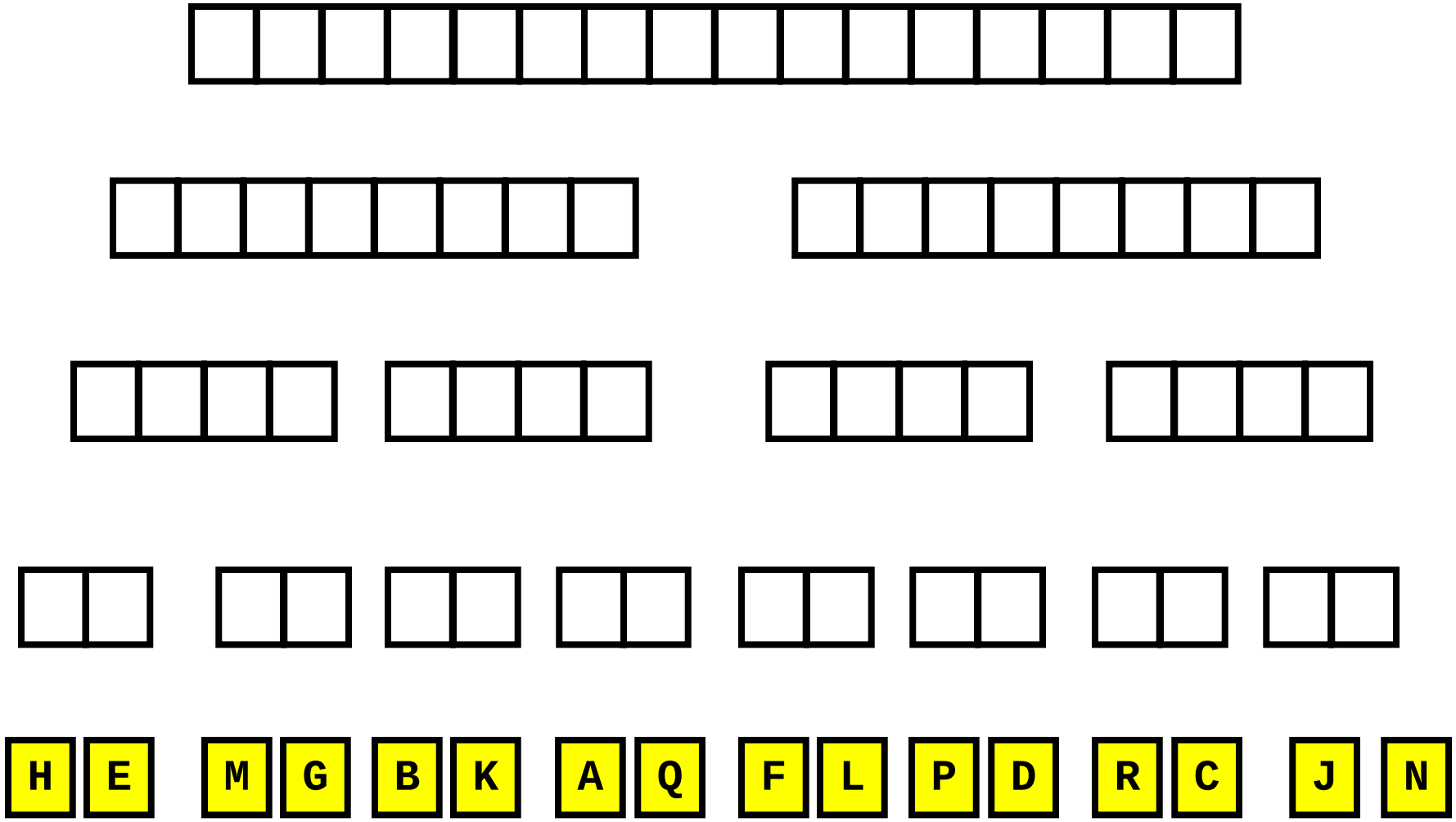
- Previous Lecture:
  - "Divide and conquer" strategies
    - Binary search
    - Merge sort

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

Read Insight 14.1

- Announcements
  - Project 6 due May 5th at 11pm
  - CS1112 final will be 5/12 (Thurs) 9am in Barton indoor field East. Email Prof Fan your *entire exam schedule* if you have a conflict. *We must have this information by next Thursday (5/5).*

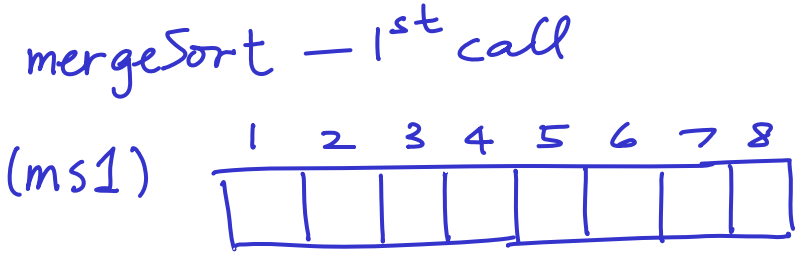
Merge sort is a "divide-and-conquer" strategy



