**IKEMKA ROMASON ROMANUS**

**BHU/20/04/05/0016**

**CMP 418 ASSIGNMENT**

**Part A**

**Problem 1-1**

**Question: Group 0**

Given the functions:

f1 = n^2, f2 = n , f3= n log n, f4 = 2n, f5 = (log n)^2

Arranged In increasing order of growth:

f2 = n , (linear growth, increase in n grows by the same amount)

f3= n log n, (linear-logarithmic, grows faster due to log function)

f1 = n^2, (quadratic function, grows by the square of n as n increases)

f5 = (log n^2)^2 , (logarithmic- quadratic, grows faster than quadraic due to log function)

f4 = 2^n, (exponential , grows exponentially as n increases)

### Question Group 1

Given the functions:

f1 = log(log(n)^3), f2 = (log n )^3(log 3n), f3 = 3 ^ log n , f4 = n^3log n, f5 =log 3n^n^3, f6 = (log log n)3

Arranged in order of growth:

f6 = (log log n)3,

f1 = log(log(n)^3),

f2 = (log n )^3(log 3n),

f3 = 3 ^ log n,

f4 = n^3log n,

f5 =log 3n^n^3,

### Question Group 2

Given the functions:

f1 = 4^3n, f2 = 2^n^4, f3 = 2^3^n+1, f4= 2^3^2, f5 = 2^5n

Arranged in order of growth:

f4 = 2^3^2

f5 = 2^5n

f1 = 4^3n

f2 = 2^n^4

f3 = 2^3^n+1

**Problem 1-2**

1. If T(n) ∈ O(g(n)), then g(n) ∈ Ω(t(n)).

By definition, t(n) ∈ O(g(n)) means there exist constants c > 0 and n0 such that t(n) ≤ c.g(n) for all n ≥ n0.

This means that g(n) grows at least as fast as t(n) for large n, thus g(n) ∈ Ω t(n)).

1. Θ(αg(n)) = Θ(g(n)), where α > 0.

By definition, Θ (g(n)) means c1g(n) ≤ f(n) ≤ c2g(n) for some constants c1, c2 > 0 .

Multiplying through by a positive constant α, we have αc1​g(n) ≤ αf(n) ≤ αc2g(n).

This shows that scaling by a positive constant does not change the growth rate class, hence Θ(g(n)) ∩ Ω(g(n))

C) Θ(g(n))=O(g(n))∩Ω(g(n)).

Θ(g(n)) is defined as functions that are both O(g(n)) and Ω(g(n)), i.e., they have both upper and lower bounds relative to g(n).

**Problem 2-1**

(a) Given Recurrence Relation:

T(n) = aT(n/2) + Θ(n^2)

Using the Master Theorem:

Compare f(n) = Θ(n^2) with n^c, where c = log\_b a.

Here, b = 2 and f(n) = Θ(n^2).

Thus, c = log base 2 of a.

Case 1: If f(n) = O(n^c), when c > 2, then T(n) = Θ(n^c).

Case 2: If f(n) = Θ(n^c), when c = 2, then T(n) = Θ(n^c log n).

Case 3: If f(n) = Ω(n^c), when c < 2, then T(n) = Θ(f(n)).

Given Solution:

Since T(n) = Θ(n^2 log n), it matches Case 2.

Therefore, c = 2.

log base 2 of a = 2

Solving for 'a':

a = 2^2 = 4

Therefore, the possible value of 'a' is 4.

(b) Given Recurrence Relation:

T(n) = aT(n/3) + Θ(n)

Using the Master Theorem:

Compare f(n) = Θ(n) with n^c, where c = log\_b a.

Here, b = 3 and f(n) = Θ(n).

Thus, c = log base 3 of a.

Case 1: If f(n) = O(n^c), when c > 1, then T(n) = Θ(n^c).

Case 2: If f(n) = Θ(n^c), when c = 1, then T(n) = Θ(n^c log n).

