Lab 2: Sorting and Searching

ICT 2202: Algorithm Analysis Design and Lab

Submitted By

Group 1

Tahshin Islam Tuhin(ID: 1957)

Nobin Chandra(ID: 1958)

Md. Shaon Khan(ID: 1984)

Submitted To

Professor Fahima Tabassum, PhD

Institute of Information Technology

Jahangirnagar University

Submission Date: 14/10/2025

1.1 Overview

This laboratory assignment involves the practical implementation and comparative analysis of fundamental sorting and searching algorithms in computer science. The assignment is divided into two main parts: sorting algorithm analysis and searching algorithm implementation.

In the first part, three sorting algorithms—Selection Sort, Insertion Sort, and Bubble Sort—are implemented and tested on a dataset containing 25 order records. Each record consists of an order ID and a corresponding quantity. The objective is to sort these records in ascending order based on order ID. The relative performance and efficiency of each algorithm are analyzed through empirical time measurements.

The second part focuses on implementing two searching techniques—Sequential Search and Binary Search. These algorithms are used to identify orders eligible for promotional discounts based on quantity thresholds. This section demonstrates how different search strategies perform in practical scenarios and emphasizes the importance of selecting appropriate algorithms based on specific data characteristics and requirements.

Throughout this assignment, hands-on experience is gained in algorithm implementation and performance measurement using Python's time module. Additionally, a comparative analysis between the theoretical time complexity and practical performance is conducted. This exercise helps in understanding algorithm behavior, time complexity considerations, and the factors influencing the choice of algorithms for real-world applications.

1.2 Problem 01: Sorting Algorithm Implementation

1.2.1 Algorithms

Selection Sort Algorithm:

1. Start with the first element as the current minimum.
2. Scan the unsorted portion of the array to find the actual minimum element.
3. Swap the minimum element found with the first unsorted element.
4. Move the boundary of the sorted subarray one element to the right.
5. Repeat steps 2-4 until the entire array is sorted.

Time Complexity: $O(n^2)$ in all cases (best, average, worst)

Space Complexity: $O(1)$ - in-place sorting

Insertion Sort Algorithm:

1. Start from the second element, considering the first element as sorted.
2. Compare the current element with elements in the sorted subarray.
3. Shift all larger elements in the sorted subarray one position to the right.
4. Insert the current element into its correct position.
5. Move to the next element and repeat the process until the entire array is sorted.

Time Complexity: $O(n^2)$ worst-case, $O(n)$ best-case (already sorted)

Space Complexity: $O(1)$ - in-place sorting

Bubble Sort Algorithm:

1. Start from the first element.
2. Compare adjacent elements in pairs.
3. Swap them if they are in the wrong order (the larger element "bubbles" up).
4. After each pass, the largest element reaches its correct position.
5. Repeat the process until a complete pass occurs with no swaps.

Time Complexity: $O(n^2)$ in worst and average cases, $O(n)$ best-case

Space Complexity: $O(1)$ - in-place sorting

1.2.2 Code

```
import time

# Selection Sort
def selectionSort(arr):
    n = len(arr)
    for i in range(n-1):
        min_index = i
        for j in range(i+1, n):
            if arr[j][0] < arr[min_index][0]:
                min_index = j
        arr[i], arr[min_index] = arr[min_index], arr[i]
    return arr

# Insertion Sort
def insertionSort(arr):
    n = len(arr)
    for i in range(1, n):
        key = arr[i]
        j = i - 1
        while j >= 0 and arr[j][0] > key[0]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key
    return arr

# Bubble Sort
def bubbleSort(arr):
    n = len(arr)
    for i in range(n - 1):
        for j in range(n - i - 1):
            if arr[j][0] > arr[j + 1][0]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr

# Dataset
arr = [
    (1025, 4),(1011, 2),(1033, 5),(1008, 1),(1042,
7),
    (1018, 3),(1055, 2),(1002, 6),(1021, 8),(1038,
1),
    (1015, 4),(1049, 5),(1029, 2),(1006, 9),(1052,
3),
    (1031, 6),(1012, 2),(1004, 7),(1040, 5),(1023,
4),
    (1010, 3),(1050, 8),(1009, 1),(1035, 9),(1027,
2)
]
```

```
arrSelectionSort = arr.copy()
arrInsertionSort = arr.copy()
arrBubbleSort = arr.copy()


def printArr(arr, times):
    for i in arr:
        print(f"({i[0]}:{i[1]}),", end=" ")
    print(f"\nTime needed: {times:.10f}
seconds\n")


print("Before sorting:")
for i in arr:
    print(f"({i[0]}:{i[1]}),", end=" ")
print("\n\n")


# Selection Sort
start = time.perf_counter()
arrSelectionSort = selectionSort(arrSelectionSort)
selectionSortTime = time.perf_counter() - start
print("Selection Sort,")
printArr(arrSelectionSort, selectionSortTime)


# Insertion Sort
start = time.perf_counter()
arrInsertionSort = insertionSort(arrInsertionSort)
insertionSortTime = time.perf_counter() - start
print("Insertion Sort,")
printArr(arrInsertionSort, insertionSortTime)


# Bubble Sort
start = time.perf_counter()
arrBubbleSort = bubbleSort(arrBubbleSort)
bubbleSortTime = time.perf_counter() - start
print("Bubble Sort,")
printArr(arrBubbleSort, bubbleSortTime)
```

