USER DEFINED FUNCTIONS*

Functions in Matlab

- The MATLAB programming language is built around functions. A *function* is a piece of computer code that accepts an input argument from the user and provides output to the program.

- Functions allow us to program efficiently, enabling us to avoid rewriting the computer code for calculations that are performed frequently.

- For example most computer programs contain a function that calculates the sine of a number.

- In MATLAB, *sin* is the function name used to call up a series of commands that perform the necessary calculations.

- The user needs to provide an angle, and MATLAB returns a result.

- It isn't necessary for the programmer to know how MATLAB calculates the value of *sin(x)*.

User Defined Functions

- We may wish to define our own functions – those are used commonly in our programming.

- User-defined functions are stored as *m-files* and can be accessed by MATLAB, if they are in the current folder or on MATLAB's search path.

- Both built-in MATLAB functions and user-defined MATLAB functions have the same structure. Each consists of a name, user-provided input, and calculated output.

- For example, the function *cos(x),* is named *cos*, takes the user input inside the parentheses (in this case, *x*), and calculates a result.

- The user does not see the calculations performed, but just accepts the answer.

Structure of a Function

- User-defined functions work the same way. Imagine that you have created a function called my_function.

- Using `my_function(x)` in a program or from the command window will return a result, as long as `x` is defined and the logic in the function definition works.

- User-defined functions are created in m-files.

- Each must start with a function definition line that contains:
  - The word **function**,
  - A variable that defines the function output,
  - A function name,
  - A variable used for the input argument.

Structure of a Function, ctd.

Input $\longrightarrow$ | FUNCTION | $\longrightarrow$ Output

- A function is a box,

- It hides the code and its workspace and communicates with the "world" using the input and output variables

Functions in m-file Structure

- A very simple MATLAB function that calculates the value of a particular polynomial:

```
function output = poly(x)
%This function calculates
%the value of a third-order polynomial
output = 3*x.^3 + 5*x.^2 - 2*x +1;
```

- The function name is `poly`, the input argument is `x`, and the output variable is named `output`.

- Before this function can be used, it **must be saved into the current folder**. The file name *must be the same* as the function name in order for MATLAB to find it.

Functions in m-file Structure, ctd.

- All of the MATLAB naming conventions we learned for naming variables apply to naming user-defined functions.

    In particular,

- The function name must start with a letter.

- It can consist of letters, numbers, and the underscore.

- Reserved names cannot be used.

- Any length is allowed, although long names are not good programming practice.

- Once the m-file has been saved, the function is available for use from the command window, from a script m-file, or from another function.

- You cannot execute a function m-file directly from the m-file itself.

- This makes sense, since the input parameters have not been defined until you call the function from the command window or a script m-file.

Functions in m-file Structure, ctd.

- Consider the poly function just created. If, in the command window, we type `poly(4)` then MATLAB responds with

  ```
  ans =
  265
  ```

- If we set `a` equal to 4 and use `a` as the input argument, we get the same result:

  ```
  a = 4;
  poly(a)
  ans =
  265
  ```

Example

- A Matlab function `sumOfNumbers.m`

```
function result = sumOfNumbers(a,b)

result = a+b;
```

The input variable(s)

output variable(s)

- We can use [ ]: if there are more than one output variables ([out1 out2]).

Example, ctd.

- Assume we wrote the function:

```
function result = sumOfNumbers(a,b)
result = a+b;
```

and in the workspace we run:

```
a = 1;
b = 2;
x = 3;
y = 4;
s = sumOfNumbers(x, y)
```

What is the output?
```
s = 7
```

Matlab

Workspace:
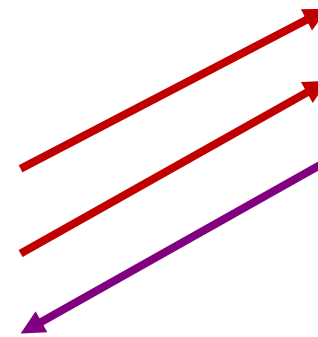
a = 1

b = 2

x = 3

y = 4

s = 7

Function

Workspace:

a = 3

b = 4

result = 7

Example-2

- Write and test a function **DR** that changes degrees to radians and another function **RD** that changes radians to degrees. The functions should be able to accept both scalar and matrix input.

