Announcements

- **Extra office hours** today (instead of DIS sections); Zoom links on Canvas
- **P6 due tonight** at 11pm
- **Test 2B** feedback and grade estimation on website
- **Final exam:** Mon, 5/18, 9am. “2.5 hr” take-home, 48 hr submission window
- Optional **review session:** Sunday, 5/17, 2pm, Zoom (see Canvas)
- Please fill out course evaluation, *worth one BONUS point*, which can be used against any point lost on the final exam (150 points).
- Regular office/consulting hours end today. **Study period hours** are posted on Canvas and course website.
Previous Lecture (and exercise):
- Algorithms for sorting and searching
  - Insertion Sort
  - (Read about Bubble Sort in *Insight*)
  - Linear Search
  - Binary Search
- Efficiency (complexity) analysis: analyze loops, count number of operations, use timing functions
- Time efficiency vs. memory efficiency

Today, Lecture 26:
- Another “divide and conquer” strategy: **Merge Sort**
- Review recursion
- Semester wrap-up
Binary search is efficient, but we need to sort the vector in the first place so that we can use binary search

- Many different algorithms out there...
- We saw insertion sort (and read about bubble sort)
- Let’s look at **merge sort**
- Another example of the “divide and conquer” approach (like binary search) but using recursion
Which task fundamentally requires less work: sort a length 1000 array, or merge* two length 500 sorted arrays into one?

A. Sort  
B. Merge  
C. The same

*Merge two sorted arrays so that the resultant array is sorted (not concatenate two arrays)
Comparison counting

How many comparisons (between elements) are required to run insertion sort on the following vector?

[ 9, 13, 24, 96, 12, 18, 56 ]

A. 6
B. 7
C. 12
D. 21
The central sub-problem is the merging of two sorted arrays into one single sorted array.

```
<table>
<thead>
<tr>
<th>12</th>
<th>33</th>
<th>35</th>
<th>45</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>42</td>
<td>55</td>
<td>65</td>
</tr>
<tr>
<td>12</td>
<td>15</td>
<td>33</td>
<td>35</td>
</tr>
</tbody>
</table>
```
Merge

\[ x: \begin{array}{cccc} 12 & 33 & 35 & 45 \end{array} \quad \text{ix:} \quad \begin{array}{c} 1 \end{array} \]

\[ y: \begin{array}{cccc} 15 & 42 & 55 & 65 & 75 \end{array} \quad \text{iy:} \quad \begin{array}{c} 1 \end{array} \]

\[ z: \begin{array}{cccccccc} \end{array} \quad \text{iz:} \quad \begin{array}{c} 1 \end{array} \]

\[ \text{ix} \leq 4 \text{ and iy} \leq 5: \quad x(\text{ix}) \leq y(\text{iy}) \quad ??? \]
ix <= 4 and iy <= 5: x(ix) <= y(iy)  YES
ix\leq 4 \text{ and } iy\leq 5: \ x(ix) \leq y(iy) \ ???
Merge

ix <= 4 and iy <= 5: x(ix) <= y(iy)  NO

x: 12 33 35 45

y: 15 42 55 65 75

z: 12 15

ix: 2

iy: 1

iz: 2
Merge

\[ \text{ix} \leq 4 \text{ and } \text{iy} \leq 5 : \ x(\text{ix}) \leq y(\text{iy}) \]
Merge

\[
\begin{align*}
\text{x:} & \quad 12 & 33 & 35 & 45 \\
\text{y:} & \quad 15 & 42 & 55 & 65 & 75 \\
\text{z:} & \quad 12 & 15 & 33 & \ldots & \ldots & \ldots & \ldots \\
\text{ix:} & \quad 2 \\
\text{iy:} & \quad 2 \\
\text{iz:} & \quad 3
\end{align*}
\]

\[\text{ix} \leq 4 \text{ and iy} \leq 5: \quad \text{x(ix)} \leq \text{y(iy)} \quad \text{YES}\]
ix \leq 4 \text{ and } iy \leq 5: \ x(ix) \leq y(iy) \quad ???
Merge

\[ x: 12 \ 33 \ 35 \ 45 \]
\[ y: 15 \ 42 \ 55 \ 65 \ 75 \]
\[ z: 12 \ 15 \ 33 \ 35 \ \ldots \]

\[ ix: 3 \]
\[ iy: 2 \]
\[ iz: 4 \]

\[ \text{ix} \leq 4 \quad \text{and} \quad iy \leq 5: \quad x(i\text{x}) \leq y(i\text{y}) \quad \text{YES} \]
Merge

ix <= 4 and iy <= 5: x(ix) <= y(iy) ???
ix <= 4 and iy <= 5: x(ix) <= y(iy)  NO
ix <= 4 and iy <= 5: \( x(ix) \leq y(iy) \)
Merge

\[ \text{ix} \leq 4 \text{ and } \text{iy} \leq 5: \quad x(\text{ix}) \leq y(\text{iy}) \quad \text{YES} \]
Merge

\[ x: \begin{array}{cccc} 12 & 33 & 35 & 45 \end{array} \]

\[ y: \begin{array}{ccccc} 15 & 42 & 55 & 65 & 75 \end{array} \]

\[ z: \begin{array}{cccccc} 12 & 15 & 33 & 35 & 42 & 45 \end{array} \]

\[ i_x: 5 \quad i_y: 3 \quad i_z: 7 \]

\[ i_x > 4 \]
Merge

\[ x: 12 33 35 45 \]

\[ y: 15 42 55 65 75 \]

\[ z: 12 15 33 35 42 45 55 \]

\[ ix: 5 \]

\[ iy: 3 \]

\[ iz: 7 \]

