Introduction to Sorting

Sorting refers to the process of arranging data in a particular order, typically ascending or descending. In the context of elementary sorting, it involves basic sorting techniques that are easy to implement and understand. These sorting algorithms provide foundational understanding of how data organization affects performance and are commonly used in teaching, small datasets, or as building blocks for more advanced sorting techniques.
Sorting is essential for:

- Optimizing search algorithms
- Generating reports
- Preparing data for other operations (e.g., merging, analysis)

Types of Elementary Sorting Algorithms

The common types of elementary sorting algorithms include:

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Shell Sort
5. Counting Sort
6. Radix Sort

1. Bubble Sort

Bubble Sort is one of the simplest elementary sorting algorithms. It works by repeatedly swapping the adjacent elements if they are in the wrong order. This process continues until the entire list is sorted. It is called "bubble" sort because smaller elements gradually "bubble" up to the top (beginning of the array) with each iteration.

Working Principle

- Start from the first element, compare it with the next.
- If the current element is greater than the next, swap them.
- Move to the next element and repeat the comparison until the end of the array.
- After the first pass, the largest element is at the end.

- Repeat the process for the remaining elements, excluding the last sorted ones.
- Continue until no more swaps are needed.

Algorithm

```
for i = 0 to n - 1
    for j = 0 to n - i - 2
        if A[j] > A[j + 1]
            swap A[j] with A[j + 1]
```

Example

Sort the array [5, 3, 8, 4, 2] using Bubble Sort:

- Pass 1: [5, 3, 8, 4, 2] → [3, 5, 8, 4, 2] → [3, 5, 8, 4, 2] → [3, 5, 4, 8, 2] → [3, 5, 4, 2, 8]
- Pass 2: [3, 5, 4, 2, 8] → [3, 5, 4, 2, 8] → [3, 4, 5, 2, 8] → [3, 4, 2, 5, 8]
- Pass 3: [3, 4, 2, 5, 8] → [3, 4, 2, 5, 8] → [3, 2, 4, 5, 8]
- Pass 4: [3, 2, 4, 5, 8] → [2, 3, 4, 5, 8]

Sorted array: [2, 3, 4, 5, 8]

Applications

- Useful for teaching purposes to explain sorting logic.
- Suitable for small datasets.
- Can be used where memory space is limited, as it is in-place.
- Not preferred for large datasets due to inefficiency.

Complexity

Case	Complexity
Best Case	$O(n)$ (when the array is already sorted)
Average Case	$O(n^2)$
Worst Case	$O(n^2)$
Space Complexity	$O(1)$

2. Selection Sort

Selection Sort repeatedly selects the smallest (or largest) element from the unsorted part and moves it to the sorted part.

Algorithm

```
for i = 0 to n - 2
    minIndex = i
    for j = i + 1 to n - 1
        if A[j] < A[minIndex]
            minIndex = j
    swap A[i] with A[minIndex]
```

Applications

- Memory write is costly
- Simple embedded systems
- Unstable sorting where minimal swaps are preferred

Complexity

Case	Complexity
Best Case	$O(n^2)$
Average Case	$O(n^2)$
Worst Case	$O(n^2)$
Space Complexity	$O(1)$

3. Insertion Sort

Insertion Sort builds the final sorted array one element at a time by comparing each new element with the already sorted elements and inserting it into its correct position.

Algorithm

```
for i = 1 to n - 1
    key = A[i]
    j = i - 1
    while j >= 0 and A[j] > key
        A[j + 1] = A[j]
```

```
        j = j - 1
    A[j + 1] = key
```

Applications

- Small datasets
- Online sorting (real-time input)
- Nearly sorted arrays

Complexity

Case	Complexity
Best Case (Sorted Input)	$O(n)$
Average Case	$O(n^2)$
Worst Case (Reversed Input)	$O(n^2)$
Space Complexity	$O(1)$

4. Shell Sort

Shell Sort improves Insertion Sort by allowing comparison and exchange of elements far apart. It uses a gap sequence to perform comparisons, which is reduced until a final insertion sort pass.

