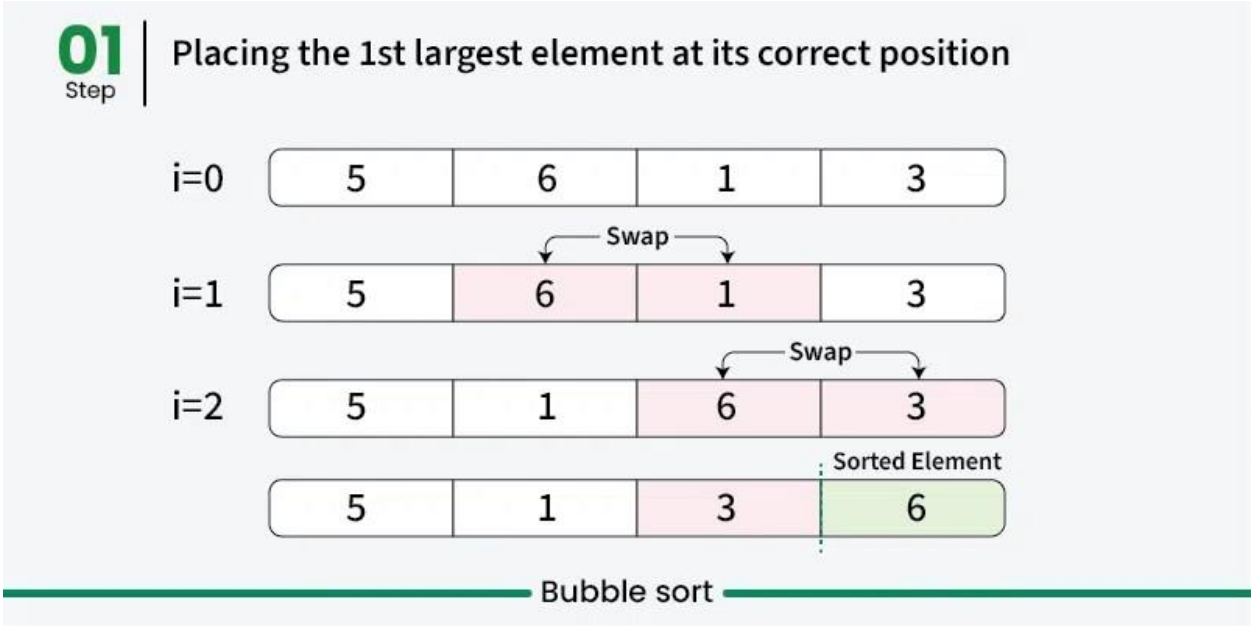
SORTING ALGORITHMS

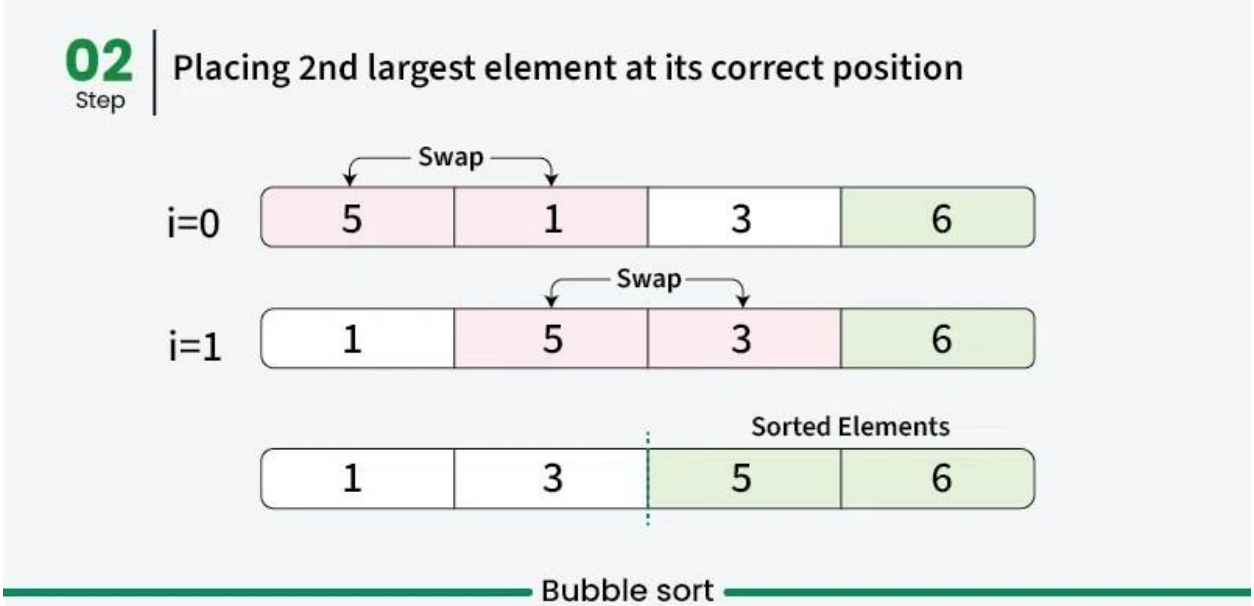
Bubble Sort Algorithm

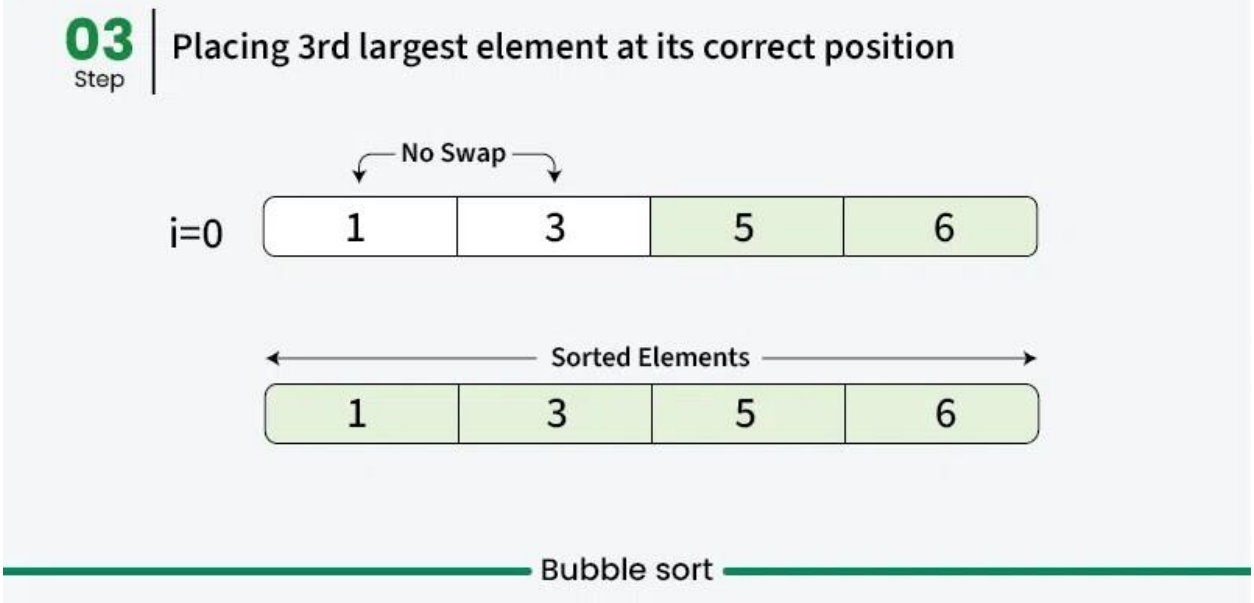
Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in the wrong order. This algorithm is not suitable for large data sets as its average and worst-case time complexity are quite high.

- We sort the array using multiple passes. After the first pass, the maximum element goes to end (its correct position). Same way, after second pass, the second largest element goes to second last position and so on.
- In every pass, we process only those elements that have already not moved to correct position. After k passes, the largest k elements must have been moved to the last k positions.
- In a pass, we consider remaining elements and compare all adjacent and swap if larger element is before a smaller element. If we keep doing this, we get the largest (among the remaining elements) at its correct position.

How does Bubble Sort Work?







Below is the implementation of the bubble sort. It can be optimized by stopping the algorithm if the inner loop didn't cause any swap.

```
// Optimized implementation of Bubble sort
#include <stdbool.h>
#include <stdio.h>

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;
    }
}

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

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
```

Complexity Analysis of Bubble Sort:

Time Complexity: $O(n^2)$ **Auxiliary Space:** $O(1)$

Advantages of Bubble Sort:

- Bubble sort is easy to understand and implement.
- It does not require any additional memory space.
- It is a stable sorting algorithm, meaning that elements with the same key value maintain their relative order in the sorted output.

Disadvantages of Bubble Sort:

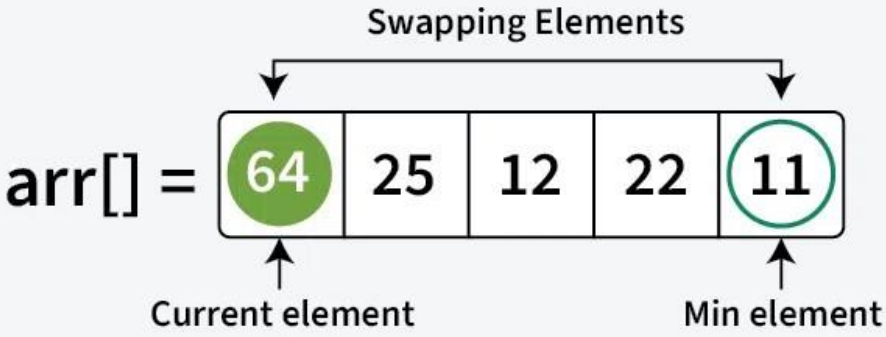
- Bubble sort has a time complexity of $O(n^2)$ which makes it very slow for large data sets.
- Bubble sort has almost no or limited real world applications. It is mostly used in academics to teach different ways of sorting.

Selection Sort

Selection Sort is a comparison-based sorting algorithm. It sorts an array by repeatedly selecting the **smallest (or largest)** element from the unsorted portion and swapping it with the first unsorted element. This process continues until the entire array is sorted.

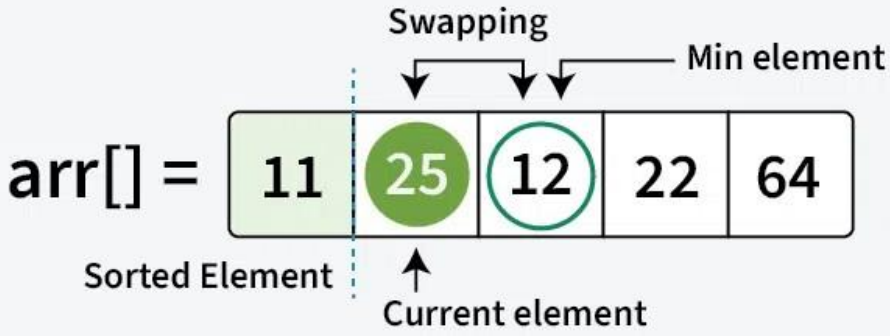
1. First we find the smallest element and swap it with the first element. This way we get the smallest element at its correct position.
2. Then we find the smallest among remaining elements (or second smallest) and swap it with the second element.
3. We keep doing this until we get all elements moved to correct position.

01 Step | Start from the first element at index 0, find the smallest element in the rest of the array which is unsorted, and swap (11) with current element(64).



Selection Sort Algorithm

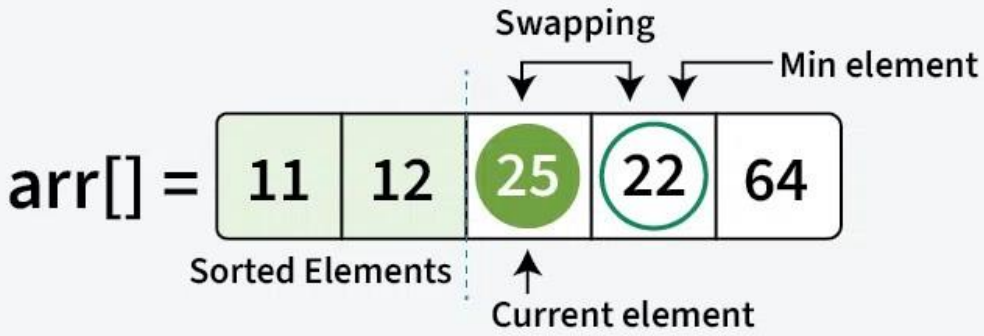
02 Step | Move to the next element at index 1 (25). Find the smallest in unsorted subarray, and swap (12) with current element (25).



Selection Sort Algorithm

03 Step

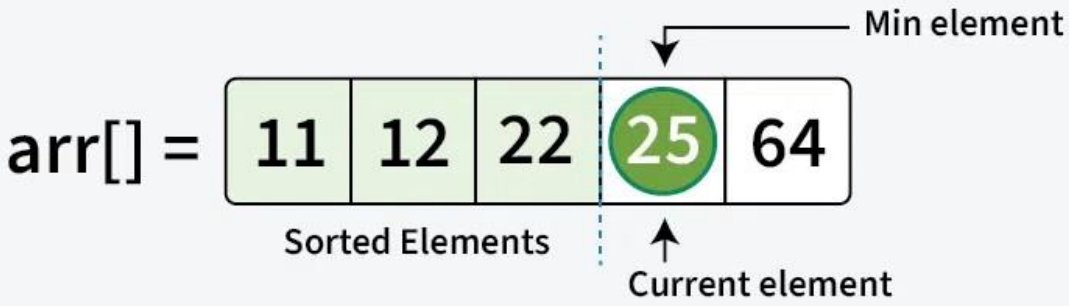
Move to element at index 2 (25). Find the minimum element from unsorted subarray, Swap (22) with current element (25).



