**ADVANCED DATA STRUCTURE AND ALGORITHM**

**Submitted by:**

*Meena Rhoshini. C*

*CSE – ‘A’*

**ADVANCED DATA STRUCTURE AND ALGORITHM**

**ASSIGNMENT 1:**

**Problem Statement:**

*You are given an array of integers, and you are required to sort this array using one of the following sorting algorithms: Bubble Sort, Selection Sort, or Insertion Sort. Your task is to implement the chosen sorting algorithm and analyse its time complexity.*

**Problem Analysis:**

*1. Implement one of the sorting algorithms mentioned above (Bubble Sort, Selection Sort, or Insertion Sort) in Python.*

*2.Apply your sorting algorithm to the given array of integers.*

*3.Provide the sorted array as the output.*

*4.Analyze the time complexity of the sorting algorithm you implemented. Explain whether it is a stable sort and how it performs on different types of input data (e.g., already sorted, reverse sorted, random data).*

*5.Compare the time complexity of your chosen sorting algorithm with at least one other sorting algorithm (e.g., Quick Sort, Merge Sort, or Python's built-in sorted function). Explain the differences and scenarios where one algorithm might be preferred over the other.*

**Input:**

*An unsorted list of integers (e.g., [5, 2, 9, 1, 5, 6]).*

**Output:**

*The sorted list of integers.*

**Algorithm:**

*STEP 1: Choose one of the sorting algorithms (Bubble Sort, Selection Sort, or Insertion Sort) and implement it in Python.*

*STEP 2: Apply your chosen sorting algorithm to the provided input array.*

*STEP 3: Provide the sorted array as the output.*

*STEP 4: Analyse the time complexity of the sorting algorithm and discuss its stability and performance on different input data.*

*STEP 5: Compare the time complexity of your chosen sorting algorithm with at least one other sorting algorithm, and explain when you would prefer one over the other.*

**Program:**

Let’s choose Bubble Sort for this task.

* Python code for Bubble Sort:

def bubble sort(arr):

n = len(arr)

for i in range(n):

for j in range(0, n-i-1):

if arr[j] > arr[j+1]:

arr[j], arr[j+1] = arr[j+1], arr[j]

return arr

* To apply the above python to sort the array:

arr = [64, 34, 25, 12, 22, 11, 90]

sorted\_arr = bubble\_sort(arr)

print(sorted\_arr)

**Input:** *[64, 34, 25, 12, 22, 11, 90]*

**Output:** *[11, 12, 22, 25, 34, 64, 90]*

**Time Complexity Analysis:**

* *Bubble Sort has a worst-case and average time complexity of O(n^2), where n is the number of items being sorted. This happens when the input array is reverse sorted, meaning it's in descending order, as every pair of adjacent elements needs to be swapped.*
* *When the list is already sorted (best-case), the complexity of bubble sort is only O(n). In this case, Bubble Sort works optimally as it only needs to go through the array once.*
* *For a randomly ordered array, Bubble Sort would still have a time complexity of O(n^2) on average.*
* *Bubble Sort is a stable sort. This means that two equal elements will maintain their original order in sorted output. If stability is a concern in your use case (for example, when you're sorting a list of objects by one attribute, and you want to maintain the order of objects with equal keys), Bubble Sort would be an appropriate choice.*

**Comparing with Quick Sort:**

* *Quick Sort has a worst-case time complexity of O(n^2), but this scenario is quite rare. It has an average time complexity of O(n log n) and it’s one of the fastest algorithms for sorting large datasets on average.*
* *However, Quick Sort is not a stable sort.*
* *If stability is not a concern and average performance matters more than the worst-case scenario, Quick Sort can be preferred over Bubble Sort.*
* *On the other hand, if the data is mostly sorted or the array size is small, Bubble Sort can be faster than Quick Sort.*

**Difference between sorting algorithm:**

* ***Bubble Sort****: This is a simple comparison-based algorithm. It works by repeatedly swapping the adjacent elements if they are in the wrong order. It has a worst-case and average time complexity of O(n^2). Bubble Sort is stable and performs well on small lists or nearly sorted lists, but it’s inefficient on larger lists.*
* ***Selection Sort****: This algorithm sorts an array by repeatedly finding the minimum element from the unsorted part and putting it at the beginning. It also has a time complexity of O(n^2) in all cases. Selection Sort is not stable and, like Bubble Sort, it’s generally inefficient on large lists.*
* ***Insertion Sort****: This algorithm builds a sorted list one item at a time. It’s much less efficient on large lists than more advanced algorithms like Quick sort, Heap sort, or Merge Sort. However, Insertion Sort provides several advantages:*

1. *It has a simple implementation.*
2. *It’s efficient for (quite) small data sets and mostly sorted lists, much like Bubble Sort.*
3. *Unlike Bubble Sort, however, it’s stable.*