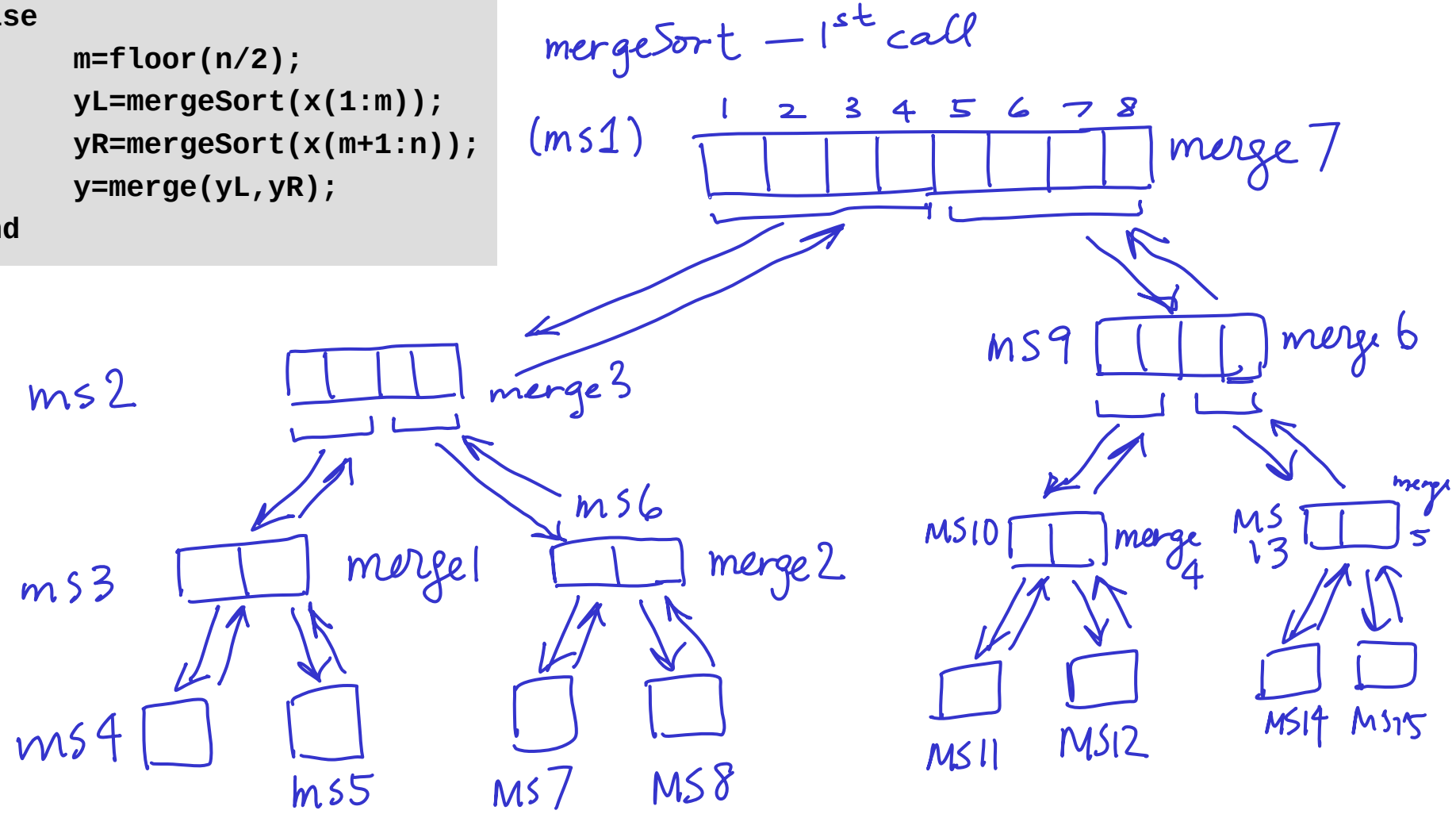
```
function y = mergeSort(x)
% x is a vector.  y is a vector
% consisting of the values in x
% sorted from smallest to largest.

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
```



```
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:

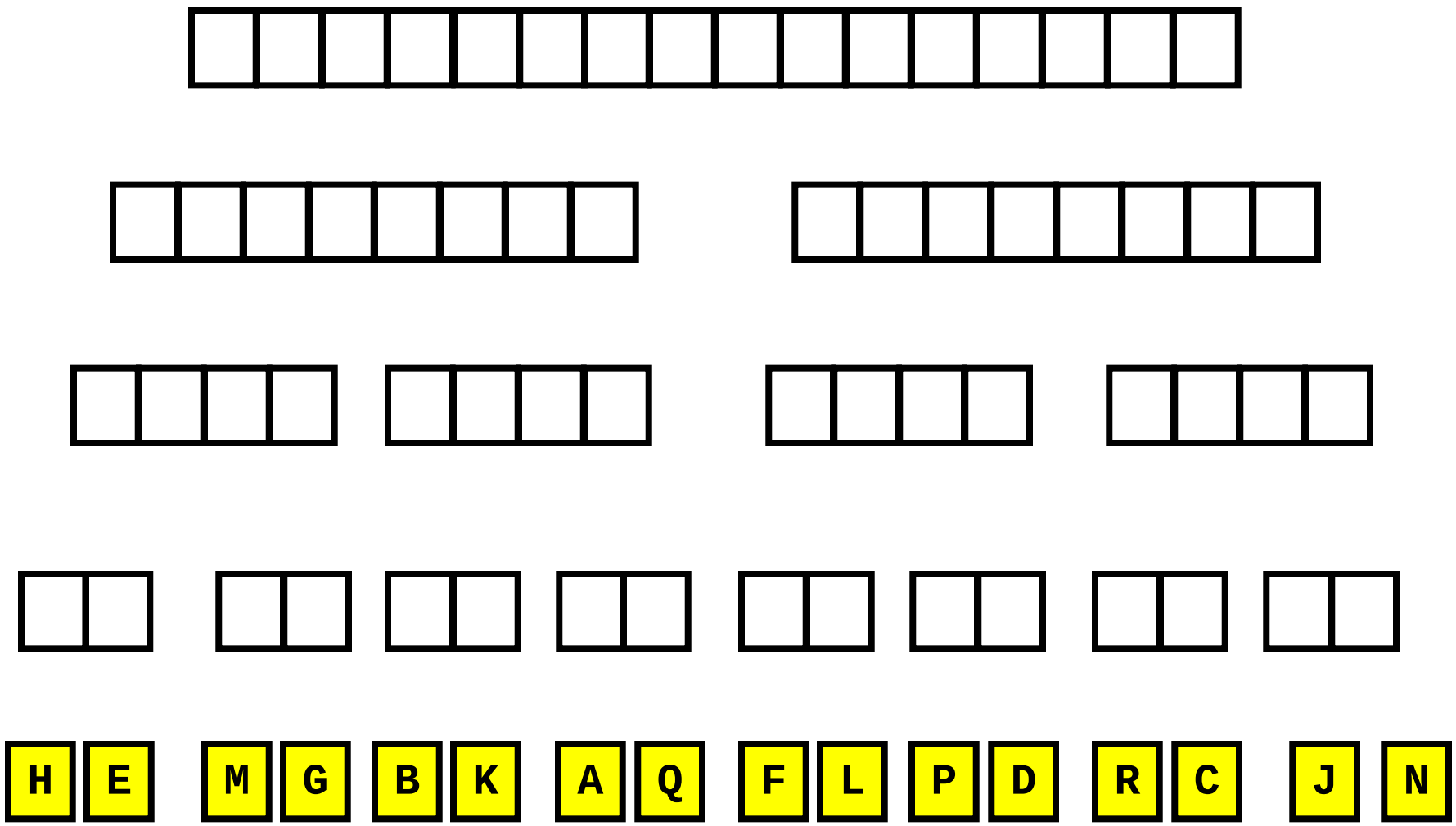
  $1+2+\ldots+(N-1) = N(N-1)/2$, say N^2 for big N

- 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**

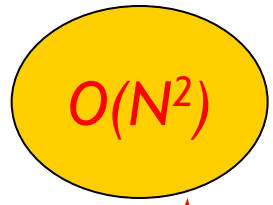
Merge sort: $\log_2(N)$ "levels"; N comparisons each level



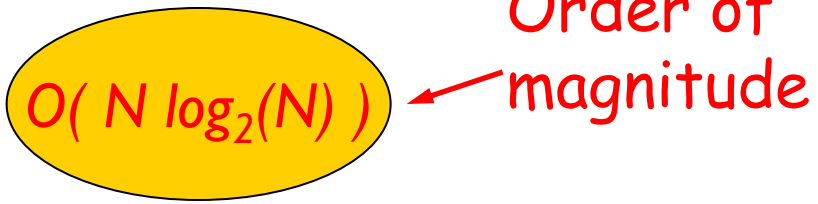
How do merge sort and insertion sort compare?

- 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



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



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

How to choose??

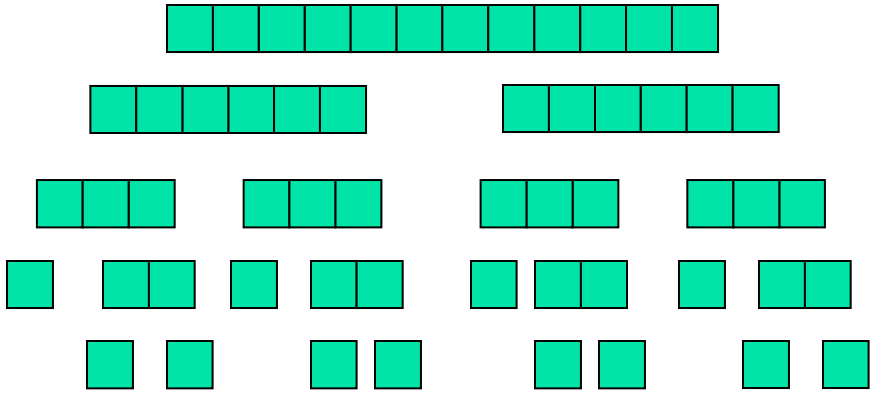
- Depends on application
- Merge sort is especially good for sorting large data set (but watch out for memory usage)
- Insertion sort is "order N^2" at worst case, but what about an average case? If the application requires that you *maintain* a sorted array, insertion sort may be a good choice

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

Back to Recursion

- Merge sort

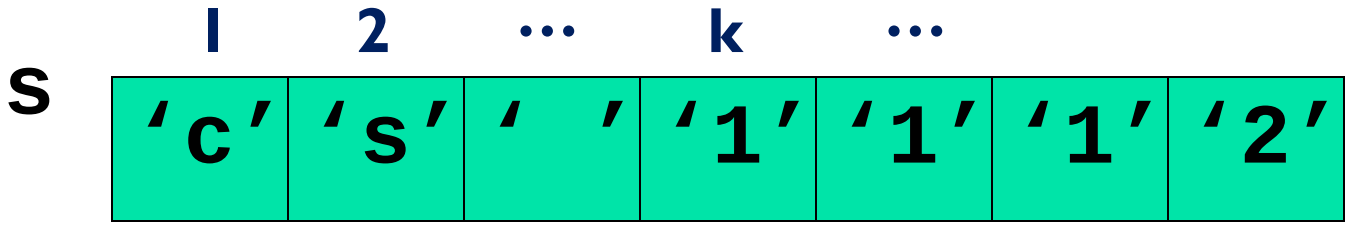


- Remove all occurrences of a character from a string