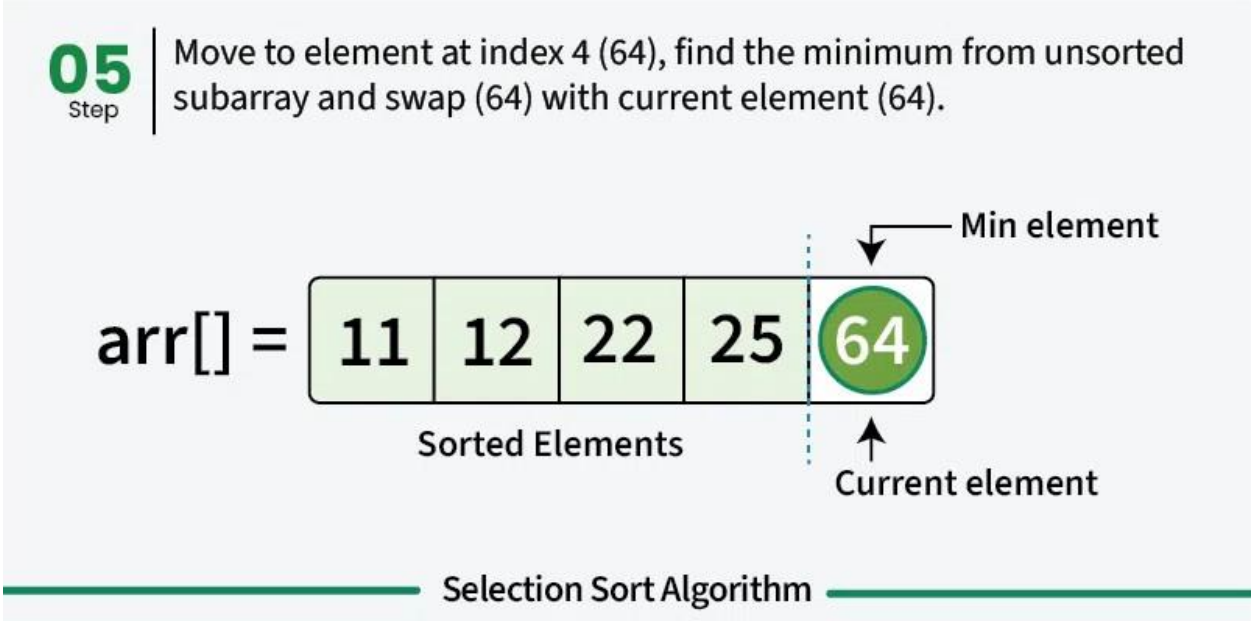
Selection Sort Algorithm

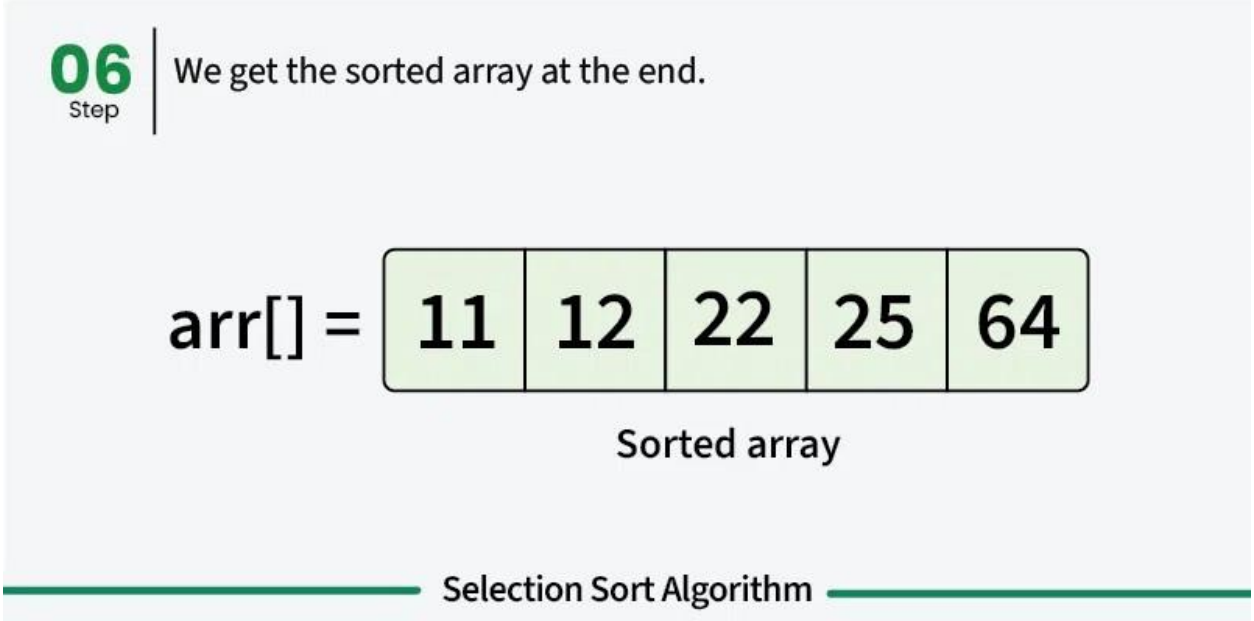
04 Step

Move to element at index 3 (25), find the minimum from unsorted subarray and swap (25) with current element (25).



Selection Sort Algorithm






```
// C program for implementation of selection sort
#include <stdio.h>

void selectionSort(int arr[], int n) {
   for (int i = 0; i < n - 1; i++) {

      // Assume the current position holds
      // the minimum element
      int min_idx = i;
```

```
        // Iterate through the unsorted portion
        // to find the actual minimum
        for (int j = i + 1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {

                // Update min_idx if a smaller element is found
                min_idx = j;
            }
        }

        // Move minimum element to its
        // correct position
        int temp = arr[i];
        arr[i] = arr[min_idx];
        arr[min_idx] = temp;
    }
}

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

int main() {
    int arr[] = {64, 25, 12, 22, 11};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Original array: ");
    printArray(arr, n);

    selectionSort(arr, n);

    printf("Sorted array: ");
    printArray(arr, n);

    return 0;
}
```

Output

```
Original vector: 64 25 12 22 11

Sorted vector:   11 12 22 25 64
```

Complexity Analysis of Selection Sort

Time Complexity: O(n^2) ,as there are two nested loops:

- One loop to select an element of Array one by one = O(n)
- Another loop to compare that element with every other Array element = O(n)
- Therefore overall complexity = O(n) * O(n) = O(n*n) = $O(n_2)$

Auxiliary Space: O(1) as the only extra memory used is for temporary variables.

Advantages of Selection Sort

- Easy to understand and implement, making it ideal for teaching basic sorting concepts.
- Requires only a constant O(1) extra memory space.
- It requires less number of swaps (or memory writes) compared to many other standard algorithms. Only cycle sort beats it in terms of memory writes. Therefore it can be simple algorithm choice when memory writes are costly.

Disadvantages of the Selection Sort

- Selection sort has a time complexity of O(n^2) makes it slower compared to algorithms like Quick Sort or Merge Sort.
- Does not maintain the relative order of equal elements which means it is not stable.

Applications of Selection Sort

- Perfect for teaching fundamental sorting mechanisms and algorithm design.
- Suitable for small lists where the overhead of more complex algorithms isn't justified and memory writing is costly as it requires less memory writes compared to other standard sorting algorithms.
- Heap Sort algorithm is based on Selection Sort.

Question 1: Is Selection Sort a stable sorting algorithm?
Answer: *No, Selection Sort is* ***not stable*** *as it may change the relative order of equal elements.*

Question 2: What is the time complexity of Selection Sort?
Answer: *Selection Sort has a time complexity of O(n^2) in the best, average, and worst cases.*

Question 3: Does Selection Sort require extra memory?
Answer: *No, Selection Sort is an in-place sorting algorithm and requires only O(1) additional space.*

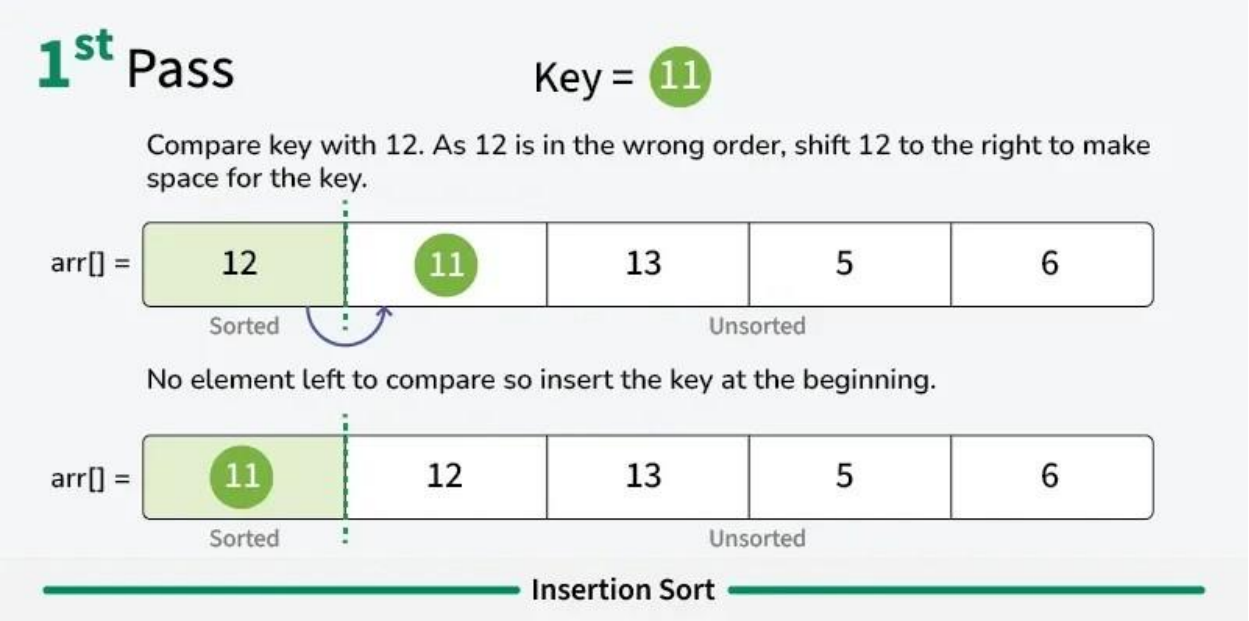
Question 4: When is it best to use Selection Sort?
Answer: *Selection Sort is best used for small datasets, educational purposes, or when memory usage needs to be minimal.*

Question 5: How does Selection Sort differ from Bubble Sort?
Answer: *Selection Sort selects the minimum element and places it in the correct position with fewer swaps, while Bubble Sort repeatedly swaps adjacent elements to sort the array.*

Insertion Sort Algorithm

Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

- We start with the second element of the array as the first element is assumed to be sorted.
- Compare the second element with the first element if the second element is smaller then swap them.
- Move to the third element, compare it with the first two elements, and put it in its correct position
- Repeat until the entire array is sorted.



2nd Pass

Key = 13

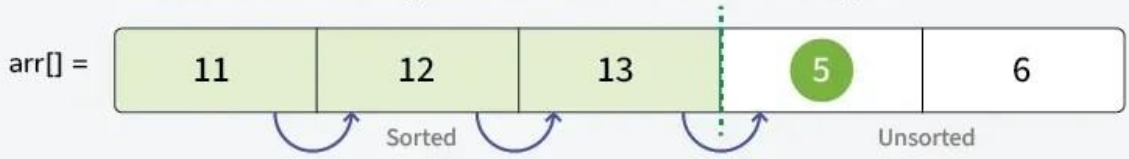
Compare key with 12, 12 is in the correct order, so no changes take place.

Insertion Sort

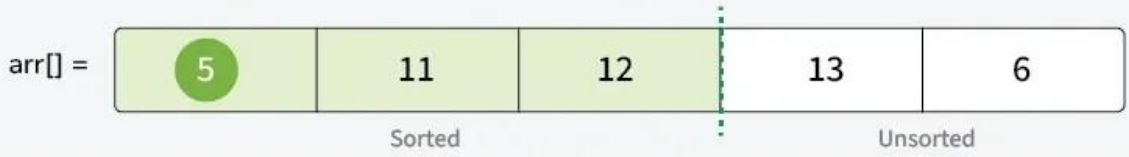
3rd Pass

Key = 5

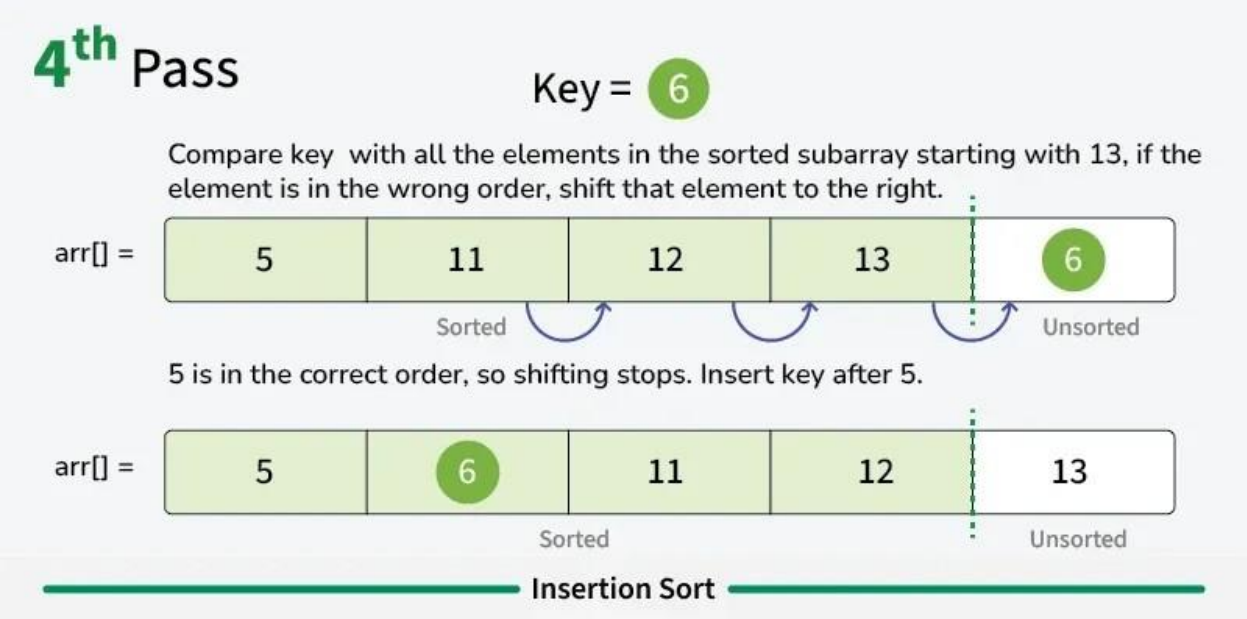
Compare key with all the elements in the sorted subarray starting with 13, if the element is in the wrong order, shift that element to the right.

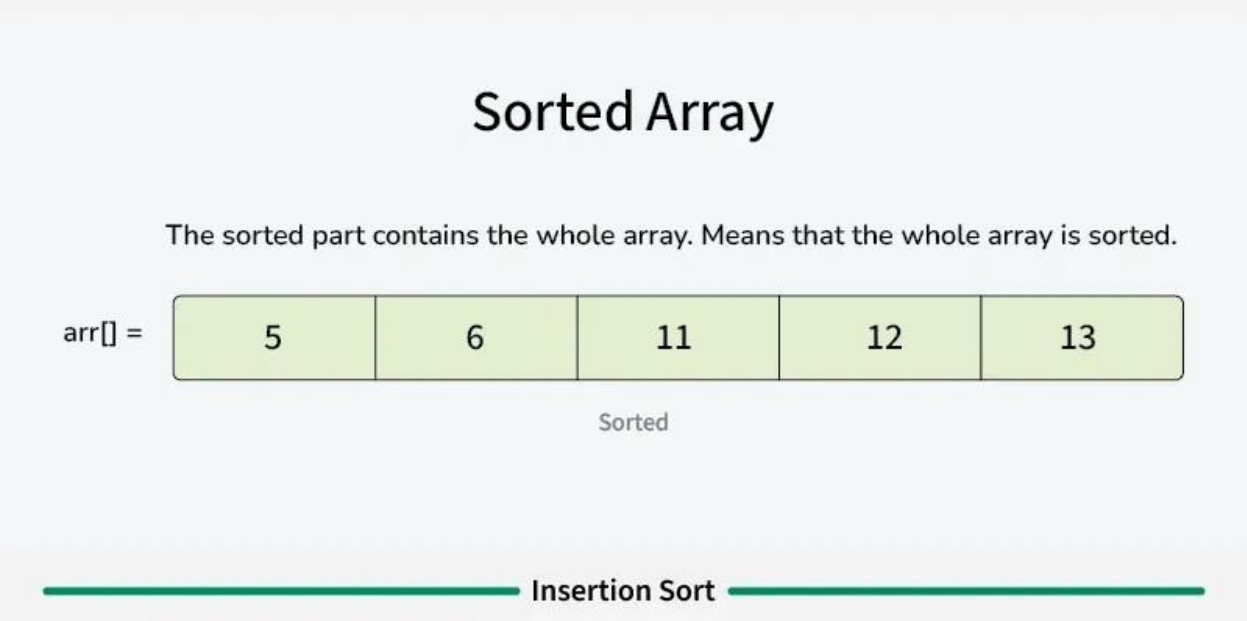


No element left to compare, so insert key at the beginning.