Case 3: If f(n) = Ω(n^c), when c < 1, then T(n) = Θ(f(n)).

Given Solution:

Since T(n) = Θ(n^2), it matches Case 1.

Therefore, c = 2.

log base 3 of a = 2

Solving for 'a':

a = 3^2 = 9

Therefore, the possible value of 'a' is 9.

(c) Given Recurrence Relation:

T(n) = 4T(n/b) + Θ(n^2)

Using the Master Theorem:

Compare f(n) = Θ(n^2) with n^c, where c = log\_b a.

Here, a = 4 and f(n) = Θ(n^2).

Thus, c = log\_b 4.

Case 1: If f(n) = O(n^c), when c > 2, then T(n) = Θ(n^c).

Case 2: If f(n) = Θ(n^c), when c = 2, then T(n) = Θ(n^c log n).

Case 3: If f(n) = Ω(n^c), when c < 2, then T(n) = Θ(f(n)).

Given Solution:

Since T(n) = Θ(n^2), it matches Case 2.

Therefore, c = 2.

log\_b 4 = 2

Solving for 'b':

b^2 = 4

b = 2

Therefore, the possible value of 'b' is 2.

(d) Given Recurrence Relation:

T(n) = 5T(n/b) + Θ(n^5)

Using the Master Theorem:

Compare f(n) = Θ(n^5) with n^c, where c = log\_b a.

Here, a = 5 and f(n) = Θ(n^5).

Thus, c = log\_b 5.

Case 1: If f(n) = O(n^c), when c > 5, then T(n) = Θ(n^c).

Case 2: If f(n) = Θ(n^c), when c = 5, then T(n) = Θ(n^c log n).

Case 3: If f(n) = Ω(n^c), when c < 5, then T(n) = Θ(f(n)).

Given Solution:

Since T(n) = Θ(n^6.006), it matches Case 1.

Therefore, c = 6.006.

log\_b 5 = 6.006

Solving for 'b':

b^6.006 = 5

b = 5^(1/6.006)

Therefore, the possible value of 'b' is 5^(1/6.006).

(e) Given Recurrence Relation:

T(n) = 6T(n/6) + f(n)

Using the Master Theorem:

We need to find f(n).

Here, a = 6 and b = 6.

Thus, c = log\_b a.

c = log base 6 of 6 = 1.

Case 1: If f(n) = O(n^c), when c > 2, then T(n) = Θ(n^c).

Case 2: If f(n) = Θ(n^c), when c = 2, then T(n) = Θ(n^c log n).

Case 3: If f(n) = Ω(n^c), when c < 2, then T(n) = Θ(f(n)).

Given Solution:

Since T(n) = Θ(n^2), it matches Case 2, suggesting f(n) should be such that T(n) has this growth rate.

To achieve T(n) = Θ(n^2), one possible function for f(n) can be:

f(n) = Θ(n^2)

Problem 2-2

**PART B**

**Problem 2-1**

1. Using the Algorithm to sort the array A[50, 30, 80, 10, 40, 60, 20]
2. Initialize Count[i] for all i from 0 to n-1:

Count = [0, 0, 0, 0, 0, 0, 0]

1. Perform the nested loops to update Count[i]:

* For ( i = 0 ):

- ( j = 1 ): ( 50 > 30 ), ( Count[0] ) = 1

- ( j = 2 ): ( 50 < 80 ), ( Count[2] ) = 1

- ( j = 3 ): ( 50 > 10 ), ( Count[0] ) = 2

- ( j = 4 ): ( 50 > 40 ), ( Count[0] ) = 3

- ( j = 5 ): ( 50 < 60 ), ( Count[5] ) = 1

- ( j = 6 ): ( 50 > 20 ), ( Count[0] ) = 4

* For ( i = 1 ):

