Problem Set 0 - Solutions

CSC263 - Data Structures and Analysis

January 15, 2026

Problem 1: Runtime Analysis of 0-1 Sorting Algorithm [19 points]

Algorithm Description

The provided pseudocode implements a sorting algorithm for lists containing only 0s and 1s:

$n \leftarrow$ length of L
while true **do**
    for i from 1 to n **do**
        if $L[i] = 1$ **then**
            break
    for j from n to 1 **do**
        if $L[j] = 0$ **then**
            break
    if $j > i$ **then**
        swap $L[i], L[j]$
    else
        break

Part (a): High-Level Description [2 points]

Solution:

The algorithm repeatedly finds the leftmost 1 and the rightmost 0, swapping them if they are out of order. It terminates when all 0s are before all 1s (i.e., when the leftmost 1 is to the right of the rightmost 0).

Part (b): Runtime Analysis [17 points]

For a list of length $2n$ containing exactly n zeros and n ones:

	O	Ω	Θ
best-case	$O(n)$	$\Omega(n)$	$\Theta(n)$
worst-case	$O(n^2)$	$\Omega(n^2)$	$\Theta(n^2)$
average-case	$O(n^2)$	$\Omega(n^2)$	$\Theta(n^2)$

Justification:

Best-Case: $\Theta(n)$ The best case occurs when the list is already sorted (all 0s before all 1s). For example: $[0, 0, \ldots, 0, 1, 1, \ldots, 1]$.

In this case:

- The first for-loop scans through all n zeros before finding the first 1 at position $n+1$, taking $\Theta(n)$ time
- The second for-loop scans from position $2n$ down to $n+1$ (all 1s) before finding the last 0 at position n, taking $\Theta(n)$ time
- Since $j = n < i = n + 1$, the condition $j > i$ fails, and the algorithm terminates
- Total: $\Theta(n)$ operations in a single pass

Worst-Case: $\Theta(n^2)$ The worst case occurs when all 1s precede all 0s, requiring the maximum number of swaps. For example: $[1, 1, \ldots, 1, 0, 0, \ldots, 0]$.

Analysis:

- The algorithm performs exactly n swaps (each swap moves one 1 to its correct position in the right half and one 0 to its correct position in the left half)
- In iteration k (for $k = 1, 2, \ldots, n$), the array already has $k-1$ zeros at the beginning and $k-1$ ones at the end (from previous swaps)
- The first for-loop must scan past the $k-1$ zeros to find the first 1 at position k. Cost: $\Theta(k)$
- The second for-loop must scan past the $k-1$ ones at the end to find the first 0 at position $2n-k+1$. Cost: $\Theta(k)$
- Total cost per iteration: $\Theta(k)$

The total runtime is the sum of these increasing costs:

$$T(n) = \sum_{k=1}^{n} \Theta(k) = \Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2)$$

Concrete example: For $[1, 1, 1, 0, 0, 0]$ (where $n = 3$):

- Iteration 1: Scan to position 1 (1 step) + scan from position 6 (1 step, breaks immediately at 0), swap $L[1]$ and $L[6] \to [0, 1, 1, 0, 0, 1]$
- Iteration 2: Scan to position 2 (2 steps) + scan from position 5 (2 steps, scans past one 1), swap $L[2]$ and $L[5] \to [0, 0, 1, 0, 1, 1]$
- Iteration 3: Scan to position 3 (3 steps) + scan from position 4 (3 steps, scans past two 1s), swap $L[3]$ and $L[4] \to [0, 0, 0, 1, 1, 1]$
- Iteration 4: Scan to position 4 (4 steps) + scan from position 3 (4 steps), check fails, terminate
- Total: $(1+1) + (2+2) + (3+3) + (4+4) = 20 = \Theta(n^2)$ operations

Average-Case: $\Theta(n^2)$ For the average case over all possible arrangements of n zeros and n ones:

In a random permutation of n zeros and n ones, we expect approximately $\Theta(n)$ elements to be out of place, requiring $\Theta(n)$ swaps.

Crucially, the cost of finding the next elements to swap **increases** as the algorithm progresses:

- In iteration k, the first $k-1$ zeros and the last $k-1$ ones are already in their correct positions
- The pointers must scan past these sorted sections to find the next targets
- Therefore, the k-th iteration performs $\Theta(k)$ comparisons

The total work is the sum of these increasing costs:

$$T(n) = \sum_{k=1}^{n} \Theta(k) = \Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2)$$

This demonstrates that the runtime is quadratic not because every iteration costs $\Theta(n)$, but because the cost per iteration grows linearly from $\Theta(1)$ to $\Theta(n)$, and these costs sum to $\Theta(n^2)$.

Bonus Part: Improved Implementation [1 point]

A better algorithm uses the two-pointer technique:

$i \leftarrow 1$
$j \leftarrow n$
while $i < j$ **do**
    while $i < j$ and $L[i] = 0$ **do**
        $i \leftarrow i + 1$
    while $i < j$ and $L[j] = 1$ **do**
        $j \leftarrow j - 1$
    if $i < j$ **then**
        swap $L[i], L[j]$
        $i \leftarrow i + 1$
        $j \leftarrow j - 1$

Runtime: $\Theta(n)$ in all cases (best, worst, and average).

Each element is examined at most once because the pointers only move inward and never backtrack. This gives $\Theta(n)$ worst-case and average-case runtime, which is optimal since we must examine each element at least once.

Problem 2: String Concatenation Runtime [10 points]

Given Code:

```
def list_of_numbers(num):
    n = int(num)
    out = ""
    for i in range(n):
        out += str(i+1) + ","
    return out
```

Solution:

Runtime: $\Theta(n^2)$

Explanation: In Python, strings are *immutable*. This means that the concatenation operation `out += str(i+1) + ","` does not modify the existing string. Instead, it:

1. Creates a new string object
2. Copies all characters from the old `out` string
3. Appends the new characters
4. Reassigns `out` to point to this new string

Detailed Analysis: At iteration i (where $i = 0, 1, \ldots, n-1$):

- The current length of `out` is approximately proportional to i (specifically, it's roughly $2i$ characters, accounting for numbers and commas)
- The concatenation operation takes $\Theta(i)$ time to copy the existing string
- Adding the new number takes $\Theta(\log i)$ time (for converting the number to a string)
- Total time for iteration i: $\Theta(i)$

Summing over all iterations:

$$T(n) = \sum_{i=1}^{n} \Theta(i) = \Theta\left(\sum_{i=1}^{n} i\right) = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2)$$

Why This Is Tricky: At first glance, the code appears to be $\Theta(n)$ since there's only one loop. However, the hidden cost of string concatenation in Python makes this quadratic. Each `+=` operation on strings is $O(\mathit{length})$, not $O(1)$.

Efficient Alternative: To achieve $\Theta(n)$ runtime, use a list to accumulate strings and join them once:

```
def list_of_numbers(num):
    n = int(num)
    parts = []
    for i in range(n):
        parts.append(str(i+1))
    return ",".join(parts)
```

Or more concisely:

```python
def list_of_numbers(num):
    n = int(num)
    return ",".join(str(i+1) for i in range(n))
```

Both alternatives run in $\Theta(n)$ time because:

- Appending to a list is amortized $O(1)$
- The `join` operation concatenates all strings in a single pass, taking $\Theta(total\_length)$ time, which is $\Theta(n)$ for this problem