Announcements

- P6 due Wednesday at 11pm
- Final exam:
  - Thursday, 5/19, 9am, Barton Hall Indoor FieldWEST
- Please fill out two surveys online: course evaluation (College of Engrg) and experience in CS undergraduate courses (CS Dept). Each will be worth 0.5 BONUS point. Bonus point can be used against any point lost on the final exam (150 points).
- Regular office/consulting hours end Wednesday night. New hours to be posted online.
- Pick up papers during consulting hours at Carpenter
- Read announcements on course website!
Previous Lecture:
- Recursion – partitioning a triangle
- Insertion Sort
- (Read about Bubble Sort in *Insight*)

Today’s Lecture:
- “Divide and conquer” strategies
  - Binary search
  - Merge sort
Insertion

one insert process

Insert into sorted array of length 4

one insert process

Insert into sorted array of length 5
Sort vector $\mathbf{x}$ using the Insertion Sort algorithm

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

Length 1 subvector is “sorted”

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

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

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

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

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

InsertionSort.m
Sort an array of objects

- Given \( x \), a 1-d array of Interval 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
Sort an array of objects

- Given \( x \), a 1-d array of \textit{Interval} references, sort \( x \) according to the widths of the \textit{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

The only change is in how we do the comparison!

See InsertionSortIntervals.m
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!
Searching for an item in an unorganized collection?

- May need to look through the whole collection to find the target item
- E.g., find value $x$ in vector $v$

- Linear search
% f is index of first occurrence
% of value x in vector v.
% f is -1 if x not found.
k= 1;
while  k<=length(v) && v(k)~=x
    k= k + 1;
end
if   k>length(v)
    f= -1; % signal for x not found
else
    f= k;
end
% Linear Search
% f is index of first occurrence
% of value x in vector v.
% f is -1 if x not found.

k = 1;
while k <= length(v) && v(k) ~= x
    k = k + 1;
end

if k > length(v)
    f = -1; % signal for x not found
else
    f = k;
end
% Linear Search
% f is index of first occurrence
% of value x in vector v.
% f is -1 if x not found.
k = 1;
while  k<=length(v) && v(k)~=x
  k= k + 1;
end
if  k>length(v)
  f= -1; % signal for x not found
else
  f= k;
end

Suppose another vector is twice as long as v. The expected “effort” required to do a linear search is …
% Linear Search
% f is index of first occurrence
% of value x in vector v.
% f is -1 if x not found.
k = 1;
while k <= length(v) && v(k) ~= x
    k = k + 1;
end
if k > length(v)
    f = -1; % signal for x not found
else
    f = k;
end
% Linear Search
% f is index of first occurrence
% of value x in vector v.
% f is -1 if x not found.
k = 1;
while k <= length(v) && v(k) ~ = x
    k = k + 1;
end
if k > length(v)
    f = -1; % signal for x not found
else
    f = k;
end

What if v is sorted?

Searching in a sorted list should require less work
An ordered (sorted) list

The Manhattan phone book has 1,000,000+ entries.

How is it possible to locate a name by examining just a tiny, tiny fraction of those entries?
Key idea of “phone book search”: repeated halving

To find the page containing Pat Reed’s number...

while (Phone book is longer than 1 page)
    Open to the middle page.
    if “Reed” comes before the first entry,
        Rip and throw away the 2nd half.
    else
        Rip and throw away the 1st half.
end
end
What happens to the phone book length?

Original: 3000 pages
After 1 rip: 1500 pages
After 2 rips: 750 pages
After 3 rips: 375 pages
After 4 rips: 188 pages
After 5 rips: 94 pages
: 
After 12 rips: 1 page
Binary Search

Repeatedly halving the size of the “search space” is the main idea behind the method of binary search.

An item in a sorted array of length $n$ can be located with just $\log_2 n$ comparisons.
% Linear Search
% f is index of first occurrence of value x in vector v.
% f is -1 if x not found.

k = 1;
while k <= length(v) && v(k) ~= x
    k = k + 1;
end
if k > length(v)
    f = -1; % signal for x not found
else
    f = k;
end

n comparisons against the target are needed in worst case, n = length(v).
Binary Search

Repeatedly halving the size of the “search space” is the main idea behind the method of binary search.

An item in a sorted array of length \( n \) can be located with just \( \log_2 n \) comparisons.

