**1. Counting Sort**

**History: Developed by Harold H. Seward in the early 1950s (often cited around 1954).Recognized for its ability to sort integers in linear time under the right conditions.**

**Definition: Counting Sort is a non-comparison-based sorting algorithm that sorts elements by counting the number of occurrences of each unique value in the input array. It then uses these counts to determine the positions of each element in the final, sorted array.**

* **The algorithm operates in linear time,**
* **Time complexity: O(n + k), where n is the number of elements and k is the range of the input values**

**Algorithm:**

1. **Determine the Range:Find the minimum and maximum values in the input array to determine the range of the data. (Sometimes min is assumed to be 0 to simplify the process.)**
2. **Initialize the Count Array:Create a count array of size (max - min + 1) and initialize all elements to 0.**
3. **Count the Occurrences:Iterate over the input array and for each element, increment the corresponding index in the count array. For an element x, increment count[x-min].**
4. **Transform the Count Array into a Cumulative Count Array:Modify the count array such that each element at index i contains the sum of previous counts. This cumulative count tells you the final position of each element in the output array.**
5. **Build the Output Array:**

* **Traverse the input array once more, usually in reverse order to ensure stability (maintaining the original order of equal elements).**
* **Place each element x into its correct position in the output array by using the cumulative count, then decrement the count value for x.**

**Stability**

* **Stable: Maintains the relative order of equal elements if implemented correctly.**

**Complexity**

* **Time Complexity: O(n + k), where nn is the number of elements and kk is the range of values.**
* **Space Complexity: O(n + k) due to the need for count and output arrays.**

**Pros**

* **Linear Time: Highly efficient when the value range is not significantly larger than the number of elements.**
* **Simple and Easy to Implement.**
* **Stable.**

**Cons**

* **Memory Intensive: Requires significant extra memory when the value range kk is large.**
* **Not Flexible: Primarily suitable for integers or discrete values.**
* **Inefficient for Wide Value Range: Less effective for data with a wide range of values.**

**2. Pigenohole sort**

**Peter Gustav Lejeune Dirichlet, who formally stated the idea in 1834.**

**Definition:** [**Pigeonhole sorting**](https://en.wikipedia.org/wiki/Pigeonhole_sort) **is a sorting algorithm that is suitable for sorting lists of elements where the number of elements and the number of possible key values are approximately the same.**

**It requires O(*n* + *Range*) time where n is the number of elements in the input array and ‘Range’ is the number of possible values in the array.**

**Algorithm:**

**-It is similar to** [**counting sort**](https://www.geeksforgeeks.org/counting-sort/)**, but differs in that it “moves items twice: once to the bucket array and again to the final destination.**

1. **Determine the range: Find minimum and maximum values in array. Let the minimum and maximum values be ‘min’ and ‘max’ respectively. Also find range as ‘max-min+1’.**
2. **Initialize an array:  Set up an array of initially empty “pigeonholes” the same size as of the range.**
3. **Distribution step: Visit each element of the array and then put each element in its pigeonhole. An element arr[i] is put in hole at index arr[i] – min.**
4. **Output reconstruction phase: Start the loop all over the pigeonhole array in order and put the elements from non- empty holes back into the original array.**

**Stability**

* **Potentially Stable: Can maintain the relative order of equal elements if implemented properly.**

**Complexity**

* **Time Complexity: O(n + r), where nn is the number of elements and rr is the range of values.**
* **Space Complexity: O(n + r) due to the need for pigeonholes for each possible value.**

**Pros**

* **Efficient: When the data values are within a small and almost continuous range.**
* **Simple Implementation: Easy to implement for small value ranges.**
* **Fast: When the range rr is not much larger than the number of elements nn.**

**Cons**

* **Memory Intensive: Requires significant extra memory when the range rr is large.**
* **Reduced Efficiency: Not effective for data with a wide value range.**
* **Less Common: Less frequently used compared to Counting Sort due to strict data constraints.**

**3.Heap sort**

**History:Heap Sort was first introduced by J. W. J. Williams in 1964. This innovation aimed to exploit the heap data structure to improve sorting performance.**

