# UNIT-I

**INTRODUCTION TO ALGORITHMS AND DATA STRUCTURES**

**Definition: -** An algorithm is a ***Step By Step*** process to solve a problem, where each step indicates an intermediate task. Algorithm contains finite number of steps that leads to the solution of the problem.

**Properties /Characteristics of an Algorithm**:- Algorithm has the following basic properties

* Input-Output:- Algorithm takes ‘0’ or more input and produces the required output. This is the basic characteristic of an algorithm.
* Finiteness:- An algorithm must terminate in countable number of steps.
* Definiteness: Each step of an algorithm must be stated clearly and unambiguously.
* Effectiveness: Each and every step in an algorithm can be converted in to programming language statement.
* Generality: Algorithm is generalized one. It works on all set of inputs and provides the required output. In other words it is not restricted to a single input value.

## Categories of Algorithm:

Based on the different types of steps in an Algorithm, it can be divided into three categories, namely

* Sequence
* Selection and
* Iteration

**Sequence:** The steps described in an algorithm are performed successively one by one without skipping any step. The sequence of steps defined in an algorithm should be simple and easy to understand. Each instruction of such an algorithm is executed, because no selection procedure or conditional branching exists in a sequence algorithm.

Example:

// adding two numbers Step 1: start

Step 2: read a,b Step 3: Sum=a+b Step 4: write Sum Step 5: stop

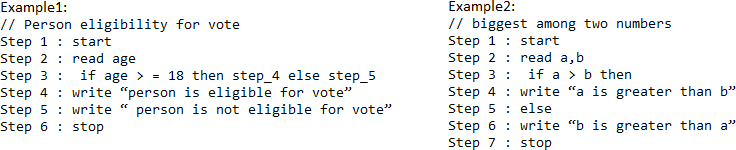
**Selection:** The sequence type of algorithms are not sufficient to solve the problems, which involves decision and conditions. In order to solve the problem which involve decision making or option selection, we go for Selection type of algorithm. The general format of Selection type of statement is as shown below:

if(condition) Statement-1;

else

Statement-2;

The above syntax specifies that if the condition is true, statement-1 will be executed otherwise statement-2 will be executed. In case the operation is unsuccessful. Then sequence of algorithm should be changed/ corrected in such a way that the system will re- execute until the operation is successful.



**Iteration:** Iteration type algorithms are used in solving the problems which involves repetition of statement. In this type of algorithms, a particular number of statements are repeated ‘n’ no. of times.

Example1:

Step 1 : start Step 2 : read n

Step 3 : repeat step 4 until n>0 Step 4 : (a) r=n mod 10

1. s=s+r
2. n=n/10 Step 5 : write s

Step 6 : stop

## Performance Analysis an Algorithm:

The Efficiency of an Algorithm can be measured by the following metrics.

1. Time Complexity and
2. Space Complexity. i.Time Complexity:

The amount of time required for an algorithm to complete its execution is its time complexity. An algorithm is said to be efficient if it takes the minimum (reasonable) amount of time to complete its execution.

1. ​Space Complexity:

The amount of space occupied by an algorithm is known as Space Complexity. An algorithm is said to be efficient if it occupies less space and required the minimum amount of time to complete its execution.

## Write an algorithm for roots of a Quadratic Equation?

// Roots of a quadratic Equation Step 1 : start

Step 2 : read a,b,c

Step 3 : if (a= 0) then step 4 else step 5

Step 4 : Write “ Given equation is a linear equation “ Step 5 : d=(b \* b) \_ (4 \*a \*c)

Step 6 : if ( d>0) then step 7 else step8

Step 7 : Write “ Roots are real and Distinct” Step 8: if(d=0) then step 9 else step 10

Step 9: Write “Roots are real and equal” Step 10: Write “ Roots are Imaginary” Step 11: stop

## Write an algorithm to find the largest among three different numbers entered by user

Step 1: Start

Step 2: Declare variables a,b and c. Step 3: Read variables a,b and c.

Step 4: If a>b

If a>c

Display a is the largest number. Else

Display c is the largest number.

Else

If b>c

Display b is the largest number. Else

Display c is the greatest number.

Step 5: Stop

## Write an algorithm to find the factorial of a number entered by user.

Step 1: Start

Step 2: Declare variables n,factorial and i.

Step 3: Initialize variables factorial←1

i←1

Step 4: Read value of n

Step 5: Repeat the steps until i=n

* 1. : factorial←factorial\*i
  2. : i←i+1

Step 6: Display factorial Step 7: Stop

## Write an algorithm to find the Simple Interest for given Time and Rate of Interest .

Step 1: Start

Step 2: Read P,R,S,T.

Step 3: Calculate S=(PTR)/100 Step 4: Print S

Step 5: Stop

**ASYMPTOTIC NOTATIONS**

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as *f*(n) and may be for another operation it is computed as *g*(n2). This means the first operation running time will increase linearly with the increase in **n** and the running time of the second operation will increase exponentially when **n** increases. Similarly, the running time of both operations will be nearly the same if **n** is significantly small.

The time required by an algorithm falls under three types −

* **Best Case** − Minimum time required for program execution.
* **Average Case** − Average time required for program execution.
* **Worst Case** − Maximum time required for program execution.

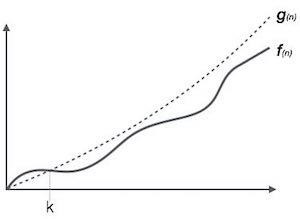
## Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

* Ο Notation
* Ω Notation
* θ Notation

## Big Oh Notation, Ο

The notation Ο(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

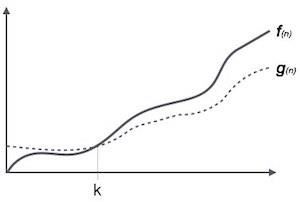


**For example**, for a function ***f*(n)**

Ο(*f*(n)) = { *g*(n) : there exists c > 0 and n0 such that *f*(n) ≤ c.*g*(n) for all n > n0. }

## Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

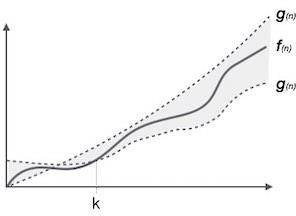


**For example**, for a function ***f*(n)**

Ω(*f*(n)) ≥ { *g*(n) : there exists c > 0 and n0 such that *g*(n) ≤ c.*f*(n) for all n > n0. }

## Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows −



θ(*f*(n)) = { *g*(n) if and only if *g*(n) = Ο(*f*(n)) and *g*(n) = Ω(*f*(n)) for all n > n0. }

# DATA STRUCTURES

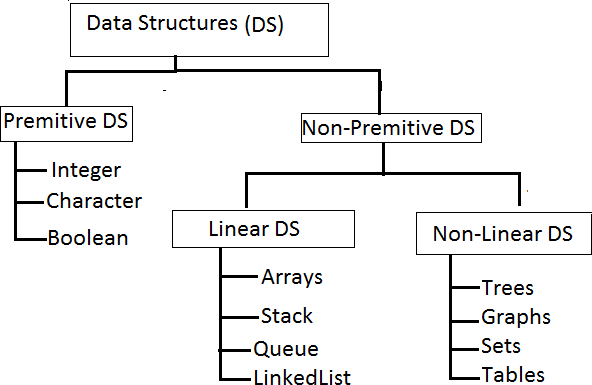
Data may be organized in many different ways logical or mathematical model of a program particularly organization of data. This organized data is called “Data Structure”.

## Or

The organized collection of data is called a ‘Data Structure’.

**Data Structure=Organized data +Allowed operations**

Data Structure involves two complementary goals. The first goal is to identify and develop useful, mathematical entities and operations and to determine what class of problems can be solved by using these entities and operations. The second goal is to determine representation for those abstract entities to implement abstract operations on this concrete representation.



Primitive Data structures are directly supported by the language ie; any operation is directly performed in these data items.

Ex: integer, Character, Real numbers etc.

Non-primitive data types are not defined by the programming language, but are instead created by the programmer.

Linear data structures organize their data elements in a linear fashion, where data elements are attached one after the other. Linear data structures are very easy to implement, since the memory of the computer is also organized in a linear fashion. Some commonly used linear data structures are arrays, linked lists, stacks and queues.

In nonlinear data structures, data elements are not organized in a sequential fashion. Data structures like multidimensional arrays, trees, graphs, tables and sets are some examples of widely used nonlinear data structures.

## Operations on the Data Structures:

Following operations can be performed on the data structures:

1. Traversing
2. Searching
3. Inserting
4. Deleting
5. Sorting
6. Merging
7. *​Traversing-* It is used to access each data item exactly once so that it can be processed.
8. *​ Searching-* It is used to find out the location of the data item if it exists in the given collection of data items.
9. *​Inserting-* It is used to add a new data item in the given collection of data items.
10. *​Deleting-* It is used to delete an existing data item from the given collection of data items.
11. *Sorting-* It is used to arrange the data items in some order i.e. in ascending or descending order in case of numerical data and in dictionary order in case of alphanumeric data.
12. *​ Merging-* It is used to combine the data items of two sorted files into single file in the sorted form.