- ( j = 2 ): ( 30 < 80 ), ( Count[2] ) = 2

- ( j = 3 ): ( 30 > 10 ), ( Count[1] ) = 1

- ( j = 4 ): ( 30 < 40 ), ( Count[4] ) = 1

- ( j = 5 ): ( 30 < 60 ), ( Count[5] ) = 2

- ( j = 6 ): ( 30 > 20 ), ( Count[1] ) = 2

* For ( i = 2 ):

- ( j = 3 ): ( 80 > 10 ), ( Count[2] ) = 1

- ( j = 4 ): ( 80 > 40 ), ( Count[2] ) = 2

- ( j = 5 ): ( 80 > 60 ), ( Count[2] ) = 3

- ( j = 6 ): ( 80 > 20 ), ( Count[2] ) = 1

* For ( i = 3 ):

- ( j = 4 ): ( 10 < 40 ), ( Count[4] ) = 3

- ( j = 5 ): ( 10 < 60 ), ( Count[5] ) = 4

- ( j = 6 ): ( 10 < 20 ), ( Count[6] ) = 2

* For ( i = 4 ):

- ( j = 5 ): ( 40 < 60 ), ( Count[5] ) = 5

- ( j = 6 ): ( 40 > 20 ), ( Count[4] ) = 4

* For ( i = 5 ):

- ( j = 6 ): ( 60 > 20 ), ( Count[6] ) = 3

- ( Count ) after the counting phase: [4, 3,6,0,2,3,1]

Use Count[i] to place elements into S:

A = [50, 30, 80, 10, 40, 60, 20]

Count = [4, 3,6,0,2,3,1]

S = [10, 20, 30, 40, 50, 60, 80]

b) the basic operation is the comparison of elements in the array at line 5

c)It is computing the number of elements in the array that are smaller than the current element. The count value is then used to sort the array by order of how many smaller elements are before it

d) No, the algorithm is not stable. In this algorithm, elements with the same value might change their relative positions because the placement in the sorted array is determined solely by the count values, not by their original order.

e)No, the algorithm is not in place because it uses an additional array S to store the sorted elements. This consumes more space.

f) The overall time complexity of the algorithm is dominated by the counting comparisons step, which has a time complexity of O(n^2). Therefore, the complexity class of this algorithm is:O(n^2)

**Problem 2-2**

### Sort [E,X,A,M,P,L,E,S] using Quicksort

#### 1. Choose a Pivot

**Pivot**: S

#### 2. Partitioning

Rearrange the array such that elements less than the pivot are on the left, and elements greater than the pivot are on the right.

**Initial Array**: [E,X,A,M,P,L,E,S]

We compare each element with S and rearrange:

E (less than S) stays on the left.

X (greater than S) moves to the right.

A (less than S) stays on the left.

M (less than S) stays on the left.

P (less than S) stays on the left.

L (less than S) stays on the left.

E (less than S) stays on the left.

After partitioning: [E,A,M,P,L,E,S,X]

#### 3. Recursively Apply Quicksort

Quicksort on the subarrays left and right of the pivot S.

**Left Subarray**: [E,A,M,P,L,E]

1. **Choose a Pivot**: Choose the last element E as the pivot.
2. **Partitioning**:

Compare each element with E and rearrange:

E (equal to E) stays.

A (less than E) stays on the left.

M (greater than E) moves to the right.

P (greater than E) moves to the right.

L (greater than E) moves to the right.

