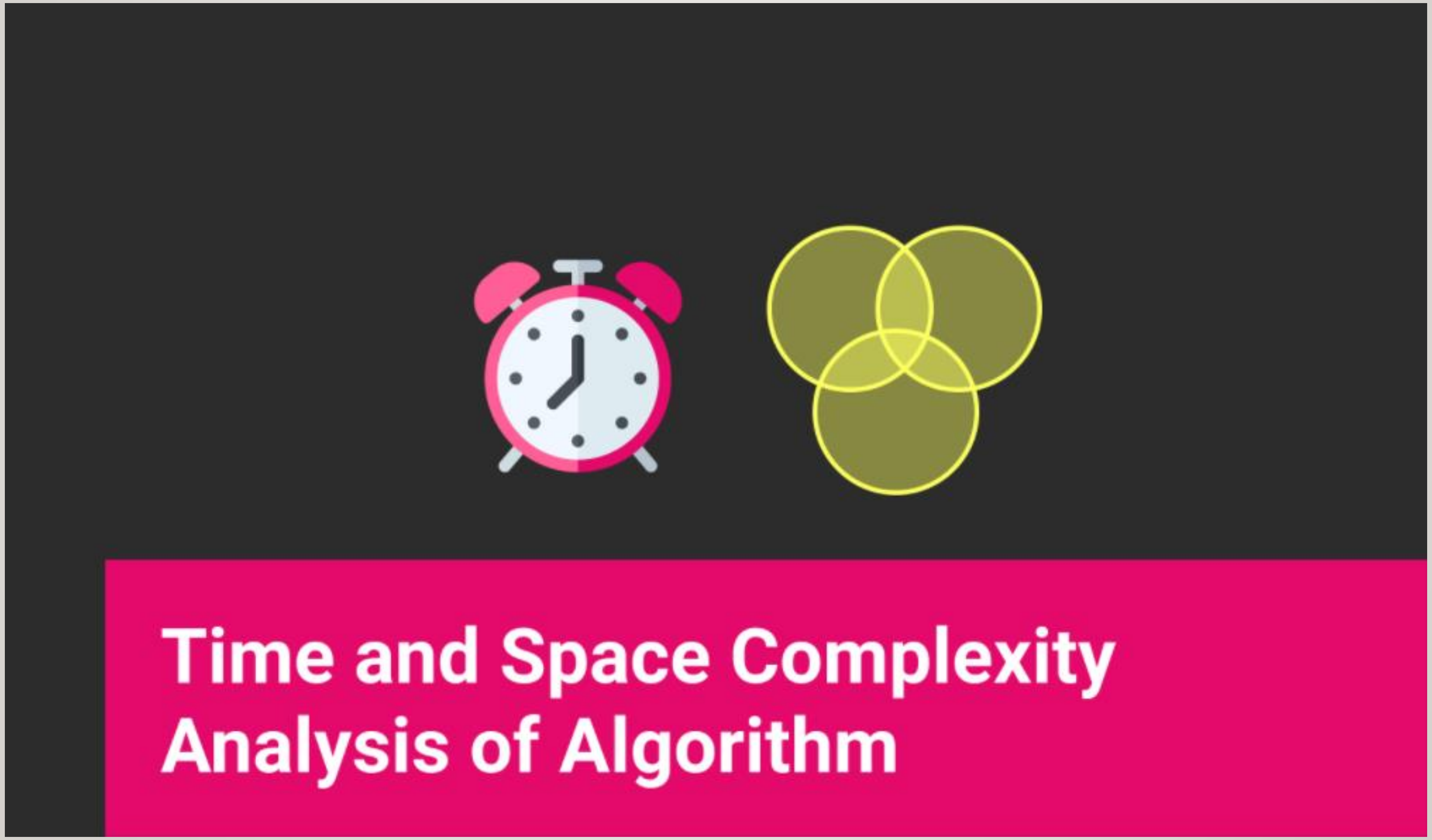
AMIT LEARNING

SESSION 7 ALGORITHMS

CONTENTS


Time and Space Complexity
Analysis of Algorithm

1 WHAT IS Algorithm

1.1 Algorithm analysis

- Time Complexity: is a concept in computer science that deals with the quantification of the amount of time taken by a set of code or algorithm to process or run as a function of the amount of input. In other words, the time complexity is how long a program takes to process a given input. The efficiency of an algorithm depends on two parameters:
  - Time Complexity.
  - Space Complexity.
- **Time Complexity:** It is defined as the number of times a particular instruction set is executed rather than the total time is taken. It is because the total time took also depends on some external factors like the compiler used, processor's speed, etc.
- **Space Complexity:** It is the total memory space required by the program for its execution.
- Descibe time complexity and space complexity by Asymptotic analysis
- Asymptotic analysis : are the mathematical notations that calculate, how the time (or space) taken by an algorithm increases with the input size. evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time).

- There are three asymptotic notations that are used to represent the time complexity of an algorithm. They are:

1. **Θ Notation (theta).**
2. **Big O Notation.**
3. **Ω Notation.**

- An algorithm can have different time for different inputs. It may take 1 second for some input and 10 seconds for some other input. So, this can be divided into three cases:

1. **Best case:** This is the lower bound on running time of an algorithm.

2. **Average case:** We calculate the running time for all possible inputs, sum all the calculated values and divide the sum by the total number of inputs.

3. **Worst case:** This is the upper bound on running time of an algorithm. We must know the case that causes the maximum number of operations to be executed.

