LAB MANUAL

FOR

Computer Algorithms Lab (CSE-2204)

Credit: 1.5, Contact hour: 3 Hours Per week

Department of
Computer Science and Engineering
Varendra University
Rajshahi, Bangladesh

Varendra University

Department of Computer Science and Engineering

Rajshahi, Bangladesh

Course Code: CSE 2204

Course Title: Computer Algorithms Lab

Semester: Spring, 2025

LIST OF EXPERIMENTS

Lab No.	Lab Topic
Lab 1	Elementary Problems : Loop, Arrays, Functions, Recursions, Matrix
Lab 2	Searching Techniques : Linear Search, Binary Search
Lab 3	Elementary Sorting: Bubble Sort, Insertion Sort, Selection Sort
Lab 4	Divide and Conquer: Quick-Sort & Merge-Sort
Lab 5	Greedy Design: Fractional Knapsack, Activity Scheduling.
Lab 6	Dynamic Programming: Elementary – Coin Toss, 0/1 Knap Sack
Lab 7	**Assessment Test 1**: Based on Lab 2- 6: Searching, Sorting & DP
Lab 8	Dynamic Programming: LCS, Rod Cutting
Lab 9	Graph: Elementary – Adjacency List, Adjacency Matrix, Weighted Matrix, Self-Loop etc.
Lab 10	Graph Coding: Searching- BFS, DFS
Lab 11	Graph Coding: Shortest Path- Dijkstra, Bell-Man Ford
Lab 12	Graph Coding: MST – Prims, Kruskal
Lab 13	**Assessment Test 2**: Based on Lab 02 - 12
Lab 14	**Assessment Test 3:** Based on lab work

Assessment and Marks Distribution
❖ Attendance (AT) (10%) ❖ Regular Assessment and Performance (20%) ❖ Viva Voce (10%) ❖ Lab Quiz (20%) ❖ Lap Report (LR) (20%) ❖ Lab Final (LF) (20%)

Lab Instructions:

- Students should come with thorough preparation for the experiment to be conducted.
- Students will not be permitted to attend the laboratory unless they bring the practical record fully completed in all respects pertaining to the experiment conducted in the previous class.
- Be honest in developing and representing your program. If a particular program output appears wrong repeat the program carefully.
- Strictly observe the instructions given by the Faculty / Lab. Instructor.
- Take permission before entering in the lab and keep your belongings in the racks.
- NO FOOD, DRINK, IN ANY FORM is allowed in the lab.
- TURN OFF CELL PHONES! If you need to use it, please keep it in bags.
- Avoid all horseplay in the laboratory. Do not misbehave in the computer laboratory. Work quietly.
- Save often and keep your files organized
- Do not reconfigure the cabling/equipment without prior permission.
- Do not play games on systems.
- If you finish early, spend the remaining time to complete the laboratory report writing. Come equipped with calculator and other materials related to lab works.
- Handle instruments with care. Report any breakage or faulty equipment to the Instructor. Shutdown your computer you have used for the purpose of your experiment before leaving the Laboratory.
- .Violation of the above rules and etiquette guidelines will result in disciplinary action.

Lab 01

Elementary Problems : Loop, Arrays, Functions, Recursions, Matrix

SN	Problems	Outcome/Understanding
01.	Compute the sum of the first N natural numbers using loops.	***Loop***
02.	Calculate the sum of all elements in a one-dimensional array.	***Array***
03.	Using a user defined function, find the maximum element in an array.	***Array***
04.	Compute the factorial of a given number using recursion.	***Recursion***
05.	Multiply two matrices of order M×N and N×P.	***Matrix, 2D Array***

Aims & Objectives

- Demonstrate use of for-loops for iterative computation.
- Practice array traversal and accumulation.
- Implement modular code via user-defined functions.
- Understand recursion and its stack-based execution.
- Apply nested loops for two-dimensional matrix operations.

Theory

Loops: Loops are control structures that execute a block of code repeatedly based on a condition. Common types include:

I. For Loop: Executes a block a specific number of times.
II. While Loop: Continues execution as long as a condition is true.
III. Do-While Loop: Executes the block at least once before checking the condition.

Arrays: Arrays are linear data structures that store elements of the same type in contiguous memory locations. They allow constant time access to elements using indices. However, their size is fixed upon creation, and inserting or deleting elements can be costly due to shifting operations.

Functions: Functions are reusable blocks of code designed to perform specific tasks. They promote modularity and code reusability. Functions can accept inputs (parameters) and return outputs (return values).

Recursion: Recursion is a programming technique where a function calls itself to solve a problem. It's particularly useful for problems that can be broken down into similar subproblems, such as computing factorials or traversing tree structures. However, excessive recursion can lead to stack overflow errors.

Matrices: Recursion is a programming technique where a function calls itself to solve a problem. It's particularly useful for problems that can be broken down into similar subproblems, such as computing factorials or traversing tree structures. However, excessive recursion can lead to stack overflow errors.

Pseudocode

Problem 1: Sum of N natural numbers

```
INPUT: N
sum ← 0
FOR i ← 1 TO N DO
    sum ← sum + i
ENDFOR
OUTPUT: sum
```

Problem 2: Sum of array elements

```
INPUT: array A[0..n-1], n
sum ← 0
FOR i ← 0 TO n-1 DO
    sum ← sum + A[i]
ENDFOR
OUTPUT: sum
```

Problem 3: Maximum in array using function

```
FUNCTION findMax(A[0..n-1], n)
    maxVal ← A[0]
    FOR i ← 1 TO n-1 DO
        IF A[i] > maxVal THEN
            maxVal ← A[i]
        ENDIF
    ENDFOR
    RETURN maxVal
ENDFUNCTION

INPUT: array A[ ], n
maxElement ← findMax(A, n)
OUTPUT: maxElement
```

Problem 4: Factorial via recursion

```
FUNCTION factorial(n)
    IF n = 0 OR n = 1 THEN
        RETURN 1
    ELSE
        RETURN n * factorial(n-1)
    ENDIF
ENDFUNCTION

INPUT: n
result ← factorial(n)
OUTPUT: result
```

Problem 5: Matrix multiplication

```
INPUT: A[M][N], B[N][P]
DECLARE C[M][P] initialized to 0
FOR i ← 0 TO M-1 DO
    FOR j ← 0 TO P-1 DO
        FOR k ← 0 TO N-1 DO
            C[i][j] ← C[i][j] + A[i][k] * B[k][j]
        ENDFOR
    ENDFOR
ENDFOR
OUTPUT: C
```

Observation Table

Problem No.

ID	Input	Expected Output	Actual Output	Execution Time
1				
2				
3				

Student Tasks

1. **Implementation:** Write the code for each of the problems in your preferred programming language. This task must be completed individually, and any collaborative submissions will be treated as plagiarism.

