# Homework 4 Solutions

March 2025

## 1 Problem 1: Bubble Sort Analysis

**Pseudocode**

```
void bubbleSort(int arr[], int n) {
    bool swapped;
    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }
        // If no elements were swapped, the array is already sorted
        if (!swapped)
            break;
    }
}
```

**Best Case and Worst-Case Time Efficiency**
**Best Case (Time Complexity):** $O(n)$

- **Explanation:** In the best case, the array is already sorted. In this scenario, Bubble Sort makes one complete pass through the array without making any swaps. After the first pass, it recognizes that the array is sorted and terminates early. This gives a time complexity of $O(n)$, since the algorithm performs $n$ comparisons and no swaps in one pass.
- **Number of Swaps (Best Case):** 0 swaps. No elements need to be swapped as the array is already in sorted order.

**Worst Case (Time Complexity):** $O(n^2)$

- **Explanation:** In the worst case, the array is sorted in reverse order. In this situation, every pair of adjacent elements needs to be swapped to get the correct order. For an array of $n$ elements, the algorithm makes $(n-1)$ comparisons in the first pass, $(n-2)$ in the second pass, and so on, leading to approximately $\frac{n(n-1)}{2}$ comparisons, which results in $O(n^2)$ time complexity.

- **Number of Swaps (Worst Case):** $O(n^2)$ swaps. Every comparison results in a swap since all elements are in the reverse order, requiring the maximum number of swaps.

**Swaps Efficiency**
**Best Case (Swaps Efficiency):** $O(1)$

- In the best case, no swaps are needed. The algorithm checks the array once and exits early, so the swap count is minimal (zero swaps).
- **Best Case Justification:** In the best case (already sorted array), the algorithm makes one pass through the array and detects no swaps are necessary. It can then terminate early, giving it a linear $O(n)$ time complexity.

**Worst Case (Swaps Efficiency):** $O(n^2)$

- In the worst case, every adjacent pair needs to be swapped. The number of swaps equals the number of comparisons, which is $O(n^2)$.
- **Worst Case Justification:** In the worst case (reverse order array), the algorithm must make $n$ passes through the array, performing approximately $\frac{n(n-1)}{2}$ comparisons and swaps, which results in a quadratic $O(n^2)$ time complexity. Each adjacent pair must be swapped until the array is fully sorted.

## 2 Problem 2: Bubble Sort - Loop Invariant and Correctness Proof

A **loop invariant** is a condition that holds true before and after each iteration of the loop. For Bubble Sort:

**Invariant:** After the $i$-th iteration, the last $i$ elements of the array are in sorted order.

**Proof of Correctness (using the loop invariant):**

- **Initialization:** Before the first iteration, no elements are guaranteed to be in the correct order.
- **Maintenance:** After each pass through the array, the largest unsorted element is *bubbled* to its correct position at the end of the array.
- **Termination:** Once the loop terminates, the entire array is sorted, and the invariant holds for the whole array.

## 3 Problem 3: Building a Max Heap

**Pseudocode**

```
for i = n/2 down to 1:
    maxHeapify(A, i)

function maxHeapify(A, i):
    left = 2 * i
    right = 2 * i + 1
    largest = i

    if left  n and A[left] > A[largest]:
        largest = left
    if right  n and A[right] > A[largest]:
        largest = right
    if largest  i:
        swap(A[i], A[largest])
        maxHeapify(A, largest)
```

**Worst-Case Time Complexity:** $O(n)$

The time complexity for building a max-heap is often miscalculated as $O(n \log n)$ due to the $O(\log n)$ time required for the heapify operation. However, this estimate is not tight. In reality, the complexity is $O(n)$ because the number of nodes decreases as we ascend the heap.

The maximum height of a binary heap is $h = \log n$. Nodes at greater heights require more time to heapify, but they are fewer in number. Most nodes are near the bottom of the heap, where their height is small, requiring less time.

To derive the total work done, we calculate the number of nodes at each height $h$, given by $n/2^{h+1}$, and the cost to heapify is $O(h)$. Thus, the total time complexity can be expressed as:

$$T(n) = \sum_{h=0}^{\log n} \left( \frac{n}{2^{h+1}}(h+1) \right) \times O(h)$$

This simplifies to:

$$T(n) = O(n \sum_{h=0}^{\log n} \frac{h}{2^h})$$

The inner summation converges to a constant:

$$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$$

Thus, we have:

$$T(n) = O(n \times 2) = O(n)$$

In conclusion, even in the worst-case scenario, the time complexity for building a max-heap is $O(n)$. The workload is distributed such that, while a few nodes at the top require more time, the vast majority at the bottom require very little, resulting in an overall linear time complexity.

**Best-Case Time Complexity:**

- **Best Case:** $O(n)$, since the best case involves performing a constant number of swaps at each level of the heap tree.
- **Justification:** The overall time complexity for building a max-heap is $O(n)$ in both best and worst cases. This result can be shown through a more detailed analysis of the number of swaps per level in the heap, where the top-level nodes require fewer swaps than the leaf nodes.

