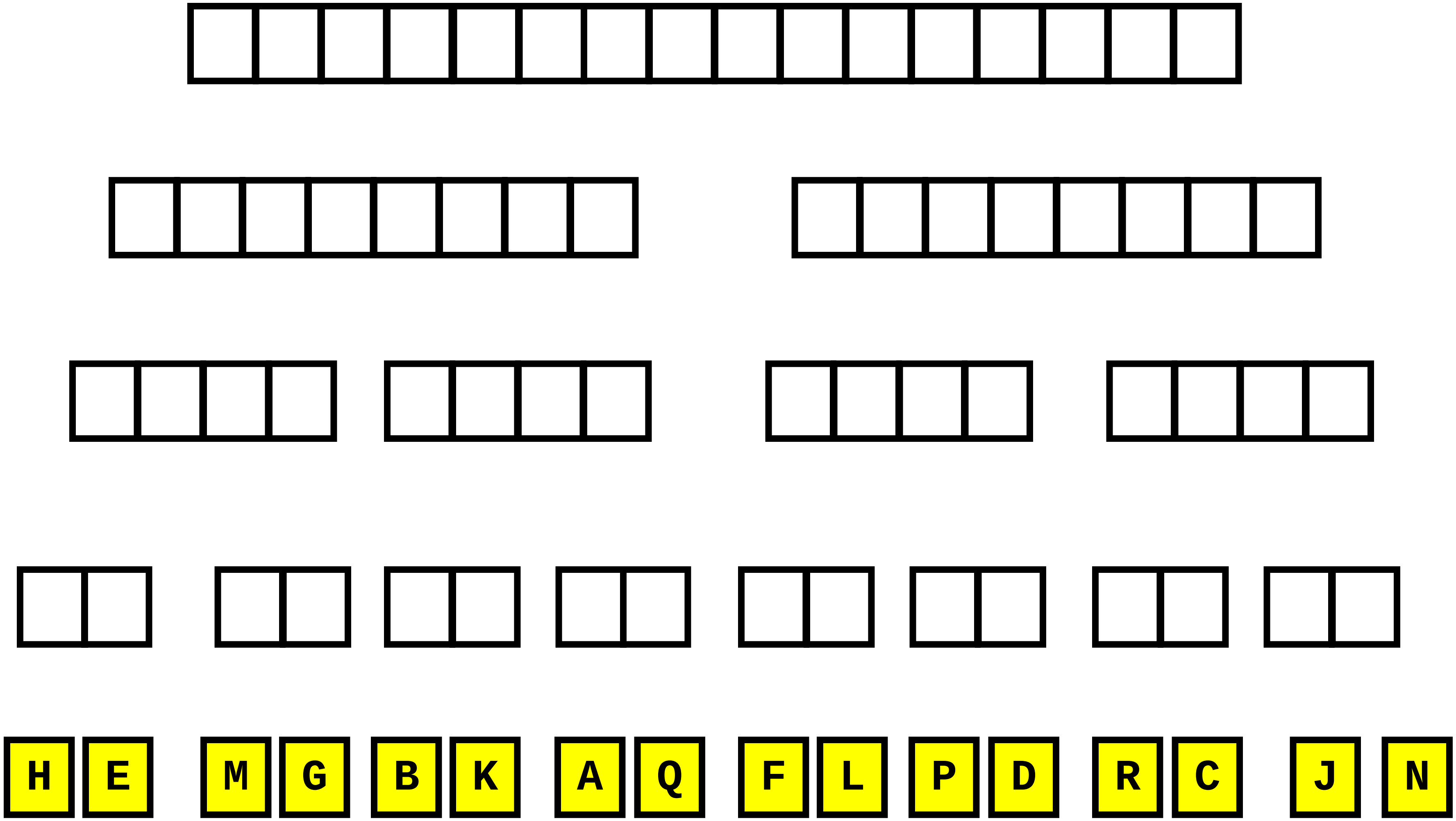
```
function y = mergeSort(x)
% x is a vector.  y is a vector
% consisting of the values in x
% sorted from smallest to largest.

n = length(x);
if n==1
      y = x;
else
      m  = floor(n/2);
      yL = mergeSort(x(1:m));
      yR = mergeSort(x(m+1:n));
      y  = merge(yL,yR);
end
```

All the comparisons between vector values are done in **merge**

```
function z = merge(x,y)
nx = length(x); ny = length(y);
z = zeros(1, nx+ny);
ix = 1; iy = 1; iz = 1;
while ix<=nx && iy<=ny
    if  x(ix) <= y(iy)
        z(iz)= x(iy);  ix=ix+1;  iz=iz+1;
    else
        z(iz)= y(iy);  iy=iy+1;  iz=iz+1;
    end
end
while ix<=nx  % copy remaining x-values
  z(iz)= x(ix);  ix=ix+1;  iz=iz+1;
end
while iy<=ny  % copy remaining y-values
  z(iz)= y(iy);  iy=iy+1;  iz=iz+1;
end
```

See Section Ex. 13 for the code to count the # of coomparisons

Merge sort: $\log_2(N)$ "levels"; N comparisons each level



How do merge sort and insertion sort compare?

- Insertion sort: (worst case) makes i comparisons to insert an element in a sorted array of i elements. For an array of length N:

  $1+2+\ldots+(N-1) = N(N-1)/2$, say N^2 for big N

  $O(N^2)$

- Merge sort: $N \cdot \log_2(N)$

  $O(\ N \log_2(N)\ )$

  Order of magnitude

- Insertion sort is done *in-place*; merge sort (recursion) requires much more memory

How to choose??

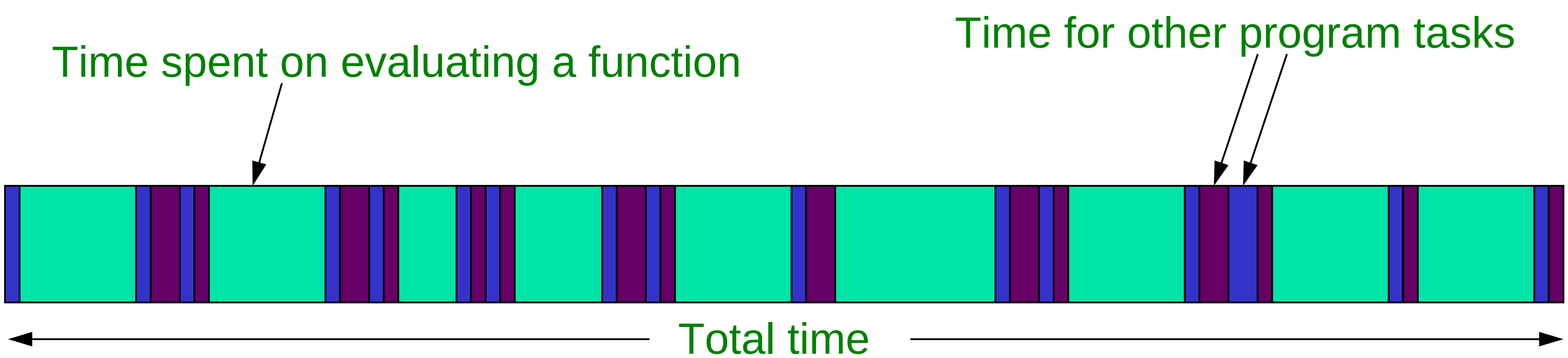
- Depends on application
- Merge sort is especially good for sorting large data set (but watch out for memory usage)
- Insertion sort is "order N^2" at worst case, but what about an average case? If the application requires that you *maintain* a sorted array, insertion sort may be a good choice

Why not just use Matlab's sort function?

- Flexibility
- E.g., to maintain a sorted list, just write the code for insertion sort
- E.g., sort strings or other complicated structures
- Sort according to some criterion set out in a function file
  - Observe that we have the comparison `x(j+1)<x(j)`
  - The comparison can be a function that returns a boolean value
- Can combine different sort/search algorithms for specific problem

Expensive function evaluations

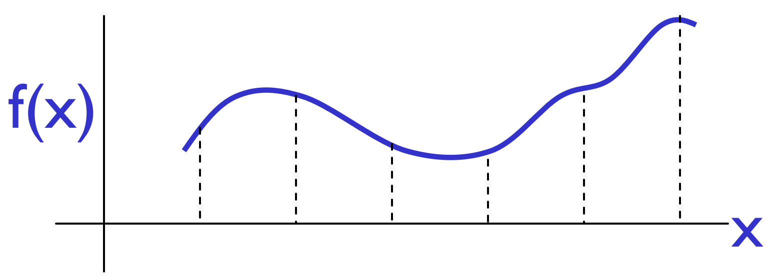
- Consider the execution of a program that is dominated by multiple calls to an expensive-to-evaluate function (e.g., climate simulation models)



- 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?

x	f(x)
1	1.01
2	2.67
3	5.71
4	9.12
5	7.98
:	:

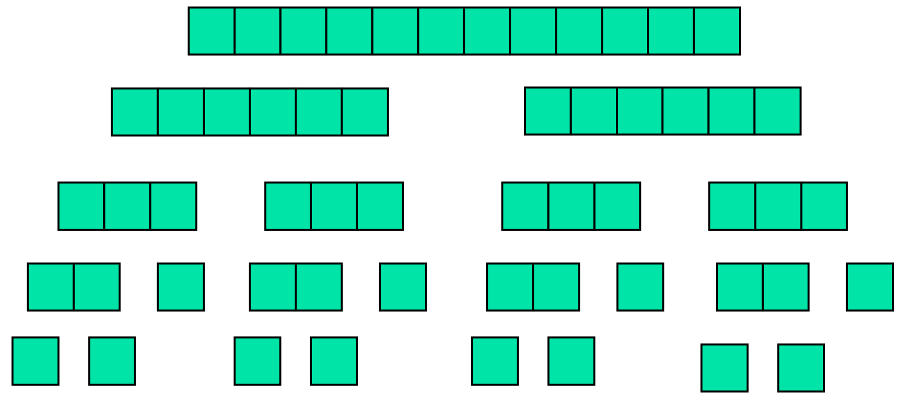
Pre-calculate and store these values (e.g., in a vector **H**)

- Accuracy—need a "dense grid" to get high accuracy → 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)?

To be continued in this week's lab.

Back to Recursion

- Merge sort



- 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
- Can solve using ***recursion***
  - E.g., remove all the blanks in string s

    Same as remove blank in s(1)

    and 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




  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
```

```
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) 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
```

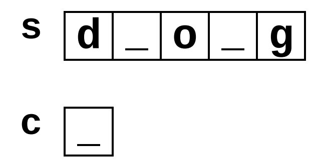
s	d	_	o	_	g

c	_

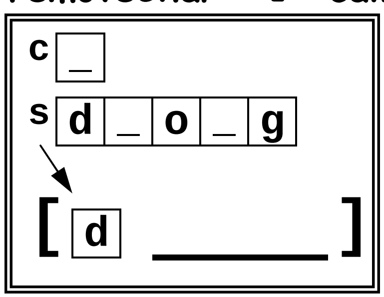
removeChar – 1st call

c	_				
s	d	_	o	_	g

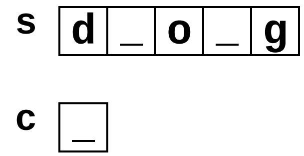
```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```