Algorithm

```
gap = n / 2
while gap > 0
    for i = gap to n - 1
        temp = A[i]
        j = i
        while j >= gap and A[j - gap] > temp
            A[j] = A[j - gap]
            j -= gap
        A[j] = temp
    gap = gap / 2
```

Applications

- Medium-sized datasets
- Non-recursive environments
- Performance-sensitive embedded systems

Complexity (depends on gap sequence)

Case	Complexity
Best Case	$O(n \log n)$ (depends on gap sequence)
Average Case	$O(n^{3/2})$ (or better with optimal gaps)
Worst Case	$O(n^2)$
Space Complexity	$O(1)$

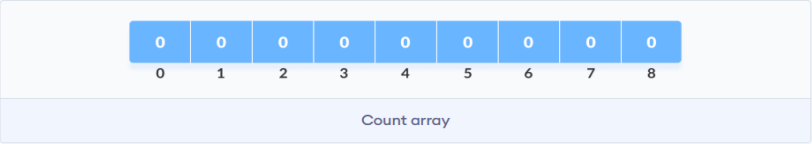
5. Counting Sort

Counting Sort is an integer sorting algorithm that works by counting the occurrences of each unique element in the input array and storing these counts in an auxiliary array. It uses the counts to determine the position of each element in the sorted output array.

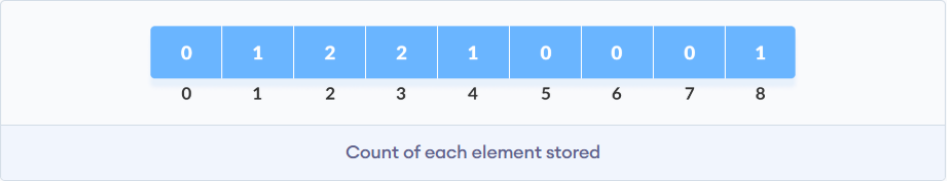
Working of Counting Sort

1. Find out the maximum element (let it be `max`) from the given array.

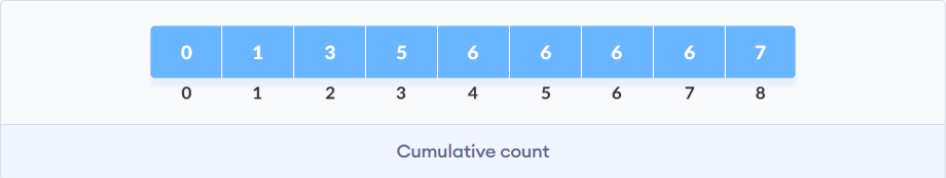
2. Initialize an array of length `max+1` with all elements 0. This array is used for storing the count of the elements in the array.



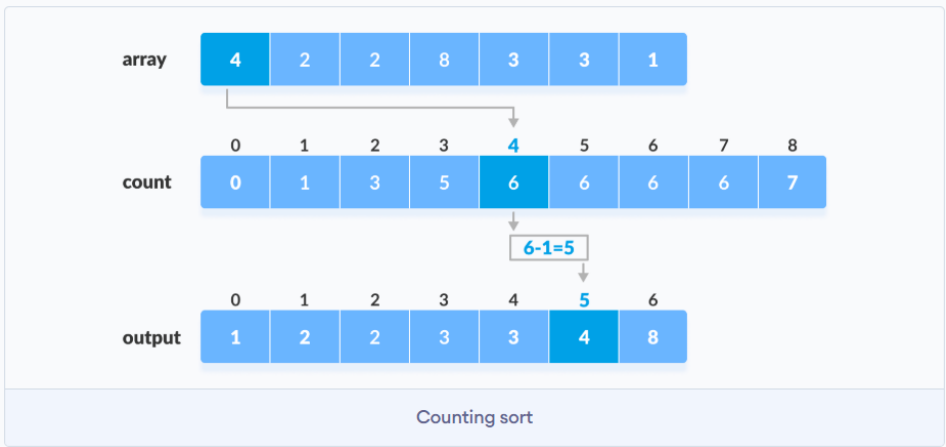
3. Store the count of each element at their respective index in count array. For example: if the count of element 3 is 2 then, 2 is stored in the 3rd position of count array. If element "5" is not present in the array, then 0 is stored in 5th position.



4. Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array.



5. Find the index of each element of the original array in the count array. This gives the cumulative count. Place the element at the index calculated as shown in figure below.



6. After placing each element at its correct position, decrease its count by one.

Algorithm

```
countingSort(array, size)
  max <- find largest element in array
  initialize count array with all zeros
  for j <- 0 to size
    find the total count of each unique element and
    store the count at jth index in count array
  for i <- 1 to max
    find the cumulative sum and store it in count array itself
  for j <- size down to 1
    restore the elements to array
    decrease count of each element restored by 1
```

Time Complexity

- **Best, Average, and Worst Case Time Complexity:** $O(n + k)$, where:
  - n is the number of elements in the array.
  - k is the range of the input (maximum element in the array).
- The time complexity is linear with respect to both the number of elements and the range of the input.

Space Complexity

- **Space Complexity:** $O(n + k)$, where:
  - n is the number of elements in the array.
  - k is the range of the input (maximum element in the array).
- Extra space is required for the `count` array.

Applications

- **Sorting Integer Values:** Counting Sort is particularly efficient when the range of input values is not significantly larger than the number of elements.
- **Non-negative Integers:** Suitable for sorting non-negative integers in a variety of applications.
- **Frequency Distribution:** Used in problems where frequency counts of items are needed, such as counting words in a document or items in a dataset.
- **Radix Sort:** Counting Sort is used as a subroutine to sort individual digits in Radix Sort.
- **Image Processing:** Often used in image processing tasks, such as image histogram equalization.

Counting Sort Complexity

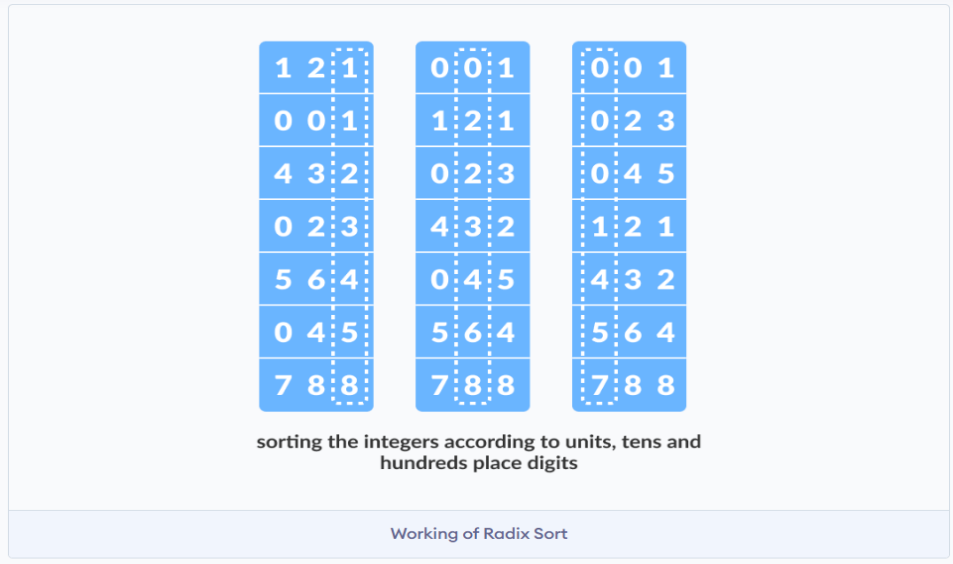
Case	Time Complexity
Best Case	$O(n + k)$
Average Case	$O(n + k)$
Worst Case	$O(n + k)$
Space Complexity	$O(n + k)$

6. Radix Sort

Radix sort is a sorting algorithm that sorts the elements by first grouping the individual digits of the same place value. Then, it sorts the elements according to their increasing or decreasing order.

