University of Exeter

# COURSEWORK SPECIFICATION

ECM1414 – Algorithms and Data Structures

Module Leader: **Prof Ronaldo Menezes**
Lecturers**: Prof R. Menezes, Prof A. Kayem, Dr R. Hancocks**

---

## Title: Student Society Event Planner

## Submission deadline: **24 February, 2026 (Presentations starting on 3 March 2026)**

This assessment contributes **30%** of the total module mark and assesses the following **intended learning outcomes**:

- describe the properties of algorithms, explaining clearly their relationship to programs and to heuristics
- give rigorous specifications of algorithms and construct solutions to algorithmic tasks using pseudocode
- show an awareness of the issues of termination and correctness, demonstrating these for some simple examples
- make use of recursion and iteration in constructing algorithms
- explain the meaning of computational complexity, distinguishing space and time complexity, and evaluating the complexity of a range of different algorithms
- demonstrate enhanced practical skills in programming and problem analysis as a result of the deeper theoretical understanding gained from this module
- describe computational phenomena using a range of key theoretical concepts
- ability to work in groups; teamwork.
- present and explain abstract ideas using a systematic approach

## Plagiarism

This is group assessment.

Plagiarism is interpreted by the university as the act of presenting the work of others as one's own work, without acknowledgement. It is considered academically

fraudulent and an offence against university discipline. Your attention is drawn to the university's regulations on plagiarism.

## Generative AI

This assessment has been categorised as **AI-Minimal** (see Section 12.2 below). You can find further information in the university's policies around using AI in assessed work

**Permitted Uses of Generative AI:**

Students may use Large Language Models (LLMs) such as ChatGPT, Claude, or similar tools **only** for the following non-coding tasks:

- **Improving written clarity**: Rephrasing or improving the clarity of sentences in your written report
- **Document structuring**: Generating ideas for how to structure your document
- **Concept clarification**: Explaining general concepts (e.g., "What is dynamic programming?") as a learning aid to support your understanding before writing your own explanations and code

**Prohibited Uses:**

Students **must not** use LLMs for:

- Writing functions, algorithms, or any portions of code
- Debugging code by pasting code and asking the tool to fix errors
- Generating pseudocode that is then directly translated into code
- Any task that involves generating, debugging, or substantially altering source code

**Declaration Requirement:**

All students must include a "Generative AI Statement" in their written report stating either:

- If LLMs were used: the tool name (e.g., ChatGPT, Claude), what it was used for, and representative examples of prompts/questions asked
- If LLMs were not used: a clear statement that no LLMs were used

# Project: Student Society Event Planner

Module Code: ECM1414
Academic Year: 2025/26
Submission Deadline: 24 February 2026

## 1. Introduction

This coursework is designed to help you apply the fundamental concepts of algorithms and data structures that you have studied in the first half of this module, although there is an expectation for some independent inquiry. You will work in a group (group size 5–6 people) to design, implement, and evaluate a program that solves a realistic computational problem. The coursework emphasises both correctness and algorithmic efficiency, requiring you to compare a straightforward brute-force approach with a more sophisticated algorithmic solution.

## 2. Coursework Objectives

By completing this coursework, you will:

- Apply your understanding of fundamental data structures (arrays, lists, stacks, queues) to solve a practical problem.
- Implement and compare multiple algorithmic approaches to the same problem, including brute-force and more efficient strategies.
- Gain experience in designing, implementing, testing, and documenting a non-trivial software system.
- Develop your ability to work collaboratively in a group, dividing tasks effectively and integrating your contributions.
- Communicate your design decisions and results both in writing and through an oral presentation.

## 3. Scenario Description

You are part of the committee for a university student society that is planning a special event weekend. Your society has received funding and a fixed amount of time to run activities during this weekend, but you cannot do everything you'd like to do. You have a list of potential activities, each with a specific time requirement, a cost, and an estimated “enjoyment value” based on feedback from society members.

Your task is to build an **Event Planner** system that helps the committee decide which activities to include in the weekend schedule. The goal is to maximise the total enjoyment value of the selected activities while staying within the constraints of:

- **Available time:** The weekend has a fixed number of hours available for activities (for example, 12 hours on Saturday and Sunday combined).
- **Budget:** The society has a fixed amount of funding (for example, £500) that can be spent on activities.

Each activity can either be included in the schedule or excluded, meaning that you cannot partially do an activity, and you cannot repeat the same activity multiple times during the weekend. Some activities might be very enjoyable but expensive or time-consuming, while others might be cheaper and quicker but offer less enjoyment. The challenge is to find the best combination of activities that fits within both the time and budget limits (or whichever single constraint is most pressing for a given scenario).

