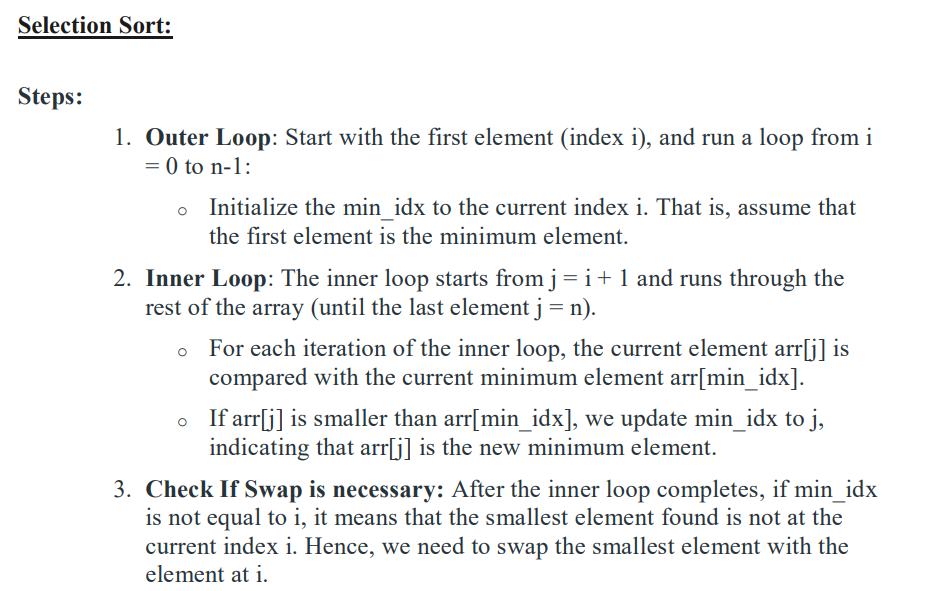
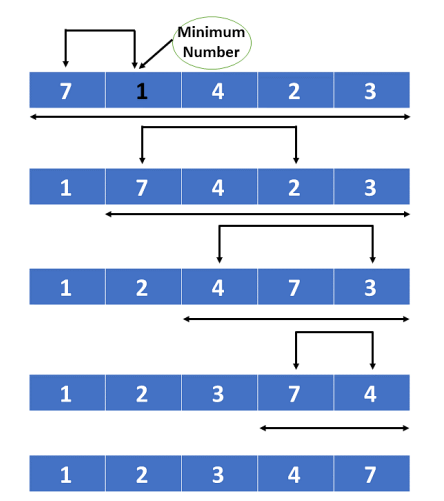
**Sorting Algorithms Notes**

****

****

**Selection Sort**

**Performance**

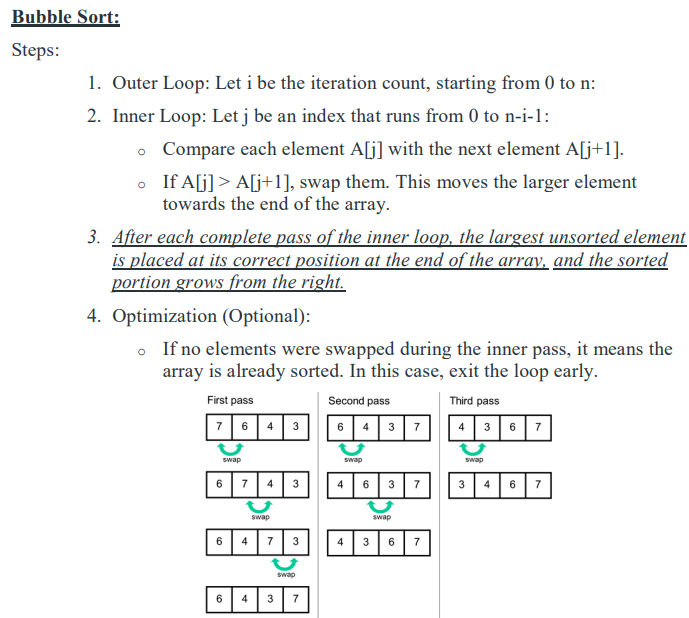
* **Data Size**: Selection Sort performs consistently with a time complexity of O(n2) regardless of dataset size. It's not efficient for large datasets.
* **Data Characteristics**: The performance does not significantly vary with the nature of the data (random, sorted, nearly sorted). It always performs O(n2) comparisons.

**Stability**

* **Stable Sorting**: Selection Sort is not stable. It may change the relative order of equal elements.

**Space Complexity**

* **In-Place Sorting**: Selection Sort is an in-place sorting algorithm. It requires a constant amount of additional memory (O(1)).