“Savings” is significant!

<table>
<thead>
<tr>
<th>( n )</th>
<th>( \log_2(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>10</td>
</tr>
<tr>
<td>10000</td>
<td>13</td>
</tr>
</tbody>
</table>
Binary search: target $x = 70$

$L$: 1  2  3  4  5  6  7  8  9  10  11  12

$R$: 12

$Mid$: 6

$v$: 12 15 33 35 42 45 51 62 73 75 86 98

$v(Mid) \leq x$

So throw away the left half...
Binary search: target $x = 70$

L: 6  
Mid: 9  
R: 12

$x < v(Mid)$

So throw away the right half...
Binary search: target $x = 70$

So throw away the left half...
Binary search: target $x = 70$

L: 7
Mid: 8
R: 9

$v(Mid) \leq x$

So throw away the left half...
Binary search: target $x = 70$

Done because $R - L = 1$
function L = binarySearch(x, v)
% Find position after which to insert x. v(1)<...<v(end).
% L is the index such that v(L) <= x < v(L+1);
% L=0 if x<v(1). If x>v(end), L=length(v) but x~=v(L).

% Maintain a search window [L..R] such that v(L)<=x<v(R).
% Since x may not be in v, initially set ... 
L=0;  R=length(v)+1;

% Keep halving [L..R] until R-L is 1,
% always keeping v(L) <= x < v(R)
while R ~= L+1
    m= floor((L+R)/2);  % middle of search window
    if
        else
    end
end
function L = binarySearch(x, v)
\% Find position after which to insert x. v(1)<...<v(end).
\% L is the index such that v(L) \leq x < v(L+1);
\% L=0 if x<v(1). If x>v(end), L=length(v) but x\neq v(L).
\%
\% Maintain a search window [L..R] such that v(L) \leq x < v(R).
\% Since x may not be in v, initially set ...
L=0; R=length(v)+1;

\%
\% Keep halving [L..R] until R-L is 1,
\% always keeping v(L) \leq x < v(R)
while R \neq L+1
    m= floor((L+R)/2); \% middle of search window
    if v(m) \leq x

        L= m;
    else

        R= m;
    end
end

\%
\% This version is different from that in Insight
function L = binarySearch(x, v)
% Find position after which to insert x. v(1)<…<v(end).
% L is the index such that v(L) <= x < v(L+1);
% L=0 if x<v(1). If x>v(end), L=length(v) but x~=v(L).

% Maintain a search window [L..R] such that v(L)<=x<v(R).
% Since x may not be in v, initially set ...
L=0; R=length(v)+1;

% Keep halving [L..R] until R-L is 1,
% always keeping v(L) <= x < v(R)
while R ~= L+1
    m = floor((L+R)/2); % middle of search window
    if v(m) <= x
        L = m;
    else
        R = m;
    end
end

% Play with showBinarySearch.m
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**
- An example of the “divide and conquer” approach using recursion
Which task is “easier,” sort a length 1000 array or merge* two length 500 sorted arrays into one?

A. Sort  
B. Merge

*Merge two sorted arrays so that the resultant array is sorted
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

E G H M   A B K Q   D F L P   C J N R
E H   G M   B K   A Q   F L   D P   C R   J N
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 n==1
    y = x;
else
    m  = floor(n/2);yL = mergeSortL(x(1:m));yR = mergeSortR(x(m+1:n));y  = merge(yL,yR);
end
The central sub-problem is the *merging* of two sorted arrays into one single sorted array.
ix <= 4 and iy <= 5:  x(ix) <= y(iy)  ???
Merge

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

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

\[ z: \begin{array}{cccccccc}
12 & & & & & & & \\
\end{array} \quad \text{iz:} \begin{array}{c}
1 \\
\end{array} \]

\[ \text{ix} \leq 4 \text{ and iy} \leq 5: \quad x(\text{ix}) \leq y(\text{iy}) \quad \text{YES} \]
ix <= 4 and iy <= 5:  x(ix) <= y(iy)  ???
ix <= 4 and iy <= 5: x(ix) <= y(iy)  NO
ix \leq 4 \text{ and } iy \leq 5: \ x(ix) \leq y(iy) \ ??
Merge

