RMIT University

Problem Solving and Coding Activity

COSC2658 Data Structure & Algorithm

Lecturer: Dr. Minh Dinh
Author: Huynh Nguyen Nguyen
ID: s3694703

1-10-2021

Contents

1. Question 1: Jagged List 4
   1.1. Problem Analysis 4
   1.2. Algorithm Description 5
   1.3. Software Implementation 8
   1.4. Formal Analysis 10
   1.5. Sample run 14
2. Question 2: Non-recursion conversion 15
   2.1. Conversion 15
   2.2. Sample Implementation and Testing 15
3. Question 3: Tower of Hanoi 16
   3.1. Algorithm Description 16
   3.2. Software Implementation 18
   3.3. Sample run 20
4. Question 4: Consecutive List 22
   4.1. Algorithm Description 22
   4.2. Software Implementation 22
   4.3. Sample run 24
5. References 25

List of Figures

Figure 1. Code for uptrend and downtrend function 8
Figure 2. Code for testing jagged 9
Figure 3. Code for running the simulation 9
Figure 4. Code of the main program to convert list into jagged list 10
Figure 5. Special test cases 14
Figure 6. Jagged program outputs 14
Figure 7. R(x) procedure 15
Figure 8. Arbitrary functions 16
Figure 9. R(x) function 16
Figure 10. R(x) function in non-recursive version 16
Figure 11. Result from recursive version 16

Figure 12. Result from non-recursive version 16
Figure 13. Main operation to solve TOH puzzle 17
Figure 14. Sub dividing step 1 into smaller operations using same method 18
Figure 15. Function to solve the Tower of Hanoi puzzle with text description 19
Figure 16. Function to solve the Tower of Hanoi puzzle with illustration 20
Figure 17. The initial Tower of Hanoi 20
Figure 18. Solution TOH with text description 20
Figure 19. Program to check consecutive list 22
Figure 20. Function to validate user input 23
Figure 21. Main program of question 4 23

List of Tables

Table 1. Jagged list example 1 4
Table 2. Jagged list example 2 4
Table 3. Main operation of simulating run 5
Table 4. Main operation of uptrend function 6
Table 5. Main operation of downtrend function 6
Table 6. Swap cases for jagged list 7
Table 7. Main operation of testing jagged function 7
Table 8. Operation in the main function of converting into jagged list 8
Table 9. Formal analysis of down_trend function 10
Table 10. Formal analysis of up_trend function 10
Table 11. Formal analysis of test_jagged function 11
Table 12. Formal analysis of simulate_run function 11
Table 13. Formal analysis of jagged_list function 12
Table 14. Input parameters for the tower of hanoi function 19
Table 15. Solution TOH with illustration 21
Table 16. Consecutive program output in different cases 24

1. Question 1: Jagged List

1.1. Problem Analysis

To begin with, the jagged list is defined as a list of numbers in which the values increase and decrease sequentially. For instance, the array list [2, 3, 1, 4, 0] is a jagged list because the value increases as it goes from the first element to the second, and goes down when going from the second element to the third element.

From an array of unique elements, it is possible to produce more than one jagged list; each will require a different number of changes in element position called ***distance***. As an example, given the array [4, 3, 7, 8, 6, 2, 1], it can be converted into various jagged list which two of them could be:

Table 1. Jagged list example 1

NORMAL LIST	JAGGED LIST	CHANGES	DISTANCE
[4, 3, 7, 8, 6, 2, 1]	[4, 7, 3, 8, 2, 6, 1]	Move element 3 from index 1 to 2 Move element 7 from index 2 to 1 Move element 6 from index 4 to 5 Move element 2 from index 5 to 4	4
	[4, 3, 7, 6, 8, 1, 2]	Move element 8 from index 3 to 4 Move element 6 from index 4 to 3 Move element 2 from index 5 to 6 Move element 1 from index 6 to 5	4

Hence, for the array [4, 3, 7, 8, 6, 2, 1], there are two possible jagged solutions that require the minimum changes on the initial array. An another example is presented as follows

Table 2. Jagged list example 2

NORMAL LIST	JAGGED LIST	DISTANCE
[6, 5, 2, 100, 1, 200]	[6, 5, 100, 2, 200, 1]	4
	[5, 6, 2, 100, 1, 200]	2

For a second example, the results are different in efficiency. As an interpretation, it can be seen that the normal list has a sub jagged list inside - [5, 2, 100, 1, 200] which follows the trend of “down up down up”. Since the task is to convert an input array into a jagged list with as few changes as possible, it is opted to prioritize keeping the current changing trend of the array list. As a result, the [6, 5, 2, 100, 1, 200] should start with the uptrend to made up the overall trend of “up down up down up”. This example is a hint to follow to current trend of the list in order to obtain and maximize the distance in input array.

Hence, for further analysis, the jagged list will be separated into two types based on the tendency of elements to change in the list:

- Type 1: Downtrend jagged list is those starting with the down trend

- Type 2: Uptrend jagged list is those starting with the up trend

1.2. Algorithm Description

This algorithm consists of two parts: (1) Tracing the input array to figure out the best swapping approach that costs minimum changes on the initial array and (2) Traversing the input array and performing the swap operation.

Part 1: Determining the best swapping approach

As being discussed, given a normal list, there will be two practical solutions: jagged list starting with downtrend and jagged list starting with uptrend. Depending on the input, these two solutions can result in same or different efficiency. Moreover, since the key task is to minimize the number of changes in elements position, the overall picture or the current jagged status of the array must be found out before any swapping can be operated. Hence, this section concentrates on performing a simulation for both methods such that the conclusion upon which method should be executed can be made.