1. State the Problem

Create and test two functions, **DR** and **RD** , to change degrees to radians and radians to degrees

2. Describe the Input and Output

| **Input** | A vector of degree values |
| | A vector of radian values |

| **Output** | A table converting degrees to radians |
| | A table converting radians to degrees |

3. Develop a Hand Example

*degrees = radians * 180/pi*

*radians = degrees * pi/180*

Example-2, ctd.

4. Develop a MATLAB Solution

```
function y = dr(x)              function deg = rd(rad)
y = x*pi/180;                   deg = rad*180/pi;
```

5. Test the solution
```
x = 0:45:180;
in_rad = dr(x)
in_rad =
    0    0.7854    1.5708    2.3562    3.1416
rd(in_rad)
ans =
    0     45     90    135    180
```

Comments & Help for Functions

In a MATLAB function, the comments on the line immediately following the very first line serve a special role. These lines are returned when the help function is queried from the command window.

Consider, for example, the following function:

```
function results = func(x)
%This function converts seconds to minutes
results = x./60;
```

- Querying the help function from the command window

```
help func
```

- returns

```
This function converts seconds to minutes
```

Multi Input and Output Functions

```
function [x,v,a] = motion(t,x0)
% This function calculates the lateral position (x),
% velocity (v), and acceleration (a) of a mass attached
% to a spring for a given value of time (t) and initial
% position (x0) assuming the velocity is initially 0.
x = x0.*cos(t);
v = -x0.*sin(t);
a = -x0.*cos(t);
```

We can use the function to find values of position, velocity, and acceleration of the mass at specified times:

```
[x,v,a] = motion(3,1)
x =
   -0.9900
v =
   -0.1411
a =
    0.9900
```

Multi Input and Output Func's, ctd.

- If we call the motion function without specifying all three outputs, only the first output will be returned:

```
motion(3,1)
ans =
    -0.9900
```

- Using a vector of time values from 0 to 3 in the motion function returns three row vectors of answers:

```
time=0:1:3;
[x,v,a] = motion(time,1)
x =
    1.0000     0.5403    -0.4161    -0.9900
v =
         0    -0.8415    -0.9093    -0.1411
a =
   -1.0000    -0.5403     0.4161     0.9900
```

Some Useful Commands for Func's

Command	Description
nargin	Number of function input arguments
nargout	Number of function output arguments
narginchk()	Validate number of input arguments
error()	Display message and abort function
warning()	Display descriptive warning message

Examples:

```
nargin('motion')
ans =
     2
nargout('motion')
ans =
     3
```

Local Variables

- The variables used in function m-files are known as *local variables* . The only way a function can communicate with the workspace is through input arguments and the output it returns.

- Any variables defined within the function exist only for the function to use. For example, consider the *g* function:

```
function output = g(x,y)
% This function multiplies x and y together
% x and y must be the same size matrices
a = x .*y;
output = a;
```

- The variables *a, x, y,* and *output* are local variables. They can be used for additional calculations inside the *g* function, but they are not stored in the workspace.

Global Variables

- Unlike local variables, global variables are available to all parts of a computer program. In general, *it is a bad idea* to define global variables. However, MATLAB protects users from unintentionally using a global variable by requiring that it be identified both in the command-window environment (or in a script m-file) and in the function that will use it. Consider the distance function:

```
function result = distance(t)
%This function calculates the distance a falling
% object travels due to gravity
global G
result = 1/2*G*t.^2;
```

- The global command alerts the function to look in the workspace for the value of `G`. `G` must also have been defined in the command window (or script m-file) as a global variable:

```
global G
G = 9.8;
```

- This approach allows you to change the value of `G` without needing to redefine the distance function or providing the value of `G` as an input argument to the distance function.

