**ACIT 4610**

**Final Group Project -2025**

**Purpose:** Implement and study the application of algorithms/techniques discussed in this course.

The project will be carried out in **groups (2-4 students)**.

**Submission Deadline:** November 21 (Friday) at 12:00 (mid-day).

**Submission Channel:** Inspera.

- Each group must submit **ONLY ONE final version**, submitted by any member of the group.
- The Inspera submission will be available approximately one week prior to the deadline.

**Cheating / Plagiarism:** Any cheating/plagiarism will be handled by OsloMet's policy. You can find more about it at https://student.oslomet.no/en/cheating

## 1 Assignment

There are **Four (05)** problems in this Portfolio. You need to implement any **Four (04)** of these problems as per the requirements and instructions mentioned in each problem.

## 2 Submission

You should submit your report, including **the GitHub link** with the code, datasets, and a README file containing the necessary instructions to run and test your implementation. Note that we will download the code immediately after the submission deadline to prevent any later updates. **You need to ensure I have access to the GitHub repository you submitted before the deadline expires**.

## 3 Report

A combined report covering all the requirements outlined in the four problems. The report must meet the following criteria:

- The combined report should be between 3,000 and 7,500 words in total.
- **The report must be submitted in PDF format**.
- The report should contain the names of all group members.
- The report should include the GitHub link for each problem, containing the code, datasets, and a README file with the necessary instructions for running and testing your implementation.
- Do not include the code within the report.

## 4 GitHub Link to Monitor the Group Activities

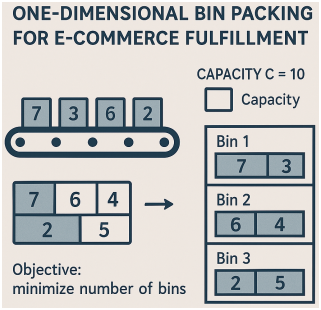
The final submission is not something you can complete in a single day. As part of the grading process, your group activities will be monitored. To meet the requirements for the final submission, you must address the following criteria:

- You need to create a GitHub repository within this week (week 41) and grant me access to it for demonstrating gradual improvement and facilitating group collaboration.
- Send a weekly group report, including.
    - Percentages and activity lists for each member.
    - Planned activities for the upcoming week.
- Demonstrating actual improvement in implementation in the weekly group report is not mandatory.

## Problem-1: One-Dimensional Bin Packing for E-commerce Fulfillment Using Ant Colony Optimization (ACO)

### Overview

Use Ant Colony Optimization (ACO) to pack items (customer orders) into the fewest shipping boxes without exceeding the box capacity C. Each ant assigns items to boxes; **pheromone** and a **tight-fit heuristic** guide good assignments.




### Problem Description

ACO for bin packing builds a packing by assigning each item to a feasible box, guided by pheromone (what worked well before) and a heuristic that prefers **tight fits**. After all ants propose packings, the best one **reinforces** its assignments, and pheromone **evaporates** elsewhere. Over iterations, ants converge to using **fewer boxes** than simple heuristics like FFD (First-Fit Decreasing, a classic greedy heuristic for the 1-D bin packing problem).

You're given a list of item sizes $w_1, \ldots, w_n \in \mathbb{Z}^+$ and box capacity $C \in \mathbb{Z}^+$, assign each item to a box so **no box exceeds capacity** and the **number of boxes used** is minimized.

### Primary objective (minimize):

$$\text{Cost}(S)=B(S) \text{ (number of non-empty boxes)}$$

*(Optionally, use a tiebreaker: smaller total unused space.)*

### Data (explicit, public)

Use standard **bin packing** benchmarks from the **OR-Library** (Beasley):

- 1-D Bin Packing (Beasley / Falkenauer instances):
  https://people.brunel.ac.uk/~mastjjb/jeb/orlib/binpackinfo.html
  (Note: OR-Library names these files **binpack1 … binpack8**, not bp1…bp5. The page also links to the actual data files.)
- **2-D Bin Packing (rectangular):**
  https://people.brunel.ac.uk/~mastjjb/jeb/orlib/binpacktwoinfo.html

If you're open to a broader collection, **BPPLIB** is another widely used bin-packing library:
https://site.unibo.it/operations-research/en/research/bpplib-a-bin-packing-problem-library

**Detailed Instructions**

**Formulation**

- Items: sizes $w_1, \ldots, w_n \in \mathbb{Z}^+$.
- Boxes: capacity $C$.
- Decision: assign each item $i$to a box label $b \in \{1, \ldots, B\}$.
- Feasibility: for every box $b$, $\sum_{i \in b} w_i \leq C$
- Objective: minimize $B$ (boxes actually used).

