Sorting

Sorting is the process of arranging elements in a specific order, typically in ascending or descending numerical or lexicographical order. It is a fundamental operation in computer science with applications in searching, data analysis, and algorithm optimization.

Efficient sorting improves the performance of other algorithms that require sorted input, such as binary search or merge-based algorithms.

1 Bubble Sort

Bubble Sort repeatedly steps through the list, compares adjacent elements and swaps them if they are in the wrong order.

```
// Bubble Sort
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n - 1; ++i)
        for (int j = 0; j < n - i - 1; ++j)
            if (arr[j] > arr[j + 1])
                std::swap(arr[j], arr[j + 1]);  // Swap if out of order
}
```

Time complexity: $O(n^2)$ Space complexity: $O(1)$

2 Insertion Sort

Insertion Sort builds the sorted array one item at a time by comparing and inserting the current element at the right position.

```
// Insertion Sort
// Sorts arr[] of size n in ascending order by inserting each element into its correct
    position
void insertionSort(int arr[], int n) {
    // Start from the second element (index 1), as the first element is trivially sorted
    for (int i = 1; i < n; ++i) {
        int key = arr[i];         // The element to insert into the sorted portion
        int j = i - 1;            // Index of the last element in the sorted portion

        // Shift elements in the sorted portion that are greater than key to the right
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j]; // Move element one position to the right
            --j;                  // Move to the next element on the left
        }

        // Insert key at its correct position
        arr[j + 1] = key;
    }
}
```

Time complexity: $O(n^2)$ Space complexity: $O(1)$

3 Selection Sort

Selection Sort repeatedly finds the minimum element from the unsorted part and puts it at the beginning.

```
// Selection Sort
// Sorts arr[] of size n in ascending order by selecting the minimum element at each step
void selectionSort(int arr[], int n) {
    // Move the boundary of the unsorted subarray one by one
    for (int i = 0; i < n - 1; ++i) {
        int min_idx = i; // Assume the current position has the minimum

        // Find the minimum element in the unsorted subarray
        for (int j = i + 1; j < n; ++j) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }

        // Swap the found minimum element with the first element of the unsorted part
        std::swap(arr[min_idx], arr[i]);
    }
}
```

Time complexity: $O(n^2)$ Space complexity: $O(1)$

4 Merge Sort

Merge Sort is a divide-and-conquer algorithm that divides the array into halves, sorts them recursively, and merges them.

```
// Merge two sorted halves: arr[l..m] and arr[m+1..r]
void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;     // Size of left subarray
    int n2 = r - m;         // Size of right subarray
    int L[n1], R[n2];       // Temporary arrays

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

    int i = 0, j = 0, k = l;

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

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

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

// Merge Sort: sorts arr[l..r] using merge()
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2; // Avoid overflow

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

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}
```

4.1 Merge Sort and the Master Method

Merge Sort uses a divide-and-conquer approach. It divides the array into two halves, recursively sorts each half, and then merges them. Its time recurrence is:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

This fits the general divide-and-conquer form:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

with parameters:

- $a = 2$ (number of subproblems),
- $b = 2$ (each subproblem is half the size),
- $f(n) = O(n)$ (cost of merging two sorted subarrays).

Let $d = 1$ since $f(n) = O(n^1)$, and compute:

$$c = \log_b a = \log_2 2 = 1$$

We now compare d and c:

- Since $d = c = 1$, this is **Case 2** of the Master Theorem.

By Case 2:

$$T(n) = O(n^d \log n) = O(n \log n)$$

The time complexity of Merge Sort is $O(n \log n)$, derived using the Master Method with $a = 2$, $b = 2$, and $f(n) = O(n^1)$. Space complexity is $O(n)$

5 Quick Sort

Quick Sort picks a pivot element, partitions the array around the pivot, and sorts the partitions recursively.

```cpp
// Partition function: places pivot at correct position
// and arranges smaller elements to the left and larger to the right
int partition(int arr[], int low, int high) {
    int pivot = arr[high];  // Choose the last element as pivot
    int i = low - 1;        // Index of smaller element

    // Traverse the array and rearrange elements
    for (int j = low; j < high; ++j) {
        if (arr[j] < pivot) {
            ++i;
            std::swap(arr[i], arr[j]);
        }
    }

    // Place pivot at its correct position
    std::swap(arr[i + 1], arr[high]);
    return i + 1;           // Return the partition index
}

// Quick Sort: recursively sorts arr[low..high]
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        // Partition the array and get pivot index
        int pi = partition(arr, low, high);

        // Recursively sort elements before and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
```

Average-Case Complexity of Quick Sort

Quick Sort is a divide-and-conquer algorithm that partitions an array around a pivot element such that elements less than the pivot go to the left subarray and greater elements to the right. It then recursively sorts each subarray.

The efficiency of Quick Sort depends heavily on the quality of the pivot selection. If the pivot splits the array into two equal halves at each step—i.e., is always the median—the recurrence becomes:

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + O(n)$$

This is the same recurrence as Merge Sort and yields an overall time complexity of:

$$T(n) = O(n \log n)$$

However, unlike Merge Sort, Quick Sort does not guarantee a balanced split. The algorithm randomly selects pivots or uses heuristics (like picking the first or last element), making it a **probabilistic algorithm**.