The main operations for the simulation run is presented as follows:

Table 3. Main operation of simulating run

FUNCTION SIMULATE_RUN
Step 1: If the array size is 0 or 1, there is no swap needed. Return 0, 0 **Step 2**: Initialize the two distance variables: • "down_distance" is to save the distance in converting the input into the jagged list starting with downtrend • "up_distance" is to save the distance in converting the input into the jagged list starting with uptrend **Step 3**: For each conversion method, initialize the n1 pointer as the first element and n2 pointer as the second element: • "down_n1" and "down_n2" are a pair of pointers for down jagged list • "up_n1" and "up_n2" are a pair of pointers for up jagged list **Step 4**: Run the loop when "i" is from 0 to the array size - 1: 4.1. Calculate the next element from the current pair: 4.1.1. If the array length is greater than i + 2, the next element is the element at index "i +2". 4.1.2. Else, the next element is None 4.2. Test if the current pair of downtrend ("down_n1" and "down_n2") follows its desired trend 4.2.1. Calculate the desired trend by using "**down_trend**" function 4.2.2. The testing executed by the "**test_jagged**" function will return 4 data: • The first one shows whether any change is necessary • Three other data show the updated value of n1 and n2 4.2.3. If need change, increase the "down_distance" by 2. 4.2.4. Update the ("down_n1" and "down_n2") pair based on the "**test_jagged**" result 4.3. Test if the current pair of downtrend ("up_n1" and "up_n2") follows its desired trend

4.3.1. Calculate the desired trend by using "**up_trend**" function

4.3.2. The testing will be executed by the "**test_jagged**" function

4.3.3. If need change, increase the "up_distance" by 2.

4.3.4. Update the ("up_n1" and "up_n2") pair based on the "**test_jagged**" result

Step 5: return ("down_distance" and "up_distance")

Explanation: The program takes the array as its input without changing or swapping any elements during the traversal. For the simulation of each method, it is required with three extra variables: "n1", "n2" and "distance". As the function operation relies on the increasing/decreasing trend of two consecutive numbers within the input array, those two pointers – "n1" and "n2" – is to save two current examined positions and come up with the current trend. The "distance" variable is to store the number of changes needed for a specific method; thus, being the major factor to determine whether the method is efficient or not. Moreover, the main operation of "simulate_run" function lies on the iteration part where each pair of elements is audited inside the function "test_jagged" which will return True if any changes is essential for the array to follow the desired trend. Although the simulation is performed based on "test_jagged" method for both downtrend and uptrend method, the desired trend inputting into "test_jagged" method will be different for each method. In specific, when the jagged list starting with the uptrend, the element at odd index must be smaller than its even element on the right side. In reverse, when the jagged list starting with the downtrend, the element at odd index must be bigger than its even element on the right side. As a result, based on the current location of the element, we can figure out the essential trend for the conversion. The desired trend for those downtrend jagged list can be computed using the "up_trend" function and the desired trend for uptrend list is based on the "down_trend" function.

Table 4. Main operation of uptrend function

FUNCTION UP_TREND
Divide index by 2 1.1. If the index is divisible by 2, return True (standing for the desired trend is uptrend) 1.2. If not, return False (standing for the desired trend is downtrend)

Table 5. Main operation of downtrend function

FUNCTION DOWN_TREND
Divide index by 2 1.1. If the index is divisible by 2, return False (standing for the desired trend is downtrend) 1.2. If not, return True (standing for the desired trend is uptrend)

As mentioned, the swapping strategy is lied on the "test_jagged" function which takes four input parameters - three consecutive elements in the array and the desired trend. As a reason to take three consecutive elements instead of two, it is referred to the intending to minimize the changes in element

positions. Based on many test cases, it was found that, regardless of the current trend, there would be a way to change the trend of any three elements using just one swap. The example will be shown in following table.

Table 6. Swap cases for jagged list

SCENARIO	CASE	INPUT	OPERATION	OUTPUT
Convert any trend to down up	Up up → down up	[1,2,3]	Swap the first pair	[2,1,3]
	Down down → down up	[3,2,1]	Swap the second pair	[3,1,2]
	Up down → down up	[1,3,2]	Swap the first pair	[3,1,2]
		[2,3,1]	Swap the second pair	[2,1,3]
	Down up → down up	[3,1,2]	No swap needed	[3,1,2]
Convert any trend to up down	Up up → up down	[1,2,3]	Swap the second pair	[1,3,2]
	Down down → up down	[3,2,1]	Swap the first pair	[2,3,1]
	Down up → up down	[3,1,2]	Swap the first pair	[1,3,2]
		[2,1,3]	Swap the second pair	[2,3,1]
	Up down → up down	[1,3,2]	No swap need	[1,3,2]

As being derived from above example, except for the case where the input trend is matched with the desired one, the other cases enter the same swap pattern such that the position of the pair be swapped depends on the comparison between the first element and the third element. In specific, noting that the trend is saved in Boolean type, if the first element is smaller than the third one, the swap position will be the integer version of the desired trend. Taking the first case as an example – [1,2,3], when the desired trend is downtrend (False) and the first element is smaller than the third one (1 < 3), the swap position is the integer conversion of False which is index 0 – showing the first pair. In sum up, the main operation in the test_jagged function is:

Table 7. Main operation of testing jagged function

FUNCTION TEST_JAGGED
Step 1: Compare n1 and n2 to check if they follow up trend
Step 2: If n1 and n2 trend is the same as the desired trend, return no swap need
Step 3: Otherwise when the current trend is different than desired trend, it is opted to change element positions.
Step 4: If next_value is None, return swap needed and swap the first pair
Step 5: Otherwise, calculate the swap pair position based on the desired trend. If the first element is smaller than the third one, swap position is integer of (desired trend). Otherwise, swap_position is integer of (not desired trend). In specific,
5.1. If the desired trend is uptrend (True)
5.1.1. If n1 is less than next_value, return swap needed and swap the second pair- int(True) = 1

5.1.2. Otherwise, return swap needed and swap the first pair – int (False) = 0
5.2. If the desired trend is downtrend (False)
5.2.1. If n1 is less than next_value, return swap needed and swap the first pair – int (False) = 0
5.2.2. Otherwise, return swap needed and swap the second pair – int(True) = 1

Part 2: Performing swap operation onto the input array

After running the simulation for the swapping method, it is referred to the comparison between the distance needed for the down jagged list conversion and the distance for the uptrend one. Therefore, if the converting into an up jagged list costs fewer movements, the program will perform the actual operation to convert the input array into a jagged list starting from uptrend and vice versa.