**Bubble Sort**

**Performance**

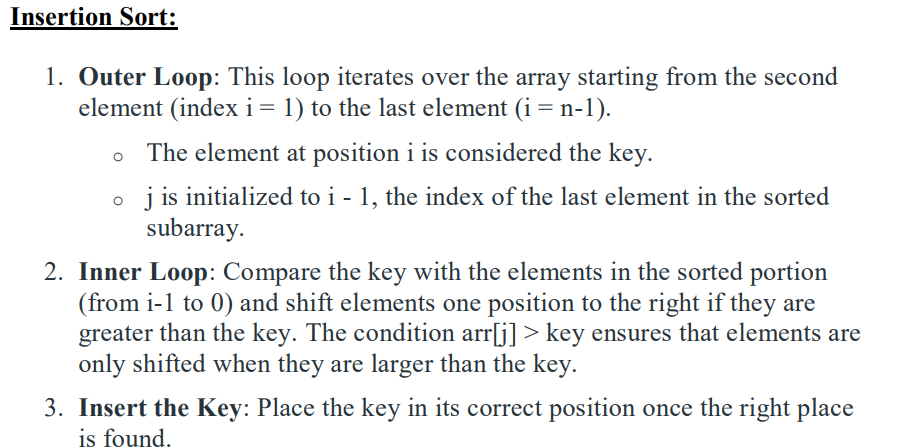
* **Data Size**: Bubble Sort also has a time complexity of O(n2). It is generally inefficient for large datasets.
* **Data Characteristics**: Bubble Sort performs better on nearly sorted datasets due to its optimization for detecting sorted sections. It can be improved with an optimized version that stops early if no swaps are made.

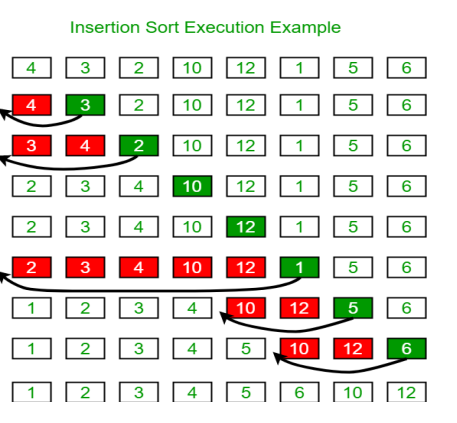
**Stability**

* **Stable Sorting**: Bubble Sort is stable. It preserves the relative order of equal elements.

**Space Complexity**

* **In-Place Sorting**: Bubble Sort is an in-place sorting algorithm. It requires a constant amount of additional memory (O(1)).





**Insertion Sort**

**Performance**

* **Data Size**: Insertion Sort has a time complexity of O(n2) in the average and worst cases, but it performs well for small datasets or nearly sorted data, with a best-case time complexity of O(n).
* **Data Characteristics**: It is particularly efficient for datasets that are already mostly sorted or small in size.

**Stability**

* **Stable Sorting**: Insertion Sort is stable. It maintains the relative order of equal elements.

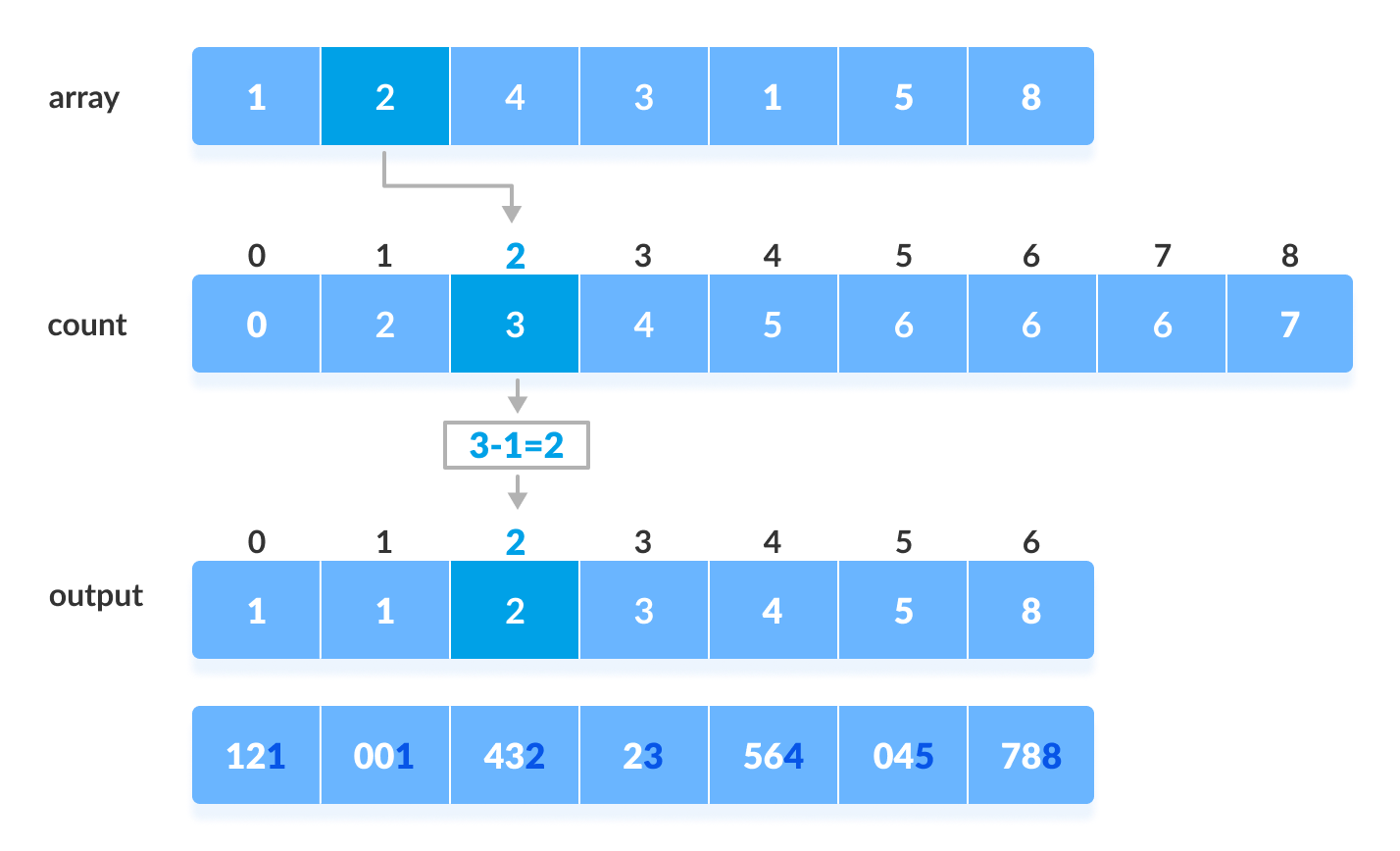
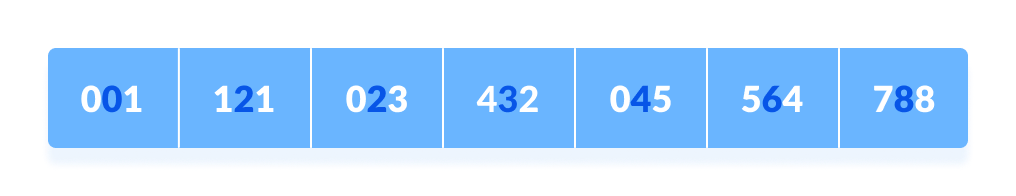
**Space Complexity**

