Previous Lecture:
- Insertion Sort
- (Bubble Sort in *Insight*)
- Dealing with expensive function evaluation

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

- P6 due today at 11pm
- Final exam:
  - Thursday, 5/9, 9am, Barton West (indoor field)
- Please fill out course evaluation on-line, see “Exercise 15”
- Regular office/consulting hours end today. Revised hours start Sunday
- Pick up papers during consulting hours at Carpenter
- Read announcements on course website!
% 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?

v = 12 15 33 35 42 45
x = 31

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></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 $x = 70$

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

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

L:  1
Mid:  6
R:  12

$v(Mid) \leq x$

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

$x < v(\text{Mid})$

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

$L$: 6

Mid: 7

$R$: 9

$v(Mid) \leq x$

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

$v(Mid) \leq x$

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

$$
\begin{array}{cccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline
12 & 15 & 33 & 35 & 42 & 45 & 51 & 62 & 73 & 75 & 86 & 98 \\
\end{array}
$$

L: 8
Mid: 8
R: 9

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) <= 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

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

HEMGGBKQAQFLPDRCJN

HEMGGBKQAQ FLPDRCJN
Subdivide again
And again
And one last time
Now merge
And merge again
And again
And one last time

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>M</th>
<th>N</th>
<th>P</th>
<th>Q</th>
<th>R</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>E</th>
<th>G</th>
<th>H</th>
<th>K</th>
<th>M</th>
<th>Q</th>
</tr>
</thead>
</table>

| C | D | F | J | L | N | P | R |
Done!

A B C D E F G H J K L M N P Q R
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
The central sub-problem is the merging of two sorted arrays into one single sorted array.
Merge

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

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

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

\( \text{x: } [12, 33, 35, 45] \)
\( \text{y: } [15, 42, 55, 65, 75] \)
\( \text{z: } [12, 15] \)

\( \text{ix: } 2 \)
\( \text{iy: } 1 \)
\( \text{iz: } 2 \)

\( \text{ix} \leq 4 \text{ and } \text{iy} \leq 5: \ x(\text{ix}) \leq y(\text{iy}) \text{ NO} \)
Merge

\[ i_x \leq 4 \text{ and } i_y \leq 5: \quad x(i_x) \leq y(i_y) \quad ??? \]
Merge

ix <= 4 and iy <= 5:  x(ix) <= y(iy)  YES
Merge

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

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

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

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

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

\[
ix: 4 \\
iy: 2 \\
iz: 5 \\
\]

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

\[ x: \begin{array}{cccc} 12 & 33 & 35 & 45 \end{array} \quad \text{ix:} \begin{array}{c} 5 \end{array} \]

\[ y: \begin{array}{ccccc} 15 & 42 & 55 & 65 & 75 \end{array} \quad \text{iy:} \begin{array}{c} 3 \end{array} \]

\[ z: \begin{array}{cccccc} 12 & 15 & 33 & 35 & 42 & 45 \end{array} \quad \text{iz:} \begin{array}{c} 6 \end{array} \]

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

x: 12 33 35 45

y: 15 42 55 65 75

z: 12 15 33 35 42 45 55

ix: 5

iy: 4

iz: 8

iy <= 5
Merge

\[ \text{ix: 5} \]

\[ \text{iy: 4} \]

\[ \text{iz: 8} \]

\[ \text{iy} \leq 5 \]
Merge

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

x: 12 33 35 45

y: 15 42 55 65 75

z: 12 15 33 35 42 45 55 65 75

ix: 5

iy: 5

iz: 9

iy <= 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)
x = length(x); ny = length(y);
z = zeros(1, nx+ny);
ix = 1; iy = 1; iz = 1;
while ix<=nx && iy<=ny
    % Deal with remaining values in x or y
end
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 = 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, 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
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 $N$:
  $$1+2+\ldots+(N-1) = \frac{N(N-1)}{2},$$
  say $N^2$ 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
function z = merge(x,y)
x = 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 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 $N$:
  \[1 + 2 + \ldots + (N-1) = \frac{N(N-1)}{2}, \text{ say } N^2 \text{ for big } N\]
  \[O(N^2)\]

- **Merge sort**: $N \cdot \log_2(N)$
  \[O(N \log_2(N))\]

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

See `compareInsertMerge.m`
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
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 5/9, 9-11:30am, Barton 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