# DAA Practical File

Himanshu Jain

August 10, 2023

**Date:** August 10, 2023

## 1 Sorting Algorithms

Sorting is a fundamental process in computer science and data management. It involves arranging a collection of items or data elements in a specific order, which could be ascending (from smallest to largest) or descending (from largest to smallest). The primary objective of sorting is to make it easier to search for, access, or analyze the data efficiently.

There are various sorting algorithms available, each with its own characteristics in terms of efficiency and suitability for different types of data. Common sorting algorithms include bubble sort, selection sort, insertion sort, quick sort, merge sort, heap sort etc.

The choice of sorting algorithm depends on factors such as the size of the data set, its initial order, and the resources available (memory and processing power). Efficient sorting is essential for the performance of many computer applications, making sorting a fundamental concept in computer science and data processing.

### 1.1 Bubble Sort

Bubble sort is a basic sorting method used to arrange a list of items in a specific order, like ascending or descending. It's named for the way smaller items gradually move to their correct positions, making it one of the simplest sorting techniques. Although easy to understand, bubble sort is relatively inefficient for large data sets compared to more advanced sorting methods. It's often used in educational contexts to introduce sorting concepts in computer science.

#### 1.1.1 Methodology/Algorithm

The methodology of bubble sort in a series of points:

- Comparison: Bubble sort starts by comparing the first two elements in a list or array.

- Swap: If the first element is larger (in ascending order) than the second element, the two elements are swapped.
- Repeat: The algorithm then moves to the next pair of adjacent elements in the list and repeats the comparison and swapping process. • Passes: This process continues for multiple passes through the list, with each pass potentially moving the largest unsorted element to its correct position.
- Completing a Pass: A pass is complete when no more swaps are needed in that pass, indicating that the largest unsorted element has ”bubbled up” to its final position.
- Multiple Passes: Bubble sort repeats these passes until the entire list is sorted, meaning that no more swaps are required in a complete pass. • Termination: The algorithm terminates when a pass is completed without any swaps, indicating that the list is sorted.

### 1.1.2 Flow Chart

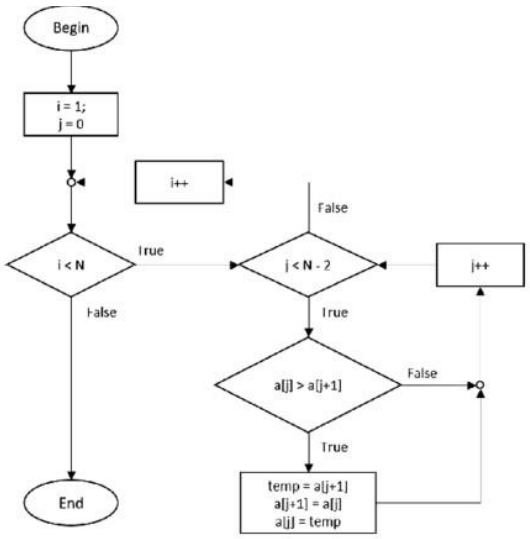


Figure 1: Bubble Sort

### 1.1.3 Code

```
#include <stdio.h>


void bubbleSort(int arr[], int n) {
int swapped; do { swapped = 0; for (int i = 1; i < n;
i++) { if (arr[i - 1] > arr[i]) { int temp = arr[i - 1]; arr[i -
1] = arr[i]; arr[i] = temp; swapped = 1;
}
}
} while (swapped);




} int }




main() { int n;

printf("Enter the number of elements: "); scanf("%d", &n); int

arr[n];

printf("Enter %d elements:\n", n); for (int i = 0; i < n;
i++) { scanf("%d", &arr[i]);
}

bubbleSort(arr, n); printf("Using Bubble
Sort,\n"); printf("Sorted elements: ");
for (int i = 0; i < n; i++) { printf("%d ", arr[i]);
}

return 0;
```

Listing 1: Bubble Sort

### 1.1.4 Output

```
Enter the number of elements: 5
Enter 5 elements:
6
2
9
4
1
Using Bubble Sort,
Sorted elements: 1 2 4 6 9
```

Figure 2: Bubble Sort Output

## 1.2 Selection Sort

Selection sort is a simple and intuitive sorting algorithm that works by repeatedly selecting the smallest (or largest, depending on the desired order) element from the unsorted part of the list and placing it in its correct position. This process continues until the entire list is sorted. Although not the most efficient sorting algorithm for large data sets, selection sort is straightforward to understand and implement, making it suitable for small to moderately sized lists or as an educational tool for learning about sorting techniques.

