Previous Lecture:
- Linear search, binary search
- Insertion sort
- (Reading: Bubble Sort)

Today’s Lecture:
- Merge Sort
- What’s next?

Announcements
- **P6** due Thursday at 11pm
- **Final exam:** Dec 7th 2-4:30pm, Rockefeller Hall
  - Last names beginning with A-N: Room 201
  - Last names beginning with O-Z: Room 203
Announcements

- **P6 due Thursday at 11pm**
- **Final exam:**
  - Dec 7, 2-4:30pm, Rockefeller Hall 201 (A-N), 203 (O-Z)
- Please fill out course evaluation on-line, see “Exercise 16”
- Revised office/consulting hours during study break
- Pick up papers during consulting hours at Carpenter
- Read announcements on course website!
Linear search and binary search

- Linear search
  - “Effort” is \( \text{linearly proportional to } n \), the size of the search space (e.g., the length of the vector)
  - Can represent effort by the number of comparisons against the search target done during the search

- Binary search
  - Effort is proportional to \( \log_2(n) \) where \( n \) is the size of the search space
  - Saving of \( \log_2(n) \) over \( n \) is significant when \( n \) is large! But binary search requires sorted vector
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
Motivation: merging is an easier job than sorting!

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

<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>
<tbody>
<tr>
<td>E</td>
<td>G</td>
<td>H</td>
<td>M</td>
<td>A</td>
<td>B</td>
<td>K</td>
<td>Q</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
<th>F</th>
<th>J</th>
<th>L</th>
<th>N</th>
<th>P</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>F</td>
<td>L</td>
<td>P</td>
<td>C</td>
<td>J</td>
<td>N</td>
<td>R</td>
</tr>
</tbody>
</table>
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.

<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>
</tbody>
</table>

| 12 | 15 | 33 | 35 | 42 | 45 | 55 | 65 | 75 |
Merge

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

ix <= 4 and iy <= 5: \( x(ix) \leq y(iy) \) ???

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

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

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

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

\[
\begin{align*}
\text{x: } & \quad 12 \quad 33 \quad 35 \quad 45 \\
\text{y: } & \quad 15 \quad 42 \quad 55 \quad 65 \quad 75 \\
\text{z: } & \quad 12 \quad 15 \quad 33 \quad 35 \quad \ldots \quad \ldots \quad \ldots \\
\text{ix: } & \quad 4 \\
\text{iy: } & \quad 2 \\
\text{iz: } & \quad 5
\end{align*}
\]

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

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

x: 12 33 35 45

y: 15 42 55 65 75

z: 12 15 33 35 42 45

ix: 4

iy: 3

iz: 6

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

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

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

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

\[ \text{ix} > 4 \]

\[ \text{ix}: 5 \]

\[ \text{iy}: 3 \]

\[ \text{iz}: 7 \]
Merge

\[
\begin{align*}
&x: & 12 & 33 & 35 & 45 \\
&y: & 15 & 42 & 55 & 65 & 75 \\
&z: & 12 & 15 & 33 & 35 & 42 & 45 & 55 \\
&ix: & 5 \\
&iy: & 3 \\
&iz: & 7 \\
\end{align*}
\]

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

x: 12 33 35 45

y: 15 42 55 65 75

z: 12 15 33 35 42 45 55

iy <= 5
Merge

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

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

\(iy \leq 5\)
Merge

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

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

\[ z: \begin{array}{cccccccc} 12 & 15 & 33 & 35 & 42 & 45 & 55 & 65 & 75 \end{array} \]

\[ i_y \leq 5 \]
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
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

See InsertionSort.m
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 j > 0 && x(j+1) < x(j)

% swap x(j+1) and x(j)
temp = x(j);
x(j) = x(j+1);
x(j+1) = temp;

j = j-1;
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

All the comparisons between vector values are done in merge
function \( z = \text{merge}(x,y) \)
\[ nx = \text{length}(x); \ ny = \text{length}(y); \]
\[ z = \text{zeros}(1, nx+ny); \]
\[ ix = 1; \ iy = 1; \ iz = 1; \]
\[ \text{while } ix \leq nx \text{ } \&\text{ } iy \leq ny \]
\[ \text{if } x(ix) \leq y(iy) \]
\[ z(iz) = x(ix); \ ix=ix+1; \ iz=iz+1; \]
\[ \text{else} \]
\[ z(iz) = y(iy); \ iy=iy+1; \ iz=iz+1; \]
\[ \text{end} \]
\[ \text{end} \]
\[ \text{while } ix \leq nx \text{ } \% \text{ copy remaining } x\text{-values} \]
\[ z(iz) = x(ix); \ ix=ix+1; \ iz=iz+1; \]
\[ \text{end} \]
\[ \text{while } iy \leq ny \text{ } \% \text{ copy remaining } y\text{-values} \]
\[ z(iz) = y(iy); \ iy=iy+1; \ iz=iz+1; \]
\[ \text{end} \]
Merge sort: \( \log_2(N) \) “levels”; \( N \) comparisons each level

Starting with an unsorted array, the algorithm divides the array into smaller subarrays, sorts them, and then merges them back together.

H E M G B K A Q F L P D R C J N
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 \textit{in-place}; merge sort (recursion) requires much more memory

See \texttt{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
Why not just use Matlab’s sort function?

- **Flexibility**
- E.g., to maintain a sorted list, just write the code for insertion sort
- E.g., sort strings or other complicated structures
- Sort according to some criterion set out in a function file
  - Observe that we have the comparison $x(j+1) < x(j)$
  - The comparison can be a function that returns a boolean value
- Can combine different sort/search algorithms for specific problem
We’ve reached the end of CS1112… now what?

- Continue practicing your problem solving—problem decomposition—skills, in programming and other arenas!
- Interested in further study?
  - ENGRD/CS 2110 Object-oriented programming and data structure
ENGRG/CS 2110 OOP and Data Structures

- Learn new programming concepts and further explores those you’ve seen in CS1112
  - OOP, program design and development
  - Recursion
  - Complex data structures and related algorithms

- Taught in Java

- Optional CS 2111 meets 1 hr/week; additional practice with OOP, Java, and other course topics

- During break, check out this website: http://www.cs.cornell.edu/courses/CS1130/2015sp/
We’ve reached the end of CS1112... now what?

- Continue practicing your problem solving—problem decomposition—skills, in programming and other arenas!
- Interested in further study?
  - ENGRD/CS 2110 Object-oriented programming and data structure
  - Short courses in Python (CS 1133), C++ (CS 2024), …, etc.
  - More general CS courses: CS 2800 Discrete structures, CS 2850 Networks
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

- Dec 7, 2-4:30pm, Rockefeller Hall 201 (A-N), 203 (O-Z)
- 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

- Dec 7, 2-4:30pm, Rockefeller Hall 201 (A-N), 203 (O-Z)
- 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!