2. **Observation Table:** Complete the observation table with your findings.

3. **Report:** Prepare and submit a reflective report summarizing your implementation, findings, plots, and discussions describing any challenges encountered during experimentation and how they were addressed.

4. **Viva:** A viva session will be scheduled during the next lab slot to verify individual student reports and code, and marks for the assignment will be substantially influenced by performance in that session.

Lab 02

Searching Techniques: Linear Search, Binary Search

SN	Problems	Outcome/Understanding
01.	Implement linear search to find the index of a target element in an unsorted array.	***Linear Search***
02.	Implement binary search to locate a target element in a sorted array.	***Binary Search***

Aims & Objectives

- Understand sequential traversal for searching and analyze its time complexity O(n).
- Comprehend the divide-and-conquer search strategy and derive its logarithmic time complexity O(log n).
- Compare empirical performance of both algorithms and interpret result graphs.
- Develop modular code with robust handling of edge cases, such as target not present.
- Fill observation tables and reflect on the impact of input distribution and size on performance.

Theory

Linear Search: Linear search scans elements one by one until a match is found or the end is reached, exhibiting a best-case time complexity of O(1) and a worst-case of O(n) when the target is absent. Its space complexity is O(1) as it uses constant extra memory.

Binary Search: Binary search requires a sorted array and repeatedly compares the target to the middle element, discarding half of the search space at each step. The best-, average-, and worst-case time complexities of binary search are all O(log n), with space complexity O(1) for the iterative implementation.

Pseudocode

Problem 1: Linear Search

```
FUNCTION LinearSearch(A[0..n-1], key)
    FOR i ← 0 TO n-1 DO
        IF A[i] = key THEN
            RETURN i
        ENDIF
    ENDFOR
    RETURN -1
ENDFUNCTION
```

Problem 2: Binary Search

```
FUNCTION BinarySearch(A[0..n-1], key)
    low ← 0
    high ← n - 1
    WHILE low ≤ high DO
        mid ← (low + high) / 2
        IF A[mid] = key THEN
            RETURN mid
        ELSE IF A[mid] < key THEN
            low ← mid + 1
        ELSE
            high ← mid - 1
        ENDIF
    ENDWHILE
    RETURN -1
ENDFUNCTION
```

Result Graph

Students should measure execution times for each algorithm over a range of input sizes (e.g., 100, 1,000, 10,000, 100,000) and plot time (y-axis) vs. input size (x-axis), overlaying linear O(n) and binary O(log n) curves.
Typically, linear search exhibits a straight-line increase in runtime proportional to n, while binary search grows slowly in a logarithmic fashion, highlighting its scalability advantage for large datasets.

Observation Table

Use the following template to record test case details and execution metrics during the experiment:

ID	Input Size (n)	Target Value	Linear Comparisons	Binary Comparisons	Time (Linear)	Time (Binary)
1						
2						
3						

Student Task

1. **Implementation:** Implement both LinearSearch and BinarySearch functions in your chosen programming language and test them on sample datasets.

2. **Testing:** Empirically measure runtime for array sizes of at least 100, 1,000, 10,000, and 100,000, and populate the observation table.

3. **Performance Analysis:** Measure and record the execution time of both algorithms on different graph sizes and densities.

4. **Observation Table:** Complete the observation table with your findings.

5. **Result Graph:** Plot the result graph overlaying measured data points and theoretical O(n) and O(log n) curves.
6. **Discussion:** Analyze scenarios where binary search may be slower than linear search (e.g., small arrays) and document your findings.

7. **Report:** Prepare a report summarizing your implementation, findings, plots, and discussions.

8. **Viva:** A viva session will be scheduled during the next lab slot to verify individual student reports and code, and marks for the assignment will be substantially influenced by performance in that session.

Lab 03

Elementary Sorting: Bubble Sort, Insertion Sort, Selection Sort

SN	Problems	Outcome/Understanding
01.	Sort a List Using Bubble Sort Algorithm	***Bubble Sort***
02.	Sort a List using Selection Sort Algorithm	***Selection Sort***
03.	Sort a List Using Insertion Sort Algorithm	***Insertion Sort***

Aims & Objectives

- Examine the **mechanics** of elementary sorts and their in-place, comparison-based strategies, noting that all three exhibit $O(n^2)$ worst-case time complexity.
- Compare **best**, **average**, and **worst**-case behaviors: Bubble Sort ($O(n)$, $O(n^2)$, $O(n^2)$); Insertion Sort ($O(n)$, $O(n^2)$, $O(n^2)$); Selection Sort ($O(n^2)$ in all cases.
- Assess **stability** (Bubble & Insertion stable; Selection unstable) and **adaptiveness** (Insertion adapts to nearly-sorted data).
- Develop **modular pseudocode** and understand space complexity (all use $O(1)$ auxiliary space).
- Empirically measure **execution times** for array sizes (e.g., 100, 1 000, 5 000, 10 000), plot quadratic growth, and reflect on deviations due to constant factors.

Theory

Bubble Sort: Repeatedly traverses the array, comparing and swapping adjacent elements; after each pass the largest unsorted element "bubbles" to its correct position. Best case (already sorted) runs in $O(n)$ with early-exit optimization; otherwise $O(n^2)$. Space complexity $O(1)$.

Insertion Sort: Builds a sorted prefix by inserting each new element into its proper place via shifting; best case $O(n)$ for nearly sorted arrays, worst and average $O(n^2)$. Requires $O(1)$ extra space; adaptive and stable.

Selection Sort: Selects the minimum element from the unsorted subarray and swaps it with the current index; always performs $O(n^2)$ comparisons regardless of input order, but only $O(n)$ swaps. In-place but unstable by default.

Pseudocode

Bubble Sort:

```
FUNCTION BubbleSort(A[0..n-1])
    FOR i ← 0 TO n-2 DO
        FOR j ← 0 TO n-2-i DO
            IF A[j] > A[j+1] THEN
                SWAP A[j], A[j+1]
            ENDIF
        ENDFOR
    ENDFOR
ENDFUNCTION
```

Insertion Sort:

```
FUNCTION InsertionSort(A[0..n-1])
    FOR i ← 1 TO n-1 DO
        key ← A[i]
        j ← i - 1
        WHILE j ≥ 0 AND A[j] > key DO
            A[j+1] ← A[j]
            j ← j - 1
        ENDWHILE
        A[j+1] ← key
    ENDFOR
ENDFUNCTION
```

Selection Sort:

```
FUNCTION SelectionSort(A[0..n-1])
    FOR i ← 0 TO n-2 DO
        minIdx ← i
        FOR j ← i+1 TO n-1 DO
            IF A[j] < A[minIdx] THEN
                minIdx ← j
            ENDIF
        ENDFOR
        SWAP A[i], A[minIdx]
    ENDFOR
ENDFUNCTION
```

Result Graph

Plot **Execution Time** vs. **Input Size (n)** for each algorithm: Bubble Sort and Selection Sort will show near-identical quadratic curves, while Insertion Sort will be slightly lower for partially sorted data but still $O(n^2)$. Overlay theoretical $O(n^2)$ reference lines to validate empirical behavior. Include annotations for crossover points and anomalies due to cache effects or early-exit in Bubble Sort.

