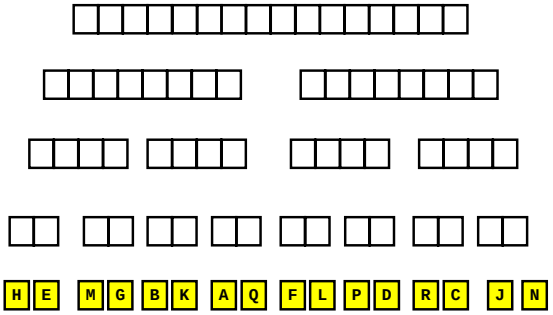
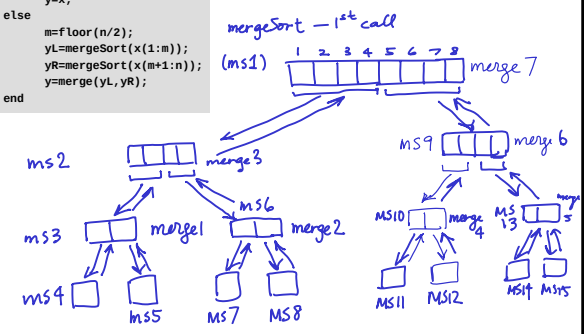
- Previous Lecture:
  - "Divide and conquer" strategies
    - Binary search
    - Merge sort
- Today's Lecture:
  - "Divide and conquer" strategies (cont'd)—recursion
    - Merge sort
    - Removing a character (e.g., the blank) from a string
    - Tiling (subdividing) a triangle, e.g., Sierpinski Triangle
  - Some efficiency considerations
- Announcements
  - Project 6 due Dec 2nd at 11pm
  - CS1112 final will be 12/10 (Fri) 9am in Barton indoor field West. Email Randy Hess (rbhess@cs.cornell.edu) your *entire exam schedule* if you have a conflict. *We must have this information by Thursday (12/2).*

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




```
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:

  _____ for big N

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

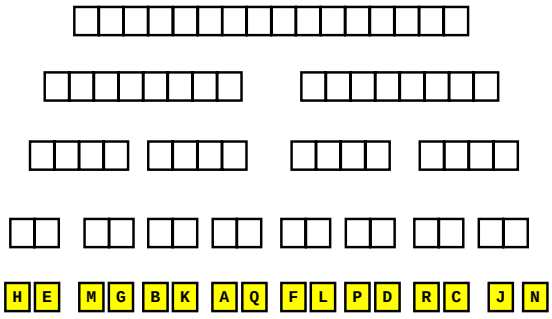
```
function y = mergeSort(x)
% x is a vector.  y is a vector
% consisting of the values in x
% sorted from smallest to largest.

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(ix);  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
```


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



How to choose??

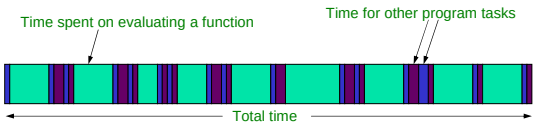
- Depends on application
- Merge sort is especially good for sorting large data set (but watch out for memory usage)
- Insertion sort is "order N^2" at worst case, but what about an average case? If the application requires that you *maintain* a sorted array, insertion sort may be a good choice

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

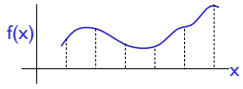
- Consider the execution of a program that is dominated by multiple calls to an expensive-to-evaluate function (e.g., climate simulation models)



- 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.

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
  if s(1)~=c



  else



  end
end
```

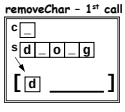


```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```

s [d][ ][o][ ][g]

c [ ]

removeChar - 1st call

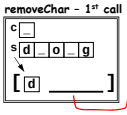



```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```

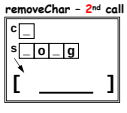
s [d][ ][o][ ][g]

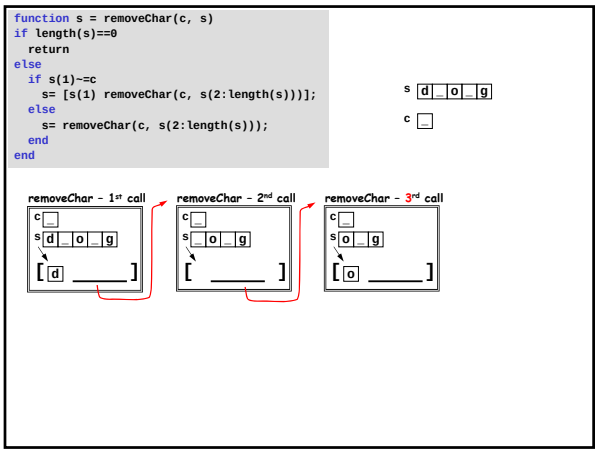
c [ ]

