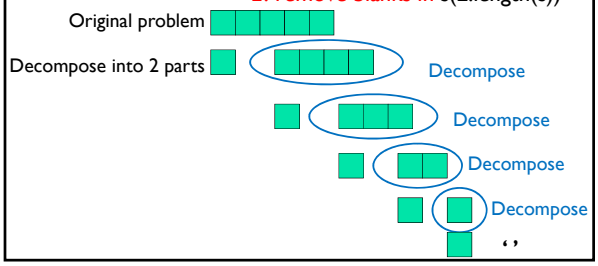
- Previous Lecture:
  - OOP: Overriding methods
  - Recursion example: Remove all occurrences of a character in a string
- Today's Lecture:
  - Recursion example: A mesh of triangles
  - Sort algorithm: Insertion Sort
  - Efficiency Analysis
  - See *Insight* §8.2 for the Bubble Sort algorithm
- Announcements:
  - Discussion this week in Upson B7 computer lab
  - Reminder: Project 6 due Thurs 11pm

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 ***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))



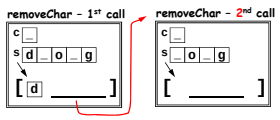

```
function s = removeChar(c, s)
% Return string s with character c removed

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
    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

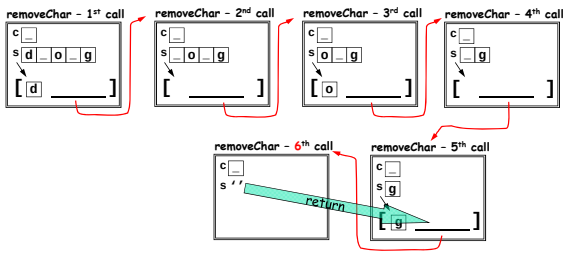
c _

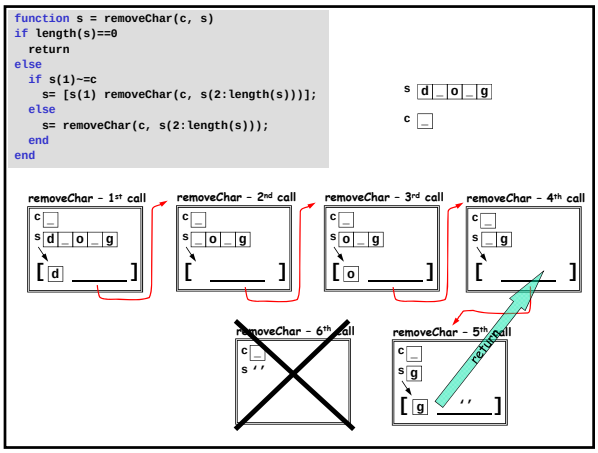


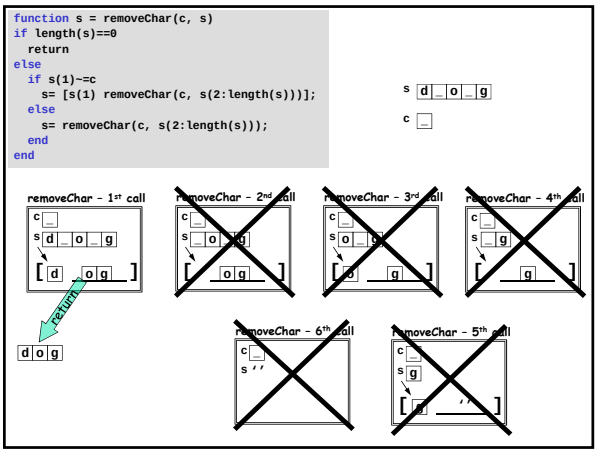

```
function s = removeChar(c, s)
if length(s)==0
  return
else
  if s(1)~=c
    s= [s(1) removeChar(c, s(2:length(s)))];
  else
    s= removeChar(c, s(2:length(s)));
  end
end
```

s d _ o _ g

c _

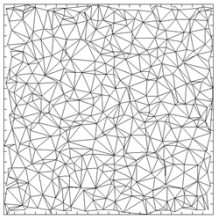






Divide-and-conquer methods, such as recursion, is useful in geometric situations

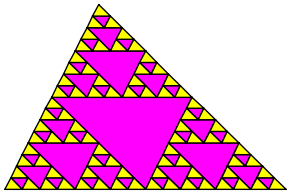
Chop a region up into triangles with smaller triangles in "areas of interest"



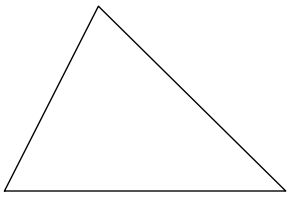
Recursive mesh generation

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

Let's draw this graphic

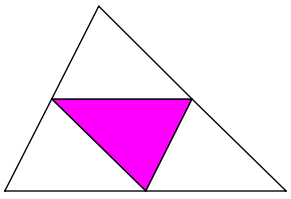


Start with a triangle



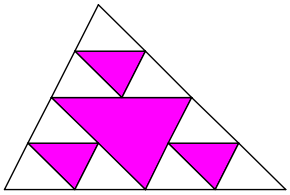
A "level-1" partition of the triangle

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

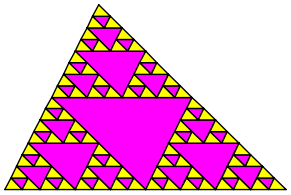


Now do the same partitioning (connecting midpts) on each corner (white) triangle to obtain the "level-2" partitioning

The "level-2" partition of the triangle



The "level-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:

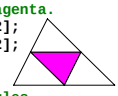
Connect the side midpoints;

color the interior triangle magenta;

apply same process to each outer triangle.

end

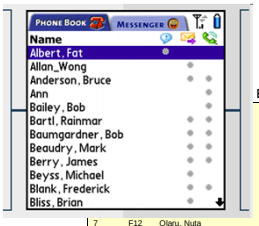

```
function MeshTriangle(x,y,L)
% x,y are 3-vectors that define the vertices of a triangle.
% Draw level-L partitioning.  Assume hold is on.

