DSA Assignment 2

Do Huynh - 220556908

February 2024

1 Algorithm Analysis

1.1 Program A

- Post-condition: All 'C' elements precede 'S' elements in the list
- Loop invariant: $lnv(p1, p2)$: $\forall i \in [0, p1] : lst[i] \equiv$ 'C'$\rightarrow i < p2$. This expression states that for all indices i up to $p1$ where the element is 'C',i must be less than $p2$, ensuring that all 'C' elements encountered so far are placed before all 'S' elements.
- Post-condition Proof:
  - **Initialization:** Before the loops starts, $p1 = 0$ and $p2 = 0$. The list is initially unaltered. Hence, the post-condition holds
  - **Maintenance:** During the loop, the algorithm swaps elements whenever it encounters a 'C' at the index $p1$. As a result of the swap, the 'C' element is shifted towards the beginning of the list, increasing $p2$ by 1 and $p1$ by 1. After the swap, the invariant still holds since the index of any 'C' element is moved to the index position $p2$ and $p2$ is increased by 1, making it greater than or equal to the index of any 'C' element. If a 'S' is found, then $p1$ is increased by 1 and the invariant still holds since $p2$ is not affected by this case and is still greater than or equal to any index position of C.
  - **Termination:** The loop terminates when $p1$ is equal to the length of the list. At this point, all elements have been processed and the pointer is at the end of the list. Since the loop invariant has been maintained throughout the loop, all occurrences of 'C' are placed before all occurrences of 'S' in the list. Therefore, the post-condition holds after the termination of the algorithm, proving that all occurrences of 'C' precede all occurrences of 'S' in the list.