```
'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



Subproblem 1:
Keep or discard s(1)

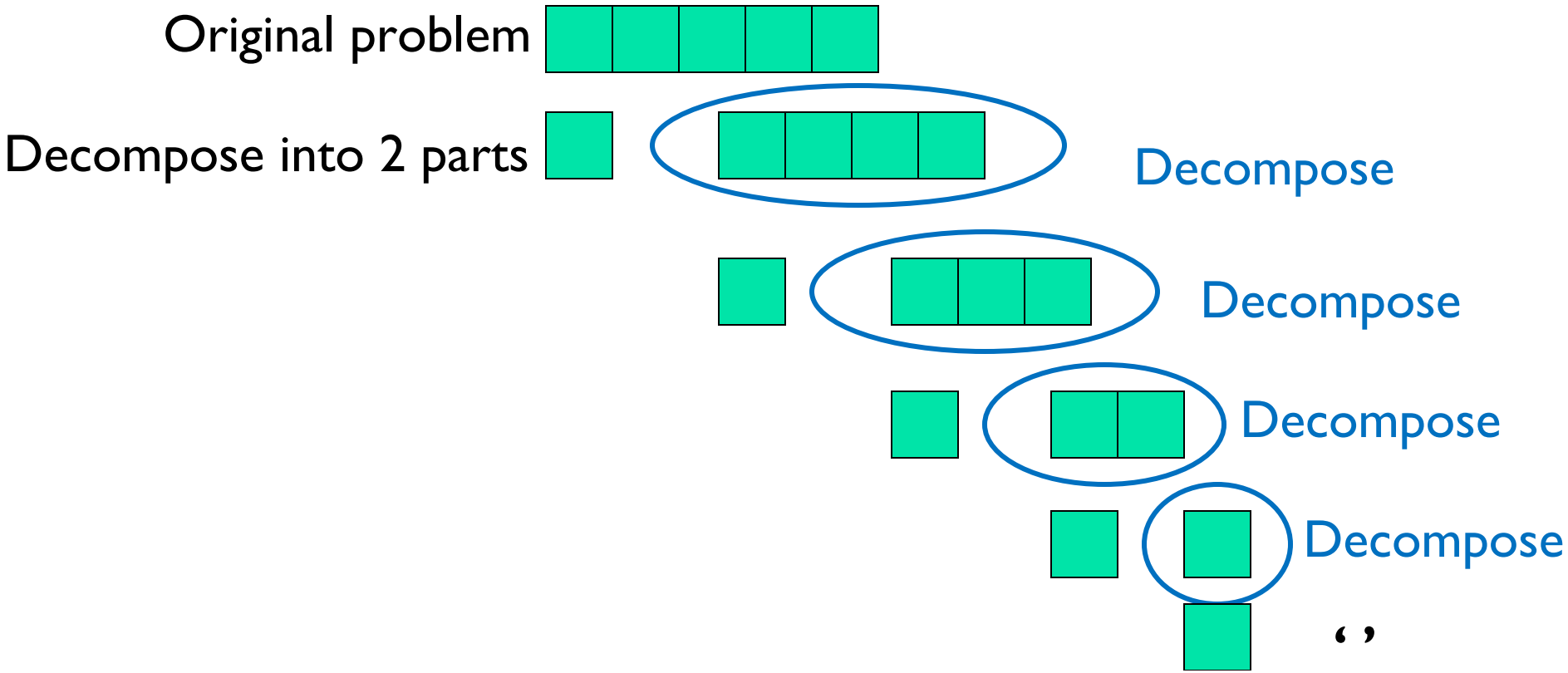
Subproblem 2:
Keep or discard s(2)