Suppose, we have an array of 8 elements. First, we will sort the elements based on the value at the unit place. Then, we sort the elements based on the value at the tenth place. This process continues until the most significant place.

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.



Working of Radix Sort

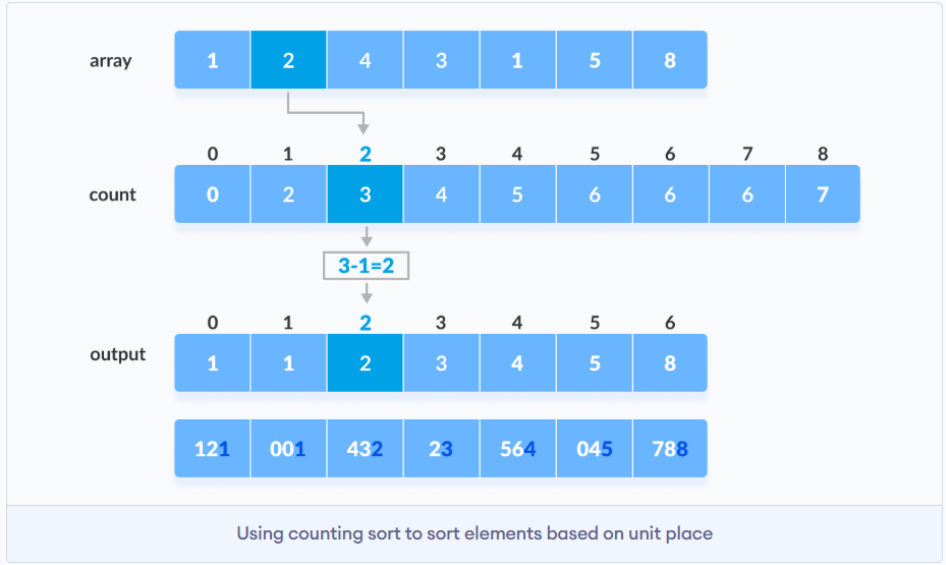
1. Find the largest element in the array, i.e. `max`. Let X be the number of digits in `max`. X is calculated because we have to go through all the significant places of all elements.

   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.

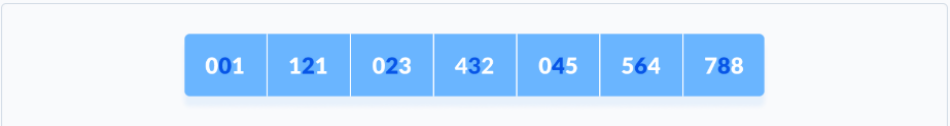
   Use any stable sorting technique to sort the digits at each significant place. We have used **counting sort** for this.

   Sort the elements based on the unit place digits ($X = 0$).



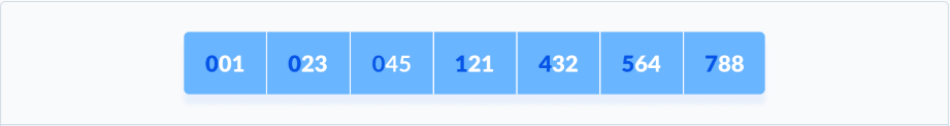
Using counting sort to sort elements based on unit place

3. Now, sort the elements based on digits at tens place.



Sort elements based on tens place

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



Sort elements based on hundreds place

Algorithm

```
radixSort(array)
  d <- maximum number of digits in the largest element
  create d buckets of size 0-9
  for i <- 0 to d
    sort the elements according to i-th place digits using countingSort

countingSort(array, d)
  max <- find largest element among dth place elements
  initialize count array with all zeros
  for j <- 0 to size
    find the total count of each unique digit in dth place of elements and
    store the count at jth index in count array
```

```
for i <- 1 to max
  find the cumulative sum and store it in count array itself
for j <- size down to 1
  restore the elements to array
  decrease count of each element restored by 1
```

Complexity

Radix Sort Time Complexity	
Best Case	$O(n \cdot k)$
Average Case	$O(n \cdot k)$
Worst Case	$O(n \cdot k)$
Space Complexity	$O(n + k)$
Stability	Yes

- n = Number of elements in the input array.
- k = Number of digits in the largest number (or maximum number of characters if sorting strings)

The space complexity of Radix Sort is $O(n + k)$.