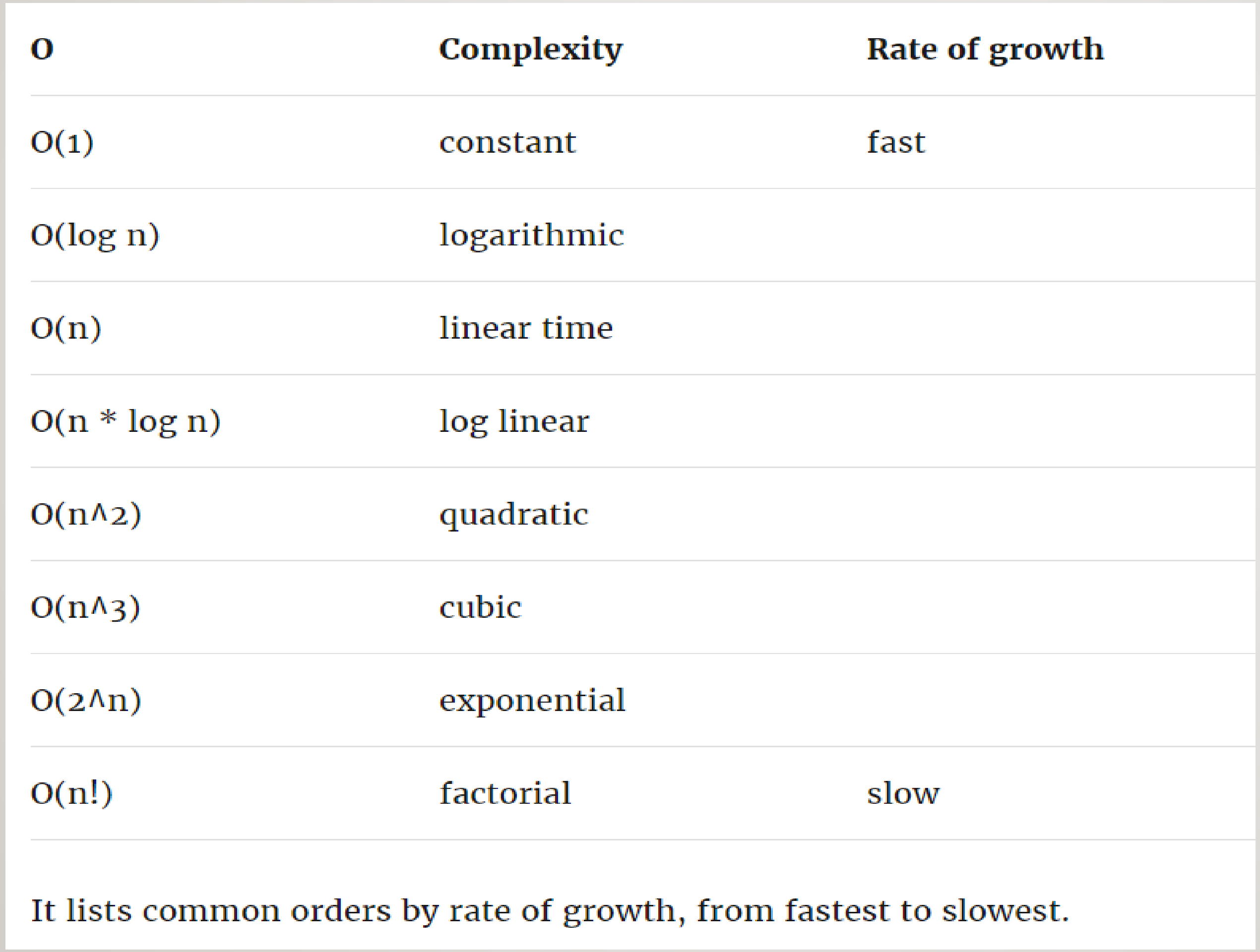
- **Θ Notation (theta):**
  - The Θ Notation is used to find the average bound of an algorithm i.e. it defines an upper bound and a lower bound, and your algorithm will lie in between these levels.
- **Big O Notation:**
  - The Big O notation defines the upper bound of any algorithm i.e. you algorithm can't take more time than this time. In other words, we can say that the big O notation denotes the maximum time taken by an algorithm or the worst-case time complexity of an algorithm. So, big O notation is the most used notation for the time complexity of an algorithm.
- **Ω Notation:**
  - The Ω notation denotes the lower bound of an algorithm i.e. the time taken by the algorithm can't be lower than this. In other words, this is the fastest time in which the algorithm will return a result

  - We can express algorithmic complexity using the big-O notation. For a problem of input size N:
    - ✓ A constant-time function/method is "order 1" : O(1)
    - ✓ A linear-time function/method is "order N" : O(N)
    - ✓ A quadratic-time function/method is "order N squared" : O(N 2)

1 WHAT IS Algorithm

1.1 Algorithm analysis

- We can express algorithmic complexity using the big-O notation. For a problem of input size N:



O	Complexity	Rate of growth
O(1)	constant	fast
O(log n)	logarithmic	
O(n)	linear time	
O(n * log n)	log linear	
O(n^2)	quadratic	
O(n^3)	cubic	
O(2^n)	exponential	
O(n!)	factorial	slow

It lists common orders by rate of growth, from fastest to slowest.

1 WHAT IS Algorithm

1.1 Algorithm analysis

CODE

```
#include <stdio.h>
int main()
{
    printf("Hello World");
}
```

OUTPUT

```
Hello World
```

DESCRIPTION

- So, time complexity is constant: O(1) i.e. every time constant amount of time require to execute code, no matter which operating system or which machine configurations you are using.

1.1 Algorithm analysis

CODE

```c
#include <stdio.h>
void main()
{
    int i, n = 8;
    for (i = 1; i <= n; i++) {
        printf("Hello Word !!!\n");
    }
}
```

OUTPUT

```
Hello Word !!!
Hello Word !!!
Hello Word !!!
Hello Word !!!
Hello Word !!!
Hello Word !!!
Hello Word !!!
Hello Word !!!
```

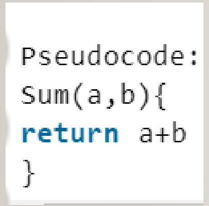
DESCRIPTION

- "Hello World!!!" will print N times. So, time complexity of above code is O(N).