Table 8. Operation in the main function of converting into jagged list

MAIN FUNCTION
Step 1: If the array size is less than 3, there is no swap needed. Return distance as 0
Step 2: Run the simulation function to get two distances
Step 3: Compare two distances
If the up distance is less than the down distance, go with the uptrend-first jagged list
Otherwise, go with the downtrend-first jagged list
Step 4: Perform the actual swap
4.1. Get the desired trend based on up_trend or down_trend function
4.2. Calculate the next element from the current pair: (element at index i and at index i + 1)
4.1.1. If the array length is greater than i + 2, the next element is the element at index “i +2”.
4.1.2. Else, the next element is None
4.3. Test if the current pair follows its desired trend by calling the “test_jagged” function
4.3.1. If “test_jagged” returns that there is swap needed, update the input array
Step 5: Return the min distance

1.3. Software Implementation

```
# Function designing for jagged list that starts with uptrend
# Return the desired trend for current element given its index
def up_trend(index):
  return not bool(index % 2)

# Function designing for jagged list that starts with downtrend
# Return the desired trend for current element given its index
def down_trend(index):
  return bool(index % 2)
```

Figure 1. Code for uptrend and downtrend function

```python
# Function to check if the values need to be swap
# ---- Inputs: three consecutive numbers in the array
# ------------ & the desired trend
# ---- Outputs: (is swap needed & the swap order)
def test_jagged(n1, n2, next_value, trend):
  # True is up trend, False is down trend
  # Compare n1 and n2 to check if they follow up trend
    is_up = n1 <= n2

    # If the current trend is different than desired trend
    if is_up != trend:
        if next_value is not None:
            # Calculate swap position based on trend and (first and third element)
            swap_position = int(trend) if n1 < next_value else int(not trend)
            if swap_position == 0:
                return (True, n2, n1, next_value) # Swap the first pair
            return (True, n1, next_value, n2) # Swap the second pair
        return (True, n2, n1, next_value) # If the next value is none & different trend, swap the first pair
    return (False, n1, n2, next_value) # If n1 and n2 trend is the same as the desired trend, no swap need
```

Figure 2. Code for testing jagged

```python
# Function to run simulation for swapping array into two types of jagged list
# and calculate distance required
def simulate_run(arr):
    if len(arr) < 2: return 0, 0
    # Initialize distances
    down_distance = up_distance = 0

    # down_n1 and down_n2 are pointers for down jagged list
    # up_n1 and  up_n2 are pointers for up jagged list
    down_n1 = up_n1 = arr[0]
    down_n2 = up_n2 = arr[1]

    for i in range(len(arr) - 1):
        # Get the element at index i + 2 if possible
        next_value = arr[i+2] if i + 2 < len(arr) else None

        # Simulate downtrend jagged list conversion
        down_result = test_jagged(down_n1, down_n2, next_value, down_trend(i))
        if down_result[0] is True: down_distance += 2
        down_n1, down_n2 = down_result[2], down_result[3]

        # Simulate uptrend jagged list conversion
        up_result = test_jagged(up_n1, up_n2, next_value, up_trend(i))
        if up_result[0] is True: up_distance += 2
        up_n1, up_n2 = up_result[2], up_result[3]
    # Return two distances
    return down_distance, up_distance
```

Figure 3. Code for running the simulation

```python
# MAIN function to convert normal list to the jagged list
def jagged_list(arr):
    # If array has 2 elements or less, no change need
    if len(arr) <= 2: return 0
    # Run simulation
    distance_down, distance_up = simulate_run(arr)
    # Decide trend based on distance
    func = up_trend if distance_up < distance_down else down_trend

    # Actual swap
    for i in range(len(arr) - 1):
        desired_trend = func(i)
        next_value = arr[i+2] if i + 2 < len(arr) else None

        # Test if need swap
        result = test_jagged(arr[i], arr[i+1], next_value, desired_trend)

        if result[0] is True: # If need, update array
            arr[i], arr[i + 1] = result[1] , result[2]
            if next_value is not None:
                arr[i+2] = result[3]
    return min(distance_up, distance_down)
```

Figure 4. Code of the main program to convert list into jagged list

1.4. Formal Analysis

Table 9. Formal analysis of down_trend function

CODE	COST	OCCURRENCES
def down_trend(index): return bool(index % 2)	C1	1

Time complexity: Based on the show cost and occurrences, the growth rate of the "down_trend" function – which aims to compute the desired trend based on the current index given that the first trend is downtrend – is: $T(n) = c1 * 1 = c1$. Hence, the "down_trend" function has **O(1)** as its time complexity.

Space complexity:

- Return type: Boolean that contains 4 or 8 bytes

Therefore, the space complexity for the "down_trend" function is **O(1).**

Table 10. Formal analysis of up_trend function

CODE	COST	OCCURRENCES
def up_trend(index): return not bool(index % 2)	C1	1

Time complexity: Based on the show cost and occurrences, the growth rate of the "up_trend" function – which aims to compute the desired trend based on the current index given that the first trend is uptrend – is: $T(n) = c1 * 1 = c1$. Hence, the "up_trend" function has **O(1)** as its time complexity.

Space complexity:

- Return type: Boolean that contains 4 or 8 bytes

Therefore, the space complexity for the "up_trend" function is **O(1).**

Table 11. Formal analysis of test_jagged function

CODE	COST	OCCURRENCES
def test_jagged(n1, n2, next_value, trend):		
is_up = n1 <= n2	C1	1
if is_up != trend:	C2	1
if next_value is not None:	C3	1
swap_position = int(trend) if n1 < next_value else int(not trend)	C4	1
if swap_position == 0:	C5	1
return (True, n2, n1, next_value)	C6	1
return (True, n1, next_value, n2)	C7	1
return (True, n2, n1, next_value)	C8	1
return (False, n1, n2, next_value)	C9	1

Time complexity: Based on the show cost and occurrences, the growth rate of the "test_jagged" function – which aims to compare and swap values if needed when being given three consecutive values in the array and its desired trend – is: $T(n) = c1 + c2 + MAX((c3 + MAX((c4 + c5 + MAX(c6, c7)), c8), c9) = c$. Hence, the "test_jagged" function has **O(1)** as its time complexity.

Space complexity:

- Is_up: Boolean that contains 4 or 8 bytes
- Swap_position: Integer that contains 4 bytes
- Return type: a tuple that contains 4 data values:
  - 1st: Boolean that contains 4 or 8 bytes
  - 2nd, 3rd and 4th: 3 integers. Each contains 4 bytes; thus, 12 bytes in total.