### 1.2.1 Methodology/Algorithm

The methodology of selection sort in a series of points:

- Unsorted and Sorted Sublists: Selection sort maintains two sublists within the list, one for sorted elements (initially empty) and one for unsorted elements (initially the entire list).
- Selection of the Minimum (or Maximum): The algorithm repeatedly scans the unsorted sublist to find the minimum (or maximum, depending on the desired order) element.
- Swap: Once the minimum (or maximum) element is found, it is swapped with the leftmost element in the unsorted sublist, effectively adding it to the sorted sublist.
- Shrinking the Unsorted Sublist: The boundary between the sorted and unsorted sublists moves one position to the right, and the process continues.

- Repeat: Steps 2-4 are repeated until the entire list is sorted, meaning that the unsorted sublist becomes empty.
- Termination: The algorithm terminates once the entire list is sorted. **1.2.2**

**Flow Chart**

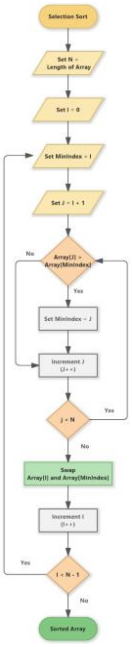


Figure 3: Selection Sort

### 1.2.3 Code

```
#include <stdio.h>

void selectionSort(int arr[], int n) { for (int i = 0; i < n - 1; i++) {
        int minIndex = i;

                for (int j = i + 1; j < n; j++) { if (arr[j] < arr[minIndex])
                        { minIndex = j;
                        }
                }

                int temp = arr[i];
```

```
                arr[i] = arr[minIndex]; arr[minIndex] = temp;
        }

} int
}
```

```
        main() { int n;

        printf("Enter the number of elements: "); scanf("%d", &n); int
        arr[n];

        printf("Enter %d elements:\n", n); for (int i = 0; i < n;
        i++) { scanf("%d", &arr[i]);
        } selectionSort(arr, n);

        printf("Using Selection Sort,\n"); printf("Sorted elements: ");
        for (int i = 0; i < n; i++) { printf("%d ", arr[i]);
        }

        return 0;
```

Listing 2: Selection Sort

### 1.2.4 Output

```
Enter the number of elements: 5
Enter 5 elements:
7
3
6
9
2
Using Selection Sort,
Sorted elements: 2 3 6 7 9
```

Figure 4: Selection Sort Output

## 1.3 Insertion Sort

Insertion sort is a straightforward and efficient sorting algorithm that builds the final sorted list one element at a time. It iterates through the input list, repeatedly taking the next unsorted element and inserting it into its correct position within the sorted portion of the list. While it may not be the fastest sorting algorithm for large datasets, insertion sort's simplicity and adaptability make it a practical choice for small to moderately sized lists, and it is often used as part of more complex sorting algorithms.

### 1.3.1 Methodology/Algorithm

The methodology of insertion sort in a series of points:

- Initial Division: Start with the first element of the list as the initially sorted part, and the remaining elements as the unsorted part. • Iterative Selection: Iterate through the unsorted part of the list, one element at a time.
- Comparisons and Insertion: For each unsorted element, compare it to the elements in the sorted part from right to left until the correct position is found.
- Shift: While comparing, shift the larger elements to the right to make room for the current element. • Insertion: Once the correct position is found, insert the current element into the sorted part of the list.
- Repeat: Continue this process until all elements are sorted. The unsorted part shrinks with each iteration, and the sorted part grows. • Termination: The algorithm terminates when the entire list is sorted, meaning that the unsorted part becomes empty. **1.3.2 Flow Chart**

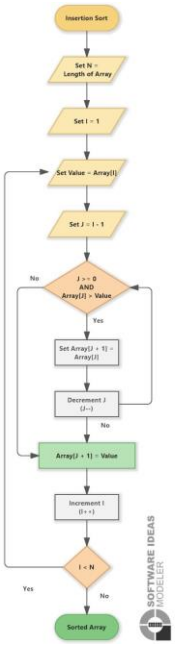


Figure 5: Insertion Sort

### 1.3.3 Code

```
#include <stdio.h>

void insertionSort(int arr[], int n) { for (int i = 1; i < n; i++) {
        int key = arr[i]; int j = i - 1;

                while (j >= 0 && arr[j] > key) { arr[j + 1] = arr[j]; j
                        = j - 1;
                }
                arr[j + 1] = key;
```

```
        }

} int
}
```

```
        main() { int n;

        printf("Enter the number of elements: "); scanf("%d", &n); int

        arr[n];

        printf("Enter %d elements:\n", n); for (int i = 0; i < n;
        i++) { scanf("%d", &arr[i]);
        } insertionSort(arr, n);

        printf("Using Insertion Sort,\n"); printf("Sorted elements: ");
        for (int i = 0; i < n; i++) { printf("%d ", arr[i]);
        }

        return 0;
```