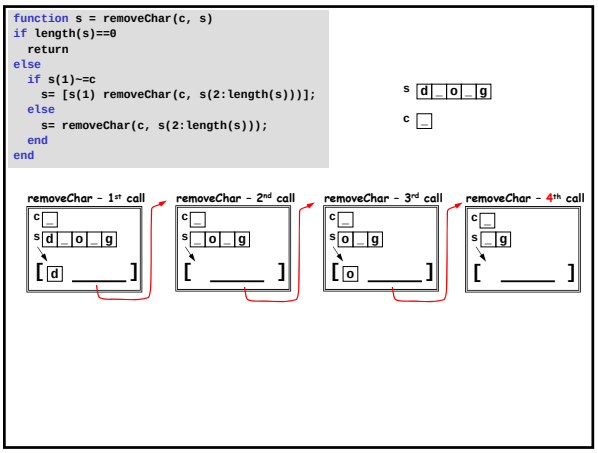
removeChar - 1st call

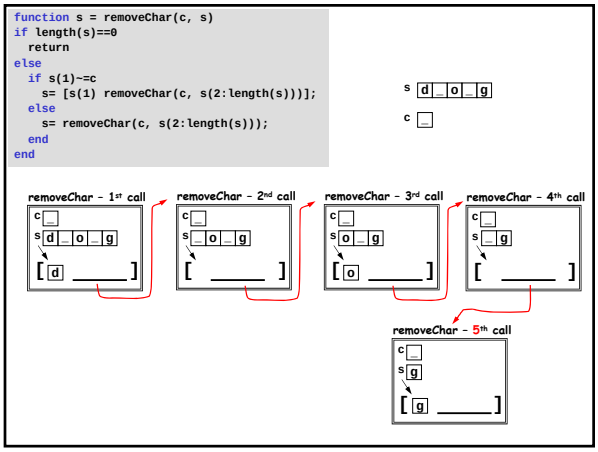


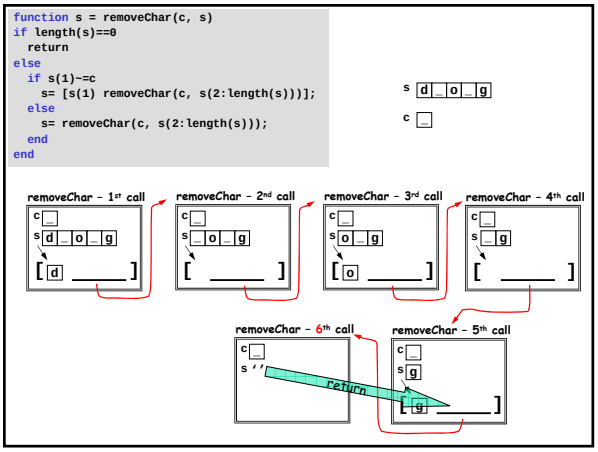
removeChar - 2nd call

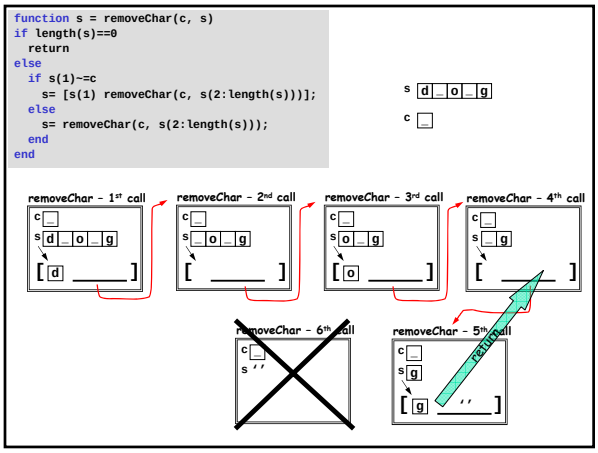


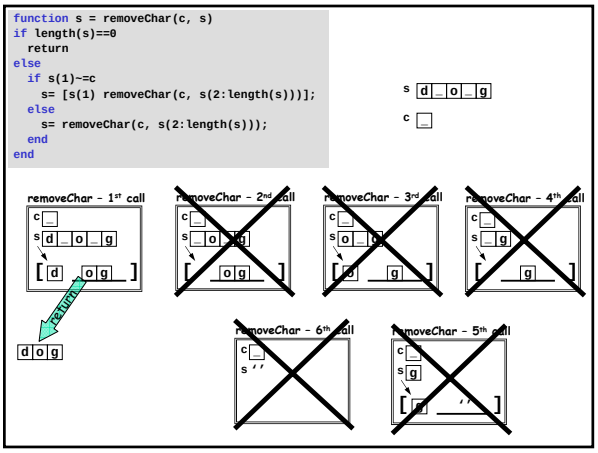

function s = removeChar(c, s)
if length(s)==0
return
else
if s(1)~=c
s= [s(1) removeChar(c, s(2:length(s)))];
else
s= removeChar(c, s(2:length(s)));
end
end
s d _ o _ g
c _
removeChar - 1st call
removeChar - 2nd call
removeChar - 3rd call


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
removeChar - 2nd call
removeChar - 3rd call
removeChar - 4th 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
removeChar - 2nd call
removeChar - 3rd call
removeChar - 4th call
removeChar - 5th 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
removeChar - 2nd call
removeChar - 3rd call
removeChar - 4th call
removeChar - 6th call
removeChar - 5th call
return


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
removeChar - 2nd call
removeChar - 3rd call
removeChar - 4th call
removeChar - 6th call
removeChar - 5th call
return


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
removeChar - 2nd call
removeChar - 3rd call
removeChar - 4th call
return
d o g
removeChar - 6th call
removeChar - 5th call