Observation Table

ID	Input Size (n)	Bubble Swaps	Insertion Shifts	Selection Swaps	Time (Bubble)	Time (Insertion)	Time (Selection)
1							
2							
3							

Student Tasks

1. **Implementation:** Implement BubbleSort, InsertionSort, and SelectionSort in your language of choice and verify correctness on diverse datasets.

2. **Testing:** Measure runtimes for array sizes 100, 1 000, 5 000, and 10 000, recording swaps, shifts, and execution times in the observation table.

3. **Observation Table:** Complete the observation table with your findings. Test best case (sorted) and worst case (reverse sorted) inputs; explain performance differences and stability implications.

4. **Result Graph:** Generate and annotate the result graph overlaying empirical data with $O(n^2)$ curves; discuss discrepancies and constant factors.

5. **Discussion:** Analyze scenarios where binary search may be slower than linear search (e.g., small arrays) and document your findings.

6. **Report:** Prepare a report summarizing your implementation, findings, plots, and discussions.

7. **Viva:** A viva session will be scheduled during the next lab slot to verify individual student reports and code, and marks for the assignment will be substantially influenced by performance in that session.

Lab 04

Divide and Conquer: Quick-Sort & Merge-Sort

SN	Problems	Outcome/Understanding
01.	Implement Quick Sort to Sort an Array	***Quick Sort***
02.	Implement Merge Sort to Sort an Array	***Merge Sort***

Aims & Objectives

- Understand the **pivot selection** and **partitioning** strategy in Quick Sort, including common schemes (first, last, median).
- Comprehend Merge Sort's **stable** merging process and its requirement for auxiliary space O(n).
- Derive and compare **best-, average-, and worst-case** time complexities: Quick Sort (O(n log n) avg, O(n^2) worst) and Merge Sort (O(n log n) all cases).
- Measure **space complexity**, noting Quick Sort's O(log n) recursion depth versus Merge Sort's O(n) auxiliary space.
- Empirically evaluate performance on random, sorted, and reverse-sorted datasets, and plot **execution time vs. input size** curves.

Theory

Quick Sort:
Quick Sort selects a pivot, partitions the array into elements less than and greater than the pivot, and then recursively sorts the partitions. Its average-case time complexity is O(n log n) with space complexity O(log n) due to recursion, but it degrades to O(n^2) when pivots are poorly chosen.

Merge Sort:
Merge Sort divides the array into two halves, recursively sorts them, and merges the sorted halves in O(n) time per merge step. It guarantees O(n log n) time complexity in all cases, with O(n) auxiliary space for merging. Merge Sort is stable, preserving the order of equal elements.

Pseudocode

Quick Sort

```
FUNCTION QuickSort(A[low..high])
    IF low < high THEN
        pivotIndex ← Partition(A, low, high)
        QuickSort(A, low, pivotIndex - 1)
        QuickSort(A, pivotIndex + 1, high)
    ENDIF
ENDFUNCTION
```

```
FUNCTION Partition(A, low, high)
    pivot ← A[high]
    i ← low - 1
    FOR j ← low TO high - 1 DO
        IF A[j] ≤ pivot THEN
            i ← i + 1
            SWAP A[i], A[j]
        ENDIF
    ENDFOR
    SWAP A[i + 1], A[high]
    RETURN i + 1
ENDFUNCTION
```

Merge Sort

```
FUNCTION MergeSort(A[low..high])
    IF low < high THEN
        mid ← ⌊(low + high) / 2⌋
        MergeSort(A, low, mid)
        MergeSort(A, mid + 1, high)
        Merge(A, low, mid, high)
    ENDIF
ENDFUNCTION

FUNCTION Merge(A, low, mid, high)
    n1 ← mid - low + 1
    n2 ← high - mid
    create arrays L[1..n1], R[1..n2]
    copy A[low..mid] → L and A[mid+1..high] → R
    i ← 1; j ← 1; k ← low
    WHILE i ≤ n1 AND j ≤ n2 DO
        IF L[i] ≤ R[j] THEN
            A[k] ← L[i]; i ← i + 1
        ELSE
            A[k] ← R[j]; j ← j + 1
        ENDIF
        k ← k + 1
    ENDWHILE
    copy remaining elements of L and R, if any
ENDFUNCTION
```

Result Graph

Students should measure execution time for input sizes (e.g., n=1 000, 10 000, 100 000) on random, sorted, and reverse-sorted arrays, then plot **time vs. n** for both algorithms. The graph will show Merge Sort's consistent O(n log n) growth and Quick Sort's average O(n log n) with potential worst-case spikes. Annotate pivot selection impacts and memory overhead effects during merging.

Observation Table

ID	Input Type	Size n	Quick Comparisons	Merge Comparisons	Time (Quick)	Time (Merge	Aux Space
1	Random						
2	Already Sorted						
3	Reverse Sorted						

Student Tasks

1. **Implementation:** Implement Quick Sort and Merge Sort in your chosen language, ensuring correct pivot handling and in-place partitioning.

2. **Testing:** Empirically measure runtime and comparison counts for each algorithm on random, sorted, and reverse-sorted datasets of at least 1 000, 10 000, and 100 000 elements.

3. **Observation Table:** Complete the observation table with your findings.

4. **Result Graph:** Plot execution time vs. n curves overlaid with theoretical O(n log n) reference lines and annotate deviations due to worst-case Quick Sort behavior.

5. **Discussion:** Analyze how pivot selection schemes (first, last, median-of-three, random) affect Quick Sort performance and document results in an addendum.

6. **Report:** Prepare a report summarizing your implementation, findings, plots, and discussions.

7. **Viva:** A viva session will be scheduled during the next lab slot to verify individual student reports and code, and marks for the assignment will be substantially influenced by performance in that session.

Lab 05

Greedy Design: Fractional Knapsack, Activity Scheduling.

SN	Problems	Outcome/Understanding
01.	Solve Fractional Knapsack problem by Implementing Greedy Design	***Greedy Design***
02.	Solve Activity Scheduling problem by Implementing Greedy Design	***same***

Problem Statements

- **Problem 1: Fractional Knapsack.** Given n items each with profit p_i and weight w_i, and a knapsack capacity W, maximize total profit by selecting items or fractions thereof.
- **Problem 2: Activity Scheduling.** Given a set of n activities each with a start time s_i and finish time f_i, select the largest subset of non-overlapping activities.