Listing 3: Insertion Sort

### 1.3.4 Output

```
Enter the number of elements: 5
Enter 5 elements:
6
4
2
9
7
Using Insertion Sort,
Sorted elements: 2 4 6 7 9
```

Figure 6: Insertion Sort Output

## 1.4 Quick Sort

Quick sort is a highly efficient, divide-and-conquer sorting algorithm known for its speed and practicality. It works by dividing an unsorted array into two sub-arrays, sorting each sub-array independently, and then combining them to create a fully sorted array. Quick sort is often preferred for large data sets due to its excellent average-case performance, making it one of the most widely used sorting algorithms in computer science and software development.

### 1.4.1 Methodology/Algorithm

The quick sort algorithm in a series of points: • Pivot Selection: Choose a pivot element from the array. The pivot is used to partition the array into two sub-arrays.

- Partitioning: Rearrange the array so that elements less than the pivot are on the left, and elements greater than the pivot are on the right. The pivot is now in its final sorted position.

- Recursion: Recursively apply quick sort to the left and right sub-arrays created in the previous step. This process continues until each sub-array is small enough to be considered sorted (typically containing one or zero elements).
- Combine: As the recursion unwinds, the sub-arrays are combined to produce the fully sorted array.
- Termination: The algorithm terminates when the entire array is sorted.

### 1.4.2 Flow Chart

### 1.4.3 Code

```
#include <stdio.h>

void quickSort(int arr[], int low, int high) { if (low < high) { int pivot =

        partition(arr, low, high);

                quickSort(arr, low, pivot - 1); quickSort(arr, pivot + 1, high);
        }
}

int partition(int arr[], int low, int high) { int pivot = arr[high]; int i = (low
        - 1);

        for (int j = low; j <= high - 1; j++) { if (arr[j] < pivot) {
                        i++;

                        int temp = arr[i]; arr[i] = arr[j];
                        arr[j] = temp;
                }
        }

        int temp = arr[i + 1]; arr[i + 1] = arr[high];
        arr[high] = temp;

        return (i + 1);
}

int main() { int n;

        printf("Enter the number of elements: "); scanf("%d", &n); int
        arr[n]; printf("Enter %d elements:\n", n);
```

```
        for (int i = 0; i < n; i++) { scanf("%d", &arr[i]);
    }

    quickSort(arr, 0, n - 1);

    printf("Using Quick Sort,\n"); printf("Sorted elements: ");
    for (int i = 0; i < n; i++) { printf("%d ", arr[i]);
    }

    return 0;


}
```

Listing 4: Quick Sort

### 1.4.4 Output

```
Enter the number of elements: 5
Enter 5 elements:
9
5
7
3
1
Using Quick Sort,
Sorted elements: 1 3 5 7 9
```

Figure 7: Quick Sort Output

## 1.5 Merge Sort

Merge sort is a highly efficient and stable sorting algorithm known for its divideand-conquer approach. It works by dividing the unsorted list into smaller sublists, sorting each sub-list, and then merging them back together to create a fully sorted list. Merge sort's consistent performance and ability to handle large datasets make it a popular choice for various applications, including sorting large sets of data in computer science, software development, and beyond.

### 1.5.1 Methodology/Algorithm

The merge sort algorithm in a series of points:

- Divide: Divide the unsorted list into two equal halves (or approximately equal if the list has an odd number of elements).
- Recursion: Recursively apply the merge sort algorithm to each of the two halves, effectively dividing them into smaller sub-lists.
- Conquer: Continue dividing each sub-list until each sub-list contains one or zero elements, which are, by definition, sorted.
- Merge: Merge the sorted sub-lists back together, ensuring that the elements are arranged in the correct order.
- Repeat: Continue merging sub-lists until a single, fully sorted list is obtained.
- Termination: The algorithm terminates when the entire list is sorted.

