# COSC 3P03 Assignment 2

Solutions

**Due Date:** February 13, 2026

**Total Marks:** 45

# 1 Non-recursive Tower of Hanoi

## 1.1 Analysis of Four Algorithms

**Algorithm 0 (Recursive):** The classic recursive Tower of Hanoi algorithm that moves $n$ disks from peg 0 to peg 2.

**Algorithm 1:** If $i$ is even, swap pegs and the puzzle is solved. Make the only legal move that avoids peg $i \bmod 3$. If there is no legal move, then all disks are on peg $i \bmod 3$, and the puzzle is solved.

**Algorithm 2:** For the first move, move disk 1 to peg 1 if $n$ is even and to peg 2 if $n$ is odd. Then repeatedly make the only legal move that involves a different disk from the previous move. If no such move exists, the puzzle is solved.

**Algorithm 3:** Pretend that disks $n+1$, $n+2$, and $n+3$ are at the bottom of pegs 0, 1, and 2, respectively. Repeatedly make the only legal move that satisfies the following three constraints, until no such move is possible:

- Do not place an odd disk directly on top of another odd disk.
- Do not place an even disk directly on top of another even disk.
- Do not undo the previous move.

## 1.2 Question 1: Most Efficient Algorithm

All four algorithms are equally efficient. Each algorithm requires exactly $\mathbf{2^n - 1}$ moves to solve the Tower of Hanoi puzzle with $n$ disks. This is the theoretical minimum number of moves required, as proven by induction.

## 1.3 Question 2: Moves for $n = 1, 2, 3$ disks

### 1.3.1 For $n = 1$ disk:

**Algorithm 0 (Recursive):**

| Move | Action |
|---|---|
| 1 | Disk 1: Peg 0 $\rightarrow$ Peg 2 |

**Algorithm 1:**

| Move | Action |
|---|---|
| 1 | Disk 1: Peg 0 $\rightarrow$ Peg 2 |

**Algorithm 2:**

| Move | Action |
|---|---|
| 1 | Disk 1: Peg 0 $\rightarrow$ Peg 2 |

**Algorithm 3:**

| Move | Action |
|---|---|
| 1 | Disk 1: Peg 0 $\rightarrow$ Peg 2 |

### 1.3.2 For $n = 2$ disks:

**Algorithm 0 (Recursive):**

| Move | Action |
|---|---|
| 1 | Disk 1: Peg 0 $\rightarrow$ Peg 1 |
| 2 | Disk 2: Peg 0 $\rightarrow$ Peg 2 |
| 3 | Disk 1: Peg 1 $\rightarrow$ Peg 2 |

**Algorithm 1:**

| Move | Action |
|---|---|
| 1 | Disk 1: Peg 0 $\rightarrow$ Peg 1 |
| 2 | Disk 2: Peg 0 $\rightarrow$ Peg 2 |
| 3 | Disk 1: Peg 1 $\rightarrow$ Peg 2 |

**Algorithm 2:**

| Move | Action |
|---|---|
| 1 | Disk 1: Peg 0 $\rightarrow$ Peg 1 |
| 2 | Disk 2: Peg 0 $\rightarrow$ Peg 2 |
| 3 | Disk 1: Peg 1 $\rightarrow$ Peg 2 |

**Algorithm 3:**

| Move | Action |
|---|---|
| 1 | Disk 1: Peg 0 $\rightarrow$ Peg 1 |
| 2 | Disk 2: Peg 0 $\rightarrow$ Peg 2 |
| 3 | Disk 1: Peg 1 $\rightarrow$ Peg 2 |

### 1.3.3 For $n = 3$ disks:

**Algorithm 0 (Recursive):**

| Move | Action |
|---|---|
| 1 | Disk 1: Peg 0 $\rightarrow$ Peg 2 |
| 2 | Disk 2: Peg 0 $\rightarrow$ Peg 1 |
| 3 | Disk 1: Peg 2 $\rightarrow$ Peg 1 |
| 4 | Disk 3: Peg 0 $\rightarrow$ Peg 2 |
| 5 | Disk 1: Peg 1 $\rightarrow$ Peg 0 |
| 6 | Disk 2: Peg 1 $\rightarrow$ Peg 2 |
| 7 | Disk 1: Peg 0 $\rightarrow$ Peg 2 |

All other algorithms (1, 2, and 3) produce the same sequence of moves for $n = 3$.

