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:
- Project 6 due Tue 11pm. Remember academic integrity!
- Regular office/consulting hours end Wednesday. Study break hours will be posted next week
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
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

removeChar – 1st call

removeChar – 2nd call

removeChar – 3rd call

removeChar – 4th call

removeChar – 5th call

removeChar – 6th call

return

d_og

co

d_og

_og

_og

_d_og

dog

dog

do
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
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
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
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
Mesh Generation

Step one in simulating flow around an airfoil is to generate a mesh and (say) estimate velocity at each mesh point.
Why is mesh generation a divide-and-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-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 \textit{the triangle is small}

Don’t subdivide and just color it \textcolor{yellow}{yellow}.

else

Subdivide:

Connect the side midpoints;

\textcolor{magenta}{color the interior triangle magenta};

\textit{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
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
Searching for an item in a collection

Is the collection organized?
What is the organizing scheme?
Sorting data allows us to search more easily.

### Boston Marathon Top Women Finishers

<table>
<thead>
<tr>
<th>Name</th>
<th>Official Time</th>
<th>State</th>
<th>Country</th>
<th>Ctz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tune, Dire</td>
<td>2:25:25</td>
<td>ETH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Biktimirova, Alevtina</td>
<td>2:25:27</td>
<td>RUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jeptoo, Rita</td>
<td>2:26:34</td>
<td>KEN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prokopcuka, Jelena</td>
<td>2:28:12</td>
<td>LAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Magarsa, Askale Tafa</td>
<td>2:29:48</td>
<td>ETH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Genovese, Bruna</td>
<td>2:30:52</td>
<td>ITA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Olaru, Nuta</td>
<td>2:33:56</td>
<td>ROM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Guta, Robe Tola</td>
<td>2:34:37</td>
<td>ETH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grigoryeva, Lidiya</td>
<td>2:35:37</td>
<td>RUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hood, Stephanie A.</td>
<td>2:44:44</td>
<td>IL</td>
<td>USA</td>
<td>CAN</td>
</tr>
<tr>
<td>Robson, Denise C.</td>
<td>2:45:54</td>
<td>NS</td>
<td>CAN</td>
<td></td>
</tr>
<tr>
<td>Chemjor, Magdaline</td>
<td>2:46:25</td>
<td>KEN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sultanova-Zhdanova, Firaya</td>
<td>2:47:17</td>
<td>FL</td>
<td>USA</td>
<td>RUS</td>
</tr>
<tr>
<td>Mayger, Eliza M.</td>
<td>2:47:36</td>
<td>AUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Anklam, Ashley A.</td>
<td>2:48:43</td>
<td>MN</td>
<td>USA</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Score</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jorge</td>
<td>92.1</td>
<td></td>
</tr>
<tr>
<td>Ahn</td>
<td>91.5</td>
<td></td>
</tr>
<tr>
<td>Oluban</td>
<td>90.6</td>
<td></td>
</tr>
<tr>
<td>Chi</td>
<td>88.9</td>
<td></td>
</tr>
<tr>
<td>Minale</td>
<td>88.1</td>
<td></td>
</tr>
<tr>
<td>Bell</td>
<td>87.3</td>
<td></td>
</tr>
</tbody>
</table>
There are many algorithms for sorting

- **Insertion Sort** (to be discussed today)
- **Bubble Sort** (read *Insight* §8.2)
- **Merge Sort** (to be discussed next lecture)
- **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

one insert process

sorted

2 3 6 9 8

Insert 8 into the sorted segment

2 3 6 8 9

Just swap 8 & 9
Insertion

2 3 6 9 8

2 3 6 8 9

sorted

2 3 6 8 9 4

Insert 4 into the sorted segment
Insertion

Compare adjacent components:
swap 9 & 4
Insertion

Compare adjacent components: swap 8 & 4
Insertion

Compare adjacent components: swap 6 & 4
Insertion

one insert process

Compare adjacent components: DONE! No more swaps.

See Insert.m for the insert process
Sort vector \( \mathbf{x} \) using the Insertion Sort algorithm

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

\[ \mathbf{x} \]

Length 1 subvector is “sorted”

\[ \text{Insert } \mathbf{x}(2): \ [\mathbf{x}(1:2), C, S] = \text{Insert}(\mathbf{x}(1:2)) \]

\[ \text{Insert } \mathbf{x}(3): \ [\mathbf{x}(1:3), C, S] = \text{Insert}(\mathbf{x}(1:3)) \]

\[ \text{Insert } \mathbf{x}(4): \ [\mathbf{x}(1:4), C, S] = \text{Insert}(\mathbf{x}(1:4)) \]

\[ \text{Insert } \mathbf{x}(5): \ [\mathbf{x}(1:5), C, S] = \text{Insert}(\mathbf{x}(1:5)) \]

\[ \text{Insert } \mathbf{x}(6): \ [\mathbf{x}(1:6), C, S] = \text{Insert}(\mathbf{x}(1:6)) \]

InsertionSort.m
How much “work” is insertion sort?

- In the worst case, make $k$ comparisons to insert an element in a sorted array of $k$ elements.
Insertion

one insert process

Insert into sorted array of length 4

one insert process

Insert into sorted array of length 5
How much “work” is insertion sort?

- In the worst case, make $k$ comparisons to insert an element in a sorted array of $k$ elements. For an array of length $N$:

$$1 + 2 + \ldots + (N-1) = \frac{N(N-1)}{2},$$

say $N^2$ for big $N$. 

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 = InsertionSortInplace(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
end
function x = InsertionSortInplace(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