1.2.3 Output

```
Before sorting:

(1025:4), (1011:2), (1033:5), (1008:1), (1042:7), (1018:3), (1055:2), (1002:6), (1021:8), (1038:1),
(1015:4), (1049:5), (1029:2), (1006:9), (1052:3), (1031:6), (1012:2), (1004:7), (1040:5), (1023:4),
(1010:3), (1050:8), (1009:1), (1035:9), (1027:2),



Selection Sort,

(1002:6), (1004:7), (1006:9), (1008:1), (1009:1), (1010:3), (1011:2), (1012:2), (1015:4), (1018:3),
(1021:8), (1023:4), (1025:4), (1027:2), (1029:2), (1031:6), (1033:5), (1035:9), (1038:1), (1040:5),
(1042:7), (1049:5), (1050:8), (1052:3), (1055:2),

Time needed: 0.0000206000 seconds


Insertion Sort,

(1002:6), (1004:7), (1006:9), (1008:1), (1009:1), (1010:3), (1011:2), (1012:2), (1015:4), (1018:3),
(1021:8), (1023:4), (1025:4), (1027:2), (1029:2), (1031:6), (1033:5), (1035:9), (1038:1), (1040:5),
(1042:7), (1049:5), (1050:8), (1052:3), (1055:2),

Time needed: 0.0000208000 seconds


Bubble Sort,

(1002:6), (1004:7), (1006:9), (1008:1), (1009:1), (1010:3), (1011:2), (1012:2), (1015:4), (1018:3),
(1021:8), (1023:4), (1025:4), (1027:2), (1029:2), (1031:6), (1033:5), (1035:9), (1038:1), (1040:5),
(1042:7), (1049:5), (1050:8), (1052:3), (1055:2),

Time needed: 0.0000267000 seconds
```

1.2.4 Explanation

All three sorting algorithms—Selection Sort, Insertion Sort, and Bubble Sort—successfully sorted the dataset of 25 order records in ascending order based on order ID, while preserving the integrity of the (order_id, quantity) pairs.

Time Complexity Analysis

- Selection Sort: $O(n^2)$ in all cases, performing approximately 300 comparisons for $n = 25$.
- Insertion Sort: $O(n^2)$ worst-case, but $O(n)$ best-case when the data is nearly sorted.
- Bubble Sort: $O(n^2)$ worst-case, with around 300 comparisons and multiple swaps.

Space Complexity

All three algorithms have a space complexity of $O(1)$, as they sort the data in-place without requiring additional memory.

Performance Results Analysis

- Selection Sort achieved the fastest time of 0.0000206 seconds, benefiting from its predictable $O(n^2)$ comparisons and minimal swapping overhead—only $n - 1$ swaps total.
- Insertion Sort closely followed at 0.0000208 seconds, taking advantage of its efficiency with the partially ordered dataset.
- Bubble Sort was the slowest at 0.0000267 seconds, due to its multiple passes and frequent swaps, resulting in more element movements.

The very small time differences, in microseconds, align with the small dataset size ($n = 25$), where all $O(n^2)$ algorithms perform adequately. For larger datasets, these differences would become more significant. In such cases, Insertion Sort generally outperforms the others on partially sorted data, while Selection Sort offers consistent performance regardless of initial ordering.

1.3 Problem 02: Searching Algorithm Implementation

1.3.1 Algorithms

1. Start from the first element of the array.
2. Iterate through each element sequentially.
3. Check if the element meets the search criteria (quantity > 3).
4. If the condition is satisfied, add the element to the result list.
5. Continue this process until all elements have been checked.
6. Return the filtered result list.

Time Complexity: O(n) - linear search through all elements

Space Complexity: O(k) - where k is number of elements meeting criteria

Binary Search Algorithm:

1. Require a sorted array as input (sorted by order_id).
2. Initialize low = 0 and high = n - 1.
3. While low is less than or equal to high:
    - Calculate mid = (low + high) // 2.
    - If arr[mid][0] == target_order_id, return the element.
    - Else if arr[mid][0] < target_order_id, search the right half by setting low = mid + 1.
    - Otherwise, search the left half by setting high = mid - 1.
4. Return None if the element is not found.

Time Complexity: O(log n) for single search

Space Complexity: O(1) for single search

1.3.2 Implementation Approach

The implementation demonstrates two different searching strategies:

- **Sequential Search**: Directly filters the dataset in a single pass
- **Binary Search**: Performs individual lookups for each order ID (25 searches) and filters results.