After partitioning: [A,E,E,M,P,L]

Now we need to sort the subarrays left and right of the pivot E in the new array [A,E,E,M,P,L]

**Right Subarray**: [E,M,P,L]

For the right subarray [E,M,P,L]

1. **Choose a Pivot**: Choose the last element L as the pivot.
2. **Partitioning**:
   * Compare each element with L and rearrange:
     + E (less than L) stays on the left.
     + M (greater than L) moves to the right.
     + P (greater than L) moves to the right.
   * After partitioning: [E,L,M,P][E, L, M, P][E,L,M,P]

Now we need to sort the subarrays left and right of the pivot L in the new array [E,L,M,P]

**Left Subarray**: [E]

For the right subarray [M,P]

1. **Choose a Pivot**: Choose the last element P as the pivot.
2. **Partitioning**:
   * Compare each element with P and rearrange:
     + M (less than P) stays on the left.
   * After partitioning: [M,P][M, P][M,P]

**Left Subarray**: [M] (already sorted because it has only one element)

**Right Subarray**: [P] (already sorted because it has only one element)

### Combining All Sorted Subarrays

* Right subarray of the pivot S: [X]
* Left subarray of the pivot S: [A,E,E,L,M,P]

**Sorted Array**: [A,E,E,L,M,P,S,X]

This gives you the final sorted array [A,E,E,L,M,P,S,X]

**PART C**

**Problem 3-1**

1. Pattern "0 0 0 0 1"

The text has 1000 zeros.

Since our pattern is 5 numbers long, we can start checking at the first position up to the (1000 - 5 + 1) = 996th position.

At each position, we compare the pattern to the next 5 numbers in the text.

In our text, the first 4 numbers of the pattern will always match The 5th number in the pattern ("1") will never match (because the text is all zeros).

At each of the 996 positions, we do 5 comparisons.

The total number of comparisons is 996 positions \* 5 comparisons per position = 4980 comparisons.

(b): Pattern "0 1 0 1 0"

At each position, we compare the pattern to the next 5 numbers in the text.

The first number of the pattern will match (because it's 0).

The second number of the pattern ("1") will never match (because the text is all zeros).

At each of the 996 positions, we do 2 comparisons (the first one matches, the second one fails).

So, the total number of comparisons is 996 positions \* 2 comparisons per position = 1992 comparisons.

Problem 3-1

Brute-force analysis is a straightforward approach to solving problem,

It involves using computer power not intellectual power to run through multiple iterations of a solution until it a problem is solved.It generally always works although not always with an efficient solution,

Strengths:

1. Simplicity: Brute force algorithms are straightforward to implement and understand.

2. Completeness: They guarantee finding a solution if one exists, since they explore all possibilities.

3. Versatility: Can be applied to a wide range of problems without the need for problem-specific optimizations.

Weaknesses:

1. Inefficiency: Often highly inefficient, especially for large input sizes, due to the exponential growth of possibilities.

2. High Computational Cost: Consumes a significant amount of computational resources, including time and memory.

3. Scalability Issues: Not suitable for large-scale problems where more efficient algorithms are required.

1. Minimum Number of character comparisons in Brute force search

Given:

Pattern: BRANDING (length = 8)

Text: THER\_IS\_MORE\_TO\_LIFE\_THAN\_INCREASING\_SPEED (length = 47)

Brute-force string search algorithm: This algorithm compares the pattern with every possible substring of the text that can accommodate the pattern.

STEPS

We compare the pattern over the text from the start to the end, making a total of (47−8+1) = 40 possible starting positions for the pattern in the text.

At each starting position, the algorithm performs character comparisons until either a mismatch is found or the pattern is completely matched.

Start at Position 0: Compare "BRANDING" with "THERE\_IS".

B ≠ T (1 comparison)

Start at Position 1: Compare "BRANDING" with "HERE\_IS\_".

B ≠ H (1 comparison)

Continue this process for each position: "BRANDING" will be compared to each 8-character substring of the text, always failing at the first character.

This process continues for 40 positions, and at each position, only 1 comparison is made before determining the mismatch.

MINIMUM NUMBER OF COMPARISONS

Since the pattern "BRANDING" does not exist in the text, the algorithm will make only one comparison per position before encountering a mismatch.

