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

Linear search and binary search

Linear search
- “Effort” is 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.

Subdivide the sorting task
Subdivide again

And again

And one last time

Now merge

And merge again

And again
And one last time

The central sub-problem is the merging of two sorted arrays into one single sorted array

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 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
    \[\text{Deal with remaining values in } x \text{ or } y\]
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
    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 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, 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

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: ___________________ for big N
- Merge sort: ___________________
- Insertion sort is done in-place; merge sort (recursion) requires much more memory

See InsertionSort.m
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)
% All the comparisons between
% vector values are done in merge
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

Merge sort:  log2(N) “levels”; N comparisons each level

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