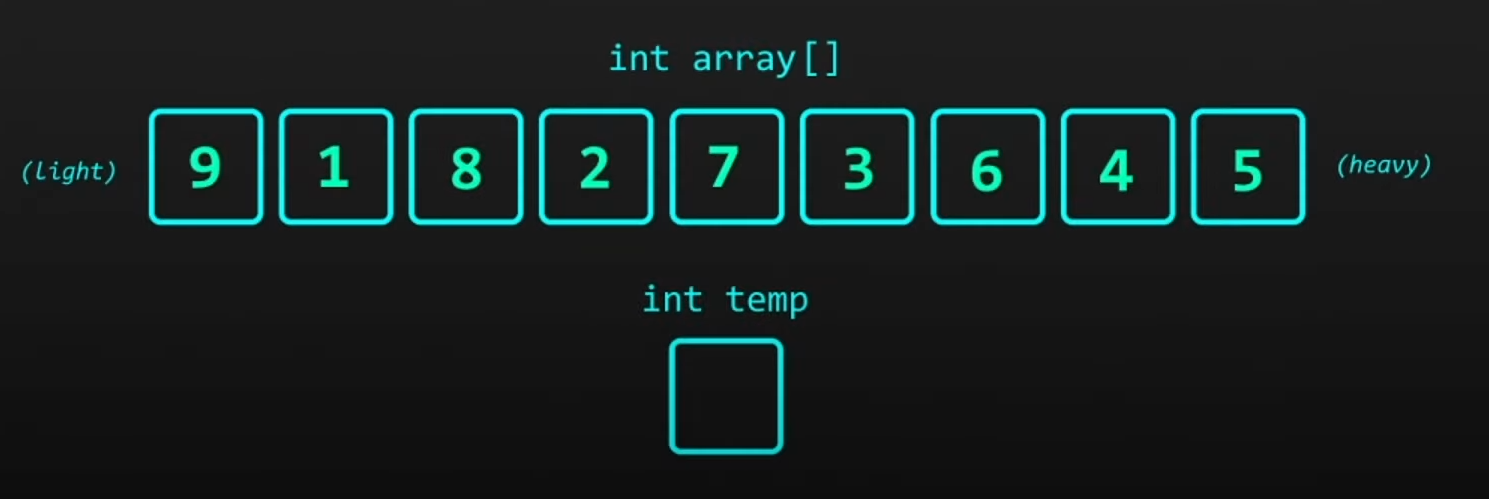
**. Bubble Sort**

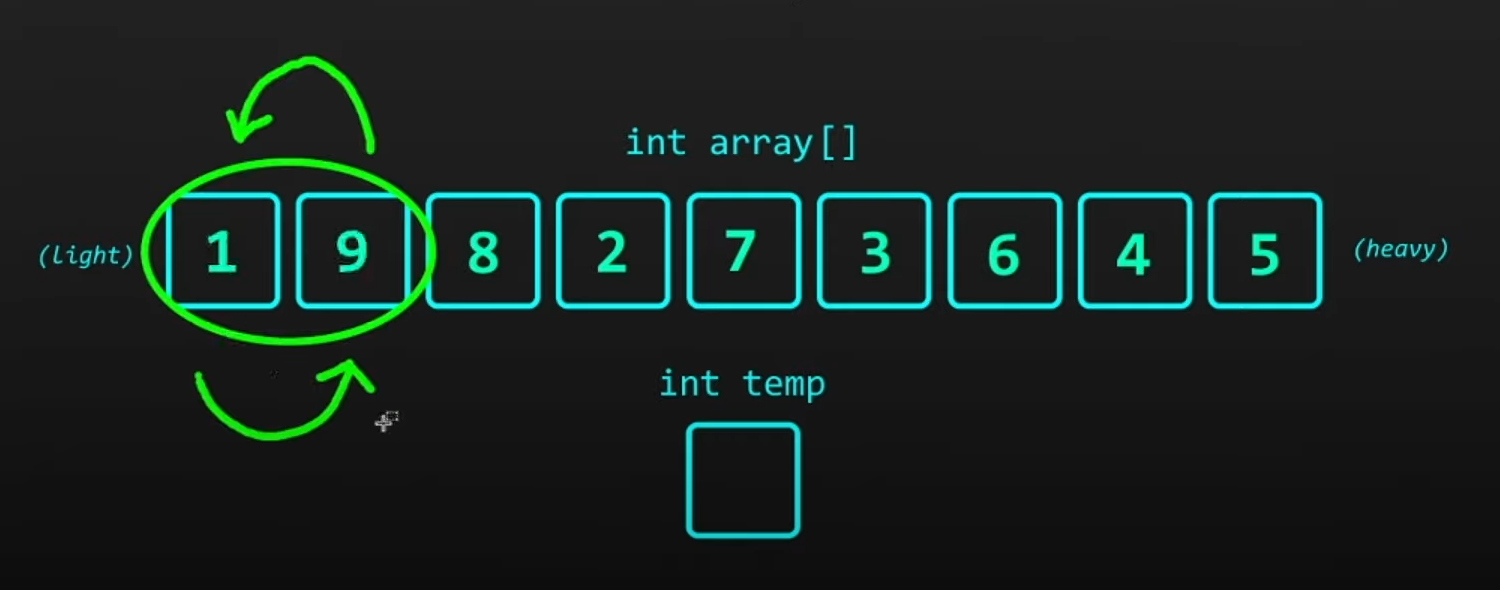
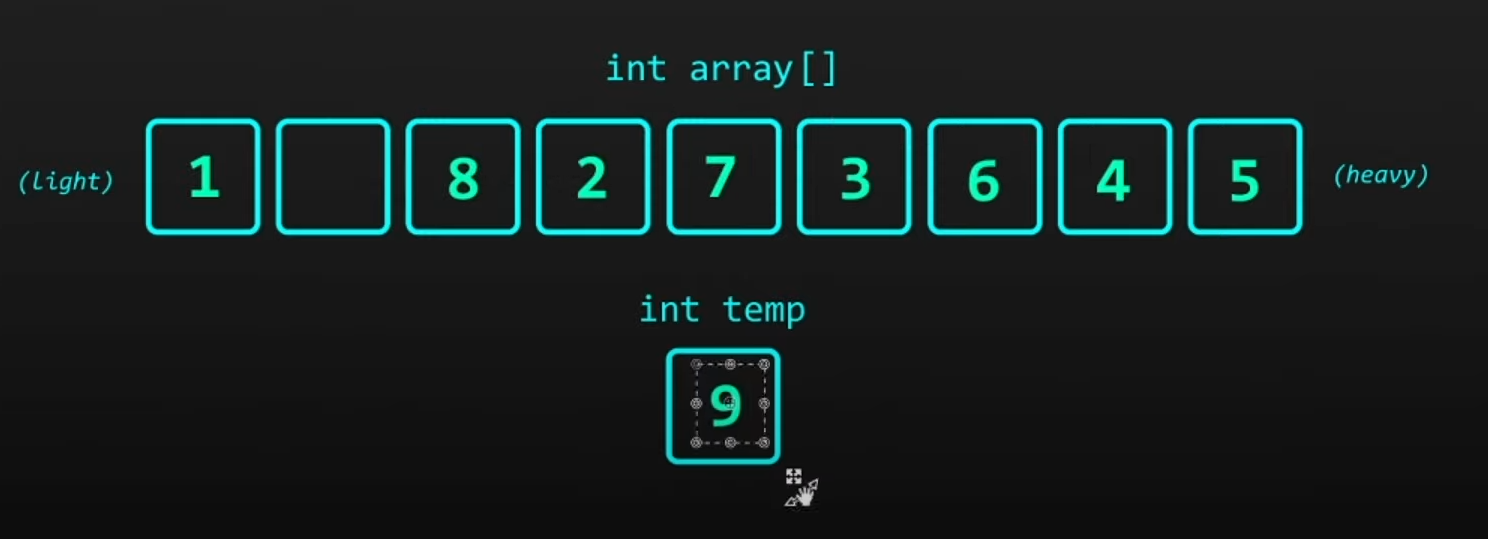
**Concept:**

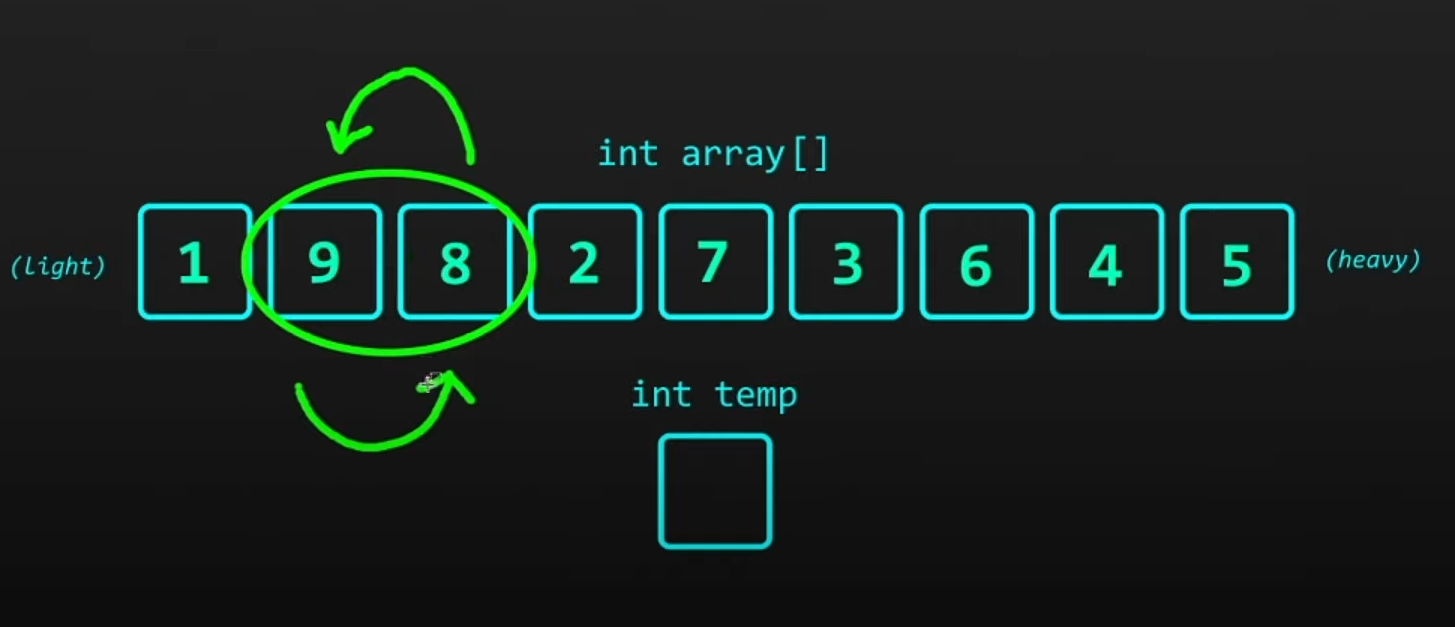
* Bubble Sort compares adjacent elements and swaps them if they are in the wrong order.
* It continues this process for each element in the list until the list is sorted.