* ***Quick Sort****: This is a highly efficient sorting algorithm and is based on partitioning of array of data into smaller arrays. It performs well on randomized arrays or large datasets with an average case time complexity of O(n log n). However, it has a worst-case time complexity of O(n^2), which occurs when the array is already sorted in ascending or descending order. Quick Sort is not stable.*
* ***Merge Sort****: This is an efficient, stable sorting algorithm that makes use of the divide-and-conquer principle. It has a best, worst and average time complexity of O(n log n). Merge Sort requires more space compared to other sort algorithms as it’s not an in-place sort (it requires additional storage for temporarily storing the divided arrays). It performs well on large datasets or data stored in slower mediums like disk storage.*

**Summary:**

* *For small datasets or nearly sorted lists, Bubble Sort or Insertion Sort could be used.*
* *For larger datasets or lists that are in random order, Quick Sort or Merge Sort would usually be more efficient.*
* *If you need a stable sort (i.e., preserve the relative order of equal sort items), then Bubble Sort, Insertion Sort or Merge Sort would be appropriate.*
* *If memory usage is a concern, then you might want to avoid Merge Sort as it requires additional storage space.*

**ASSIGNMENT 2:**

**Problem Statement:**

*You are given a sequence of matrices with dimensions that are suitable for matrix multiplication. Your task is to find the optimal way to parenthesize the matrices to minimize the total number of scalar multiplications required to compute their product.*

**Problem Analysis:**

*1.Implement a dynamic programming algorithm in Python to solve the matrix chain multiplication problem.*

*2.Apply your algorithm to the given sequence of matrices and find the optimal parenthesization.*

*3.Calculate and provide the minimum number of scalar multiplications required for the optimal parenthesization.*

*4.Explain the dynamic programming approach you used, including the initialization, recurrence relation, and how you reconstructed the optimal parenthesization.*

*5.Analyze the time and space complexity of your algorithm, and discuss its efficiency in solving large instances of the problem.*

**Input:**

*A list of matrices, each represented by its dimensions. For example, a list of matrices [A, B, C] could be represented as [(2, 3), (3, 4), (4, 2)] where the dimensions of matrix A are 2x3, the dimensions of matrix B are 3x4, and the dimensions of matrix C are 4x2.*

**Output:**

* *The optimal parenthesization of matrices as a sequence of matrix multiplications.*
* *The minimum number of scalar multiplications required for the optimal parenthesization.*

**Algorithm:**

*STEP 1: Implement a dynamic programming algorithm to solve the matrix chain multiplication problem in Python.*

*STEP 2: Apply your algorithm to the provided list of matrices to find the optimal parenthesization.*

*STEP 3: Calculate and provide the minimum number of scalar multiplications required for the optimal parenthesization.*

*STEP 4: Explain the dynamic programming approach, including the initialization, recurrence relation, and reconstruction of the optimal parenthesization.*

*STEP 5: Analyse the time and space complexity of your algorithm and discuss its efficiency for large instances of the problem.*

**Program:**

* Python code for implementing the dynamic programming algorithm:

def matrix\_chain\_order(dimensions):

n = len(dimensions)

m = [[0 for \_ in range(n)] for \_ in range(n)]

s = [[0 for \_ in range(n)] for \_ in range(n)]

for l in range(2, n):

for i in range(n - l):

j = i + l - 1

m[i][j] = float('inf')

for k in range(i, j):

q = m[i][k] + m[k + 1][j] + dimensions[i][0] \* dimensions[k + 1][1] \* dimensions[j + 1][1]

if q < m[i][j]:

m[i][j] = q

s[i][j] = k + 1

return m, s

* Applying the code to the given sequence of matrix:

matrices = [(2, 3), (3, 4), (4, 2)]

m, s = matrix\_chain\_order(matrices)

* Finding the optimal parenthesization and the minimum number of scalar multiplications:

def print\_optimal\_parens(s, i, j):

if i == j:

print(f"A{i}", end="")

else:

print("(", end="")

print\_optimal\_parens(s, i, s[i][j] - 1)

print\_optimal\_parens(s, s[i][j], j)

print(")", end="")

print("Optimal parenthesization: ", end="")

print\_optimal\_parens(s, 0, len(matrices) - 2)

print("\nMinimum number of scalar multiplications: ", m[0][len(matrices) - 2])

**Input:***[(2,3), (3,4), (4,2)]*

**Output:** *Optimal parenthesization: ((A0 A1) A2)*

*Minimum number of scalar multiplications: 24*

**Explanation:**

*The problem is known as the Matrix Chain Multiplication problem. It's a classic optimization problem that can be solved using dynamic programming. Here's how the dynamic programming approach works:*