In the average case, random pivots tend to split the input into reasonably balanced partitions on average. The expected depth of the recursion tree remains $O(\log n)$, and each level performs $O(n)$ work for partitioning. Hence, the expected total running time is:

$$O(n \log n)$$

Quick Sort runs in $O(n \log n)$ time on average due to its expected balanced partitions, though in the worst case (e.g., always picking the smallest or largest element as pivot), it degrades to $O(n^2)$. Space complexity is $O(1)$

6 Heap Sort

Heap Sort converts the array into a max heap and extracts the maximum repeatedly.

```
void heapSort(std::vector<int>& arr) {
    // Create a max heap
    std::priority_queue<int> pq;

    // Insert all elements into the priority queue
    for (int num : arr) {
        pq.push(num);
    }

    // Extract elements from the heap and store in reverse order
    for (int i = arr.size() - 1; i >= 0; --i) {
        arr[i] = pq.top();
        pq.pop();
    }
}
```

To sort in ascending order using a **min-heap**, reverse the order of extraction or use a custom comparator:

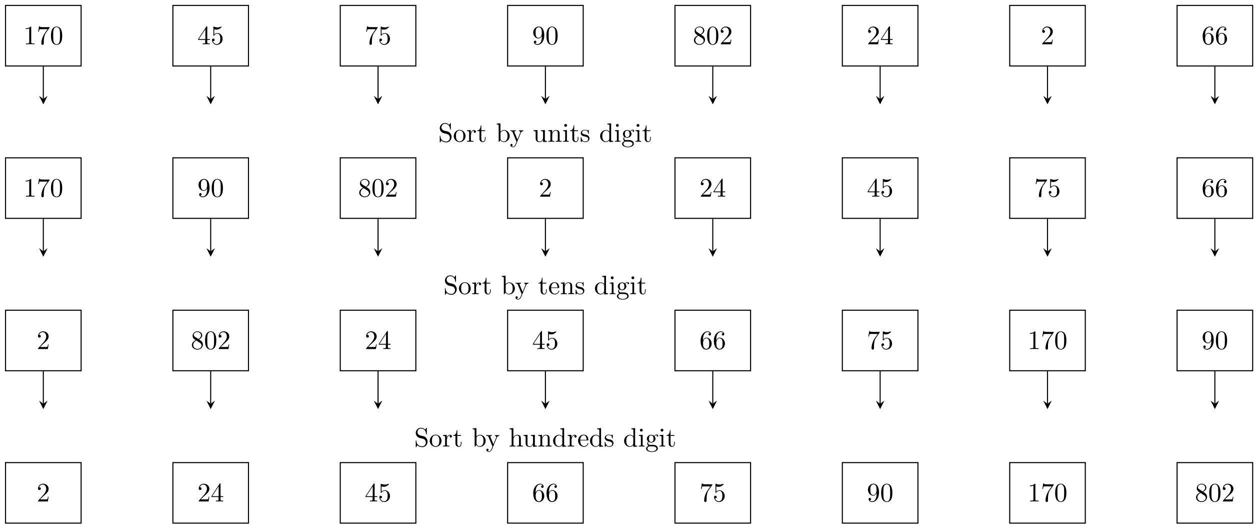
```
// Min-heap using greater<int> comparator
std::priority_queue<int, std::vector<int>, std::greater<int>> minHeap;
```

Since, heap operations are $O(logn)$, the time complexity is $O(nlogn)$. The space complexity is $O(1)$.

7 Radix Sort

Radix Sort is a non-comparison-based sorting algorithm that sorts **integers** by processing individual digits. It operates by performing a stable sort (often counting sort) on each digit, starting from the least significant digit (LSD) to the most significant digit (MSD).

7.1 Illustrated example



7.2 C++ implementation

```cpp
// Counting sort by digit represented by exp
void countingSort(int arr[], int n, int exp) {
    int output[n];          // Output array
    int count[10] = {0};    // Count array for digits 0-9

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

    // Update count[i] so it contains actual position of the digit in output
    for (int i = 1; i < 10; ++i)
        count[i] += count[i - 1];

    // Build the output array by placing elements at correct positions
    for (int i = n - 1; i >= 0; --i) {
        int digit = (arr[i] / exp) % 10;
        output[count[digit] - 1] = arr[i];
        count[digit]--;
    }

    // Copy output array back to arr[]
    for (int i = 0; i < n; ++i)
        arr[i] = output[i];
}

// Radix Sort: sorts arr[] using counting sort on each digit
void radixSort(int arr[], int n) {
    int max_val = *std::max_element(arr, arr + n); // Find the maximum number

    // Apply counting sort for each digit place
    for (int exp = 1; max_val / exp > 0; exp *= 10)
        countingSort(arr, n, exp);
}
```

Time Complexity: $O(d \cdot n)$ — linear with respect to input size if d is constant.
Space Complexity: $O(n)$