Example-3

- Factorial calculation of a scalar in a function:

```
function y = fact(x)
if length(x)~=1 || x < 0
    error('You entered an invalid number');
end
f = 1;
if x>0
    for i=1:x
        f = f * i;
    end
else
    f = 1;
end
y=f;
```

Example-4

- Calculation the roots of a quadratic polynomial:

```
function [x1 x2] = findroots(a,b,c)

% This function computes the roots of a
% quadratic polynomial whose coefficients
% a,b and c are given (ax^2+bx+c=0).

delta = b^2-4*a*c;
x1 = (-b+sqrt(delta))/(2*a);
x2 = (-b-sqrt(delta))/(2*a);
```

Subfunctions

- A function M-file may contain the code for more than one function.

- The first function in a file is the *primary function,* and is the one invoked with the M-file name.

- Additional functions in the file are called *subfunctions,* and are visible only to the primary function and to other subfunctions.

- Each *subfunction* begins with its own function definition line.

- Subfunctions follow each other in any order *after* the primary function.

Subfunctions - Example

```
function [x2 x3] = xsc(x)
% Calculation of the square and the cube of the entered
scalar
x2 = sq(x);
x3 = cube(x);


function x2 = sq(x)
x2 = x*x;


function y = cube(a)
y = sq(a)*a;
```

```
>> [sq cu] = xsc(5)
sq =
    25
cu =
   125
```

Some Examples about **life span** and **scope**

- Variable **life span** - from the moment it is created until its memory is released

- Variable **scope** – the places in the program where it is recognized and can be modified.

- **Local variables:**
  - The scope is restricted to within the function in which variable was created.
  - The lifespan is limited to the running time of the function in which variable was created (or until cleared)
  - If a local variable is created in the workspace:
    - The scope is restricted to the workspace
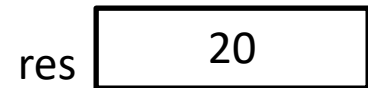   - The lifespan is as long as Matlab runs (or until cleared)

Scope


```
function res = mult2nums(x,y)
res = x * y;
```
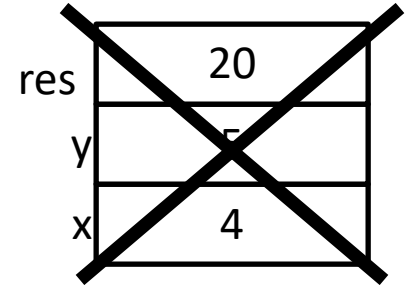
```
>> res = mult2nums(4, 5);
>> x * y
```

??? Undefined function or variable 'x'.

Scope

```
function res = mult2nums(x,y)
res = a * b;
```

y	5
x	4

```
>> a = 100;
>> b = 200;
>> res = mult2nums(4, 5);
>> x * y
```

b	200
a	100

??? Undefined function or variable 'a'.

Error in ==> mult2nums at 2
a * b
>>

Call by value

- ## Formal parameters

  named in the statement of the function

- ## Actual parameters

  with which the function is called

Formal parameters (copy)

```
function res = mult2nums(x,y)
                res = x * y;
```

Actual parameters (original)

```
>> x = 
>> b = y;
>> res = mult2nums(x, y);
```
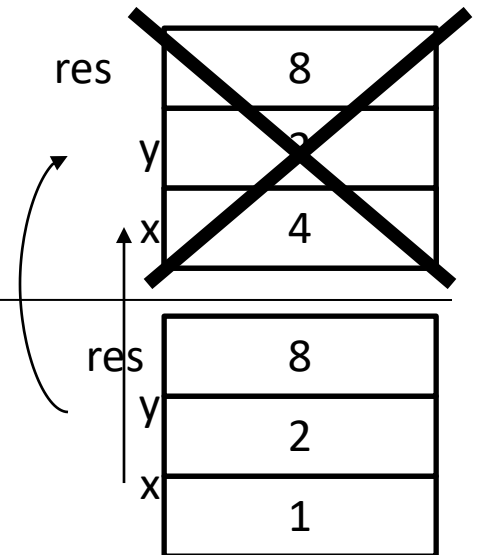
Call by value