For example, imagine the society is considering activities such as:

- A guided campus tour for new members (2 hours, £20, enjoyment value 50).
- A game night with board games and snacks (3 hours, £80, enjoyment value 120).
- A trip to a local museum (4 hours, £100, enjoyment value 150).
- A pizza-making workshop (2 hours, £60, enjoyment value 100).
- A hiking excursion to a nearby trail (5 hours, £30, enjoyment value 140).

If the society has only 10 hours available and a budget of £200, which activities should be chosen to maximise enjoyment? A naive approach might be to try every possible combination of activities, but as the number of potential activities grows, this becomes computationally prohibitive. Your program should implement both a straightforward exhaustive search (brute-force) and a more efficient algorithmic solution, allowing you to compare their performance and correctness.

This scenario is representative of many real-world decision-making problems where resources are limited, and choices must be made to optimise some objective. By the end of this coursework, you will have built a practical tool that could genuinely assist a student society (or similar organisation) in planning their activities more effectively. Furthermore, such projects represent real-world applications that you can demonstrate to future employers. It is recommended that you think about having a github page where all your projects (such as this one) are available; you should have a portfolio.

# 4. Formal Problem Specification

## 4.1 Input Space

You are given:

- **n** potential activities, where each activity *i* (for *i* = 1, 2, ..., *n*) has the following attributes:
  - **Name:** A short descriptive name or identifier for the activity.
  - **Time Required (hours):** A positive integer representing how many hours the activity takes.
  - **Cost (£):** A positive integer representing how much the activity costs.
  - **Enjoyment Value:** A positive integer representing the estimated enjoyment or benefit the activity provides to society members.
- Constraints:
  - **Maximum Available Time (T):** The total number of hours available for activities during the event weekend.

- **Maximum Budget (B):** The total funding available to spend on activities.

For the purposes of this coursework, you should simplify the problem by focusing on a **single primary constraint** (either time or budget) in your core implementation. If you wish to handle both constraints simultaneously, this can be considered as an optional extension (see Section 8).

- If using time as primary, simply ignore the value for budget.
- If using budget as primary, simply ignore the value for time.
- See Section 5 for input format.

## 4.2 Constraints

- Each activity can be selected **at most once**. You cannot choose the same activity multiple times.
- The sum of the time required by all selected activities must not exceed the maximum available time **T**. If considering budget as the primary constraint instead, the sum of the costs of all selected activities must not exceed the maximum budget **B**.
- You must select a **subset** of the available activities (possibly empty, though typically at least one activity will exist if you have appropriate scenarios).

## 4.3 Objective

Your goal is to **maximise the total enjoyment value** of the selected activities, subject to the constraints above. In other words, find a combination of activities such that:

- The total time used ≤ T (or total cost ≤ B, depending on your choice of primary constraint).
- The total enjoyment value is as large as possible.

## 4.4 Algorithmic Approaches

You are required to implement and compare **two algorithmic strategies**:

1. **Brute-Force (Exhaustive Search):**
   Generate and evaluate all possible subsets of activities (there are $2^n$ such subsets). For each subset, check whether it satisfies the constraint(s), and if so, compute its total enjoyment value. Keep track of the subset with the highest enjoyment value that is feasible. This approach guarantees finding the optimal solution but becomes impractical as *n* grows.
2. **Improved Algorithm (Dynamic Programming):**
   Implement a more efficient algorithm based on dynamic programming principles. This approach should systematically build up solutions to larger problems from solutions to smaller subproblems, avoiding redundant computation. The algorithm should still produce the optimal solution, but do so more efficiently than brute force for larger instances. Note that it is expected

that the group will do some independent research about dynamic programming. Not all the concepts will be explained in class. Dynamic programming will be mentioned in class, but we **do not** recommend you wait until it is. If you do, you may not have enough time to work on your project.

You are expected to explain both algorithms clearly in your documentation, provide pseudocode, and compare their performance on inputs of varying sizes.

# 5. Input File Format

Your program must read its configuration from a **plain text input file** (this is not rtf, docx, etc. It's plain text, .txt). The format is defined as follows:

## 5.1 General Description

- **Line 1:** A single integer **n**, the number of activities available.
- **Line 2:** Two integers separated by a space: **T** (maximum available time in hours) and **B** (maximum budget in pounds). You may choose to enforce only one of these constraints in your core implementation, but both values should be present in the input file for completeness. When dealing with one constraint, simply ignore the other value. This helps us having to deal with just one format for the input file.
- **Lines 3 to (n+2):** Each line describes one activity, in the following format:
  - **Activity Name** (a single word or a hyphen-separated phrase with no spaces, e.g., Board-Games or Museum-Trip)
  - **Time Required** (a positive integer, in hours)
  - **Cost** (a positive integer, in pounds)
  - **Enjoyment Value** (a positive integer)

These four fields are separated by single spaces.