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

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

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

\[ \begin{align*}
ix & \leq 4 \quad \text{and} \quad iy & \leq 5: \quad x(ix) \leq y(iy) \quad \text{YES}
\end{align*} \]
ix <= 4 and iy <= 5:  x(ix) <= y(iy)  ??
ix <= 4 and iy <= 5: x(ix) <= y(iy)  NO
Merge

\[
\begin{align*}
\text{x:} & \quad 12 & 33 & 35 & 45 \\
\text{y:} & \quad 15 & 42 & 55 & 65 & 75 \\
\text{z:} & \quad 12 & 15 & 33 & 35 & 42 & \ldots & \ldots \\
\text{ix:} & \quad 4 \\
\text{iy:} & \quad 3 \\
\text{iz:} & \quad 6
\end{align*}
\]

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

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

\[\text{ix > 4}\]
Merge

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

\[
\begin{align*}
\text{x:} & \quad 12 & 33 & 35 & 45 \\
\text{y:} & \quad 15 & 42 & 55 & 65 & 75 \\
\text{z:} & \quad 12 & 15 & 33 & 35 & 42 & 45 & 55 & \quad \text{iz:} & \quad 8 \\
\text{iy} & \leq 5
\end{align*}
\]
Merge

\[ i_y \leq 5 \]
Merge

x: [12, 33, 35, 45]

y: [15, 42, 55, 65, 75]

z: [12, 15, 33, 35, 42, 45, 55, 65]

iy <= 5

ix: [5]

iy: [5]

iz: [9]
Merge

\( x: \begin{array}{cccc} 12 & 33 & 35 & 45 \end{array} \)

\( y: \begin{array}{cccc} 15 & 42 & 55 & 65 & 75 \end{array} \)

\( z: \begin{array}{cccc} 12 & 15 & 33 & 35 & 42 & 45 & 55 & 65 & 75 \end{array} \)

\( i_x: 5 \)

\( i_y: 5 \)

\( i_z: 9 \)

\( i_y \leq 5 \)
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
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 = mergeSortL(x(1:m));
    yR = mergeSortR(x(m+1:n));
    y  = merge(yL,yR);
  end
function y = mergeSortL(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 = mergeSortL_L(x(1:m));
yR = mergeSortL_R(x(m+1:n));
y  = merge(yL,yR);
end
function y = mergeSortL_L(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 = mergeSortL_L_L(x(1:m));
    yR = mergeSortL_L_R(x(m+1:n));
    y  = merge(yL,yR);
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
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 $N$:
  \[1 + 2 + \ldots + (N-1) = \frac{N(N-1)}{2}, \text{ say } N^2 \text{ for big } N\]

- 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.
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 \( x(\text{ix}) \leq y(\text{iy}) \)
        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
    z(iz)= x(ix);  ix=ix+1;  iz=iz+1;
end
while iy<=ny
    z(iz)= y(iy);  iy=iy+1;  iz=iz+1;
end
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 \( N \):
  \[
  1+2+\ldots+(N-1) = \frac{N(N-1)}{2}, \text{ say } N^2 \text{ for big } N
  \]

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

- Insertion sort is done \textit{in-place}; merge sort (recursion) requires much more memory

See \texttt{compareInsertMerge.m}
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
  - Graphics
  - File handling
What we learned... (cont’d)

- Applications and concepts
  - Image processing
  - Object-oriented programming
  - Sorting and searching—you should know the algorithms covered
  - Divide-and-conquer strategies
  - Approximation and error
  - Simulation
  - Computational effort and efficiency
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 modular and cleanly organized
  - are well-documented
  - use appropriate data structures and algorithms
  - are reasonably efficient in time and memory
Final Exam

- Thursday 5/19, 9-11:30am, Barton Hall indoor tracks WEST
- Covers entire course; some emphasis on material after Prelim 2
- Closed-book exam, no calculators
- Bring student ID card

- Check for announcements on webpage:
  - Study break office/consulting hours
  - Review session time and location
  - Review questions
  - List of potentially useful functions
Final Exam

- Thursday 5/19, 9-11:30am, Barton Hall indoor tracks WEST
- Covers entire course; some emphasis on material after Prelim 2
- Closed-book exam, no calculators
- Bring student ID card

Check for announcements on webpage:
- Study break office/consulting hours
- Review session time and location
- Review questions
- List of potentially useful functions

Best wishes and good luck with all your exams!