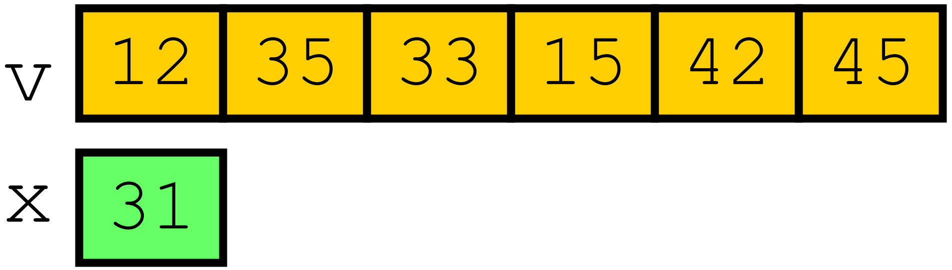
- Previous Lecture:
  - Characters arrays (type `char`)
  - Review top-down design
  - Linear search
- Today's Lecture:
  - More on linear search
  - Cell arrays
  - Application of cell array: input from (and output to) a text file
- Announcements:
  - Discussion section in Zoom today/tomorrow
  - Tutoring Wed-Mon (sign up on Canvas)
  - Thurs lecture quizzes to be submitted via Gradescope

From last lecture: Linear Search

- Search: Linear Search Algorithm

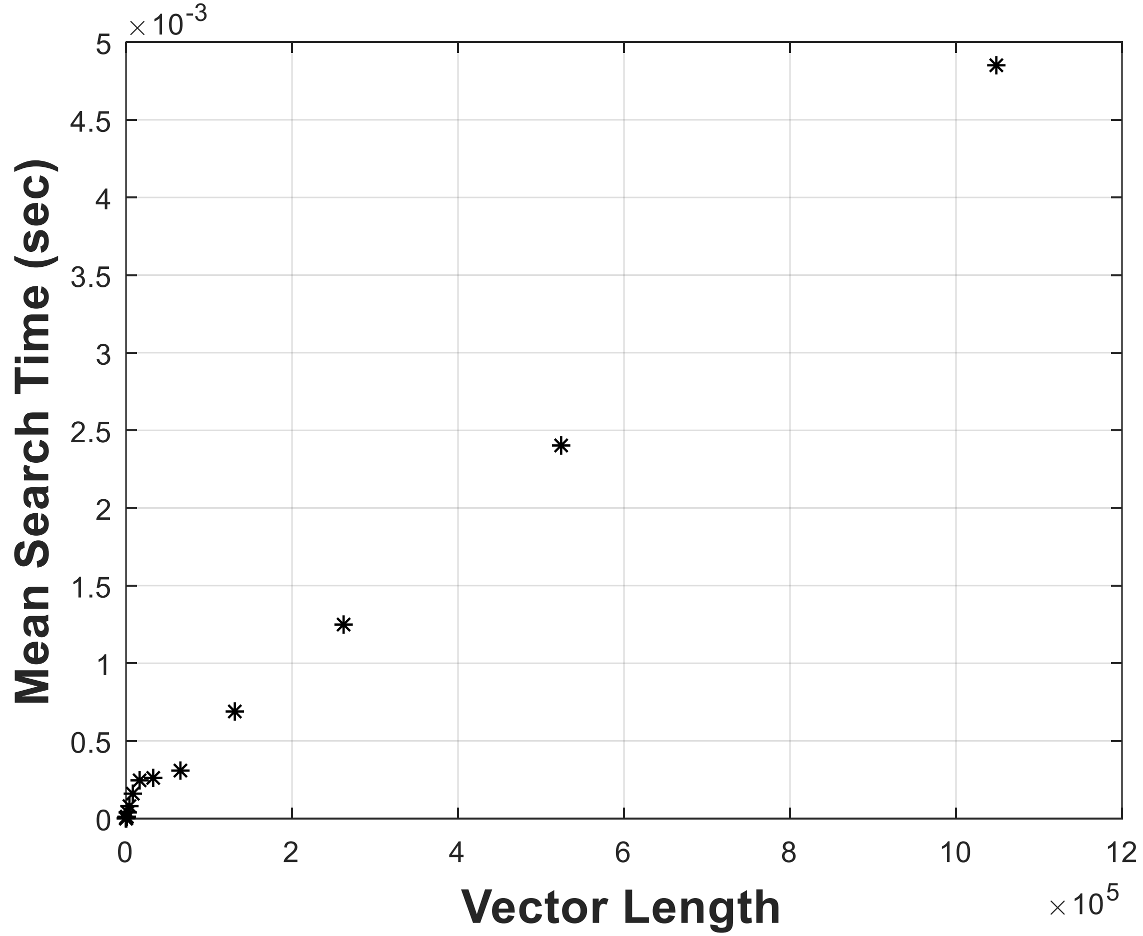
```
k= 1
while  k is valid and
           item at k does not match search target

       k= k + 1
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
```