Subtle yet very important point:

When a function is called the actual parameters are copied into the formal parameters

```
function res = mult2nums(x,y)
x    = x + 3
res = x * y;
```

```
>> x = 1;
>> y = 2;
>> res = mult2nums(x, y);
>> x * y
```

How much is x * y ???
X retained its original value.

res	8
y	
x	4

res	8
y	2
x	1

Call by value

```
function printAnimals(a1, a2)

disp(['Before the swap function: ', a1, ' ', a2]);
swapAnimals(a1, a2);
disp(['After the swap function: ', a1, ' ', a2]);
```

```
function swapAnimals(a1, a2)

temp = a1;
a1 = a2;
a2 = temp;
disp(['Inside the swap function: ', a1, ' ', a2]);
```

```
>> printAnimals('frog', 'cow')
```

What does the program print?

Call by value

The content of the local copy of variables within the swap function is swapped.

The content of the variables outside the swap function is NOT swapped.

How can we cause the swap to persist after function call is terminated?

Call by value

```
function printAnimals(a1, a2)

disp(['Before the swap function: ', a1, ' ', a2]);
[a1, a2] = swapAnimals(a1, a2);
disp(['After the swap function: ', a1, ' ', a2]);
```

```
function [a1, a2] = swapAnimals(a1, a2)

temp = a1;
a1 = a2;
a2 = temp;
disp(['Inside the swap function: ', a1, ' ', a2]);
```

```
>> printAnimals('frog', 'cow')
```

Is there a difference in the output of the following two functions?

```
function func1()
i = 0;
while  (i < 5)
    increment(i);
    disp('Same same but different…');
end

function increment(i)
i = i + 1;
```

Endless loop

```
function func2()
i = 0;
while (i < 5)
        disp('Same same but different…');
        i = i + 1;
end
```

DNA Palindrome Detection

DNA Palindrome Detection

- What are palindromes?

Too bad I	h	id a boot

Rats live on	no evil star

- Palindromes appear in the DNA/RNA sequences. This is often indicative of functionality (e.g. RNA secondary structure, miRNA, Transcription binding sites etc.)

G A A A A A	T T T T T C

G A A A A A	C	T T T T T C

C C A G G G	C	C C C T G G

DNA Palindrome Detection

- Build a program that identifies all the palindromes of size 11 bases in the following sequence:

```
seq=
   'CCTGATCATCCACGGGGGACCCCCCAGTCGTACGTAGCTTGGGGG
   CCCCCAGCATACGTACGATCGCCCACAGTGGGAAAAAAAAAAAAAA
   AAAAAAAAAAAAAAAAAAAACATGCTACGATCGATCGATCGATCGAT
   CGATCGATCGATCGTAGTCAGTATTTGTGTGCAGCATGCACAGTCG
   ATCGATCGTGTCGTACCCCCCCCCTCCCCCCCCCCTCCCCCCCTAC
   GTAGCTACGTACGATCGATCG';
```

- Print these palindromes and their sequence positions.

DNA Palindrome Detection

```matlab
for ii = 6:length(seq) - 5
    % Checking if subsequence is a palindrome
    n_match = 0;
    for i_p = 1:5
        if seq(ii - i_p) == 'A' && seq(ii + i_p) == 'T' || ...
            seq(ii - i_p) == 'T' && seq(ii + i_p) == 'A' || ...
            seq(ii - i_p) == 'C' && seq(ii + i_p) == 'G' || ...
            seq(ii - i_p) == 'G' && seq(ii + i_p) == 'C'
            n_match = n_match + 1;
        else
            break;
        end
    end

    % If palindrome then display its sequence and position
    if n_match == 5
        pal = seq(ii - 5:ii + 5);
        disp(pal)
        disp([ii - 5, ii + 5])
    end
end
```

DNA Palindrome Detection

Results:

```
        GGGGGACCCCC
            14        24


        CCCACAGTGGG
            67        77
```

DNA Palindrome Detection

- How can we change the previous program to:
  - Find all palindromes of sizes ranging between 7 and 21?