Aims & Objectives

- **Illustrate the greedy choice property** by selecting locally optimal options that lead to global optima in Fractional Knapsack and Activity Scheduling.
- **Implement sorting-based strategies**, understanding why sorting by ratio or finish time underpins the algorithms' correctness.
- **Analyze time and space complexities**, deriving O(n log n) for both problems and O(n) additional space for knapsack's fractional array.
- **Develop clear, language-agnostic pseudocode**, emphasizing modularity and clarity in control structures.
- **Experimentally measure performance**, record results in tables, and plot graphs to contrast theoretical predictions with empirical data.

Theory

Greedy algorithms build solutions by making the best local choice at each step without revisiting prior decisions.

Fractional Knapsack: In the fractional knapsack problem, items can be broken into smaller parts. The goal is to maximize the total value in the knapsack by taking fractions of items with the highest value-to-weight ratio. Sort items by decreasing profit/weight ratio, then iterate, adding full items while capacity remains, and a final fractional addition if needed. The dominant cost is sorting n items: O(n log n), plus a single O(n) pass.

Activity Scheduling: The activity scheduling problem involves selecting the maximum number of non-overlapping activities from a set. Sort activities by increasing finish times, then scan, selecting each activity that starts after the last selected finish. Sorting dominates with O(n log n), and selection is O(n).

Pseudocode

Problem 1: Fractional Knapsack

```
FUNCTION FractionalKnapsack(items, n, W)
    SORT items in descending order of (profit/weight)
    capacity ← W; totalProfit ← 0
    FOR i ← 1 TO n DO
        IF items[i].weight ≤ capacity THEN
            capacity ← capacity - items[i].weight
            totalProfit ← totalProfit + items[i].profit
        ELSE
            fraction ← capacity / items[i].weight
            totalProfit ← totalProfit + items[i].profit *
fraction
            BREAK
        ENDIF
    ENDFOR
    RETURN totalProfit
ENDFUNCTION
```

Problem 2: Activity Scheduling

```
FUNCTION ActivitySelection(s[1..n], f[1..n], n)
    CREATE list A of activity indices 1..n
    SORT A by increasing f[i]
    selected ← [A[1]]
    lastFinish ← f[A[1]]
    FOR k ← 2 TO n DO
        i ← A[k]
        IF s[i] ≥ lastFinish THEN
            APPEND i to selected
            lastFinish ← f[i]
        ENDIF
    ENDFOR
    RETURN selected
ENDFUNCTION
```

Result Graph

Measure execution time for varying nn (e.g., 1 000; 10 000; 100 000) and plot **Time (y-axis)** vs. **Input Size n (x-axis)** for both algorithms. The graph should overlay empirical points with theoretical O(n log n) curves, illustrating sorting dominance and linear pass effects. Annotate where overheads (e.g., fraction calculations or list appends) cause deviations.

Observation Table

Use this template to record runtime metrics and solution quality:

ID	n (items/activities)	Capacity W (knapsack)	Profit/Weight Sorting Time	Greedy Pass Time	Total Time	Total Profit	# Activities Selected
1							
2							
3							

Student Tasks

1. **Implementation:** Implement the Fractional Knapsack and Activity Scheduling functions in your preferred language, ensuring correct sorting and greedy selection.
2. **Testing:** Empirically measure sorting time, greedy pass time, and total solution metrics for n=1 000,10 000,100 000 and knapsack capacities that trigger fractional additions.
3. **Observation Table:** Complete the observation table with your findings.
4. **Result Graph:** Plot the result graph overlaying empirical runtimes with O(n log n) theoretical curves and analyze any discrepancies and constant-factor effects.
5. **Discussion:** Evaluate solution optimality: verify that the knapsack solution equals the LP relaxation's bound, and that the activity count matches the known maximum for given test sets.
6. **Report:** Prepare a report summarizing your implementation, findings, plots, and discussions.
7. **Viva:** A viva session will be scheduled during the next lab slot to verify individual student reports and code, and marks for the assignment will be substantially influenced by performance in that session.

Lab 06

Dynamic Programming: Elementary – Coin Toss, 0/1 Knap Sack

SN	Problems	Outcome/Understanding
01.	Solve Coin Toss Problem by Using Dynamic Programming	***Dynamic Programming***
02.	Solve 0/1 Knapsack Problem by Using Dynamic Programming	***Dynamic Programming***

Problem Statements

- **Problem 1:** Count the number of ways to get exactly kk heads in n fair coin tosses using DP to compute $P(n,k) = P(n-1,k) + P(n-1,k-1)$ with base cases $P(i,0)=1$, $P(i,i)=1$.
- **Problem 2:** Solve the **0/1 Knapsack** problem: given n items with weights w_i and profits p_i and capacity W, maximize total profit by choosing items subject to weight ≤ W via bottom-up DP table of size $(n+1)\times(W+1)$.

Aims & Objectives

- Demonstrate DP implementation to exploit **optimal substructure** and **overlapping** subproblems in coin-toss and knapsack contexts.
- Construct and populate **DP tables** iteratively (tabulation), understanding transitions and base cases.
- Analyze **time complexity**: $O(n_k)$ for coin toss DP and $O(n_W)$for 0/1 Knapsack DP.
- Evaluate **space complexity**: $O(n_k)$ and $O(n_W)$ table sizes and discuss techniques to reduce to one-dimensional arrays.
- Empirically measure runtime for varying n, k, W, compare with theoretical predictions, and plot **Result Graphs**.

Theory

DP transforms recursive relations into table-driven computations to avoid redundant calls.
Coin Toss DP:
This problem involves determining the number of ways to achieve a certain outcome (e.g., a specific number of heads) when tossing coins. Dynamic programming efficiently computes the solution by storing intermediate results to avoid redundant calculations. The recurrence for the number of ways to obtain **k** heads in **n** tosses is
$P(n,k)=P(n-1,k)+P(n-1,k-1)$
with base values $P(i,0)=1P(i,0)=1$ and $P(i,i)=1P(i,i)=1$ for $0 \le i \le n$.

0/1 Knapsack DP:
In the 0/1 knapsack problem, each item can be included (1) or excluded (0). The goal is to maximize the total value without exceeding the knapsack's capacity. Dynamic programming solves this by building a table that considers each item's inclusion or exclusion. Define dp[i][j] as maximum profit using first ii items with capacity jj.
Recurrence:
$dp[i][j]=\max(dp[i-1][j], pi+dp[i-1][j-wi])$
for $j \geq w_i$, else $dp[i][j]=dp[i-1][j]$, with $dp[0][*]=0, dp[*][0]=0$.