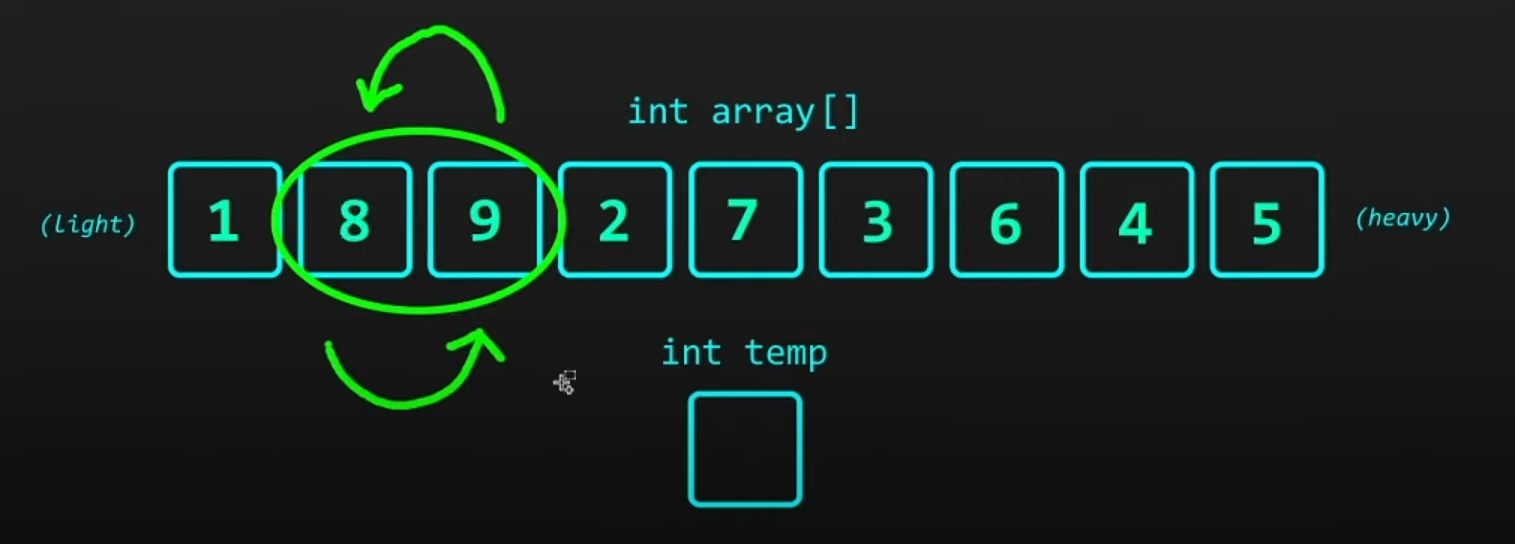
**Algorithm:**

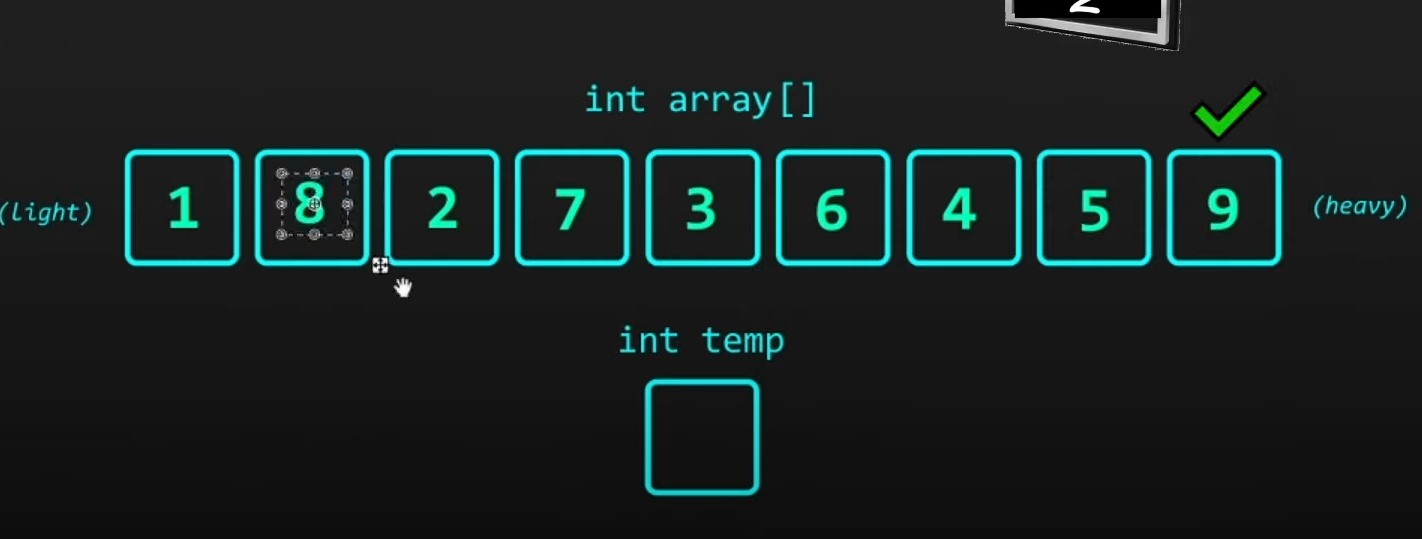
1. Start with the first element.
2. Compare it with the next element.
3. Swap if the first element is greater than the second.
4. Repeat for all elements in the list.
5. After each pass, the largest element "bubbles up" to its correct position.
6. Repeat the process for the remaining unsorted elements.

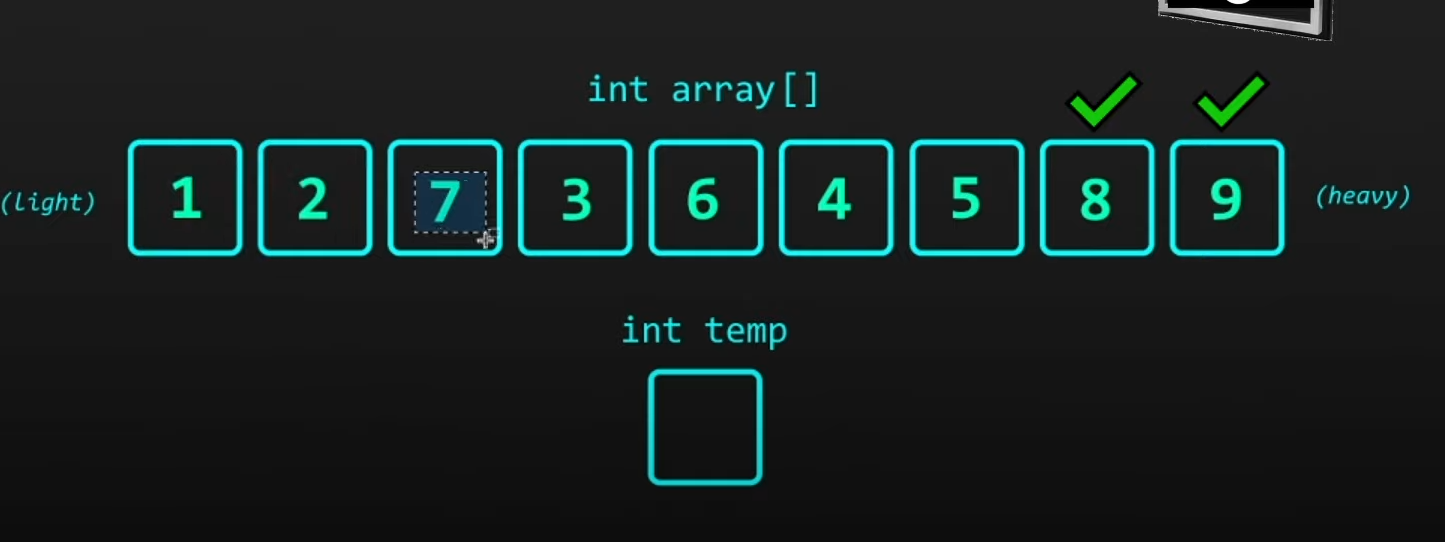


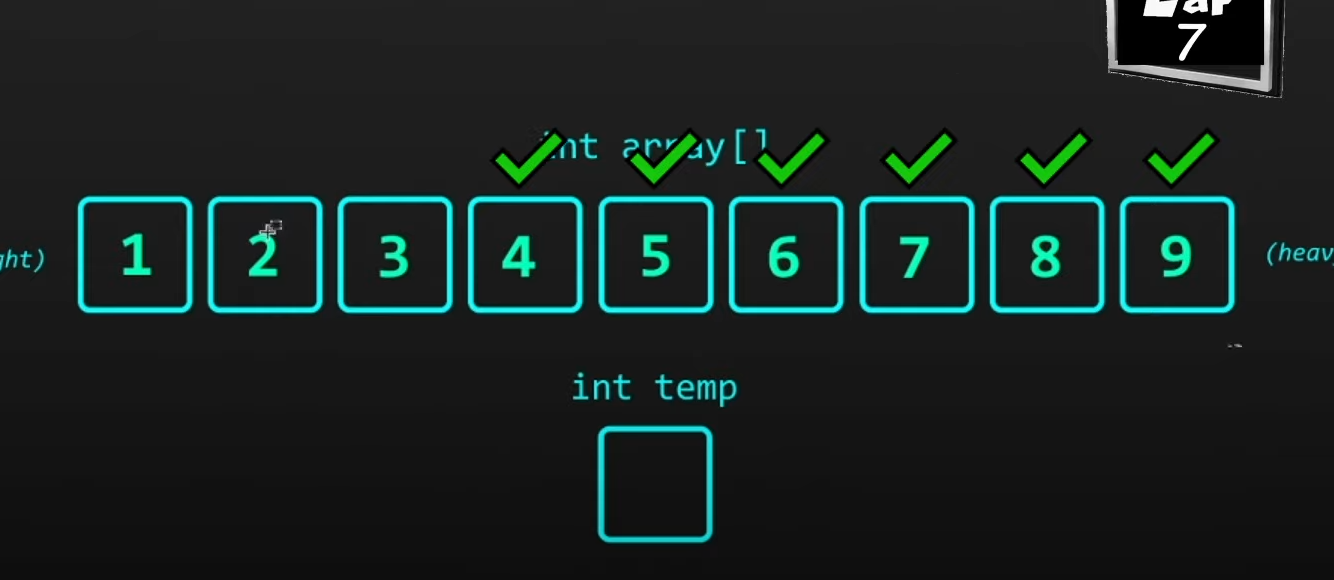


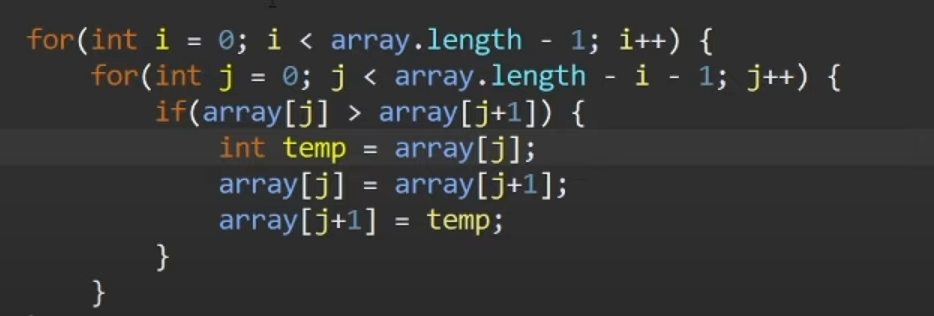












**Performance Analysis:**

* **Best Case (Already Sorted):** O(n)
* **Worst Case (Reverse Sorted):** O(n²)
* **Average Case:** O(n²)
* **Space Complexity:** O(1) (In-place)