Subproblem k:
Keep or discard s(k)

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

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



  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)                                    ];
  else
    % return string is just
    % the remaining s with char c removed

  end
end
```

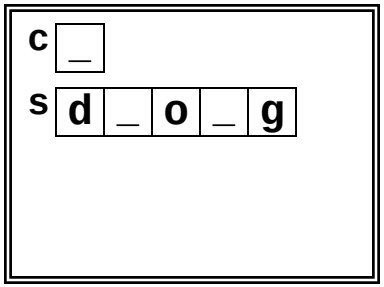
```
function s = removeChar(c, s)
% Return string s with character c removed

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

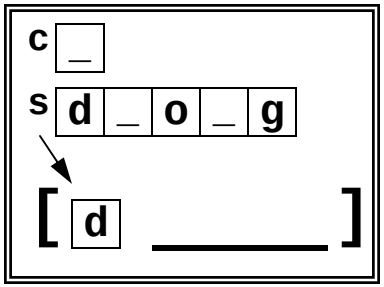


```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```

s | d | _ | o | _ | g |

c | _ |

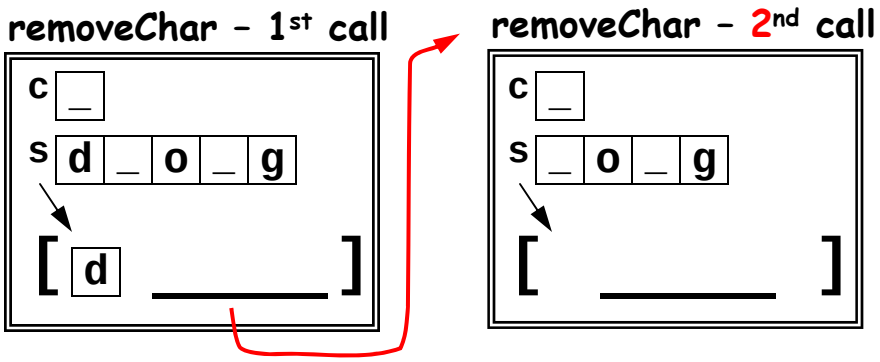
removeChar - 1st call



```
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 |

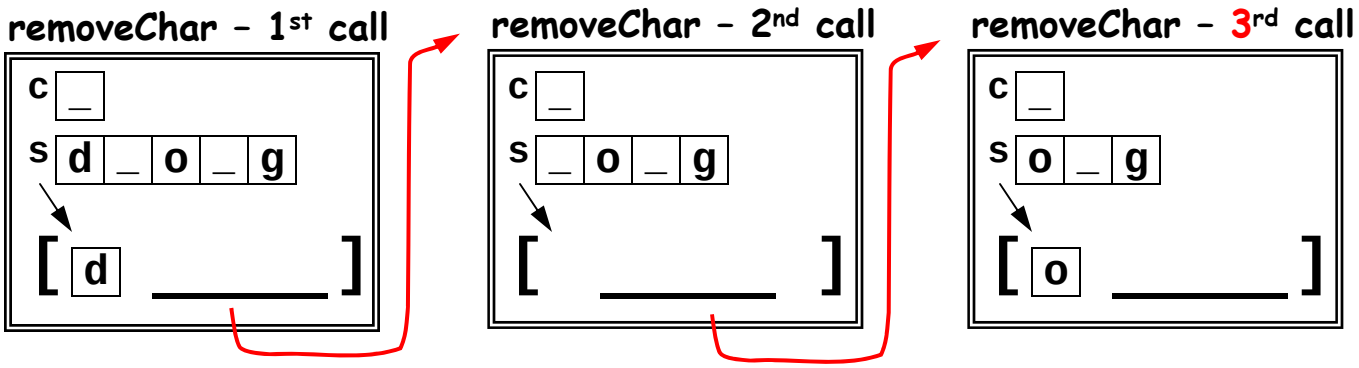
c | _ |



```
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 |

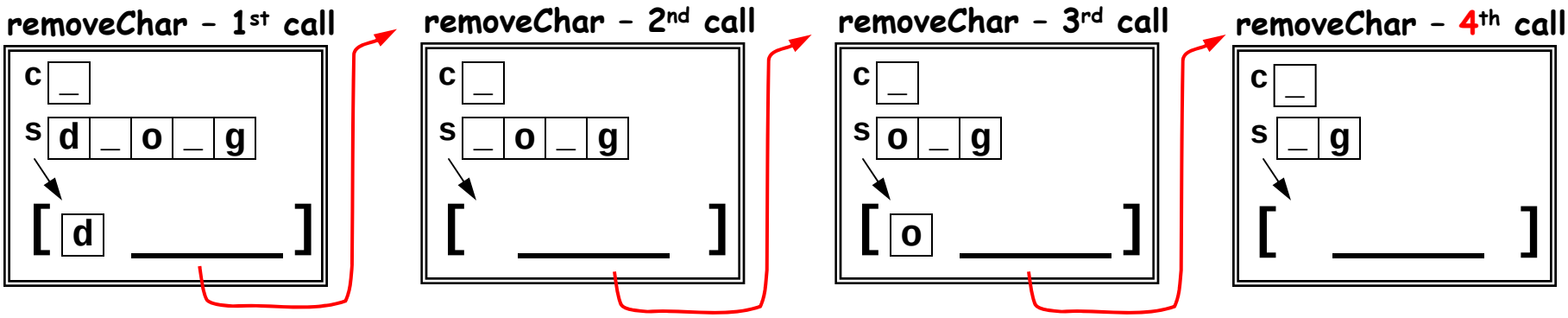
c | _ |



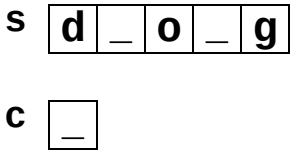
```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```