Pseudocode

Problem 1: Coin Toss DP

```
INPUT: n, k
DECLARE integer P[0..n][0..k]
FOR i ← 0 TO n DO
    FOR j ← 0 TO min(i,k) DO
        IF j = 0 OR j = i THEN
            P[i][j] ← 1
        ELSE
            P[i][j] ← P[i-1][j] + P[i-1][j-1]
        ENDIF
    ENDFOR
ENDFOR
OUTPUT: P[n][k]
```

Problem 2: 0/1 Knapsack DP

```
INPUT: n, W, profits p[1..n], weights w[1..n]
DECLARE integer dp[0..n][0..W]
FOR j ← 0 TO W DO
    dp[0][j] ← 0
ENDFOR
FOR i ← 1 TO n DO
    FOR j ← 0 TO W DO
        IF w[i] ≤ j THEN
            dp[i][j] ← max(dp[i-1][j], p[i] + dp[i-1][j-
w[i]])
        ELSE
            dp[i][j] ← dp[i-1][j]
        ENDIF
    ENDFOR
ENDFOR
OUTPUT: dp[n][W]
```

Result Graph

Students should empirically measure execution **time** for Coin Toss DP over n=100,200,500,1000 with k=[n/2], and for 0/1 Knapsack DP varying W=100,200,500,1000 with fixed n, then plot **Time vs. Problem Size** curves to illustrate polynomial (quadratic) growth.

Observation Table

Problem 1: Coin Toss

ID	n	k/W	DP Table Size	Time (ms)	Memory (MB)
1					
2					
3					

Problem 2: Knapsack

ID	n	k/W	DP Table Size	Time (ms)	Memory (MB)
1					
2					
3					

Student Tasks

1. **Implementation:** Implement the Coin Toss DP and 0/1 Knapsack DP algorithms in your preferred language, ensuring correct table initialization and recurrence transitions.
2. **Testing:** Empirically measure runtimes and memory usage for specified n, k, W values using high-resolution timers or profiling tools.
3. **Observation Table:** Complete the observation table with your findings.
4. **Result Graph:** Plot the Result Graphs overlaying empirical data points with theoretical $O(n_k)$ and $O(n_W)$curves; annotate any deviations due to constant factors or hardware limits.
5. **Discussion:** Analyze and discuss how increasing k (for coin toss) or W (for knapsack) affects performance and resource usage, relating back to theoretical complexities.
6. **Report:** Prepare a report summarizing your implementation, findings, plots, and discussions.
7. **Viva:** A viva session will be scheduled during the next lab slot to verify individual student reports and code, and marks for the assignment will be substantially influenced by performance in that session.

Assessment Test 1:
Based on Lab 2- 6: Searching, Sorting & DP

Lab 08

Dynamic Programming: LCS, Rod Cutting

SN	Problems	Outcome/Understanding
01.	Find The Longest Common Sub-Sequence from Two Given Sequence Using Dynamic Programming	***Dynamic Programming***
02.	Solve The Rod Cutting Problem by Implementing Dynamic Programming	***Same***

Problem Statements

- **Problem 1: Longest Common Subsequence (LCS).** Given strings X of length m and Y of length n, find the length of the longest subsequence common to both using DP table L[0..m][0..n].
- **Problem 2: Rod Cutting.** Given a rod of length n and price array P[1..n], determine the maximum profit by cutting the rod into integer lengths via DP table dp[0..n] using the recurrence $dp[i]=\max(P[j]+dp[i-j-1])$ for $0 \leq j < i$.

Aims & Objectives

- Illustrate **optimal substructure** and **overlapping subproblems** in combinatorial optimization.
- Construct **tabulation tables** for LCS (size $(m+1)\times(n+1)$) and Rod Cutting (size $n+1$) bottom-up.
- Analyze **time complexity** $O(m\times n)$ for LCS and $O(n^2)$ for Rod Cutting, and **space complexity** of DP tables.
- Empirically measure **runtime** for varying input sizes and validate against theoretical predictions.

Theory

Longest Common Subsequence (LCS):
The LCS problem seeks the longest sequence common to two sequences. Dynamic programming solves this by constructing a matrix that records the lengths of common subsequences at different points, enabling efficient computation. Recurrence:

$$L[i][j] = \begin{cases} 0, & \text{if } i=0 \text{ or } j=0; \\ L[i-1][j-1]+1, & \text{if } X[i-1]=Y[j-1]; \\ \max(L[i-1][j], L[i][j-1]), & \text{otherwise.} \end{cases}$$

This yields a fill cost of $O(m\times n)$ and reconstructs sequence with backtracking.

Rod Cutting Recurrence:
The rod cutting problem aims to determine the optimal way to cut a rod to maximize profit, given prices for different lengths. Dynamic programming approaches this by solving smaller subproblems and building up to the solution for the full rod length.
$dp[i] = \max_{0 \le j < i} (P[j+1] + dp[i-j-1])$, with $dp[0]=0$, i from 1 to n, yielding $O(n^2)$ due to nested loops.

Pseudocode

Problem 1: LCS Tabulation

```
FUNCTION LCS(X[1..m], Y[1..n])
    CREATE L[0..m][0..n]
    FOR i ← 0 TO m DO
        L[i][0] ← 0
    ENDFOR
    FOR j ← 0 TO n DO
        L[0][j] ← 0
    ENDFOR
    FOR i ← 1 TO m DO
        FOR j ← 1 TO n DO
            IF X[i] = Y[j] THEN
                L[i][j] ← L[i-1][j-1] + 1
            ELSE
                L[i][j] ← max(L[i-1][j], L[i][j-1])
            ENDIF
        ENDFOR
    ENDFOR
    RETURN L[m][n]
ENDFUNCTION
```

Problem 2: Rod Cutting

```
FUNCTION RodCutting(P[1..n], n)
    DECLARE dp[0..n]
    dp[0] ← 0
    FOR i ← 1 TO n DO
        maxVal ← 0
        FOR j ← 0 TO i-1 DO
            maxVal ← max(maxVal, P[j+1] + dp[i-j-1])
        ENDFOR
        dp[i] ← maxVal
    ENDFOR
    RETURN dp[n]
ENDFUNCTION
```

Result Graph

Measure execution **time** versus **input sizes**: for LCS vary m,n pairs (e.g., m=n=200, 500, 1000); for Rod Cutting vary n (e.g., 500, 1000, 2000). Plot empirical points and overlay theoretical $O(m \times n)$ and $O(n^2)$ curves to demonstrate polynomial scaling and validate algorithmic efficiency.