1 WHAT IS Algorithm

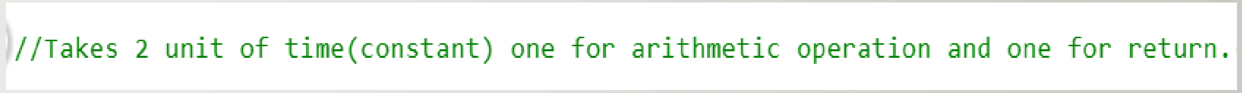
1.1 Algorithm analysis

CODE

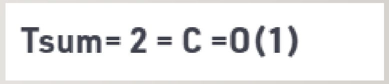


```
Pseudocode:
Sum(a,b){
return a+b
}
```

DESCRIPTION



```
//Takes 2 unit of time(constant) one for arithmetic operation and one for return.
```



Tsum= 2 = C =O(1)

1 WHAT IS Algorithm

1.1 Algorithm analysis

CODE

```
Pseudocode:
list_Sum(A,n){//A->array and n->number of elements in the array
total =0          // cost=1  no of times=1
for i=0 to n-1    // cost=2  no of times=n+1 (+1 for the end false condition)
sum = sum + A[i]  // cost=2  no of times=n
return sum        // cost=1  no of times=1
}
```

DESCRIPTION

$T_{sum} = 1 + 2 * (n+1) + 2 * n + 1 = 4n + 4 = C1 * n + C2 = O(n)$

1 WHAT IS Algorithm

1.1 Algorithm analysis

CODE

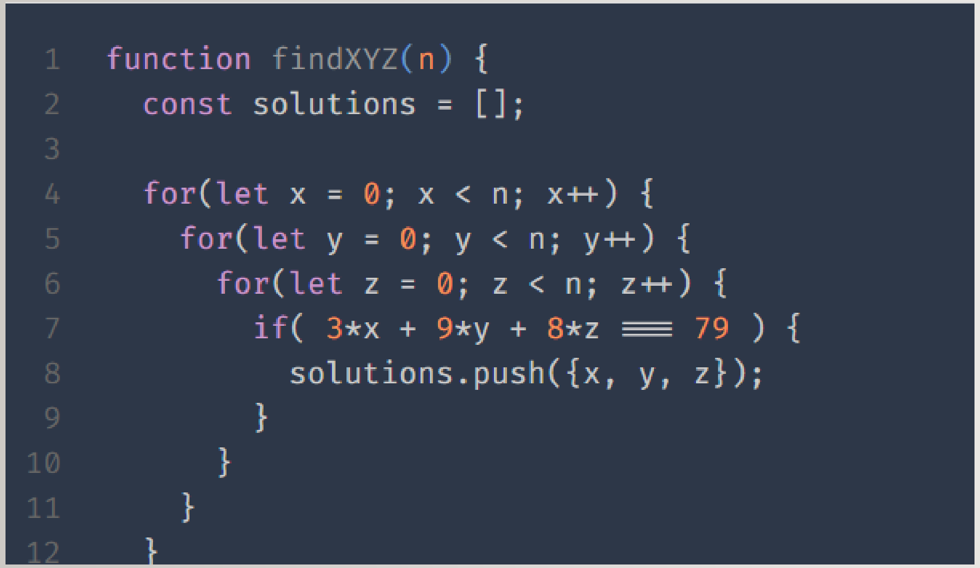
```
void printAllPossibleOrderedPairs(int arr[], int size)
{
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            printf("%d = %d\n", arr[i], arr[j]);
        }
     }
}
```

DESCRIPTION

- If n is true So time complexity will be o(n)2
- Here we're nesting two loops. If our array has n items, our outer loop runs n times and our inner loop runs n times for each iteration of the outer loop, giving us n2 total prints,
- Thus this function runs in O(n2) time (or "quadratic time"). If the array has 10 items, we have to print 100 times. If it has 1000 items, we have to print 1000000 times.

1.1 Algorithm analysis

CODE



```
function findXYZ(n) {
  const solutions = [];

  for(let x = 0; x < n; x++) {
    for(let y = 0; y < n; y++) {
      for(let z = 0; z < n; z++) {
        if( 3*x + 9*y + 8*z === 79 ) {
          solutions.push({x, y, z});
        }
      }
    }
  }
```

DESCRIPTION

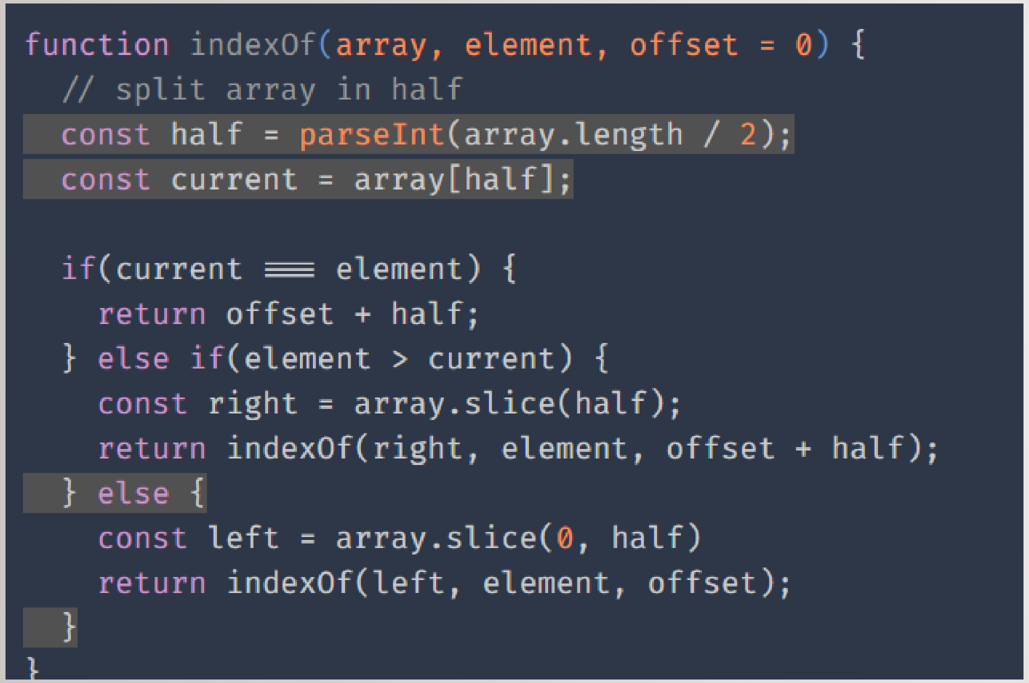
- Polynomial time, **Triple nested loop ,**
- This algorithm has a cubic running time: O(n^3).