1.2 Program B

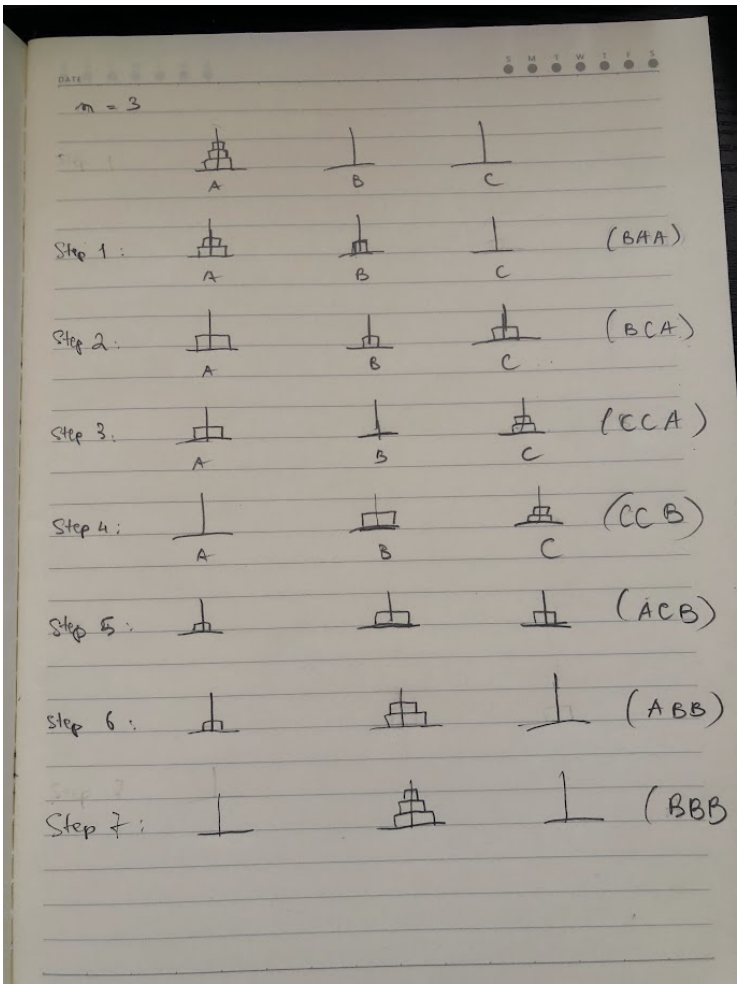
- Post-condition: return $\frac{1}{n}$

- Loop invariant: $r = \frac{1}{2} \times \frac{2}{3} \times ... \times \frac{k-1}{k}$ where k is the value of the loop counter
- Post-condition Proof:
  - **Initialization:** when $n = 1$, the loop doesn't execute, and the function returns 1. This is because the loop condition $k < n$ fails, and the function directly returns r, which is initialized to 1. Therefore, the post-condition holds when $n = 1$
  - **Maintenance:** Assume that the post-condition holds for some arbitrary m where $n = m$, meaning the function returns $\frac{1}{m}$. Now, consider $n = m + 1$. During the execution of the loop, r is updated according to the lop body for each iteration. At the beginning of each iteration, r contains the product of the fraction up to $\frac{k-1}{k}$ where k is the index of the next fraction to be processed. After the loop terminates ($k = n$), r accumulates the product of the fractions up to $\frac{n-1}{n}$ as followed $r = \frac{1 \times 2 \times ... \times n-1}{2 \times 3 \times ... \times n}$ which - after cancelling out the numbers - becomes $\frac{1}{n}$. Therefore, the function correctly returns $\frac{1}{n}$ as required by the post-condition for $n = m + 1$
  - **Termination:** The loop terminates when $k = n$. At this point, r contains the product of the fractions up to

2 Programming Question

2.1 3-peg TOH Analysis

First of all, I label the 3 pegs as A, B, and C with the destination peg as B. I start with the number of disks $n = 3$ and the experiment is conducted below

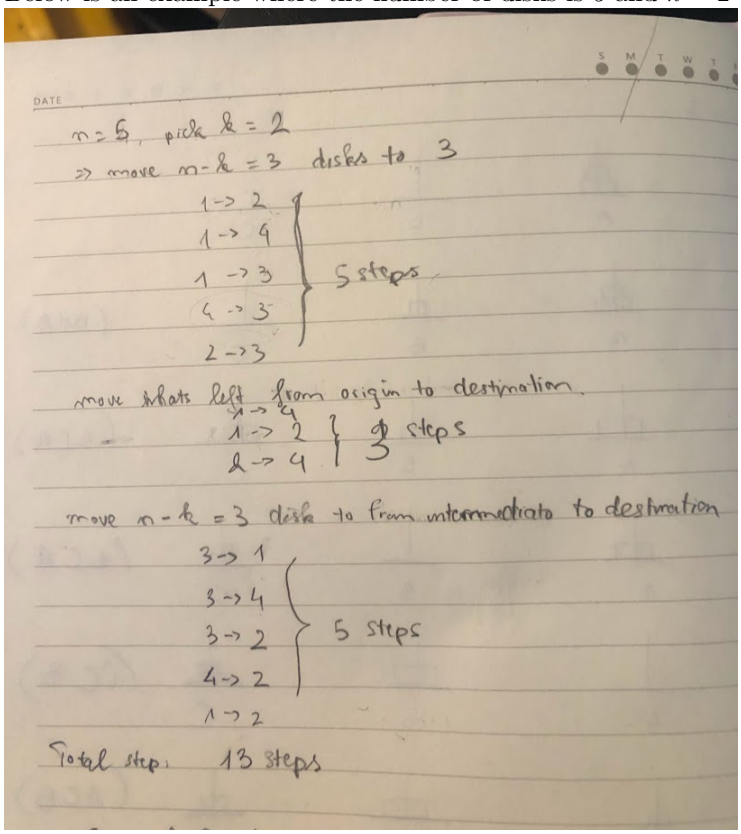


The first thing I notice about the steps it takes to solve this puzzle is $2^n - 1$ where n is the number of disks. The second thing is that the smallest disk will go from A to B to C and then go back to A and continue the pattern. It also stays in the same position for 2 moves after it moves to the next peg. The general pattern to solve is to move $n - 1$ disks from A to C, move a disk from A to B, and move $n - 1$ disks from C to B.
In my code, I label A as "from", B as "to" (this is the destination peg), and C as "auxiliary"