Observation Table

ID	Problem	m, n, or n	DP Table Size	Time (ms)	Memory (MB)
1	LCS	200×200	201×201		
2	LCS	500×500	501×501		
3	Rod Cut	n=1000	1001		

Student Tasks

1. **Implementation:** Implement LCS and Rod Cutting DP in your chosen language, ensuring accurate table initialization and recurrence logic.
2. **Testing:** Empirically measure runtime and memory metrics for specified input sizes.
3. **Observation Table:** Complete the observation table with your findings.
4. **Result Graph:** Plot the Result Graphs overlaying empirical data with theoretical $O(m \times n)$ and $O(n^2)$ trendlines, annotating any anomalies due to cache or language overhead.
5. **Discussion:** Analyze and discuss how dynamic programming optimizes the solutions for LCS and Rod Cutting problems.
6. **Report:** Prepare a report summarizing your implementation, findings, plots, and discussions.
7. **Viva:** A viva session will be scheduled during the next lab slot to verify individual student reports and code, and marks for the assignment will be substantially influenced by performance in that session.

Lab 09

Graph Representations: Adjacency List, Adjacency Matrix, Weighted Matrix, Self Loops

SN	Problems	Outcome/Understanding
01.	Given a Graph with Vertices and Edges, Implement Its Representation Using an Adjacency List	***Graph Representations***
02.	Represent The same Graph as an Adjacency Matrix	***Same***
03.	Extend The Matrix to a Weighted Adjacency Matrix, Storing Edge Weights	***Same***
04.	Detect and Record Self-Loops in Each Representation	***Same***

Problem Statements

- **Problem 1:** Given a graph with V vertices and E edges, implement its representation using an **Adjacency List**.
- **Problem 2:** Represent the same graph as an **Adjacency Matrix**, a V×V boolean matrix indicating edge presence.
- **Problem 3:** Extend the matrix to a **Weighted Adjacency Matrix**, storing edge weights or ∞ for no edge.
- **Problem 4:** Detect and record **self-loops** (edges (v,v)) in each representation.

Aims & Objectives

- **Compare memory usage:** Adjacency List uses O(E) space versus Adjacency Matrix's $O(V^2)$ bits/entries.
- **Measure build time:** Analyze time to construct each representation—O(V+E) for lists, $O(V^2)$ for matrices.
- **Evaluate query performance:** Edge-existence queries run in O(1) for matrices and O(degree(v)) for lists.
- **Understand weighted graphs:** Store and query edge weights in a matrix, handling infinity or sentinel for non-edges.
- **Implement self-loop handling:** Recognize loops in representations and analyze their impact on degree calculations.

Theory

Adjacency List: An adjacency list represents a graph by maintaining a list of adjacent vertices for each vertex. It's efficient for sparse graphs, offering a space complexity of O(V + E), where V is the number of vertices and E is the number of edges. Each vertex maintains a list of adjacent vertices, enabling O(V+E) storage and efficient neighbor iteration.

Adjacency Matrix: An adjacency matrix uses a 2D array to represent a graph, where each cell (i, j) indicates the presence (and possibly the weight) of an edge between vertices i and j. It's suitable for dense graphs but consumes $O(V^2)$ space. A V×V boolean (or numeric) matrix where entry A[i][j]=1 if an edge exists, else 0; uses $O(V^2)$ space.

Weighted Adjacency Matrix: In a weighted matrix, the adjacency matrix stores the actual weights of edges instead of just binary indicators. This allows for representing weighted graphs, where the value at cell (i, j) denotes the weight of the edge from vertex i to vertex j. It extends the matrix to store edge weights w_{ij} or ∞/0 for no edge; supports O(1) weight lookup.

Self-Loops: A self-loop is an edge that connects a vertex to itself. In an adjacency matrix, this is represented by a non-zero value on the diagonal (i, i). Self-loops can affect algorithms like DFS and BFS by introducing cycles. Edges from a vertex to itself are represented as nonzero diagonal entries in matrices and self-references in lists; they increase vertex degree by 2 in undirected graphs.

Pseudocode

Build Adjacency List:

```
FUNCTION BuildAdjMatrix(V, edgeList)
    CREATE matrix A[0..V-1][0..V-1] initialized to 0
    FOR each (u,v) in edgeList DO
        A[u][v] ← 1
        A[v][u] ← 1  // if undirected
    ENDFOR
    RETURN A
ENDFUNCTION
```

Build Weighted Matrix

```
FUNCTION FindSelfLoops(adjList or matrix, V)
    loops ← empty list
    FOR v ← 0 TO V-1 DO
        IF matrix[v][v] ≠ 0 OR ∞ then APPEND v to loops
        // or if adjList[v] contains v
    ENDFOR
    RETURN loops
ENDFUNCTION
```

Result Graph

Plot **Memory (y-axis)** vs. **V (x-axis)** for each representation, demonstrating linear versus quadratic growth.
Plot **Build Time** and **Edge Query Time** versus **V** and **E**, highlighting adjacency list's O(V+E) build and matrix's $O(V^2)$ behavior.

Observation Table

ID	Rep. Type	V	E	Memory Usage	Query Time
1	Adjacency List				
2	Adjacency Matrix				
3	Weighted Matrix				
4	List w/ Self-Loops				

Student Tasks

1. **Implementation:** Implement each graph representation and verify correctness on sample graphs.
2. **Testing:** Empirically measure memory footprint (bytes or entries) and build/query times using timers/profilers for increasing V and E.
3. **Observation Table:** Complete the observation table with your findings.
4. **Result Graph:** Plot the Result Graphs, annotate crossover points where lists outperform matrices and vice versa.
5. **Discussion:** Analyze and discuss each of the graph representation techniques and how they overlap.
6. **Report:** Prepare a report summarizing your implementation, findings, plots, and discussions.
7. **Viva:** A viva session will be scheduled during the next lab slot to verify individual student reports and code, and marks for the assignment will be substantially influenced by performance in that session.

Lab 10

Graph Coding: Searching- BFS, DFS

SN	Problems	Outcome/Understanding
01.	Implement Breath First Search to Traverse All Reachable Vertices from a Given Source in an Undirected Graph	***Graph Search (BFS)***
02.	Implement Depth First Search to Traverse All Reachable Vertices from a Given Source in an Undirected Graph	***Graph Search (DFS)***

Problem Statements

- **Problem 1:** Implement **Breadth-First Search** to traverse all reachable vertices from a given source in an undirected graph, returning the visitation order.
- **Problem 2:** Implement **Depth-First Search** to traverse all reachable vertices from a given source in an undirected graph, returning the visitation order.

