Islamic University of Technology (IUT)

Organization of Islamic Cooperation (OIC)

Department of Electrical and Electronic Engineering (EEE)

COURSE NO	**:**	**EEE 4416**
LAB NO	**:**	**10**
TOPIC	**:**	**Algorithms**

Introduction to Algorithms

So far, you have learned the **syntax** of the MATLAB programming language, many of the built-in functions that MATLAB offers, and applied your knowledge to solve different problems. Understanding the syntax is the first step towards learning a programming language. The syntax varies from language to language, but the approach may remain the same. You will find a large amount of similarity between MATLAB and Python's 'Numpy' library, as they are designed in a similar fashion.

However, just knowing the syntax is not always enough. In real-world applications, many such problems appear where **a straightforward approach may not always work**. Furthermore, it could take a **long time** to execute (high time complexity). You have already learned some important techniques, such as vectorization or Boolean masking, which can help avoid loops and significantly reduce the runtime of your program. However, there are many other problems where more intelligent approaches may become necessary.

For instance, look at the following example –

You are given a grid. Each cell of the grid has a value associated with it. You start from the upper left corner, and you need to get to the bottom right. You can move only to adjacent cells, either by sliding right, up, or down. Your goal is to find the path that makes the sum of the values you've passed the lowest.

Give it a try.

1	**12**	**4**	**6**	**8**	**10**	**100**
1	**5**	**7**	**87**	**98**	**2**	**200**
20	**56**	**74**	**1**	**34**	**56**	**21**

In this small grid, you can see - there are many possible paths available. For example,

- 1=> 1=> 5=> 7=> 87=> 1=> 34 => 56=> 21 (total cost – 213, steps = 9)

This is a possible path. But is it the optimal one with the lowest cost? What about the following one -

- 1=> 1=> 5=> 7=> 4=>6=>8=>10=>2=>56=>21 (total cost – 121, steps = 11)

As you can see, this path, although lengthier, has a lower cost than the previous one. But there is a catch – you cannot be sure this is the optimal path until you try all possible combinations.

To find the optimal path, you need to check each and every one of them, which is quite a time-consuming task. If the size of the **grid is larger**, say, 1000*1000, imagine how many hundreds and thousands of paths are possible, and checking each path one by one does not seem like a proper approach. This type of approach is called '**brute-forcing**', and it is usually impossible to obtain the desired result this way, even with a powerful processor.

To deal with such problems, there are other intelligent approaches that can solve the problem in real time. To solve different problems, many different algorithms have been developed. Different algorithms are suitable for different tasks. Such algorithms can solve even a seemingly impossible task in linear time.

Some of the most popular algorithms are mentioned below –

	Algorithm	**Types/Examples**
i.	Divide and Conquer	
ii.	Searching algorithms	Linear search, binary search, jump search, adversary search, etc.
iii.	Sorting algorithms	Quick sort, merge sort, bubble sort, heap sort, insertion sort, etc.
iv.	Graph Algorithms	Breadth-first search (BFS), Depth-first search (DFS), Dijkstra's Algorithm, Kruskal's Algorithm, Minimum spanning tree (MST), etc.
v.	Dynamic Programming (DP)	Longest Common Subsequence (LCS), Knapsack Problem
vi.	Greedy Algorithms	Activity Selection, Huffman Coding, Fractional Knapsack
vii.	Backtracking Algorithms	N-Queens Problem, Sudoku Solver
viii.	Computational Geometry Algorithms	Convex Hull, Sweep Line Algorithm

The above problem in the example can be solved very easily using graph algorithms, in particular Dijkstra's shortest path approach.

In today's lab, we will explore some of the commonly used algorithms and try to understand their use cases where they are needed. Algorithms are a vast concept, so the goal is not to go deeper, but to introduce their applications.

Search Algorithms

In the earlier labs, you have frequently used the **'find'** function to extract the index of a particular element in an array. How does the function work? While built-in functions in MATLAB make our tasks easier but it is also necessary to understand **how those built-in functions actually work**. These functions employ different algorithms and techniques to get the job done.

The purpose of search algorithms is to locate or retrieve an element from data. The data can be stored in any kind of data structure, like a list or a dictionary. These searching algorithms can be divided into two categories –

i. **Sequential Search:** The sequential search algorithms traverse the list sequentially to find the desired value.

ii. **Interval Search:** The interval search algorithms work for sorted data. Instead of checking every element in the list like sequential search, interval search algorithms look for the desired element heuristically. It is based on the 'divide and conquer' algorithm.

Different data types, such as time series data when it is recorded, are already sorted with respect to time, so there is no need to sort them externally before applying interval search algorithms.