WHAT IS ALGORITHM

1 WHAT IS Algorithm

1.1 Algorithm analysis

CODE



```javascript
function indexOf(array, element, offset = 0) {
  // split array in half
  const half = parseInt(array.length / 2);
  const current = array[half];

  if(current === element) {
    return offset + half;
  } else if(element > current) {
    const right = array.slice(half);
    return indexOf(right, element, offset + half);
  } else {
    const left = array.slice(0, half)
    return indexOf(left, element, offset);
  }
}
```

DESCRIPTION

- **Logarithmic time complexities usually apply to algorithms that divide problems in half every time.**
- the algorithm B split the problem in half on each iteration *O(log n)*.
- **O(log n) - Logarithmic time**
- **The runtime of the work done outside the recursion (line 3-4): O(1)**
- **How many recursive calls the problem is divided (line 11 or 14): 1 recursive call. Notice only one or the other will happen, never both.**
- **How much n is reduced on each recursive call (line 10 or 13): 1/2. Every recursive call cuts n in half.**

1.2 Types of Algorithm

- Types of Algorithm
  - Searching .
  - Sorting.
  - Searching:
    - Linear Search
    - Binary Search
  - Sorting:
    - Selection Sort
    - Bubble Sort
    - Insertion Sort

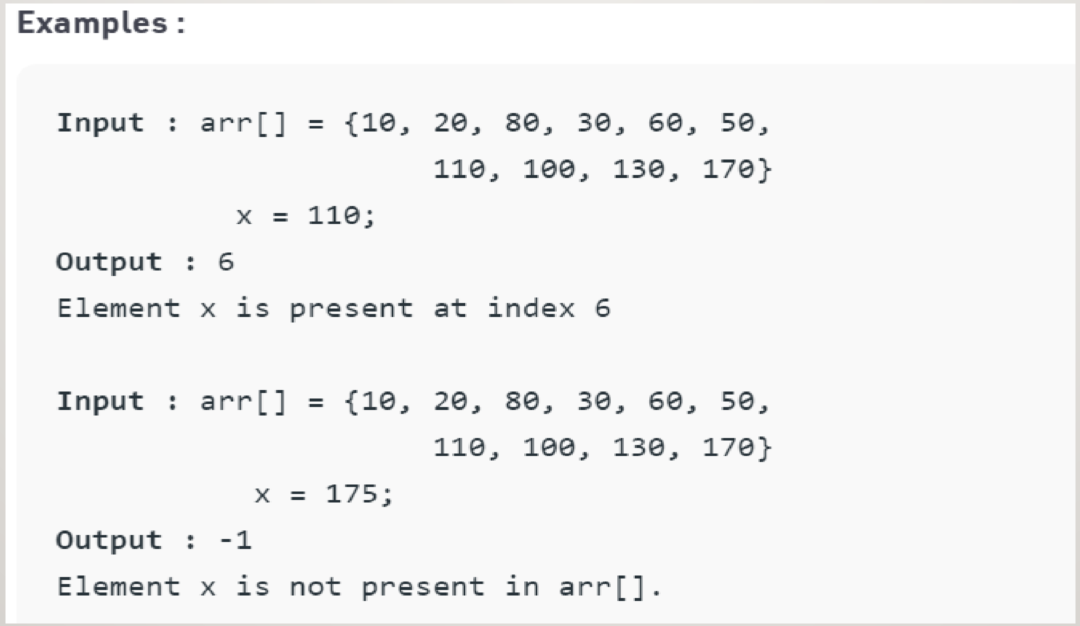
1 WHAT IS Algorithm

1.3 What is Searching Algorithm

- Searching:
  - Linear Search:

- **Problem:** Given an array arr[] of n elements, write a function to search a given element x in arr[].

- A simple approach is to do a **linear search**, i.e
  - Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
  - If x matches with an element, return the index.
  - If x doesn't match with any of elements, return -1.



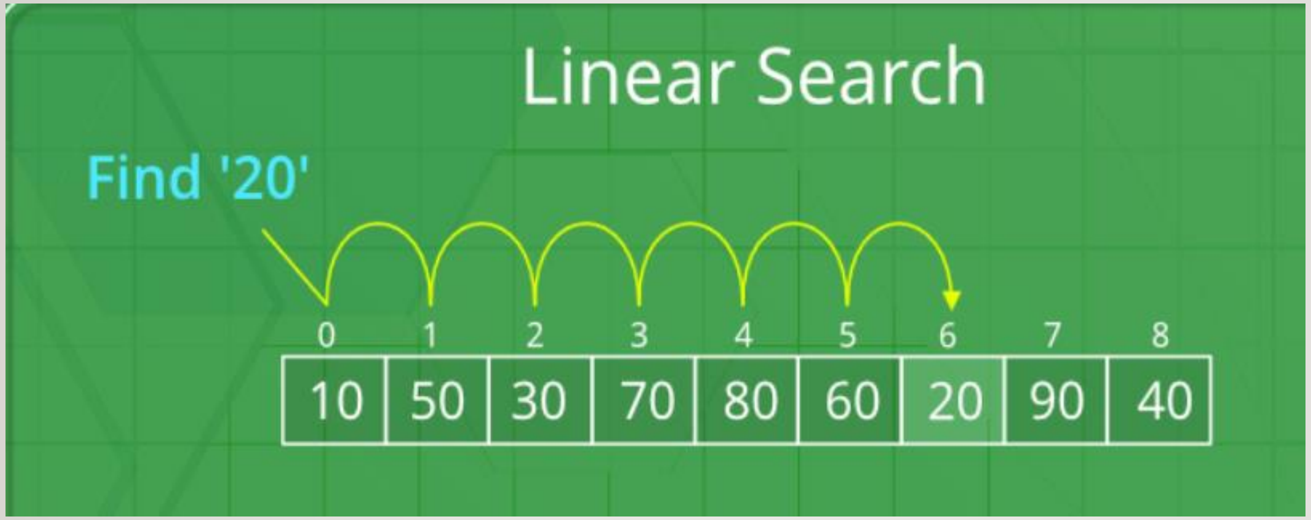
Examples :

```
Input : arr[] = {10, 20, 80, 30, 60, 50,
                     110, 100, 130, 170}
          x = 110;
Output : 6
Element x is present at index 6

Input : arr[] = {10, 20, 80, 30, 60, 50,
                     110, 100, 130, 170}
           x = 175;
Output : -1
Element x is not present in arr[].
```