- How expensive (in terms of computer operations) is it to compute all this?

DNA Palindrome Detection

```matlab
tic
for m = 7:2:21
    for ii = (m+1)/2:length(seq) - (m-1)/2
        % Checking if subsequence is a palindrome
        n_match = 0;
        for i_p = (m-1)/2
            if seq(ii - i_p) == 'A' && seq(ii + i_p) == 'T' || ...
                seq(ii - i_p) == 'T' && seq(ii + i_p) == 'A' || ...
                seq(ii - i_p) == 'C' && seq(ii + i_p) == 'G' || ...
                seq(ii - i_p) == 'G' && seq(ii + i_p) == 'C'
                n_match = n_match + 1;
            else
                break;
            end
        end
        % If palindrome then display its sequence and position
        if n_match == (m-1)/2
            pal = seq(ii - (m-1)/2:ii + (m-1)/2);
            disp(pal)
            disp([ii - (m-1)/2, ii + (m-1)/2])
        end
    end
end
toc
```
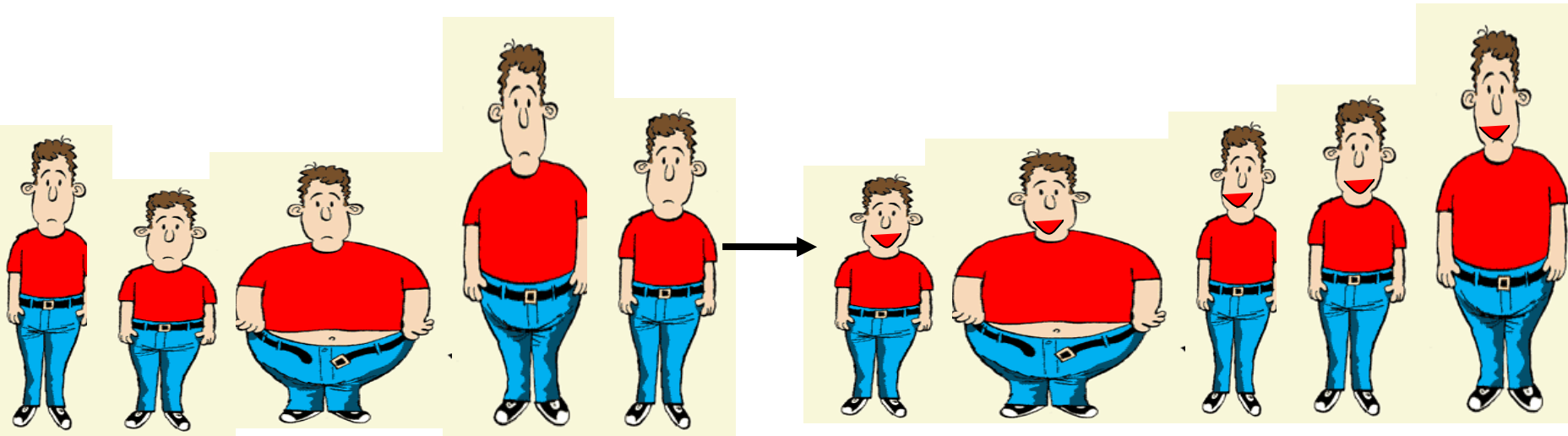
DNA Palindrome Detection

RESULTS

```
GATCATC
      4       10

TGCAGCA
      167      173

GGGGGCCCC
      41       49

GGGGCCCCC
      42       50

GGGGGACCCCC
      14       24

CCCACAGTGGG
      67       77
```