There for the minimum number of comparisons made by the brute force algorithm in searching for “BRANDING” is 8

**Problem 3-2**

1. Construct a heap for the list by successive key insertions

Given list: 1,8,6,5,3,7,4,2

1. Insert 1:

- Heap: `[1]`

2. Insert 8:

- Heap: `[1, 8]`

- Heapify up: 8 > 1, swap 8 and 1.

- Heap: `[8, 1]`

3. Insert 6:

- Heap: `[8, 1, 6]`

- Heapify up: 6 > 1, swap 6 and 1.

- Heap: `[8, 6, 1]`

4. Insert 5:

- Heap: `[8, 6, 1, 5]`

- Heapify up: 5 > 1, swap 5 and 1.

- Heap: `[8, 6, 5, 1]`

5. Insert 3:

- Heap: `[8, 6, 5, 1, 3]`

- Heapify up: 3 > 1, swap 3 and 1.

- Heap: `[8, 6, 5, 3, 1]`

6. Insert 7:

- Heap: `[8, 6, 5, 3, 1, 7]`

- Heapify up: 7 > 5, swap 7 and 5.

- Heap: `[8, 6, 7, 3, 1, 5]`

- Heapify up: 7 > 6, swap 7 and 6.

- Heap: `[8, 7, 6, 3, 1, 5]`

7. Insert 4:

- Heap: `[8, 7, 6, 3, 1, 5, 4]`

- Heapify up: 4 > 3, swap 4 and 3.

- Heap: `[8, 7, 6, 4, 1, 5, 3]`

8. Insert 2:

- Heap: `[8, 7, 6, 4, 1, 5, 3, 2]`

- Heapify up: 2 > 1, swap 2 and 1.

- Heap: `[8, 7, 6, 4, 2, 5, 3, 1]`

Final Max-Heap:

- [8, 7, 6, 4, 2, 5, 3, 1]

1. Sort the List Using Heapsort

Heapsort involves two main steps: building the heap and then sorting the elements by repeatedly extracting the maximum element from the heap.

Sorting Steps:

1. Build the Max-Heap:

- We have already built the max-heap: `[8, 7, 6, 4, 2, 5, 3, 1]`

2. Sort the Heap:

- Extract 8:

- Swap 8 with the last element (1).

- Heap after swap: `[1, 7, 6, 4, 2, 5, 3, 8]`

- Heapify: Heapify subtree rooted at index 0.

- Heapify result: `[7, 4, 6, 1, 2, 5, 3, 8]`

- Extract 7:

- Swap 7 with the last element (1).

- Heap after swap: `[1, 4, 6, 3, 2, 5, 7, 8]`

- Heapify: Heapify subtree rooted at index 0.

- Heapify result: `[6, 4, 5, 3, 2, 1, 7, 8]`

- Extract 6:

- Swap 6 with the last element (1).

- Heap after swap: `[1, 4, 5, 3, 2, 6, 7, 8]`

- Heapify: Heapify subtree rooted at index 0.

- Heapify result: `[5, 4, 1, 3, 2, 6, 7, 8]`

- Extract 5:

- Swap 5 with the last element (2).

- Heap after swap: `[2, 4, 1, 3, 5, 6, 7, 8]`

- Heapify: Heapify subtree rooted at index 0.

- Heapify result: `[4, 3, 1, 2, 5, 6, 7, 8]`

- Extract 4:

- Swap 4 with the last element (2).

- Heap after swap: `[2, 3, 1, 4, 5, 6, 7, 8]`

- Heapify: Heapify subtree rooted at index 0.

- Heapify result: `[3, 2, 1, 4, 5, 6, 7, 8]`

- Extract 3:

- Swap 3 with the last element (1).

- Heap after swap: `[1, 2, 3, 4, 5, 6, 7, 8]`

- Heap subtree rooted at index 0.

- Heap result: `[2, 1, 3, 4, 5, 6, 7, 8]`

- Extract 2:

- Swap 2 with the last element (1).

- Heap after swap: `[1, 2, 3, 4, 5, 6, 7, 8]`

- Heap subtree rooted at index 0.