1 WHAT IS Algorithm

1.4 Linear Search

- Searching:
  - Linear Search:



1 WHAT IS Algorithm

1.4 Linear Search

CODE

```
#include <stdio.h>

int search(int arr[], int n, int x)
{
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}

// Driver code
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int x = 10;
    int n = sizeof(arr) / sizeof(arr[0]);

    // Function call
    int result = search(arr, n, x);
    (result == -1)
        ? printf("Element is not present in array")
        : printf("Element is present at index %d", result);
    return 0;
}
```

OUTPUT

```
Element is present at index 3
```

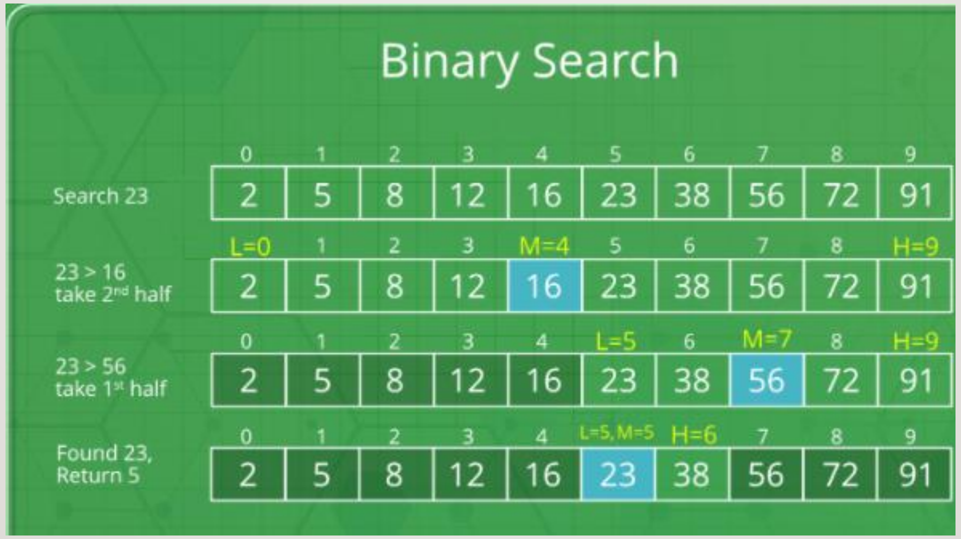
DESCRIPTION

- The time complexity of the above algorithm is O(n).
- Linear search is rarely used practically because other search algorithms such as the binary search algorithm more faster to found key and reduce the time

1 WHAT IS Algorithm

1.5 Binary search

- Searching:
  - **Binary** Search:
    - Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty



➢ Binary Searching:

- We basically ignore half of the elements just after one comparison.
- Compare x with the middle element.
- If x matches with middle element, we return the mid index.
- Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
- Else (x is smaller) recur for the left half.

1 WHAT IS Algorithm

1.5 Binary search

```c
int binarySearch(int arr[], int l, int r, int x)
{
    while (l <= r) {
        int m = l + (r - l) / 2;

        // Check if x is present at mid
        if (arr[m] == x)
            return m;

        // If x greater, ignore left half
        if (arr[m] < x)
            l = m + 1;

        // If x is smaller, ignore right half
        else
            r = m - 1;
    }

    // if we reach here, then element was
    // not present
    return -1;
}
```

```c
int main(void)
{
    int arr[] = { 2, 3, 4, 10, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);
    int x = 10;
    int result = binarySearch(arr, 0, n - 1, x);
    (result == -1) ? printf("Element is not present"
                            " in array")
                   : printf("Element is present at "
                            "index %d",
                            result);
    return 0;
}
```

OUTPUT

```
Element is present at index 3
```

DESCRIPTION

- The time complexity is O(log n) - Logarithmic time

1 WHAT IS Algorithm

1.5 what is sorting Algorithm

- what is sorting Algorithm:
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
  - Merge Sort
  - Heap Sort
  - QuickSort
  - Radix Sort
  - Counting Sort
  - Bucket Sort
  - ShellSort
  - Comb Sort
  - Pigeonhole Sort
  - Cycle Sort

1.6 Selection Sort

- what is Selection Sort Algorithm:

  - The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

1) The subarray which is already sorted.

2) Remaining subarray which is unsorted.