## 1.4 Question 3: Comparison Table

| Algorithm | $n = 1$ | $n = 2$ | $n = 3$ |
|---|---|---|---|
| Algorithm 0 | 1 | 3 | 7 |
| Algorithm 1 | 1 | 3 | 7 |
| Algorithm 2 | 1 | 3 | 7 |
| Algorithm 3 | 1 | 3 | 7 |

All algorithms require $2^n - 1$ moves, which is the theoretical minimum.

## 2 Sorting - StoogeSort Analysis

### 2.1 Question 1: Would $m = \lfloor 2n/3 \rfloor$ work instead of $m = \lceil 2n/3 \rceil$?

**Answer:** No, STOOGESORT would NOT sort correctly with $m = \lfloor 2n/3 \rfloor$.

**Justification:**

The algorithm works by sorting the first 2/3, then the last 2/3, then the first 2/3 again. The key requirement is that these two overlapping regions must have sufficient overlap to ensure all elements end up in the correct positions.

With $m = \lceil 2n/3 \rceil$:

- First 2/3: positions 0 to $m - 1$
- Last 2/3: positions $n - m$ to $n - 1$
- Overlap: at least $\lceil n/3 \rceil$ positions

With $m = \lfloor 2n/3 \rfloor$:

- First 2/3: positions 0 to $m - 1$
- Last 2/3: positions $n - m$ to $n - 1$
- Overlap: can be as small as 1 position for certain values of $n$

**Counter-example:** Consider $n = 4$ with array $[4, 3, 2, 1]$:

- With $m = \lfloor 8/3 \rfloor = 2$:
  - First sort: indices 0–1 → [3,4,2,1]
  - Second sort: indices 2–3 → [3,4,1,2]
  - Third sort: indices 0–1 → [3,4,1,2]
- The array is NOT sorted!

The problem is that with $m = 2$, the first 2/3 is $[0, 1]$ and the last 2/3 is $[2, 3]$, which have NO overlap. The algorithm can only swap within each half independently, so it cannot sort the entire array.

For $n = 5$ with $m = \lfloor 10/3 \rfloor = 3$:

- First 2/3: indices 0–2
- Last 2/3: indices 2–4
- Overlap: only position 2 (insufficient for proper sorting)
- Result with $[5, 4, 3, 2, 1]$: produces $[1, 3, 4, 2, 5]$ (not sorted!)

The ceiling function ensures sufficient overlap for the algorithm to work correctly.

### 2.2 Question 2: Recurrence for Number of Comparisons

Let $T(n)$ be the number of comparisons executed by STOOGESORT on an array of size $n$.

**Base cases:**

$$
\begin{aligned}
T(1) &= 0 \quad \text{(no comparisons needed)} \\
T(2) &= 1 \quad \text{(one comparison: } A[0] \text{ vs } A[1])
\end{aligned}
$$

**Recursive case ($n > 2$):**

- $m = \lceil 2n/3 \rceil$
- The algorithm makes three recursive calls on subarrays of size at most $m$

**Recurrence:**

$$T(n) = \begin{cases} 0 & \text{if } n = 1 \\ 1 & \text{if } n = 2 \\ 3T(\lceil 2n/3 \rceil) & \text{if } n > 2 \end{cases} \tag{1}$$

Or ignoring the ceiling:

$$T(n) = 3T(2n/3) \quad \text{for } n > 2 \tag{2}$$

### 2.3 Question 3: Solve the Recurrence

Ignoring the ceiling, we have: $T(n) = 3T(2n/3)$

Using the Master Theorem or solving directly:

Let's use substitution. Assume $n = 2 \cdot (3/2)^k$ for some integer $k$.

