Data Structure Algorithms & Applications (CT-159)
Lab 07

Advanced Sorting Techniques

Objectives

The objective of the lab is to get students familiar with Advanced sorting techniques which include quick sort and merge sort.

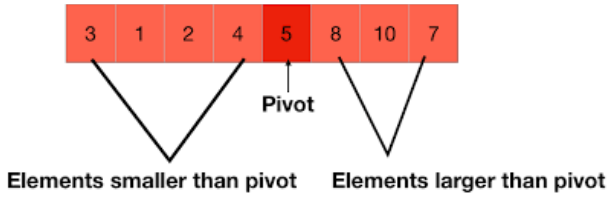
Tools Required

Dev C++ IDE

Course Coordinator –
Course Instructor –
Lab Instructor –
Prepared By Department of Computer Science and Information Technology
NED University of Engineering and Technology

Quick Sort Algorithm

Quicksort is an application of sorting using stacks. It also uses divide-and-conquer approach. In quicksort, a pivot element is selected and positioned in such a manner that all the elements smaller than the pivot element are to its left and all the elements greater than the pivot are to its right. The elements to the left and right of the pivot form two separate arrays and are sorted using this same approach. This continues until each subarray consists of a single element. The elements are merged to get the required sorted list. The pivot element can be the first, last, random, or median value in the given list.



Time Complexity: O(n log n) on average, $O(n^2)$ in the worst case.
Space Complexity: O(log n) for recursive calls.
Stability: Not stable by default.
Aglorithm main steps:

1. Pick a pivot element.
2. Partition the array such that elements less than the pivot go to the left and greater elements go to the right.
3. Recursively apply the process to the subarrays.

Applications: Often used in practice due to its in-place sorting and good average performance.

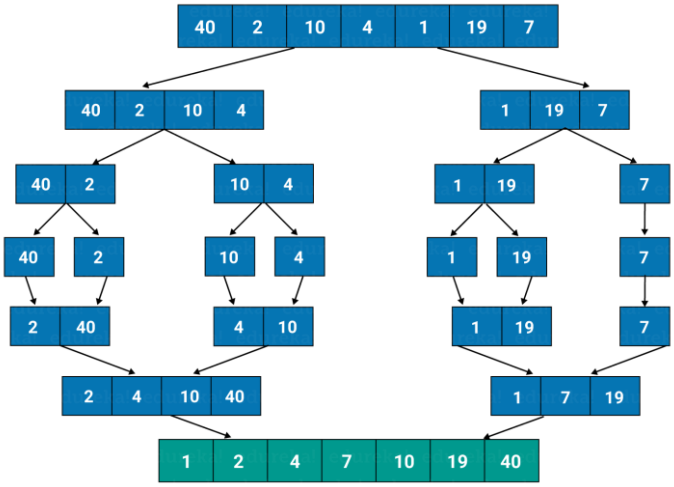
Example 01: Recursive Implementation of Quick Sort Algorithm

```
#include <iostream>
using namespace std;

// Function to swap two elements
void swap(int& a, int& b) {
   int temp = a;
   a = b;
   b = temp;
}

// Partition function: places the pivot element in its correct position
int partition(int arr[], int low, int high) {
   int pivot = arr[high]; // Choosing the last element as pivot
   int i = low - 1; // Index of smaller element

   for (int j = low; j < high; j++) {
      // If the current element is smaller than or equal to the pivot
      if (arr[j] <= pivot) {
         i++; // Move the smaller element's index forward
         swap(arr[i], arr[j]);
      }
   }
   // Place the pivot in its correct position
   swap(arr[i + 1], arr[high]);
   return i + 1; // Return the partition index
}

// Quick Sort function
void quickSort(int arr[], int low, int high) {
   if (low < high) {
      // Partition the array
      int pi = partition(arr, low, high);

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

// Function to print the array
void printArray(int arr[], int size) {
   for (int i = 0; i < size; i++) {
      cout << arr[i] << " ";
```

```
    }
    cout << endl;
}

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

    cout << "Unsorted array: ";
    printArray(arr, n);

    // Perform Quick Sort
    quickSort(arr, 0, n - 1);

    cout << "Sorted array: ";
    printArray(arr, n);

    return 0;
}
```

Merge Sort Algorithm

Merge Sort is a divide-and-conquer algorithm. It divides the input array into smaller subarrays, recursively sorts them, and then merges the sorted subarrays back together.