***1. Initialization****: Create a 2D array `m` where `m[i][j]` represents the minimum number of scalar multiplications needed to compute the matrix `A[i]A[i+1]...A[j]`. Initialize all diagonal elements in `m` to 0 because a single matrix doesn't need any multiplication.*

***2.******Recurrence Relation****: For chains of length `l` from 2 to `n`, and for each `i` from 1 to `n-l+1`, set `j = i+l-1` and compute `m[i][j]` as follows:*

*m[i][j] = min {m[i][k] + m[k+1][j] + p[i-1]\*p[k]\*p[j]} for i <= k < j*

*Here, `p[]` is an array such that `p[i-1] x p[i]` gives the dimensions of matrix `i`.*

***3.******Reconstruction of Optimal Parenthesization****: Create another 2D array `s` to store the index at which to split the product in order to minimize the cost. After computing `m`, you can use `s` to determine where to split the chain:*

*- If `i != j`, then print "("*

*- Recursively call the function on the subproblem `(i, s[i][j])`*

*- Recursively call the function on the subproblem `(s[i][j]+1, j)`*

*- If `i != j`, then print ")"*

*This approach ensures that all smaller subproblems are solved before solving a particular problem. The time complexity of this approach is O(n^3), and it requires O(n^2) auxiliary space for storing the `m` and `s` tables.*

**Time Complexity:**

*The time complexity of the dynamic programming solution for the Matrix Chain Multiplication problem is O(n^3). This is because there are O(n^2) subproblems (we need to fill up an n x n table), and it takes O(n) time to solve each subproblem (as we need to find the minimum over n possibilities for each cell in the table).*

**Space Complexity:**

*The space complexity of the algorithm is O(n^2). This is due to the 2D arrays used to store the cost of multiplication operations (m array) and the position at which to split the chain of matrices (s array).*

*So, while this algorithm is efficient in terms of solving the problem with minimal multiplications, it can be quite space-consuming for large inputs due to its quadratic space complexity.*

**Efficiency:**

*The dynamic programming solution for the Matrix Chain Multiplication problem is quite efficient for large instances of the problem, given its polynomial time complexity of O(n^3). This means that the time taken by the algorithm grows cubically with the size of the input, which is manageable for large inputs.*

*However, it's important to note that while the time complexity is polynomial, the constant factors involved in the cubic time complexity can make the algorithm slow for very large instances of the problem. The algorithm needs to fill up an n x n table and find the minimum over n possibilities for each cell in the table, which can be computationally intensive.*

**Summary:**

*In terms of space complexity, the algorithm requires O(n^2) space to store the cost of multiplication operations and the position at which to split the chain of matrices. This quadratic space complexity can also be a concern for very large inputs as it might require a significant amount of memory.*

*So, while dynamic programming provides an efficient solution to this problem compared to naive approaches, there are still practical limitations when dealing with very large instances of the problem due to its cubic time complexity and quadratic space complexity.*

**ASSIGNMENT 3:**

**Problem Statement:**

*You are tasked with solving the N-Queens problem using a backtracking algorithm. The N-Queens problem is to place N chess queens on an N×N chessboard so that no two queens threaten each other. Thus, a solution requires that no two queens share the same row, column, or diagonal. Your goal is to implement the algorithm, find all possible solutions for a given N, and analyze its time complexity.*

**Problem Analysis:**

*1.Implement a backtracking algorithm in Python to solve the N-Queens problem.*

*2.Apply your algorithm to find all possible solutions for a given N (e.g., N = 4 or N = 8). Ensure that you generate all unique solutions.*

*3.Present the solutions as chessboard representations, indicating the placement of queens (e.g., using 'Q' for queens and '.' for empty squares).*

*4.Explain the backtracking approach, including how you generate and validate solutions and how you handle conflicts between queens.*

*5.Analyze the time complexity of your algorithm and discuss its efficiency for larger values of N.*

**Input:**

*An integer N (N ≥ 4) representing the size of the N×N chessboard and the number of queens to place.*

**Output:**

*All possible solutions to the N-Queens problem for the given N, presented as chessboard representations.*

**Algorithm:**

*STEP 1: Implement a backtracking algorithm to solve the N-Queens problem in Python.*

*STEP 2: Apply your algorithm to find all possible solutions for the provided value of N (e.g., N = 4 or N = 8).*

*STEP 3: Present the solutions as chessboard representations, indicating the placement of queens (e.g., using 'Q' for queens and '.' for empty squares).*

*STEP 4: Explain the backtracking approach, including solution generation, validation, and conflict resolution.*

*STEP 5: Analyse the time complexity of your algorithm and discuss its performance for larger N values.*