Different searching algorithms –

- Linear Search
- Binary Search
- Jump Search
- Interpolation Search
- Exponential Search
- Breadth First Search (BFS)

Linear Search

In this algorithm, you search each element in the list sequentially to find the desired value. You start with the first element and continue until a match is found. The time complexity of the algorithm is: **O(n)**.

- Best-case scenario: O(1) => 1st element is the desired one
- Worst-case scenario: O(n) => value not in the array. You need to search the whole array (n elements)

It is not the most efficient approach when the sample size is very large. If you have 1 million samples in the data, you have to go through a large number of samples to find the desired element.

MATLAB also provides some built-in functions to allow you to search for an item in an array.

- find (a = = val) will return the indices where the element is located.
- ismember (val, a) will return logical true/false whether the value exists in the array or not.

- Let's write a code to check whether an element exists in an array or not using the linear search algorithm.
- Class task: Find the index values as well.

Asif Newaz
Lecturer, EEE, IUT

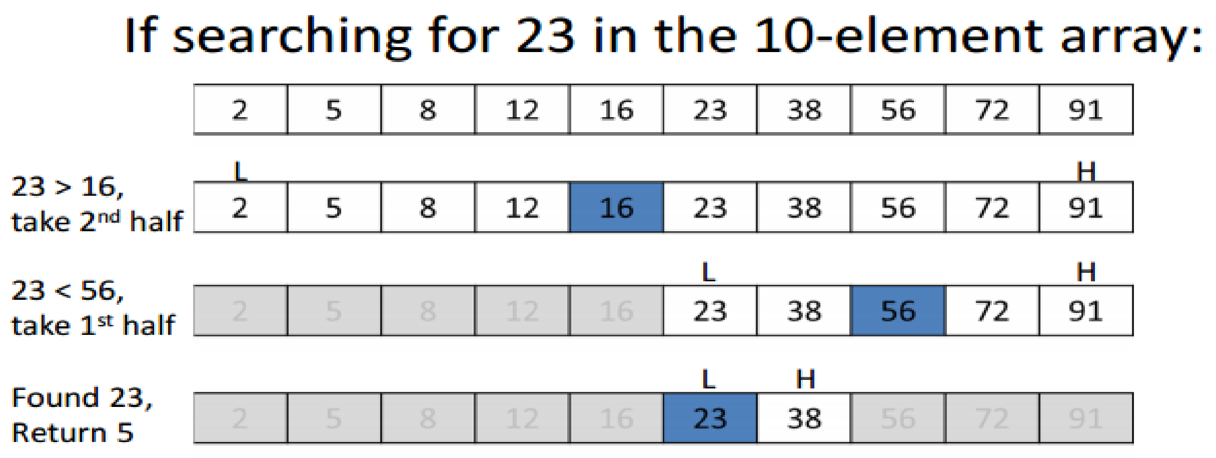
```
% Linear search

tic

a= randi(100000,1,1000000);  % creating a random 1D array with 1 million parameters
val= 1200000;                % try different search values

for i= 1:length(a)
    if a(i)==val
        out='value found';
        break                    % break out of the loop to stop searching
    else
        out='value not found';
    end
end
out

toc
```

Binary Search

This is an interval search technique. It only works in a sorted array. So, the original list needs to be arranged in an ascending order (numeric) or dictionary order (string) if not originally sorted. This method works based on the divide and conquer technique to find the desired value with less time complexity. In the case of large data, the number of iterations required to find the value is significantly reduced in binary search compared to a sequential search algorithm.

In binary search, we start with the middle element in the sorted array. If that is the desired element, then the search is complete with minimum steps (best-case). If not, then we find whether the desired value is **smaller or larger than the middle element.** Based on that, we narrow the search to only half of the original array. Then the middle element of the new array (size = n/2) is found, and the process is repeated. This way, with each step, the size of the array is halved, and the total number of iterations required to find the value is reduced.

The time complexity of the algorithm is: **O(log(n))**, which makes it much more efficient than linear search.