Insertion Sort





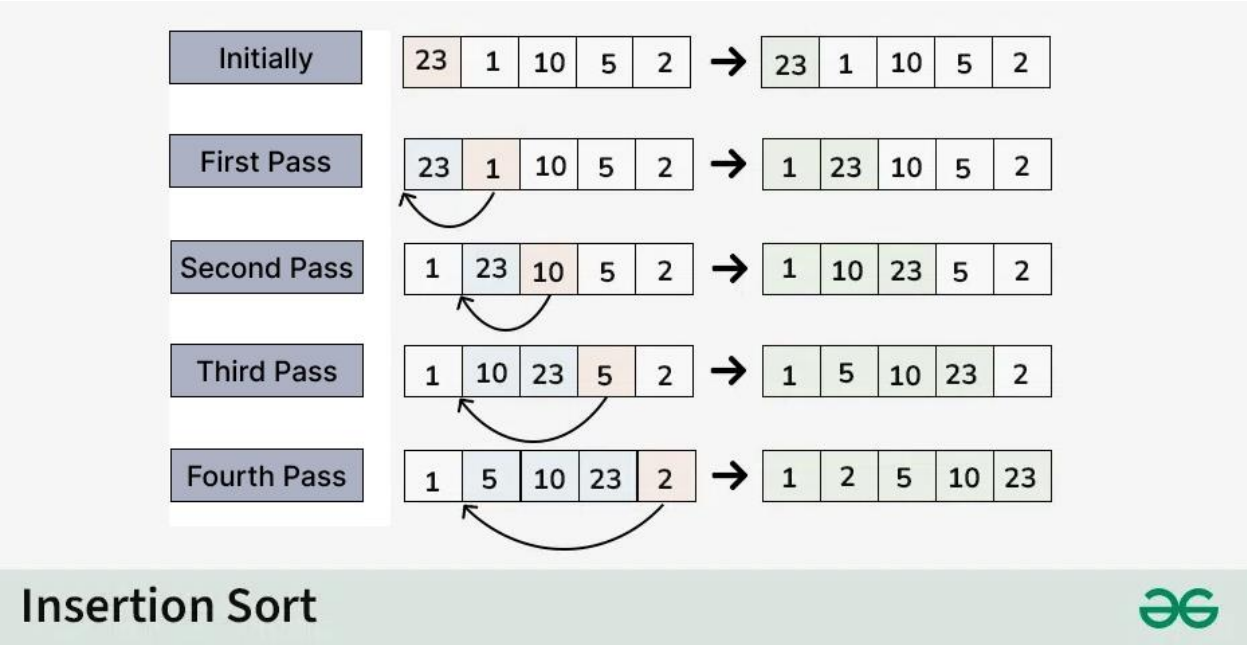

```
// C program for implementation of Insertion Sort
#include <stdio.h>

/* Function to sort array using insertion sort */
void insertionSort(int arr[], int n)
{
   for (int i = 1; i < n; ++i) {
      int key = arr[i];
      int j = i - 1;
```

```
        /* Move elements of arr[0..i-1], that are
          greater than key, to one position ahead
          of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}

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

// Driver method
int main()
{
    int arr[] = { 12, 11, 13, 5, 6 };
    int n = sizeof(arr) / sizeof(arr[0]);

    insertionSort(arr, n);
    printArray(arr, n);

    return 0;
}

/* This code is contributed by Hritik Shah. */
```

Output

```
5 6 11 12 13
```

Illustration



arr = {23, 1, 10, 5, 2}

Initial:

- *Current element is **23***
- *The first element in the array is assumed to be sorted.*
- *The sorted part until **0th** index is : **[23]***

First Pass:

- *Compare **1** with **23** (current element with the sorted part).*
- *Since **1** is smaller, insert **1** before **23** .*
- *The sorted part until **1st** index is: **[1, 23]***

Second Pass:

- *Compare **10** with **1** and **23** (current element with the sorted part).*
- *Since **10** is greater than **1** and smaller than **23** , insert **10** between **1** and **23** .*
- *The sorted part until **2nd** index is: **[1, 10, 23]***

Third Pass:

- *Compare **5** with **1** , **10** , and **23** (current element with the sorted part).*
- *Since **5** is greater than **1** and smaller than **10** , insert **5** between **1** and **10***
- *The sorted part until **3rd** index is **: [1, 5, 10, 23]***

Fourth Pass:

- *Compare **2** with **1, 5, 10** , and **23** (current element with the sorted part).*
- *Since **2** is greater than **1** and smaller than **5** insert **2** between **1** and **5** .*
- *The sorted part until **4th** index is: **[1, 2, 5, 10, 23]***

Final Array:

- *The sorted array is: **[1, 2, 5, 10, 23]***

Complexity Analysis of Insertion Sort

Time Complexity

- **Best case: O(n)**, If the list is already sorted, where n is the number of elements in the list.
- **Average case: O(n^2)**, If the list is randomly ordered
- **Worst case: O(n^2)**, If the list is in reverse order

Space Complexity

- **Auxiliary Space:** O(1), Insertion sort requires **O(1)** additional space, making it a space-efficient sorting algorithm.

Advantages and Disadvantages of Insertion Sort

Advantages

- Simple and easy to implement.
- **Stable** sorting algorithm.
- Efficient for small lists and nearly sorted lists.
- Space-efficient as it is an in-place algorithm.
- Adoptive. the number of inversions is directly proportional to number of swaps. For example, no swapping happens for a sorted array and it takes O(n) time only.

Disadvantages

- Inefficient for large lists.
- Not as efficient as other sorting algorithms (e.g., merge sort, quick sort) for most cases.

Applications of Insertion Sort

Insertion sort is commonly used in situations where:

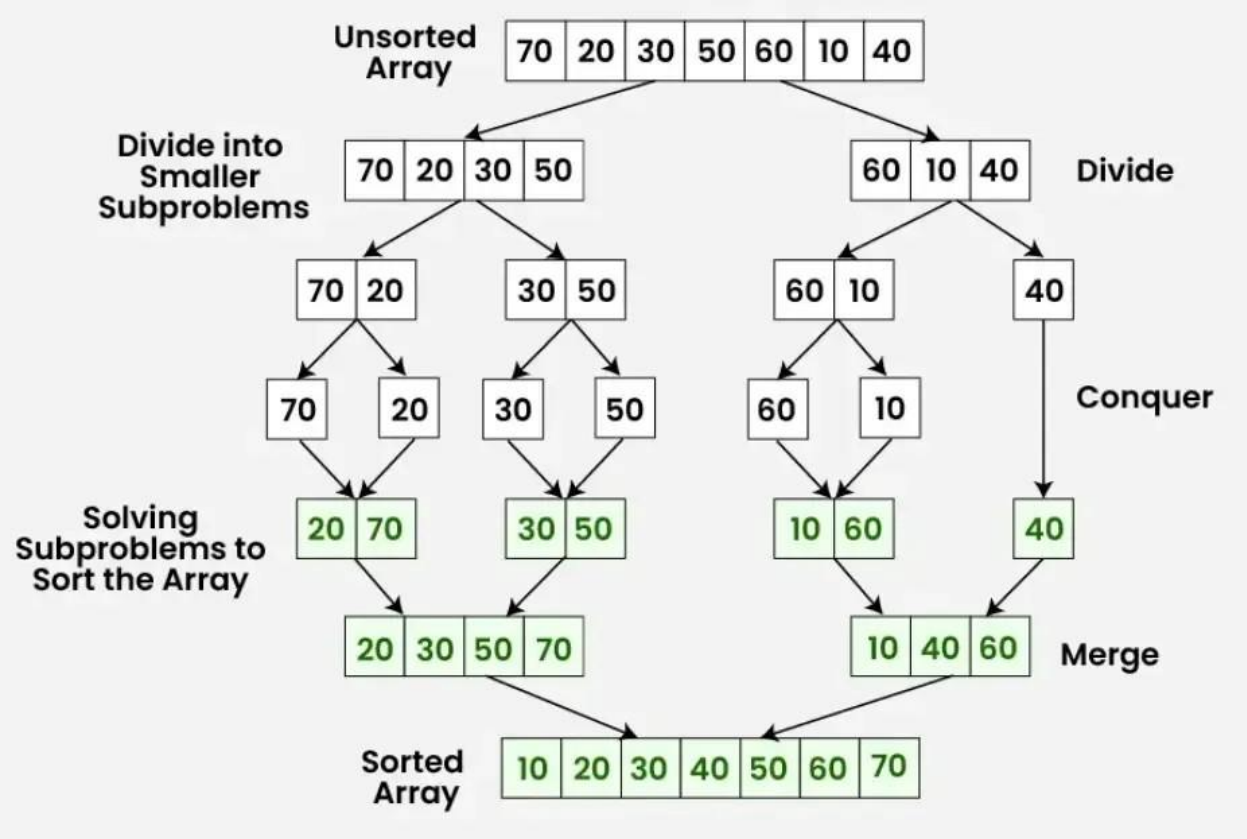
- The list is small or nearly sorted.
- Simplicity and stability are important.

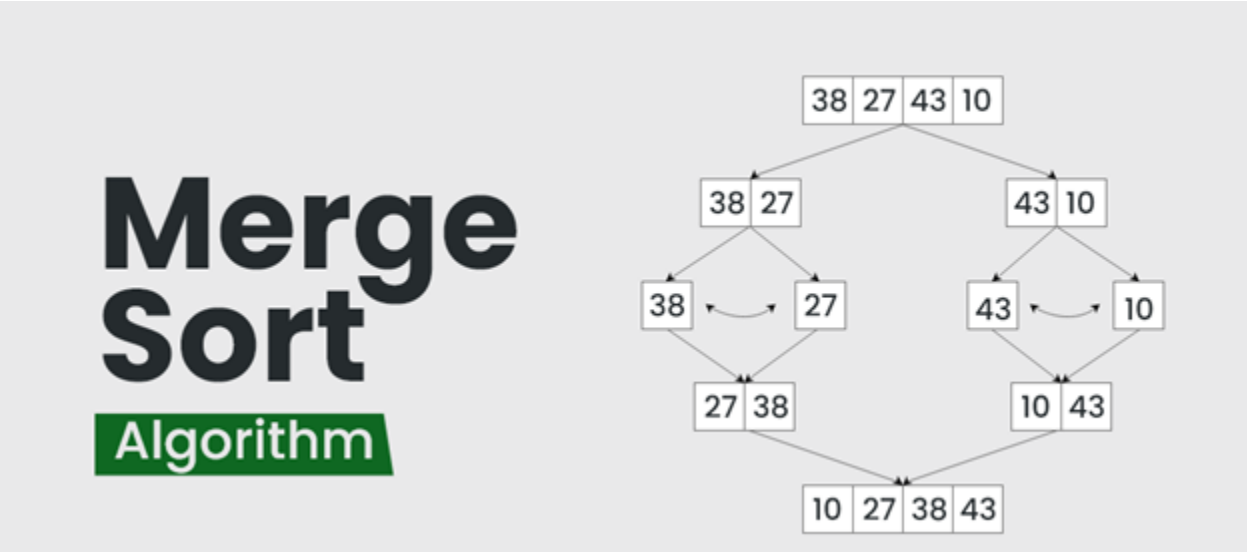
- Used as a subroutine in Bucket Sort
- Can be useful when array is already almost sorted (very few inversions)
- Since Insertion sort is suitable for small sized arrays, it is used in Hybrid Sorting algorithms along with other efficient algorithms like Quick Sort and Merge Sort. When the subarray size becomes small, we switch to insertion sort in these recursive algorithms. For example IntroSort and TimSort use insertions sort.

What are the Boundary Cases of the Insertion Sort algorithm?
Insertion sort takes the maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.
What is the Algorithmic Paradigm of the Insertion Sort algorithm?
The Insertion Sort algorithm follows an incremental approach.
Is Insertion Sort an in-place sorting algorithm?
Yes, insertion sort is an in-place sorting algorithm.
Is Insertion Sort a stable algorithm?
Yes, insertion sort is a stable sorting algorithm.
When is the Insertion Sort algorithm used?
Insertion sort is used when number of elements is small. It can also be useful when the input array is almost sorted, and only a few elements are misplaced in a complete big array.

Merge Sort

Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the **divide-and-conquer** approach. It works by recursively dividing the input array into two halves, recursively sorting the two halves and finally merging them back together to obtain the sorted array.





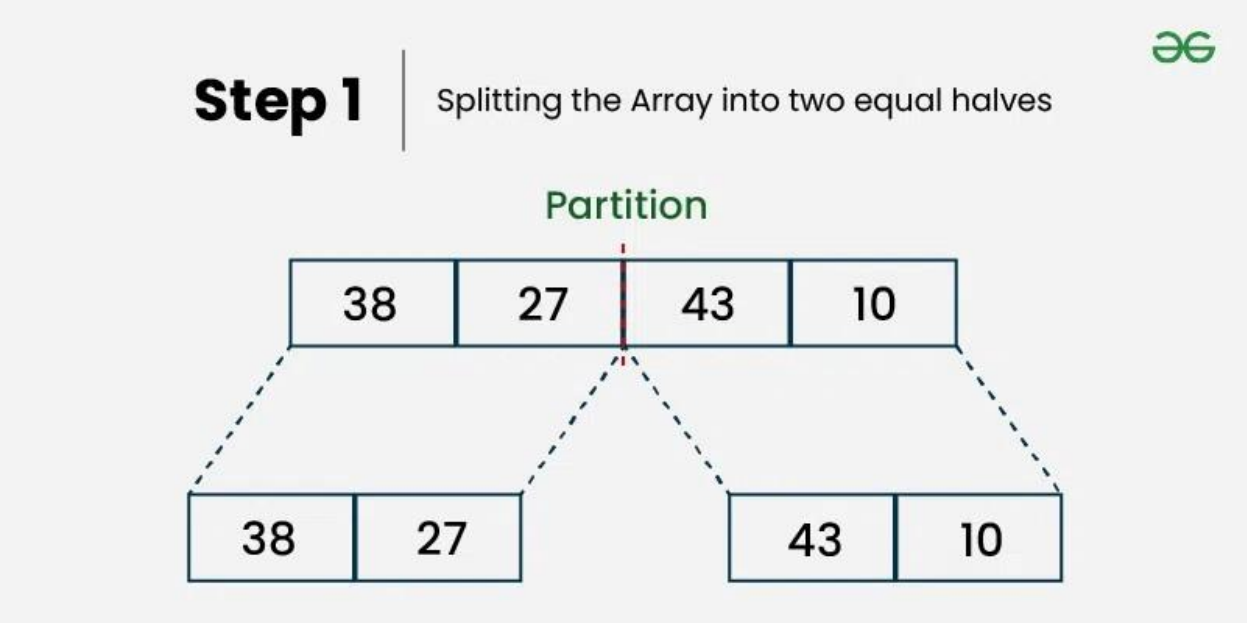
How does Merge Sort work?