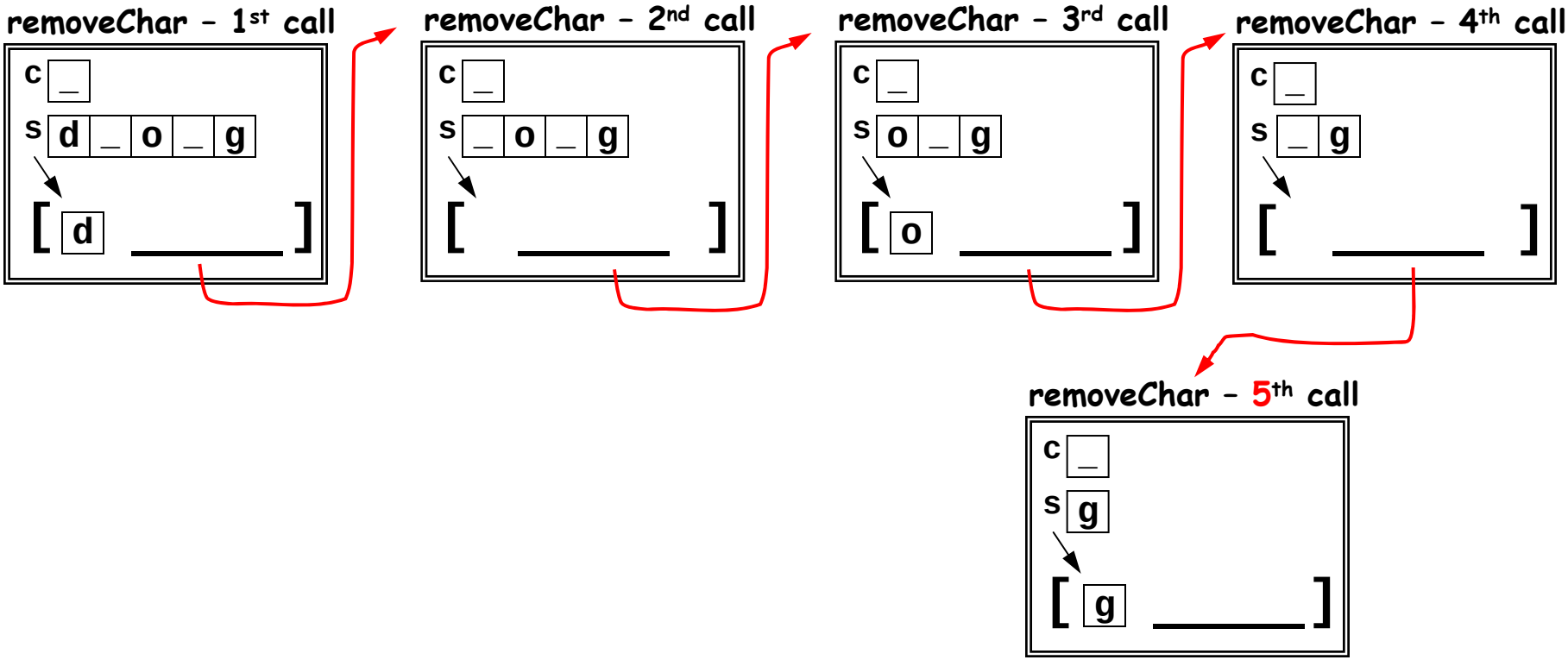
s | d | _ | o | _ | g

c | _ |

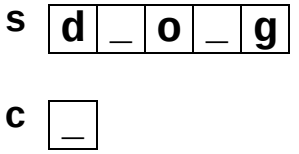


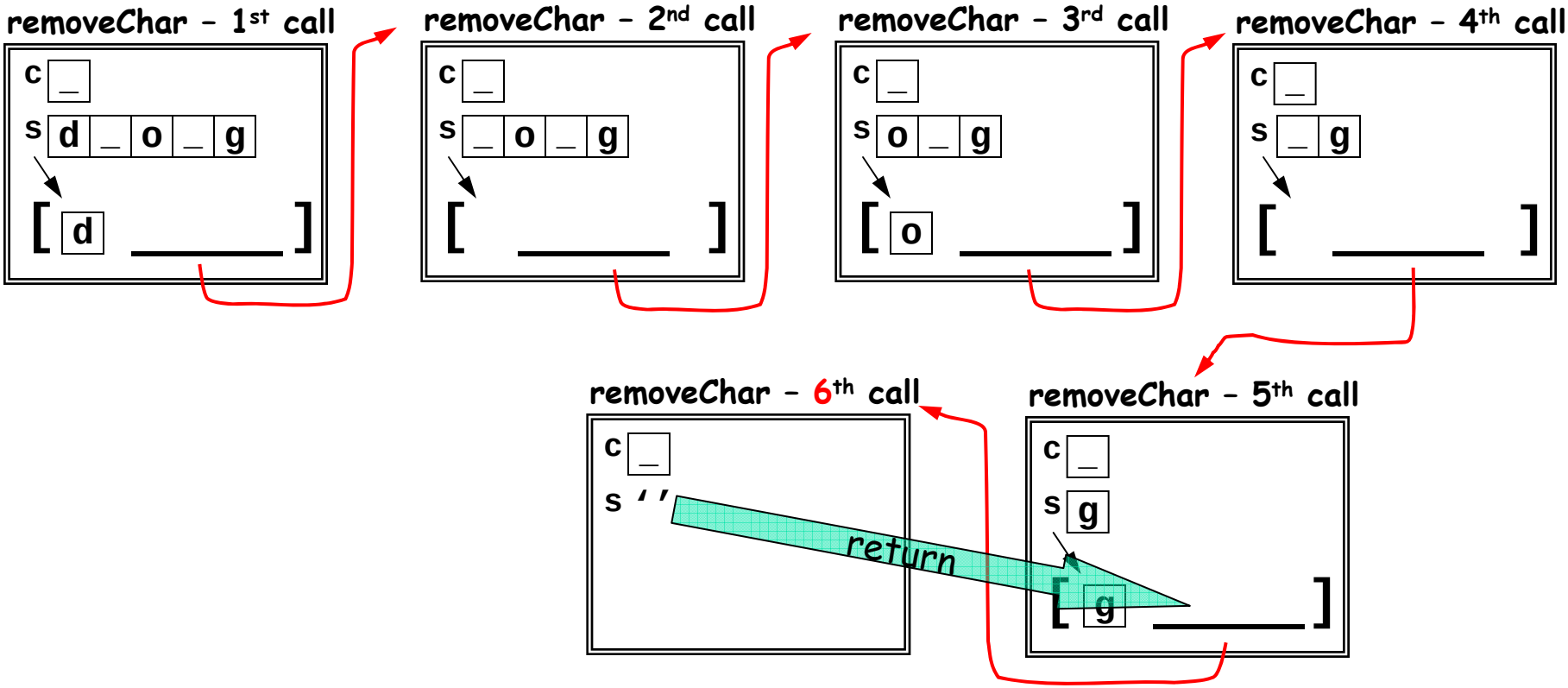
```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```



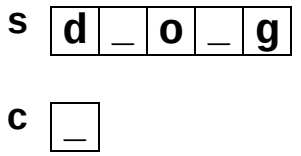


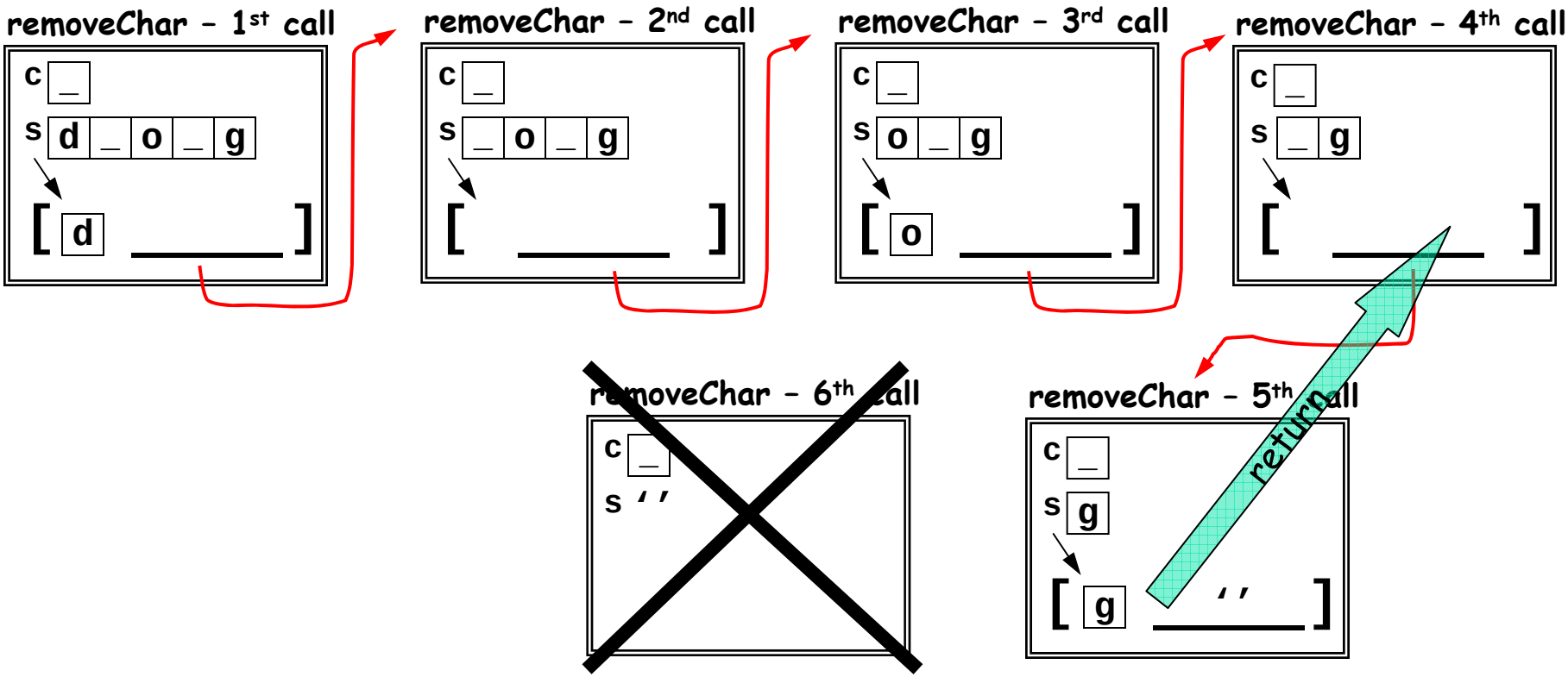