### 1.5.2 Flow Chart

### 1.5.3 Code

```
#include <stdio.h>

void mergeSort(int arr[], int left, int right); void merge(int arr[], int left, int mid,
int right);

int main() { int n;

        printf("Enter the number of elements: "); scanf("%d", &n); int
        arr[n];

        printf("Enter %d elements:\n", n); for (int i = 0; i < n;
        i++) { scanf("%d", &arr[i]);
        }

        mergeSort(arr, 0, n - 1);

        printf("Using Merge Sort,\n"); printf("Sorted elements: ");
        for (int i = 0; i < n; i++) { printf("%d ", arr[i]);
```

21

22

23

24

```
        }

        return 0;
}

void mergeSort(int arr[], int left, int right) { if (left < right) { int mid = left +
        (right - left) / 2;

                mergeSort(arr, left, mid); mergeSort(arr, mid + 1, right);

                merge(arr, left, mid, right);
        }
}

void merge(int arr[], int left, int mid, int right) { int n1 = mid - left + 1; int n2 =
        right - mid; int L[n1], R[n2];

        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];
        }

        int i = 0, j = 0, k = left; while (i < n1 && j <
        n2) { if (L[i] <= R[j]) { arr[k] = L[i]; i++;
                } else { arr[k] = R[j]; j++;
                } k++;
        }

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

```
        while (j < n2) { arr[k] = R[j];
                j++; k++;
        }

}
```

Listing 5: Merge Sort

### 1.5.4 Output

```
Enter the number of elements: 6
Enter 6 elements:
6
5
4
3
2
1
Using Merge Sort,
Sorted elements: 1 2 3 4 5 6
```

Figure 8: Merge Sort Output

## 1.6 Heap Sort

Heap sort is an efficient and comparison-based sorting algorithm that leverages the properties of a binary heap data structure. It works by transforming the input array into a max-heap (for ascending order) or a min-heap (for descending order), and then repeatedly extracting the root element of the heap to build the sorted list. Heap sort is particularly useful for large data sets and is known for its stable and predictable performance characteristics, making it valuable in various computing applications and real-time systems.

### 1.6.1 Methodology/Algorithm

The methodology of the heap sort algorithm in a series of points: • Build a Heap: Create a binary heap from the unsorted array. This heap can be either a max-heap (for ascending order) or a min-heap (for descending order). The heapify process ensures that the largest (or smallest) element is at the root.

• Extract and Sort: Repeatedly extract the root element from the heap (which is the largest for max-heap or the smallest for min-heap) and place it at the end of the array. This process effectively builds the sorted list from the back. • Heapify Again: After each extraction, the heap property may be violated. Perform heapify (sift-down or sift-up) to restore the heap property. • Repeat: Continue extracting and sorting elements until the heap contains only one element, which is the smallest (for max-heap) or largest (for min-heap) element. The sorted list is built in reverse order. • Final Result: The sorted list is now fully constructed and in the desired order (ascending or descending).

### 1.6.2 Flow Chart

### 1.6.3 Code

```
#include <stdio.h>

void heapSort(int arr[], int n); void heapify(int arr[], int n,
int i);

int main() { int n;

        printf("Enter the number of elements: "); scanf("%d", &n); int
        arr[n];

        printf("Enter %d elements:\n", n); for (int i = 0; i < n;
        i++) { scanf("%d", &arr[i]);
        } heapSort(arr, n);

        printf("Using Heap Sort,\n"); printf("Sorted elements: ");
        for (int i = 0; i < n; i++) { printf("%d ", arr[i]);
        }

        return 0;
}
```

17
18
19
20
21
22
23
24
25
26
27
28

```
void


    heapSort(int arr[], int n) {
  for (int i = n / 2 - 1; i >= 0; i--) { heapify(arr, n, i);
  }

  for (int i = n - 1; i >= 0; i--) { int temp = arr[0]; arr[0]
      = arr[i]; arr[i] = temp;

      heapify(arr, i, 0);
  }
} void


    heapify(int arr[], int n, int i) { int largest = i;
  int   left = 2 * i + 1;

  int   right = 2 * i + 2;

  if
      (left < n && arr[left] > arr[largest]) { largest = left;
  } if


      (right < n && arr[right] > arr[largest]) { largest = right;
  } if




      (largest != i) { int swap = arr[i]; arr[i] =
        arr[largest]; arr[largest] = swap;
        heapify(arr, n, largest);
  }
}
```

Listing 6: Heap Sort

### 1.6.4 Output