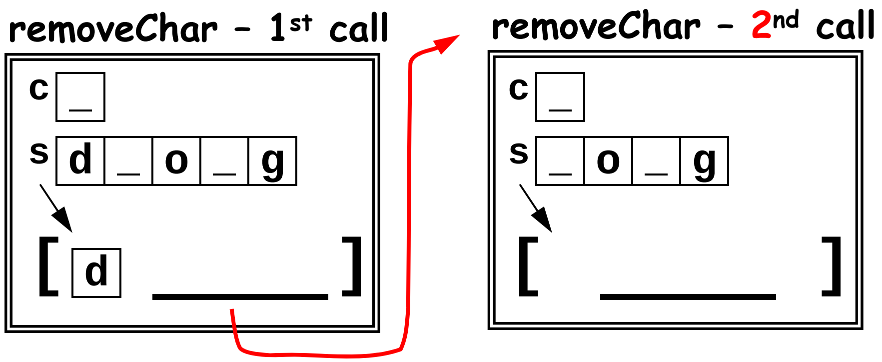


removeChar – 1st call

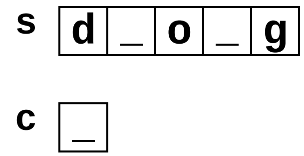


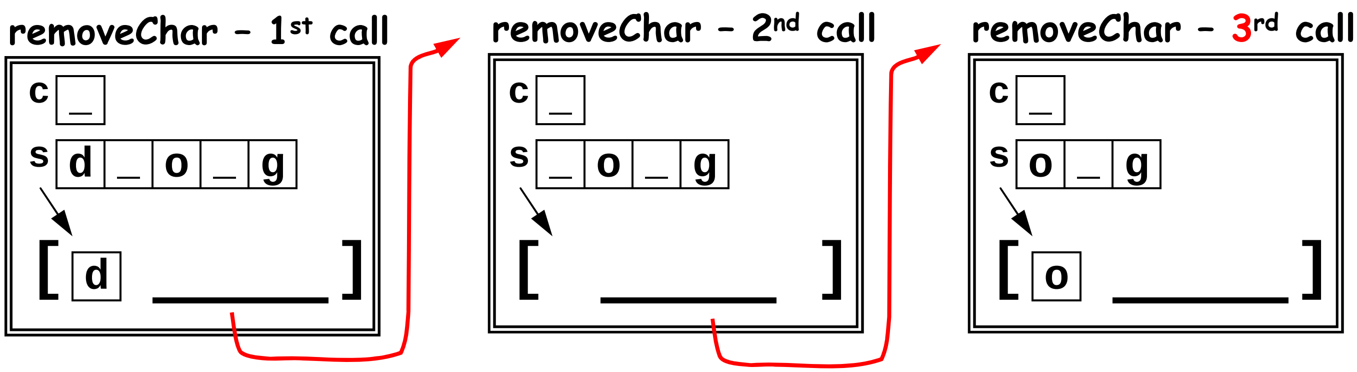
```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```



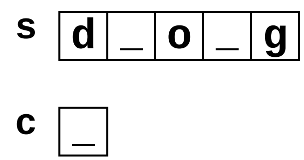


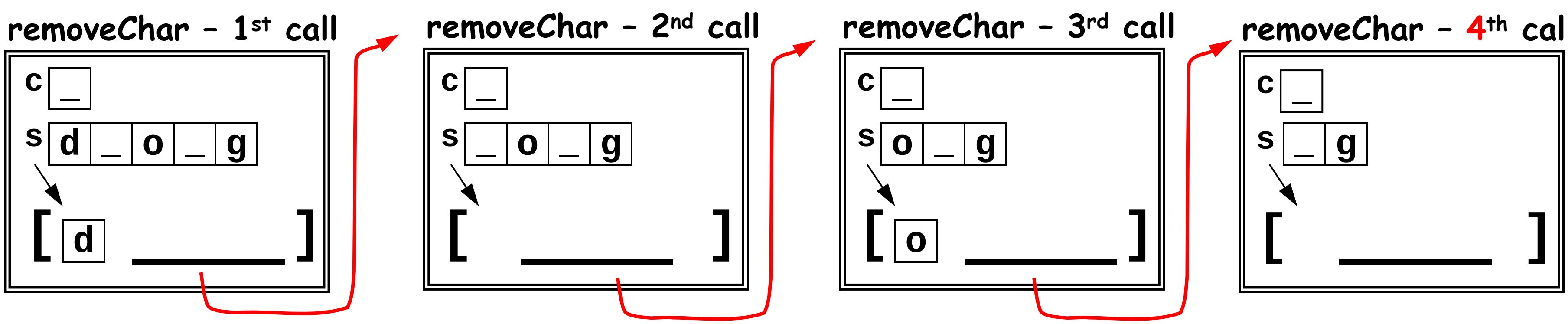
```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```



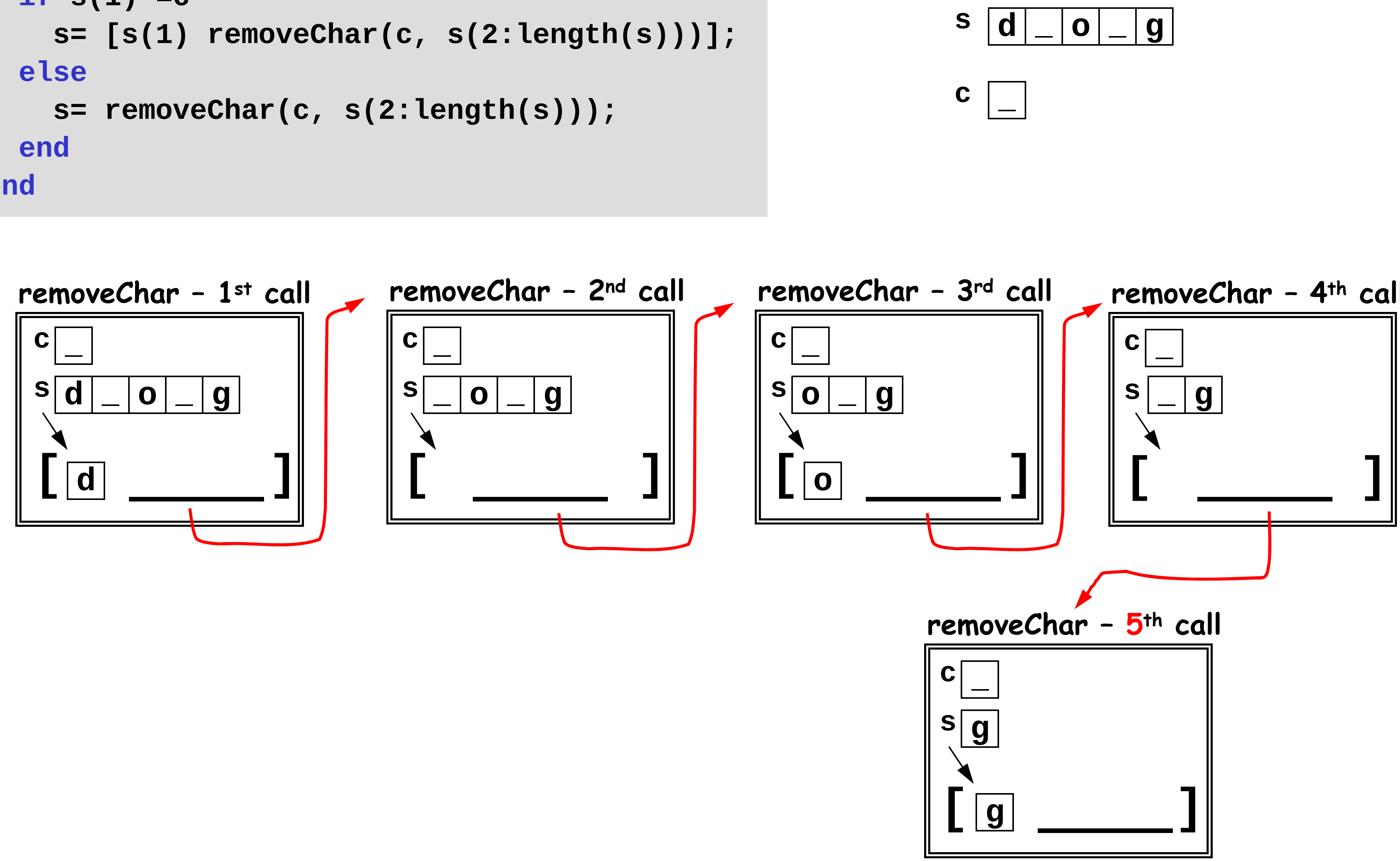


```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```

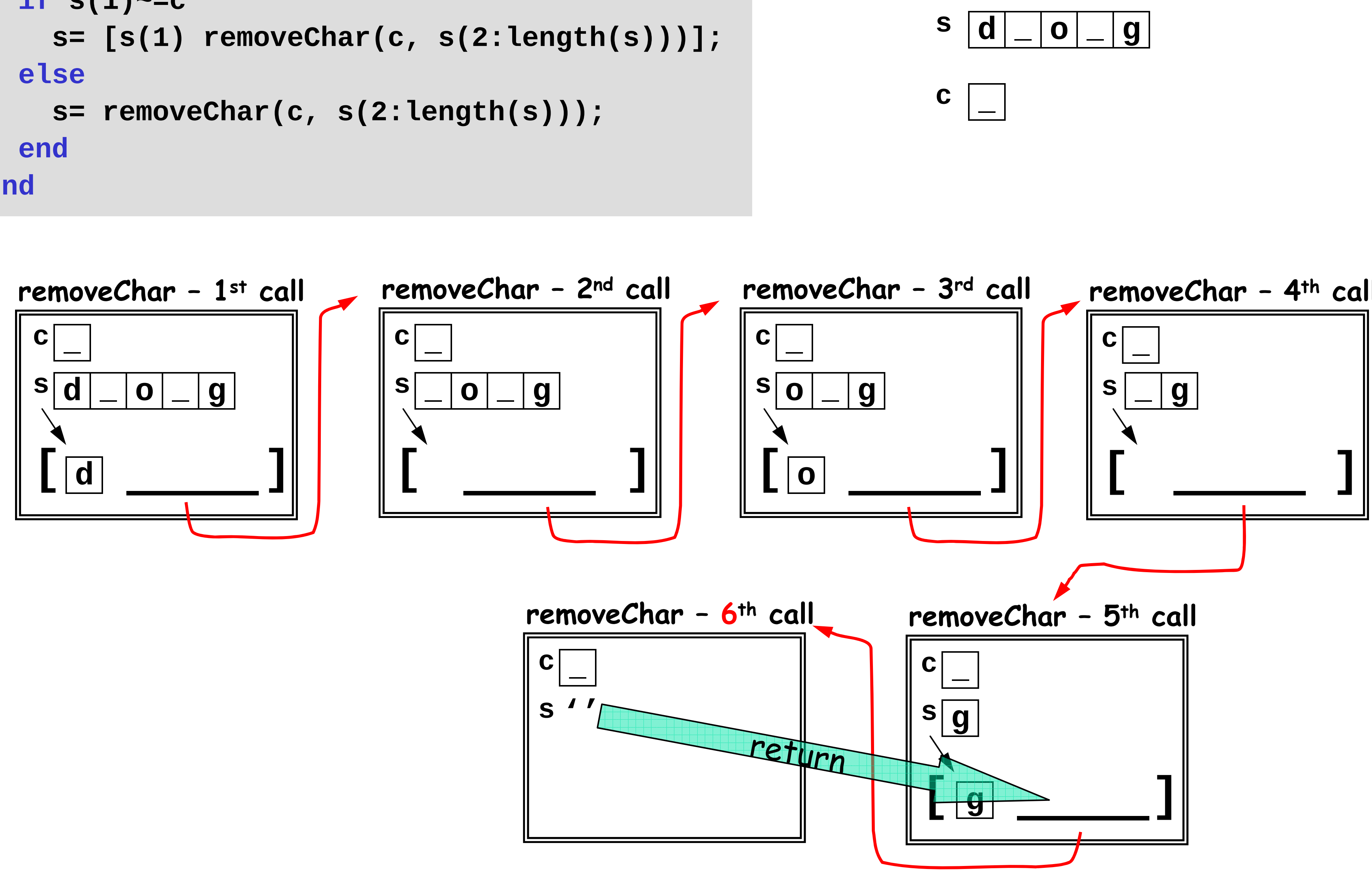




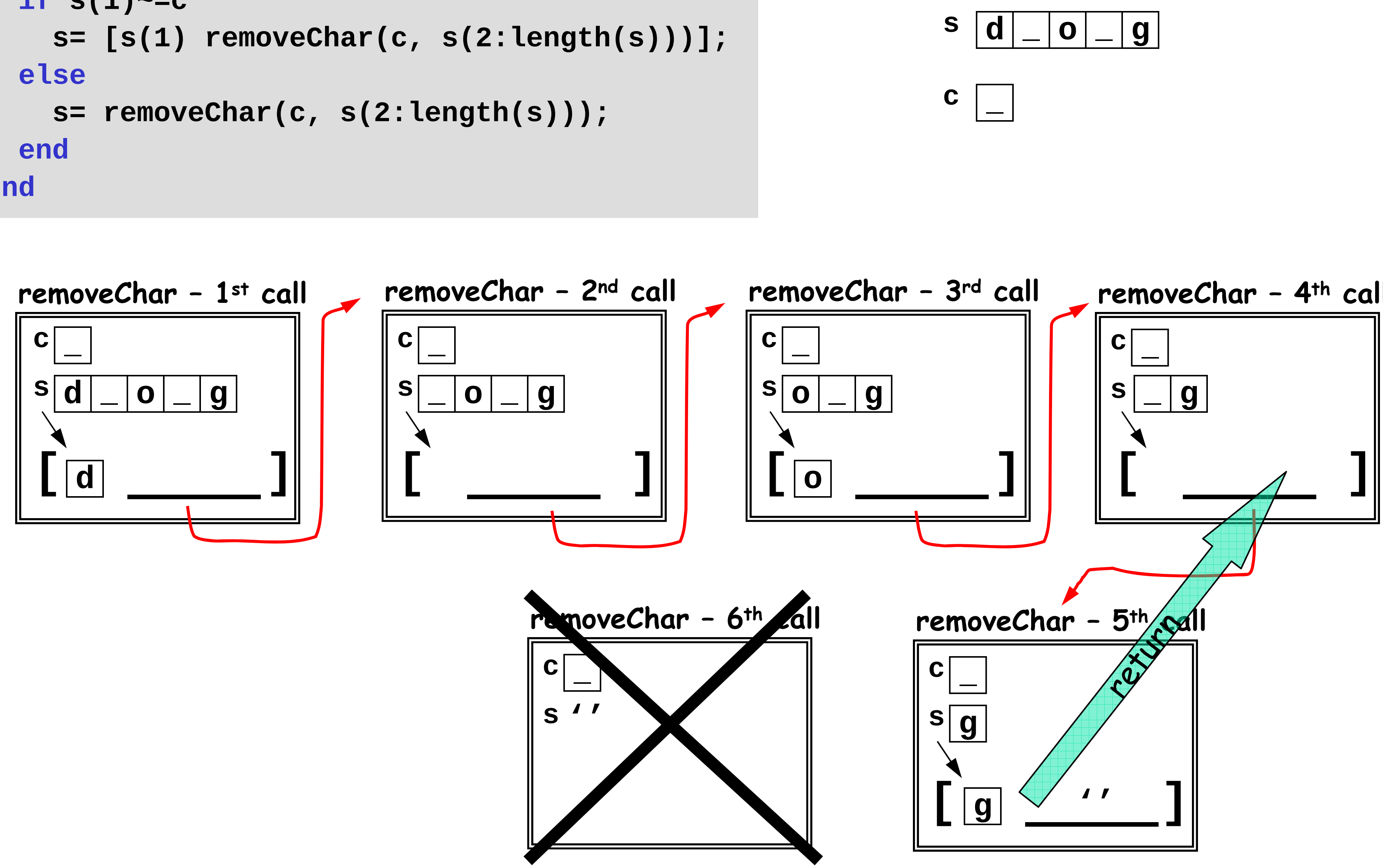
```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```