**Advantages:** Simple to implement. **Disadvantages:** Inefficient for large datasets.

**2. Insertion Sort**

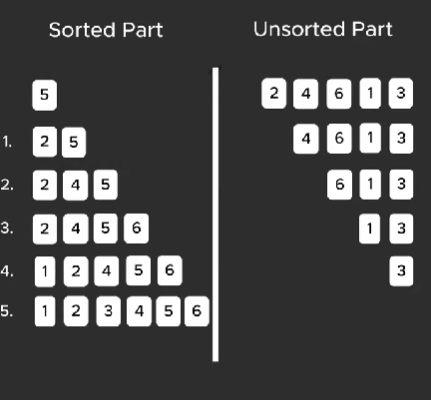
**Concept:**

* Insertion Sort builds the final sorted array one element at a time by picking elements from the unsorted part and inserting them into their correct position in the sorted part.

**Algorithm:**

1. Start with the second element, treating the first element as sorted.
2. Compare the current element with elements in the sorted part.
3. Shift all larger elements to the right.
4. Insert the current element into its correct position.
5. Repeat for all elements.

[5,2,4,6,1,3]



void insertionSort(int array[], int size) {

for (int step = 1; step < size; step++) {

int key = array[step];

int j = step - 1;

// Compare key with each element on the left of it until an element smaller than

// it is found.

// For descending order, change key<array[j] to key>array[j].

while (j >=0 && key < array[j]) {

array[j + 1] = array[j];

--j;

}

array[j + 1] = key;

}

}

**Performance Analysis:**

* **Best Case (Already Sorted):** O(n)
* **Worst Case (Reverse Sorted):** O(n²)
* **Average Case:** O(n²)
* **Space Complexity:** O(1) (In-place)

**Advantages:** Efficient for small datasets or nearly sorted data. **Disadvantages:** Inefficient for large, randomly sorted datasets.