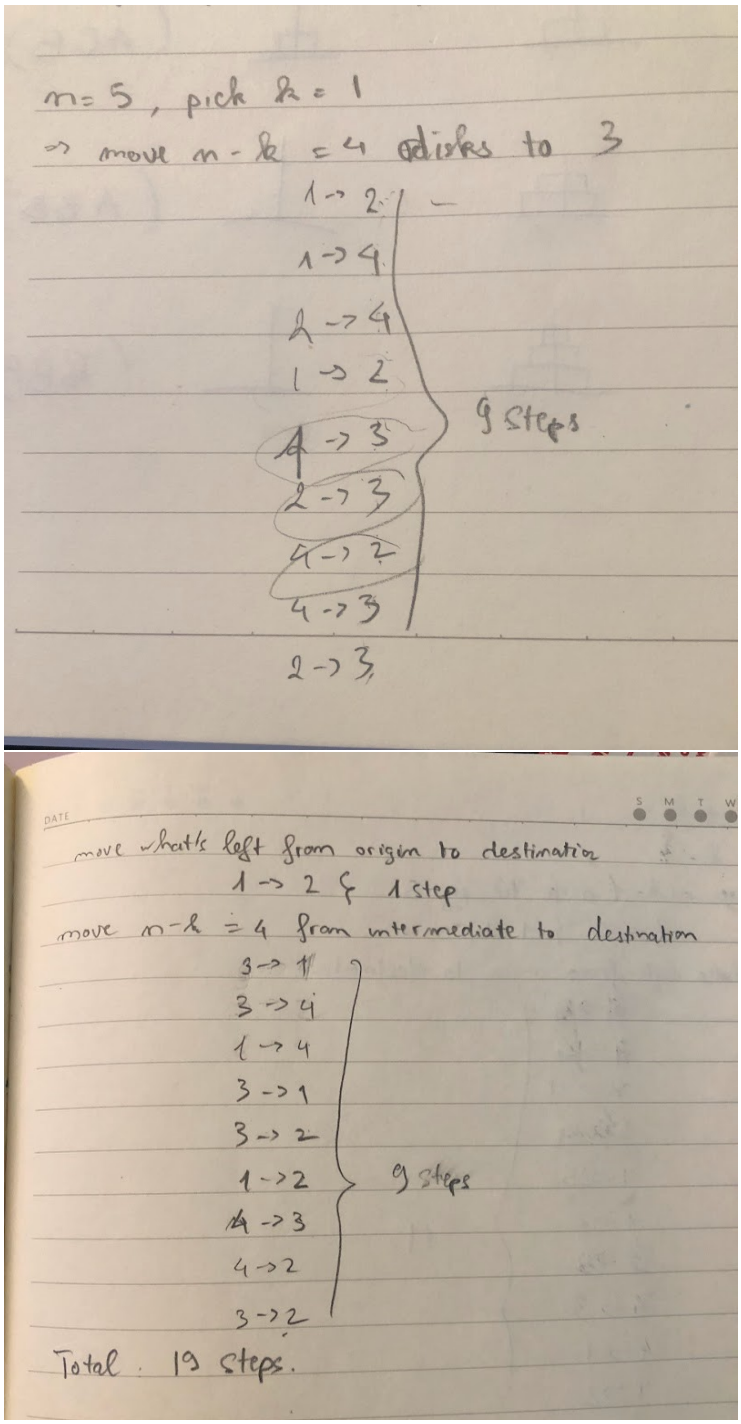
2.2 4-peg TOH Analysis

Based on my experiment to figure out an optimal k value, k should be 2. The reason is that when you move $n - k$ disk(s) to an intermediate peg, you are left with exactly 2 disks left in the origin peg and then it only takes 3 moves to bring