  - In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

1 WHAT IS Algorithm

1.6 Selection Sort

➢ what is Selection Sort Algorithm:

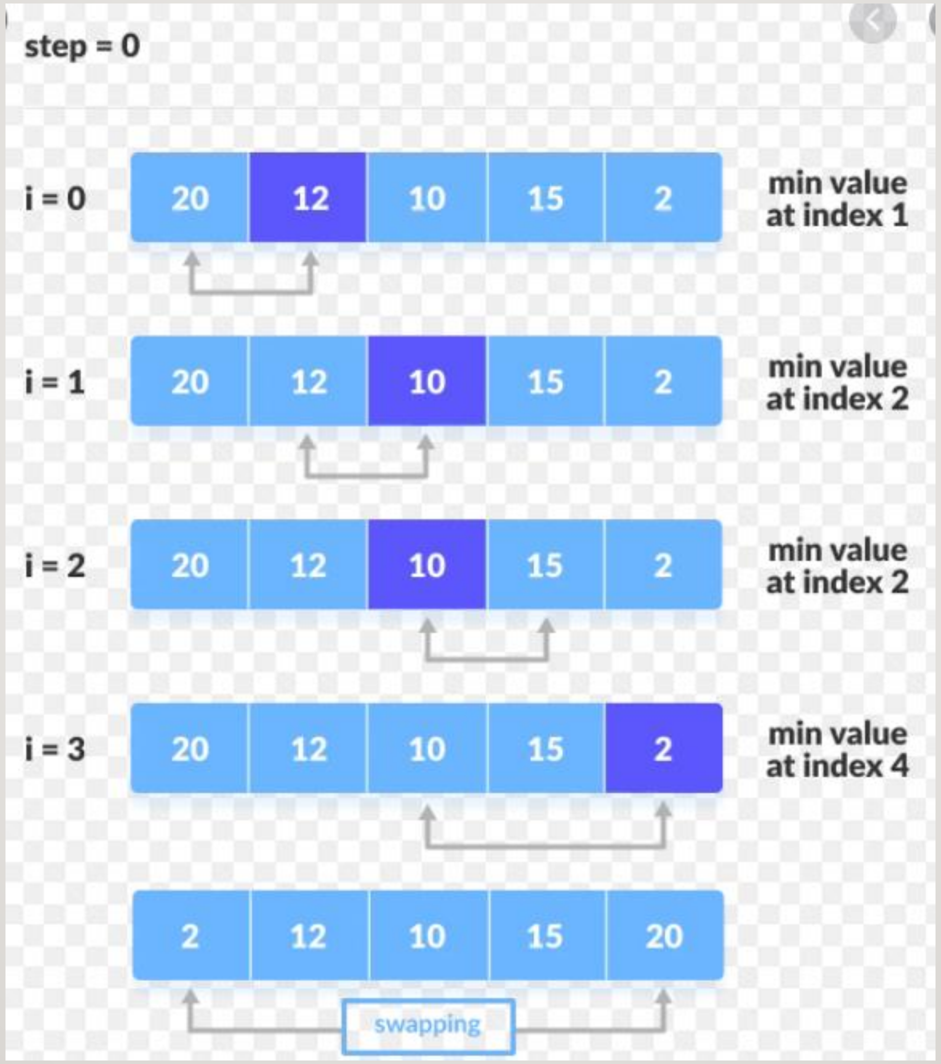
- **Following example explains the above steps**

```
arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

1 WHAT IS Algorithm

1.6 Selection Sort

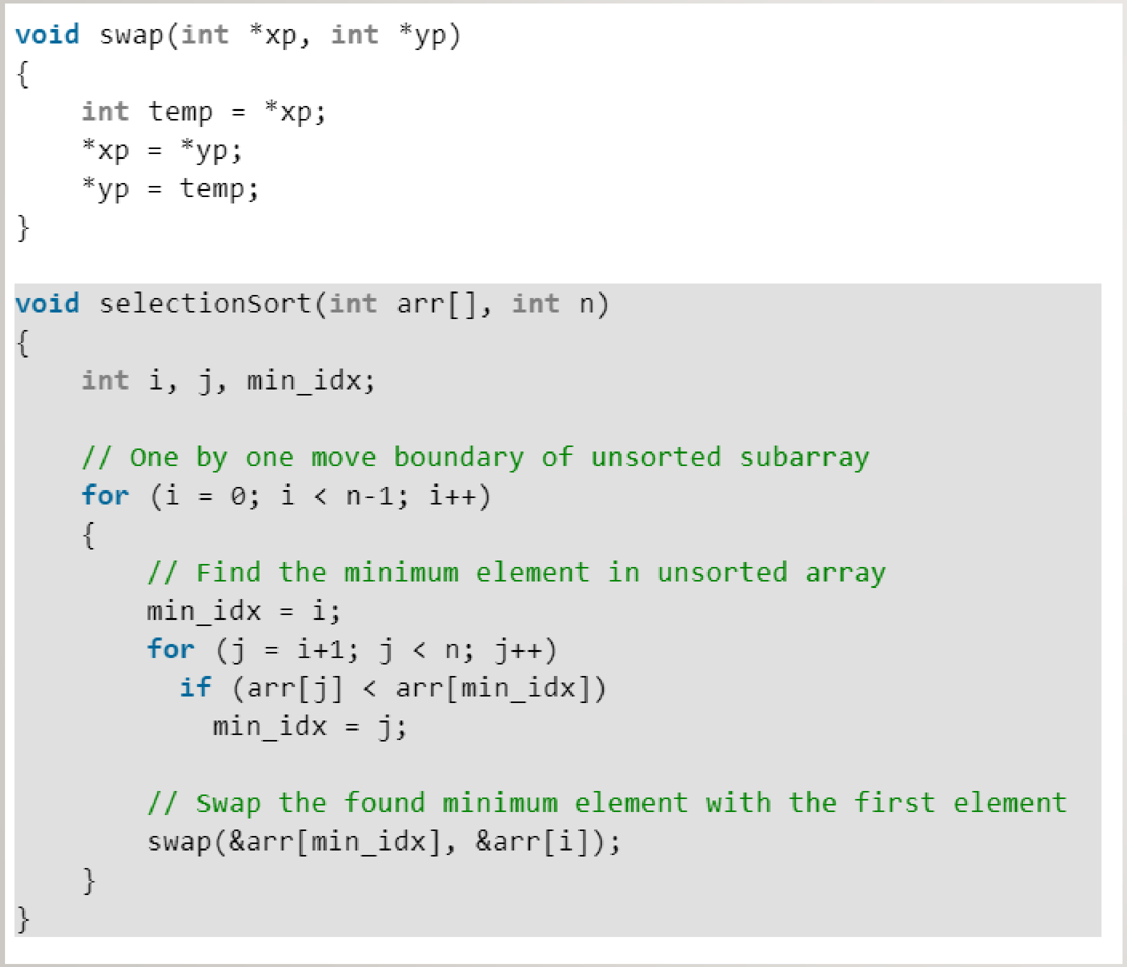
- what is Selection Sort Algorithm:



1 WHAT IS Algorithm

1.6 Selection Sort

CODE