**3. Selection Sort**

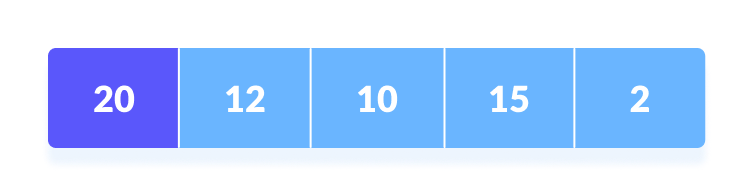
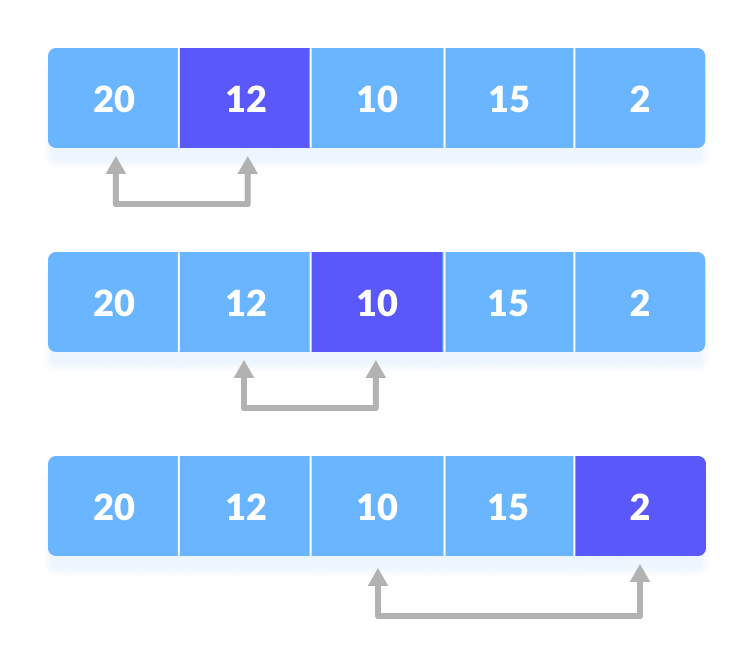
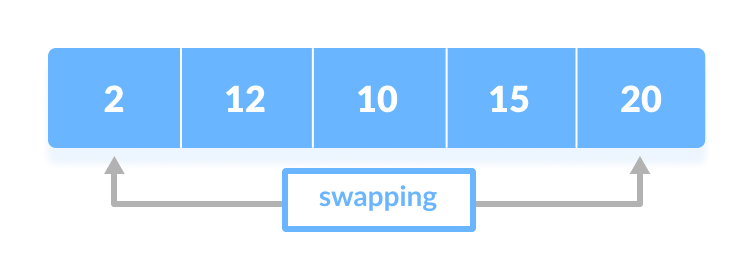
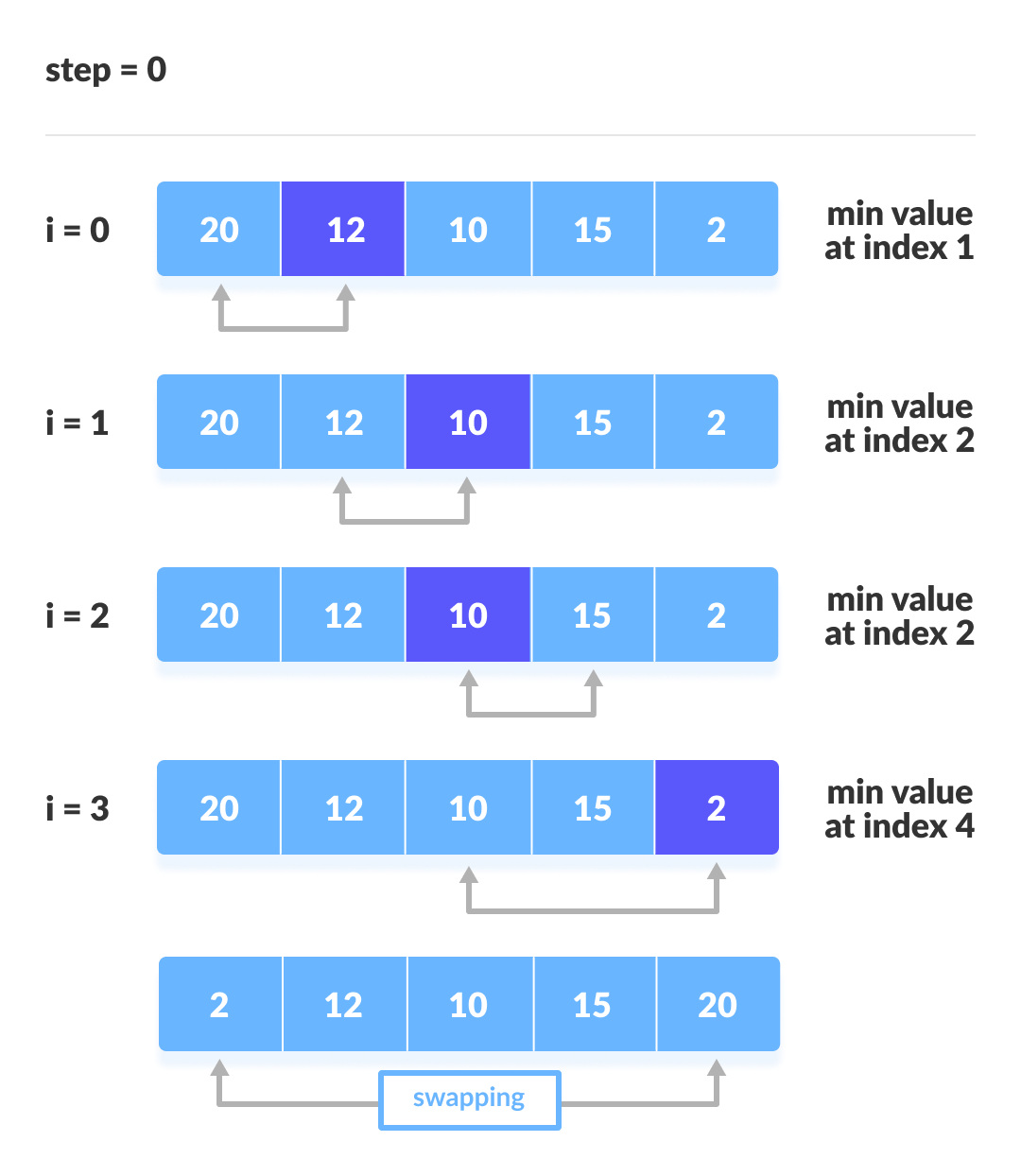
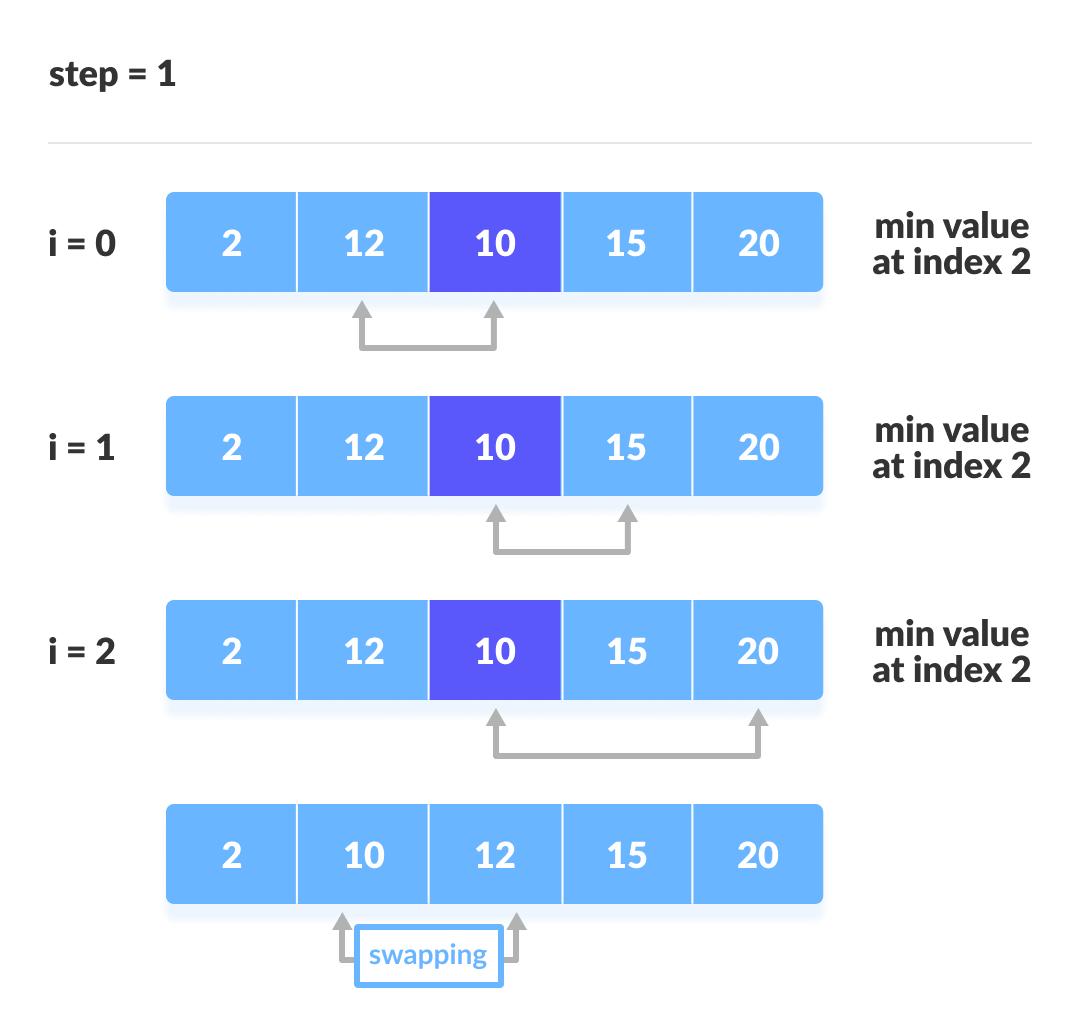
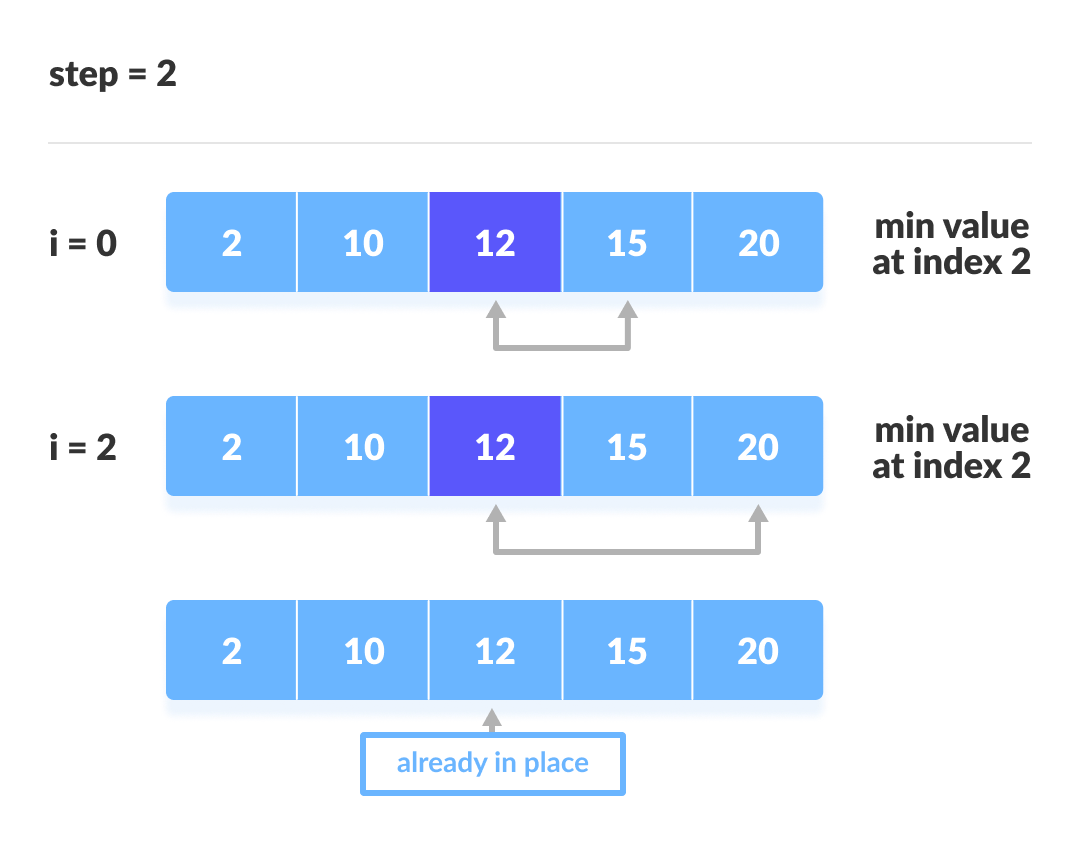
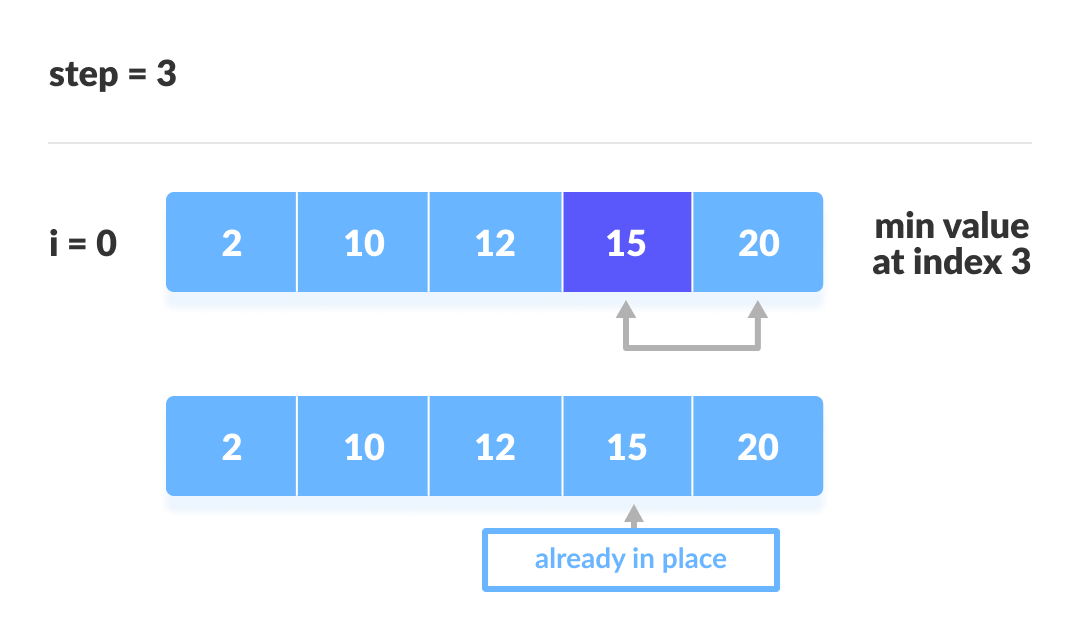
**Concept:**

* Selection Sort divides the array into a sorted and unsorted part. It repeatedly selects the smallest element from the unsorted part and swaps it with the first element of the unsorted part.

**Algorithm:**

1. Find the smallest element in the array.
2. Swap it with the first element.
3. Repeat the process for the remaining unsorted elements.

**Working of Selection Sort**

1. Set the first element as minimum.Select first element as minimum
2. Compare minimum with the second element. If the second element is smaller than minimum, assign the second element as minimum.  
     
   Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.Compare minimum with the remaining elements
3. After each iteration, minimum is placed in the front of the unsorted list.Swap the first with minimum
4. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.The first iterationThe second iterationThe third iterationThe fourth iteration

**Selection Sort Algorithm**

selectionSort(array, size)

for i from 0 to size - 1 do

set i as the index of the current minimum

for j from i + 1 to size - 1 do

if array[j] < array[current minimum]

set j as the new current minimum index

if current minimum is not i

swap array[i] with array[current minimum]

end selectionSort

Example code :

// Selection sort in C

#include <stdio.h>

// function to swap the the position of two elements

void swap(int \*a, int \*b) {

int temp = \*a;

\*a = \*b;

\*b = temp;

}

void selectionSort(int array[], int size) {

for (int step = 0; step < size - 1; step++) {

int min\_idx = step;

for (int i = step + 1; i < size; i++) {

// To sort in descending order, change > to < in this line.

// Select the minimum element in each loop.

if (array[i] < array[min\_idx])

min\_idx = i;

}

// put min at the correct position

swap(&array[min\_idx], &array[step]);

}

}

// function to print an array

void printArray(int array[], int size) {

for (int i = 0; i < size; ++i) {

printf("%d ", array[i]);

}

printf("\n");

}

// driver code

int main() {

int data[] = {20, 12, 10, 15, 2};

int size = sizeof(data) / sizeof(data[0]);

selectionSort(data, size);

printf("Sorted array in Acsending Order:\n");

printArray(data, size);

}

**Performance Analysis:**

* **Best Case:** O(n²)
* **Worst Case:** O(n²)
* **Average Case:** O(n²)
* **Space Complexity:** O(1) (In-place)

**Advantages:** Simple to implement. **Disadvantages:** Always performs O(n²) comparisons, even if the array is already sorted.