Then:

$$\begin{aligned} T(2 \cdot (3/2)^k) &= 3T(2 \cdot (3/2)^{k-1}) \\ &= 3^2 T(2 \cdot (3/2)^{k-2}) \\ &= \cdots \\ &= 3^k T(2) \\ &= 3^k \end{aligned}$$

Since $(3/2)^k = n/2$, we have $k = \log_{3/2}(n/2)$, so:

$$\begin{aligned} 3^k &= 3^{\log_{3/2}(n/2)} \\ &= (n/2)^{\log_{3/2}(3)} \\ &= (n/2)^{\frac{\log 3}{\log(3/2)}} \end{aligned}$$

Since $\frac{\log 3}{\log(3/2)} = \frac{\log 3}{\log 3 - \log 2} \approx 2.71$

Therefore: $\mathbf{T(n) = \Theta(n^{\log_{3/2}(3)})} \approx \mathbf{\Theta(n^{2.71})}$

More precisely: $\mathbf{T(n) = \Theta(n^{\frac{\log 3}{\log(3/2)}})}$ where $\frac{\log 3}{\log(3/2)} \approx 2.7095$

**Proof by Induction:**

Base case: $T(2) = 1$ ✓

To verify this matches our formula, we need $c$ such that:

$$c \cdot 2^{\frac{\log 3}{\log(3/2)}} = 1 \tag{3}$$

This confirms our base case is consistent.

Inductive hypothesis: Assume $T(k) = c \cdot k^{\frac{\log 3}{\log(3/2)}}$ for all $k < n$.

Inductive step:

$$\begin{aligned} T(n) &= 3T(2n/3) \\ &= 3 \cdot c \cdot (2n/3)^{\frac{\log 3}{\log(3/2)}} \\ &= 3c \cdot (2/3)^{\frac{\log 3}{\log(3/2)}} \cdot n^{\frac{\log 3}{\log(3/2)}} \\ &= 3c \cdot (2/3)^{\log_{3/2}(3)} \cdot n^{\frac{\log 3}{\log(3/2)}} \end{aligned}$$

Note that $(3/2)^{\log_{3/2}(3)} = 3$, so $(2/3)^{\log_{3/2}(3)} = 1/3$.
Therefore:

$$T(n) = 3c \cdot (1/3) \cdot n^{\frac{\log 3}{\log(3/2)}} = c \cdot n^{\frac{\log 3}{\log(3/2)}} \tag{4}$$

This confirms our solution. ✓

### 2.4 Question 4: Prove the number of swaps is at most $n^3/3$

**Claim 1.** *The number of swaps executed by STOOGESORT is at most $n^3/3$.*

*Proof.* Let $S(n)$ be the maximum number of swaps for an array of size $n$ in the worst case.
**Base cases:**

$$S(1) = 0 \leq 1^3/3 = 0.333 \quad ✓$$
$$S(2) = 1 \leq 2^3/3 = 8/3 \approx 2.67 \quad ✓$$

**Recursive case ($n > 2$):**
Each of the three recursive calls operates on arrays of size at most $m = \lceil 2n/3 \rceil$.

$$S(n) \leq 3S(\lceil 2n/3 \rceil) \tag{5}$$

With $S(n) = 3S(2n/3)$, we get the same form as $T(n)$:

$$S(n) = \Theta(n^{\log_{3/2}(3)}) = \Theta(n^{2.71}) \tag{6}$$

where $\log_{3/2}(3) = \frac{\log 3}{\log(3/2)} \approx 2.71$.
**Showing $S(n) \leq n^3/3$:**
Since $S(n) = \Theta(n^{2.71})$, we need to verify that $n^{2.71} < n^3/3$ for all $n \geq 1$.
This is equivalent to showing: $3n^{2.71} < n^3$, which simplifies to $3 < n^{3-2.71} = n^{0.29}$.
Solving $n^{0.29} = 3$: $n = 3^{1/0.29} \approx 31$. Thus for $n \geq 31$, we have $n^{0.29} > 3$, ensuring $n^{2.71} < n^3/3$.
For small values of $n$ ($n < 31$), we verify the bound directly with the induction proof below.
**Proof by Strong Induction:**
We'll prove $S(n) \leq n^3/3$ by strong induction.
Base cases:

$$S(1) = 0 \leq 1^3/3 = 1/3 \quad ✓$$
$$S(2) = 1 \leq 2^3/3 = 8/3 \quad ✓$$

Hypothesis: Assume $S(k) \leq k^3/3$ for all $k < n$.
Step: For $n > 2$, let $m = \lceil 2n/3 \rceil$. Then:

$$\begin{aligned} S(n) &\leq 3S(m) \\ &\leq 3 \cdot m^3/3 \quad \text{(by hypothesis)} \\ &= m^3 \end{aligned}$$