**Outputs**

- Primary: B (boxes used).
- Secondary: total unused capacity $\sum_b (\mathrm{C} - \mathrm{load}_b)$, runtime.

**Evaluation Protocol**

Report the results of at least 8 benchmark problems and include the following metrics (mentioning the name of the benchmark problems).

**Metrics:**

- #boxes (lower is better).
- unused capacity,
- runtime

**Plots:**

- Convergence: iteration vs. best #boxes.
- Load distribution across boxes.

**Deliverables**

- Code: ACO bin-packing solver (+ optional local repair), README.
- Report:
  - Dataset details (instance names)
  - Implementation details including every step and component of ACO so that the implementation can be reproduced.
  - Parameters ($\alpha, \beta, \rho, Q$, #ants, iterations)
  - Results: Report the results of at least 8 benchmark problems and include the above metrics and plots.

**Problem-2: Optimize Classic Benchmark Functions using Particle Swarm Optimization (PSO)**

**Overview**

Use Particle Swarm Optimization (PSO) to **minimize** several standard continuous test functions. Compare convergence speed and final accuracy across functions and dimensions.

**Functions**

Evaluate your implementation using all of the test functions listed below.

1. **Sphere (convex, easy)**
   $f_{\text{Sphere}}(x) = \sum_{i=1}^{n} x_i^2$
   Global min at $x^{\backslash *} = \mathbf{0}$, $f^{\backslash *} = 0$.
   **Bounds:** $x_i \in [-5.12,\ 5.12]$.
2. **Rosenbrock (valley, nonconvex)**
   $f_{\text{Ros}}(x) = \sum_{i=1}^{n-1} \left[100\,(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right]$
   Global min at $x^{\backslash *} = \mathbf{1}$, $f^{\backslash *} = 0$.
   **Bounds:** $x_i \in [-5,\ 10]$.
3. **Rastrigin (multimodal, many local minima)**
   $f_{\text{Rast}}(x) = 10n + \sum_{i=1}^{n} \left[x_i^2 - 10\cos(2\pi x_i)\right]$
   Global min at $x^{\backslash *} = \mathbf{0}$, $f^{\backslash *} = 0$.
   **Bounds:** $x_i \in [-5.12,\ 5.12]$.
4. **Ackley (multimodal, bowl near optimum)**
   $f_{\text{Ack}}(x) = -20\exp\left(-0.2\sqrt{\frac{1}{n}\sum x_i^2}\right) - \exp\left(\frac{1}{n}\sum\cos(2\pi x_i)\right) + 20 + e$
   Global min at $x^{\backslash *} = \mathbf{0}$, $f^{\backslash *} = 0$.
   **Bounds:** $x_i \in [-32.768,\ 32.768]$.

**Experimental setup**

- Dimensions: run each chosen function at n = 2, 10, 30.
- Budget: max function evaluations (e.g., 30,000) or **iterations** (e.g., 300).
- Runs: 30 independent runs per (function, n) to capture variance.

**PSO configuration (good defaults)**

- Swarm size: 30 (increase to 50 for n=30).
- Iterations: 300 (or until evaluation budget hits).
- Parameters: $\omega = 0.7$, $c_1 = c_2 = 1.5$.
- Init: sample positions uniformly in bounds; velocities in ~10–20% of range.
- Velocity clamp: $|\ v_i\ | \leq 0.5\ (\text{upper}_i - \text{lower}_i)$.
- Position update: $x \leftarrow x + v$, then clip to bounds.
- Stopping: budget exhausted or $\min f$ below a tiny threshold (e.g., $10^{-8}$ for Sphere/Rosenbrock, $10^{-4}$ for Rastrigin/Ackley).