- **Input Array (n):** The original array that needs to be sorted.
- **Output Array (n):** An additional array of the same size as the input array is used to store the sorted elements temporarily during each pass of the counting sort.
- **Count Array (k):** An auxiliary array of size 10 (since digits range from 0–9) is used to store the frequency of digits. This size remains constant regardless of the number of elements but is considered in the space complexity calculation.

Applications

Radix sort is implemented in:

- DC3 algorithm (Kärkkäinen-Sanders-Burkhardt) while making a suffix array.
- Places where there are numbers in large ranges.

Lab exercises

Exercise 1 .....

Classroom Seating – Avoid Cheating: You are organizing an exam in a square $N \times N$ classroom. You want to seat students such that:

- **No student is allowed to sit directly adjacent** (up, down, left, or right) to another student.
- You must place **exactly** K students in the classroom.

Your task is to find the **total number of valid ways** to seat the students under this constraint.

Write a **recursive backtracking solution** to count all possible valid arrangements.

Function Signature:

```
int countSeatingArrangements(int N, int K);
```

Example

```
Input: N = 3, K = 2
Output: 16
```

Additional Insight

This problem is inspired by the classic **N-Queens** problem but comes with a different type of constraint — **no two students can be placed adjacent to each other**, meaning they can't sit next to each other **horizontally or vertically** in the grid.

To solve this:

- Use **backtracking** to try placing students one-by-one into valid grid positions.
- At each step, ensure the current position is not adjacent to any already seated student.
- Recurse to the next position, keeping track of the number of students placed.
- Stop when K students have been placed, and count that arrangement as valid.

This problem tests your understanding of **recursive thinking and pruning invalid choices**, which are at the heart of backtracking.

Note

There was no problem on **backtracking** in the previous lab, so this exercise has been added to fill that gap. It serves as a hands-on opportunity to understand how recursive search with constraints works.

Exercise 2

Generic Sort Using Callbacks and Operator Overloading: Implement a generic, templatized sort function in C++ that can sort elements of any data type. The sort function must support custom comparison logic provided in one of the following forms:

- A **function pointer**
- A **functor (function object)**
- A **lambda function**
- Or, if none is provided, it should use the **default overloaded $<$ operator**

You must use `std::function<bool(const T&, const T&)>` as the comparator parameter to support all types of callables.

Why Use `std::function`?

To support function pointers, functors, and lambdas as comparison logic, use:

```
std::function<bool(const T&, const T&)>
```

This is necessary because:

- You cannot assign a functor or lambda to a raw function pointer.
- `std::function` can encapsulate all types of callable entities.
- During template argument deduction, raw function pointers or lambdas are not automatically converted to `std::function`. So you must wrap them explicitly.

Function Signature

```
template<typename T>
void sort(std::vector<T>& arr, std::function<bool(const T&, const T&)> comp = nullptr);
```

Key Requirements

- Implement the **Selection Sort** algorithm inside your `sort()` function.
- Inside the function, check if `comp` is provided:
  - If `comp` is not `nullptr`, use it for comparisons.
  - If `comp` is `nullptr`, fall back to using the default `<` operator.