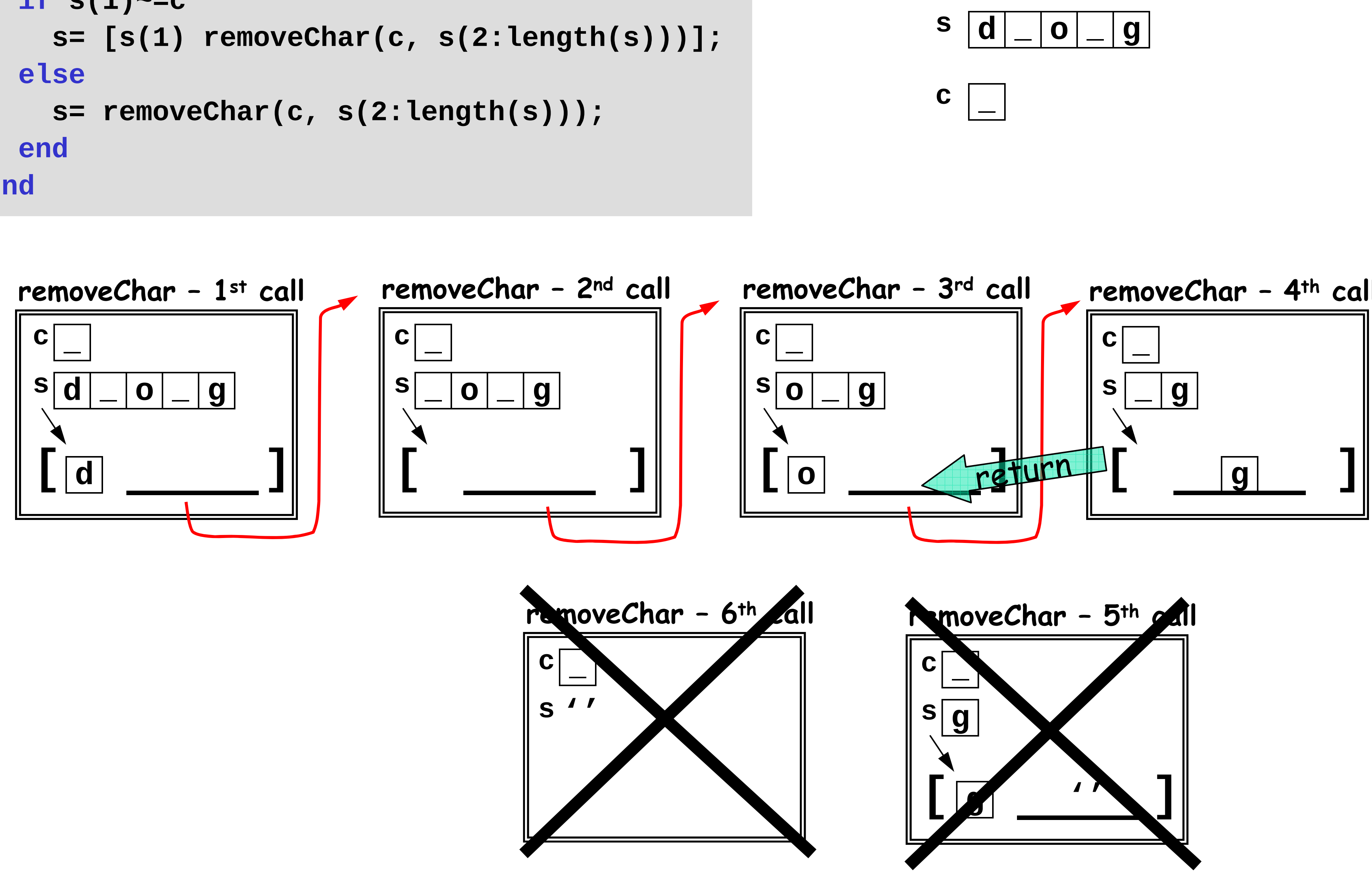
```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```



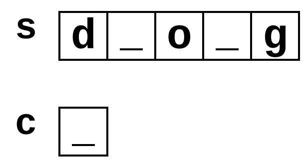
```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```

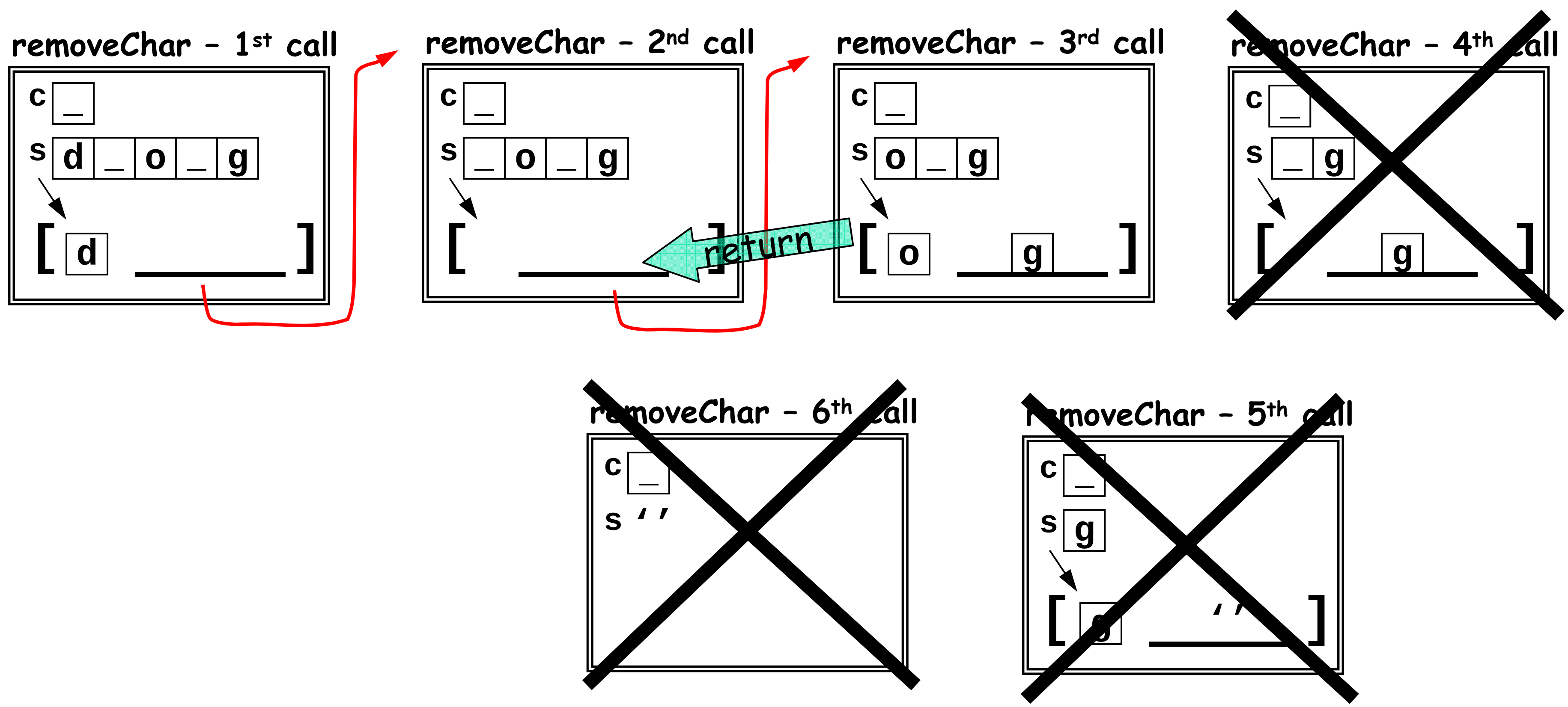


```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```

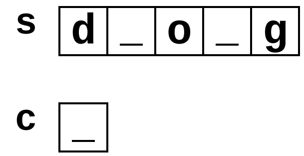


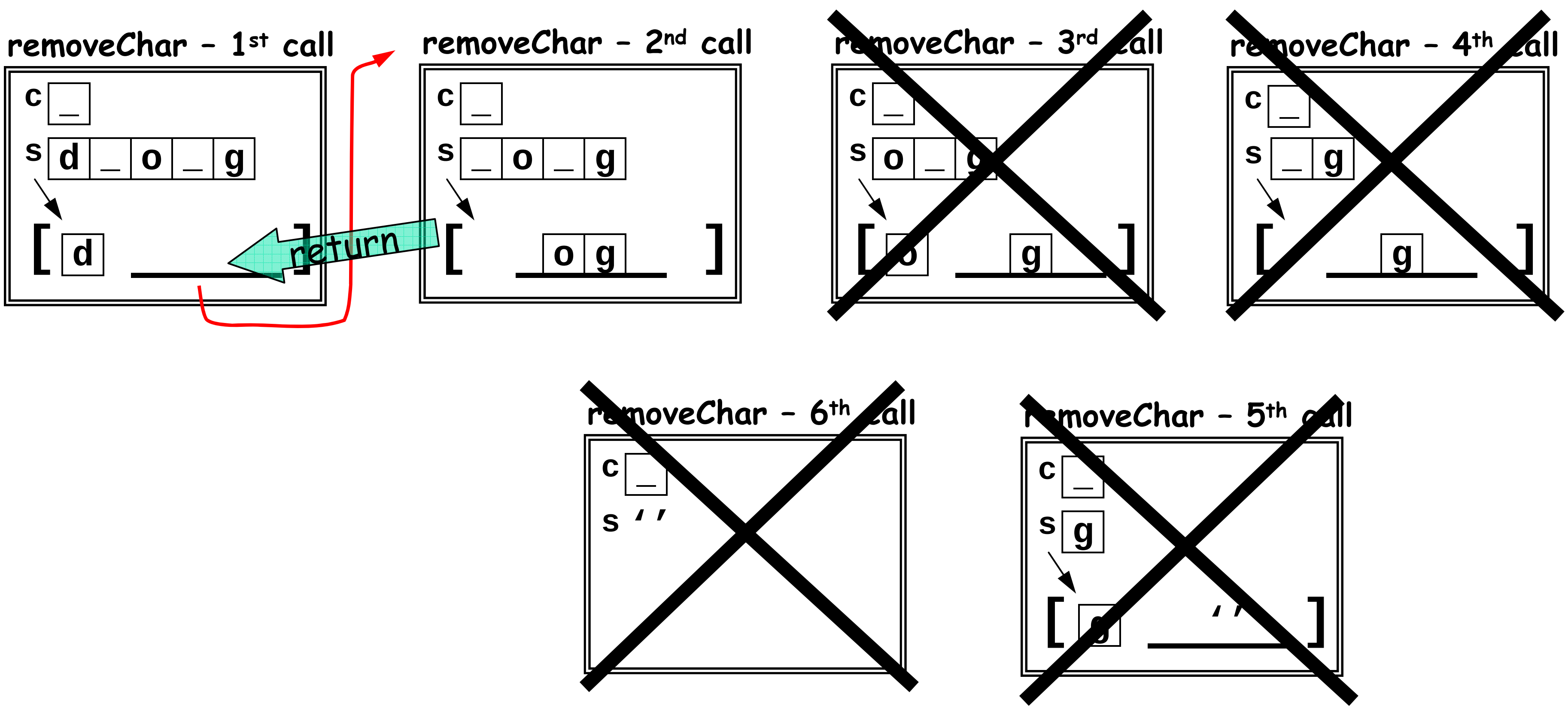
```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```