Aims & Objectives

- Understand and implement the **queue-based** approach of BFS and the **stack/recursive** approach of DFS.
- Analyze and confirm the **time complexity** of O(V + E) and **space complexity** of O(V) for both algorithms.
- Compare **traversal orders** produced by BFS vs. DFS on various graph structures (trees, cycles, disconnected graphs).
- Measure and plot **execution time** as graph size (vertices V and edges E) increases to validate theoretical behavior.
- Examine memory usage and stack/queue depth under different graph densities.

Theory

Breadth-First Search (BFS) BFS is a traversal algorithm that explores vertices in the order of their distance from the starting point, using a queue to manage the exploration frontier. It's particularly useful for finding the shortest path in unweighted graphs. It uses a FIFO queue to explore neighbors layer by layer, ensuring the shortest-path tree in unweighted graphs.

Depth-First Search (DFS) DFS explores as far as possible along each branch before backtracking, typically implemented using recursion or a stack. It's effective for tasks like detecting cycles and performing topological sorts. It uses LIFO recursion or a stack to explore as far as possible along each branch before backtracking.

Both algorithms visit each vertex once and examine each edge at most twice (once per direction in undirected graphs), yielding O(V + E) time.

Pseudocode

Breadth-First Search

```
FUNCTION BFS(graph, source)
    CREATE array visited[0..V-1] initialized to FALSE
    CREATE queue Q
    CREATE list order
    visited[source] ← TRUE
    ENQUEUE(Q, source)
    WHILE Q is not empty DO
        u ← DEQUEUE(Q)
        APPEND u to order
        FOR each v in graph.adj[u] DO
            IF NOT visited[v] THEN
                visited[v] ← TRUE
                ENQUEUE(Q, v)
            ENDIF
        ENDFOR
    ENDWHILE
    RETURN order
ENDFUNCTION
```

Depth-First Search

```
FUNCTION DFS(graph, source)
    CREATE array visited[0..V-1] initialized to FALSE
    CREATE list order
    CALL DFS-Visit(source)
    RETURN order

FUNCTION DFS-Visit(u)
    visited[u] ← TRUE
    APPEND u to order
    FOR each v in graph.adj[u] DO
        IF NOT visited[v] THEN
            CALL DFS-Visit(v)
        ENDIF
    ENDFOR
ENDFUNCTION
```

Result Graph

Measure and plot **execution time (y-axis)** vs. **graph size parameters**—number of vertices V (e.g., 1 000; 5 000; 10 000) and average degree E/V—overlaying BFS and DFS curves. Both should scale linearly, but queue management overhead may show slight constant-factor differences. Annotate any divergences due to graph density or memory/cache effects.

Observation Table

ID	V	E	BFS Time (ms)	DFS Time (ms)	Max Queue/Stack Depth	Memory (MB)
1	100	200				
2	1 000	5 000				
3	5 000	50 000				

Student Tasks

1. **Implementation:** Implement both BFS and DFS using adjacency lists and test on directed and undirected graphs of varying sizes.
2. **Testing:** Empirically measure execution times, memory usage, and maximum queue/stack depth using high resolution timers or profilers.
3. **Observation Table:** Complete the observation table with your findings.
4. **Result Graph:** Plot the Result Graphs overlaying empirical data with theoretical O(V + E) trendlines; discuss any constant factor differences and anomalies.
5. **Discussion:** Compare traversal orders of BFS vs. DFS on sample graphs (e.g., binary tree, cyclic graph, disconnected graph) and explain differences in exploration strategy.
6. **Report:** Prepare a report summarizing your implementation, findings, plots, and discussions.
7. **Viva:** A viva session will be scheduled during the next lab slot to verify individual student reports and code, and marks for the assignment will be substantially influenced by performance in that session.

Lab 11

Graph Coding: MST – Prims, Kruskal

SN	Problems	Outcome/Understanding
01.	Implement Prim's Algorithm to find the Minimum Spanning Tree (MST) of a connected, undirected, weighted graph.	***Spanning Tree (Prim's Algorithm)***
02.	Implement Kruskal's Algorithm to find the MST of a connected, undirected, weighted graph.	***Same (Kruskal's Algorithm)***

Aims & Objectives

- Understand and implement Prim's and Kruskal's algorithms for finding MSTs.
- Analyze and compare the time and space complexities of both algorithms.
- Evaluate the performance of both algorithms on various graph structures and sizes.

Theory

Prim's Algorithm
Prim's algorithm constructs an MST by starting from an arbitrary vertex and repeatedly adding the smallest edge that connects a vertex in the MST to a vertex outside it. It's efficient for dense graphs when implemented with a priority queue.

- **Approach:** Starts with a single vertex and grows the MST by adding the smallest edge connecting the tree to a vertex not yet in the tree.
- **Time Complexity:**
    - Using an adjacency matrix: $O(V^2)$
    - Using an adjacency list with a binary heap: $O((V + E) \log V)$
- **Space Complexity:** $O(V + E)$

Kruskal's Algorithm
Kruskal's algorithm builds an MST by sorting all edges in increasing order of weight and adding them one by one, ensuring that no cycles are formed. It uses a disjoint-set data structure to detect cycles and is effective for sparse graphs.

- **Approach:** Sorts all edges in non-decreasing order of weight and adds them one by one to the MST, ensuring that no cycles are formed.
- **Time Complexity:** $O(E \log E)$
- **Space Complexity:** $O(V + E)$

Pseudocode

Prim's Algorithm

```
FUNCTION PrimMST(Graph):
    // Graph.Vertices = set of vertices; Graph.Adj[u] lists
(v, weight)
    for each vertex u in Graph.Vertices do
        key[u] ← ∞
        parent[u] ← NIL
    end for
    // Start from an arbitrary source (e.g., the first
vertex)
    key[source] ← 0
    Q ← min-priority-queue containing all vertices, keyed by
key[.]

    while Q is not empty do
        u ← EXTRACT_MIN(Q)
        for each (v, w) in Graph.Adj[u] do
            if v is in Q AND w < key[v] then
                parent[v] ← u
                key[v] ← w
                DECREASE_KEY(Q, v, w)
            end if
        end for
    end while

    return parent   // defines the MST edges (v, parent[v])
ENDFUNCTION
```

Kruskal's Algorithm

```
FUNCTION KruskalMST(Graph):
    // Graph.Edges = list of (u, v, weight)
    sort Graph.Edges in non-decreasing order by weight
    for each vertex u in Graph.Vertices do
        MAKE_SET(u)      // initializes disjoint sets
    end for

    MST ← empty set
    for each (u, v, w) in Graph.Edges do
        if FIND_SET(u) ≠ FIND_SET(v) then
            MST.add((u, v, w))
            UNION(u, v)
        end if
        if size(MST) = |Graph.Vertices| − 1 then
            break
        end if
    end for

    return MST // set of edges in the minimum spanning tree
ENDFUNCTION
```

Result Graph

Plot the execution time (in milliseconds) of Prim's and Kruskal's algorithms against varying numbers of vertices (V) and edges (E). Use different graph densities (sparse and dense) to observe performance differences. The x-axis represents the number of vertices, and the y-axis represents the execution time. Use separate lines for each algorithm to compare their scalability and efficiency.