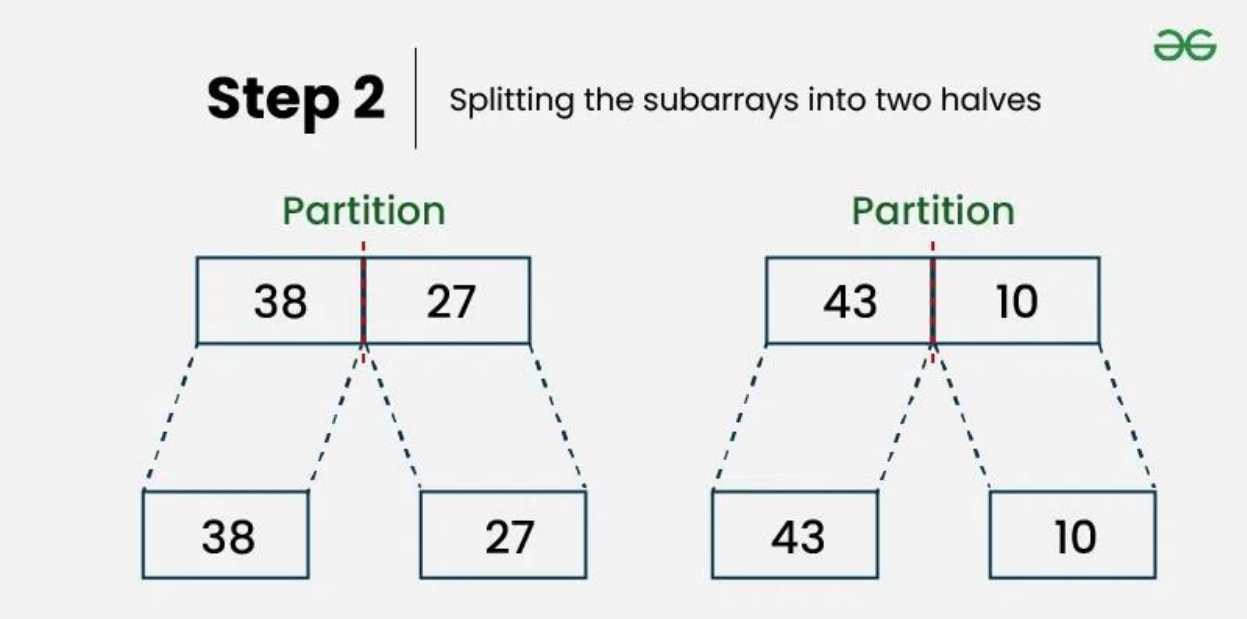
Here's a step-by-step explanation of how merge sort works:

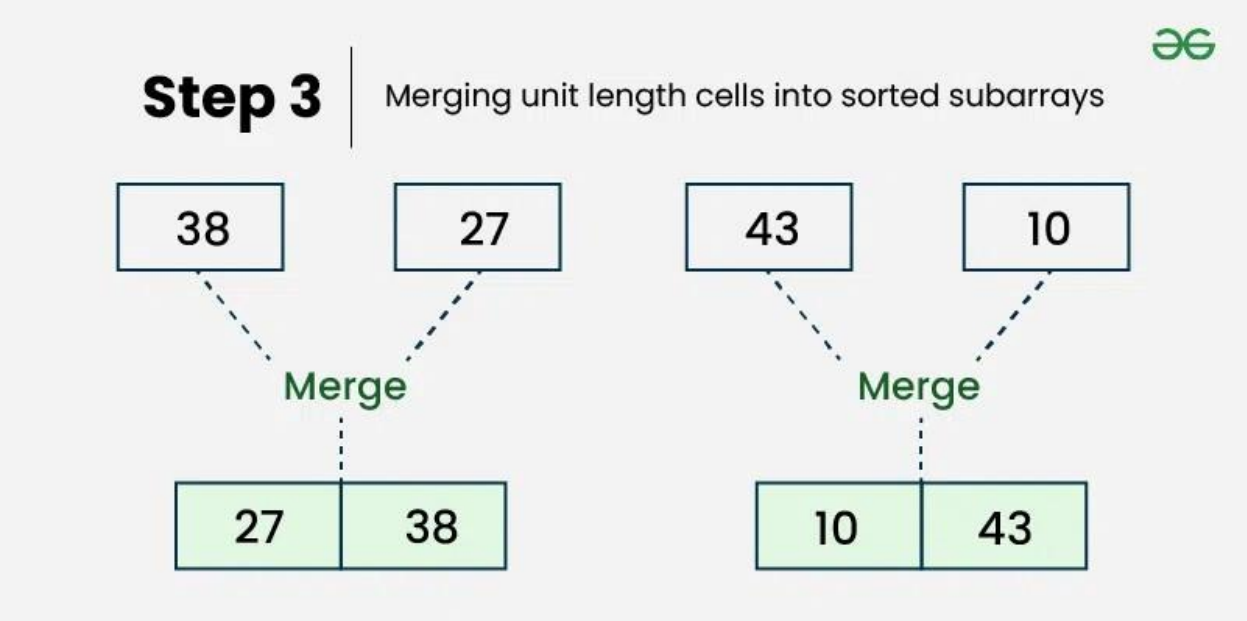
1. ***Divide:*** *Divide the list or array recursively into two halves until it can no more be divided.*
2. ***Conquer:*** *Each subarray is sorted individually using the merge sort algorithm.*
3. ***Merge:*** *The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.*

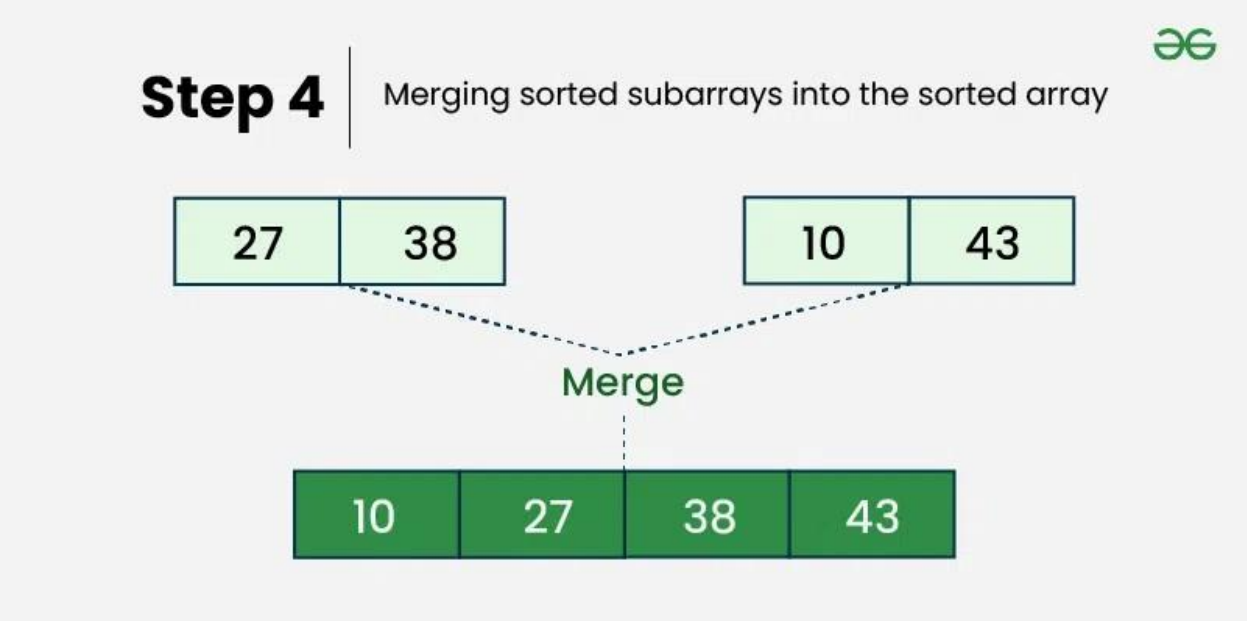
Illustration of Merge Sort:

Let's sort the array or list **[38, 27, 43, 10]** using Merge Sort




Step 2
Splitting the subarrays into two halves
Partition
38
27
38
27
Partition
43
10
43
10


Step 3
Merging unit length cells into sorted subarrays
38
27
Merge
27
38
43
10
Merge
10
43



Let's look at the working of above example:

Divide:

- ***[38, 27, 43, 10]*** *is divided into* ***[38, 27]*** *and* ***[43, 10]*** *.*
- ***[38, 27]*** *is divided into* ***[38]*** *and* ***[27]*** *.*
- ***[43, 10]*** *is divided into* ***[43]*** *and* ***[10]*** *.*

Conquer:

- ***[38]*** *is already sorted.*
- ***[27]*** *is already sorted.*
- ***[43]*** *is already sorted.*
- ***[10]*** *is already sorted.*

Merge:

- *Merge* ***[38]*** *and* ***[27]*** *to get* ***[27, 38]*** *.*
- *Merge* ***[43]*** *and* ***[10]*** *to get* ***[10,43]*** *.*
- *Merge* ***[27, 38]*** *and* ***[10,43]*** *to get the final sorted list* ***[10, 27, 38, 43]***

Therefore, the sorted list is ***[10, 27, 38, 43]*** *.*

```
#include <stdio.h>
#include <stdlib.h>

// Merges two subarrays of arr[].
```

```
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r){

   int i, j, k;
   int n1 = m - l + 1;
   int n2 = r - m;

   // Create temp arrays
   int L[n1], R[n2];

   // Copy data to temp arrays L[] and R[]
   for (i = 0; i < n1; i++)
      L[i] = arr[l + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[m + 1 + j];

   // Merge the temp arrays back into arr[l..r
   i = 0;
   j = 0;
   k = l;
   while (i < n1 && j < n2) {
      if (L[i] <= R[j]) {
         arr[k] = L[i];
         i++;
      }
      else {
         arr[k] = R[j];
         j++;
      }
      k++;
```

```
    }

    // Copy the remaining elements of L[],
    // if there are any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of R[],
    // if there are any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

// l is for left index and r is right index of the
// sub-array of arr to be sorted
void mergeSort(int arr[], int l, int r){

    if (l < r) {
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
```

```
    }
}

// Driver code
int main(){

    int arr[] = {38, 27, 43, 10};
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    mergeSort(arr, 0, arr_size - 1);
    int i;
    for (i = 0; i < arr_size; i++)
        printf("%d ", arr[i]);
    printf("\n");

    return 0;
}
```

Output

```
10 27 38 43
```

Recurrence Relation of Merge Sort

The recurrence relation of merge sort is:

$$T(n)=\{\Theta(1) \text{if } n=1 2T(n2)+\Theta(n) \text{if } n>1 T(n)=\{\Theta(1) 2T(2n)+\Theta(n) \text{if } n=1 \text{if } n>1$$

- T(n) Represents the total time time taken by the algorithm to sort an array of size n.
- 2T(n/2) represents time taken by the algorithm to recursively sort the two halves of the array. Since each half has n/2 elements, we have two recursive calls with input size as (n/2).
- O(n) represents the time taken to merge the two sorted halves

Complexity Analysis of Merge Sort

- **Time Complexity:**

    - **Best Case:** O(n log n), When the array is already sorted or nearly sorted.
    - **Average Case:** O(n log n), When the array is randomly ordered.
    - **Worst Case:** O(n log n), When the array is sorted in reverse order.
- **Auxiliary Space:** O(n), Additional space is required for the temporary array used during merging.

Applications of Merge Sort:

- Sorting large datasets
- External sorting (when the dataset is too large to fit in memory)
- Inversion counting
- Merge Sort and its variations are used in library methods of programming languages.
    - Its variation TimSort is used in Python, Java Android and Swift. The main reason why it is preferred to sort non-primitive types is stability which is not there in QuickSort.
    - Arrays.sort in Java uses QuickSort while Collections.sort uses MergeSort.
- It is a preferred algorithm for sorting Linked lists.
- It can be easily parallelized as we can independently sort subarrays and then merge.
- The merge function of merge sort to efficiently solve the problems like union and intersection of two sorted arrays.

Advantages and Disadvantages of Merge Sort

Advantages

- **Stability** : Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
- **Guaranteed worst-case performance:** Merge sort has a worst-case time complexity of **O(N logN)** , which means it performs well even on large datasets.
- **Simple to implement:** The divide-and-conquer approach is straightforward.

- **Naturally Parallel** : We independently merge subarrays that makes it suitable for parallel processing.

Disadvantages

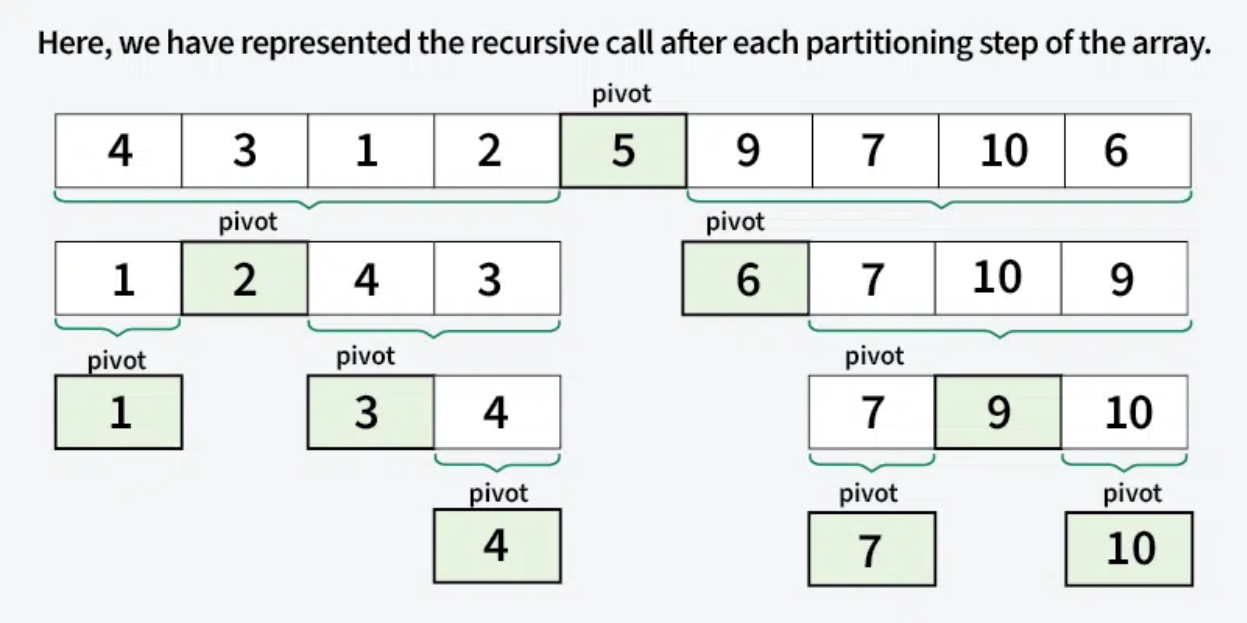
- **Space complexity:** Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
- **Not in-place:** Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
- Merge Sort is **Slower than QuickSort in general as** QuickSort is more cache friendly because it works in-place.