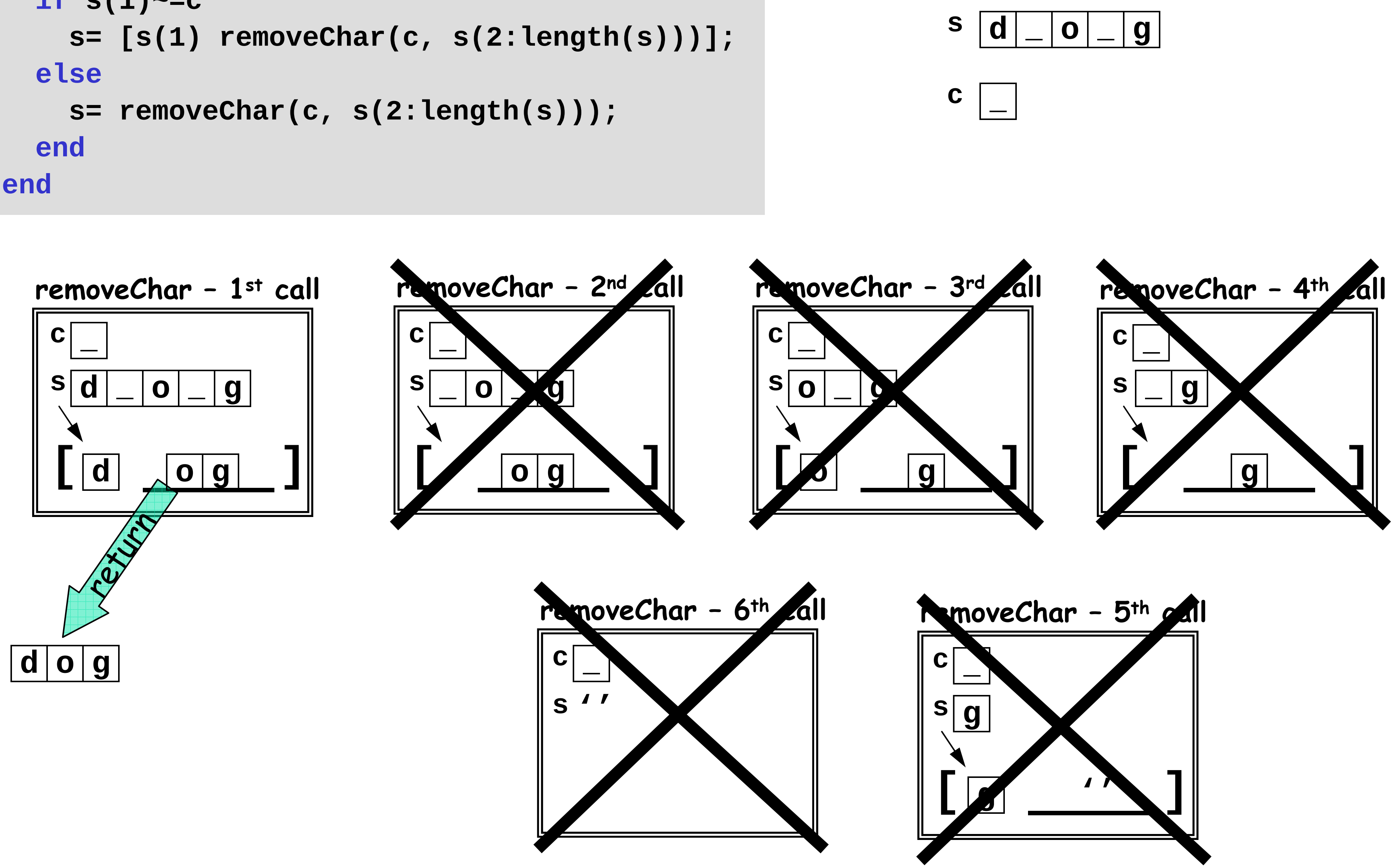


```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```



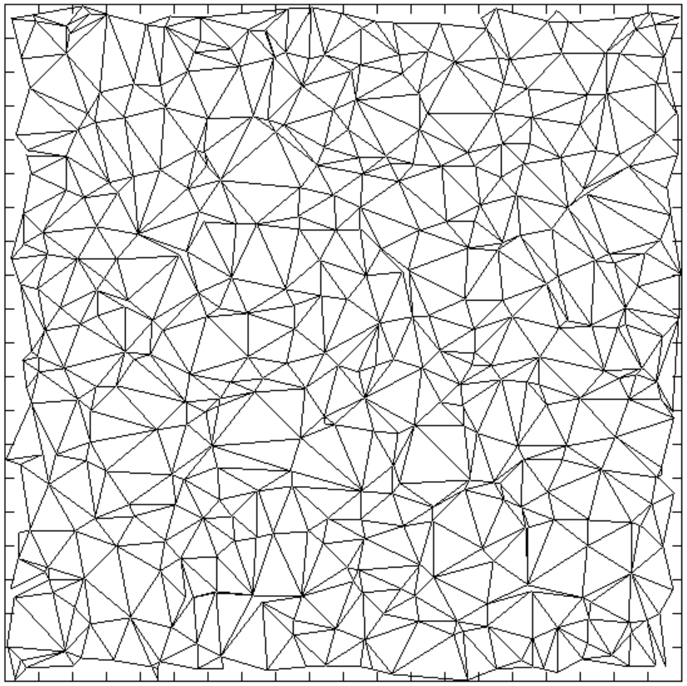


```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```



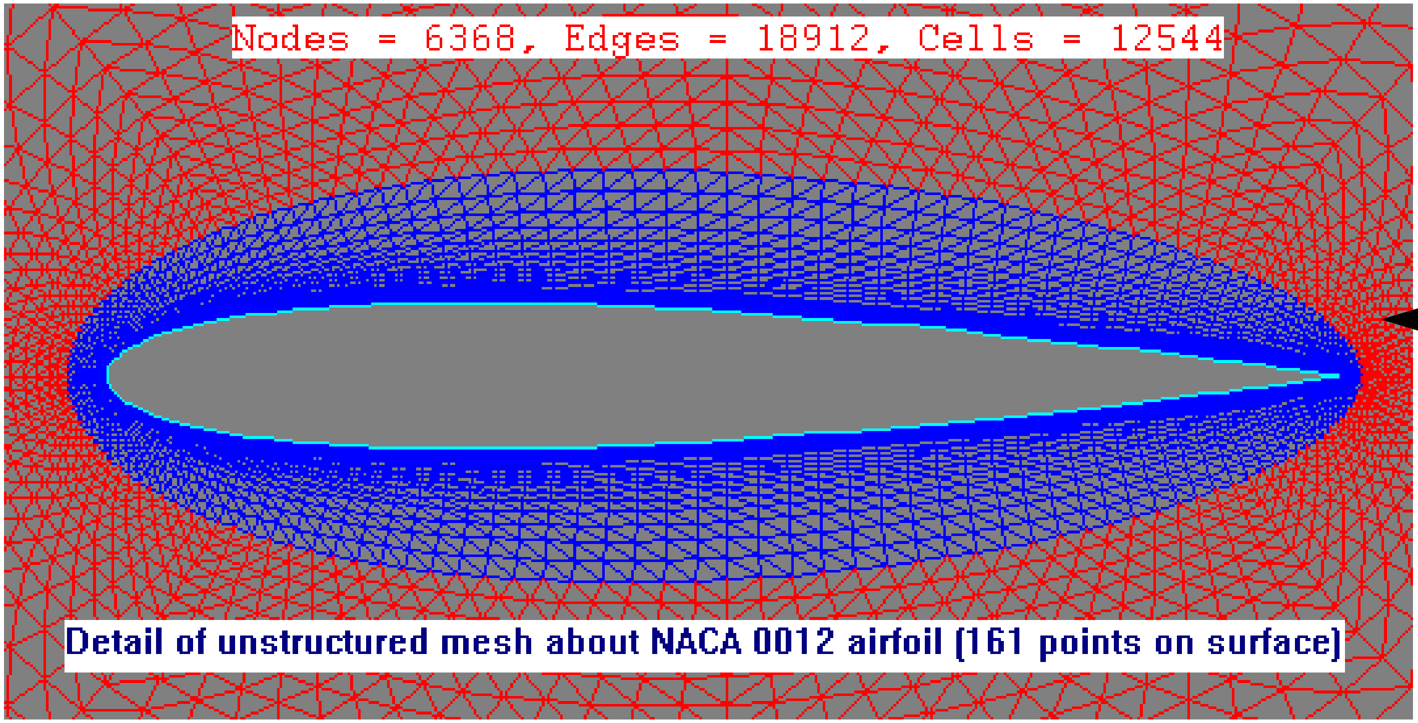
Divide-and-conquer methods also show up in geometric situations