1.3.3 Code

```
import time

# Sequential Search
def sequentialSearch(arr, number):
    result = []
    for order in arr:
        if order[1] > number:
            result.append(order)
    return result

# Binary Search (for demonstration, assuming we search specific order_ID)
# We'll use it to find the quantity of a given order_ID
def binarySearch(arr, order_id):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid][0] == order_id:
            return arr[mid]
        elif arr[mid][0] < order_id:
            low = mid + 1
        else:
            high = mid - 1
    return None

# Printing helper
def printArr(arr, times):
    for i in arr:
        print(f"({i[0]}:{i[1]}),", end=" ")
    print(f"\tTime needed: {times:.10f} seconds\n")

print("Orders eligible for 50% discount (quantity > 3):\n")
```

```
# Sorted order list (from your previous sorting)
orders = [
    (1002, 6),(1004, 7),(1006, 9),(1008, 1),(1009, 1),
    (1010, 3),(1011, 2),(1012, 2),(1015, 4),(1018, 3),
    (1021, 8),(1023, 4),(1025, 4),(1027, 2),(1029, 2),
    (1031, 6),(1033, 5),(1035, 9),(1038, 1),(1040, 5),
    (1042, 7),(1049, 5),(1050, 8),(1052, 3),(1055, 2)
]

# Sequential Search Timing
start = time.perf_counter()
discountSequential = sequentialSearch(orders, 3)
sequentialTime = time.perf_counter() - start
print("Sequential Search:")
printArr(discountSequential, sequentialTime)

# Binary Search Timing (we search for each order_ID to simulate lookups)
start = time.perf_counter()
discountBinary = []
for order in orders:
    found = binarySearch(orders, order[0])
    if found and found[1] > 3:
        discountBinary.append(found)
binaryTime = time.perf_counter() - start

print("Binary Search:")
printArr(discountBinary, binaryTime)
```

1.3.4 Output

```
Orders eligible for 50% discount (quantity > 3):


Sequential Search:
(1002:6), (1004:7), (1006:9), (1015:4), (1021:8), (1023:4), (1025:4), (1031:6), (1033:5), (1035:9),
(1040:5), (1042:7), (1049:5), (1050:8), Time needed: 0.0000072000 seconds


Binary Search:
(1002:6), (1004:7), (1006:9), (1015:4), (1021:8), (1023:4), (1025:4), (1031:6), (1033:5), (1035:9),
(1040:5), (1042:7), (1049:5), (1050:8), Time needed: 0.0000218000 seconds
```

1.3.5 Explanation

Both searching algorithms successfully identified the same 14 orders with quantities greater than 3, qualifying them for the 50% discount.

Time Complexity Analysis

- Sequential Search: $O(n)$ — requires a single pass through all 25 elements.
- Binary Search: Approximately $O(n\log\ n)$ in this implementation, since it performs 25 separate searches, each taking $O(\log\ 25) \approx 5$ operations, totaling around 125 operations.

Space Complexity

- Sequential Search: $O(k)$, where $k = 14$ is the number of qualifying orders stored.
- Binary Search: $O(k)$ for storing results, plus $O(1)$ space for individual searches.

Performance Results Analysis

- Sequential Search showed superior performance, completing in 0.0000072 seconds with just 25 comparisons in a single pass.
- Binary Search was comparatively slower at 0.0000218 seconds, due to executing 25 separate binary searches, each with logarithmic time complexity.

This analysis underscores that for small datasets or when searching for multiple elements, Sequential Search can be more efficient than repeatedly applying Binary Search.

1.4 Discussion

This laboratory exercise provided valuable insights into the practical implementation and performance characteristics of fundamental sorting and searching algorithms. The comparative analysis highlights important considerations for effective algorithm selection in real-world scenarios.

Sorting Algorithms Performance

The sorting experiments confirmed that although Selection Sort, Insertion Sort, and Bubble Sort share an $O(n^2)$ theoretical time complexity, their practical performances differ notably. Selection Sort was the fastest (0.0000206s), closely followed by Insertion Sort (0.0000208s), while Bubble Sort was the slowest (0.0000267s). This ranking reflects Selection Sort's minimal swapping overhead (exactly $n-1$ swaps), Insertion Sort's efficiency on partially ordered data, and Bubble Sort's less efficient multiple passes with numerous adjacent comparisons. All three demonstrated $O(1)$ space complexity, performing in-place sorting without additional memory requirements.

Searching Algorithms Insights

The searching experiments produced unexpected but insightful results. Sequential Search (0.0000072s) significantly outperformed Binary Search (0.0000218s) in filtering tasks despite Binary Search's superior $O(\log n)$ theoretical complexity for individual lookups. This outcome underscores the importance of selecting algorithms aligned with specific use cases. Binary Search was inefficient here due to repeated individual searches, resulting in $O(n\log n)$ time complexity for bulk filtering. In contrast, Sequential Search efficiently completed the filtering in a single $O(n)$ pass.

Algorithm Selection

For sorting, Selection Sort is well suited to memory-limited environments, Insertion Sort performs optimally on nearly sorted datasets, and Bubble Sort is generally the least efficient option. For searching, Binary Search excels in single-element lookups on sorted data, whereas Sequential Search is preferable for filtering tasks. This exercise emphasizes that theoretical complexity alone is insufficient; practical factors such as data properties, operation type, and implementation context must guide algorithm choice.