\[ ix > 4: \text{ take } y(iy) \]
Merge

\[ x: \begin{array}{cccc} 12 & 33 & 35 & 45 \end{array} \]

\[ y: \begin{array}{cccccc} 15 & 42 & 55 & 65 & 75 \end{array} \]

\[ z: \begin{array}{cccccccc} 12 & 15 & 33 & 35 & 42 & 45 & 55 & \end{array} \]

\[ iy: 4 \]

\[ iy \leq 5 \]

\[ iz: 8 \]
Merge

\[ \text{i}_{y} \leq 5 \]
Merge

\[ x: \begin{array}{cccc} 12 & 33 & 35 & 45 \end{array} \]

\[ y: \begin{array}{ccccc} 15 & 42 & 55 & 65 & 75 \end{array} \]

\[ z: \begin{array}{cccccccc} 12 & 15 & 33 & 35 & 42 & 45 & 55 & 65 \end{array} \]

\[ \begin{array}{c} iy \leq 5 \end{array} \]

\[ \begin{array}{c} ix: 5 \end{array} \]

\[ \begin{array}{c} iy: 5 \end{array} \]

\[ \begin{array}{c} iz: 9 \end{array} \]
Merge

x: 12 33 35 45

y: 15 42 55 65 75

z: 12 15 33 35 42 45 55 65 75

iy <= 5

ix: 5
i y: 5
iz: 9
function z = merge(x,y)
nx = length(x); ny = length(y);
z = zeros(1, nx+ny);
ix = 1; iy = 1; iz = 1;
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

end

% Deal with remaining values in x or y
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
% Deal with remaining values in x or y
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
If I have two helpers, I’d...

- Give each helper half the array to sort
- Then I get back the sorted subarrays and merge them.
Cost of dividing work

Suppose each comparison we make costs $1

Given a vector with N elements,
- Insertion sort costs $N(N-1)/2
- Merge costs $(N-1)$

(worst case)

Consider a vector with 8 elements
- Sorting by ourselves: $26
- Sorting by delegating work:
  - Left delegate (4 elements): $6
  - Right delegate (4 elements): $6
  - Merge (8 elements): $7
  - **Profit: $7!**
Merge sort: Motivation

If I have two helpers, I’d...
- Give each helper half the array to sort
- Then I get back the sorted subarrays and merge them.

What if those two helpers each had two sub-helpers?
And the sub-helpers each had two sub-sub-helpers? And...
Subdivide the sorting task
Subdivide again
And again
And one last time
Now merge
And merge again
And again
And one last time
Done!
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 (task is trivial)
    % Base case
else
    % Divide work
    % Delegate subproblems

    % Merge results
end
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
    % Divide work
    % Delegate subproblems

    % Merge results
end
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 and insertion sort compare?

- **Insertion sort**: (worst case) makes $k$ comparisons to insert an element in a sorted array of $k$ elements. For an array of length $N$: $1+2+...+(N-1) = N(N-1)/2$, say $N^2$ for big $N$

- **Merge sort**: 

---

**Note:** The formula for insertion sort is simplified to $N^2$ for large $N$. The merge sort complexity, however, is not specified in the text provided.
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
Merge sort: about $\log_2(N)$ “levels”;
about $N$ 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 $N$:
  $$1+2+\ldots+(N-1) = \frac{N(N-1)}{2}, \text{ say } N^2 \text{ for big } N$$

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

- Insertion sort is done *in-place*; merge sort (recursion) requires extra memory (call frames plus merge area)

See compareInsertMerge.m
How to choose??

- Depends on application
- Merge sort is especially good for sorting large data sets
  - Easily adapted to work with files if data is too big for memory
- Sort “stability” matters for object handles (elements may compare equal, but are actually distinct)
  - Insertion, Merge are intrinsically stable. QuickSort is not, but MATLAB’s `sort()` does extra work to stabilize
- Insertion sort is “order \(N^2\)” at worst case, but what about an average case?
  - Insertion good for “fixing” a mostly sorted array, or adding just a few new elements
What we learned…

- Develop/implement **algorithms** for problems
- Develop programming skills
  - Design, implement, document, test, and debug
- Programming “tool bag”
  - Functions for reducing redundancy
  - Control flow (if-else; loops)
  - Recursion
  - Data structures, type
  - Graphics
  - File handling
What we learned… (cont’d)

- Applications and concepts
  - Image processing
  - Object-oriented programming—custom type
  - Sorting and searching (you should know the algorithms covered)
  - Approximation and error
  - Simulation, sensitivity analysis
  - Computational effort and efficiency
Where to go from here?

- **Mathworks.com** – Many free tutorials available on specific topics, e.g., signal processing, Simulink, …, etc.

- More detailed intro to scientific and engineering uses: “*Getting Started with MATLAB*” by Rudra Pratap. Excellent for independent, non-course-based learning

- Just play, i.e., experiment, with MATLAB programs! Many programs available in MATLAB “Community” forum “File Exchange”
Some courses for future consideration

- **ENGRD/CS 2110** Object-oriented programming and data structure
  - CS 2111 Programming practicum

- **CS 2800** Discrete Math (logic, proof, probability theory)

- **CS 3220** Computational Mathematics for Computer Science

- Short language courses (e.g., Python, C++)
Computing gives us *insight* into a problem

- Computing is **not** about getting one answer!
- We build models and write programs so that we can “play” with the models and programs, learning—gaining insights—as we vary the parameters and assumptions
- Good models require domain-specific knowledge (and experience)
- Good **programs** …
  - are correct and have been thoroughly tested
  - are modular and cleanly organized
  - are well-documented
  - use appropriate data structures and algorithms
  - are reasonably efficient in time and memory
Best wishes and good luck with all your exams!