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



Recursive mesh generation

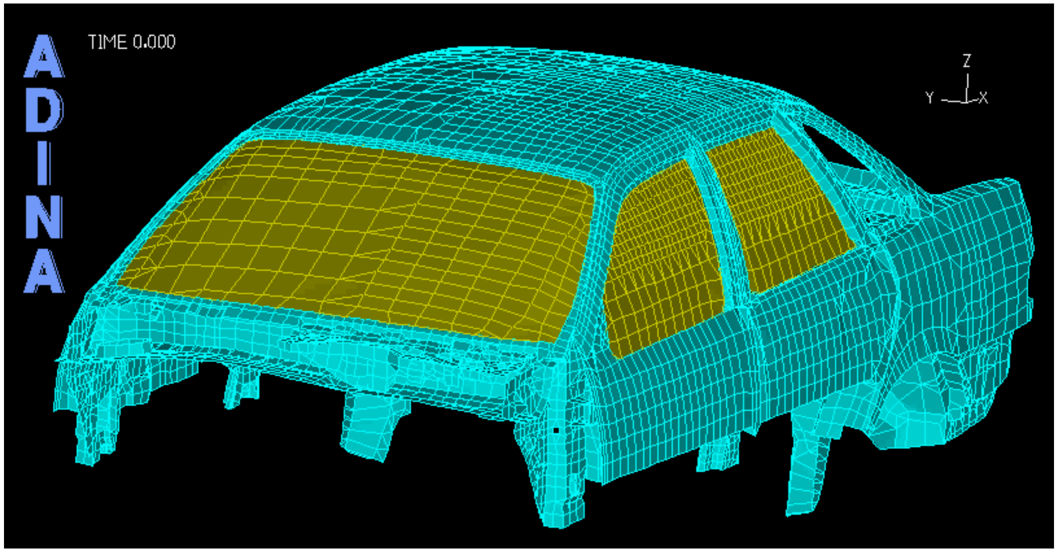
Mesh Generation



An area of interest

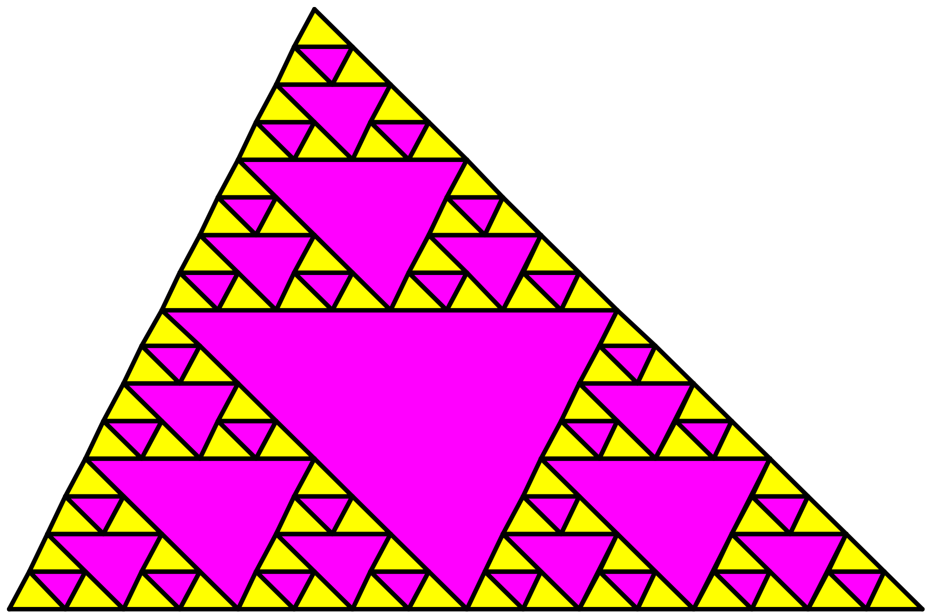
Step one in simulating flow around an airfoil is to generate a mesh and (say) estimate velocity at each mesh point.

Mesh Generation in 3D

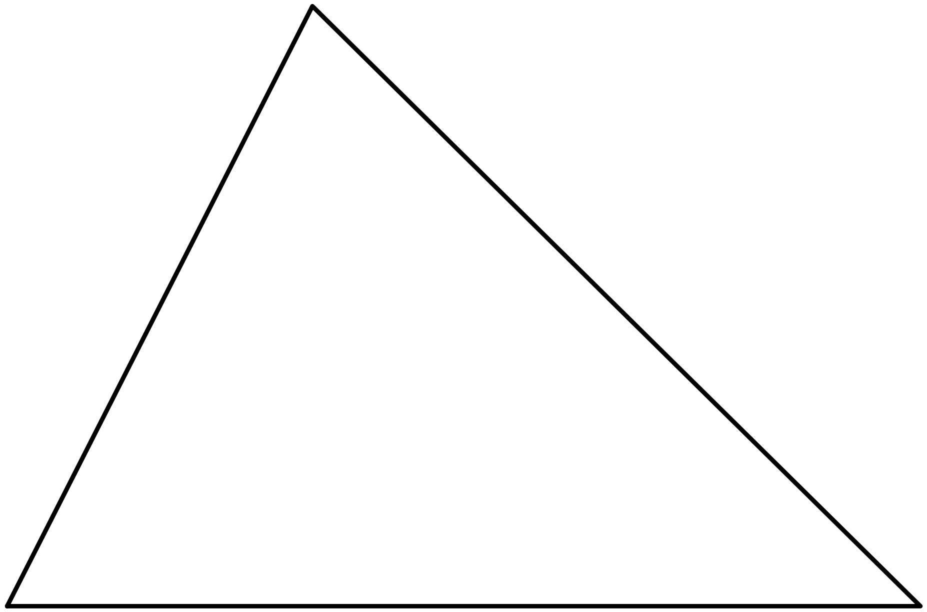


Why is mesh generation a divide-&-conquer process?

Let's draw this graphic

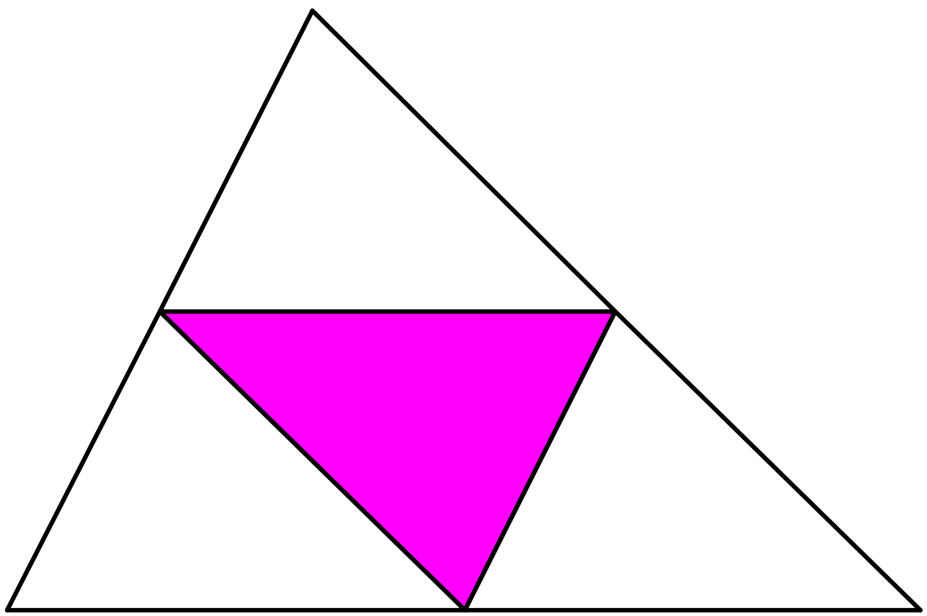


Start with a triangle



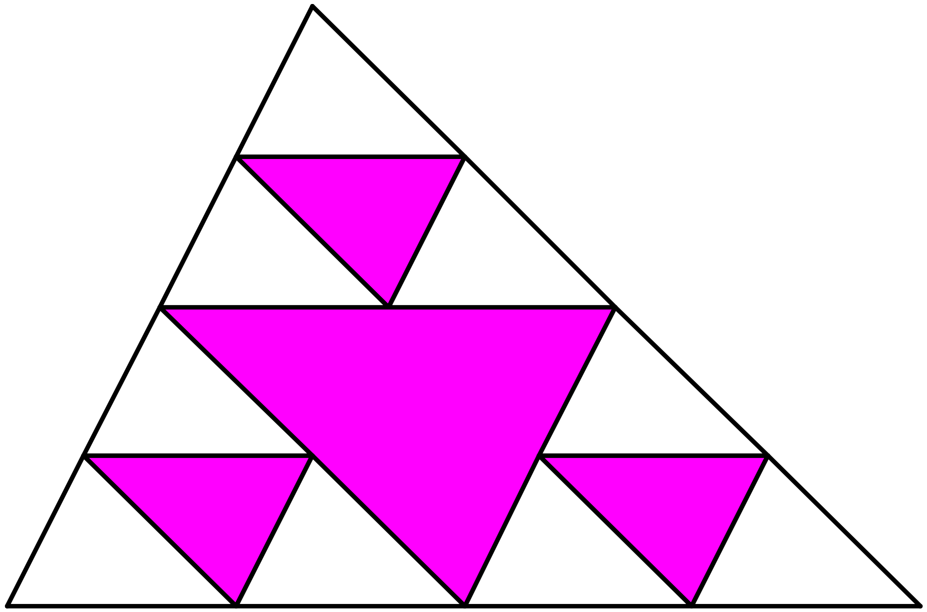
A “level-1” partition of the triangle

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

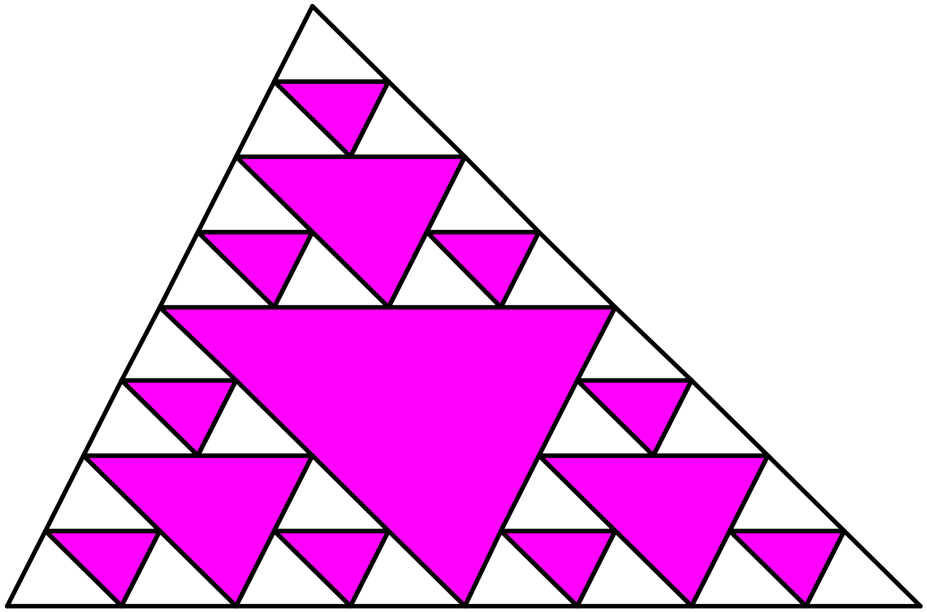


Now do the same partitioning (connecting midpts) on each corner (~~yellow~~ white) triangle to obtain the “level-2” partitioning