See `linearSearch.m`, `analyzeLinearSearch.m`

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

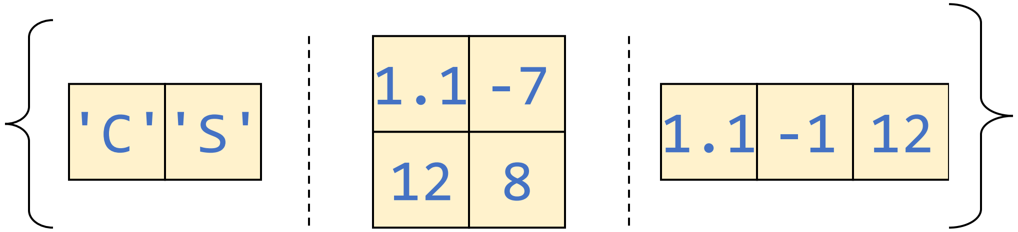
- Homogeneous data type
  - Can't represent tables
- Not nestable
  - No ragged arrays, lists-of-lists
  - Concatenation always "flattens"
- Multiple strings are awkward

'A'	'l'	'a'	'b'	'a'	'm'	'a'	' '
'N'	'e'	'w'	' '	'Y'	'o'	'r'	'k'
'U'	't'	'a'	'h'	' '	' '	' '	' '

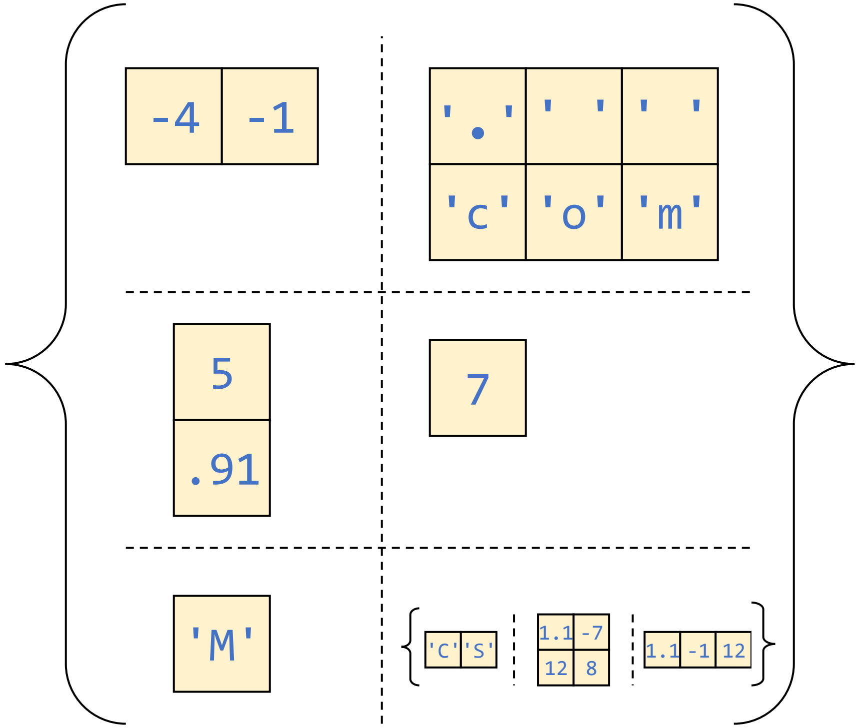
- `['John Doe', 33, true]`
  - `Error using` **`horzcat`**
- `[1, 2, 3; ...`
  `4, 5]`
  - `Error: Invalid expression.`
- `[1, [2, 3], 4]`
  - `1 2 3 4`