- Heap result: `[1, 2, 3, 4, 5, 6, 7, 8]`

- Extract 1:

- The last element is already 1. No swap needed.

- Heap after sorting: `[1, 2, 3, 4, 5, 6, 7, 8]`

Final Sorted List:

- [1, 2, 3, 4, 5, 6, 7, 8]

**Problem 3-3**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 | 5 |
| initial | 68 | 55 | 44 | 79 | 19 | 9 |
| Pass 1 | 55 | 68 | 44 | 79 | 19 | 9 |
| Pass 2 | 44 | 55 | 68 | 79 | 19 | 9 |
| Pass 3 | 44 | 55 | 68 | 79 | 19 | 9 |
| Pass 4 | 19 | 44 | 55 | 68 | 79 | 9 |
| Pass 5 | 9 | 19 | 44 | 55 | 68 | 79 |
| Pass 6 | 9 | 19 | 44 | 55 | 68 | 79 |

1. The basic operation is the comparison of elements and swapping of elements
2. The complexity class is O(n^2)
3. The algorithm is in-place and sorts without using extra space
4. No the algorithm is not stable becasuse it does not acount for duplicates, the code can be modified to account for duplicate’=

**PART D**

**Problem 4-1**

1. Using exhaustive search to consider all possible subsets and check their total weight and value:

Subset: { }

Total Weight: 0

Total Value: $0

Subset: {1}

Total Weight: 2

Total Value: $15

Subset: {2}

Total Weight: 3

Total Value: $20

Subset: {3}

Total Weight: 1

Total Value: $10

Subset: {4}

Total Weight: 2

Total Value: $12

Subset: {1, 2}

Total Weight: 5

Total Value: $35

Subset: {1, 3}

Total Weight: 3

Total Value: $25

Subset: {1, 4}

Total Weight: 4

Total Value: $27

Subset: {2, 3}

Total Weight: 4

Total Value: $30

Subset: {2, 4}

Total Weight: 5

Total Value: $32

Subset: {3, 4}

Total Weight: 3

Total Value: $22

Subset: {1, 2, 3}

Total Weight: 6 (exceeds capacity)

Total Value: $45 (invalid)

Subset: {1, 2, 4}

Total Weight: 7 (exceeds capacity)

Total Value: $47 (invalid)

Subset: {1, 3, 4}

Total Weight: 5

Total Value: $37

Thus a subset of items 1,3,4 is the most valuable subset that fits into the knapsack

B)

V[i][w] represents the maximum value achievable with the first

Fill the table using the recurrence relation:

If the item can be included (weight of item w≥weight of item):

V[i][w]=max(V[i−1][w],V[i−1][w−weight of item]+value of item)

Using dynamic programming a matrix of V[4][5]=37. reveals that the subset 1,3,4 is the most valuable subset using dynamic programming

**Problem 4-2**

a. Recurrence Relation

The recurrence relation for the LCS can be defined as follows:

[ L[i][j] = begin{cases}

0 & text{if } i = 0 text{ or } j = 0

L[i-1][j-1] + 1 & text{if } X[i-1] = Y[j-1]

max(L[i-1][j], L[i][j-1]) & text{if } X[i-1] ne Y[j-1]

end{cases} ]

Here:

- The base case ( L[i][j] = 0 ) when either sequence is empty.

- If the last characters of both substrings match, the LCS length increases by 1.

- If the last characters do not match, the LCS length is the maximum of the LCS lengths obtained by excluding one character from either of the sequences.

b. Algorithm to Compute LCS Length

Here is an algorithm to compute the length of the LCS using dynamic programming:

1. Initialize: Create a 2D array ( L ) of size ((m+1) times (n+1)) where ( m ) and ( n ) are the lengths of sequences ( X ) and ( Y ) respectively. Initialize all entries to 0.