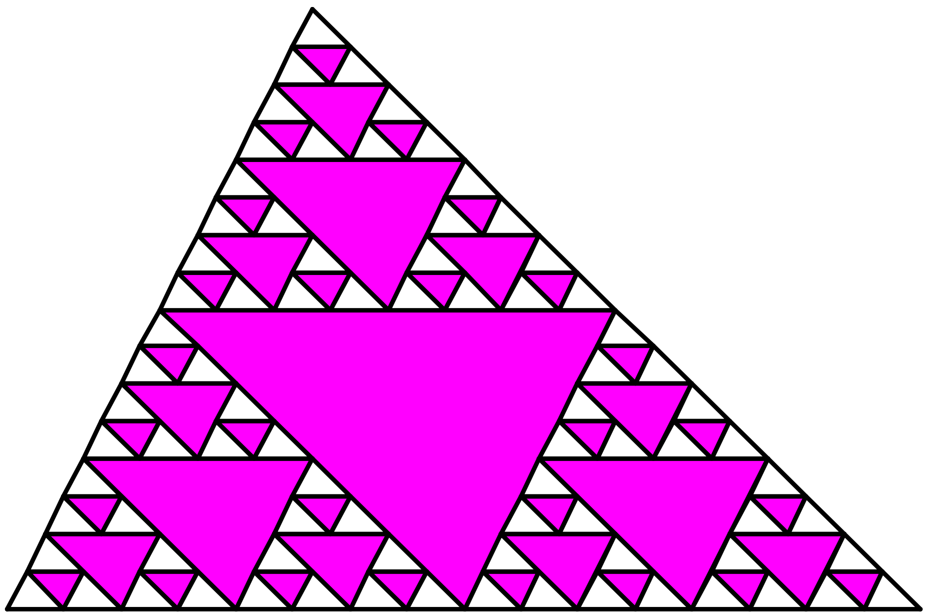
The "level-2" partition of the triangle



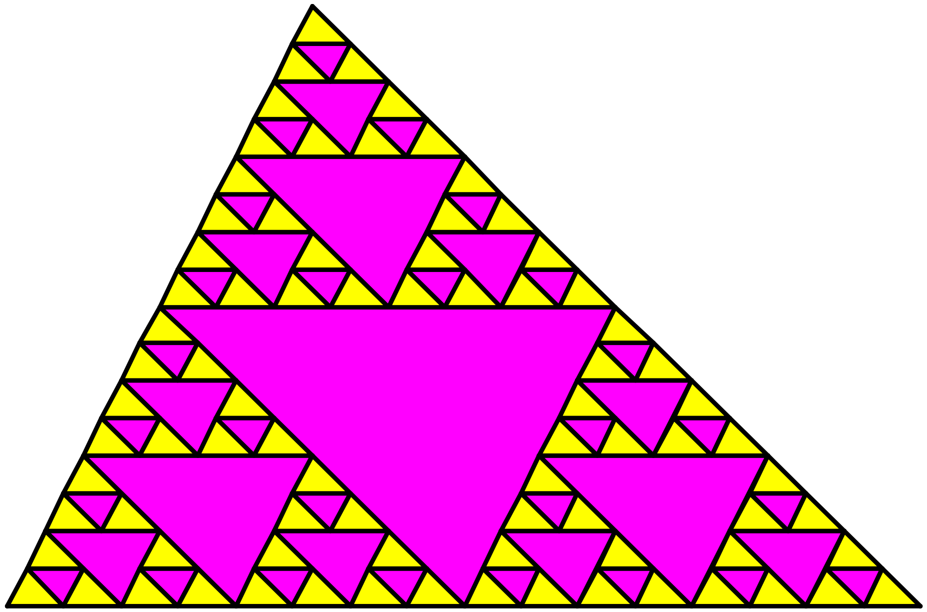
The "level-3" partition of the triangle



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;

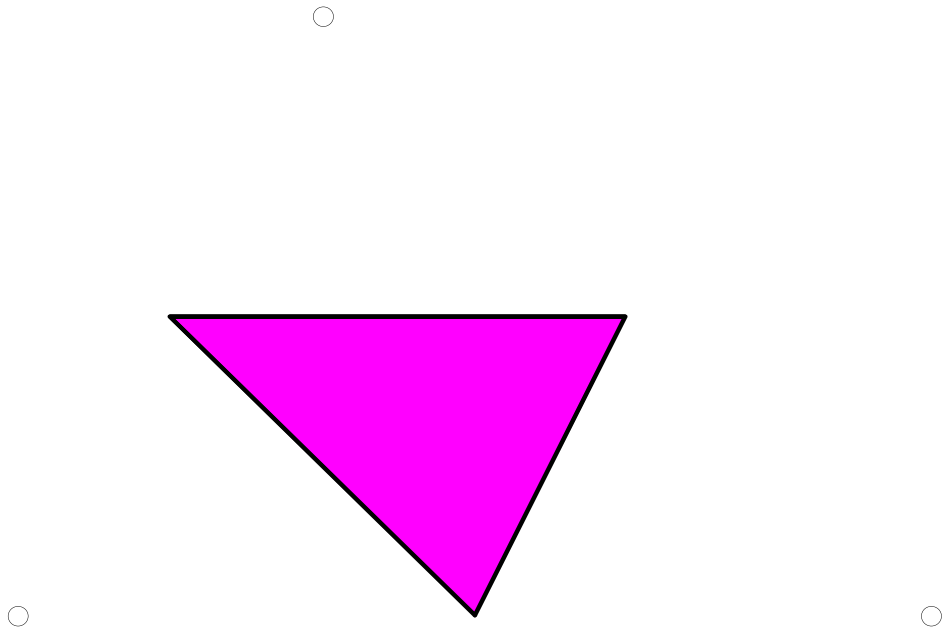
color the interior triangle magenta;

Apply same process to each outer triangle.

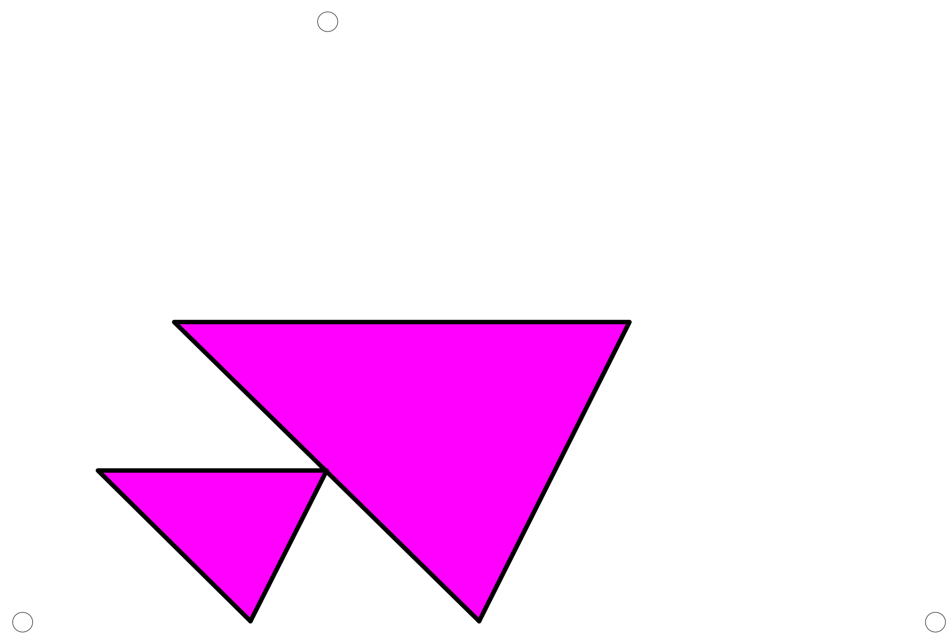
end

Draw a level-4 partition of the triangle with these vertices

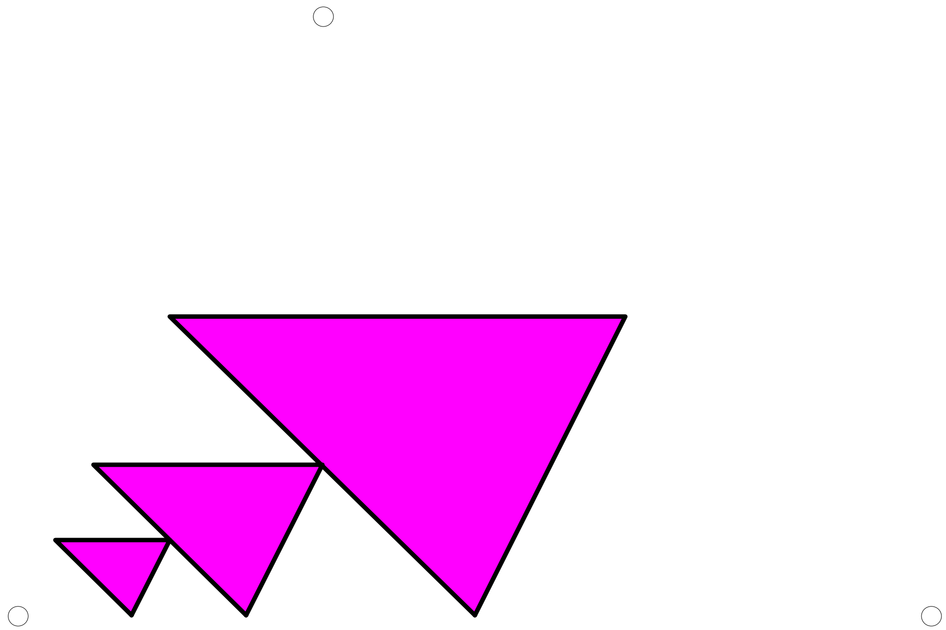
At the start…



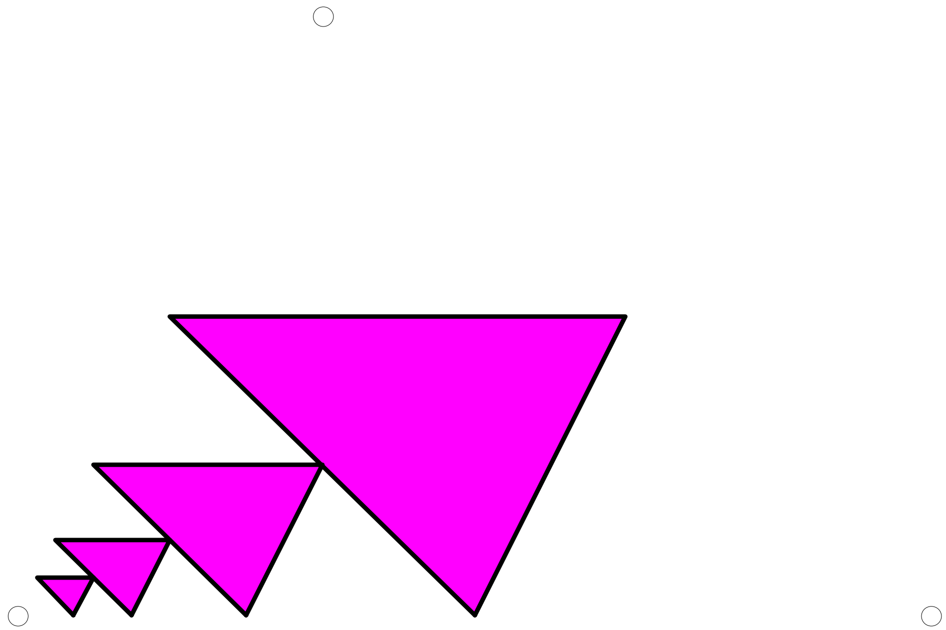
Recur: apply the same process on the lower left triangle