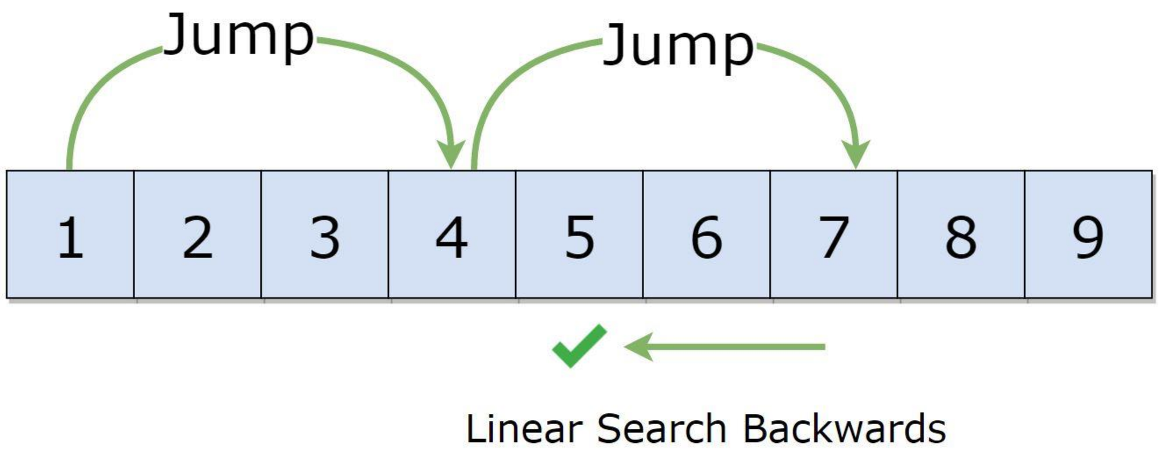
Algorithm

i. Start by assigning an index value → low = 1, high = number of elements in the array.
ii. Find the mid element index of the array using **mid = (low+high)/2**. Take the floor value in case of a fraction.
iii. Check whether the array(mid) is lower or higher than the item we are searching for.
iv. If array(mid) = search_item, then the item is found.
v. If array(mid) < search_item, then the item exists in the upper half. So, we can narrow the search in the upper half portion only **by moving low to mid+1.**
vi. If array(mid) > search_item, then the item exists in the lower half. So, we can narrow the search in the lower half portion of the array only by **moving high to mid-1**.
vii. Repeat steps 2 to 5 until the element is found. Otherwise, the item is not on the list.

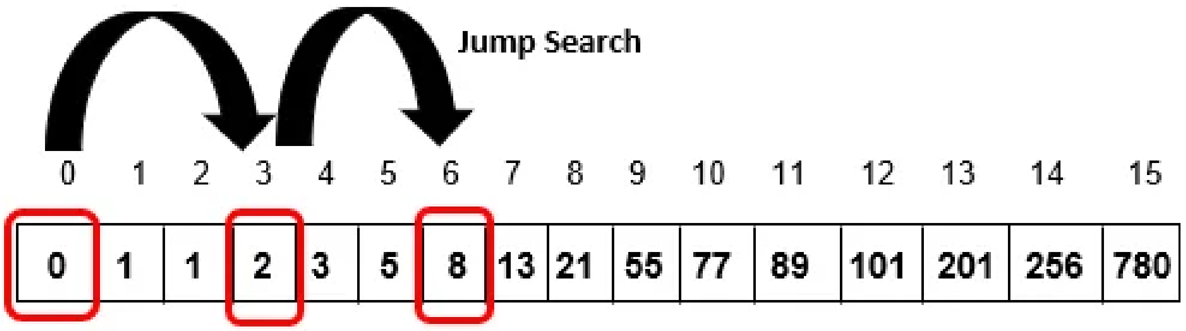
```
% binary search

rng(1,'twister');   % generating same numbers on each run
s=rng;              % for more info - https://www.mathworks.com/help/matlab/math/controlling-
random-number-generation.html

a= randi(100000,1,1000000);  % creating a random 1D array with 1 million parameters
a= sort(a,'a');              % binary search only works on sorted array
%a=[1,2,3,3,4,4,5,5,10,20];
val= 1031;

low=1;
high= length(a);
mid= floor((low+high)/2);

while low<=high
    if a(mid)<val
        low= mid+1;
    elseif a(mid)>val
        high= mid-1;
    else
        out= 'value is found';
        val_index= mid;
        break;
    end

    mid= floor((low+high)/2);
end

if low>high
    out= 'value not found';
    val_index=nan;
end

out
val_index
```

Jump Search

Jump search is an **improvement over linear search, but less efficient than binary search**. Unlike linear search, it works only on a sorted array. Instead of searching through all elements sequentially, it jumps a few steps/skips a few elements to reduce the number of iterations required.



In the figure, a jump step of 3 is utilized. So, the method checks the index – 1, 4, 7, and so on. The step size is taken depending on the size of the original array. If the array is large, then a higher jump value is preferred.



Say, the element we are looking for is 77. Using jump search, we will take four jumps first and reach index 12, and find the value larger than the search key. Then we will **go backwards** and perform a linear search operation (two steps required) to find the desired value. As can be seen, the number of iterations required to find the value is lower than linear search.