2. Fill the Table:

- For ( i = 1 ) to ( m ):

- For ( j = 1 ) to ( n ):

- If ( X[i-1] = Y[j-1] ):

- ( L[i][j] = L[i-1][j-1] + 1 )

- Else:

- ( L[i][j] = max(L[i-1][j], L[i][j-1]) )

3. Result: The length of the LCS of ( X ) and ( Y ) will be in ( L[m][n] ).

c. Example: Compute LCS for Strings "KADUNA" and "KAKNO"

Let’s compute the length of the LCS for the strings `KADUNA` and `KAKNO` using dynamic programming.

1. Initialize Table:

- Strings: ( X = text{"KADUNA"} ) and ( Y = text{"KAKNO"} )

- Lengths: ( m = 6 ), ( n = 5 )

- Create a table ( L[7][6] ) (since we use indices from 0 to 6 and 0 to 5 respectively).

2. Fill the Table:

Initialize ( L[i][0] = 0 ) for ( i ) from 0 to 6, and ( L[0][j] = 0 ) for ( j ) from 0 to 5.

Here’s the step-by-step filling of the table:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 0 | K | A | K | N | O |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| K | 0 | 1 | 1 | 1 | 1 | 1 |
| A | 0 | 1 | 1 | 2 | 2 | 2 |
| D | 0 | 1 | 1 | 2 | 2 | 2 |
| U | 0 | 1 | 1 | 2 | 2 | 2 |
| N | 0 | 1 | 1 | 2 | 3 | 3 |
| A | 0 | 1 | 2 | 2 | 3 | 3 |

Steps:

- Compare characters and fill each cell based on the recurrence relation.

- For example, when filling cell ( L[2][3] ) (where ( X[1] = A ) and ( Y[2] = K )):

- Since `A` != `K`, use ( L[1][3] ) and ( L[2][2] ) to set ( L[2][3] ) to ( max(1, 2) =

The value at ( L[6][5] ) is the length of the LCS. In this case, ( L[6][5] = 3 ), so the length of the LCS for "KADUNA" and "KAKNO" is 3.

The longest common subsequence in this example is `KAN`, and the length is indeed 3.

b. The algorithm involves filling a table based on the above recurrence relation.

c. Dynamic Programming Approach: Yes, the LCS algorithm using dynamic programming is in-place regarding the table as it does not require extra space proportional to input size apart from the table itself.

d. Stability: The LCS problem is inherently stable as it does not require modifying the original sequences but instead computes lengths and indices.

**Problem 4-3**

a. Algorithm to sort the books by catalogue number and remove duplicates.

Algorithm:

function sortAndRemoveDuplicates(books):

books is an array of tuples (cataloguenumber, bookinfo)

Step 1: Sort the books by catalogue number

sortedbooks = sort(books, key=lambda x: x.cataloguenumber)

Step 2: Remove duplicates

result = []

lastseen = None

for book in sortedbooks:

if book.cataloguenumber != lastseen:

result.append(book)

lastseen = book.cataloguenumber

return result

b. Is the Proposed Algorithm In Place?

The proposed algorithm involves sorting the books and then removing duplicates:

- Sorting: Depending on the sorting algorithm used, the sorting step might be in-place or not.

- In-Place Sorting: Algorithms like Quick Sort and Heap Sort can be done in-place, meaning they do not require extra space beyond the input list (apart from minor overhead).

- Not In-Place Sorting: Merge Sort typically requires additional space proportional to the size of the input list.

- Removing Duplicates: The step to remove duplicates involves creating a new list to store unique books, which means additional space proportional to the number of unique books.

Conclusion: The overall algorithm is not fully in-place due to the additional space required to store unique books. However, the sorting step itself could be performed in-place if an in-place sorting algorithm is used.

c. Performance Evaluation

1. Time Complexity:

- Sorting Step: ( Θ(n log n) ) using efficient sorting algorithms.