Recur again

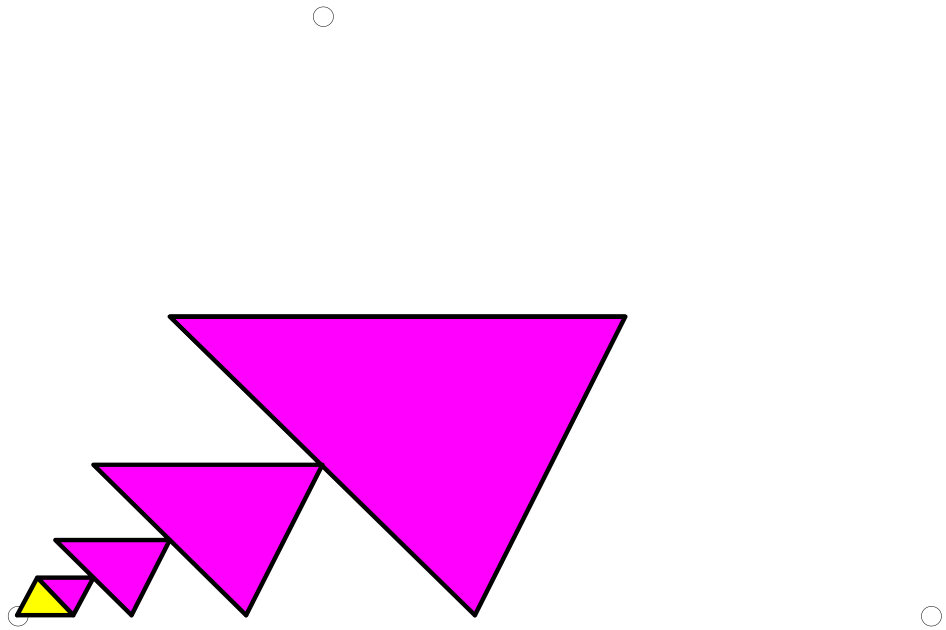


… and again

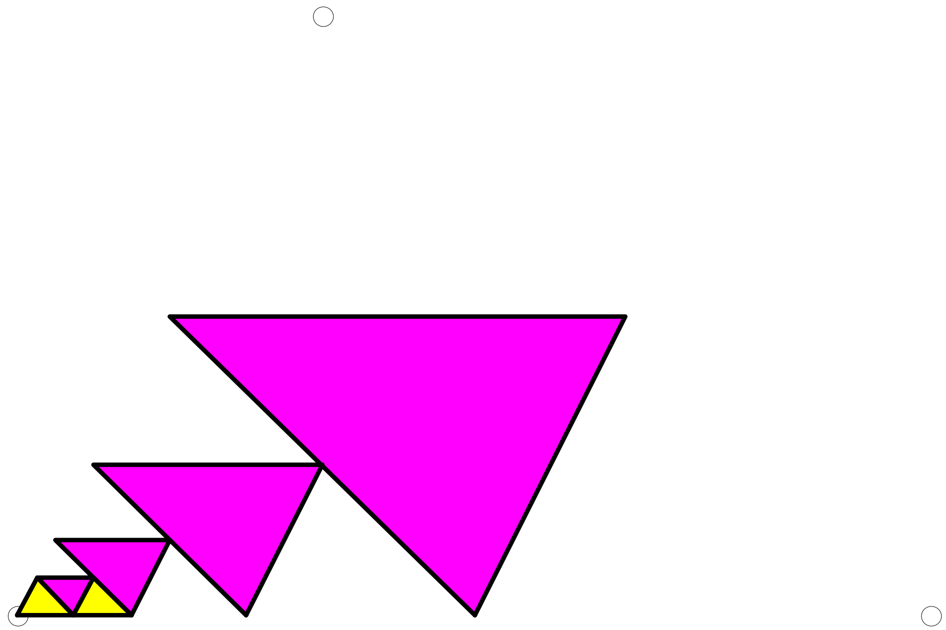


The next lower left corner triangle (white) is small—no more subdivision and just color it yellow.

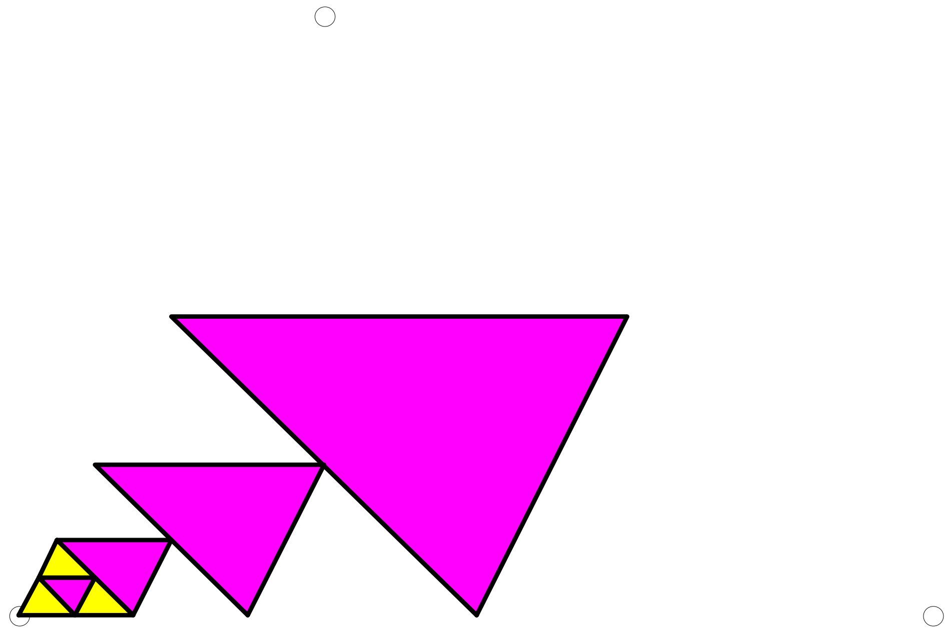
Now lower left corner triangle of the "level-4" partition is done. Continue with another corner triangle

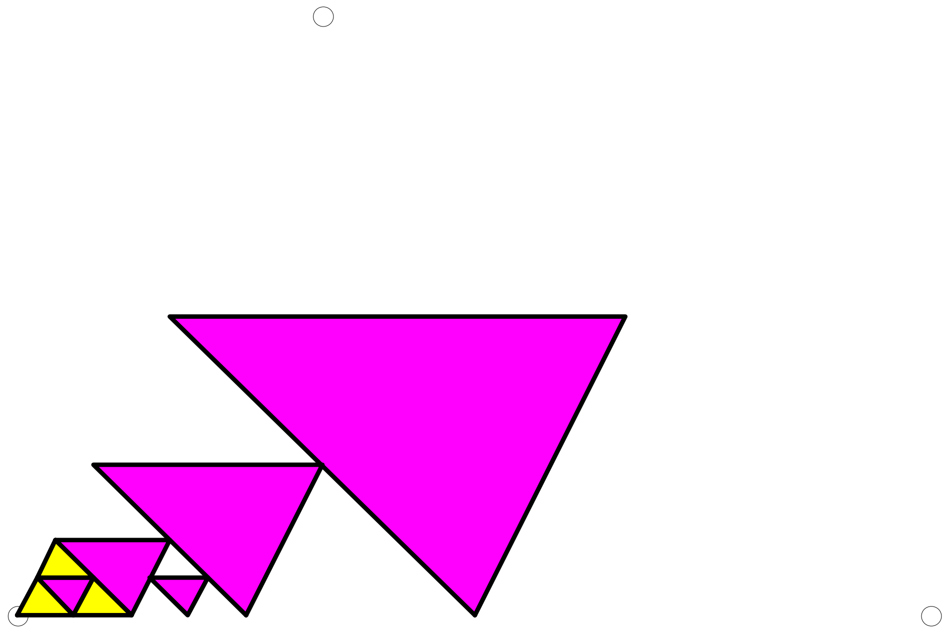


… and continue

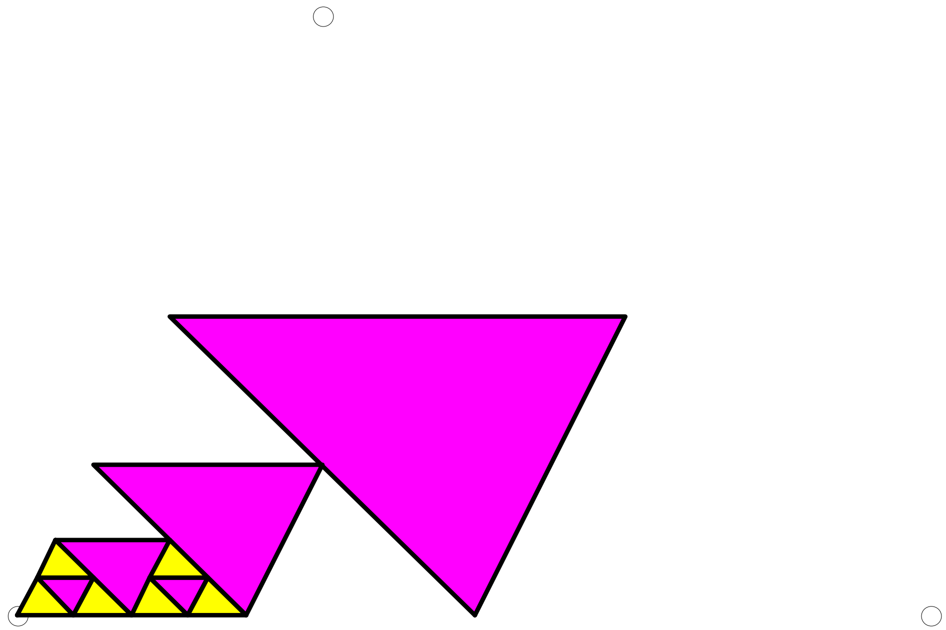


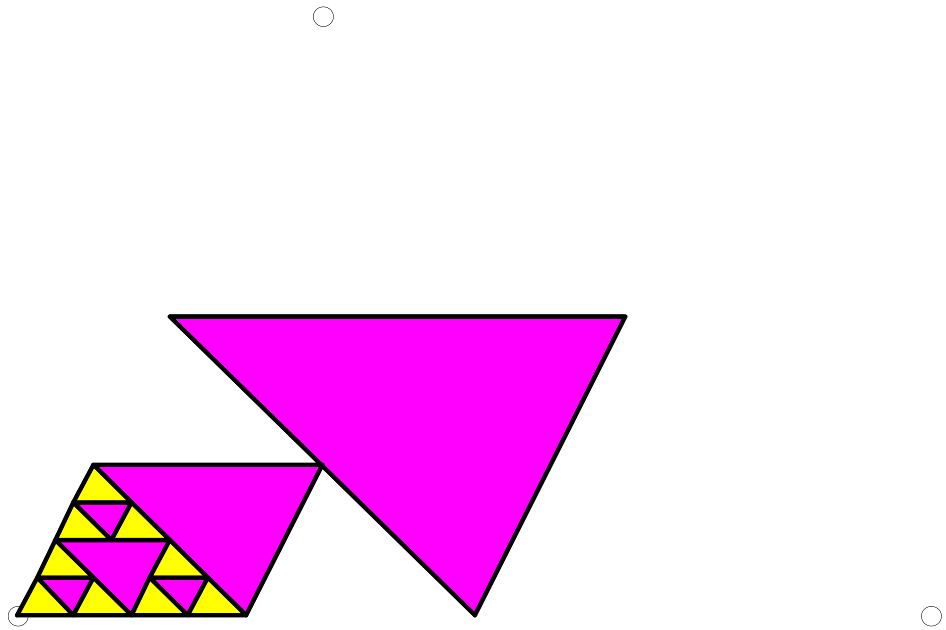
Now the lower left corner triangle of the "level-3" partition is done. Continue with another corner triangle…