```
Enter the number of elements: 6
Enter 6 elements:
9
1
8
2
7
3
Using Heap Sort,
Sorted elements: 1 2 3 7 8 9
```

Figure 9: Heap Sort Output

**Date:** August 28, 2023

# 2 Matrix Multiplication

Matrix multiplication is a fundamental operation in linear algebra and mathematics. It involves multiplying two matrices together to produce a new matrix. This operation plays a crucial role in various fields, including physics, computer graphics, engineering, and data analysis. Matrix multiplication allows for the transformation and manipulation of data and is an essential concept in understanding complex systems and solving systems of linear equations. It forms the basis for many algorithms and computations in diverse scientific and technological applications.

## 2.1 Methology/Algorithm

Matrix multiplication, also known as matrix product, is performed by following these steps:

- Matrix Dimensions: Ensure that the matrices you want to multiply are compatible, meaning the number of columns in the first matrix must be equal to the number of rows in the second matrix.
- Element-wise Multiplication: Multiply each element of the row of the first matrix by the corresponding element in the column of the second matrix. • Summation: Sum up all the products obtained in the previous step for each element of the resulting matrix.
- Resulting Matrix: Repeat steps 2 and 3 for all elements of the resulting matrix. Each element in the resulting matrix is calculated using a row from the first matrix and a column from the second matrix. • Result: The resulting matrix will have dimensions determined by the number of rows from the first matrix and the number of columns from the second matrix.

  Matrix multiplication is not commutative, meaning the order of multiplication matters, and the result can be different if the order of the matrices is reversed.

## 2.2 Flow Chart

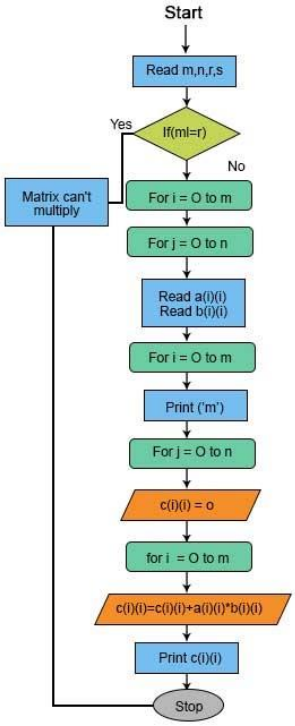


Figure 10: Matrix Multiplication Flowchart

## 2.3 Code

```
#include <stdio.h>

void multiplyMatrices(int A[][10], int B[][10], int result[][10], int rowsA, int colsA, int
      colsB) {
        for (int i = 0; i < rowsA; i++) { for (int j = 0; j < colsB; j++) {
                result[i][j] = 0;
                        for (int k = 0; k < colsA; k++) {
                                result[i][j] += A[i][k] * B[k][j];
                        }
                }
        }
}

int main() {
```

```
    int rowsA, colsA, rowsB, colsB;

    printf("Enter the number of rows and columns for
        Matrix A: "); scanf("%d %d", &rowsA,
    &colsA);

    printf("Enter the number of rows and columns for
        Matrix B: "); scanf("%d %d", &rowsB,
    &colsB);

    if (colsA != rowsB) { printf("Matrix multiplication is not
            possible.\n");
            return 1;
    }

    int matrixA[10][10], matrixB[10][10], resultMatrix[10][10];

    printf("Enter the elements of Matrix A:\n"); for (int i = 0; i < rowsA;
    i++) { for (int j = 0; j < colsA; j++) { scanf("%d", &matrixA[i][j]); }
    }

    printf("Enter the elements of Matrix B:\n"); for (int i = 0; i < rowsB;
    i++) { for (int j = 0; j < colsB; j++) { scanf("%d", &matrixB[i][j]); }
    }

    multiplyMatrices(matrixA, matrixB, resultMatrix, rowsA, colsA, colsB);

    printf("Resultant Matrix:\n");
    for (int i = 0; i < rowsA; i++) { for (int j = 0; j < colsB; j++) {
                                printf("%d ", resultMatrix[i][j]);
            } printf("\n");
    }
    return 0;









}
```

Listing 7: Matrix Multiplication

## 2.4 Output

```
Enter the number of rows and columns for Matrix A: 2
2
Enter the number of rows and columns for Matrix B: 2
2
Enter the elements of Matrix A:
1
2
3
4
Enter the elements of Matrix B:
4
3
2
1
Resultant Matrix:
8 5
20 13
```

Figure 11: Matrix Multiplication Output