**What to record**

- Per iteration: best fitness (**gbest**).
- At the end of each run: final best fitness, best position, evaluations used.
- Aggregate over 30 runs: **mean/median/best/worst** fitness, **std dev**, and **success rate** (hit threshold or not).

**Deliverables**

- Code (PSO with clear parameters & reproducibility—seed control).
- Report mentioning the below points:
    - Functions + bounds + dimensions tested.
    - Implementation details, including every step and component of PSO, so that the implementation can be reproduced.
    - Parameter study: compare $(\omega, c_1, c_2)$ settings.
    - Comparison: local best ($l_{best}$) vs. global best ($g_{best}$).
    - Convergence plots (iteration vs. best fitness) for each function (n=10 recommended).
    - Boxplots or tables of final fitness across 30 runs.
    - Brief analysis: which functions were the hardest and why (valleys vs. multimodality).

**Evaluation Criteria**

- Correctness: functions implemented exactly; bounds enforced; results reproducible.
- Effectiveness: Sphere should reliably reach ~$10^{-12}$–$10^{-8}$ by n=30; Rosenbrock should steadily decrease (may not hit 0 at n=30); Rastrigin/Ackley should improve substantially from start (may plateau above 0 for higher $n$).
- Clarity: tidy code, labeled figures, concise discussion of behavior + README.

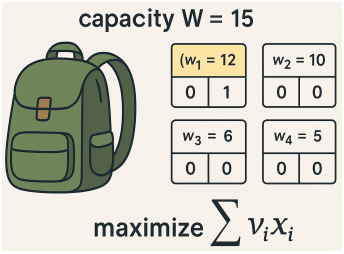
**Optional**

- Hybrid local search: a few gradient-free refinements on gbest (e.g., Nelder–Mead).

**Problem-3: Solving 0–1 Knapsack Problem Using Bees Algorithm (BA)**

**Overview**

Use the Bees Algorithm (BA) to solve 0–1 Knapsack problem— pick the best items under a weight limit. Here, the aim is to select a subset of items with **maximum total value** without exceeding a **weight capacity**.




**Problem Description**

You are given $n$ items. Each item $i$ has value $v_i > 0$ and weight $w_i > 0$. Choose a subset to **maximize total value** while keeping total weight $\leq W$.

**Decision:** $x_i \in \{0,1\}$ (1 = take item $i$, 0 = skip).

**Objective (maximize):**

$$\max_{x\in\{0,1\}^n} \sum_{i=1}^{n} v_i x_i \quad \text{s.t.} \quad \sum_{i=1}^{n} w_i x_i \leq W.$$

*(If your code minimizes, use fitness = $-\sum v_i x_i$ with a feasibility repair or penalty.)*

**Data Requirements (explicit, public)**

Use OR-Library 0/1 knapsack instances.

- OR-Library – Multidimensional Knapsack (MKP) info & downloads (incl. classic literature sets): https://people.brunel.ac.uk/~mastjjb/jeb/orlib/mknapinfo.html
- Pisinger's "hard" 0/1 knapsack instances (knapPI_… series) + generators + solutions: https://hjemmesider.diku.dk/~pisinger/codes.html
- Optional extra) Small 0/1 knapsack datasets for teaching (Burkardt collection): https://people.sc.fsu.edu/~jburkardt/datasets/knapsack_01/knapsack_01.html

In your report, name the exact instance IDs you solved (e.g., weing1, weing2, or knapPI_1_50_100).

**Evaluation Protocol**

- Instances: solve at least **6 (six)** named OR-Library instances (small, medium, large).
- **Metrics:**
  - Primary: best value $V(x)$ (feasible).
  - Secondary: runtime; iterations to best; average fitness of population.

**Plots/Tables**

- Convergence: iteration versus best value.
- Statistics of final best values over 10 runs (using different random seeds).
- Table: instance, $n$, $W$, best value, weight used, baseline value, improvement %.