Sorting Algorithms

Bubble Sort

Sorting

- Sorting is a very basic operation which is performed routinely. Therefore, we want to find an efficient sorting algorithm.
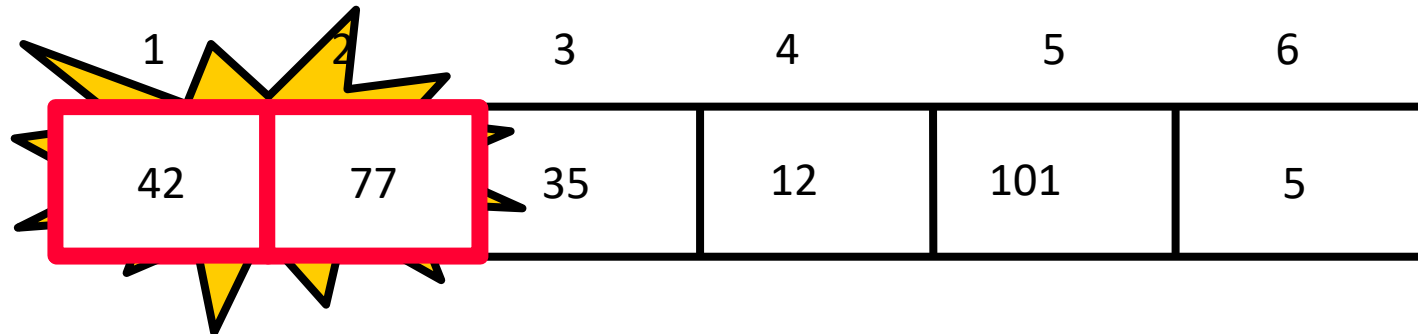
Bubble sort