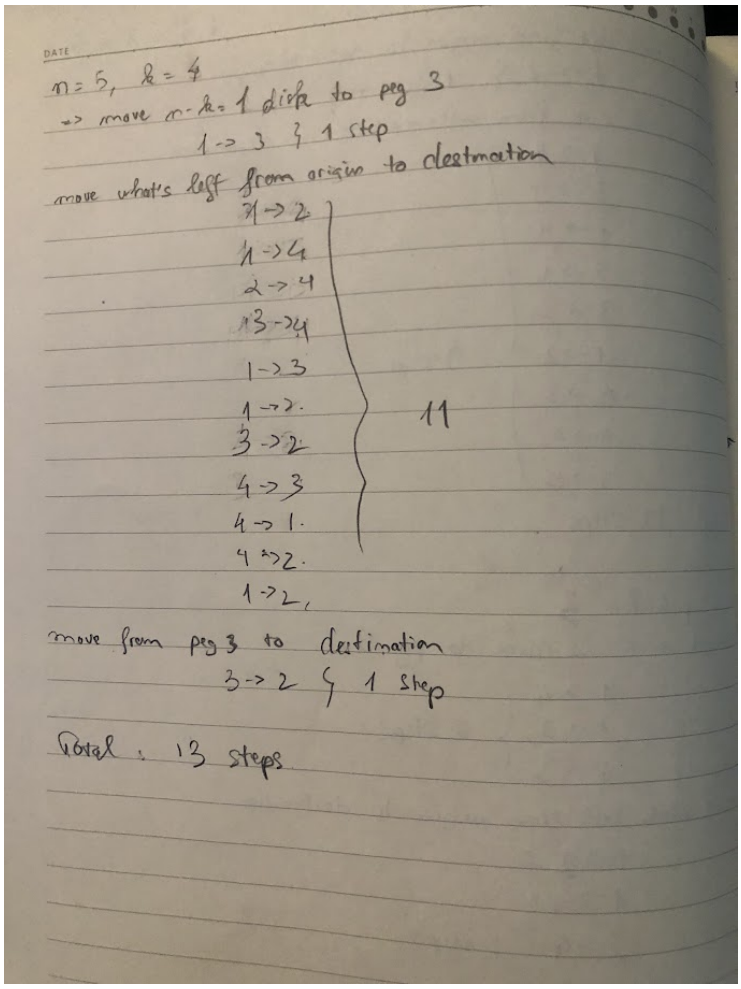
those 2 disks to the destination peg. Now we can bring the $n - k$ disk(s) from the intermediate peg to the destination peg and it requires the same amount of moves when you move them here.
Below is an example where the number of disks is 5 and $k = 2$



An example with $k = 2$. Even though the the middle step only requires 1 move, you end up with 19 steps which is not the least possible number of moves.



Another example with the same number of disks and $k = 4$ this time



Even though they take the same amount of moves as $k = 2$ does (13 moves), the case where $k = 2$ is more optimal since the middle step only takes 3 moves to complete whereas when $k = 4$, it takes 11 moves to complete.
I came up with the formula to calculate the number of moves for $k = 2$. Let f_4 be the function to calculate the number of moves using 4 pegs, and $f_3(n) = 2^n - 1$ be the function to calculate the number of moves using 3 pegs.
$f_4(n) = f_4(n-k) + f_3(k) + f_4(n-k)$ with $n \geq 2$
$f_4(n) = 2f_4(n-k) + f_3(k)$
$f_4(n) = 2f_4(n-2) + f_3(2)$
$f_4(n) = 2f_4(n-2) + 2^2 - 1$
$f_4(n) = 2f_4(n-2) + 3$

2.3 Comparison between 3-peg TOH and 4-peg TOH

With a small number of disks:
$f_4(3) = 2f_4(1) + 3 = 2(1) + 3 = 5$ moves
$f_3(3) = 2^3 - 1 = 7$ moves

With a large amount of disks, the number of moves it takes when using 4 pegs is drastically decreasing. For example, with 5 disks being moved, it takes $2^5 - 1 = 31$ moves using 3 pegs while it only takes 13 moves as shown in the example.
$f_4(5) = 2f_4(3) + 3 = 2(5) + 3 = 13$ moves
$f_3(5) = 2^5 - 1 = 31$ moves