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
Suppose another vector is twice as long as v. The expected “effort” required to do a linear search is …

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
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.

“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 \(x = 70\)

<table>
<thead>
<tr>
<th>v</th>
<th>12</th>
<th>15</th>
<th>33</th>
<th>35</th>
<th>42</th>
<th>45</th>
<th>51</th>
<th>62</th>
<th>73</th>
<th>75</th>
<th>86</th>
<th>98</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mid</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v(Mid)</td>
<td>(\leq x)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
| So throw away the left half...

function \(L = \text{binarySearch}(x, v)\)

\% \(L\) is the index such that \(v(L) \leq x < v(L+1)\).
\% \(L=0\) if \(x \leq v(1)\). If \(x > v(\text{end})\), \(L = \text{length}(v)\) but \(x > v(L)\).
\% Maintain a search window \([L..R]\) such that \(v(L) \leq x < v(R)\). Since \(x\) may not be in \(v\), initialize set \(L=0\); \(R=\text{length}(v)+1\); \(L=\text{floor}(L+R)/2\); middle of search window \(L=R\) until \(R-L=1\), \% always keeping \(v(L) \leq x < v(R)\)

while \(R-L=1\)

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

\% So throw away the right half...

else

end
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

**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

```
[ H E L L O W ]
[ C O M P U T E R ]
[ S O F T W A R E ]
```

### Function: y = mergeSort(x)

```matlab
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

```
12 33 35 45
15 42 55 65 75
12 15 33 35 42 45 55 65 75
```

### Merge

```
x: 12 33 35 45  
   \  
  ix: 1

y: 15 42 55 65 75  
   \  
  iy: 1

z: 12 15 33 35 42 45 55 65 75  
   \  
  iz: 1

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

```matlab
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;
        else
            z(iz) = y(iy);
            iy = iy + 1;
        end
        iz = iz + 1;
    end
    % Deal with remaining values in x or y
end
```

```matlab
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
end
```

```matlab
function x = insertSort(x)
    % Sort vector x in ascending order with insertion sort
    n = length(x);
    for i= 1:n-1
        if x(i+1) < x(i)
            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
    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 \( N \):
  \[
  \text{____________________ for big } N
  \]
- Merge sort:  
- Insertion sort is done \textit{in-place}; merge sort (recursion) requires much more memory

How to choose??

- Depends on application
- Merge sort is especially good for sorting \textit{large data set} (but watch out for memory usage)
- Insertion sort is “order \( N^2 \)” at \textit{worst case}, but what about an \textit{average case}? If the application requires that you maintain a sorted array, insertion sort may be a good choice

What we learned...

- Develop/implement \textit{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 \textit{insight} into a problem

- Computing is \textit{not} about getting one answer!
- We \textit{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

function \( y = \text{mergeSort}(x) \)
\% \( x \) is a vector. \( y \) is a vector
\% consisting of the values in \( x \)
\% sorted from smallest to largest.
\[
\begin{align*}
\text{n} &= \text{length}(x); \\
\text{if } n &= 1 \\
\text{y} &= x;
\text{else} \\
\text{m} &= \text{floor}(n/2); \\
\text{yL} &= \text{mergeSort}(x(1:m)); \\
\text{yR} &= \text{mergeSort}(x(m+1:n)); \\
\text{y} &= \text{merge}(yL,yR);
\end{align*}
\]