Quick Sort

QuickSort is a sorting algorithm based on the Divide and Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.
It works on the principle of **divide and conquer**, breaking down the problem into smaller sub-problems.
There are mainly three steps in the algorithm:

1. **Choose a Pivot:** Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).
2. **Partition the Array:** Re arrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right. The pivot is then in its correct position, and we obtain the index of the pivot.
3. **Recursively Call:** Recursively apply the same process to the two partitioned sub-arrays (left and right of the pivot).
4. **Base Case:** The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.

Here's a basic overview of how the QuickSort algorithm works.



Choice of Pivot

There are many different choices for picking pivots.

- Always pick the first (or last) element as a pivot. The below implementation picks the last element as pivot. The problem with this approach is it ends up in the worst case when array is already sorted.
- Pick a random element as a pivot. This is a preferred approach because it does not have a pattern for which the worst case happens.
- Pick the median element is pivot. This is an ideal approach in terms of time complexity as we can find median in linear time and the partition function will always divide the input array into two halves. But it takes more time on average as median finding has high constants.

Partition Algorithm

The key process in **quickSort** is a **partition().** There are three common algorithms to partition. All these algorithms have O(n) time complexity.

1. **Naive Partition**: Here we create copy of the array. First put all smaller elements and then all greater. Finally we copy the

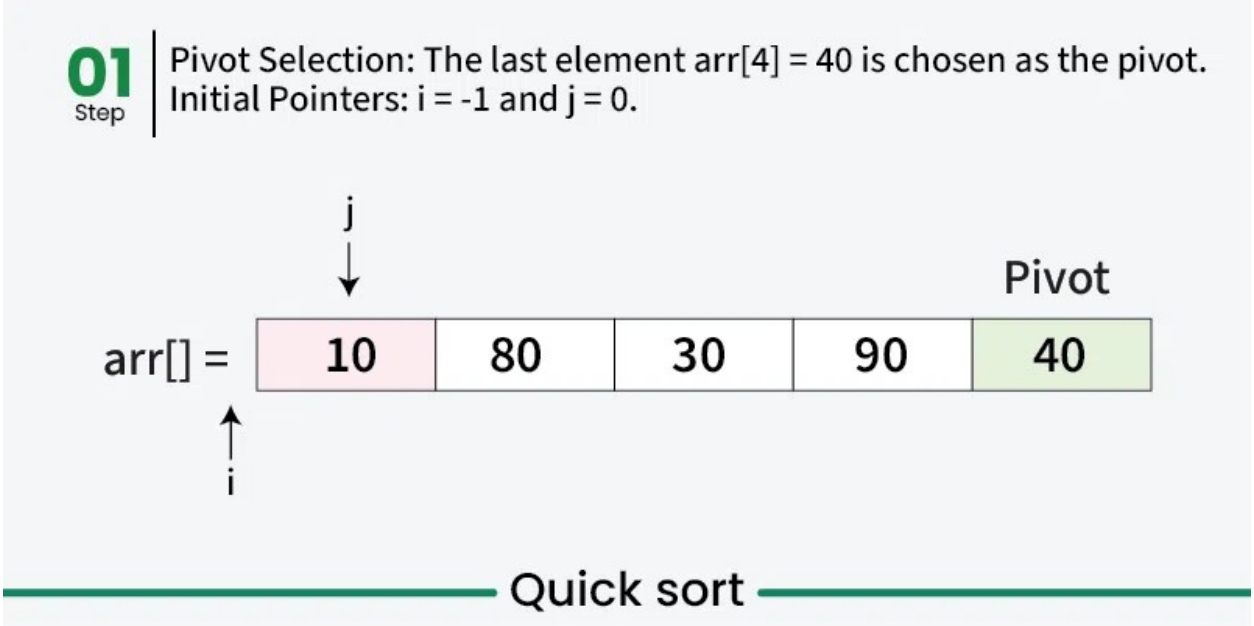
temporary array back to original array. This requires O(n) extra space.

2. **Lomuto Partition**: We have used this partition in this article. This is a simple algorithm, we keep track of index of smaller elements and keep swapping. We have used it here in this article because of its simplicity.
3. **Hoare's Partition**: This is the fastest of all. Here we traverse array from both sides and keep swapping greater element on left with smaller on right while the array is not partitioned. Please refer Hoare's vs Lomuto for details.

Working of Lomuto Partition Algorithm with Illustration

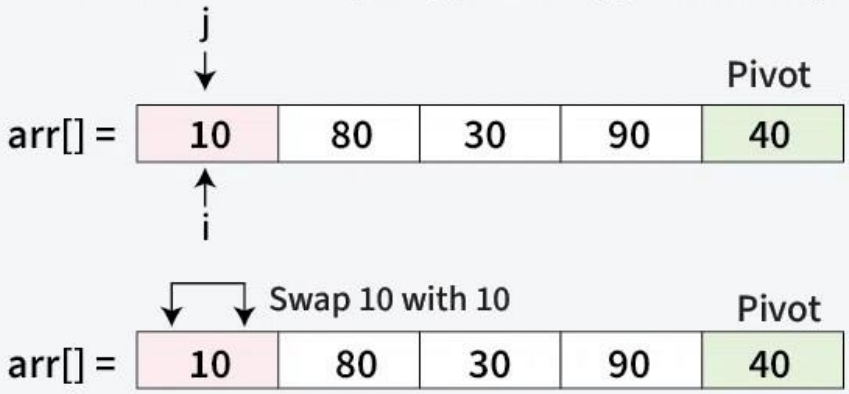
*The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal) elements as **i** . While traversing, if we find a smaller element, we swap the current element with **arr[i]**. Otherwise, we ignore the current element.*

Let us understand the working of partition algorithm with the help of the following example:



02 Step

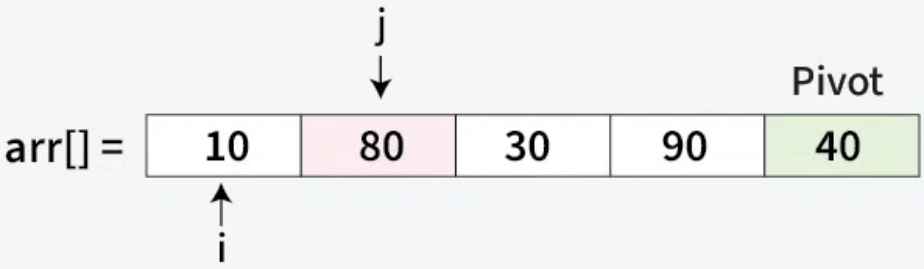
Since, arr[j] < pivot (10<40)
Increment i to 0 and swap arr[i] with arr[j]. Increment j by 1



Quick sort

03 Step

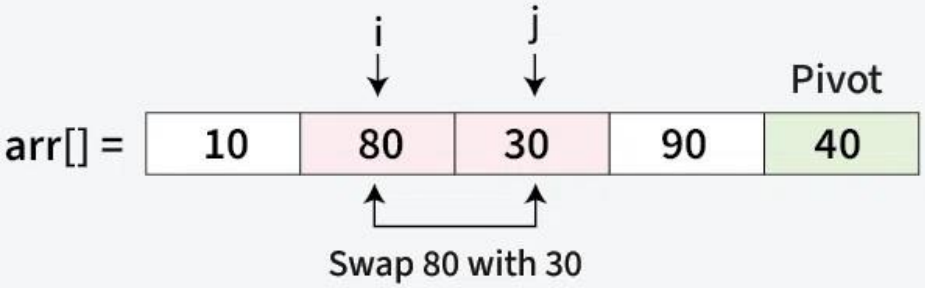
Since, arr[j] > pivot (80>40)
No swap needed. Increment j by 1



Quick sort

04 Step

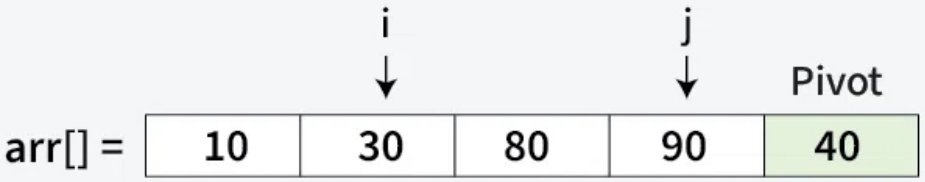
Since, arr[j] < pivot (30<40)
Increment i by 1 and swap arr[i] with arr[j]. Increment j by 1



Quick sort

05 Step

Since, arr[j] > pivot (90>40)
No swap needed. Increment j by 1



Quick sort

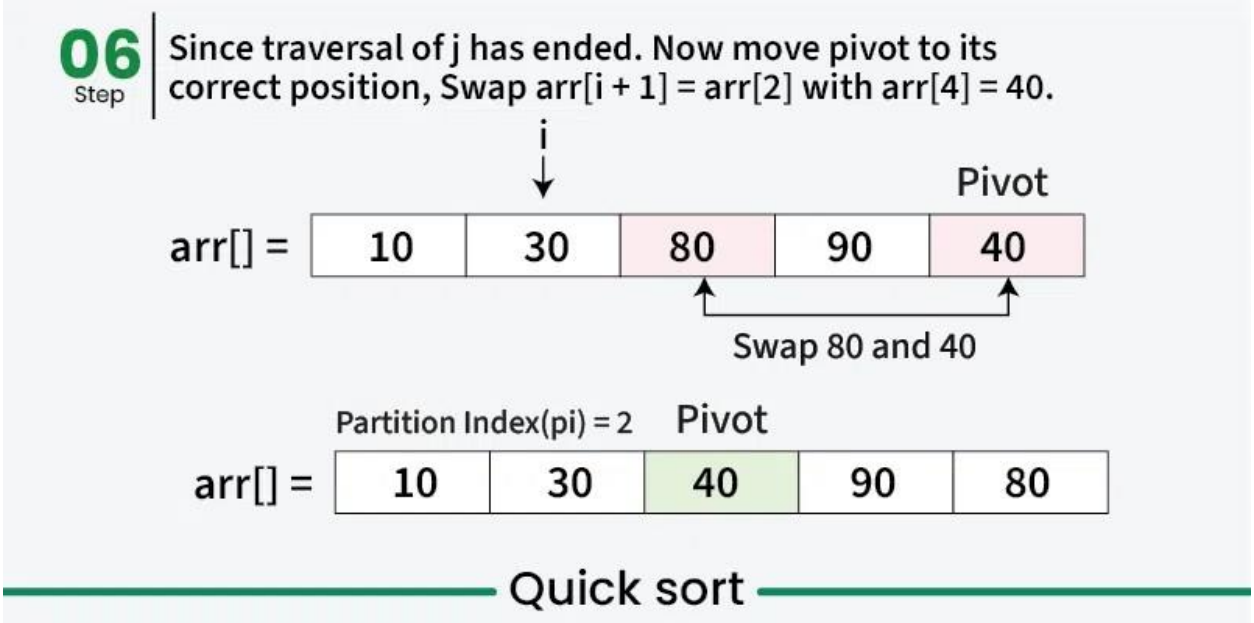
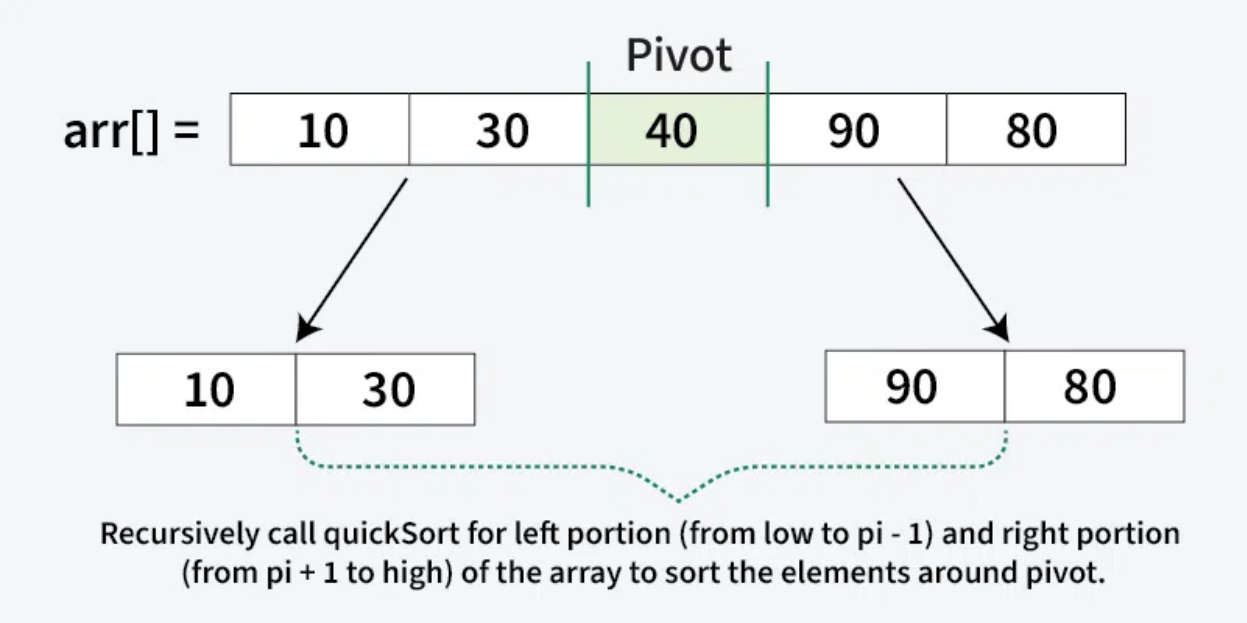


Illustration of QuickSort Algorithm

In the previous step, we looked at how the **partitioning** process rearranges the array based on the chosen **pivot**. Next, we apply the same method recursively to the smaller sub-arrays on the **left** and **right** of the pivot. Each time, we select new pivots and partition the arrays again. This process continues until only one element is left, which is always sorted. Once every element is in its correct position, the entire array is sorted.
Below image illustrates, how the recursive method calls for the smaller sub-arrays on the **left** and **right** of the **pivot**:




```
#include <stdio.h>

void swap(int* a, int* b);

// partition function
int partition(int arr[], int low, int high) {

    // Choose the pivot
    int pivot = arr[high];

    // Index of smaller element and indicates
    // the right position of pivot found so far
    int i = low - 1;

    // Traverse arr[low..high] and move all smaller
    // elements to the left side. Elements from low to
    // i are smaller after every iteration
```

```
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }

    // Move pivot after smaller elements and
    // return its position
    swap(&arr[i + 1], &arr[high]);
    return i + 1;
}

// The QuickSort function implementation
void quickSort(int arr[], int low, int high) {
    if (low < high) {

        // pi is the partition return index of pivot
        int pi = partition(arr, low, high);

        // recursion calls for smaller elements
        // and greater or equals elements
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

void swap(int* a, int* b) {
    int t = *a;
    *a = *b;
    *b = t;
```

```
}

int main() {
    int arr[] = {10, 7, 8, 9, 1, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    quickSort(arr, 0, n - 1);
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

Output