We need to show that $m^3 \leq n^3/3$.
Since $m \leq 2n/3 + 1$, for large $n$ we have $m \approx 2n/3$, so:

$$m^3 \approx (2n/3)^3 = 8n^3/27 \approx 0.296n^3 \tag{7}$$

Since $8n^3/27 < n^3/3$ (because $24n^3 < 27n^3$), we have:

$$S(n) \leq m^3 \approx (2n/3)^3 = 8n^3/27 < n^3/3 \quad ✓ \tag{8}$$

Therefore, $S(n) \leq n^3/3$ for all $n \geq 1$. ✓ □

## 3 Bonus Question - QuickSelect

### 3.1 Algorithm Description

**QuickSelect** is a selection algorithm to find the $k$-th smallest element in an unordered list. It is related to the QuickSort sorting algorithm and was developed by Tony Hoare.

**Algorithm 1** QuickSelect

```
function QuickSelect($(A, p, r, k)$)
    Input: Array $A$, indices $p$ and $r$ $(p \leq r)$, and rank $k$ $(1 \leq k \leq r - p + 1)$
    Output: The $k$-th smallest element in $A[p \dots r]$
    if $p = r$ then
        return $A[p]$
    end if
    $q \leftarrow$ Partition$(A, p, r)$                ▷ Partition around pivot
    rank $\leftarrow q - p + 1$                        ▷ Rank of pivot in subarray
    if $k =$ rank then
        return $A[q]$                                  ▷ Pivot is the $k$-th smallest
    else if $k <$ rank then
        return QuickSelect$(A, p, q - 1, k)$           ▷ Search left
    else
        return QuickSelect$(A, q + 1, r, k -$ rank$)$  ▷ Search right
    end if
end function
```

**Algorithm 2** Partition

```
function Partition$(A, p, r)$
    $x \leftarrow A[r]$                ▷ Pivot element
    $i \leftarrow p - 1$
    for $j = p$ to $r - 1$ do
        if $A[j] \leq x$ then
            $i \leftarrow i + 1$
            swap $A[i]$ with $A[j]$
        end if
    end for
    swap $A[i + 1]$ with $A[r]$
    return $i + 1$
end function
```

### 3.2 Question 1: Average-Case Time Complexity

**Answer:** The average-case time complexity of QuickSelect is $\mathbf{\Theta(n)}$.

**Recurrence Relation (Average Case):**

Let $T(n)$ be the expected number of comparisons for an array of size $n$.

In the average case:

- The partition operation takes $\Theta(n)$ comparisons
- We only recurse on one side (unlike QuickSort which recurses on both sides)
- The key insight is that we recurse into whichever subarray contains the $k$-th element

- On average, assuming the pivot is equally likely to be any element, we recurse into a subarray of expected size $n/2$

$$T(n) = T(n/2) + \Theta(n), \quad T(1) = \Theta(1) \tag{9}$$

**Solving the Recurrence:**
Using the recurrence $T(n) = T(n/2) + cn$ where $c$ is a constant:

$$\begin{aligned}
T(n) &= T(n/2) + cn \\
&= T(n/4) + c(n/2) + cn \\
&= T(n/8) + c(n/4) + c(n/2) + cn \\
&= \cdots \\
&= T(1) + cn(1/2 + 1/4 + 1/8 + \cdots)
\end{aligned}$$

The geometric series $(1/2 + 1/4 + 1/8 + \cdots)$ converges to:

$$\text{Sum} = \frac{1/2}{1 - 1/2} = \frac{1/2}{1/2} = 1 \tag{10}$$

Therefore:

$$T(n) = \Theta(1) + cn = \Theta(n) \tag{11}$$

**Proof by Induction:**
We'll prove $T(n) \leq cn$ for some constant $c > 0$.
Base case: $T(1) \leq c \cdot 1$ for sufficiently large $c$. ✓
Inductive hypothesis: Assume $T(k) \leq ck$ for all $k < n$.
Inductive step: For $n > 1$,

$$\begin{aligned}
T(n) &= T(n/2) + an \quad \text{(where } a \text{ is the partition constant)} \\
&\leq c(n/2) + an \quad \text{(by hypothesis)} \\
&= n(c/2 + a) \\
&\leq cn \quad \text{(when } c \geq 2a\text{)}
\end{aligned}$$