Therefore, the space complexity for the "test_jagged" function is **O(1).**

Table 12. Formal analysis of simulate_run function

CODE	COST	OCCURENCES
def simulate_run(arr):		

if len(arr) < 2: return 0, 0	C1	1
down_distance = up_distance = 0	C2	1
down_n1 = up_n1 = arr[0]	C3	1
down_n2 = up_n2 = arr[1]	C4	1
for i in range(len(arr) - 1):	C5	N - 1
next_value = arr[i+2] if i + 2 < len(arr) else None	C6	N - 1
down_result = test_jagged(down_n1, down_n2, next_value, **down_trend(i)**)	C7	N - 1
if down_result[0] is True: down_distance += 2	C8	N - 1
down_n1, down_n2 = down_result[2], down_result[3]	C9	N – 1
up_result = test_jagged(up_n1, up_n2, next_value, **up_trend(i)**)	C10	N - 1
if up_result[0] is True: up_distance += 2	C11	N - 1
up_n1, up_n2 = up_result[2], up_result[3]	C12	N – 1
return down_distance, up_distance	C13	1

Time complexity: Based on the show cost and occurrences, the growth rate of the "simulate_run" function – which aims to simulate the swap operation for two methods and calculate the required distances – is:

$$T(n) = MAX\big(c1, (c2 + c3 + c4 + (c5 + c6 + c7 + c8 + c9 + c10 + c11 + c12) * (N - 1) + c13)\big)$$
$$= a * N + b$$

Hence, in best-case scenario where the array list size is less than 2, the "simulate_run" function has **O(1)** as its time complexity. However, in most cases, this function takes O(n) time complexity with respect to the input array size.

Space complexity:

- Down_distance, up_distance, up_n1, up_n2, down_n1, down_n2, next_value: Seven integers. Each contains 4 bytes; thus, 28 bytes in total
- Down_result, up_result: Two tuples returned from "test_jagged" list which has O(1) space complexity

Therefore, the space complexity for the "simulate_run" function is **O(1).**

Table 13. Formal analysis of jagged_list function

CODE	COST	OCCURENCES
def jagged_list(arr):		
if len(arr) <= 2: return 0	C1	1
distance_down, distance_up = **simulate_run(arr)**	C2	N
func = up_trend if distance_up < distance_down else down_trend	C3	1

for i in range(len(arr) - 1):	C4	N - 1
desired_trend = func(i)	C5	N - 1
next_value = arr[i+2] if i + 2 < len(arr) else None	C6	N - 1
result = test_jagged(arr[i], arr[i+1], next_value, desired_trend)	C7	N - 1
if result[0] is True:	C8	N - 1
arr[i], arr[i + 1] = result[1] , result[2]	C9	N – 1 as max
if next_value is not None:	C10	N – 1 as max
arr[i+2] = result[3]	C11	N – 1 as max
return min(distance_up, distance_down)	C12	1

Time complexity: As being the main function for this problem, the "jagged_list" contains the "simulate_run" function and traverses through all elements in the array. Hence, its growth rate can be expressed as:

$$T(n) = MAX\big(c1, (c2 * N + c3 + (c4 + c5 + c6 + c7 + c8 + c9 + c10 + c11) * (N - 1) + c12)\big)$$
$$= MAX(c1, ((Cost_1 + Cost_2) * N + b))$$

Where $Cost_1$ is the cost for running the simulation and $Cost_2$ is the cost for swapping or converting operations.

Hence, although the best-case scenario where the array list size is less than 3, the whole program of jagged-list conversion can have **O(1)** in time, the time complexity for most cases will be **O(n).**

Space complexity:

- Down_distance, up_distance, next_value: Three integers. Each contains 4 bytes; thus, 12 bytes in total
- Result: a tuple returned from "test_jagged" list which has O(1) space complexity
- Desired_trend: Boolean which contains 4 or 8 bytes

Therefore, the space complexity for the "jagged_list" function is **O(1).**

In conclusion, in converting a normal list into a jagged list, the algorithm take O(n) in time and O(1) in space.

1.5. Sample run

In order to prove the correctness of the algorithm under different input scenario, the test is carried out with multiple types of arrays.

```
run_tests([
    # Sample jagged list
    [4,3,7,8,6,2,1],
    # Ascending sorted list with odd numbers of elements
    [10, 20, 30, 40, 50],
    # Ascending sorted list with even numbers of elements
    [10, 20, 30, 40],
    # Descending sorted list with odd numbers of elements
    [50, 40, 30, 20, 10],
    # Descending sorted list with even numbers of elements
    [50, 40, 30, 20],
    # Contains: sub jagged list with trend of "up down up down up down"
    [1,2,3,0,4,6,5,8,7,10,9],
    # Contains: sub jagged list with trend of "down up down up down up"
    [1,2,3,0,4,9,10,7,8,5,6],
    # Contains: sub jagged list [10,20,5,7] with trend of "up down up" but result is down up down
    [3, 10, 20, 5, 7, 9]
])
```

Figure 5. Special test cases

```
Initial list [4, 3, 7, 8, 6, 2, 1]
Jagged list: [4, 3, 7, 6, 8, 1, 2]
Distance between two lists:  4
Is the array jagged? True


Initial list [10, 20, 30, 40, 50]
Jagged list: [20, 10, 40, 30, 50]
Distance between two lists:  4
Is the array jagged? True


Initial list [10, 20, 30, 40]
Jagged list: [10, 30, 20, 40]
Distance between two lists:  2
Is the array jagged? True


Initial list [50, 40, 30, 20, 10]
Jagged list: [50, 30, 40, 10, 20]
Distance between two lists:  4
Is the array jagged? True


Initial list [50, 40, 30, 20]
Jagged list: [50, 30, 40, 20]
Distance between two lists:  2
Is the array jagged? True
```

```
Initial list [1, 2, 3, 0, 4, 6, 5, 8, 7, 10, 9]
Jagged list: [1, 2, 0, 4, 3, 6, 5, 8, 7, 10, 9]
Distance between two lists:  4
Is the array jagged? True


Initial list [1, 2, 3, 0, 4, 9, 10, 7, 8, 5, 6]
Jagged list: [2, 1, 3, 0, 9, 4, 10, 7, 8, 5, 6]
Distance between two lists:  4
Is the array jagged? True


Initial list [3, 10, 20, 5, 7, 9]
Jagged list: [3, 10, 5, 20, 7, 9]
Distance between two lists:  2
Is the array jagged? True
```

Figure 6. Jagged program outputs

2. Question 2: Non-recursion conversion

2.1. Conversion

This question asks to convert a recursion function to non-recursion one while remaining the same operations. The given recursion function is as follows:

```
procedure r(x)
  if p(x) then
    a(x)
  else
    b(x)
    r(f(x))
  end if
end r
```

Figure 7. R(x) procedure

To begin with, the recursive function r(x) starts by calling p(x). As the p is an arbitrary predicate, the p(x) function will return true or false. If the result from p(x) is true, the r program will execute a(x) which is an arbitrary procedure. Otherwise, the program will perform another procedure b(x) and recursively start a new routine of r function with the new value of x updated by the function f(x). As a result, the function r(x) will recursively execute b(x) and call itself until the p(x) predicate returns true; thus, having p(x) true as its break condition for the repetitive process. The non-recursion version of r(x) can be written as:

Procedure no_r(x):
  While not p(x) then
  b(x)
  x = f(x)
  end while
  a(x)
end no_r
	Procedure non recursive(x):
	  While the p(x) is false:
	  Execute b(x) procedure
	  Update x by f(x)
	  end while
	  Execute a(x) procedure
	end non recursive

2.2. Sample Implementation and Testing

To test the correctness of the conversion, the arbitrary functions are written as follows:

- P(x) predicate: test if x is divisible by 3, return true; otherwise, return false.
- A(x) procedure: print a message.
- B(x) procedure: print a message.
- F(x) function: increase x by one.

```python
def p(x):
  print("in P with", x)
  # If x is divisible by 3, return true
  return x % 3 == 0
def a(x):
  print("in A")

def b(x):
  print("in B")

def f(x):
  print("in F with", x)
  # Update x by adding 1
  return x + 1
```

Figure 8. Arbitrary functions

```python
def r(x):
  if p(x):
    a(x)
  else:
    b(x)
    r(f(x))
```

Figure 9. R(x) function

```python
def no_r(x):
  while not p(x):
    b(x)
    x = f(x)
  a(x)
```

Figure 10. R(x) function in non-recursive version

When passing "1" into both versions of r(x) function, the same output is observed; thus, the non-recursion function works as expected.

```
Running recursive function
==========================

in P with 1
in B
in F with 1
in P with 2
in B
in F with 2
in P with 3
in A
```

Figure 11. Result from recursive version

```
Running non-recursive function
==========================

in P with 1
in B
in F with 1
in P with 2
in B
in F with 2
in P with 3
in A
```

Figure 12. Result from non-recursive version

3. Question 3: Tower of Hanoi

The Hanoi Tower puzzle is initiated with 3 pegs and one of them has all plates of different sizes placed in descending order from the bottom up. In other word, the largest disc is placed at the bottom while the smallest one is placed on top. The puzzle asks to move all the discs to another peg with a rule not to place the larger plate on top of the smaller one and only shift one disk at a time [1].

3.1. Algorithm Description

As a puzzle rule that the larger disk cannot be placed on a smaller disk, the order in which the disk is placed over a new column should be from largest disk to smallest disk. Therefore, a prerequisite is to have a strategy to move the largest disk to the target peg. In order to be able to move the largest disk that is being squashed by other smaller disks, we must move all the upper disks to another pin. Except for the current pin and the target peg, the problem also gives another peg as a place to temporarily hold the disks, called the spare peg. After successfully moving all the smaller disks to that spare peg, the largest

disk will be able to reach the target pin. To complete the puzzle, all the smaller disks will be moved to the target peg. The general operation steps can be illustrated in figure 11 below.

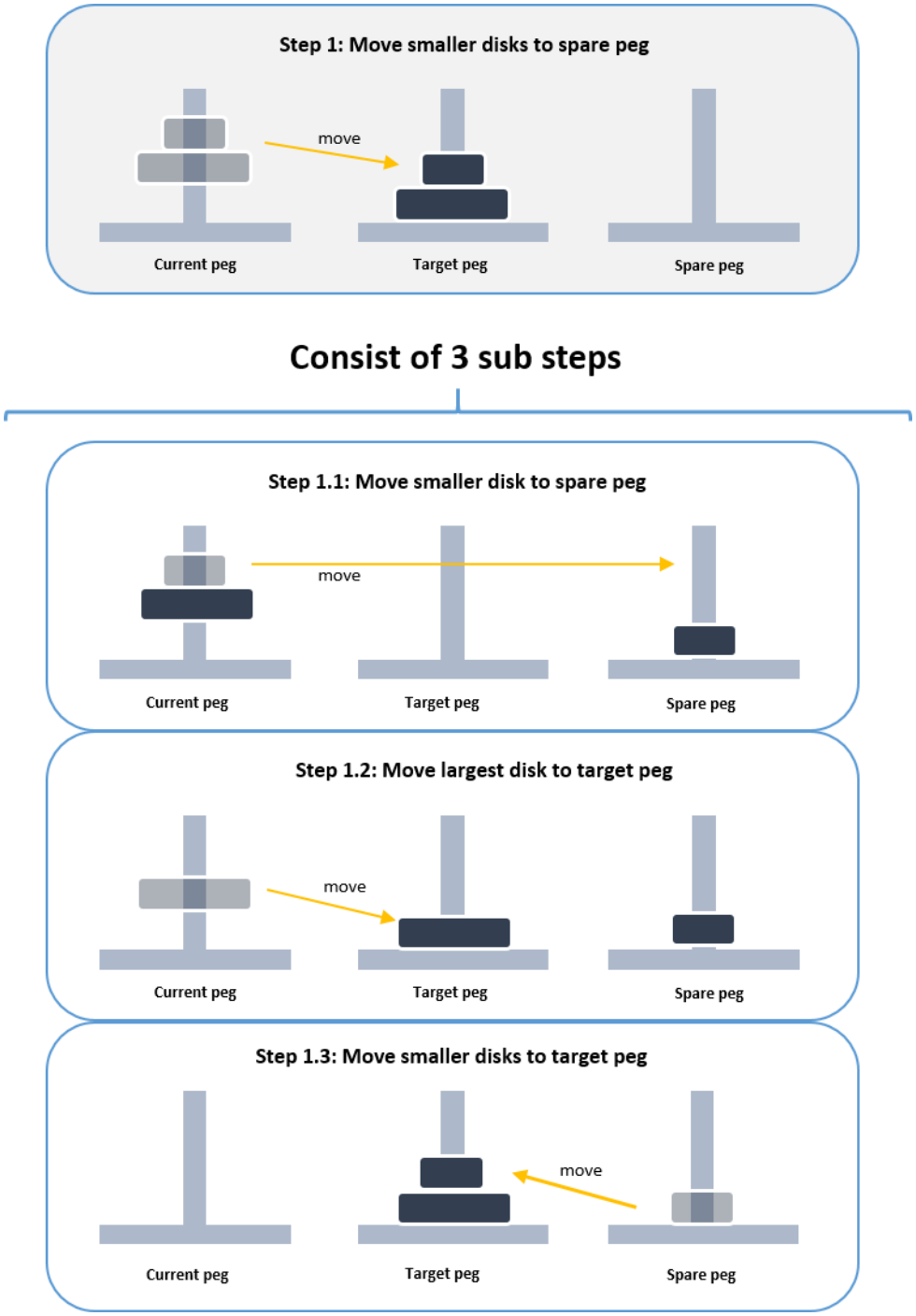


Figure 13. Main operation to solve TOH puzzle

As a condition to obtain the largest disk, in the first step shown in the figure, we encounter a recursive problem of how to move all the smaller disks from the current peg to the spare peg. This problem can be solved in similar method by considering the second largest disk of the whole problem as the largest disk for this problem and the spare peg as our new target peg.