```
void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

void selectionSort(int arr[], int n)
{
    int i, j, min_idx;

    // One by one move boundary of unsorted subarray
    for (i = 0; i < n-1; i++)
    {
        // Find the minimum element in unsorted array
        min_idx = i;
        for (j = i+1; j < n; j++)
          if (arr[j] < arr[min_idx])
            min_idx = j;

        // Swap the found minimum element with the first element
        swap(&arr[min_idx], &arr[i]);
    }
}
```

1 WHAT IS Algorithm

1.6 Selection Sort

CODE

```
/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Driver program to test above functions
int main()
{
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr)/sizeof(arr[0]);
    selectionSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

OUTPUT

```
Sorted array:
11 12 22 25 64
```

DESCRIPTION

- **Time Complexity:** O(n^2) as there are two nested loops.
- **Auxiliary Space:** O(1).

1 WHAT IS Algorithm

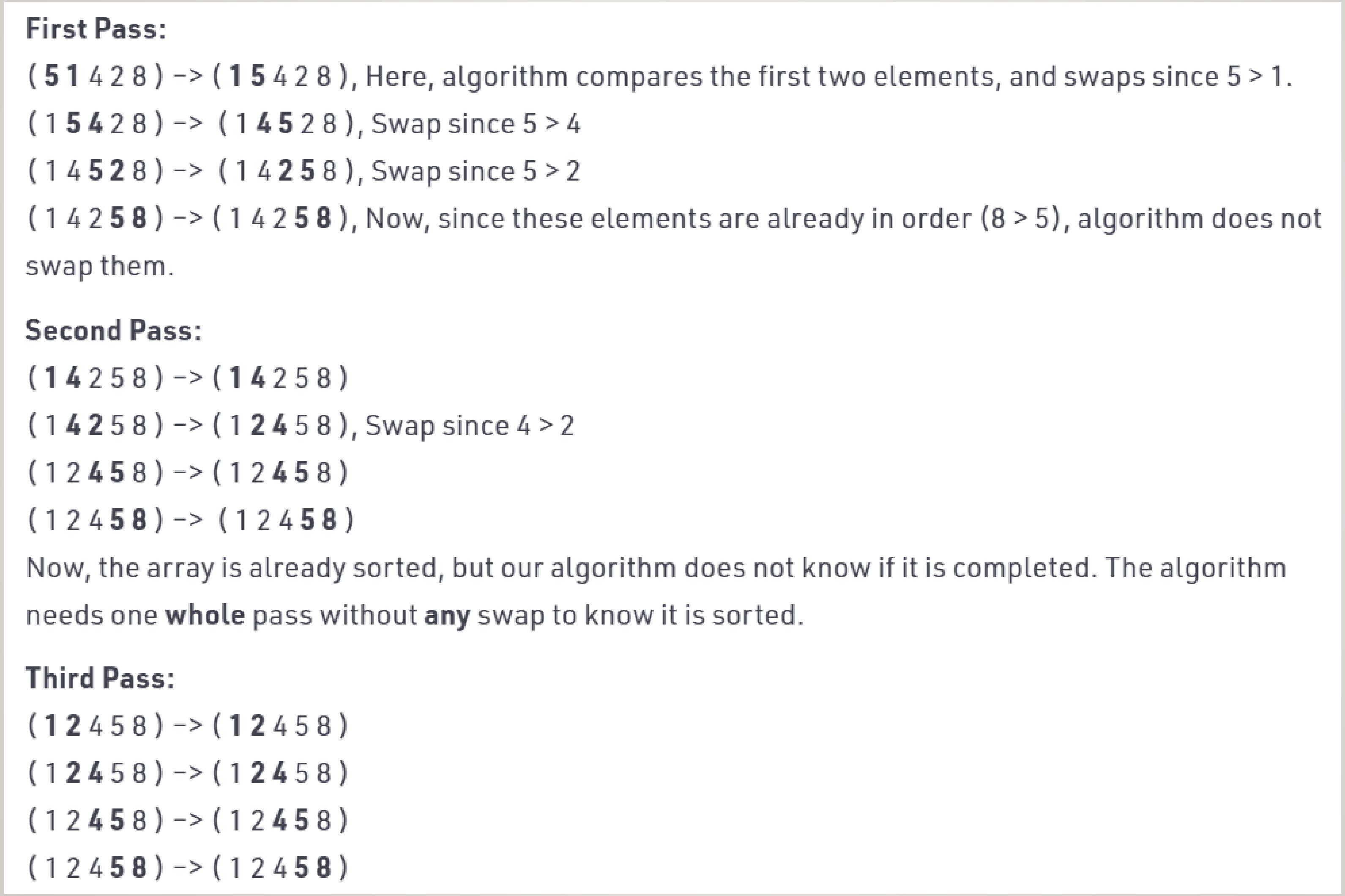
1.7 Bubble sort

➢ what is Bubble sort algorithms

- Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

1 W H A T I S Algorithm

1.7 Bubble sort



First Pass:

(**5 1** 4 2 8) –> (**1 5** 4 2 8), Here, algorithm compares the first two elements, and swaps since 5 > 1.