Time Complexity: O(n log n)
Space Complexity: O(n)
Stability: Stable (preserves the relative order of equal elements).
Algorithm main Steps:

1. Split the array into two halves.
2. Recursively sort each half.
3. Merge the two sorted halves into one sorted array.

Applications: Works well with linked lists and large datasets where stability is required.

Example 02: Recursive Implementation of Merge Sort Algorithm

```
#include <iostream>
using namespace std;

// Function to merge two subarrays
void merge(int arr[], int left, int mid, int right) {
    int n1 = mid - left + 1; // Size of the left subarray
    int n2 = right - mid;    // Size of the right subarray

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

    // Copy data to the temporary arrays L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    // Merge the temporary arrays back into arr[left..right]
```

```
    int i = 0; // Initial index of the left subarray
    int j = 0; // Initial index of the right subarray
    int k = left; // Initial index of the merged subarray

    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 any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

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

// Function to implement Merge Sort
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        // Find the middle point
        int mid = left + (right - left) / 2;

        // Recursively sort first and second halves
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Merge the sorted halves
        merge(arr, left, mid, right);
    }
}

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

// Main function
int main() {
    int arr[] = {12, 11, 13, 5, 6, 7};
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    cout << "Unsorted array: ";
    printArray(arr, arr_size);

    // Call mergeSort
    mergeSort(arr, 0, arr_size - 1);

    cout << "Sorted array: ";
    printArray(arr, arr_size);

    return 0;
}
```

Exercise

1. Implement non-recursive version of quick sort algorithm using stack.
2. Implement non-recursive version of the merge sort algorithm.
3. Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.
   Example 1: Input: intervals = [[1,3],[2,6],[8,10],[15,18]], Output: [[1,6],[8,10],[15,18]]
   Explanation: Since intervals [1,3] and [2,6] overlap, merge them into [1,6].
   Example 2: Input: intervals = [[1,4],[4,5]], Output: [[1,5]]
   Explanation: Intervals [1,4] and [4,5] are considered overlapping.
4. We define a harmonious array as an array where the difference between its maximum value and its minimum value is exactly 1. Given an integer array nums, return the length of its longest harmonious subsequence among all its possible subsequences. A subsequence of array is a sequence that can be derived from the array by deleting some or no elements without changing the order of the remaining elements.
   Example 1: Input: nums = [1,3,2,2,5,2,3,7], Output: 5
   Explanation: The longest harmonious subsequence is [3,2,2,2,3].
   Example 2: Input: nums = [1,2,3,4], Output: 2
   Example 3: Input: nums = [1,1,1,1], Output: 0
5. There is a car with capacity empty seats. The vehicle only drives east (i.e., it cannot turn around and drive west). You are given the integer capacity and an array trips where trips[i] = [numPassengersi, fromi, toi] indicates that the ith trip has numPassengersi passengers and the locations to pick them up and drop them off are fromi and toi respectively. The locations are given as the number of kilometers due east from the car's initial location. Return true if it is possible to pick up and drop off all passengers for all the given trips, or false otherwise.
   Example 1: Input: trips = [[2,1,5],[3,3,7]], capacity = 4, Output: false
   Example 2: Input: trips = [[2,1,5],[3,3,7]], capacity = 5, Output: true

Lab 05 Evaluation		
Student Name:		**Student ID:** **Date:**
Rubric	**Marks (25)**	**Remarks by teacher in accordance with the rubrics**
R1		
R2		
R3		
R4		
R5		