- Removing Duplicates Step: ( Θ(n) ) as it involves a single pass through the sorted list.

Overall Time Complexity: ( Θ(n log n) + Θ(n) = Θ(n log n) ).

2. Space Complexity:

- Sorting Step: Depends on the sorting algorithm. For in-place algorithms, it’s ( O(log n) ) or ( O(1) ). For non-in-place algorithms like Merge Sort, it’s ( O(n) ).

- Removing Duplicates Step: Requires ( O(n) ) space for storing unique books.

Overall Space Complexity: ( O(n) ), primarily due to the space needed for the list of unique books.

d. Alternative Solution with Sorting and Checking Consecutive Books

Alternative Approach:

1. Sort the Books:

- Use an efficient sorting algorithm with ( Θ(n log n) ) complexity.

2. Check Consecutive Books:

- After sorting, traverse the sorted list and remove duplicates by comparing each book with the previous one. This requires only a single pass.

Time Complexity Analysis:

1. Sorting Step: ( (n log n) )

2. Checking Consecutive Books:

- After sorting, you only need ( Θ(n) ) time to check and remove duplicates as they are adjacent.

Overall Complexity:

- Time Complexity: ( Θ(n log n) ) for sorting + ( Θ(n) ) for checking duplicates = ( Θ(n log n) ).

- Space Complexity: ( O(n) ) for storing the sorted list and unique books, if a new list is used.

Conclusion:

- The alternative solution is efficient and meets the ( Θ(n log n) ) complexity requirement.

**Problem 4-4**

a. Depth-First Traversal (from vertex a)

Depth-first traversal (DFS) explores as far as possible along each branch before backtracking. Starting from vertex `a` and exploring alphabetically:

1. a - b - c - e - f` - g - (back to f`, then e, then c) h

2. After `h`, backtrack to `c`, then to `b`, then to `a`.

3. Next, explore from `a` to `d`.

So, the depth-first traversal from vertex `a` is: a, b, c, e, f, g, h, d

b. Breadth-First Traversal (from vertex a)

Breadth-first traversal (BFS) explores all neighbors at the present depth prior to moving on to nodes at the next depth level. Starting from vertex `a`:

1. Visit `a`.

2. Visit neighbors of `a` (sorted order): `b`, `d`.

3. Visit neighbors of `b` and `d` (sorted order): `c`, `h`.

4. Visit neighbors of `c`: `e`, `f`.

5. Visit neighbors of `e` and `f`: `g`.

So, the breadth-first traversal from vertex `a` is: a, b, d, c, h, e, f, g

c. Topological Ordering of Vertices Using the Source Removal Algorithm

Topological ordering is only possible in a Directed Acyclic Graph (DAG). In each step, we select a vertex with no incoming edges (a source), remove it along with its edges, and repeat.

1. Identify vertices with no incoming edges: `a`.

2. Remove `a`, update the graph: Remaining vertices: `b, c, d, e, f, g, h`.

3. Identify vertices with no incoming edges: `b, d`.

4. Remove `b`, update the graph: Remaining vertices: `c, d, e, f, g, h`.

5. Identify vertices with no incoming edges: `c, d`.

6. Remove `c`, update the graph: Remaining vertices: `d, e, f, g, h`.

7. Identify vertices with no incoming edges: `d`.

8. Remove `d`, update the graph: Remaining vertices: `e, f, g, h`.

9. Identify vertices with no incoming edges: `e`.

10. Remove `e`, update the graph: Remaining vertices: `f, g, h`.

11. Identify vertices with no incoming edges: `f`.

12. Remove `f`, update the graph: Remaining vertices: `g, h`.

13. Identify vertices with no incoming edges: `g`.

14. Remove `g`, update the graph: Remaining vertices: `h`.

15. Remove `h`, graph is empty.

Using the source removal algorithm, the topological ordering of vertices is : a, b, c, d, e, f, g, h