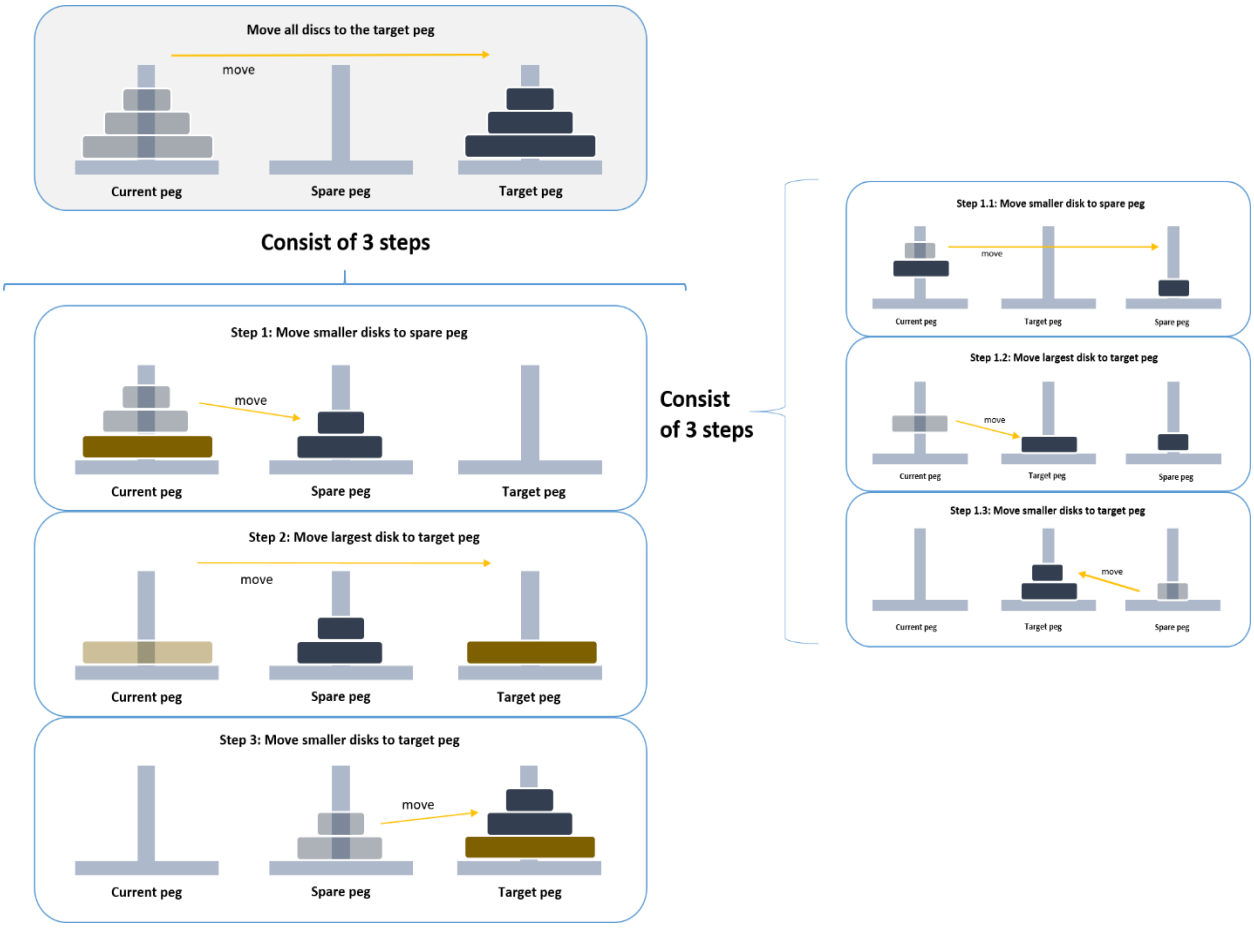


Figure 14. Sub dividing step 1 into smaller operations using same method

As a result, in order to complete the Hanoi Tower puzzle, we must first convert the smaller disks to the spare peg; therefore, narrow the problem down to resolve a smaller list. For each recursive call, the disk list is reduced by one disk. Since we cannot move more than one disk at a time, recursion will be in charge of breaking the disk list and will be no pause until the disk list size is 1 such that the movement can actually be performed. After moving a disk out of the peg, its underlying disk can be transferred to another peg and the process continues until the end of the disk list; in other words, has completed moving the largest disk to the target pin. For the next activity to complete the puzzle, it is referred to move the smaller disks from the spare disk to the target pin which can be accomplished similarly using the same method. In conclusion, regarding moving n disks from their current peg to the target peg, it is solvable by dividing into a collection of two sub similar problems of (1) moving $n-1$ disks to a spare peg so that the bottom disk is transportable and (2) moving $n-1$ disks from the spare to the destination peg. Each of the above sub-problems can be interpreted into two other smaller one; thus, generating a binary-recursive algorithm as the function calls itself twice in one routine.

3.2. Software Implementation

Solution with text description

The software implementation of the discussed algorithm is a void function which takes three input parameters and outputs the movement description onto the console.