Therefore, $\mathbf{T(n) = \Theta(n)}$ in the average case. ✓

## 3.3 Question 2: Worst-Case Time Complexity

**Answer:** The worst-case time complexity of QuickSelect is $\mathbf{\Theta(n^2)}$.

**Recurrence Relation (Worst Case):**

The worst case occurs when the pivot is always the smallest or largest element, resulting in maximally unbalanced partitions:

$$T(n) = T(n-1) + \Theta(n), \quad T(1) = \Theta(1) \tag{12}$$

**Solving the Recurrence:**

$$\begin{aligned}
T(n) &= T(n-1) + cn \\
&= T(n-2) + c(n-1) + cn \\
&= T(n-3) + c(n-2) + c(n-1) + cn \\
&= \cdots \\
&= T(1) + c(2 + 3 + \cdots + (n-1) + n) \\
&= \Theta(1) + c \cdot \frac{n(n+1)}{2} - c \\
&= \Theta(n^2)
\end{aligned}$$

**Example:** For array $[1, 2, 3, 4, 5]$ finding the 5th smallest (maximum):

- With always choosing the last element as pivot
- First partition: $n$ comparisons, recurse on $n-1$ elements
- Second partition: $n-1$ comparisons, recurse on $n-2$ elements
- Continue until 1 element remains
- Total: $n + (n-1) + (n-2) + \cdots + 2 + 1 = \frac{n(n+1)}{2} = \Theta(n^2)$

Therefore, $\mathbf{T(n) = \Theta(n^2)}$ in the worst case. ✓

### 3.4 Question 3: Comparison with Other Selection Algorithms

**QuickSelect vs. Sorting-based Selection:**

| **Aspect** | **QuickSelect** | **Sort + Index** |
|---|---|---|
| Average Time | $\Theta(n)$ | $\Theta(n \log n)$ |
| Worst Time | $\Theta(n^2)$ | $\Theta(n \log n)$ or $\Theta(n^2)$ |
| Space | $\Theta(\log n)$ | $\Theta(1)$ to $\Theta(n)$ |
| In-place | Yes | Depends on sort |
| Modifies array | Yes | Yes |

QuickSelect is **faster on average** than sorting when you only need one element.
**QuickSelect vs. Median-of-Medians:**
The Median-of-Medians algorithm guarantees $\Theta(n)$ worst-case time but with a larger constant factor:

| **Algorithm** | **Average Case** | **Worst Case** | **Practical** |
|---|---|---|---|
| QuickSelect | $\Theta(n)$ | $\Theta(n^2)$ | Fast (low constant) |
| Median-of-Medians | $\Theta(n)$ | $\Theta(n)$ | Slower (high constant) |
| Randomized QS | $\Theta(n)$ expected | $\Theta(n^2)$ worst | Fast (expected) |

**Key Insight:** While QuickSelect has a quadratic worst case, it performs excellently in practice because:

1. The average case is linear
2. The constant factors are small
3. Randomization can make worst case extremely unlikely

### 3.5 Question 4: Optimizations and Variants

**Randomized QuickSelect:**
Choose the pivot randomly instead of always choosing the last element:

**Algorithm 3** Randomized Partition

```
function RandomizedPartition($A, p, r$)
    $i \leftarrow$ Random($p, r$)
    swap $A[i]$ with $A[r]$
    return Partition($A, p, r$)
end function
```

This gives **expected** $\Theta(n)$ time complexity and makes the worst case extremely unlikely.
**Iterative QuickSelect:**
To reduce space complexity from $\Theta(\log n)$ to $\Theta(1)$, use an iterative version:

**Algorithm 4** Iterative QuickSelect

```
function IterativeQuickSelect($A, p, r, k$)
    while $p < r$ do
        $q \leftarrow$ Partition($A, p, r$)
        rank $\leftarrow q - p + 1$
        if $k =$ rank then
            return $A[q]$
        else if $k <$ rank then
            $r \leftarrow q - 1$
        else
            $k \leftarrow k -$ rank
            $p \leftarrow q + 1$
        end if
    end while
    return $A[p]$
end function
```

### 3.6 Question 5: Practical Applications

QuickSelect is used in:

1. **Finding medians** for statistical analysis
2. **Computing percentiles** in data analysis
3. **Selecting top-$k$ elements** in recommendation systems
4. **Pivot selection** in QuickSort optimizations
5. **Database query optimization** for ORDER BY . . . LIMIT queries

**Example: Finding the Median**
To find the median of an array $A[0 \dots n-1]$ of size $n$ (0-based indexing):

- If $n$ is odd: median is at position $n/2$ (integer division)
  - Example: $n = 5$, median at index 2 $\rightarrow$ QuickSelect$(A, 0, 4, 3)$ [3rd smallest]
- If $n$ is even: You may want both middle elements or just one

  - Lower median at position $(n/2 - 1) \rightarrow \textsc{QuickSelect}(A, 0, n-1, n/2)$
  - Upper median at position $n/2 \rightarrow \textsc{QuickSelect}(A, 0, n-1, n/2+1)$
  - Example: $n = 6$, lower median at index 2, upper at index 3

This is much faster than sorting the entire array: $\Theta(n)$ vs. $\Theta(n \log n)$.

### 3.7 Summary

QuickSelect is an efficient selection algorithm that:

- Achieves $\Theta(n)$ average-case time complexity
- Has $\Theta(n^2)$ worst-case time complexity
- Outperforms sorting-based approaches for single element selection
- Can be optimized with randomization to achieve expected linear time
- Is widely used in practice due to its simplicity and efficiency

## 4 Searching Lower and Upper Bounds

### 4.1 Question 1: Yes/No Answers - Worst Case

If Sam answers “Yes/No” to questions “Is the number $x$?”:

**Answer:** You will need at most $n - 1$ **questions** in the worst case.

**Explanation:**

Since Sam can change his answer as long as he doesn’t contradict previous answers, the worst-case scenario is when Sam always says “No” until you’ve asked about all but one number.

With “Yes/No” questions:

- Each “No” answer eliminates only one number from consideration
- Sam can keep changing his mind to whichever number you haven’t asked about yet
- After asking about $n - 1$ numbers and getting “No” each time, only one number remains
- This last number must be the answer (no need to ask about it)

Therefore, $n - 1$ **questions** are sufficient and necessary in the worst case.

### 4.2 Question 2: Can We Improve with Different Sequence?

**Answer:** No, we cannot improve the number of questions with “Yes/No” answers.

**Explanation:**

With “Yes/No” questions of the form “Is the number $x$?”, each question can only eliminate one possibility (when the answer is “No”). Since Sam is adversarial and can change his answer as long as it doesn’t contradict previous responses:

- The information-theoretic lower bound is $\log_2(n)$ for finding one number among $n$ possibilities
- However, with an adversarial Sam, we cannot achieve this because:
  - Each “No” answer only eliminates one specific number
  - Sam can adapt his strategy to maximize the number of questions
  - No matter what sequence we choose, Sam can always force us to ask about $n - 1$ numbers

Therefore, $n - 1$ **questions** is the lower bound for this scenario, regardless of the sequence chosen.

### 4.3 Question 3: Higher/Lower Answers

If Sam answers “higher/lower” to your inquiries:

**Answer:** You will need at most $\lceil \log_2(n) \rceil$ **questions**.

**Explanation:**

With “higher/lower” answers, we can use binary search:

- Each question of the form “Is the number $x$?” with “higher/lower” response
- Each answer eliminates approximately half of the remaining possibilities
- Even with adversarial Sam, he must be consistent with a contiguous range

The strategy:

1. Always ask about the middle element of the remaining range
2. "Higher" eliminates the lower half; "Lower" eliminates the upper half
3. After each question, the search space is halved

Number of questions needed:

- After 1 question: at most $n/2$ numbers remain
- After 2 questions: at most $n/4$ numbers remain
- After $k$ questions: at most $n/2^k$ numbers remain
- When $n/2^k \leq 1$, we've found the number

Therefore: $\mathbf{k} = \lceil \log_{\mathbf{2}}(\mathbf{n}) \rceil$ questions
For $n = 1,000,000$:

$$\lceil \log_2(1,000,000) \rceil = \lceil 19.93 \rceil = \mathbf{20} \text{ questions} \tag{13}$$

This is significantly better than the $n-1 = 999,999$ questions needed with "Yes/No" answers!