**Program:**

* Python Code for N-Queens problem:

def solve\_n\_queens(N):

def can\_place(pos, ocuppied\_positions):

for i in range(len(ocuppied\_positions)):

if ocuppied\_positions[i] == pos or \

ocuppied\_positions[i] - i == pos - len(ocuppied\_positions) or \

ocuppied\_positions[i] + i == pos + len(ocuppied\_positions):

return False

return True

def place\_queens(N, index, ocuppied\_positions, result):

if index == N:

result.append(ocuppied\_positions[:])

return result

for i in range(N):

if can\_place(i, ocuppied\_positions):

ocuppied\_positions.append(i)

result = place\_queens(N, index + 1, ocuppied\_positions, result)

ocuppied\_positions.pop()

return result

result = []

place\_queens(N, 0, [], result)

return [["."\*i + "Q" + "."\*(N-i-1) for i in sol] for sol in result]

* Apply the above function to solve the problem as follows:

N = 4

solutions = solve\_n\_queens(N)

for solution in solutions:

for row in solution:

print(row)

print("\n")

**Input:** N = 4

**Output:**

. Q . . . . Q .

. . . Q Q . . .

Q . . . . . . Q

. . Q . . Q . .

*These are the two possible solutions to the 4-Queens problem. Each “Q” represents a queen and each “.” represents an empty space. The queens are placed in such a way that no two queens threaten each other.*

**Explanation:**

*Backtracking is a general algorithm for finding all (or some) solutions to some computational problems, notably constraint satisfaction problems. It incrementally builds candidates for the solutions, and abandons a candidate as soon as it determines that the candidate cannot possibly be extended to a valid solution. The classic example of the use of backtracking is the N-queens puzzle. Here’s how it works:*

* ***Solution Generation****: The algorithm starts by placing a queen in the first row. Then it moves on to the next row and tries to find a safe spot for the second queen. It continues this process until it has placed queens in all rows.*
* ***Validation****: Before placing each queen, it checks if the current spot is safe. A spot is considered safe if no other queens threaten the spot. This is done by checking if any queen exists in the same column or on the same diagonal.*
* ***Conflict Resolution****: If a safe spot is found, the algorithm moves on to the next row. If no safe spot is found in the current row, it means that the previous queens are not placed correctly. So it goes back (backtracks) to the previous row and changes the position of the queen.*
* ***Backtracking****: If it has tried all possible positions in the current row and none of them is safe, it backtracks to the previous row and changes the position of that queen. If no safe spots are found for the first queen, it means there are no solutions to the problem.*

*This process continues until all queens are placed on the board correctly (a solution is found), or all possible board configurations have been exhausted (no solution exists). For each valid placement of queens, a solution is recorded, and then the algorithm backtracks to find more solutions.*

**Time Complexity:**

*The time complexity of the N-Queens problem solution using the backtracking algorithm is O(N!). This is because, in the worst-case scenario, we might end up exploring all possible permutations of the N rows of the board (i.e., N factorial or N!).*

*However, it’s important to note that the backtracking algorithm doesn’t always result in worst-case performance. The reason is that the algorithm prunes a large portion of the search space. As soon as it determines that a queen cannot be placed in a particular position without attacking another, it abandons that position and backtracks. This pruning of the search space makes the algorithm significantly more efficient than a brute-force approach that would attempt to place queens in every possible configuration and check each for validity.*

*For small values of N (e.g., N <= 10), the algorithm performs quite well and is able to find all solutions quickly. However, as N grows larger, the number of possible board configurations grows exponentially, and so does the time taken by the algorithm. For example, there are 92 solutions for N=8, but there are 2,680 solutions for N=9, and 14,200 solutions for N=10.*

*Therefore, while the backtracking algorithm is an effective method for solving the N-Queens problem, its performance can degrade quickly for larger values of N due to its factorial time complexity.*

**When to use?**

*Here are some scenarios where backtracking can be used:*

* ***Puzzle Solving****: Backtracking is used in solving puzzles like the N-Queens problem, Sudoku, or the Knight’s Tour problem where the solution involves a sequence of steps.*
* ***Combinatorial Optimization****: Problems like the Travelling Salesman Problem, where you need to find the shortest possible route that covers all given destinations, can be solved using backtracking.*
* ***Generating all permutations of a set****: Backtracking can be used to generate all possible permutations of a given set or string.*
* ***Partitioning problems****: Problems that involve partitioning a set into subsets that meet certain criteria, such as the subset sum problem, can be solved using backtracking.*

**Summary:**

*Backtracking involves trying out all possible solutions and therefore can be very slow for problems with large inputs or large solution spaces. The time complexity of backtracking algorithms is often high, and they can take a long time to run on large datasets. While backtracking is a powerful technique, it’s important to consider the nature of the problem and the efficiency of the algorithm before deciding to use it*.