(1 **5 4** 2 8) –> (1 **4 5** 2 8), Swap since 5 > 4

(1 4 **5 2** 8) –> (1 4 **2 5** 8), Swap since 5 > 2

(1 4 2 **5 8**) –> (1 4 2 **5 8**), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(**1 4** 2 5 8) –> (**1 4** 2 5 8)

(1 **4 2** 5 8) –> (1 **2 4** 5 8), Swap since 4 > 2

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

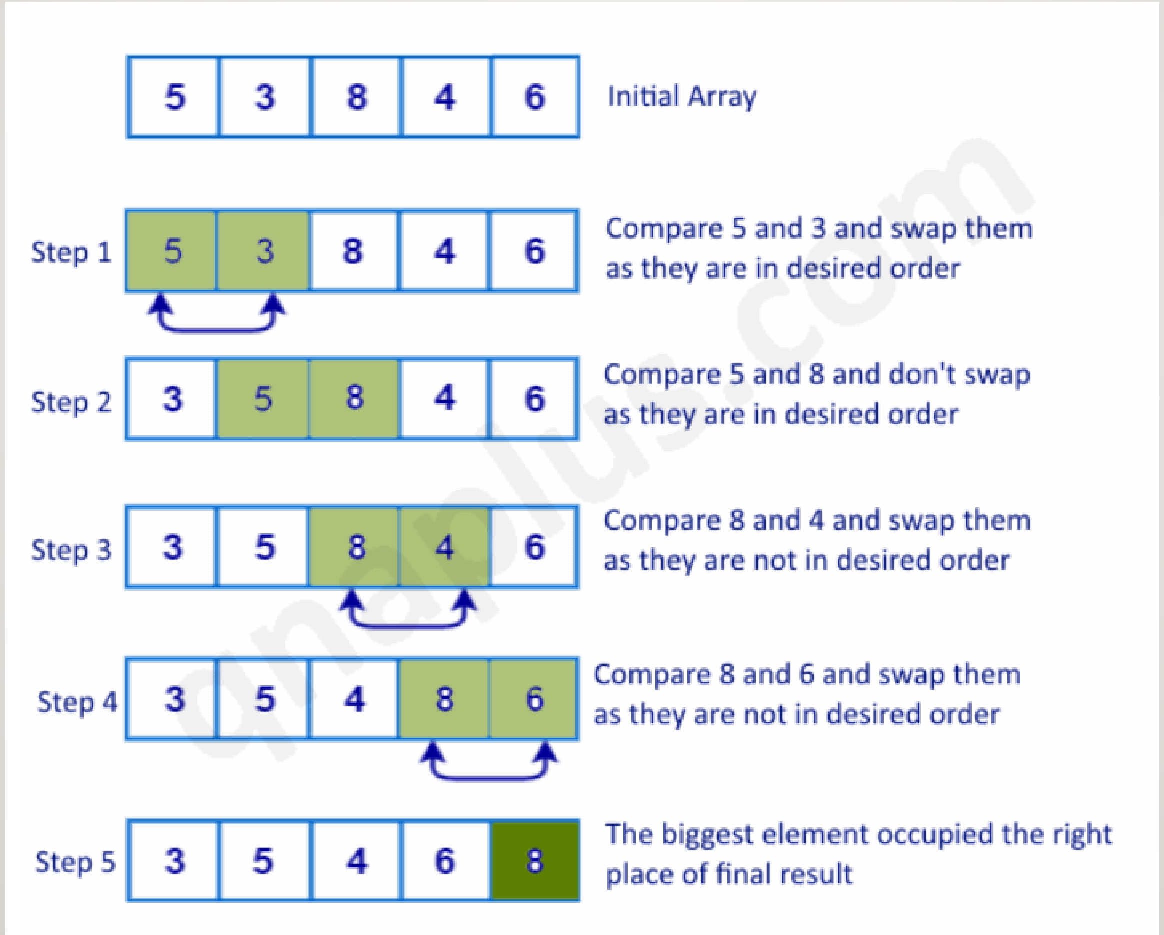
(**1 2** 4 5 8) –> (**1 2** 4 5 8)

(1 **2 4** 5 8) –> (1 **2 4** 5 8)

(1 2 **4 5** 8) –> (1 2 **4 5** 8)

(1 2 4 **5 8**) –> (1 2 4 **5 8**)

- Graphical Illustration of Bubble Sort:



5 3 8 4 6 Initial Array
Step 1 5 3 8 4 6 Compare 5 and 3 and swap them as they are in desired order
Step 2 3 5 8 4 6 Compare 5 and 8 and don't swap as they are in desired order
Step 3 3 5 8 4 6 Compare 8 and 4 and swap them as they are not in desired order
Step 4 3 5 4 8 6 Compare 8 and 6 and swap them as they are not in desired order
Step 5 3 5 4 6 8 The biggest element occupied the right place of final result

First iteration of Bubble Sort

1.7 Bubble Sort

CODE

```
void swap(int *xp, int *yp)
{
    int temp = *xp;
    *xp = *yp;
    *yp = temp;
}

// An optimized version of Bubble Sort
void bubbleSort(int arr[], int n)
{
   int i, j;
   bool swapped;
   for (i = 0; i < n-1; i++)
   {
     swapped = false;
     for (j = 0; j < n-i-1; j++)
     {
        if (arr[j] > arr[j+1])
        {
           swap(&arr[j], &arr[j+1]);
           swapped = true;
        }
     }

     // IF no two elements were swapped by inner loop, then break
     if (swapped == false)
        break;
   }
}
```

1 WHAT IS Algorithm

1.7 Bubble Sort

CODE

```
/* Function to print an array */
void printArray(int arr[], int size)
{
    int i;
    for (i=0; i < size; i++)
        printf("%d ", arr[i]);
    printf("n");
}

// Driver program to test above functions
int main()
{
    int arr[] = {64, 34, 25, 12, 22, 11, 90};
    int n = sizeof(arr)/sizeof(arr[0]);
    bubbleSort(arr, n);
    printf("Sorted array: \n");
    printArray(arr, n);
    return 0;
}
```

OUTPUT

```
Sorted array:
11 12 22 25 34 64 90
```

DESCRIPTION

- **Worst and Average Case Time Complexity:** O(n*n). Worst case occurs when array is reverse sorted.
- **Best Case Time Complexity:** O(n). Best case occurs when array is already sorted.
- **Auxiliary Space:** O(1)