Table 14. Input parameters for the tower of Hanoi function

PARAMETERS	TYPE	DESCRIPTION	NOTE
n	Integer	The number of disks be moved	
$from\_peg$	Integer	The current location of the disks	As the puzzle contains 3 pegs, these parameters will accept input from 1 to 3 inclusively
to_peg	integer	The expected location of the disks	

```
def tower_of_hanoi(n, from_peg, to_peg):
  if from_peg > 3 or from_peg < 1 or to_peg > 3 or to_peg < 1:
    print("Invalid input")
    return

  spare_peg = 6 - from_peg - to_peg
  # Move all discs sitting on top of target disc from "from_peg" to the "spare_peg"
  n == 1 or tower_of_hanoi(n - 1, from_peg, spare_peg)

  # Print movement
  print("Move", discs[n - 1][0], "from", pegs[from_peg - 1].strip(), "to", pegs[to_peg - 1].strip())

  # Move back top discs from "spare_peg" to "to_peg"
  n == 1 or tower_of_hanoi(n - 1, spare_peg, to_peg)
```

Figure 15. Function to solve the Tower of Hanoi puzzle with text description

Since the peg position is numbered from 1 to 3, given two pegs – the current peg and the target peg, the spare position can be calculated by the following expression:

$$current + spare + target = 1 + 2 + 3 = 6 \Rightarrow \boldsymbol{spare = 6 - current - target}$$

In addition, as the array of disks will be reduced in every recursion call, the break condition for the recursion will be when the disk array contains only one element which can be shifted to the target peg. The program consists of three main operations:

- **Step 1:** Move smaller disks sitting on top of the target disk from their current position to the spare one
- **Step 2:** Move the target disk (the largest disk) to the destination peg and print out the movement
- **Step 3:** Move smaller disks currently placed on the backup peg to the destination peg

Solution with illustration

By saving the current status of the Tower-of-Hanoi into an array of three stacks – each represents a peg, the following software program provides an illustrated version of each movement solution. In specific, since the peg requires us to pop all top disks to get the bottom one, its operation follows “Last in, first out” rule; thus, can be illustrated using the stack data type. As a result, the function accepts one more variable called “stacks”. For each movement made, the top element on the current peg (from_peg) will be popped and appended to its new peg (to_peg). The “print_toh” function is a support function to display the stacks with its corresponding movement description in table format.

```python
def tower_of_hanoi_2(n, from_peg, to_peg, stacks):
  if from_peg > 3 or from_peg < 1 or to_peg > 3 or to_peg < 1:
    print("Invalid input")
    return

  spare_peg = 6 - from_peg - to_peg
  # Move all discs sitting on top of target disc from "from_peg" to the "spare_peg"
  n == 1 or tower_of_hanoi_2(n - 1, from_peg, spare_peg, stacks)

  # Pop target disc from "from_peg" and push to "to_peg"
  my_disc = stacks[from_peg - 1].pop()
  stacks[to_peg - 1].append(my_disc)
  print_toh(my_disc, from_peg, to_peg, stacks)

  # Move back top discs from "spare_peg" to "to_peg"
  n == 1 or tower_of_hanoi_2(n - 1, spare_peg, to_peg, stacks)
```

Figure 16. Function to solve the Tower of Hanoi puzzle with illustration

3.3. Sample run

In configuring the sample run, it is required to build the initial Tower of Hanoi with all discs being placed in one peg. In this run, peg X is chosen as the starting position where four discs, namely “a”, “b”, “c” and “d”, will be allocated. The program will be asked for moving all discs from peg X to peg Z.

```
    X         Y         Z
---------  ---------  ---------
    a          |          |
   bbb         |          |
  ccccc        |          |
 ddddddd       |          |
```

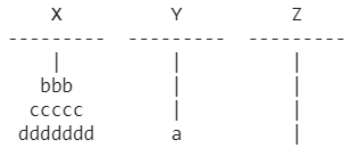
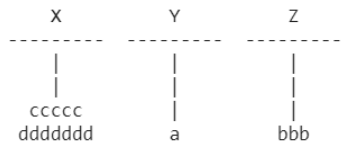
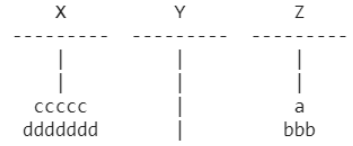
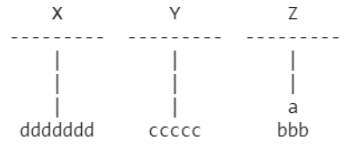
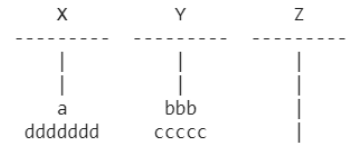
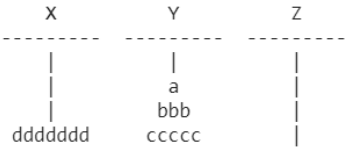
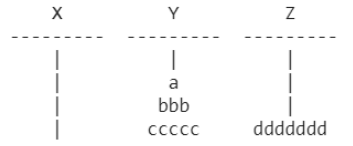
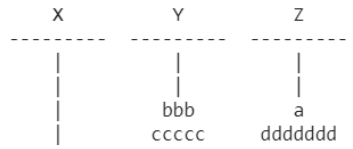
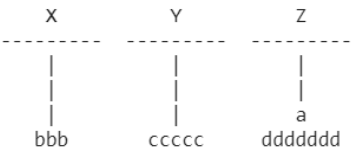
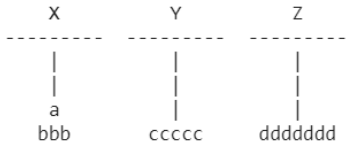
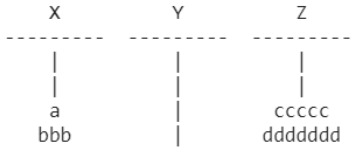
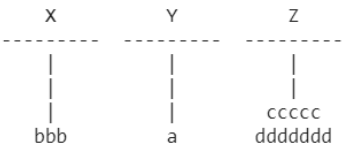
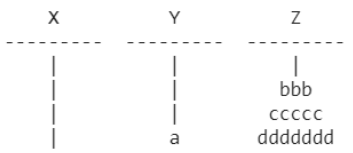
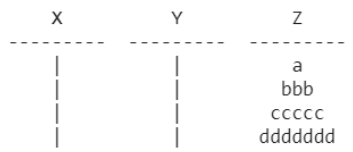
Figure 17. The initial Tower of Hanoi

In result, the program outputs a total of 15 moves to complete transferring four discs to another peg.

```
Move a from X to Y
Move b from X to Z
Move a from Y to Z
Move c from X to Y
Move a from Z to X
Move b from Z to Y
Move a from X to Y
Move d from X to Z
Move a from Y to Z
Move b from Y to X
Move a from Z to X
Move c from Y to Z
Move a from X to Y
Move b from X to Z
Move a from Y to Z
```