Observation Table

ID	Vertices (V)	Edges (E)	Graph Type	Prim's Time (ms)	Kruskal's Time (ms)
1	100	500	Sparse		
2	100	1000	Dense		
3	1000	5000	Sparse		
4	1000	10000	Dense		
5	5000	20000	Sparse		

Student Tasks

1. **Implementation:** Write code for both Prim's and Kruskal's algorithms in your preferred programming language.

2. **Testing:** Create various graph instances with different sizes and densities to test the algorithms.

3. **Observation Table:** Complete the observation table with your findings. Measure and record the execution time of both algorithms on different graph sizes and densities

4. **Result Graph:** Plot the execution times to visualize and compare the performance of the algorithms.

5. **Discussion:** Analyze the results, discussing the scenarios where each algorithm performs better and the limitations of each.

6. **Report:** Prepare a report summarizing your implementation, findings, plots, and discussions.

7. **Viva:** A viva session will be scheduled during the next lab slot to verify individual student reports and code, and marks for the assignment will be substantially influenced by performance in that session.

Lab 12

Graph Coding: Shortest Path- Dijkstra, Bell-Man Ford

SN	Problems	Outcome/Understanding
03.	Implement Dijkstra's Algorithm to compute the shortest paths from a single source vertex to all other vertices in a graph with non-negative edge weights.	***Shortest Path Calculation***
04.	Implement Bellman-Ford Algorithm to compute the shortest paths from a single source vertex to all other vertices in a graph that may contain negative edge weights, and detect negative weight cycles if present.	***Same***

Aims & Objectives

- Understand and implement Dijkstra's and Bellman-Ford algorithms for single-source shortest path problems.
- Analyze and compare the time and space complexities of both algorithms.
- Demonstrate the ability of Bellman-Ford to handle graphs with negative edge weights and detect negative cycles.
- Evaluate the performance of both algorithms on various graph structures and sizes.

Theory

Dijkstra's Algorithm
Dijkstra's algorithm finds the shortest paths from a source vertex to all other vertices in a graph with non-negative edge weights. It uses a priority queue to select the next vertex with the minimal tentative distance.

- **Approach:** Greedy algorithm that selects the unvisited vertex with the smallest tentative distance, updating the distances of its neighbors.
- **Time Complexity:**
    - Using a simple array: $O(V^2)$
    - Using a min-priority queue with a binary heap: $O((V + E) \log V)$
- **Space Complexity:** $O(V)$
- **Limitation:** Does not work with graphs containing negative edge weights.

Bellman-Ford Algorithm
The Bellman-Ford algorithm computes shortest paths from a single source vertex to all other vertices in a weighted graph, even accommodating negative edge weights. It iteratively relaxes edges and can detect negative weight cycles.

- **Approach:** Dynamic programming algorithm that relaxes all edges V-1 times, where V is the number of vertices.
- **Time Complexity:** O(V * E)
- **Space Complexity:** O(V)
- **Advantage:** Can handle graphs with negative edge weights and detect negative weight cycles.

Pseudocode

Dijkstra's Algorithm

```
FUNCTION Dijkstra(Graph, source):
    // Initialization
    for each vertex v in Graph.Vertices do
        dist[v] ← ∞
        prev[v] ← UNDEFINED
    end for
    dist[source] ← 0
    Q ← min-priority-queue containing all vertices with key
= dist[v]

    // Main loop
    while Q is not empty do
        u ← EXTRACT_MIN(Q)  // vertex with smallest dist[u]
        for each neighbor v of u do
            alt ← dist[u] + weight(u, v)
            if alt < dist[v] then
                dist[v] ← alt
                prev[v] ← u
                DECREASE_KEY(Q, v, alt)
            end if
        end for
    end while

    return dist, prev
ENDFUNCTION
```

Bellman-Ford Algorithm

```
FUNCTION BellmanFord(Graph, source):
    // Initialization
    for each vertex v in Graph.Vertices do
        dist[v] ← ∞
        prev[v] ← UNDEFINED
    end for
    dist[source] ← 0

    // Relax edges repeatedly
    for i from 1 to |V| - 1 do
        for each edge (u, v) with weight w in Graph.Edges do
            if dist[u] + w < dist[v] then
                dist[v] ← dist[u] + w
                prev[v] ← u
            end if
        end for
    end for
    // Check for negative-weight cycles
    for each edge (u, v) with weight w in Graph.Edges do
        if dist[u] + w < dist[v] then
            ERROR "Graph contains a negative-weight cycle"
        end if
    end for

    return dist, prev
ENDFUNCTION
```

Result Graph

Plot the execution time (in milliseconds) of Dijkstra's and Bellman-Ford algorithms against varying numbers of vertices (V) and edges (E). Use different graph densities (sparse and dense) to observe performance differences. The x-axis represents the number of vertices, and the y-axis represents the execution time. Use separate lines for each algorithm to compare their scalability and efficiency.

Observation Table

ID	Vertices (V)	Edges (E)	Graph Type	Dijkstra Time (ms)	Bellman-Ford Time (ms)	Negative Cycle Present
1	100	500	Sparse, positive			No
2	100	1000	Dense, positive			No
3	100	500	Sparse, negative	N/A		No
4	100	500	Sparse, negative	N/A	Ye	
5	1000	5000	Sparse, positive		No	

Note: Fill in the execution times after running the algorithms on the specified graph types.

Student Tasks

1. **Implementation:** Write code for both Dijkstra's and Bellman-Ford algorithms in your preferred programming language.
2. **Testing:** Create various graph instances, including graphs with:
    - Only positive edge weights
    - Negative edge weights without negative cycles
    - Negative edge weights with negative cycles.
3. **Observation Table:** Complete the observation table with your findings. Measure and record the execution time of both algorithms on different graph sizes and densities
4. **Result Graph:** Plot the execution times to visualize and compare the performance of the algorithms.
5. **Discussion:** Analyze the results, discussing the scenarios where each algorithm performs better and the limitations of each.
6. **Report:** Prepare a report summarizing your implementation, findings, plots, and discussions.
7. **Viva:** A viva session will be scheduled during the next lab slot to verify individual student reports and code, and marks for the assignment will be substantially influenced by performance in that session.

Assessment Test 2:
Based on Lab 02 - 12

Lab Final : 15 Points

Topics: Dynamic Programming, Shortest Path, Spanning Tree and Graph Searching.

Assessment Test 3:
Based on lab work

Quiz & Viva