```
1 5 7 8 9 10
```

Complexity Analysis of Quick Sort

Time Complexity:

- **Best Case:** (Ω(n log n)), Occurs when the pivot element divides the array into two equal halves.
- **Average Case** (θ(n log n)), On average, the pivot divides the array into two parts, but not necessarily equal.
- **Worst Case:** (O(n^2)), Occurs when the smallest or largest element is always chosen as the pivot (e.g., sorted arrays).

Auxiliary Space:

- **Worst-case scenario: O(n)** due to unbalanced partitioning leading to a skewed recursion tree requiring a call stack of size O(n).
- **Best-case scenario: O(log n)** as a result of balanced partitioning leading to a balanced recursion tree with a call stack of size O(log n).

Advantages of Quick Sort

- It is a divide-and-conquer algorithm that makes it easier to solve problems.
- It is efficient on large data sets.
- It has a low overhead, as it only requires a small amount of memory to function.
- It is Cache Friendly as we work on the same array to sort and do not copy data to any auxiliary array.
- Fastest general purpose algorithm for large data when stability is not required.
- It is **tail recursive** and hence all the tail call optimization can be done.

Disadvantages of Quick Sort

- It has a worst-case time complexity of $O(n^2)$, which occurs when the pivot is chosen poorly.
- It is not a good choice for small data sets.
- It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

Applications of Quick Sort

- Sorting large datasets efficiently in memory.
- Used in library sort functions (like C++ std::sort and Java Arrays.sort for primitives).
- Arranging records in databases for faster searching.
- Preprocessing step in algorithms requiring sorted input (e.g., binary search, two-pointer techniques).
- Finding the kth smallest/largest element using Quickselect (a variant of quicksort).
- Sorting arrays of objects based on multiple keys (custom comparators).
- Data compression algorithms (like Huffman coding preprocessing).
- Graphics and computational geometry (e.g., convex hull algorithms).

Radix Sort

Radix Sort is a linear sorting algorithm (for fixed length digit counts) that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys.

- It repeatedly distributes the elements into buckets based on each digit's value. This is different from other algorithms like Merge Sort or Quick Sort where we compare elements directly.
- By repeatedly sorting the elements by their significant digits, from the least significant to the most significant, it achieves the final sorted order.

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

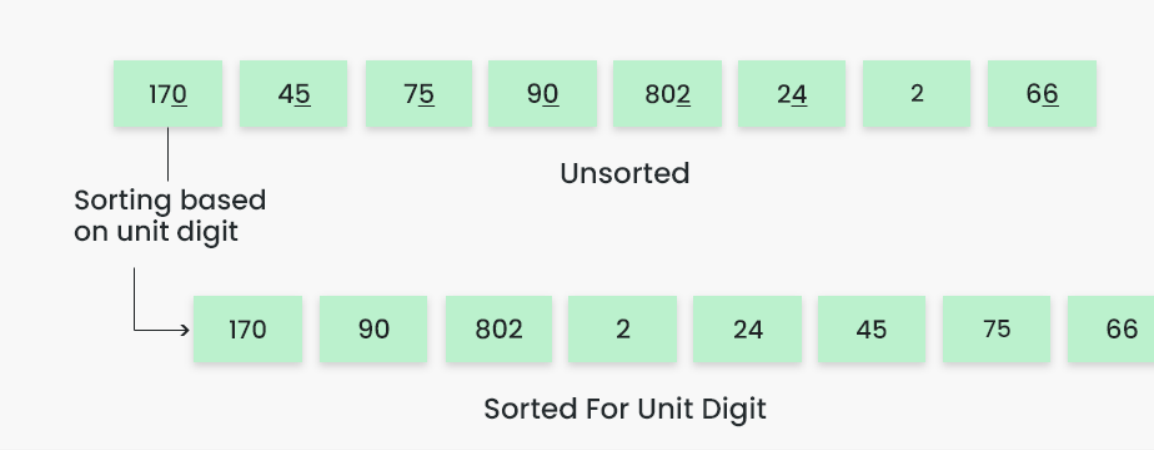
Step 1: *Find the largest element in the array, which is 802. It has three digits, so we will iterate three times, once for each significant place.*

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].*



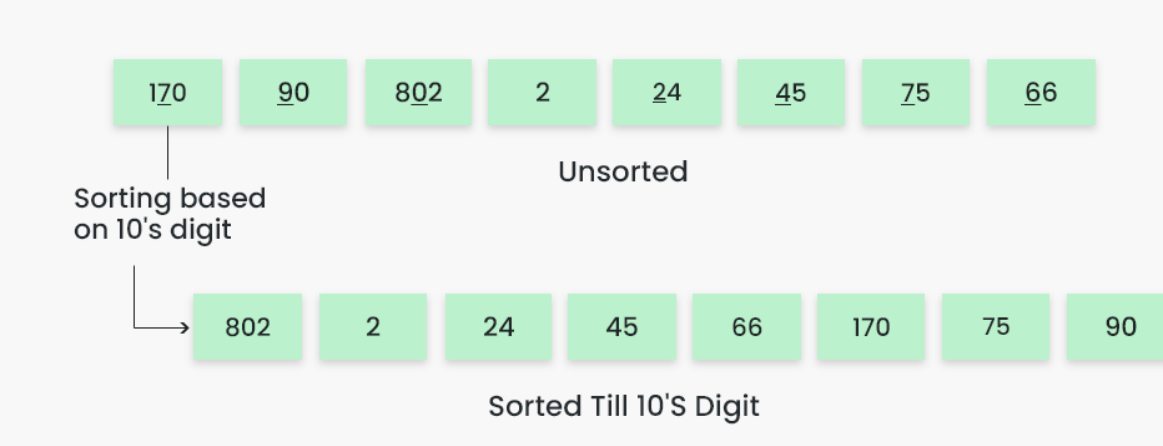
Radix Sort

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].*

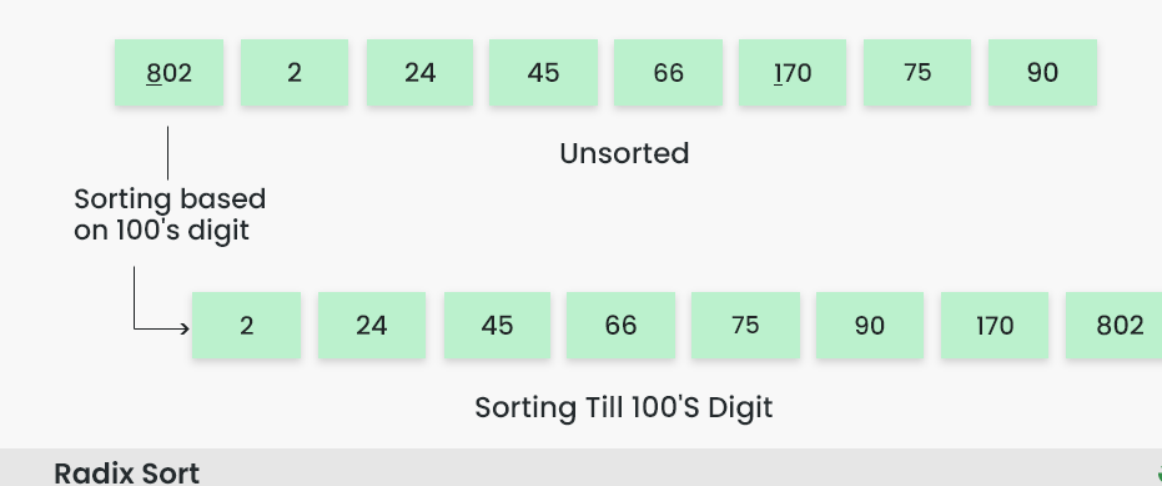


How does Radix Sort Algorithm work | Step 3

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

Sorting based on the hundreds place:

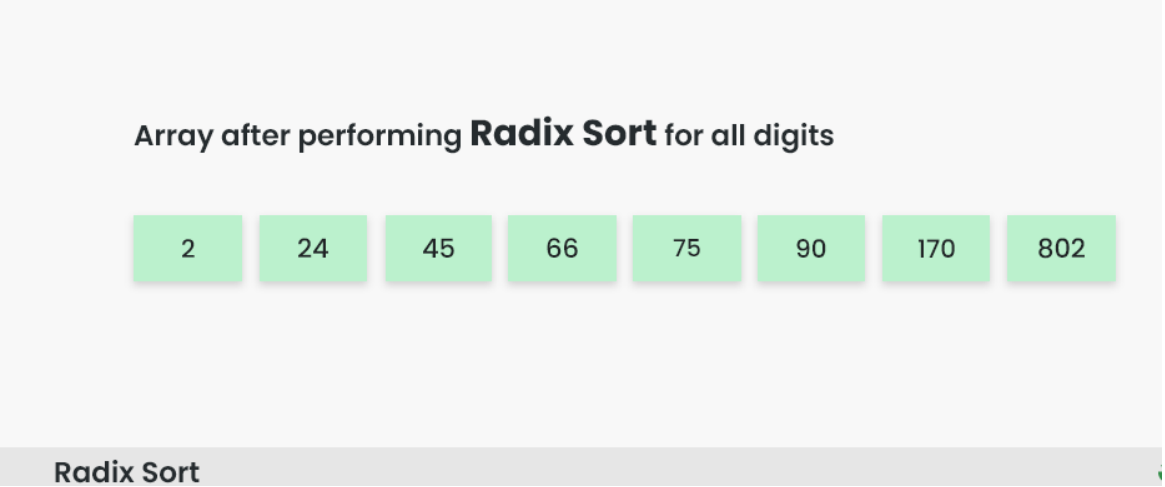
- *Perform counting sort on the array based on the hundreds place digits.*
- *The sorted array based on the hundreds place is [2, 24, 45, 66, 75, 90, 170, 802].*



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].



How does Radix Sort Algorithm work | Step 5

Below is the implementation for the above illustrations:

```
#include <stdio.h>

// A utility function to get the maximum
// value in arr[]
int getMax(int arr[], int n) {
   int mx = arr[0];
   for (int i = 1; i < n; i++)
      if (arr[i] > mx)
         mx = arr[i];
   return mx;
}

// A function to do counting sort of arr[]
// according to the digit represented by exp
void countSort(int arr[], int n, int exp) {
   int output[n]; // Output array
   int count[10] = {0}; // Initialize count array as 0

   // Store count of occurrences in count[]
   for (int i = 0; i < n; i++)
      count[(arr[i] / exp) % 10]++;

   // Change count[i] so that count[i] now
   // contains actual position of this digit
   // in output[]
   for (int i = 1; i < 10; i++)
      count[i] += count[i - 1];
```

```
    // Build the output array
    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }

    // Copy the output array to arr[],
    // so that arr[] now contains sorted
    // numbers according to current digit
    for (int i = 0; i < n; i++)
        arr[i] = output[i];
}

// The main function to sort arr[] of size
// n using Radix Sort
void radixSort(int arr[], int n) {

    // Find the maximum number to know
    // the number of digits
    int m = getMax(arr, n);

    // Do counting sort for every digit
    // exp is 10^i where i is the current
    // digit number
    for (int exp = 1; m / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}

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

// Driver code
int main() {
    int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
    int n = sizeof(arr) / sizeof(arr[0]);

    // Function call
    radixSort(arr, n);
    printArray(arr, n);
    return 0;
```