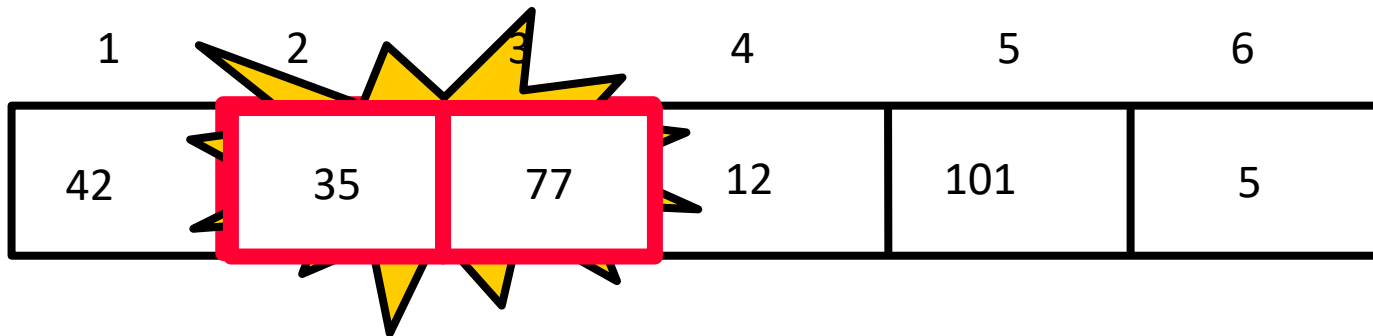| 1	2	3	4	5	6
77	42	35	12	101	5

Bubble sort

- Bubble sort

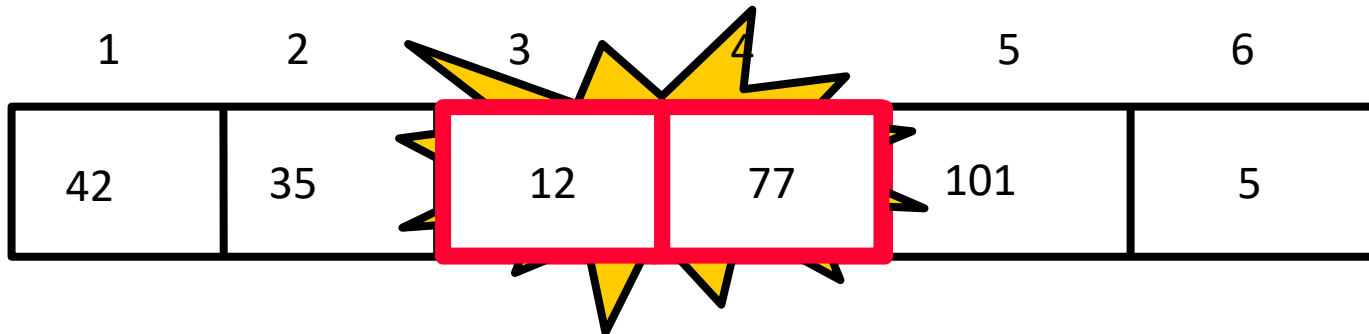| 1	2	3	4	5	6
42	77	35	12	101	5

Bubble sort

- Bubble sort

1	2	3	4	5	6
42	35	77	12	101	5

Bubble sort

- Bubble sort

1	2	3	4	5	6
42	35	12	77	101	5

Bubble sort

- Bubble sort

	1	2	3	4	5	6
	42	35	12	77	101	5
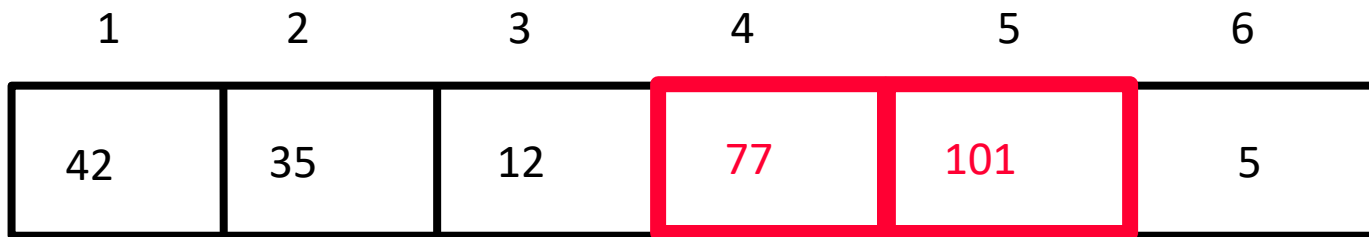
There is no need to swap

