- Previous Lecture:
  - Characters arrays (type `char`)
  - Review top-down design
  - Linear search

- Today’s Lecture:
  - More on linear search
  - Cell arrays

- Announcements:
  - P4B deadline extended until Wed
  - Tutoring Thurs, Sun, Mon (sign up on CMS)
  - Prelim 2: Apr 22 (Thurs)
Initialize array of counts to zeros

For each item/trial:
  Determine tally index
  Increment count at index

In 1D, can use `bar()` to visualize histogram of counts
% dna sequence encoding protein
p = ['TTCGGGAGCCTGGGCGTTACGTTAATGAAA' ...
     'ATATGTACCAACGACAATGACATTGAAAAC'];

count = zeros(1,20); % to store tallies

for k = 1:3:length(p)-2
    codon = p(k:k+2); % length 3 subvector
    mnem = getMnemonic(codon);
    % Tally: build histogram data
    ind = getAAIndex(mnem);
    count(ind) = count(ind) + 1;
end

bar(1:20, count) % Draw bar chart
function ind = getAAIndex(aa)
% Returns index of amino acid named by char vector aa.
% If aa does not name an amino acid, throw an error.

% Display an error message and STOP program execution. (Not just a print statement.)
% Use built-in function error.
% See getAAIndex.m

Syntax: error( )
message to display
Linear Search algorithm

\[ k = 1 \]
\[ \text{while } k \text{ is valid and item at } k \text{ does not match search target} \]
\[ k = k + 1 \]
\[ \text{end} \]
% 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
Suppose another vector is twice as long as v. The expected “effort” required to do a linear search is ...
Linear search:
Effort linearly proportional to length of vector searched

See `linearSearch.m`, `analyzeLinearSearch.m`
Basic (simple) types in MATLAB

• E.g., char, double, unit8, logical
• Each uses a set amount of memory
  • Each uint8 value uses 8 bits (=1 byte)
  • Each double value uses 64 bits (=8 bytes)
  • Each char value uses 16 bits (=2 bytes)
  • Use function whos to see memory usage by variables in workspace
• Can easily determine amount of memory used by a simple array (array of a basic type, where each component stores one simple value)
• Next: Special arrays where each component is a container for a collection of values
Limitations of primitive arrays

• Homogeneous data type
  • Can't represent tables

• Not nestable
  • No ragged arrays, lists-of-lists
  • Concatenation always "flattens"

• Multiple strings are awkward

\[
\begin{array}{cccccccccccc}
'A' & 'l' & 'a' & 'b' & 'a' & 'm' & 'a' & '\,'

'N' & 'e' & 'w' & 'Y' & 'o' & 'r' & 'k'

'U' & 't' & 'h' & ' ' & ' ' & ' ' & ' ' & ' '
\end{array}
\]

• ["John Doe", 33, true]
  • Error using `horzcat`

• [1, 2, 3; ... 4, 5]
  • Error: Invalid expression.

• [1, [2, 3], 4]
  • 1 2 3 4
New data type: Cell

• A cell's value may be of any type
  • Array of doubles
  • Array of characters
  • Array of more cells

• Each cell in an array may have a different type & size

• Arrays of cells are still rectangular
Array vs. Cell Array

- **Simple array**
  - Each component stores one scalar. E.g., one `char`, one `double`, or one `uint8` value
  - All components have the same type

- **Cell array**
  - Each cell can store something “bigger” than one scalar, e.g., a vector, a matrix, a `char` vector
  - The cells may store items of different types

```
'CS' '1'1'1'2'
```
```
'CS' 1.1 -7 12 8 1.1 -1 12
```
Application: lists of strings

- $C = \{ \text{'Alabama'}, \text{'New York'}, \text{'Utah'} \}$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>'Alabama'</td>
<td>'New York'</td>
<td>'Utah'</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