```
}
```

Output

```
2 24 45 66 75 90 170 802
```

Complexity Analysis of Radix Sort:

Time Complexity:

- Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping the keys by the individual digits which share the same significant position and value. It has a time complexity of **O(d * (n + b))**, where d is the number of digits, n is the number of elements, and b is the base of the number system being used.
- In practical implementations, radix sort is often faster than other comparison-based sorting algorithms, such as quicksort or merge sort, for large datasets, especially when the keys have many digits. However, its time complexity grows linearly with the number of digits, and so it is not as efficient for small datasets.

Auxiliary Space:

- Radix sort also has a space complexity of **O(n + b),** where n is the number of elements and b is the base of the number system. This space complexity comes from the need to create buckets for each digit value and to copy the elements back to the original array after each digit has been sorted.
-

Heap Sort

Heap sort is a comparison-based sorting technique based on Binary Heap Data Structure. It can be seen as an optimization over selection sort where we first find the max (or min) element and swap it with the last (or first). We repeat the same process for the remaining elements. In Heap Sort, we use Binary Heap so that we can quickly find and move the max element in O(Log n) instead of O(n) and hence achieve the O(n Log n) time complexity.

Heap Sort Algorithm

First convert the array into a max heap using **heapify**, Please note that this happens in-place. The array elements are re-arranged to follow heap

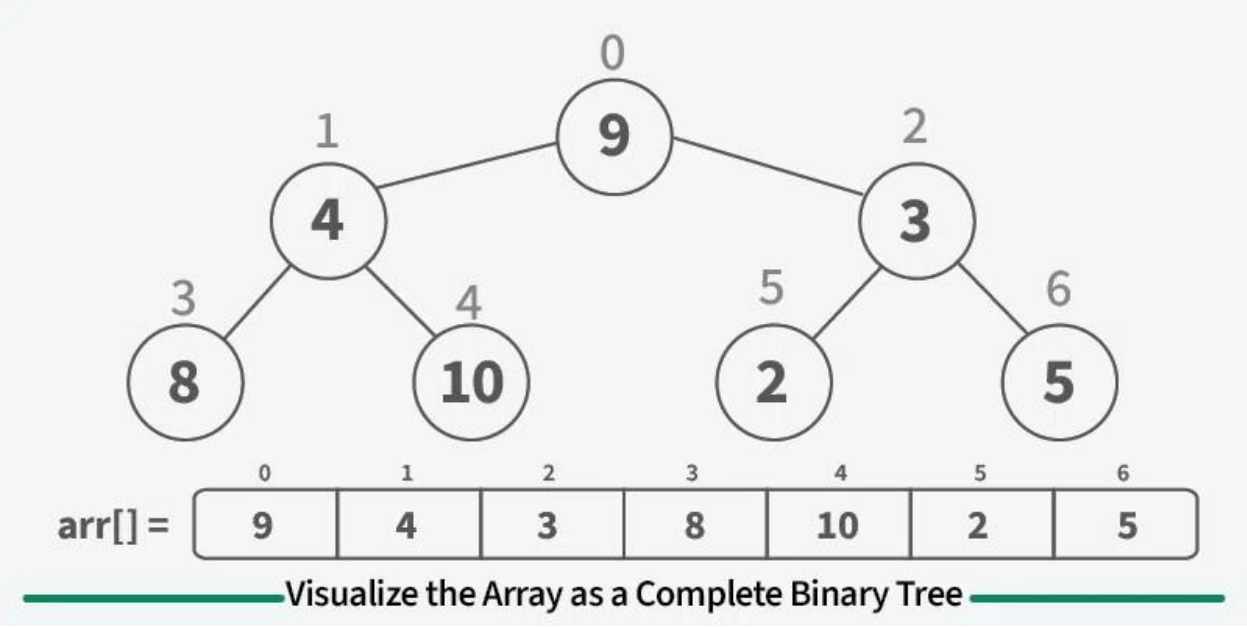
properties. Then one by one delete the root node of the Max-heap and replace it with the last node and **heapify**. Repeat this process while size of heap is greater than 1.

- Rearrange array elements so that they form a Max Heap.
- Repeat the following steps until the heap contains only one element:
    - Swap the root element of the heap (which is the largest element in current heap) with the last element of the heap.
    - Remove the last element of the heap (which is now in the correct position). We mainly reduce heap size and do not remove element from the actual array.
    - Heapify the remaining elements of the heap.
- Finally we get sorted array.

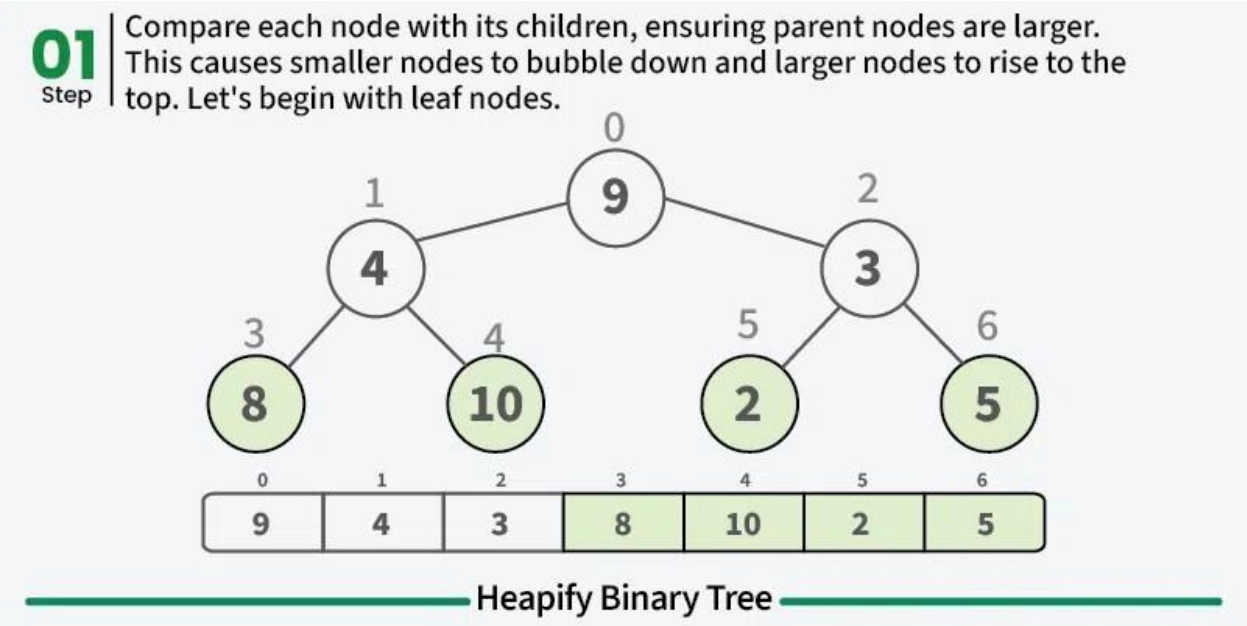
Detailed Working of Heap Sort

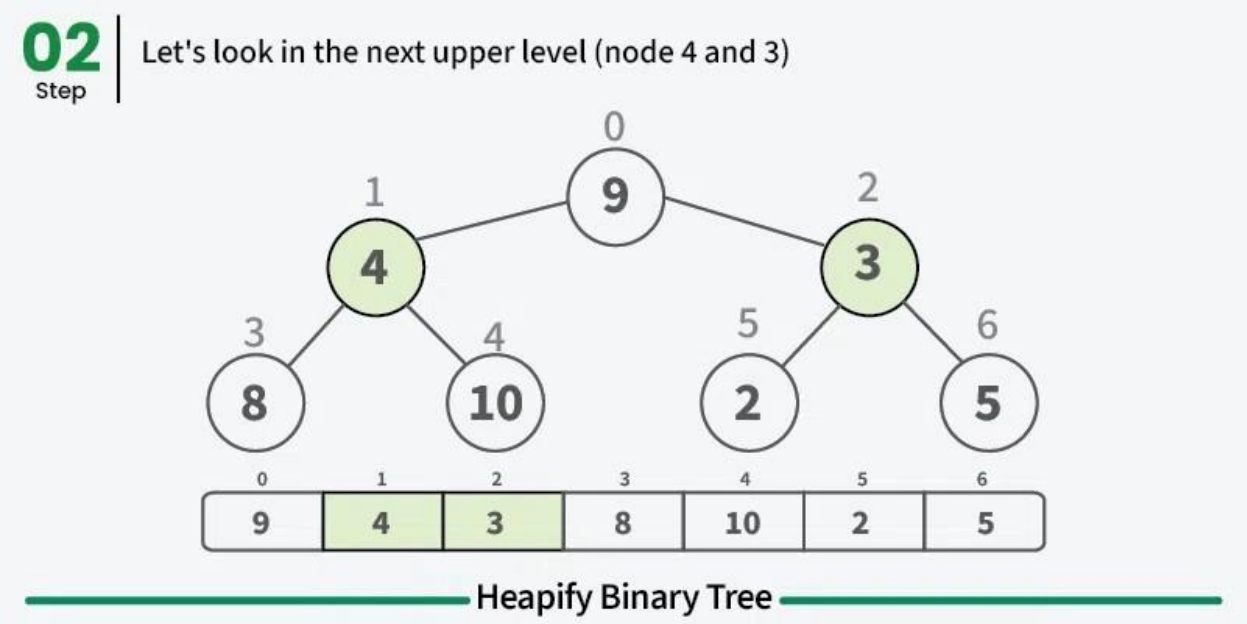
Step 1: Treat the Array as a Complete Binary Tree

We first need to visualize the array as a **complete binary tree**. For an array of size **n**, the root is at index **0**, the left child of an element at index **i** is at **2i + 1**, and the right child is at **2i + 2**.



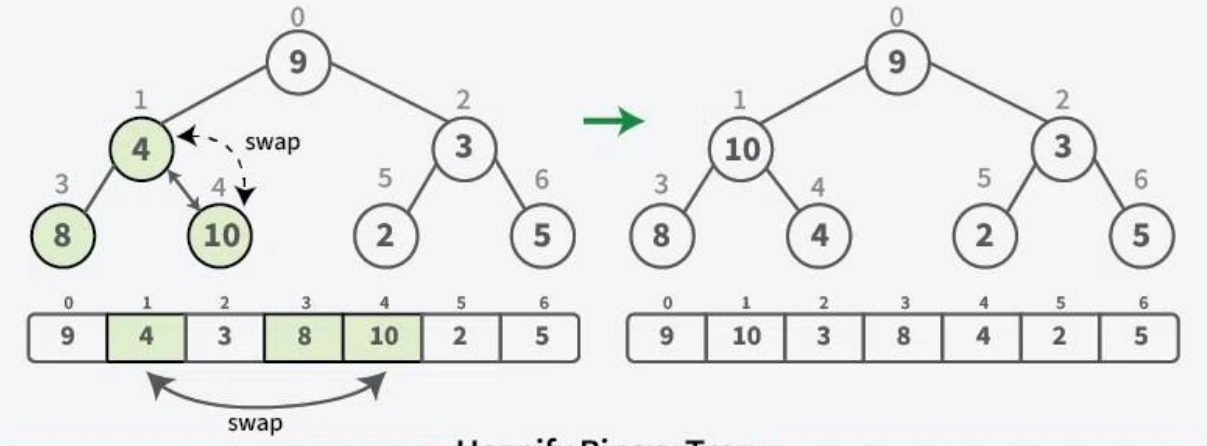
Step 2: Build a Max Heap





03 Step

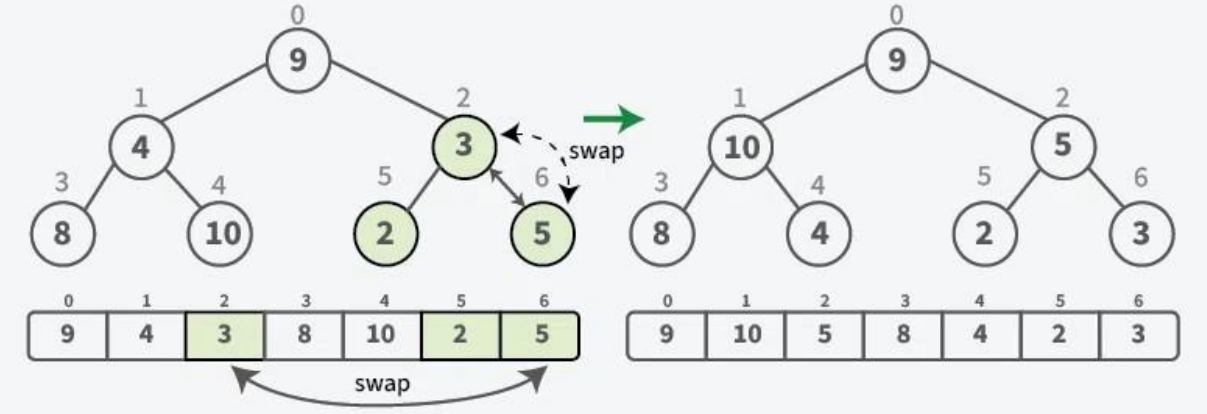
Node 4 is smaller than its child node (10), so swap it with the larger child to maintain the property that the parent should be larger than its children.



Heapify Binary Tree

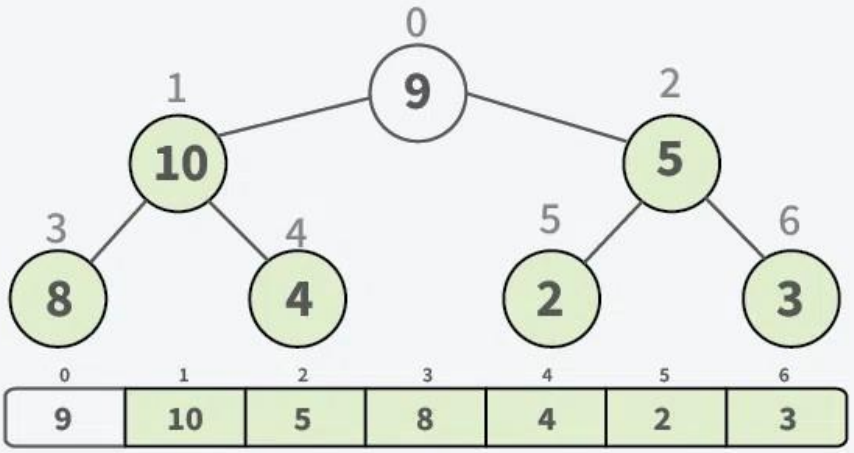
04 Step

Check for the next node (3) in current level. Since, it has a larger child, so swap it to ensure the parent has a larger value.



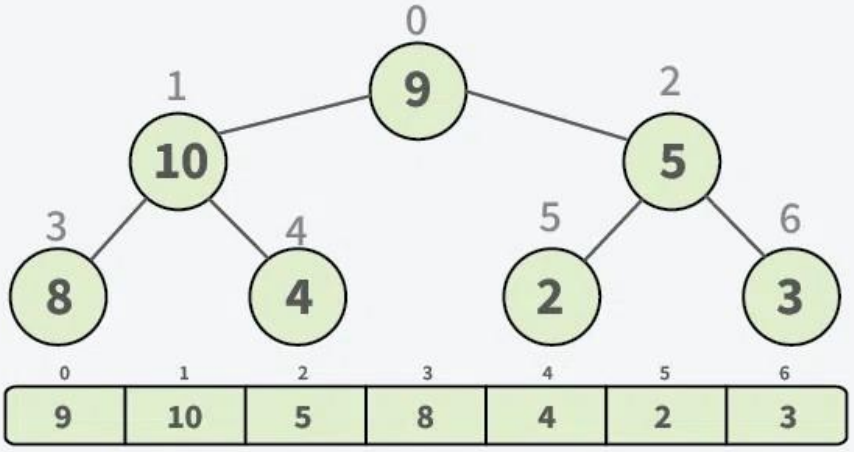
Heapify Binary Tree

05 Step | After the completion of current level, we have now two smaller valid max heap

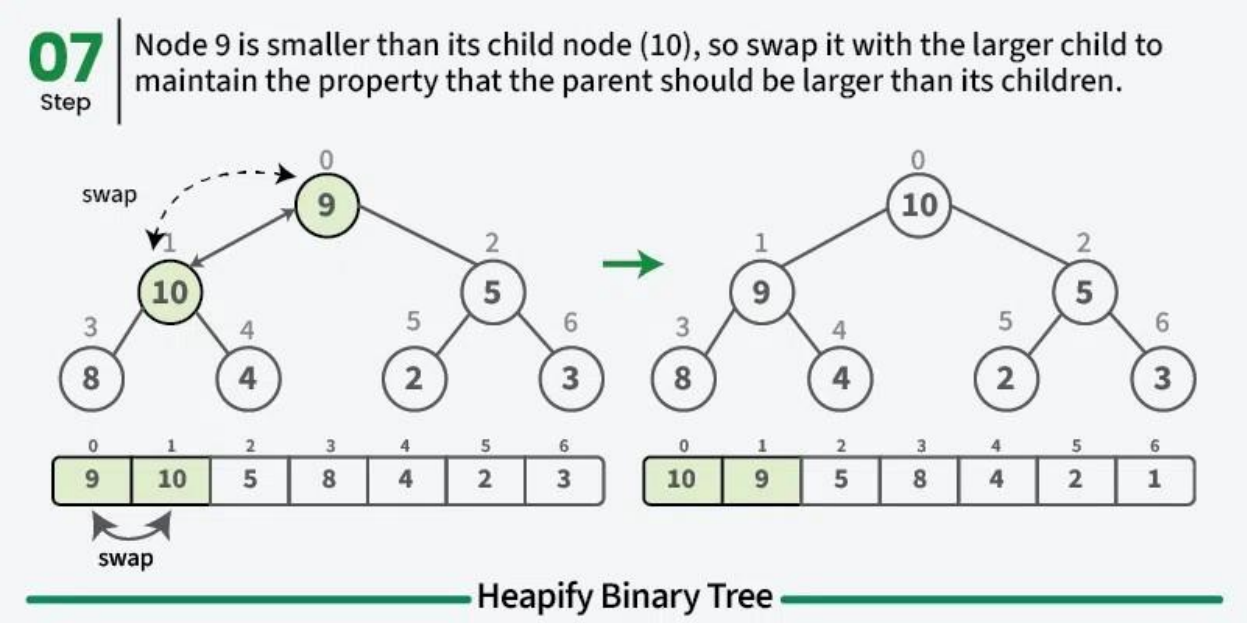


Heapify Binary Tree

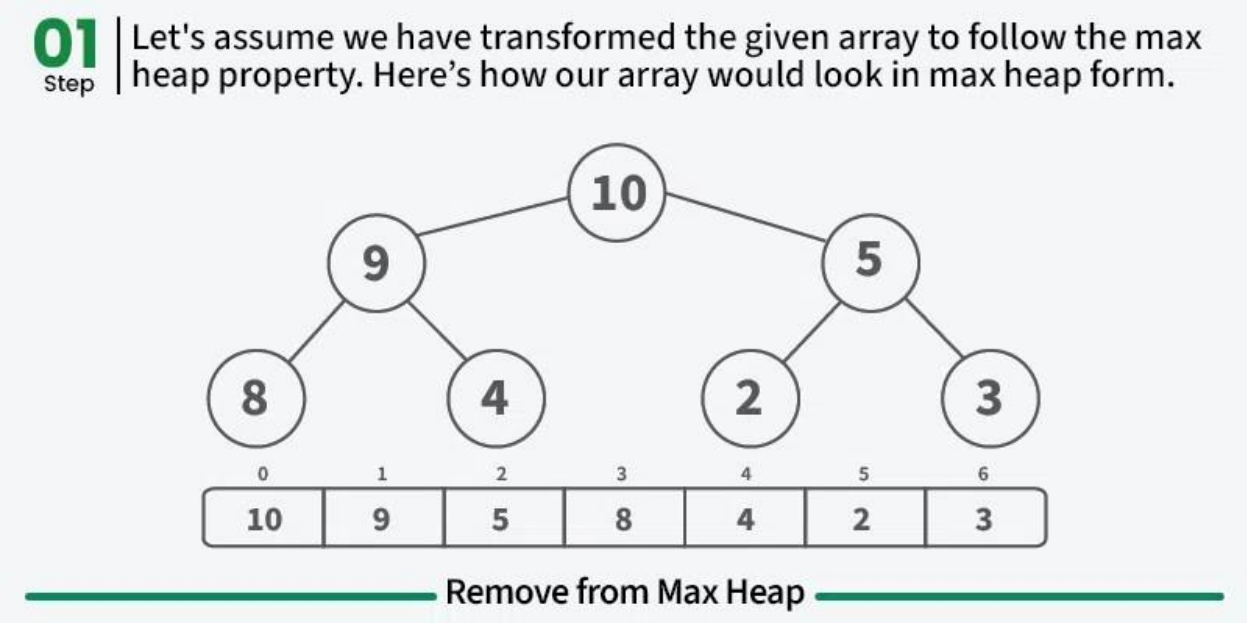
06 Step | Let's move to the next upper level. Here we've node 9 at the root.



Heapify Binary Tree

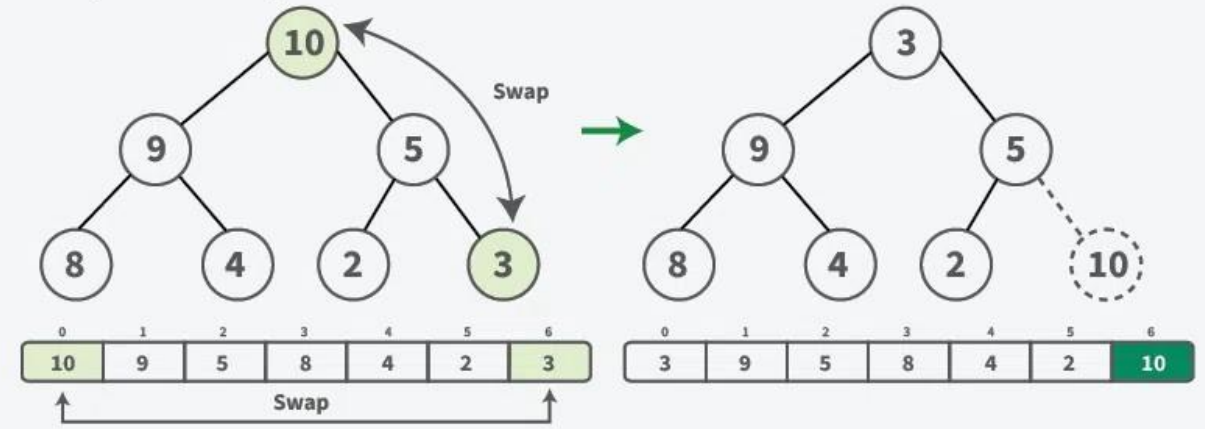


Step 3: Sort the array by placing largest element at end of unsorted array.



02 Step

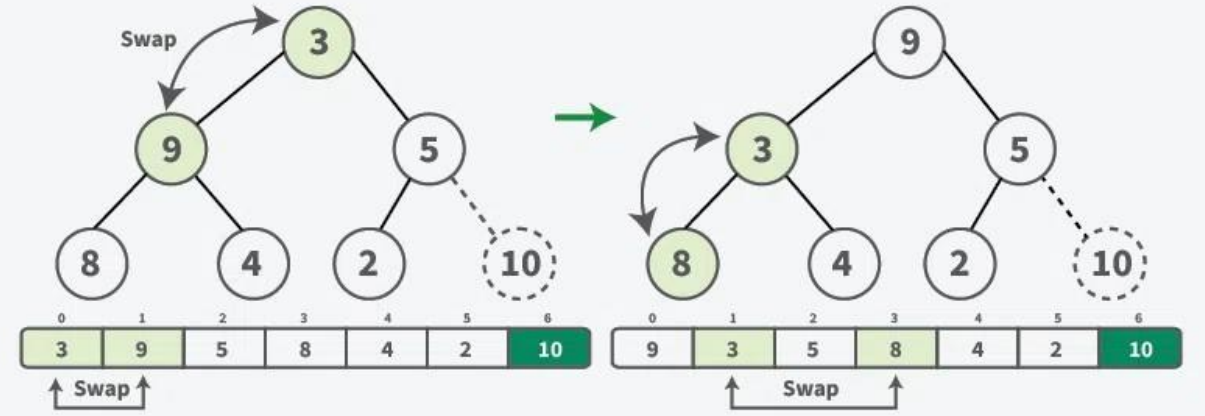
Swap the maximum element (10) with the last element (3) in the unsorted array. Decrease the size of the heap by one (ignore the last element, as it is now sorted).



Remove from Max Heap

03 Step

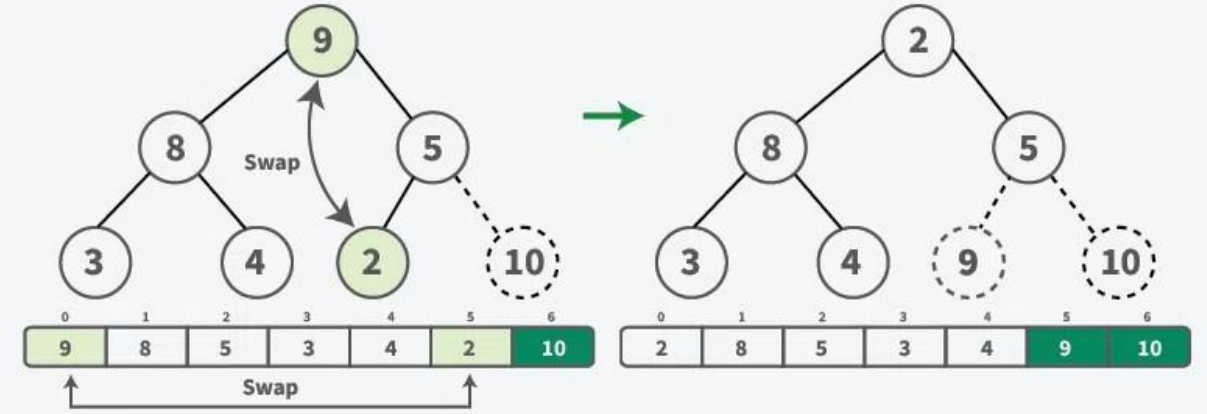
Now, root violate the max-heap property, So, heapify it.
Swap node 3 with its largest child (9).
Still node 3 have larger children, so swap it with largest one (node 8).



Remove from Max Heap

04 Step

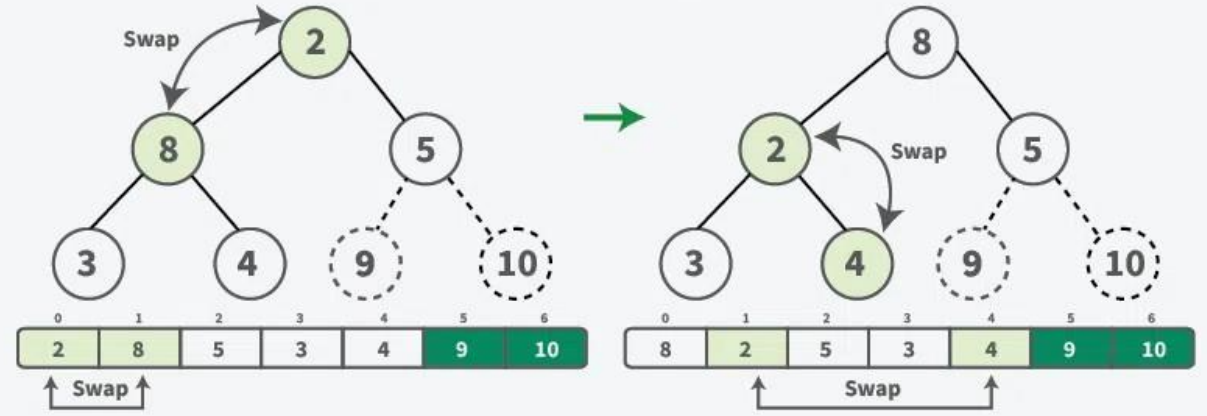
Now, we have a max heap. Swap the maximum element (9) with the last element (2) in the unsorted array, then decrease the heap size by one (ignoring the second last element as it's now sorted).



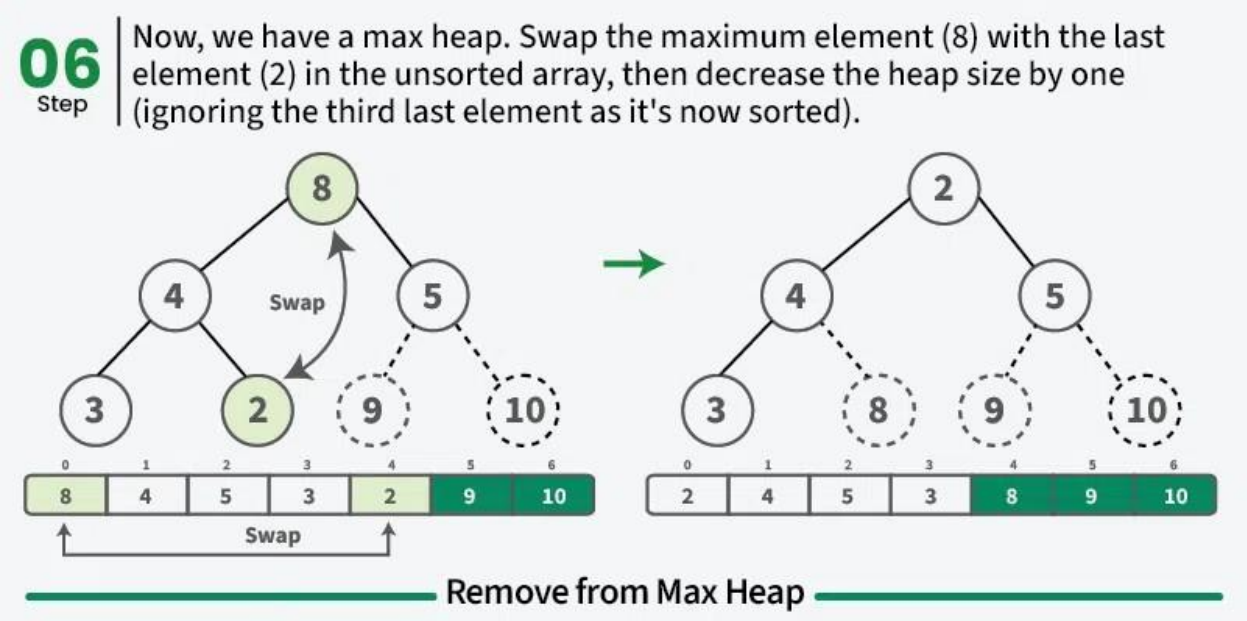
Remove from Max Heap

05 Step

Now, root violate the max-heap property, So, heapify it.
Swap node 2 with its largest child (8).
Still node 2 have larger children, so swap it with largest one (node 4)



Remove from Max Heap



In the illustration above, we have shown some steps to sort the array. We need to keep repeating these steps until there's only one element left in the heap.

