

# Sorting Algorithms

- By Gopal Landa on 6th Jan 2026

Bubble sort

```
class Base:  
    def BubbleSort(self, arr):  
        n=len(arr)  
        for i in range(n):  
            for j in range(n-i-1):  
  
                if arr[j]>arr[j+1]:  
                    arr[j],arr[j+1]=arr[j+1],arr[j]  
        return arr  
  
obj=Base()  
arr=[9,4,7,1,5,7,4]  
print(obj.BubbleSort(arr))
```

Selection sort

```
class Base:  
    def selectionSort(self, arr):  
        n=len(arr)  
        for i in range(n):  
            min_idx=i  
            for j in range(i+1,n):  
                if arr[j]<arr[min_idx]:  
                    min_idx=j  
  
            arr[i],arr[min_idx]=arr[min_idx],arr[i]  
        return arr  
  
arr=[7,1,5,9,3,8]  
obj=Base()  
print(obj.selectionSort(arr))
```

Insertion Sort

```
def insertionSort(arr):  
    n=len(arr)  
    for i in range(1,n):  
        j=i-1  
        key_element=arr[i]  
        while j>=0 and arr[j]>key_element:  
            arr[j+1]=arr[j]  
            j-=1  
        arr[j+1]=key_element  
if __name__=="__main__":  
    n=int(input())  
    arr=list(map(int,input().split()))  
    insertionSort(arr)  
    print("[ "+' '.join(map(str,arr))+" ] ")
```

```

Quick Sort

def quickSort_naive(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[0]
    left = [x for x in arr[1:] if x <= pivot]
    right = [x for x in arr[1:] if x > pivot]
    return quickSort_naive(left) + [pivot] + quickSort_naive(right)

def quickSort_lomuto(arr, low, high):
    if low < high:
        pivot_index = lomuto_partition(arr, low, high)
        quickSort_lomuto(arr, low, pivot_index - 1)
        quickSort_lomuto(arr, pivot_index + 1, high)

def lomuto_partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j in range(low, high):
        if arr[j] <= pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[high] = arr[high], arr[i + 1]
    return i + 1

def quickSort_hoare(arr, low, high):
    if low < high:
        pivot_index = hoare_partition(arr, low, high)
        quickSort_hoare(arr, low, pivot_index)
        quickSort_hoare(arr, pivot_index + 1, high)

```

```
def hoare_partition(arr, low, high):
    pivot = arr[low]
    i = low - 1
    j = high + 1
    while True:
        i += 1
        while arr[i] < pivot:
            i += 1
        j -= 1
        while arr[j] > pivot:
            j -= 1
        if i >= j:
            return j
        arr[i], arr[j] = arr[j], arr[i]

if __name__ == "__main__":
    n = int(input("enter the size of thr array:"))
    arr = list(map(int, input("enter the elements :").split()))
    sorted_arr = quickSort_naive(arr)
    print("sorted array_naive:", sorted_arr)
    quickSort_hoare(arr, 0, len(arr) - 1)
    print("sorted array_hoare:", arr)
    quickSort_lomuto(arr, 0, len(arr) - 1)
    print("sorted array_lomuto:", arr)
```

## Merge Sort

```
def mergeSort(arr):  
    n=len(arr)  
    if n>1:  
        mid=n//2  
        left_half=arr[:mid]  
        right_half=arr[mid:]  
        mergeSort(left_half)  
        mergeSort(right_half)  
        merge(arr,left_half,right_half)  
  
def merge(arr,left_half,right_half):  
    i=j=k=0  
    while i<len(left_half) and j< len(right_half):  
        if left_half[i]<right_half[j]:  
            arr[k]=left_half[i]  
            i+=1  
        else:  
            arr[k]=right_half[j]  
            j+=1  
        k+=1  
    while i< len(left_half):  
        arr[k]=left_half[i]  
        i+=1  
        k+=1  
    while j< len(right_half):  
        arr[k]=right_half[j]  
        j+=1  
        k+=1  
  
if __name__=="__main__":  
    n=int(input())  
    arr=list(map(int,input().split()))  
    mergeSort(arr)  
    print("[" + ', '.join(map(str,arr)) + "]")
```

## Counting Sort

```
def countingSort(arr):  
    if not arr:  
        return arr  
    max_val=max(arr)  
    count=[0]*(max_val+1)  
    for num in arr:  
        count[num]+=1  
    for i in range(1, len(count)):  
        count[i]+=count[i-1]  
    res=[0]*len(arr)  
    for i in range(len(arr)-1, -1, -1):  
        num=arr[i]  
        res[count[num]-1]=num  
        count[num]-=1  
    return res  
if __name__=="__main__":  
    n=int(input("enter size of array:"))  
    arr=list(map(int,input().split()))  
    sorted_arr=countingSort(arr)  
    print("sorted array:",sorted_arr)
```