**4. Shell Sort**

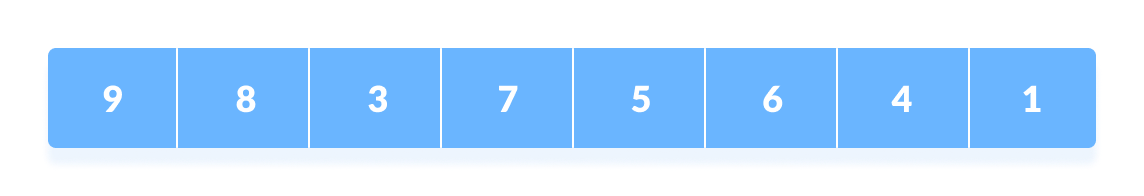
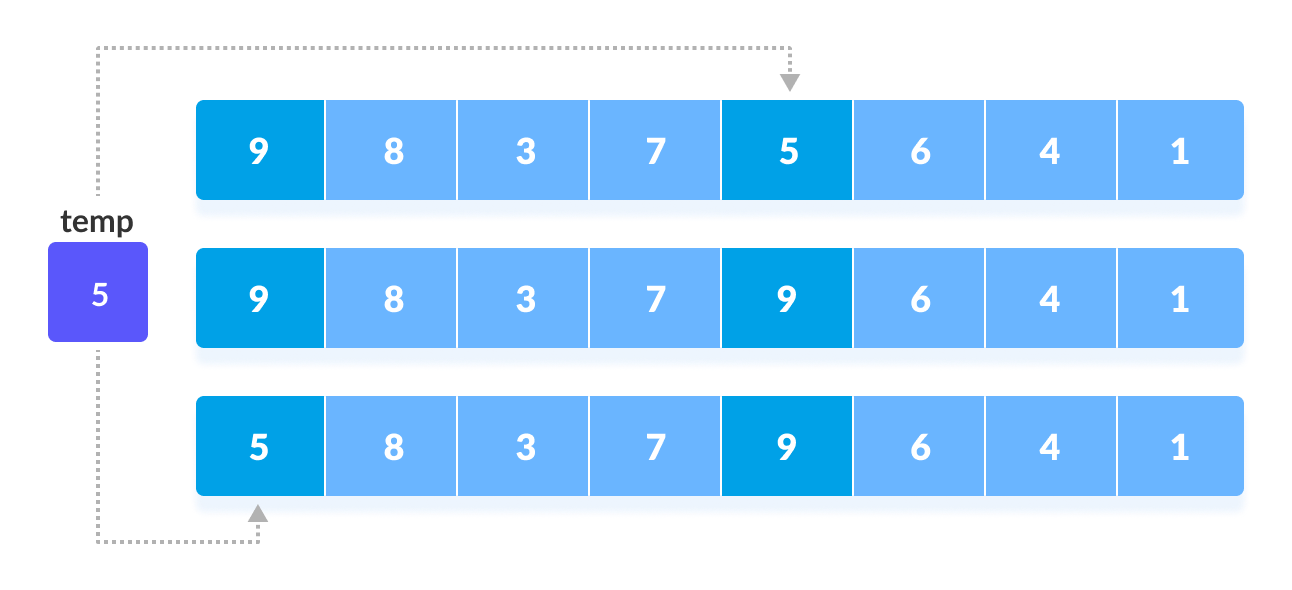
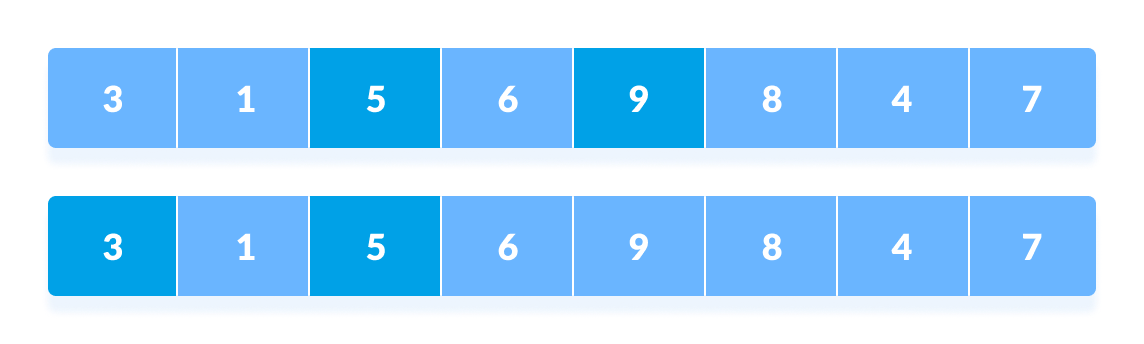
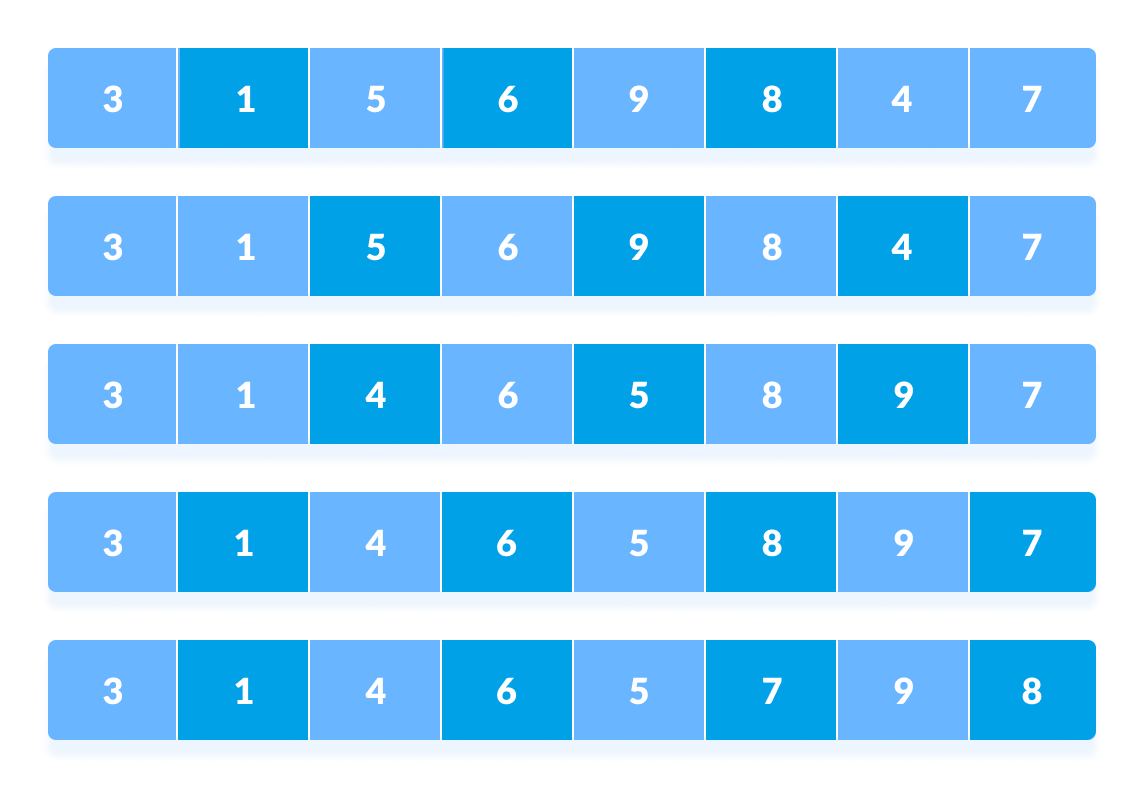
**Concept:**

* Shell Sort is an optimization over Insertion Sort. It sorts elements far apart from each other and successively reduces the interval between elements to be compared.

**Algorithm:**

1. Divide the array into subarrays using a gap value.
2. Sort each subarray using Insertion Sort.
3. Reduce the gap and repeat until the gap is 1.

**Working of Shell Sort**

1. Suppose, we need to sort the following array.Initial array
2. We are using the shell's original sequence (N/2, N/4, ...1) as intervals in our algorithm.  
     
   In the first loop, if the array size is N = 8 then, the elements lying at the interval of N/2 = 4 are compared and swapped if they are not in order.
   1. The 0th element is compared with the 4th element.
   2. If the 0th element is greater than the 4th one then, the 4th element is first stored in temp variable and the 0th element (ie. greater element) is stored in the 4th position and the element stored in temp is stored in the 0th position.Rearrange the elements at n/2 interval  
      This process goes on for all the remaining elements.Rearrange all the elements at n/2 interval
3. In the second loop, an interval of N/4 = 8/4 = 2 is taken and again the elements lying at these intervals are sorted.Rearrange the elements at n/4 interval  
   You might get confused at this point.All the elements in the array lying at the current interval are compared.  
   The elements at 4th and 2nd position are compared. The elements at 2nd and 0th position are also compared. All the elements in the array lying at the current interval are compared.
4. The same process goes on for remaining elements.Rearrange all the elements at n/4 interval
5. Finally, when the interval is N/8 = 8/8 =1 then the array elements lying at the interval of 1 are sorted. The array is now completely sorted.Rearrange the elements at n/8 interval

**Shell Sort Algorithm**

shellSort(array, size)

for interval i <- size/2n down to 1

for each interval "i" in array

sort all the elements at interval "i"

end shellSort

implementation :

// Shell Sort in C programming

#include <stdio.h>

// Shell sort



void shellSort(int array[], int n) {

// Rearrange elements at each n/2, n/4, n/8, ... intervals

for (int interval = n / 2; interval > 0; interval /= 2) {

for (int i = interval; i < n; i += 1) {

int temp = array[i];

int j;

for (j = i; j >= interval && array[j - interval] > temp; j -= interval) {

array[j] = array[j - interval];

}

array[j] = temp;

}

}

}

// Print an array

void printArray(int array[], int size) {

for (int i = 0; i < size; ++i) {

printf("%d ", array[i]);

}

printf("\n");

}

// Driver code

int main() {

int data[] = {9, 8, 3, 7, 5, 6, 4, 1};

int size = sizeof(data) / sizeof(data[0]);

shellSort(data, size);

printf("Sorted array: \n");

printArray(data, size);

}

**Time Complexity**

* **Worst Case Complexity**: less than or equal to O(n2)  
  Worst case complexity for shell sort is always less than or equal to O(n2).  
    