```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```



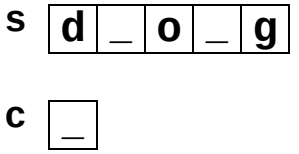


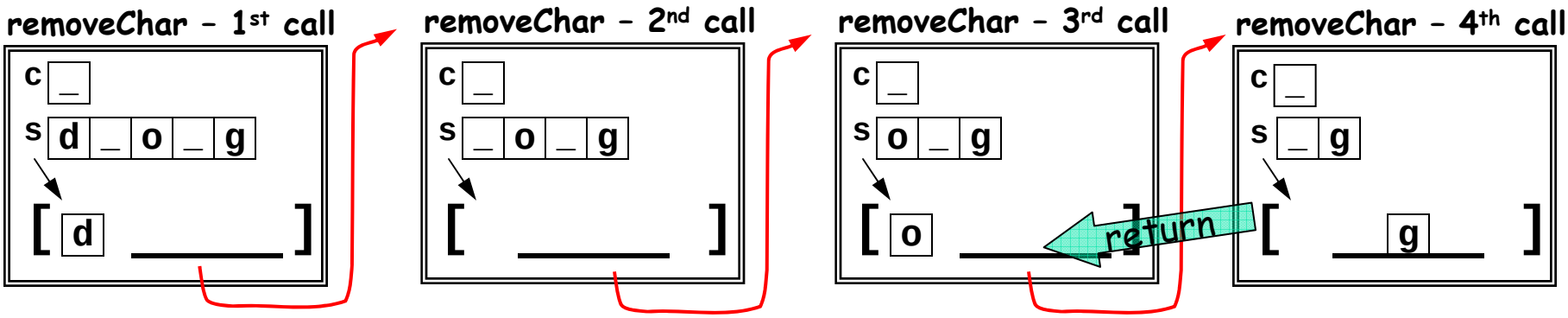
```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```

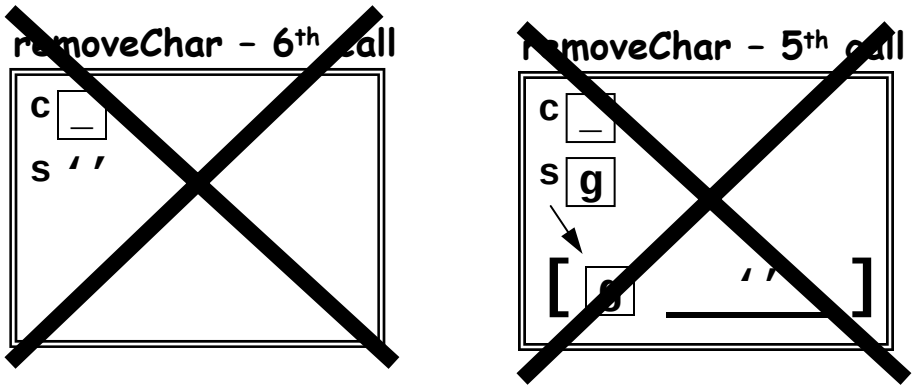




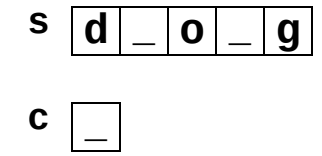
```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```