- $C = \{ \text{'Alabama'}; \text{'New York'}; \text{'Utah'} \}$

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>'Alabama'</td>
<td>'New York'</td>
<td>'Utah'</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Compare with:

```
1,: ['A' 'l' 'a' 'b' 'a' 'm' 'a'
2,: ['N' 'e' 'w' 'y' 'o' 'r' 'k'
3,: ['U' 't' 'a' 'h'
```
Use braces for creating & indexing cell arrays

**Primitive arrays**

- Create
  
  ```matlab
  m = [ 5, 4; 1, 2; 0, 8 ]
  ```

- Index
  
  ```matlab
  m(2,1) = pi
  disp(m(3,2))
  ```

**Cell arrays**

- Create
  
  ```matlab
  C = { ones(2,2), 4 ; 
       'abc', ones(3,1) ; 
       9 , 'a cell' }
  ```

- Index
  
  ```matlab
  C{2,1} = 'ABC'
  C{3,2} = pi
  disp(C{3,2})
  ```
Creating cell arrays

C = {'Oct', 30, ones(3,2)};

is the same as
C = cell(1,3); % optional
C{1} = 'Oct';
C{2} = 30;
C{3} = ones(3,2);

Can assign empty cell array
D = {};

Comparison of bracket operators

• Square brackets []
  • Create primitive array
  • Concatenate (any) array contents

\[
\begin{bmatrix}
3 & 14 & 1 & 5 & 9 \\
\end{bmatrix}
\]
\[
\begin{bmatrix}
'a' \{ 'b' ['c' 'd'] \} \\
\end{bmatrix}
\Rightarrow
\{ 'a', 'b', 'cd' \}
\]

• Curly braces {}
  • Create cell array enclosing contents

\{
3 \{ 1 4 \} 1 \{ 5 9 \} \\
\{ 'a' \{ 'b' 'cd' \} \}
\}
Example: Represent a deck of cards with a cell array

D{1} = ‘A Hearts’;
D{2} = ‘2 Hearts’;

D{13} = ‘K Hearts’;
D{14} = ‘A Clubs’;

D{52} = ‘K Diamonds’;

But we don’t want to have to type all combinations of suits and ranks in creating the deck... How to proceed?
Make use of a suit array and a rank array …

suit = {'Hearts', 'Clubs', ...
        'Spades', 'Diamonds'};

rank = {'A','2','3','4','5','6',...
       '7','8','9','10','J','Q','K'};

Then concatenate to get a card. E.g.,

str = [rank{3} ' ' suit{2} ];
D{16} = str;

So D{16} stores ‘3 Clubs’
To get all combinations, use nested loops

```
suit = {'Hearts','Clubs','Spades','Diamonds'};
rank = {'A','2','3','4','5','6','7','8','9','10','J','Q','K'};
i = 1; % index of next card
for k = 1:4
    % Set up the cards in suit k
    for j = 1:13
        D{i} = [ rank{j} ' ' suit{k} ];
        i = i + 1;
    end
end
See function CardDeck
```
Example: deal a 12-card deck
% Deal a 52-card deck

N = cell(1,13); E = cell(1,13);
S = cell(1,13); W = cell(1,13);

for k=1:13
    N{k} = D{4*k-3};
    E{k} = D{4*k-2};
    S{k} = D{4*k-1};
    W{k} = D{4*k};
end

See function Deal
The “perfect shuffle” of a 12-card deck
Perfect Shuffle, Step 1: cut the deck

A B C D E F G H I J K L

A B C D E F

G H I J K L
Perfect Shuffle, Step 2: Alternate

A B C D E F G H I J K L

A B C D E F

1 2 3 4 5 6

G H I J K L

A G B H C I D J E K F L

1 2 3 4 5 6 7 8 9 10 11 12
Perfect Shuffle, Step 2: Alternate

A B C D E F G H I J K L

A B C D E F

G H I J K L

A G B H C I D J E K F L

2 4 6 8 10 12
Perfect Shuffle, Step 2: Alternate

See function Shuffle