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`**

Sorting data allows us to search more easily



Name	Score	Grade
Jorge	92.1	
Ahn	91.5	
Oluban	90.6	
Chi	88.9	
Minale	88.1	

Boston Marathon Top Women Finishers

			Official Time	State	Country	Ctz
1	F12	Olaru, Nuta	2:25:25		ETH	
2	F6	Guta, Robe Tola	2:25:27		RUS	
3	F1	Grigoryeva, Lidiya	2:26:34		KEN	
4	F35	Hood, Stephanie A.	2:28:12		LAT	
5	F14	Robson, Denise C.	2:29:48		ETH	
6	F11	Chemjor, Magdaline	2:30:52		ITA	
7	F101	Sultanova-Zhdanova, Firaya	2:33:56		ROM	
8	F15	Mayger, Eliza M.	2:34:37		ETH	
9	F24	Anklam, Ashley A.	2:35:37		RUS	
			2:44:44	IL	USA	CAN
			2:45:54	NS	CAN	
			2:46:25		KEN	
			2:47:17	FL	USA	RUS
			2:47:36		AUS	
			2:48:43	MN	USA	

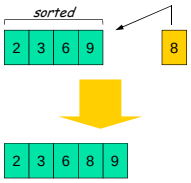
There are many algorithms for sorting

- Insertion Sort (to be discussed today)
- Bubble Sort (read *Insight* §8.2)
- Merge Sort (to be discussed Thursday)
- Quick Sort (a variant used by Matlab's built-in **sort** function)

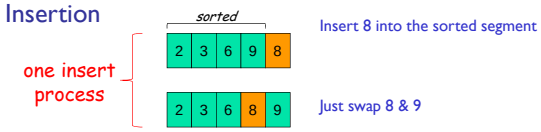
- Each has advantages and disadvantages. Some algorithms are faster (time-efficient) while others are memory-efficient
- *Great opportunity for learning how to analyze programs and algorithms!*

The Insertion Process

- Given a sorted array x, insert a number y such that the result is sorted



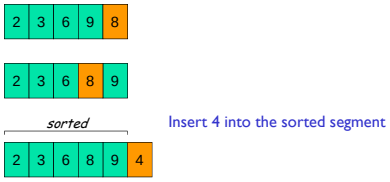
Insertion



Insert 8 into the sorted segment

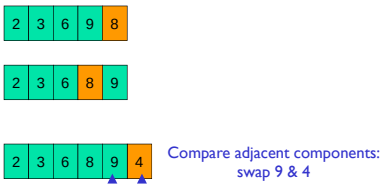
Just swap 8 & 9

Insertion



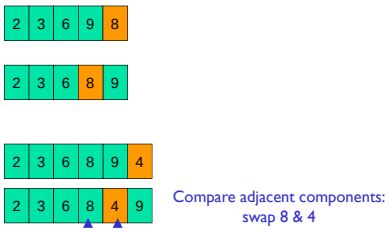
Insert 4 into the sorted segment

Insertion



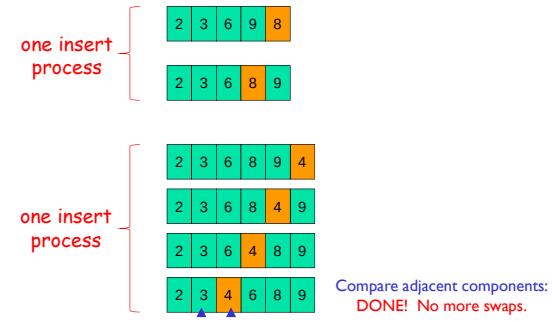
Compare adjacent components: swap 9 & 4

Insertion



Compare adjacent components: swap 8 & 4

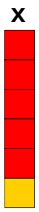
Insertion



See `Insert.m` for the insert process

Sort vector x using the Insertion Sort algorithm

Need to start with a *sorted* subvector. How do you find one?



Length 1 subvector is "sorted"

Insert `x(2)`: `[x(1:2),C,S] = Insert(x(1:2))`

Insert `x(3)`: `[x(1:3),C,S] = Insert(x(1:3))`

Insert `x(4)`: `[x(1:4),C,S] = Insert(x(1:4))`

Insert `x(5)`: `[x(1:5),C,S] = Insert(x(1:5))`

Insert `x(6)`: `[x(1:6),C,S] = Insert(x(1:6))`

`InsertionSort.m`

Insertion Sort vs. Bubble Sort

- Read about Bubble Sort in *Insight* §8.2
- Both algorithms involve the repeated comparison of adjacent values and swaps
- Find out which algorithm is more efficient on average

Other efficiency considerations

- Worst case, best case, average case
- Use of subfunction incurs an "overhead"
- Memory use and access

- Example: Rather than directing the *insert* process to a subfunction, have it done "in-line."
- Also, Insertion sort can be done "in-place," i.e., using "only" the memory space of the original vector.


```
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;

      while

          % swap x(j+1) and x(j)




          j= j-1;

      end
end
```


Sort an array of objects

- Given x, a 1-d array of Intervals references, sort x according to the widths of the Intervals from narrowest to widest
- Use the insertion sort algorithm
- How much of our code needs to be changed?

A. No change

B. One statement

C. About half the code

D. Most of the code