* **In-Place Sorting**: Insertion Sort is an in-place sorting algorithm. It requires a constant amount of additional memory (O(1)).

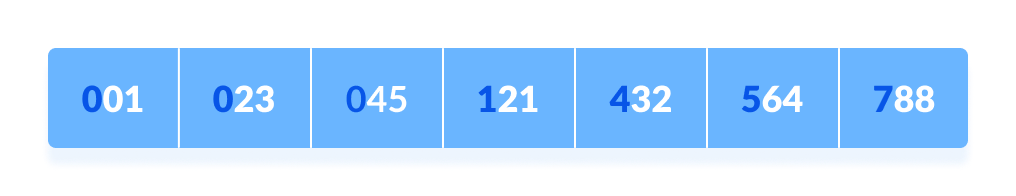
Radix Sort

1. Find the largest element in the array, i.e. max. Let X be the number of digits in max. X is calculated because we have to go through all the significant places of all elements.  
     
   In this array [121, 432, 564, 23, 1, 45, 788], we have the largest number 788. It has 3 digits. Therefore, the loop should go up to hundreds place (3 times).
2. Now, go through each significant place one by one.  
     
   Use any stable sorting technique to sort the digits at each significant place. We have used counting sort for this.  
     
   Sort the elements based on the unit place digits (X=0).

## Working of Radix Sort

1. Find the largest element in the array, i.e. max. Let X be the number of digits in max. X is calculated because we have to go through all the significant places of all elements.  
     
   In this array **[121, 432, 564, 23, 1, 45, 788]**, we have the largest number 788. It has 3 digits. Therefore, the loop should go up to hundreds place (3 times).
2. Now, go through each significant place one by one.  
     
   Use any stable sorting technique to sort the digits at each significant place. We have used counting sort for this.  
     
   Sort the elements based on the unit place digits (X=0).Using counting sort to sort elements based on unit place
3. Now, sort the elements based on digits at tens place.

Sort elements based on tens place

1. Finally, sort the elements based on the digits at hundreds place.

Sort elements based on hundreds place

**Radix Sort**

**Performance**

* **Data Size**: Radix Sort can be very efficient for large datasets, especially when the range of key values (digits) is small. Its time complexity is O(n⋅k), where k is the number of digits in the largest number.
* **Data Characteristics**: Radix Sort is most effective when dealing with integer keys or when the range of key values is limited. It does not compare elements directly but uses digit-based sorting.

**Stability**

* **Stable Sorting**: Radix Sort is stable. It uses stable sorting algorithms (like Counting Sort) as subroutines to maintain the relative order of equal elements.

**Space Complexity**

* **Non-In-Place Sorting**: Radix Sort is not in-place. It requires additional memory proportional to the size of the input data and the range of key values (O(n+k)).