## 4 Problem 4: Frying Hamburgers

**Original Algorithm (Recursive Process):**

- The algorithm works as follows:
  1. If $n \leq 2$: Fry the one or two hamburgers on the grill, taking 2 minutes (1 minute per side).
  2. If $n > 2$: Fry two hamburgers at a time (taking 2 minutes for both sides), and then apply the same process to the remaining $n - 2$ hamburgers.

**Recurrence Relation:**

- The recurrence can be described as:

$$T(n) = \begin{cases} 2 & \text{if } n \leq 2 \\ 2 + T(n-2) & \text{if } n > 2 \end{cases}$$

  where $T(n)$ represents the total time to fry $n$ hamburgers.

**Solving the Recurrence:**

- This recurrence can be solved iteratively:
  - For $n = 2$: $T(2) = 2$ minutes.
  - For $n = 3$: $T(3) = 2 + T(1) = 4$ minutes.
  - For $n = 4$: $T(4) = 2 + T(2) = 4$ minutes.
  - For $n = 5$: $T(5) = 2 + T(3) = 6$ minutes.
  - **General pattern:** $T(n) = 2 \times \lceil n/2 \rceil$.

Thus, the total time to fry $n$ hamburgers is proportional to the number of hamburgers, and the time complexity of this algorithm is $O(n)$.

**Improvement:**

- The original algorithm can be improved by recognizing that 3 hamburgers can be fried in 3 minutes instead of 4. The key observation is that after frying the first side of two hamburgers in 1 minute, a third hamburger can be added for the next minute, reducing the overall frying time.

**Initial Conditions:**

- For $n = 3$, instead of 4 minutes, we can fry all 3 hamburgers in just 3 minutes (1 minute for each side of the first two hamburgers and 1 minute for the third hamburger's second side).

**New Recurrence Relation:**

$$T(n) = \begin{cases} 3 & \text{if } n = 3 \\ 2 & \text{if } n \leq 2 \\ 2 + T(n-2) & \text{if } n > 3 \end{cases}$$

- If $n = 3$, fry all three hamburgers in 3 minutes.
- For $n > 3$, still fry two hamburgers at a time in 2 minutes and apply the same procedure to the remaining $n - 2$ hamburgers.

**Solution to the New Recurrence:**

- The recurrence follows the same pattern as the original one but starts with 3 hamburgers taking 3 minutes:
  - For $n = 4$: $T(4) = 2 + T(2) = 4$ minutes.
  - For $n = 5$: $T(5) = 2 + T(3) = 5$ minutes.
  - For $n = 6$: $T(6) = 2 + T(4) = 6$ minutes.

Thus, the overall time complexity remains $O(n)$, but the constant factor is slightly reduced due to the more efficient frying of 3 hamburgers in 3 minutes instead of 4.

**New Time Complexity:**

- The total time to fry $n$ hamburgers is still $O(n)$, but the improvement reduces the total frying time slightly because 3 hamburgers can now be fried in 3 minutes rather than 4.
- This reduces the constant factor but does not change the asymptotic time complexity.

## 5 Problem 5: Inversions and Average-Case Complexity of Insertion Sort

**Definition of an Inversion:** Let $A$ be an array of $n$ distinct elements. If $i < j$ and $A[i] > A[j]$, then the pair $(i, j)$ is called an **inversion** of $A$.

**Connection Between Inversions and Insertion Sort:** Insertion Sort works by repeatedly inserting an element into its correct position by shifting larger elements to the right. The number of times an element is shifted corresponds to the number of **inversions** in the array.

**Worst Case Analysis:** - If the array is sorted in **descending order**, then every element must be moved past all previous elements. - The total number of inversions in this case is:

$$T(n) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$$

- This gives the worst-case time complexity of $O(n^2)$.

**Best Case Analysis:** - If the array is already sorted in **ascending order**, no elements need to be moved. - The number of inversions is 0, leading to **O(n) complexity** since each element is inserted in constant time.

**Average Case Analysis:** To determine the average case complexity, we analyze the expected number of inversions in a randomly ordered array.

- The total number of possible pairs $(i, j)$ in an array of $n$ elements is:

$$\binom{n}{2} = \frac{n(n-1)}{2}$$

- In a randomly ordered array, each pair $(i, j)$ has an equal probability of being an inversion or being in order. - On average, **half** of these pairs are inversions, so the expected number of inversions is:

$$\frac{1}{2} \times \frac{n(n-1)}{2} = \frac{n(n-1)}{4}$$

- Since each inversion corresponds to a swap operation in Insertion Sort, the average number of swaps is proportional to $n^2$.

**Conclusion:** Since the number of swaps in insertion sort is proportional to the number of inversions, and the expected number of inversions is $\Theta(n^2)$, we conclude that the **average-case time complexity of Insertion Sort is $\Theta(n^2)$**.

**Reference:** - Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). The MIT Press.