Figure 18. Solution TOH with text description

Table 15. Solution TOH with illustration

Move 1	Move 2	Move 3
Description: a move from X to Y 	Description: b move from X to Z 	Description: a move from Y to Z 
Move 4	**Move 5**	**Move 6**
Description: c move from X to Y 	Description: a move from Z to X 	Description: b move from Z to Y 
Move 7	**Move 8**	**Move 9**
Description: a move from X to Y 	Description: d move from X to Z 	Description: a move from Y to Z 
Move 10	**Move 11**	**Move 12**
Description: b move from Y to X 	Description: a move from Z to X 	Description: c move from Y to Z 
Move 13	**Move 14**	**Move 15**
Description: a move from X to Y 	Description: b move from X to Z 	Description: a move from Y to Z 

4. Question 4: Consecutive List

4.1. Algorithm Description

As a list of unique number **from 1 to n** contains **n** elements, the similar laws can be inferred as:

- A unique-element list **from 2 to n** contains **n – 1** elements (missing 1 compared to the above list)
- A unique-element list **from 0 to n** contains **n + 1** elements (adding 0 compared to the above list)
- A unique-element list **from -1 to n** contains **n + 2** elements (adding 0, -1 compared to the above list)

It is clearly seen that there is a close relationship between the value range and the number of elements within the consecutive list which can be generalized using the mathematic formula below:

$$\text{upper\_bound\_range} - \text{lower\_bound\_range} + 1 = \text{number of elements}$$

In other words, $\boldsymbol{max - min + 1 = array\ size}$ $(*)$. Hence, based on this mathematic expression, the unique-element array can be checked to contain consecutives value or not.

As a result, the algorithm is tasked to find out the minimum and maximum value within the array; thus, require the traversal through all elements. In specific, there will two variables to store the current min and max of the array which will be initialized as the first element. To examine the rest, the program will start the traversal from the second element to the end of the array. In every test round, the program must compare the traversed element to the current min and max; if it is greater than max, the max variable must be updated or if it is less than min, the min variable must be updated. When finishing the whole array, there will be a check if the value range is matched with the formula $(*)$. If yes, the array is consecutive; otherwise, it is not.

4.2. Software Implementation

The software program for discussed algorithm is presented as follows:

```
# Function to check if the array contains consecutive elements
# ---- if no, return None
# ---- if yes, return a tuple representing the value range in the range [min, max]
def consecutive(arr):
  if len(arr) == 0: # If array is empty
    return None

  # my_min and my_max is to store the current min & max element so far when travering
  # initialize as first element
  my_min = my_max = arr[0]

  # LOOP from the second element till the end
  for i in range(1, len(arr)):
    # Between the current min and current element, the smaller one will be stored to my_min
    my_min = min(my_min, arr[i])
    # Between the current max and current element, the bigger one will be stored to my_max
    my_max = max(my_max, arr[i])

  # If the [min, max] range matches with the formula, return range; Else, return None
  return (my_min, my_max) if len(arr) == (my_max - my_min + 1) else None
```

Figure 19. Program to check consecutive list

In addition, since the question is asked to take input from the user at running time, it is more ideal to handle invalid input to prevent unexpected or wrong output from the program. There are three types of invalid input: (1) Empty, (2) Non-numeric value and (2) Duplicate value.

```
# Function to validate input from user and convert input to set of integers
# ---- Params: user_input is an array of string
# ---- Output: a set of integer
def validate_input(user_input):
  # Save values into a set to access in O(1)
  set_of_inputs = set()
  arr = []

  if len(user_input) == 0: # If user input is empty
    print("\nError: Empty list")

  # LOOP through every input
  for input in user_input:
    try: # Try to parse input string to integer
      num = int(input)
    except ValueError: # If error happened, the input isn't a number
      print("\nError: Invalid input. Your input has non-numeric value:", input)
      return {}

    if num in set_of_inputs: # If num exists in the set of inputs, num is repeated
      print("\nError: Invalid input. Your input has duplicates:", num)
      return {}

    # If num is valid, save it to the set
    set_of_inputs.add(num)
    arr.append(num)
  return arr
```

Figure 20. Function to validate user input

```
# Print instruction
print("Please input each number separated by whitespace. No duplicates allowed. Ex: 2 3 1 4")

# Split the input string to get list of elements, validate them and save into an array
arr = validate_input(input("\nEnter list: ").strip().split())

if len(arr) > 0: # If the array is not empty => array is valid
  print("\nInput list is:", arr)

  # Check consecutive
  result = consecutive(arr)
  if result is None:
    print("\nNot consecutive!")
  else:
    print("\nConsecutive! List contains elements from", result[0], "to", result[1])
```

Figure 21. Main program of question 4

4.3. Sample run

In order to prove the correctness of the algorithm under different input scenario, the test is carried out with multiple inputs; each input corresponds to one test scenario as follows.

Table 16. Consecutive program output in different cases

NO	TEST INPUT	OUTPUT	RESULT
1	A sorted consecutive array	Enter list: 1 2 3 4 5 6 7 8 9 10 11 Input list is: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11] Consecutive! List contains elements from 1 to 11	Correct
2	A unsorted consecutive array with min, max is positive integer	Enter list: 12 2 11 9 10 6 3 5 7 4 8 Input list is: [12, 2, 11, 9, 10, 6, 3, 5, 7, 4, 8] Consecutive! List contains elements from 2 to 12	Correct
3	A unsorted consecutive array with min, max is negative integer	Enter list: -1 -2 -11 -9 -10 -6 -3 -5 -7 -4 -8 Input list is: [-1, -2, -11, -9, -10, -6, -3, -5, -7, -4, -8] Consecutive! List contains elements from -11 to -1	Correct
4	A unsorted consecutive array with min is negative integer, max is positive integer	Enter list: -1 1 2 -2 9 10 6 3 5 7 4 8 0 Input list is: [-1, 1, 2, -2, 9, 10, 6, 3, 5, 7, 4, 8, 0] Consecutive! List contains elements from -2 to 10	Correct
5	A non-consecutive array	Enter list: 1 2 11 10 6 3 5 7 4 8 Input list is: [1, 2, 11, 10, 6, 3, 5, 7, 4, 8] Not consecutive!	Correct
6	An empty array	Enter list: Error: Empty list	Correct
7	Non-numeric value	Enter list: 1 2 s 4 Error: Invalid input. Your input has non-numeric value: s	Correct
8	An array has duplicates	Enter list: 1 2 3 4 2 5 Error: Invalid input. Your input has duplicates: 2	Correct

5. References

[1] "Program for Tower of Hanoi - GeeksforGeeks", *GeeksforGeeks*, 2021. [Online]. Available: https://www.geeksforgeeks.org/c-program-for-tower-of-hanoi/