Example logic inside sort function:

```
if (comp) {
    if (comp(arr[j], arr[min_index])) {
        // use custom comparator
    }
} else {
    if (arr[j] < arr[min_index]) {
        // use default operator<
    }
}
```

Tasks

1. Define a class `Item` with at least two fields, such as:

   ```
   int id; std::string name;
   ```

2. Overload the `<` operator in the `Item` class to compare by `id`.
3. Implement the `sort()` function as described above using **Selection Sort**.
4. In your `main()` function:
   - Create a vector of `Item` objects with test data.
   - Sort and print the vector using the following approaches:
     - Default comparison using the overloaded `<` operator.
     - A **function pointer** that compares `Item` objects by `name`.
     - A **functor** that compares `Item` objects by **reverse id**.
     - A **lambda function** that compares by **length of the name** field.

Exercise 3 .....

Height Checker: A school is trying to take an annual photo of all the students. For the photo, students are required to stand in a straight line in **non-decreasing order by height**.

You are given an integer array `heights` representing the **current order** in which the students are standing. Each `heights[i]` represents the height (in inches) of the i-th student in line.

Your task is to determine **how many students are not standing in the correct position** according to the required height order.

To solve this problem:

1. Use **Counting Sort** on the `heights` array to create a new array `expected`, representing how the students **should** be ordered (i.e., in non-decreasing order).
2. Compare the original `heights` array with the sorted `expected` array.
3. Return the **number of indices** where `heights[i] != expected[i]`.

Example:

Input: `heights = [1, 1, 4, 2, 1, 3]`
Output: 3

Explanation:

- `heights = [1, 1, 4, 2, 1, 3]`
- `expected = [1, 1, 1, 2, 3, 4]`
- Indices 2, 4, and 5 are mismatched.

Note:

- You **must** solve this problem using **counting sort** for efficiency, as the range of possible heights is small and bounded.
- Assume that **no student is taller than 7 feet**, so the maximum height value in the array will be 7.

Exercise 4

Sorting Product Prices in a Store: You are given an integer array `prices` where each element represents the price of a product in the store. Your task is to manually implement the **Radix Sort** algorithm to sort this array of product prices in ascending order.

Input:

- A list of positive integers `prices[]` where `prices[i]` represents the price of the i-th product in the store.

Output:

- The sorted array of product prices.

Example

```
INPUT:
prices = [170, 45, 75, 90, 802, 24, 2, 66]

OUTPUT:
[2, 24, 45, 66, 75, 90, 170, 802]
```

Constraints:

- The `prices` array contains at least one element.
- All product prices are **positive integers**.
- The number of products in the store can be large, so Radix Sort should be implemented efficiently.
- You must **first compute the number of digits in the maximum price** in the array to determine the number of passes needed in Radix Sort.

Exercise 5

Largest Perimeter of Triangle: You are given an integer array `nums`. Your task is to return the **largest perimeter** of a triangle that can be formed using any **three** side lengths from the array **such that the triangle has a non-zero area**.

If it is **not possible** to form any such triangle, return `0`.

Notes

- A triangle is valid if the **sum of the lengths of any two sides is greater than the length of the third side**.
- You must implement and use **Bubble Sort** to sort the array.
- To maximize the perimeter:
  - First, **sort the array in descending order**.
  - Then, **check each group of three consecutive numbers** in the sorted array.
- This approach is efficient for this purpose:
  - In a descending list, the **first valid triplet** you find will yield the **largest possible perimeter**.
  - There is no need to check all combinations, since smaller numbers after an invalid combination cannot form a valid triangle (due to the triangle inequality).

Example 1

Input: `nums = [2, 1, 2]`
After Bubble Sort (descending): `[2, 2, 1]`
Check: $2 + 1 > 2 \rightarrow$ Valid Triangle
Output: `5`

Example 2

Input: `nums = [1, 2, 1, 10]`
After Bubble Sort (descending): `[10, 2, 1, 1]`
Checked Combinations:

- $2 + 1 < 10 \rightarrow \times$
- $1 + 1 = 2 \rightarrow \times$

No valid triangle found.
Output: `0`

Reminder

Bubble Sort repeatedly compares and swaps adjacent elements to sort the array. While it is not efficient for large datasets, it is required here as a **learning constraint**.