Bubble sort

- Bubble sort

1	2	3	4	5	6
42	35	12	77	5	101

Bubble sort
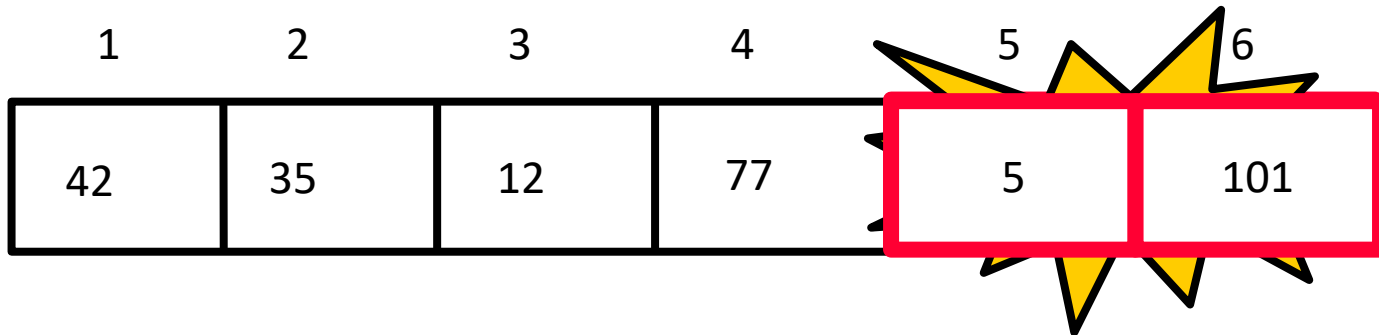
- Bubble sort

1	2	3	4	5	6
42	35	12	77	5	101
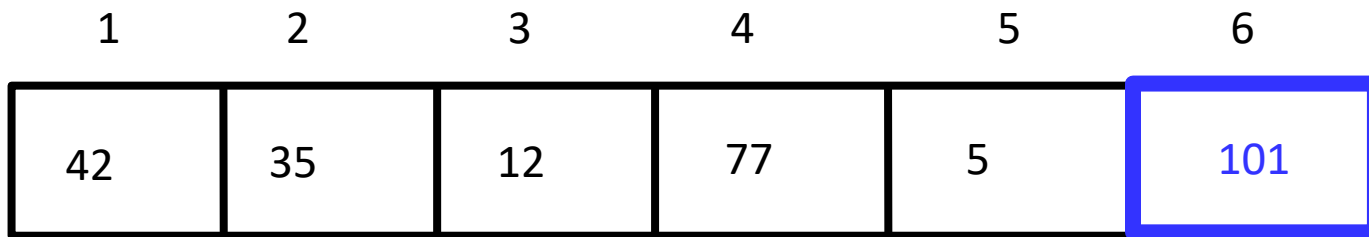
Largest value correctly placed

Bubble sort

	1	2	3	4	5	6

42	35	12	77	5	101

35	12	42	5	77	101

12	35	5	42	77	101

12	5	35	42	77	101

5	12	35	42	77	101

Array is sorted !!!

Bubble sort


```
function array = bubbleSort(array)


for i = (length(array) - 1) : -1 : 1
    for j = 1 : i
        if array(j) > array(j+1)
            %swap
            temp        = array(j);
            array(j)    = array(j + 1);
            array(j + 1) = temp;
        end
    end
    disp(array)
end
```

Bubble sort


```
>> array = randperm(10)
array =
     5     6     9     1     4     2    10     8     3     7

>> bubbleSort(array)

5     6     1     4     2     9     8     3     7    10
5     1     4     2     6     8     3     7     9    10
1     4     2     5     6     3     7     8     9    10
1     2     4     5     3     6     7     8     9    10
1     2     4     3     5     6     7     8     9    10
1     2     3     4     5     6     7     8     9    10
1     2     3     4     5     6     7     8     9    10
1     2     3     4     5     6     7     8     9    10
1     2     3     4     5     6     7     8     9    10
```

Bubble sort

- ## What is the time complexity?

  - **time complexity** - The way in which the number of steps required by an algorithm varies with the size of the problem it is solving. Time complexity is normally expressed as an order of magnitude, e.g. O(N^2) means that if the size of the problem (N) doubles then the algorithm will take four times as many steps to complete.

- ## What is the space complexity?

  - **space complexity** - The way in which the amount of storage space required by an algorithm varies with the size of the problem it is solving. Space complexity is normally expressed as an order of magnitude, e.g. O(N^2) means that if the size of the problem (N) doubles then four times as much working storage will be needed.

- ## Can we do better?

Complexity Analysis of Bubble Sort

In Bubble Sort, `n-1` comparisons will be done in the 1st pass, `n-2` in 2nd pass, `n-3` in 3rd pass and so on. So the total number of comparisons will be,

```
OUTPUT:
(n-1) + (n-2) + (n-3) + ..... + 3 + 2 + 1
Sum = n(n-1)/2
i.e O(n²)
```

Hence the **time complexity** of Bubble Sort is **O(n^2)**.

The main advantage of Bubble Sort is the simplicity of the algorithm.

The **space complexity** for Bubble Sort is **O(1)**, because only a single additional memory space is required i.e. for `temp` variable.

Also, the **best case time complexity** will be **O(n)**, it is when the list is already sorted.

Following are the Time and Space complexity for the Bubble Sort algorithm.

- Worst Case Time Complexity [Big-O]: **O(n^2)**
- Best Case Time Complexity [Big-omega]: **O(n)**
- Average Time Complexity [Big-theta]: **O(n^2)**
- Space Complexity: **O(1)**