**Deliverables**

- Code: BA knapsack solver (with repair) + baselines; a README (how to run on each instance).
- Report:
    - Instances used.
    - Implementation details, including every step and component of BA, so that the implementation can be reproduced.
    - BA settings,
    - Neighbourhood choices,
    - Convergence plot,
    - Brief discussion of sensitivity (effect of $n_{re}$, $n_{rb}$ and $n_{gh}$).
    - Plots/Tables mentioned above.

**Grading / Success Criteria**

- Correctness: final solution feasible ($W(x) \leq W$); repair implemented correctly.
- Effectiveness.
- Clarity & Reproducibility: clean code, fixed seed option, labeled figures, README.

## Problem-4: Spam Detection from Text Messages/Emails Using Artificial Immune Systems (AIS) – Negative Selection Algorithm (NSA)

### Overview

Implement Artificial Immune Systems (AIS), especially the Negative Selection Algorithm (NSA), to detect spam in text messages and emails. This task involves using NSA to detect non-self patterns (spam) after training solely on self (legitimate or ham) messages. Your system will learn exclusively from ham to define self, generating numerous detectors that do not match self according to an r-contiguous (or Hamming) rule. During testing, if a message matches any detector, it should be flagged as spam. Adjust the number of detectors and the matching radius to find a balance between recall and false positives. You will design encodings and detectors, create a set of detectors that do not match any self, and classify unseen messages as spam if they are matched by detectors.

### Problem Description

Given a corpus of text messages/emails labeled **ham** (legitimate) or **spam**, learn only from the **ham (self)** subset to produce a set of **detectors** that signal **non-self**. At test time, a message is flagged as **spam** if one or more detectors match it.

**Primary objective (maximize):** spam detection quality on a held-out test set.

**Metrics:** Precision/Recall/F1 for the spam class (and ROC-AUC/PR-AUC).

### Data Requirements (explicit, public — choose one)

- SMS Spam Collection (UCI Machine Learning Repository) — 5,574 SMS labeled ham/spam.
- SpamAssassin Public Corpus — labeled ham/spam emails (raw text).
- Enron Spam Dataset — Enron emails with spam/ham labels.

In your report, **name the exact dataset** you used and the **train/validation/test split**.
*(If email datasets are too large for your setup, prefer the SMS Spam Collection.)*

### Outputs

- Detector statistics: Average pairwise matching rule you implemented (*r*-contiguous bits/Hamming radius/*r*-chunk (wildcards))average matches per spam/ham on validation.
- Confusion matrix and Precision/Recall/F1 on test; ROC/PR curves if using a score (e.g., count of matches).

### Evaluation Protocol

**Metrics (report all):**

- **Primary: Spam F1** (or **Macro-F1** if class imbalance is high).
- **Secondary:** Spam **precision**, **recall**, and **false positive rate** on ham; **ROC-AUC/PR-AUC** (optional).

**Plots & Tables:**

- Detector **coverage curve**: recall vs. number of detectors.
- **Precision–Recall** curve (spam as positive).

**Deliverables**

- **Code:** NSA training (detector generation), inference, and metric scripts; baseline classifiers; README with exact commands.
- **Short report:**
    - Dataset and splits: Name and describe the dataset you used and the train/validation/test split.
    - Implementation details, including every step and component of NSA, so that the implementation can be reproduced.
    - Encoding and matching rule, NSA hyperparameters.
    - Results (with tables/plots/statistics mentioned above), and discussion (trade-offs between precision and recall; detector diversity).

**Evaluation Criteria**

- **Correctness:** Negative selection implemented (no detector matches any self in train).
- **Effectiveness**.
- **Clarity & Reproducibility:** clean code, fixed seed option, clear dataset citation and splits, labeled plots, README

**Problem 5: Solving a Real-World Problem Using Reinforcement Learning**

**Overview**

This lab exercise applies reinforcement learning—specifically **Q-learning**—to address a real-world-inspired control task. Students will use a publicly available environment to train an RL agent, assess its performance, and optimise it for robust behaviour.

**Problem Statement: Warehouse Robot on a Slippery Floor**

Students will develop a warehouse floor robot that must navigate from a loading bay to a target shelf without falling into hazards (holes) and while dealing with a slippery surface. Because the robot's movement can slip unpredictably, it needs to learn a safe, efficient route instead of relying on fixed plans.