**Definition:Heap Sort is a comparison-based, in-place sorting algorithm that leverages the properties of a heap data structure—specifically, a max heap when sorting in ascending order. Its basic premise is:**

1. **Max Heap Property: In a max heap, every parent node is greater than or equal to its children. This guarantees that the largest element is always at the root.**
2. **In-Place Sorting: Heap Sort rearranges the data within the input array, meaning it does not require significant additional memory.**
3. **Time Complexity: It consistently operates in O(n log n) time in the worst case, making it reliable and predictable.**

**Algorithm:**

**Step 1: Build the Max Heap**

1. **Determine the Starting Point:**
   * **The array is treated as a complete binary tree.**
   * **The last non-leaf node resides at index ⌊n/2⌋ - 1, where n is the total number of elements.**
2. **Call heapify()  on Each Non-Leaf Node:**
   * **Starting from the last non-leaf node and moving upward to the root, apply the heapify()  procedure to enforce the max heap property.**
   * **The heapify() procedure ensures that for a node at index i, the value at i is greater than or equal to its children.**

**Step 2: Sort the Array (Extract Elements from Heap)**

1. **Swap the Root with the Last Element:**
   * **The root of the max heap (index 0) holds the maximum value. Swap it with the last element in the heap (at index n-1).**
2. **Reduce Heap Size:**
   * **After the swap, consider the last element as sorted. Reduce the effective heap size by one so that it is excluded from further heap operations.**
3. **Re-heapify the Root:**
   * **Call the heapify() procedure on the root (index 0) to restore the max heap property over the reduced heap.**
4. **Repeat Extraction:**
   * **Continue this process—swapping the root with the last unsorted element and re-heapifying—until the entire array is sorted.**

**4.Odd-Even Sort**

**Stability**

**Potentially Stable: Can maintain the relative order of equal elements if implemented carefully.**

**Complexity**

**Time Complexity: O(n²), where n is the number of elements.**

**Space Complexity: O(1), as it sorts in-place with minimal extra memory.**

**Pros**

**Simple Implementation: Straightforward and easy to understand, especially for parallel processing.**

**Parallelizable: Well-suited for parallel architectures due to independent odd-even phase comparisons.**

**In-Place: Requires no additional data structures, making it memory-efficient.**

**Cons**

**Inefficient for Large Datasets: Quadratic time complexity makes it slow for large n.**

**Not Adaptive: Does not take advantage of partially sorted data.**

**Less Practical: Rarely used in practice compared to more efficient algorithms like QuickSort or MergeSort.**

**Quiz**

**1 What is Heapify in Heap Sort?**

**A. Building a complete binary tree**

**B. Ensuring the max heap or min heap property**

**C. Sorting the array by comparing each element**

**D. Creating an AVL tree**

**Answer: B. Ensuring the max heap or min heap property**

**2) Which of the following is not an application of Heap Sort?**

**A. Sorting large datasets**

**B. Implementing priority queues**

**C. Sorting data in external memory**

**D. Sorting data with a small range of values**

**Answer: D. Sorting data with a small range of values**

**3 Which of the following is not an application of Counting Sort?**

**A. Sorting integers within a small range**

**B. Generating histograms from data**

**C. Sorting large datasets with a large range of values**

**D. Sorting characters in a string**

**Answer: C. Sorting large**

**4) When does Pigeonhole Sort work most efficiently?**

**A. When the number of elements (n) is larger than the range of values (k)**

**B. When the number of elements (n) is approximately equal to the range of values (k)**

**C. When the data consists of real numbers**

**D. When the data consists of strings**

**Answer: B. When the number of elements (n) is approximately equal to the range of**

**5) How is Pigeonhole Sort different from Counting Sort?**

**A. Pigeonhole Sort moves elements twice**

**B. Counting Sort moves elements twice**

**C. Pigeonhole Sort does not require a range of values**

**D. Counting Sort does not require a range of values**

**Answer: A. Pigeonhole Sort moves elements twice**

**6 Which algorithm is most suitable for sorting integers within a small range?**

**A. Heap Sort**

**B. Counting Sort**

**C. Pigeonhole Sort**

**D. Both B and C**

**Answer: D. Both B and C**