  According to Poonen Theorem, worst case complexity for shell sort is Θ(Nlog N)2/(log log N)2) or Θ(Nlog N)2/log log N) or Θ(N(log N)2) or something in between.
* **Best Case Complexity**: O(n\*log n)  
  When the array is already sorted, the total number of comparisons for each interval (or increment) is equal to the size of the array.
* **Average Case Complexity**: O(n\*log n)  
  It is around O(n1.25).

The complexity depends on the interval chosen. The above complexities differ for different increment sequences chosen. Best increment sequence is unknown.

**Space Complexity:**

The space complexity for shell sort is O(1).

**Performance Analysis:**

* **Best Case:** O(n log n)
* **Worst Case:** O(n²) (depending on gap sequence)
* **Average Case:** O(n log n)
* **Space Complexity:** O(1) (In-place)

**Advantages:** Faster than Bubble and Insertion Sort for larger datasets. **Disadvantages:** Performance depends on the choice of gap sequence.

**5. Radix Sort**

**Concept:**

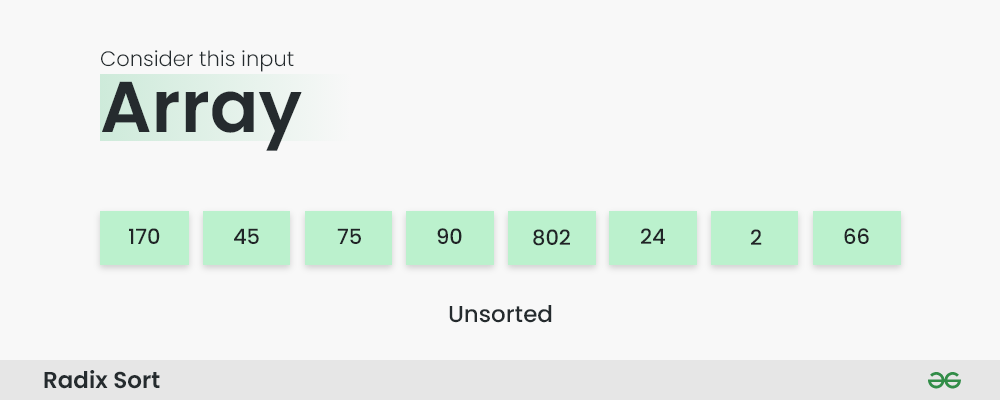
* Radix Sort sorts numbers by processing individual digits. It starts with the least significant digit and moves to the most significant digit.

**Algorithm:**

1. Find the maximum number to determine the number of digits.
2. Sort the array using counting sort for each digit.
3. Process digits from least significant to most significant.
4. Radix sort is a sorting algorithm that sorts the elements by first grouping the individual digits of the same **place value**. Then, sort the elements according to their increasing/decreasing order.
5. Suppose, we have an array of 8 elements. First, we will sort elements based on the value of the unit place. Then, we will sort elements based on the value of the tenth place. This process goes on until the last significant place.
6. Let the initial array be [121, 432, 564, 23, 1, 45, 788]. It is sorted according to radix sort as shown in the figure below.

**How does Radix Sort Algorithm work?**

*To perform radix sort on the array [170, 45, 75, 90, 802, 24, 2, 66], we follow these steps:*

**

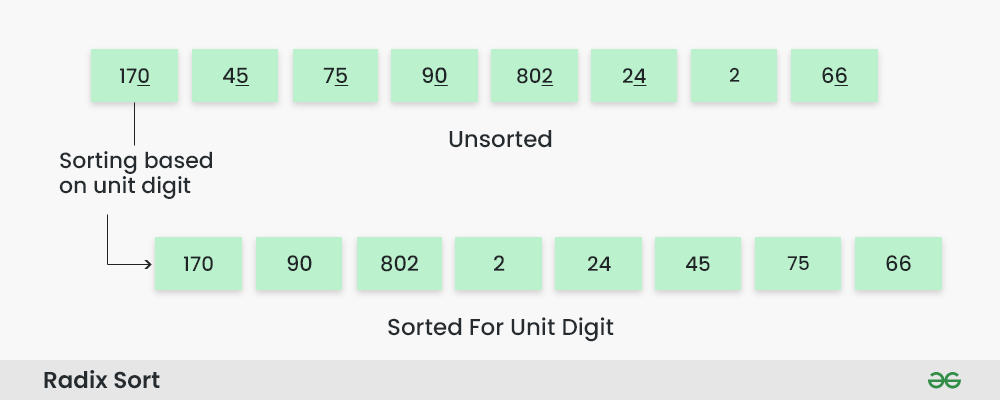
*How does Radix Sort Algorithm work | Step 1*

***Step 1:*** *Find the largest element in the array, which is 802. It has three digits, so we will iterate three times, once for each significant place.*

***Step 2:*** *Sort the elements based on the unit place digits (X=0). We use a stable sorting technique, such as counting sort, to sort the digits at each significant place. It’s important to understand that the default implementation of counting sort is unstable i.e. same keys can be in a different order than the input array. To solve this problem, We can iterate the input array in reverse order to build the output array. This strategy helps us to keep the same keys in the same order as they appear in the input array.*

***Sorting based on the unit place:***

* *Perform counting sort on the array based on the unit place digits.*
* *The sorted array based on the unit place is [170, 90, 802, 2, 24, 45, 75, 66].*

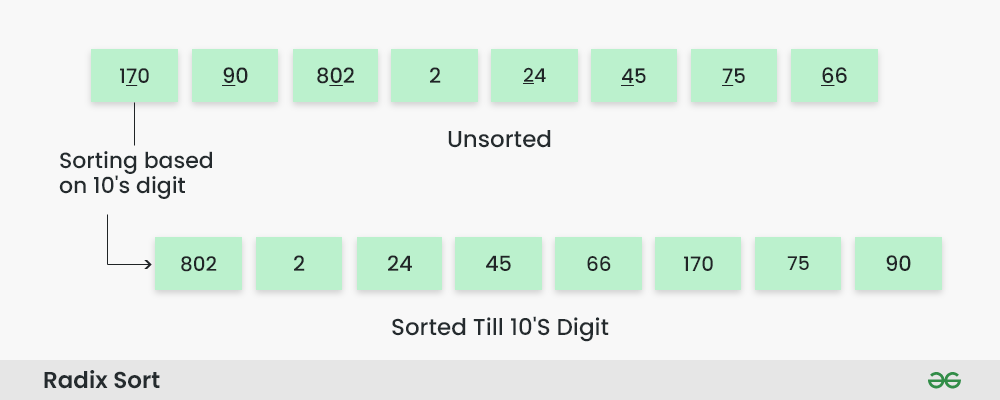
**

*How does Radix Sort Algorithm work | Step 2*

***Step 3:*** *Sort the elements based on the tens place digits.*

***Sorting based on the tens place:***

* *Perform counting sort on the array based on the tens place digits.*
* *The sorted array based on the tens place is [802, 2, 24, 45, 66, 170, 75, 90].*

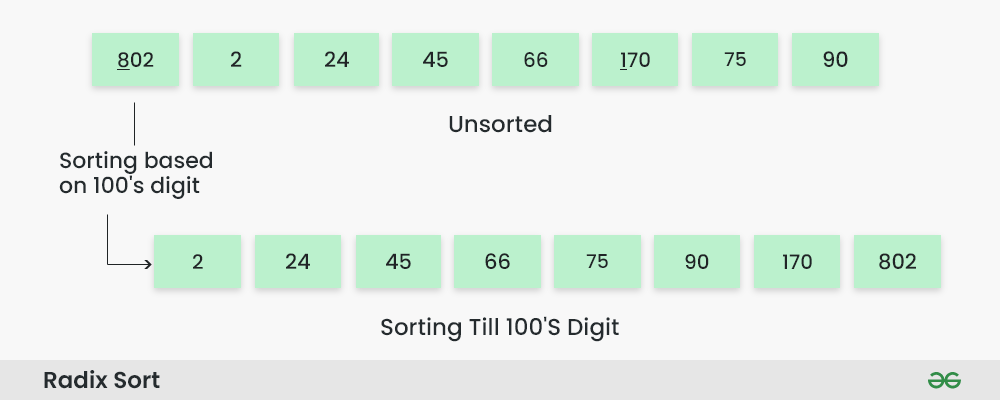
**

*How does Radix Sort Algorithm work | Step 3*

***Step 4:*** *Sort the elements based on the hundreds place digits.*

***Sorting based on the hundreds place:***

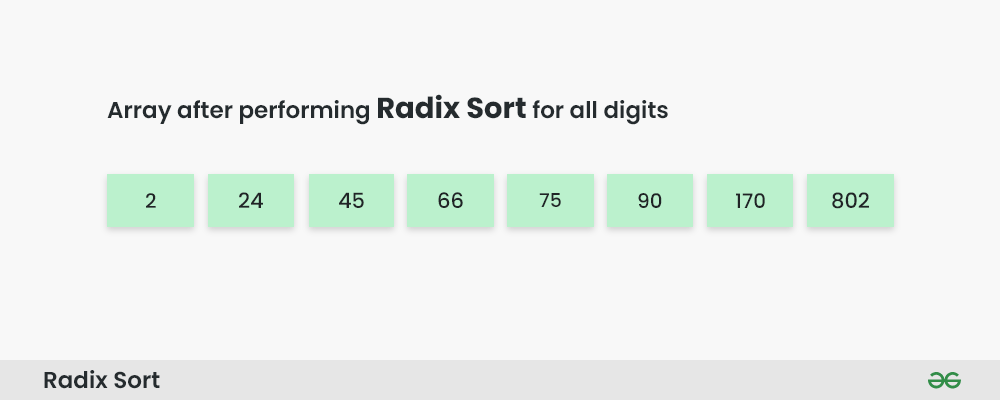
* *Perform counting sort on the array based on the hundreds place digits.*
* *The sorted array based on the hundreds place is [2, 24, 45, 66, 75, 90, 170, 802].*

**

*How does Radix Sort Algorithm work | Step 4*

***Step 5:*** *The array is now sorted in ascending order.*

*The final sorted array using radix sort is [2, 24, 45, 66, 75, 90, 170, 802].*

**

Implementation

// Radix Sort in C Programming

#include <stdio.h>

// Function to get the maximum value in the array

int getMax(int array[], int n) {

int max = array[0];

for (int i = 1; i < n; i++) {

if (array[i] > max) {

max = array[i];