This exercise uses the FrozenLake-v1 environment (Gym), which models a slippery warehouse floor as a grid with safe tiles, holes (hazards), a start point, and a goal. The agent must learn a policy that **maximizes success rate** and **minimizes steps** under stochastic transitions.

**Dataset / Environment:**

- **Environment:** FrozenLake-v1 (4×4 or 8×8 map; "slippery=True" to induce stochastic motion).
- **Observation space (state):** a discrete tile index (grid cell).
- **Action space:** {Left, Down, Right, Up}.
- **Rewards:** Reaching the goal yields **+1**; falling into a hole yields **0** (episode terminates). Stepping on safe tiles yields **0** (sparse reward).
- **Real-world analogy:** Autonomous robot navigating a slick warehouse aisle with spill zones (holes) to reach a pick location (goal).

**Environment Description:**

- **Grid size:** 6×6 (default) or 8×8 for a harder variant.
- **Start/Goal:** Fixed start at "S" and target shelf at "G".
- **Hazards:** "H" tiles represent spill pits; entering them ends the episode.
- **Stochasticity:** On a slippery floor, intended actions may **slip** to adjacent directions, modeling real-world uncertainty (wheels slipping, micro-surface variation).

**Tasks:**

1. Understanding the Environment:
    - Instantiate FrozenLake-v1 and print state and action spaces.
    - Visualize the grid and annotate S (start), G (goal), and H (holes).
    - Explain the reward structure and the effect of **slippery=True**.
2. Setting Up the RL Agent:
    - Implement **tabular Q-learning** with a Q-table of size **[n_states × n_actions]**.
    - Use **ε-greedy** exploration (ε decay), learning rate **α**, discount factor **γ**.
    - Initialize Q(s, a) = 0 for all feasible state–action pairs.

3. Training the RL Agent:
    - Train over many episodes (e.g., 10k+) to handle sparse rewards and slippage.
    - Tune **α, γ, ε schedule**; track episode returns and success rate.
    - Consider separate runs for 6×6 vs 8×8 maps to compare difficulty.
4. Evaluation:
    - Report **success rate** (fraction of episodes reaching the goal).
    - Plot **cumulative reward per episode** and a moving average.
    - Compare against:
        - **Random policy** baseline.
        - **Simple heuristic** (e.g., always attempt a shortest deterministic route) to show why planning fails on slippery floors.
5. Optimization:
    - Experiment with:
        - **ε schedules** (linear vs exponential decay),
        - **γ** (myopic vs far-sighted),
        - **α** (stability vs speed).
    - ***Optional algorithmic variants***: **SARSA** (on-policy), **Double Q-learning** (reduces overestimation).
    - ***Optional shaping***: small step penalty (−0.01) to encourage shorter paths (discuss pros/cons).
6. Reporting:
    - Document your approach, design decisions (hyperparameters, ε schedule), and training curves.
    - Discuss challenges: **sparse rewards** and **stochastic transitions**.
    - Note potential improvements: larger Q-tables (8×8), eligibility traces, or moving to function approximation (DQN) if you later convert observations to rich features (optional).

**Deliverables:**

- **Code:** Python notebook/script with Q-learning implementation, training loop, evaluation, and plots. Include a **README** with setup steps and run instructions.
- **Report:** Brief write-up describing the problem, method, results (plots + metrics), and conclusions/next steps.

**Tools and Libraries:**

- Python (3.x)
- **Gym/Gymnasium** (for FrozenLake-v1)
- **NumPy** (Q-table, numeric ops)
- Matplotlib (for plotting results)
- TensorFlow/PyTorch (optional, for more advanced RL algorithms like DQN)

**Evaluation Criteria**

- Correctness and Completeness of Implementation
- Comparison and Analysis of Algorithm Variations: learning curves; compare against random/heuristic baselines.
- Depth and Insight of Analysis
- Quality and Clarity of the Report
- **Code Quality, Documentation &** Clarity of the Report**:** clean structure, comments, clear README; reproducible results.