# Algorithms 12th Project

*Arab Academy for Science and Technology*

**Student:**

- Gasser Amr

**Supervised By:**

Eng.Engy

## Introduction:

This documentation explains the key algorithmic concepts implemented in the Arab Academy for Science and Technology algorithms project. The project focuses on two main areas: sorting algorithms and Huffman coding, providing educational understanding of these fundamental computer science concepts.

# Sorting Algorithms Implemented:

## 1. Bubble Sort

**Concept:** Bubble Sort is one of the simplest sorting algorithms that works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order.

**Process:**

- Compare adjacent elements in the array
- Swap elements if they are in the wrong order
- Repeat until no more swaps are needed

**Characteristics:**

- Time Complexity: $O(n^2)$ in worst and average cases
- Space Complexity: $O(1)$
- Stable: Yes (maintains relative order of equal elements)
- In-place: Yes (requires minimal additional memory)

**Best used for:** Small datasets or nearly sorted arrays

## 2. Insertion Sort

**Concept:** Insertion Sort builds the final sorted array one element at a time by comparing each new element with already sorted elements.

**Process:**

- Start with the second element (consider the first element as sorted)
- Compare the current element with previous elements
- Shift greater elements to the right
- Insert the current element in its correct position
- Repeat for all elements

**Characteristics:**

- Time Complexity: $O(n^2)$ worst/average case, $O(n)$ best case
- Space Complexity: $O(1)$

- Stable: Yes
- In-place: Yes

**Best used for:** Small datasets or partially sorted arrays

## 3. Selection Sort

**Concept:** Selection Sort divides the input into a sorted and an unsorted region, repeatedly selecting the smallest element from the unsorted region and moving it to the end of the sorted region.

**Process:**

- Find the minimum element in the unsorted portion
- Swap it with the element at the beginning of the unsorted portion
- Expand the sorted portion by one element

**Characteristics:**

- Time Complexity: $O(n^2)$ in all cases
- Space Complexity: $O(1)$
- Stable: No
- In-place: Yes

**Best used for:** Small arrays where simplicity is more important than efficiency

## 4. Merge Sort

**Concept:** Merge Sort is a divide-and-conquer algorithm that divides the input array into two halves, recursively sorts them, and then merges the sorted halves.

**Process:**

- Divide the array into two halves
- Recursively sort both halves
- Merge the sorted halves into a single sorted array

**Characteristics:**

- Time Complexity: $O(n \log n)$ in all cases
- Space Complexity: $O(n)$
- Stable: Yes
- In-place: No (requires additional memory for merging)

**Best used for:** Large datasets where stable sorting is required

## 5. Quick Sort

**Concept:** Quick Sort is another divide-and-conquer algorithm that selects a 'pivot' element and partitions the array around the pivot.

**Process:**

- Choose a pivot element
- Partition the array so elements less than pivot are on left, greater on right
- Recursively apply the above steps to sub-arrays

**Characteristics:**

- Time Complexity: O(n log n) average/best case, $O(n^2)$ worst case
- Space Complexity: O(log n) due to recursion stack
- Stable: No
- In-place: Yes

**Best used for:** Large unsorted arrays when average-case performance matters

## 6. Heap Sort

**Concept:** Heap Sort uses a binary heap data structure to sort elements by first building a max heap and then extracting elements one by one.

**Process:**

- Build a max heap from the input array
- Swap the root (maximum element) with the last element
- Reduce heap size and heapify the root
- Repeat until heap size becomes 1

**Characteristics:**

- Time Complexity: O(n log n) in all cases
- Space Complexity: O(1)
- Stable: No
- In-place: Yes

**Best used for:** When stable sorting is not required and guaranteed O(n log n) performance is needed

## 7. Counting Sort

**Concept:** Counting Sort is a non-comparison-based sorting algorithm that works by counting the occurrences of each element and using that information to place elements in correct positions.

**Process:**

- Count occurrences of each element
- Calculate cumulative count
- Place each element in its correct position using the counts

**Characteristics:**

- Time Complexity: O(n+k) where k is the range of input
- Space Complexity: O(n+k)
- Stable: Yes (if implemented carefully)
- In-place: No

**Best used for:** Integer data with a small range of values

## 8. Radix Sort

**Concept:** Radix Sort sorts elements by processing individual digits from least significant to most significant digits, using a stable sort (usually Counting Sort) as a subroutine.

**Process:**

- Start from the least significant digit
- Sort elements based on the current digit position
- Move to the next significant digit
- Repeat until the most significant digit

**Characteristics:**

- Time Complexity: O(d(n+k)) where d is the number of digits and k is the range of each digit
- Space Complexity: O(n+k)
- Stable: Yes
- In-place: No

**Best used for:** Integer or string data with fixed-length keys

# Huffman Coding:

## Concept

Huffman coding is a lossless data compression algorithm that assigns variable-length codes to input characters based on their frequencies. Characters that occur more frequently are assigned shorter codes.

#### Key Components

### *1. Frequency Table Generation*

- Count the occurrences of each character in the input text
- Create a frequency table mapping characters to their frequencies

### *2. Huffman Tree Construction*

- Create leaf nodes for each character with its frequency
- Build a binary tree where:
  - Lower frequency nodes are positioned deeper in the tree
  - Higher frequency nodes are closer to the root
- The tree is constructed bottom-up by repeatedly merging the two nodes with lowest frequencies

### *3. Code Assignment*

- Traverse the tree from root to each leaf
- Assign '0' for left branches and '1' for right branches
- Each character's code is the sequence of 0s and 1s in the path from root to character's leaf node

### *4. Encoding Process*

- Replace each character in the original text with its Huffman code
- The resulting bitstream is the compressed representation

## *5. Decoding Process*

- Start at the root of the Huffman tree
- Follow the path indicated by each bit (0 = left, 1 = right)
- When a leaf node is reached, output its character and return to the root

## Characteristics

- Time Complexity: O(n log n) for building the tree, O(n) for encoding/decoding
- Space Complexity: O(n) for storing the tree and codes
- Compression ratio depends on the distribution of character frequencies
- Optimal prefix-free codes (no code is a prefix of another code)

## Applications

- Text compression
- Image compression (as part of larger compression schemes)
- Data transmission to reduce bandwidth requirements
- File compression utilities

## Technologies Used:

- JavaScript for algorithm logic and animation
- HTML/CSS for user interface
- D3.js for Huffman tree visualization

## Conclusion:

This project successfully creates an interactive learning environment for fundamental algorithms. The sorting visualizer provides clear demonstrations of various sorting techniques, while the Huffman coding tool offers comprehensive insight into compression algorithms. The visual approach significantly enhances understanding of complex algorithmic processes, making abstract concepts tangible and easier to comprehend.

The implementation demonstrates how modern web technologies can be leveraged to create effective educational tools, providing students with hands-on experience in algorithm analysis and visualization. The project serves as both a learning resource and a foundation for further exploration of algorithm visualization techniques.