## Radix Sort

```
def digitSort(arr, exp):  
    n=len(arr)  
    output=[0]*n  
    count=[0]*10  
    for num in arr:  
        digit=(num//exp)%10  
        count[digit]+=1  
    for i in range(1,10):  
        count[i]+=count[i-1]  
    for i in range(n-1,-1,-1):  
        num=arr[i]  
        digit=(num//exp)%10  
        output[count[digit]-1]=num  
        count[digit]-=1  
    for i in range(n):  
        arr[i]=output[i]  
  
def radixSort(arr):  
    if not arr:  
        return arr  
    max_val=max(arr)  
    exp=1  
    while max_val//exp>0:  
        digitSort(arr,exp)  
        exp*=10  
    return arr  
  
if __name__=="__main__":  
    n=int(input("enter size of array:"))  
    arr=list(map(int,input().split()))  
    sorted_arr=radixSort(arr)  
    print("sorted array:",sorted_arr)
```

```

Bucket Sort

def insertion_sort(bucket):
    for i in range(1, len(bucket)):
        key = bucket[i]
        j = i - 1
        while j >= 0 and bucket[j] > key:
            bucket[j + 1] = bucket[j]
            j -= 1
        bucket[j + 1] = key

def bucket_sort(arr):
    n = len(arr)
    buckets = [[] for _ in range(n)]

    # Put array elements in different buckets
    for num in arr:
        bi = int(n * num)
        buckets[bi].append(num)

    # Sort individual buckets using insertion sort
    for bucket in buckets:
        insertion_sort(bucket)

    # Concatenate all buckets into arr[]
    index = 0
    for bucket in buckets:
        for num in bucket:
            arr[index] = num
            index += 1

arr = [0.897, 0.565, 0.656, 0.1234, 0.665, 0.3434]
bucket_sort(arr)
print("Sorted array is:")
print(" ".join(map(str, arr)))

```

```
Pan Cake Sort

def pancake_sort(arr):
    n = len(arr)

    # Work from end of array
    for curr_size in range(n, 1, -1):

        # Find index of maximum element
        max_index = arr.index(max(arr[:curr_size]))

        # If max element is not already at its place
        if max_index != curr_size - 1:

            # Step 1: Bring max element to front
            arr[:max_index + 1] =
reversed(arr[:max_index + 1])

        # Step 2: Move max element to its correct position
        arr[:curr_size] = reversed(arr[:curr_size])

    return arr
```

## Heap Sort

```
def heapify(arr, n, i):
    largest = i                  # Initialize largest as root
    left = 2 * i + 1              # Left child
    right = 2 * i + 2             # Right child

    # If left child exists and is greater than root
    if left < n and arr[left] > arr[largest]:
        largest = left

    # If right child exists and is greater than largest
    so far
    if right < n and arr[right] > arr[largest]:
        largest = right

    # If largest is not root
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)      # Recursively
heapify
def heap_sort(arr):
    n = len(arr)

    # Step 1: Build a max heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # Step 2: Extract elements one by one
    for i in range(n - 1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]      # Move max to
end
        heapify(arr, i, 0)                  # Heapify reduced heap
```

## Time & Space Complexity Comparison:

Algorithm	Best	Average	Worst	Space	Stable	In-Place
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	✓	✓
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	✗	✓
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	✓	✓
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	✓	✗
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	✗	✓
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	✗	✓
Counting Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	✓	✗
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n + k)$	✓	✗
Bucket Sort	$O(n)$	$O(n)$	$O(n^2)$	$O(n)$	Depends	✗

Where k is the range of elements

$$k = (\max_{ele} - \min_{ele} + 1)$$

## Real-World Use Cases & Significance

### Bubble Sort

#### Used where?

- Almost **never** in production

- Used in **teaching & debugging**

## Significance

- Simplest sorting logic

## Pros

- Easy to understand
- Detects already sorted arrays

## Cons

- Extremely slow
- 

# 🎯 Selection Sort

## Used where?

- When **memory writes are costly**
- Embedded systems (very limited cases)

## Significance

- Minimum number of swaps

## Pros

- In-place
- Simple

## Cons

- Always  $O(n^2)$

- Not stable
- 

## Insertion Sort

### Used where?

- Small datasets
- Nearly sorted data
- Used inside **TimSort** (Python)

### Significance

- Adaptive algorithm

### Pros

- Fast for small inputs
- Stable
- In-place

### Cons

- Slow for large data
- 

## Merge Sort

### Used where?

- External sorting (large files)
- Databases

- Linked lists

## Significance

- Guaranteed  $O(n \log n)$

## Pros

- Stable
- Predictable performance

## Cons

- Extra memory required
- 

# ⚡ Quick Sort

## Used where?

- **Most real-world systems**
- C/C++ standard libraries

## Significance

- Fastest in practice (cache-friendly)

## Pros

- In-place
- Very fast average case

## Cons

- Worst case  $O(n^2)$

- Not stable
- 

## Heap Sort

### Used where?

- Priority queues
- Systems needing **guaranteed  $O(n \log n)$**

### Significance

- No worst-case degradation

### Pros

- In-place
- Consistent performance

### Cons

- Slower than quick sort in practice
  - Not stable
- 

## Counting Sort

### Used where?

- Sorting marks, ages, IDs
- When range is small

### Significance

- Linear time sorting

### Pros

- Very fast
- Stable

### Cons

- Works only for integers
  - High memory for large ranges
- 

## Radix Sort

### Used where?

- Sorting large integers
- Phone numbers, ZIP codes

### Significance

- Non-comparison sorting

### Pros

- Linear time
- Stable

### Cons

- Complex
- Needs stable sub-sort

# Bucket Sort

## Used where?

- Floating-point numbers
- Uniformly distributed data

## Significance

- $O(n)$  average time

## Pros

- Very fast for uniform data

## Cons

- Poor for skewed data
- Range dependent

Scenario	Best Choice
Nearly sorted data	Insertion Sort
Large data, stable	Merge Sort
Fast general purpose	Quick Sort
Guaranteed worst case	Heap Sort
Small integer range	Counting Sort
Floating numbers	Bucket Sort
Large integers	Radix Sort