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

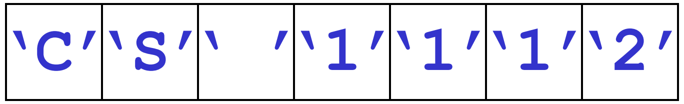


- Arrays of cells are still rectangular

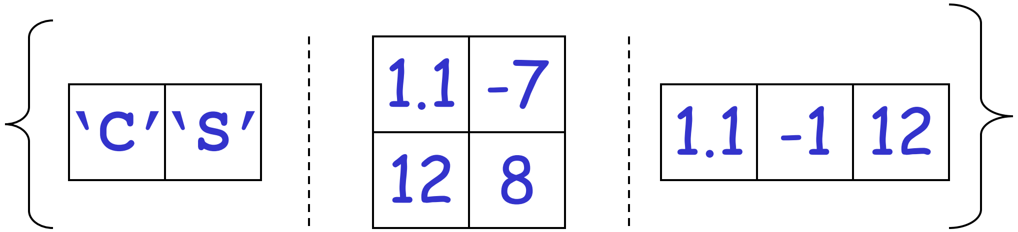


Array vs. Cell Array

- Simple array



  - Each component stores <u>one scalar</u>. 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

Application: lists of strings

- C = { 'Alabama', 'New York', 'Utah' }

'Alabama'	'New York'	'Utah'
1	2	3

- C = { 'Alabama'; 'New York'; 'Utah' }

1	'Alabama'
2	'New York'
3	'Utah'

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

```
m = [ 5, 4; …
      1, 2; …
      0, 8 ]
```

- Index

```
m(2,1) = pi
disp(m(3,2))
```

Cell arrays

- Create

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

- Index

```
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

```
[3 [1 4] 1 [5 9]]
[ 'a' {'b' ['c' 'd']} ] ⇒
{ '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

D:

N: 1,5,9 $4k-3$

E: 2,6,10 $4k-2$

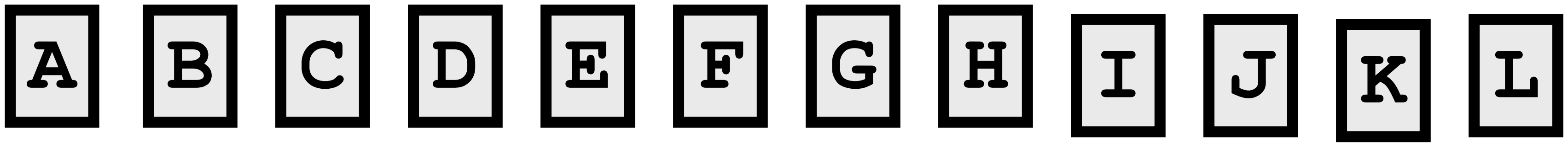
S: 3,7,11 $4k-1$

W: 4,8,12 $4k$

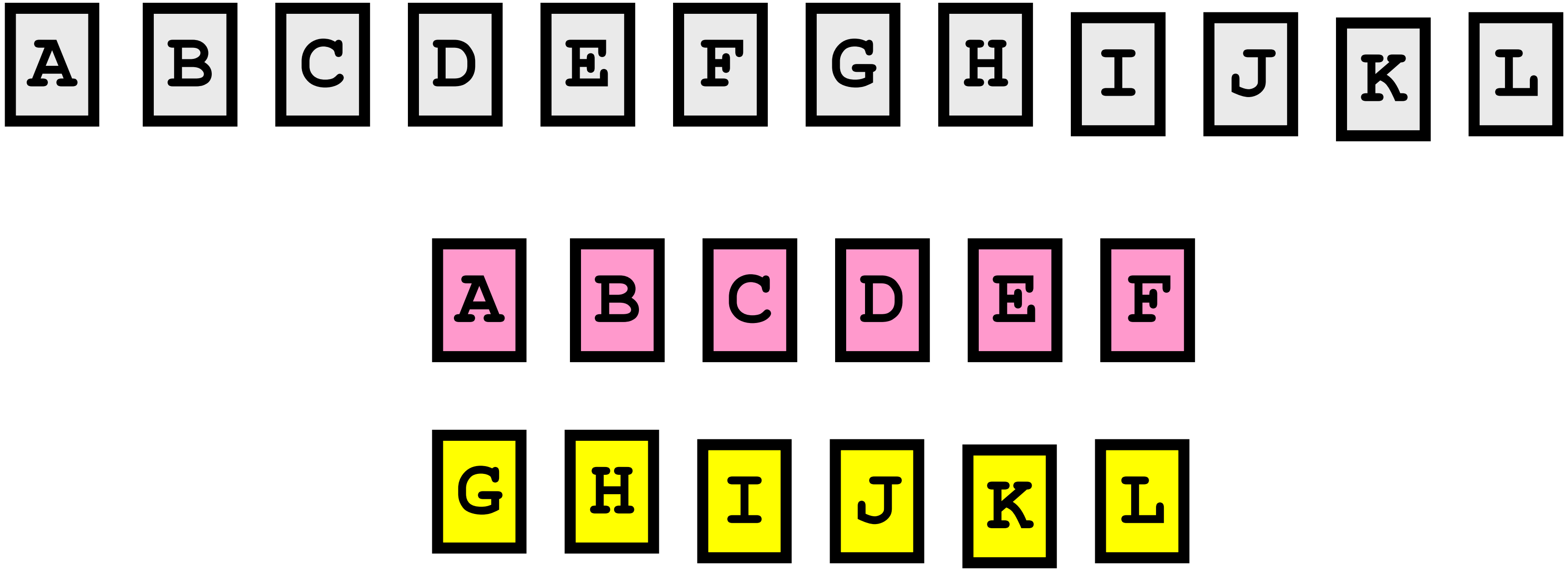
```
% Deal a 52-card deck
N = cell(1,13); E = cell(1,13);
S = cell(1,13); W = cell(1,13);

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`