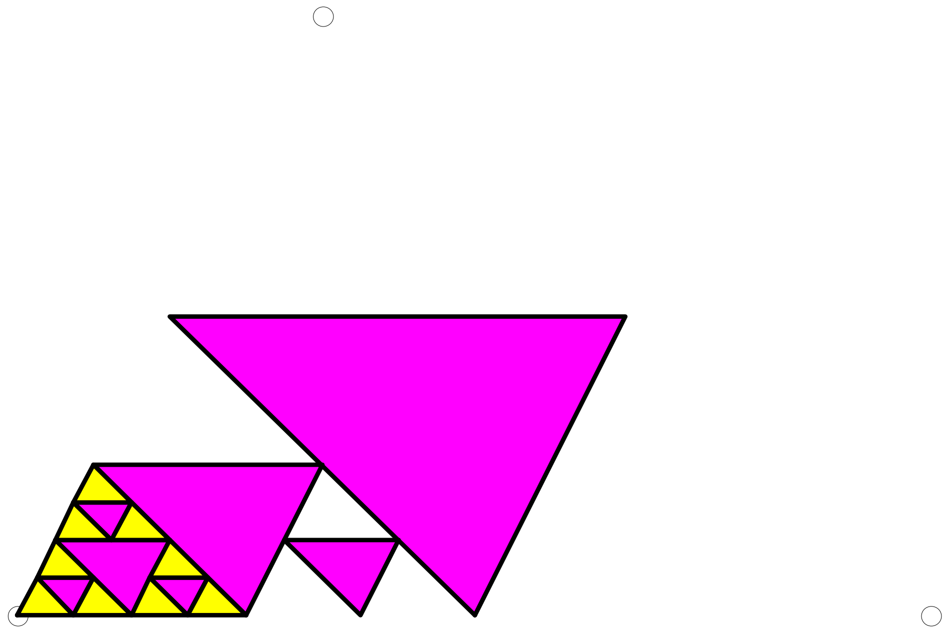


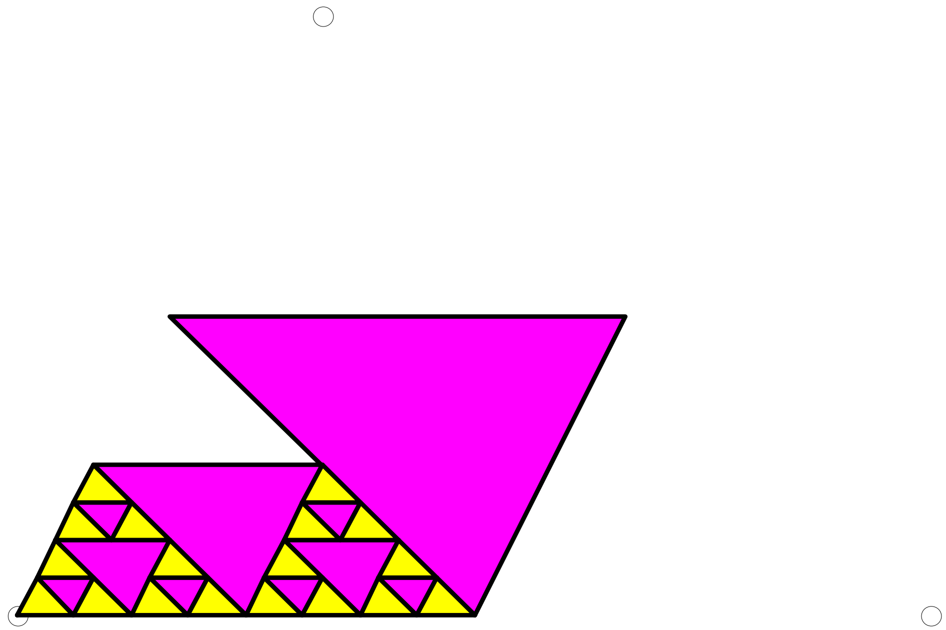


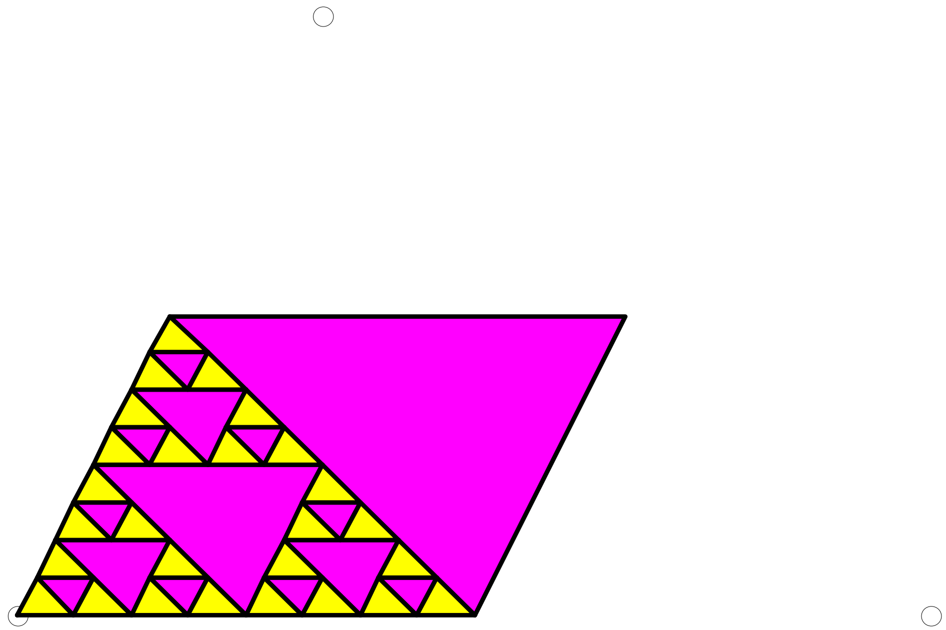
We're "climbing our way out" of the deepest level of partitioning

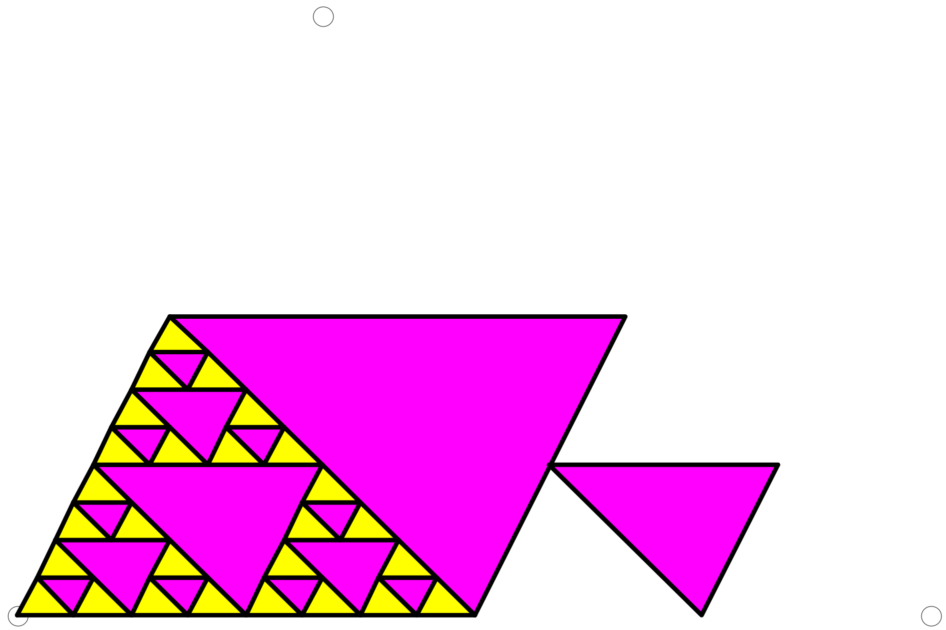


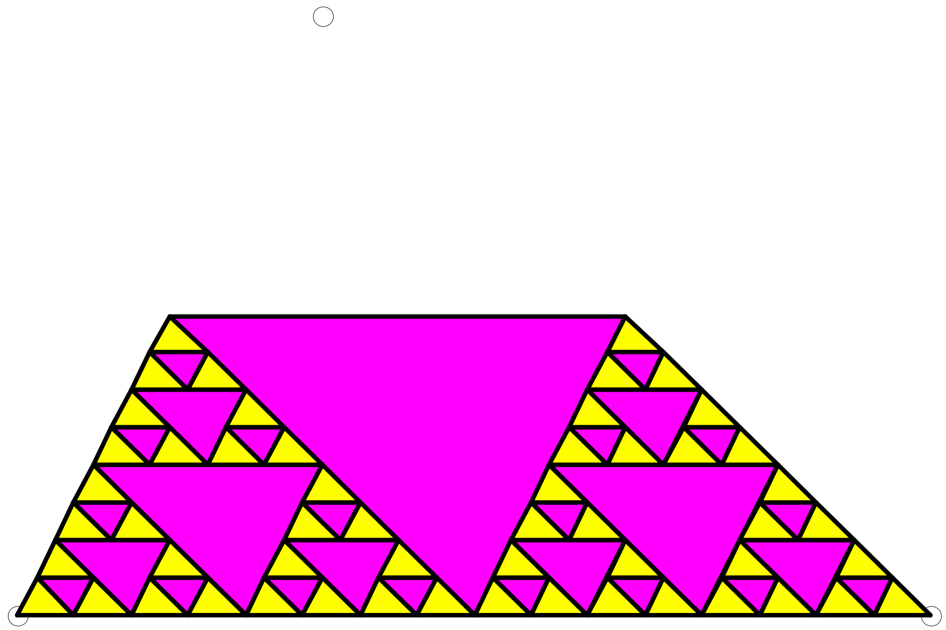


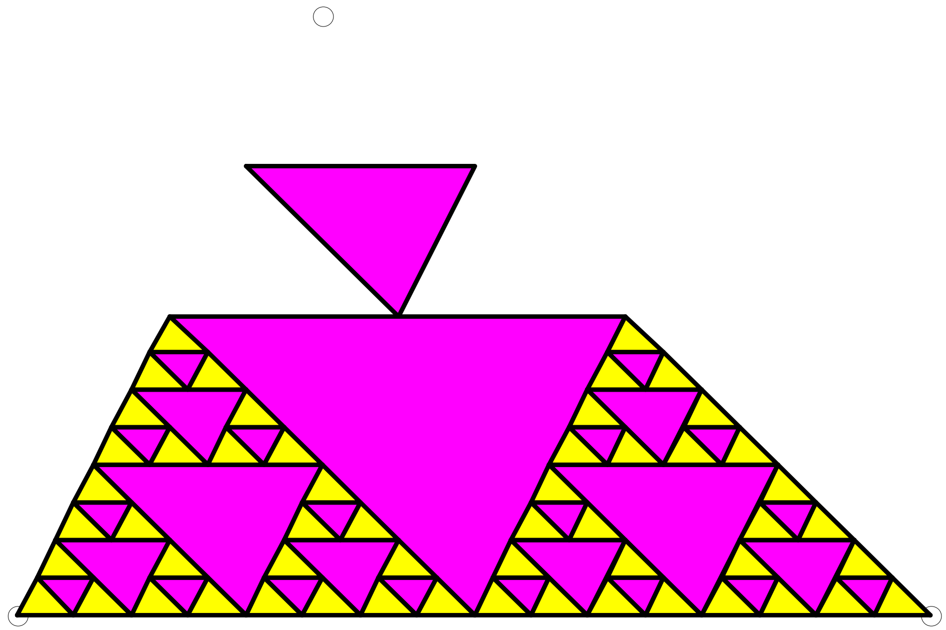




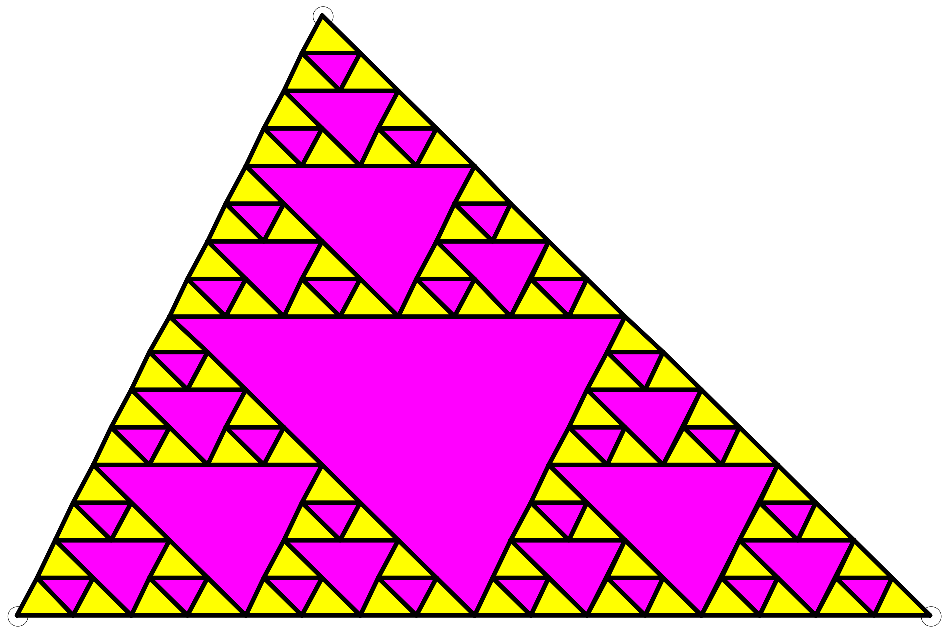








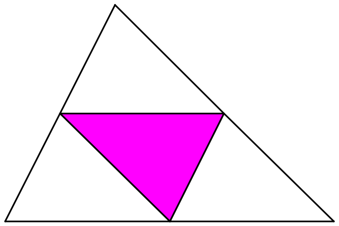
Eventually climb all the way out to get the final result



The basic operation at each level

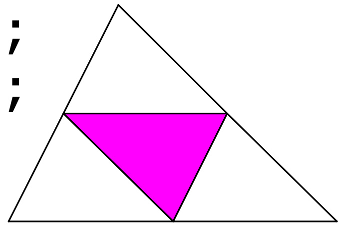
```
if    the triangle is small
        Don't subdivide and just color it yellow.
else
        Subdivide:
        Connect the side midpoints;
        color the interior triangle magenta;
        Apply same process to each outer triangle.
end
```

```
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
```