■ Previous Lecture:
  ■ Recursion (Ch. 14)

■ Today, Lecture 26:
  ■ Algorithms for sorting and efficiency analysis (Ch. 8)
    ■ Insertion Sort algorithm
    ■ See Insight §8.2 for the Bubble Sort algorithm
  ■ Algorithms for searching and analysis (Ch. 9)
    ■ Linear search (review)
    ■ Binary search

■ Thursday:
  ■ Merge sort
  ■ Watch sorting video

■ Announcements:
  ■ Project 6 due Thurs 11pm EDT
  ■ Ex 13 due Fri 11pm EDT
  ■ Fill out final exam logistics survey on Canvas
  ■ Please complete course evaluations
Sorting data allows us to search more easily.

<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>
There are many algorithms for sorting

- **Insertion Sort** (to be discussed today)
- **Bubble Sort** (read *Insight* §8.2)
- **Merge Sort** (to be discussed next lecture)
- **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.

```
2 3 6 9
sorted
8
```


```
2 3 6 8 9
```
Insertion

one insert process

Insert 8 into the sorted segment

Just swap 8 & 9
Insertion

one insert process

2 3 6 9 8
2 3 6 8 9

Insert 4 into the sorted segment

Compare adjacent components: swap 9 & 4

2 3 6 8 9 4
2 3 6 8 4 9
2 3 6 4 8 9
2 3 4 6 8 9

See function \texttt{Insert} 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”

- Insert $\mathbf{x}(2): \mathbf{x}(1:2) = \text{Insert}(\mathbf{x}(1:2))$
- Insert $\mathbf{x}(3): \mathbf{x}(1:3) = \text{Insert}(\mathbf{x}(1:3))$
- Insert $\mathbf{x}(4): \mathbf{x}(1:4) = \text{Insert}(\mathbf{x}(1:4))$
- Insert $\mathbf{x}(5): \mathbf{x}(1:5) = \text{Insert}(\mathbf{x}(1:5))$
- Insert $\mathbf{x}(6): \mathbf{x}(1:6) = \text{Insert}(\mathbf{x}(1:6))$

`insertionSortSimple.m`
Contract between Insert and InsertionSort

**Insert**
- Assumes all but the last element of x is already sorted
- Returns a fully-sorted array (one more element sorted than given)

**InsertionSort (driver)**
- Must only call Insert() on a subarray with a pre-sorted prefix
- Has a bigger pre-sorted subarray to pass to Insert() next time – progress is made each iteration

*therefore*

Size of sorted prefix grows each time.
When it equals the size of the original array, the task is done
How much “work” is insertion sort?

- In the worst case, make $k$ comparisons to insert an element in a sorted array of $k$ elements.
Insertion

one insert process

Insert into sorted array of length 4

2 3 6 9

Insert into sorted array of length 5

2 3 6 8 9

2 3 6 8 4

2 3 6 8 4 9

2 3 4 6 8 9
How much “work” is insertion sort?

- In the worst case, make $k$ comparisons to insert an element in a sorted array of $k$ elements. For an array of length $N$:

  \[ 1 + 2 + \ldots + (N-1) = \frac{N(N-1)}{2}, \text{ say } N^2 \text{ for big } N \]

`InsertionSort.m`
Checkpoint question: $N^2$ performance

Suppose it takes 5ms to sort an array with 100 elements using Insertion Sort. How long would you expect sorting 1000 elements to take?

A. 25ms  
B. 50ms  
C. 500ms  
D. 5000ms  
E. 1e6 ms
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
A note on optimization

- “Inlining” multiple pieces of an algorithm should not be your go-to strategy
  - It’s easier to understand (and verify) small pieces that do a simple task than monolithic code that does a complicated task
    - Better communication, less buggy
  - Hard to predict when it will actually be faster
    - Large code has a performance cost in addition to a maintenance cost
    - Measuring performance not as easy as it sounds
- Compilers can do this automatically
  - Auto-inlining will reveal opportunities for in-place array edits
Sort an array of objects

- Given \( x \), a 1-d array of \textit{Interval} references, sort \( x \) according to the widths of the \textit{Intervals} from narrowest to widest
- Use the insertion sort algorithm
- How much of our code needs to be changed?

A. No change
B. One statement
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
% 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).
% 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?
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 Reef’s number…

while (Phone book is longer than 1 page)
  Open to the middle page.
  if “Reef” 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.

“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>
What is true of the half we keep?

- Let L be the leftmost page we keep (may be 0, aka front cover)
- Let R be the page after the last one we keep (might be \(\text{length}(v) + 1\), aka back cover)
- Then the name we are looking for is \(\geq\) the first name on page L, and \(<\) the first name on page R
- When only one page left (\(R = L+1\)),
  - If name is in book, it will be on page L
  - If name is not in book, it should be inserted after some names already on page L
Binary search: target $x = 70$

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

So throw away the left half...