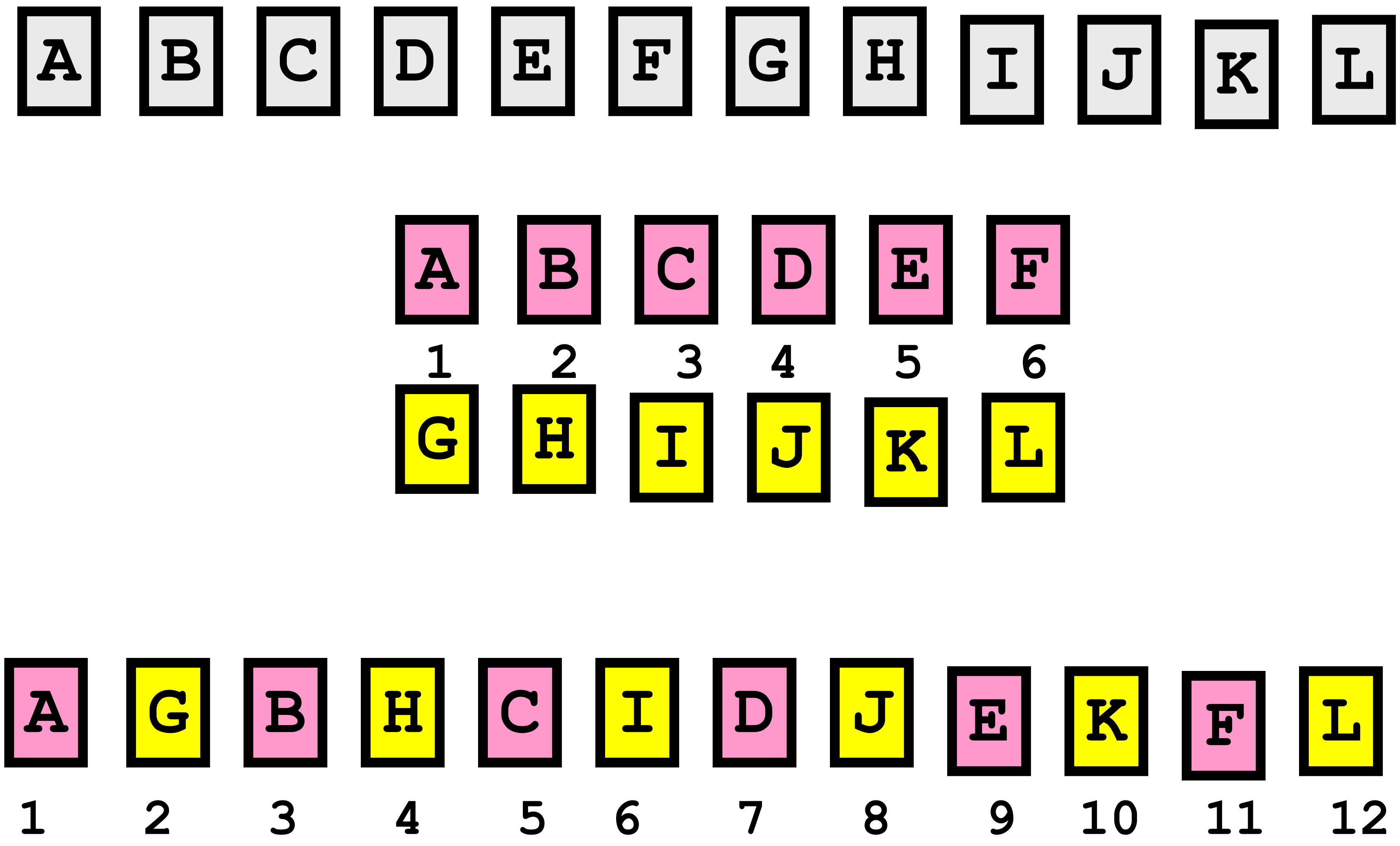
The “perfect shuffle” of a 12-card deck



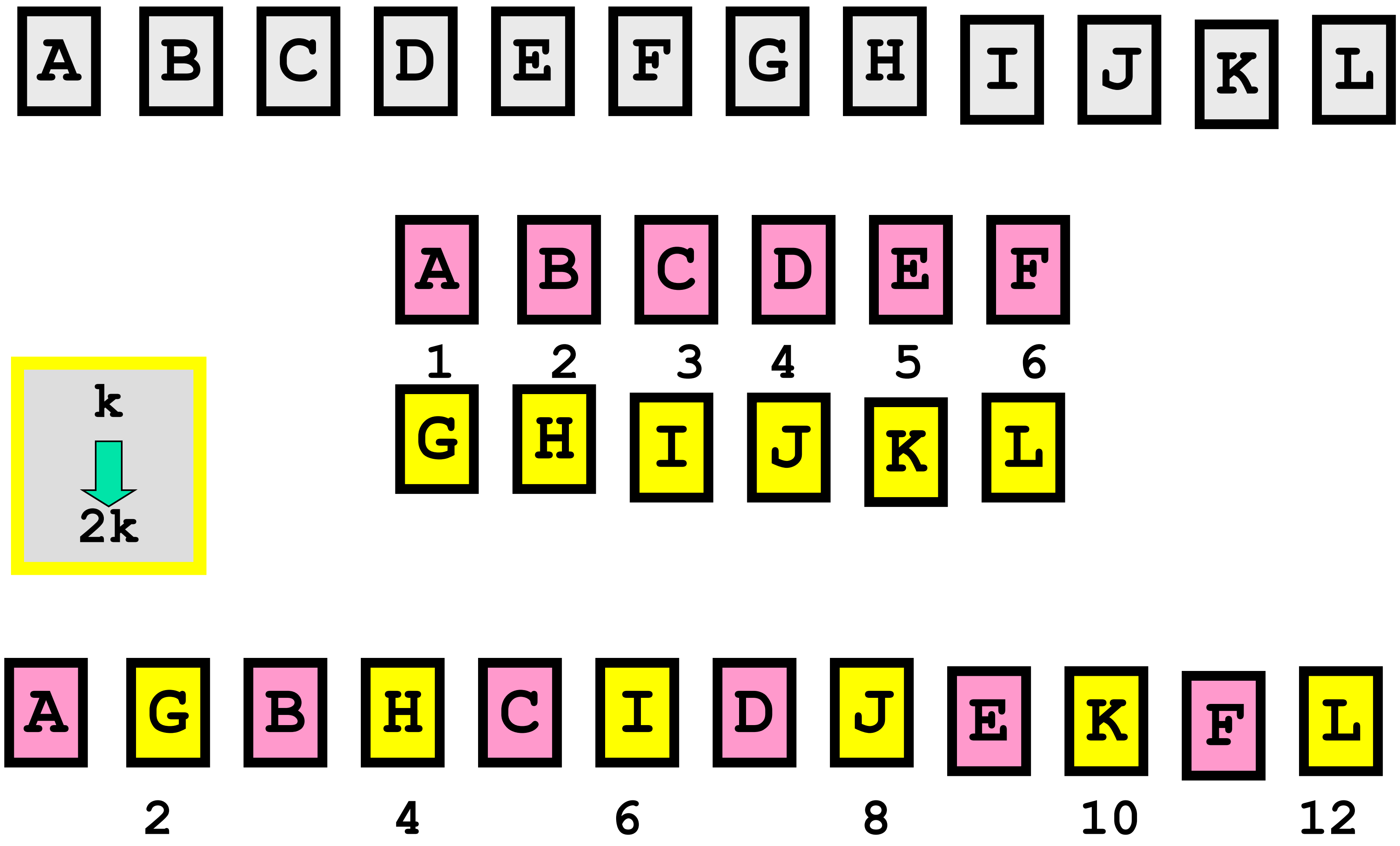
Perfect Shuffle, Step 1: cut the deck



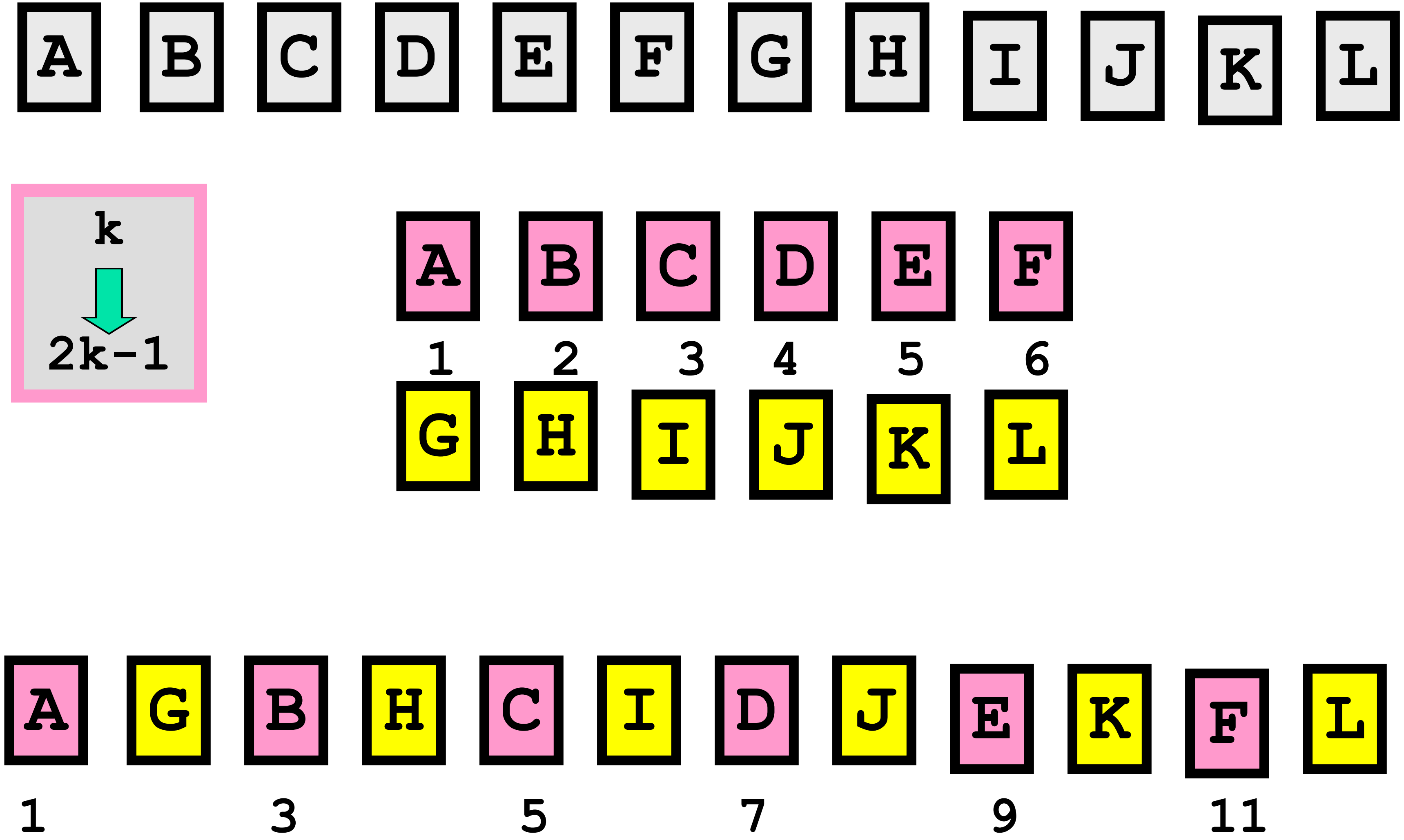
Perfect Shuffle, Step 2: Alternate



Perfect Shuffle, Step 2: Alternate



Perfect Shuffle, Step 2: Alternate



See function `Shuffle`

I want to put in the 3rd cell of cell array C a **`char`** row vector. Which is correct?

A. C{3} = 'a cat';
B. C{3} = ['a ' 'cat'];
C. C(3) = {'a ' 'cat'};
D. Two answers above are correct
E. Answers A, B, C are all correct