The time complexity of the algorithm is: O(√n)
The time complexity is between linear search and binary search.

Sorting algorithms

Sorting involves arranging data in a particular order, typically in ascending or descending sequence. There are different sorting algorithms. They vary in terms of efficiency, complexity, stability, and the type of data they handle best. Some algorithms are simple and easy to implement, like Bubble Sort or Selection Sort, making them ideal for small datasets and educational purposes. Others, such as Merge Sort, Quick Sort, and Heap Sort, are designed for high performance and are more suitable for large-scale applications.

MATLAB/Python provides a built-in function for sorting arrays. Internally, this function implements different sorting algorithms (merged) depending on the size of the array. For example, MATLAB usually uses Introsort, which is a combination of QuickSort, HeapSort, and InsertionSort.

In this lab, let us try to understand and implement one simple sorting technique: Bubble sort.

Bubble Sort

Given an array: [5, 1, 4, 2, 8]

We repeatedly compare each pair of adjacent elements and **swap them if they're in the wrong order**.

Pass 1:

Compare each pair:

- 5 > 1 → swap → [1, 5, 4, 2, 8]
- 5 > 4 → swap → [1, 4, 5, 2, 8]
- 5 > 2 → swap → [1, 4, 2, 5, 8]
- 5 < 8 → no swap → [1, 4, 2, 5, 8]

Pass 2:

- 1 < 4 → no swap
- 4 > 2 → swap → [1, 2, 4, 5, 8]
- 4 < 5 → no swap
- 5 < 8 → no swap

Pass 3:

- 1 < 2 → no swap
- 2 < 4 → no swap
- 4 < 5 → no swap

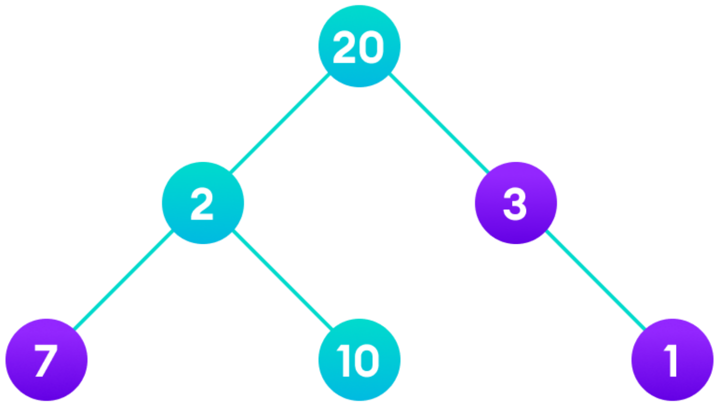
No swaps made → array is sorted.

```
function sorted_array = bubble_sort(arr)
  n = length(arr);       % Get the number of elements in the array
  sorted_array = arr;    % Make a copy to avoid modifying the input

  for i = 1:n-1          % Outer loop for each pass
    swapped = false;     % Flag to detect if any swap occurred

    for j = 1:n-i        % Inner loop: compare adjacent elements
      if sorted_array(j) > sorted_array(j+1)
        % Swap elements if they're in the wrong order
        temp = sorted_array(j);
        sorted_array(j) = sorted_array(j+1);
        sorted_array(j+1) = temp;
        swapped = true;
      end
    end

    % If no swaps were made in this pass, array is already sorted
    if ~swapped
      break;
    end
  end
```

Greedy Algorithm

A greedy algorithm is an approach for solving a problem by selecting the *'best option' available at the moment*. It doesn't worry about whether the current best result will bring the overall optimal result. It always goes for the **local best choice** to produce the global best result. Therefore, the obtained solution **may not be optimal**. However, it reaches the solution much quicker than the brute force algorithm.

- ❖ Let's look at an example –



Here, you want to find the best way (maximum gain) to traverse through the nodes. Your starting point or root node has a weight of 20. From this position, you have only two ways to go. And the greedy choice is - always go for the local best. So, you move to the node with weight 3. Then, you have only one other node to go. This way, the total weight you gained is 24. However, if you had gone the other way, you could have gained a much higher weight of 32. Since the greedy algorithm only works in the **top-down approach** and **never reverses its decision**, it failed to provide the optimal solution.

$$20+2 = 22 \Rightarrow 22+10 = 32$$

$$20+3 = 23 \Rightarrow 23+1 = 24$$

In practice, when the search space is very large, a greedy choice is usually capable of providing a **sub-optimal solution** with much shorter time complexity, making it attractive for many applications.