}

}

return max;

}

// Using counting sort to sort the elements based on significant places

void countingSort(int array[], int n, int place) {

int output[n];

int count[10] = {0};

// Calculate count of elements

for (int i = 0; i < n; i++) {

int index = (array[i] / place) % 10;

count[index]++;

}

// Calculate cumulative count

for (int i = 1; i < 10; i++) {

count[i] += count[i - 1];

}

// Place the elements in sorted order

for (int i = n - 1; i >= 0; i--) {

int index = (array[i] / place) % 10;

output[count[index] - 1] = array[i];

count[index]--;

}

// Copy the sorted elements into original array

for (int i = 0; i < n; i++) {

array[i] = output[i];

}

}

// Main function to implement radix sort

void radixSort(int array[], int n) {

// Get maximum element

int maxElement = getMax(array, n);

// Apply counting sort to sort elements based on place value

for (int place = 1; maxElement / place > 0; place \*= 10) {

countingSort(array, n, place);

}

}

int main() {

int data[] = {121, 432, 564, 23, 1, 45, 788};

int n = sizeof(data) / sizeof(data[0]);

radixSort(data, n);

printf("Sorted array in ascending order:\n");

for (int i = 0; i < n; i++) {

printf("%d ", data[i]);

}

return 0;

}

**Radix Sort Complexity**

|  |  |
| --- | --- |
| **Time Complexity** |  |
| Best | O(n+k) |
| Worst | O(n+k) |
| Average | O(n+k) |
| **Space Complexity** | O(max) |
| **Stability** | Yes |

**Performance Analysis:**

* **Best Case:** O(nk)
* **Worst Case:** O(nk)
  + Where n is the number of elements and k is the number of digits.
* **Average Case:** O(nk)
* **Space Complexity:** O(n + k)

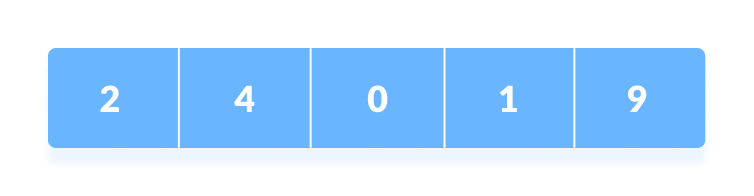
**Advantages:** Linear time complexity for fixed digit lengths. **Disadvantages:** Requires additional space and is only suitable for numeric data.

**Linear Search**

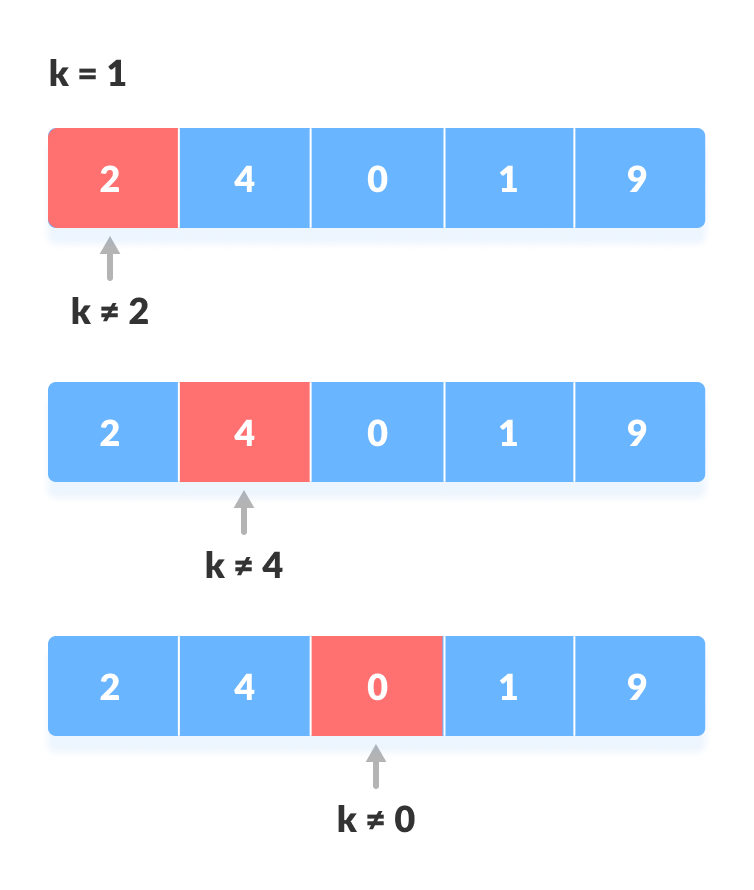
Linear search is a sequential searching algorithm where we start from one end and check every element of the list until the desired element is found. It is the simplest searching algorithm.

**How Linear Search Works?**

The following steps are followed to search for an element k = 1 in the list below.

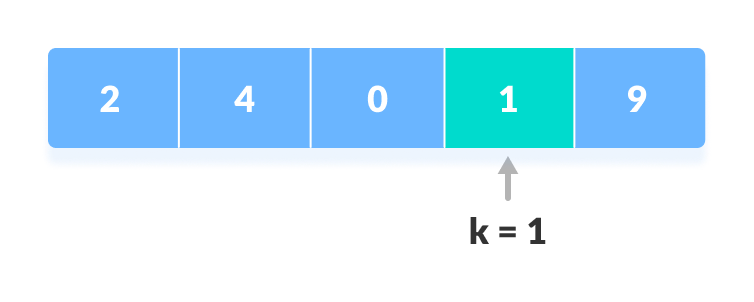


Array to be searched for

1. Start from the first element, compare k with each element x.

Compare with each element

1. If x == k, return the index.



Element found

1. Else, return not found.

**Linear Search Algorithm**

LinearSearch(array, key)

for each item in the array

if item == value

return its index

// Linear Search in C

#include <stdio.h>

int search(int array[], int n, int x) {

// Going through array sequencially

for (int i = 0; i < n; i++)

if (array[i] == x)

return i;

return -1;

}

int main() {

int array[] = {2, 4, 0, 1, 9};

int x = 1;

int n = sizeof(array) / sizeof(array[0]);

int result = search(array, n, x);

(result == -1) ? printf("Element not found") : printf("Element found at index: %d", result);

}

**Binary Search Working**

Binary Search Algorithm can be implemented in two ways which are discussed below.

1. Iterative Method
2. Recursive Method

The recursive method follows the **divide and conquer** approach.

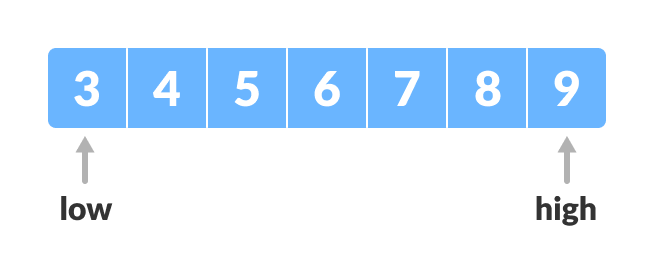
The general steps for both methods are discussed below.

The array in which searching is to be performed is:

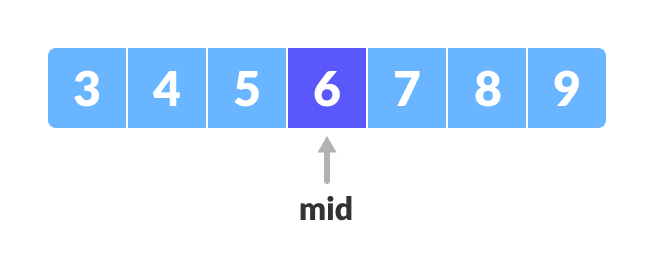
1. The array in which searching is to be performed is:

Initial array

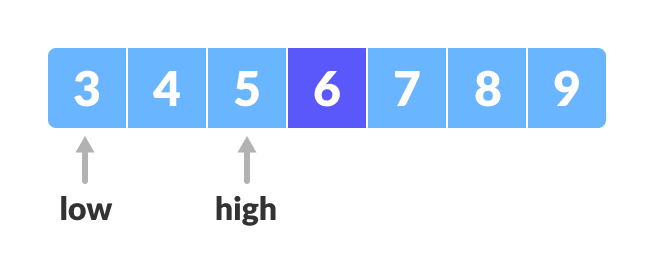
Let x = 4 be the element to be searched.

1. Set two pointers low and high at the lowest and the highest positions respectively.

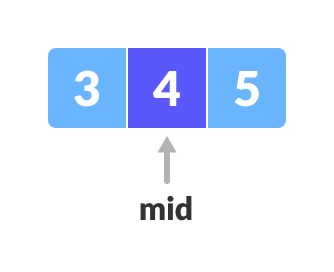
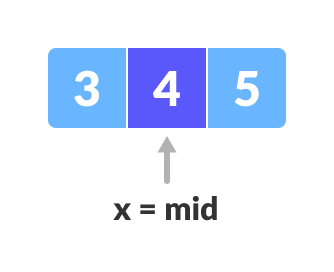
Setting pointers

1. Find the middle position mid of the array ie. mid = (low + high)/2 and arr[mid] = 6.

Mid element

1. If x == arr[mid], then return mid. Else, compare the element to be searched with arr[mid].
2. If x > arr[mid], compare x with the middle element of the elements on the right side of arr[mid]. This is done by setting low to low = mid + 1.
3. Else, compare x with the middle element of the elements on the left side of arr[mid]. This is done by setting high to high = mid - 1.

Finding mid element

1. Repeat steps 3 to 6 until low meets high.Mid element
2. x = 4 is found.Found

**Binary Search Algorithm**

**Iteration Method**

do until the pointers low and high meet each other.

mid = (low + high)/2

if (x == arr[mid])

return mid

else if (x > arr[mid]) // x is on the right side

low = mid + 1

else // x is on the left side

high = mid - 1

**Recursive Method**

binarySearch(arr, x, low, high)

if low > high

return False

else

mid = (low + high) / 2

if x == arr[mid]

return mid

else if x > arr[mid] // x is on the right side

return binarySearch(arr, x, mid + 1, high)

else // x is on the left side

return binarySearch(arr, x, low, mid - 1)