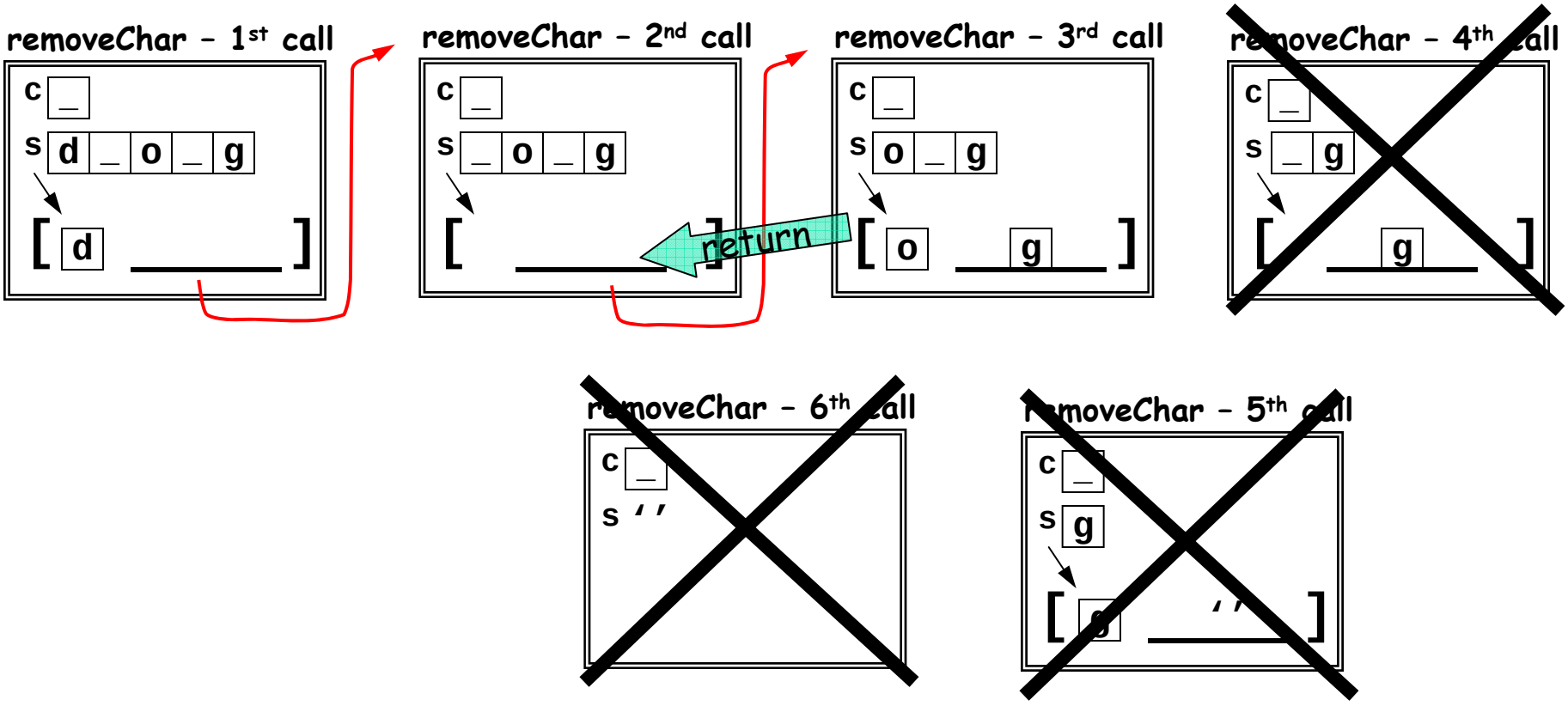




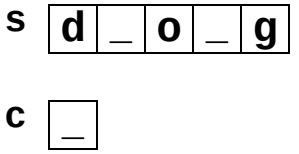


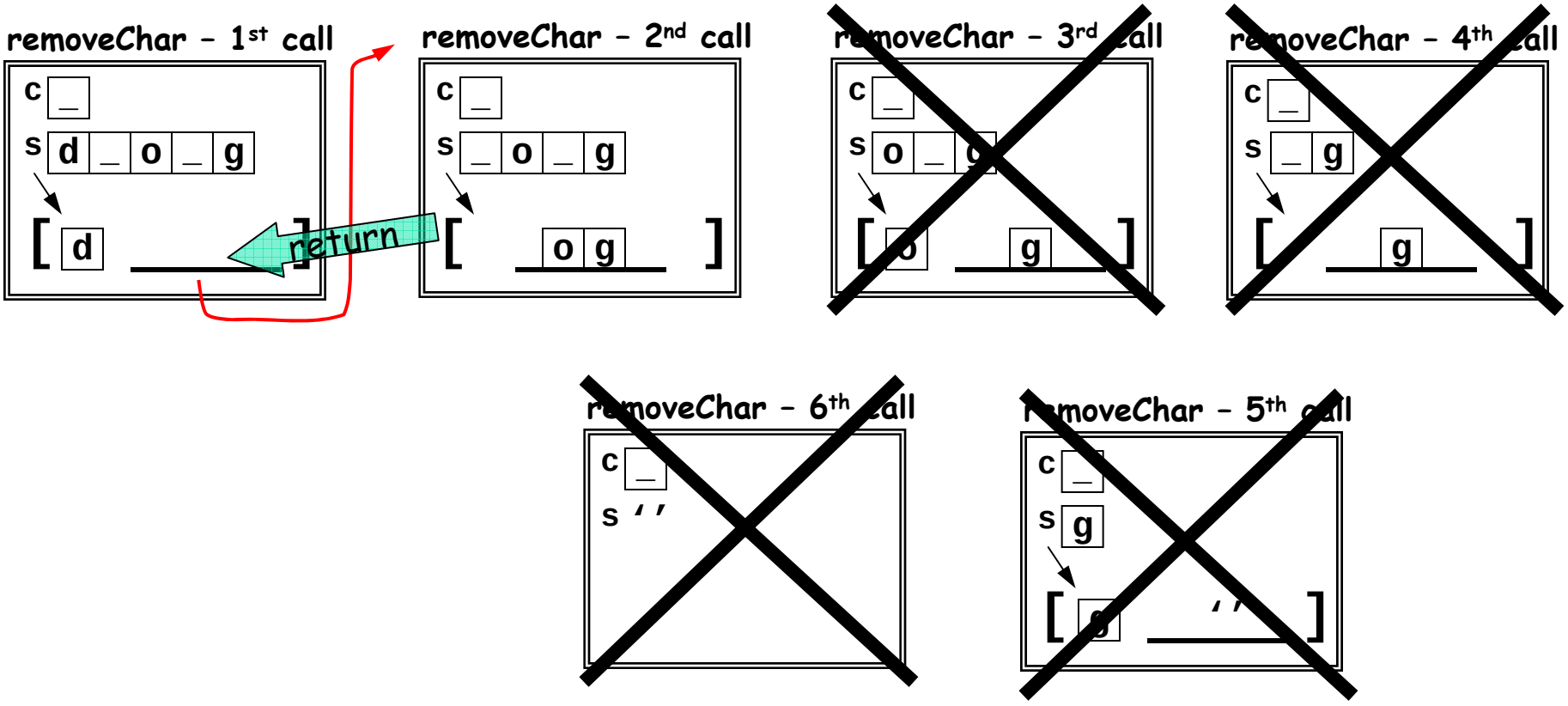
```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```



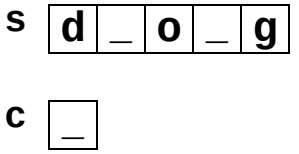


