Previous Lecture:
- Recursion

Today’s Lecture:
- Sorting and searching
  - Insertion sort, linear search
  - Read about Bubble Sort in Insight
- “Divide and conquer” strategies
  - Binary search

Announcements
- Discussion in computer lab this week
- P6 due Thursday at 11pm
- Final exam: Dec 7th 2pm for both Lec 1 and Lec 2
Sorting data allows us to search more easily.

### Boston Marathon Top Women Finishers

<table>
<thead>
<tr>
<th>Name</th>
<th>Official Time</th>
<th>State</th>
<th>Country</th>
<th>Ctz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tune, Dire</td>
<td>2:25:25</td>
<td>ETH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Biktimirova, Alevtina</td>
<td>2:25:27</td>
<td>RUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jeptoo, Rita</td>
<td>2:26:34</td>
<td>KEN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Prokopcuka, Jelena</td>
<td>2:28:12</td>
<td>LAT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Magarsa, Askale Tafa</td>
<td>2:29:48</td>
<td>ETH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Genovese, Bruna</td>
<td>2:30:52</td>
<td>ITA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Olaru, Nuta</td>
<td>2:33:56</td>
<td>ROM</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Guta, Robe Tola</td>
<td>2:34:37</td>
<td>ETH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Grigoryeva, Lidiya</td>
<td>2:35:37</td>
<td>RUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hood, Stephanie A.</td>
<td>2:44:44</td>
<td>IL</td>
<td>USA</td>
<td>CAN</td>
</tr>
<tr>
<td>Robson, Denise C.</td>
<td>2:45:54</td>
<td>NS</td>
<td>CAN</td>
<td></td>
</tr>
<tr>
<td>Chemjor, Magdaline</td>
<td>2:46:25</td>
<td>KEN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sultanova-Zhdanova, Firaya</td>
<td>2:47:17</td>
<td>FL</td>
<td>USA</td>
<td>RUS</td>
</tr>
<tr>
<td>Mayger, Eliza M.</td>
<td>2:47:36</td>
<td>AUS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Anklam, Ashley A.</td>
<td>2:48:43</td>
<td>MN</td>
<td>USA</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Score</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jorge</td>
<td>92.1</td>
<td></td>
</tr>
<tr>
<td>Ahn</td>
<td>91.5</td>
<td></td>
</tr>
<tr>
<td>Oluban</td>
<td>90.6</td>
<td></td>
</tr>
<tr>
<td>Chi</td>
<td>88.9</td>
<td></td>
</tr>
<tr>
<td>Minale</td>
<td>88.1</td>
<td></td>
</tr>
<tr>
<td>Ball</td>
<td>87.3</td>
<td></td>
</tr>
</tbody>
</table>
There are many algorithms for sorting

- **Insertion Sort** (to be discussed today)
- **Bubble Sort** *(read Insight §8.2)*
- **Merge Sort** (to be discussed Thursday)
- **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!*
The Insertion Process

- **Given a sorted array** $x$, insert a number $y$ such that the result is sorted.
Insertion

one insert process

Insert 8 into the sorted segment

Just swap 8 & 9
Insertion

one insert process

Insert 4 into the sorted segment

sorted
Insertion

one insert process

Compare adjacent components: swap 9 & 4
Insertion

one insert process

Compare adjacent components: swap 8 & 4
Insertion

one insert process

Compare adjacent components: swap 6 & 4
Insertion

one insert process

Compare adjacent components: DONE! No more swaps.

See Insert.m for the insert process
Sort vector \( \mathbf{x} \) using the Insertion Sort algorithm

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

\[
\mathbf{x}
\]

Length 1 subvector is “sorted”

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

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

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

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

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

\text{InsertionSort.m}
Insertion Sort vs. Bubble Sort

- Read about Bubble Sort in *Insight* §8.2
- Both algorithms involve the repeated comparison of adjacent values and swaps
- Find out which algorithm is more efficient on average
Other efficiency considerations

- Worst case, best case, average case
- Use of subfunction incurs an “overhead”
- Memory use and access

Example: Rather than directing the *insert* process to a subfunction, have it done “in-line.”

Also, Insertion sort can be done “in-place,” i.e., using “only” the memory space of the original vector.
function x = InsertionSortInplace(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
end
function x = InsertionSortInplace(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
        % swap x(j+1) and x(j)
        j = j-1;
    end
end
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 or two statements
C. About half the code
D. Most of the code

The only change is in how we do the comparison!

See InsertionSortIntervals.m
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
% 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

v = [12 35 33 15 42 45]
x = 31
% 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

v  12 15 33 35 42 45
x  31

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 2\textsuperscript{nd} half.
    else
        Rip and throw away the 1\textsuperscript{st} 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

\[ n \text{ comparisons against the target are needed in worst case, } \] 
\[ n = \text{length}(v). \]
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 $x = 70$

```
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 $x = 70$

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

L:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
</table>

v

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

Mid:

| 6 |

v(Mid) <= x

So throw away the left half...

R:

| 7 |

| 9 |
Binary search: target $x = 70$

$v(Mid) \leq x$

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

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

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
        % code here
    else
        % code here
    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
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

20 30 40 46 50 52 68 70
0 1 2 3 4 5 6 7 8 9

Play with showBinarySearch.m