```
#include <stdio.h>

// To heapify a subtree rooted with node i
// which is an index in arr[].
void heapify(int arr[], int n, int i) {

  // Initialize largest as root
  int largest = i;

  // left index = 2*i + 1
  int l = 2 * i + 1;

  // right index = 2*i + 2
  int r = 2 * i + 2;

  // If left child is larger than root
  if (l < n && arr[l] > arr[largest]) {
```

```
        largest = l;
    }

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest]) {
        largest = r;
    }

    // If largest is not root
    if (largest != i) {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// Main function to do heap sort
void heapSort(int arr[], int n) {

    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

    // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--) {

        // Move current root to end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // Call max heapify on the reduced heap
        heapify(arr, i, 0);
```

```
    }
}

// A utility function to print array of size n
void printArray(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
}

// Driver's code
int main() {
    int arr[] = {9, 4, 3, 8, 10, 2, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    heapSort(arr, n);

    printf("Sorted array is \n");
    printArray(arr, n);
    return 0;
}
```

Output

```
Sorted array is
2 3 4 5 8 9 10
```

Complexity Analysis of Heap Sort

Time Complexity: O(n log n)

Auxiliary Space: O(log n), due to the recursive call stack. However, auxiliary space can be O(1) for iterative implementation.

Important points about Heap Sort

- An in-place algorithm.
- Its typical implementation is not stable but can be made stable Typically 2-3 times slower than well-implemented QuickSort. The reason for slowness is a lack of locality of reference.

Advantages of Heap Sort

- **Efficient Time Complexity:** Heap Sort has a time complexity of O(n log n) in all cases. This makes it efficient for sorting large datasets. The **log n** factor comes from the height of the binary heap, and it ensures that the algorithm maintains good performance even with a large number of elements.
- **Memory Usage:** Memory usage can be minimal (by writing an iterative heapify() instead of a recursive one). So apart from what is necessary to hold the initial list of items to be sorted, it needs no additional memory space to work
- **Simplicity:** It is simpler to understand than other equally efficient sorting algorithms because it does not use advanced computer science concepts such as recursion.

Disadvantages of Heap Sort

- **Costly**: Heap sort is costly as the constants are higher compared to merge sort even if the time complexity is O(n Log n) for both.
- **Unstable**: Heap sort is unstable. It might rearrange the relative order.
- **Inefficient:** Heap Sort is not very efficient because of the high constants in the time complexity.