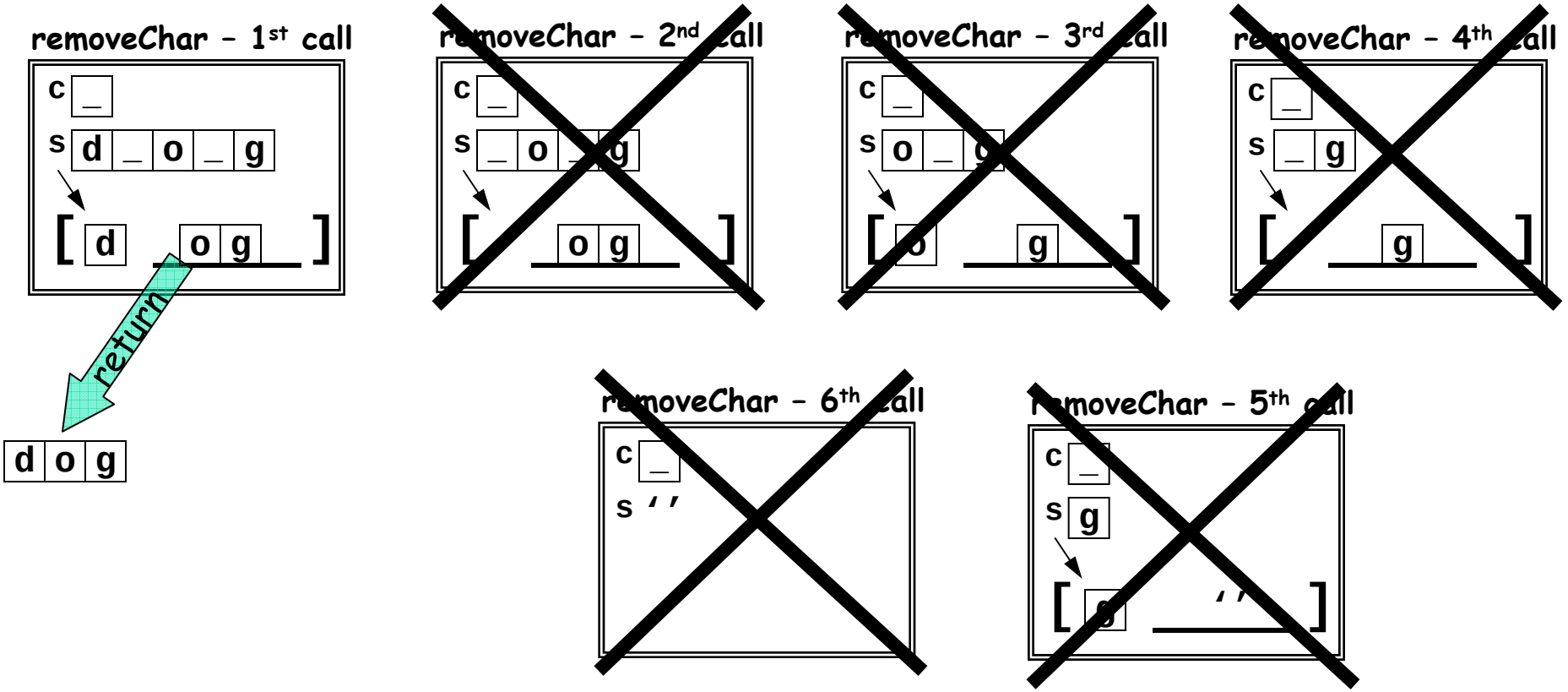
```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```





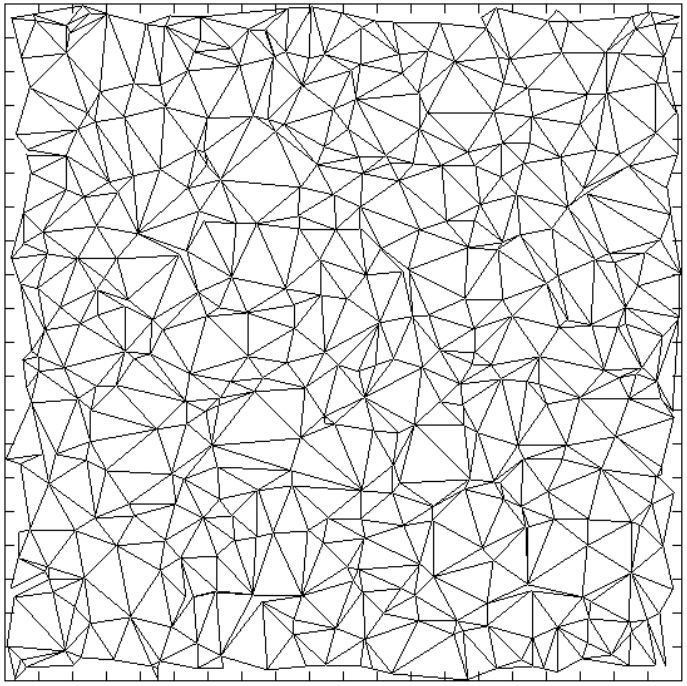
```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```





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