- ✓ When applying greedy algorithms, you have to make a **greedy choice**. Depending on the scenario, you have to identify which choice is the optimal one.

There are many practical problems where greedy algorithms are applied. Some popular applications of the greedy algorithm include –

- o Job sequencing problem
- o Fractional knapsack problem
- o Minimum Spanning Tree (MST)
- o Huffman Coding
- o Sequential feature selection (SFS)

Let’s look at a couple of examples here.

Money Change Problem

The task is to find the **minimum number of coins/notes required** to make up a given amount of money.

In our currency, we have notes of 1000, 500, 200, 100, 50, 20, 10, and 5. Say, in the ATM booth, you want to withdraw a certain amount of money. You have to write a program such that the ATM machine can provide the desired sum with the minimum number of notes.

For instance, if you want to withdraw 1200 taka – the atm machine can give you twelve 100-taka notes or six 200-taka notes, or several other ways are possible. The optimal way would be to provide one 1000 taka note along with one 200 taka note.

You have to make a greedy choice to solve the problem – always choose the highest valued note as many as possible.

Test Case - 01

- Input: 1200
- Output: 2 (1000, 200)

Test Case - 02

- Input: 4600
- Output: 6 (4*1000, 500, 100)

Test Case - 03

- Input: 1900
- Output: 4 (1000, 500, 2*200)

The greedy way to solve the problem works for the canonical coin system. However, in an arbitrary coin system, the approach would fail. For instance,

Coins = [1, 2, 3, 4, 5]

- Desired sum = 8
- Greedy choice = 3 (5+3+1)
- Optimal solution = 2 (4+4)

In scenarios like this, you need to consider another popular algorithm – “Dynamic Programming".

```
function [out, val]= money_change(x,n)

xs= sort(x,'d');    % sort in decreasing order

lx= length(x);

out=0;          % holds the number of coins

val=[];         % holds the values


for i=1:lx

  a=xs(i);          % highest valued coin

  b= floor(n/a);              % number of possible notes

  n= rem(n,a);                % remaining sum


  out= out+b;                 % increment the number of coins


  if b~=0

    val=[val,a];              % hold the output coins

  end


  if n==0

    return      % return when sum is complete; no need to run the whole loop

  end

end


end
```

Job Sequencing Problem

You are given a list of jobs and their associated profits, along with their deadline. Each job takes a single day to complete. Your task is to schedule the jobs in such a way that maximizes the profit.

Example - 01

Jobs	a	b	c	d
Deadline	4	1	1	1
Profit	20	10	40	30

Optimal solution: c, a.

Example - 02

Jobs	a	b	c	d	e
Deadline	2	1	2	1	3
Profit	100	19	27	25	15

Optimal solution: c, a, e

Example - 03

Jobs	a	b	c	d	e	f
Deadline	5	3	3	2	4	2
Profit	200	180	190	300	120	100

Optimal solution: b, d, c, e, a

In order to solve problems like these, you need to come up with an algorithm that fulfils the objective.

Objective – maximize the profit.

To realize the objective, you need to focus on the 'profit' parameter, not the deadline. You need to complete the jobs in such a way that profit can be maximized. It need not consider how many tasks you can complete.

For instance, in example 2, for both jobs ‘b’ and ‘d’, day – 1 is the deadline. But the profit from those jobs is small. Job ‘c’ provides comparatively higher profit than ‘b’ and ‘d’ but lower than ‘a’. Job ‘a’ provides the highest profit, and it can be completed by day 2. If you take this job on the first day, then you lose the opportunity to undertake the other jobs that can only be completed on day 1 and provide some profits.

So, you can see, you need to consider several associated parameters and come up with an algorithm that takes care of all these contradictory factors.

In scenarios like these, you can make what is called a ‘greedy choice’.

Approach

I. Create an empty set (array) with a **length = the maximum deadline**. This set represents the jobs that we can complete. We can do a maximum of 3 jobs here.

out			

II. Sort the given matrix **based on the profit** in descending order.

Jobs	a	c	d	b	e
Deadline	2	2	1	1	3
Profit	100	27	25	19	15

III. Take one job at a time from the sorted matrix sequentially and place the jobs on the ‘out’ array. The jobs have to be **placed at the ‘last’ possible deadline.**

For instance, job ‘a’ can be done on day 1 or day 2. You are making the choice to do it on the **last day (greedy choice)**, keeping the first day available for another job (that will offer lower profit than a).

out		a	

out	c	a	

out	c	a	e

This is the way you will obtain the maximum profit. You can try all other possible combinations – and you will see, this is the optimal one.

Class work: Solve example 3 in a similar way.