Assumptions:

- All integers are non-negative.
- Activity names contain no spaces (use hyphens or underscores if needed).
- The file format is strictly adhered to: exactly *n* activity lines follow the header lines.
- There is at least one activity ($n \geq 1$).
- It is possible that no feasible solution exists if all activities exceed the constraints; your program should handle this gracefully.

## 5.2 Sample Input File 1 (Small Configuration)

**Filename:** input_small.txt (available on ELE)

```
5
10 200
Campus-Tour 2 20 50
Game-Night 3 80 120
```

```
Museum-Trip 4 100 150
Pizza-Workshop 2 60 100
Hiking 5 30 140
```

**Description:**
This configuration represents a small event weekend with 5 possible activities. The society has 10 hours and a £200 budget. This is a manageable size for manual calculation and testing. A brute-force approach will examine $2^5 = 32$ possible subsets.

**Expected Behaviour:**
Students should be able to determine by hand (and by running their brute-force program) which combination of activities maximises enjoyment while fitting within the 10-hour limit and the £200 budget. For example, if enforcing the time constraint primarily, one might select activities that total no more than 10 hours and compare their enjoyment values. The optimal solution will depend on which constraint is binding.

## 5.3 Sample Input File 2 (Medium Configuration)

**Filename:** input_medium.txt (available on ELE)

```
12
15 300
Welcome-BBQ 3 50 80
Karaoke-Night 2 40 70
Film-Screening 3 30 90
Sports-Tournament 4 60 110
Art-Workshop 2 70 95
Pub-Quiz 2 25 60
Bowling 3 80 100
Laser-Tag 2 90 130
Cooking-Class 3 75 105
Beach-Trip 6 120 180
Escape-Room 2 85 115
Open-Mic 2 20 50
```

**Description:**
This medium-sized configuration includes 12 activities with a total available time of 15 hours and a budget of £300. There are $2^{12} = 4{,}096$ possible subsets. Brute force is still feasible but noticeably slower than for the small input. The dynamic programming solution should perform more efficiently.

**Expected Behaviour:**
The program should find the optimal set of activities that maximises enjoyment within the constraints. Students should notice that the brute-force algorithm takes longer to run compared to the small input, and they should document this observation. The dynamic programming approach should yield the same optimal result but with better performance characteristics.

### 5.4 Sample Input File 3 (Large Configuration)

**Filename:** input_large.txt (available on ELE)

```
25
20 500
Orientation-Walk 1 10 30
Ice-Breaker-Games 2 20 50
Movie-Marathon 5 60 140
City-Tour 4 80 120
Charity-Run 3 15 70
Trivia-Night 2 30 65
Wine-Tasting 2 100 150
Rock-Climbing 4 110 160
Theatre-Trip 4 90 130
Pottery-Class 3 85 125
Campus-Scavenger-Hunt 3 25 75
Photography-Walk 3 40 85
Poetry-Slam 2 20 55
Dance-Workshop 2 50 90
Baking-Competition 3 55 95
Outdoor-Cinema 4 70 115
Kayaking 5 130 190
Board-Game-Cafe 3 45 80
Comedy-Show 3 80 135
Volunteering-Event 4 10 60
Yoga-Session 2 35 70
Park-Picnic 3 30 65
Museum-Evening 3 75 110
Stargazing-Trip 4 50 100
Crafts-Fair 2 40 75
```

**Description:**
This larger configuration includes 25 activities with 20 hours available and a £500 budget. There are $2^{25}$ = 33,554,432 possible subsets. A brute-force approach will take a considerable amount of time to evaluate all possibilities (potentially minutes or more, depending on implementation and hardware). The dynamic programming solution should handle this input much more efficiently.

**Expected Behaviour:**
Students should find that the brute-force algorithm is noticeably slow or even impractical for this input size. The dynamic programming algorithm should produce the optimal solution in a fraction of the time. This input is designed to highlight the importance of algorithmic efficiency and to provide a clear demonstration of the benefits of the improved approach.

## 6. Program Output

Your program must produce clear, human-readable output that shows the results of the optimisation. The output should be printed to the console (standard output) and should include the following information:

## 6.1 Required Output Elements

1. **Selected Activities:**
   List the names of all activities that have been selected in the optimal solution.
2. **Total Enjoyment Value:**
   The sum of the enjoyment values of the selected activities.
3. **Total Time Used:**
   The sum of the time required by the selected activities.
4. **Total Cost:**
   The sum of the costs of the selected activities.
5. **Constraint Summary:**
   Display the available time and budget, and confirm that the solution respects the constraints. For example:
   - "Available Time: 10 hours | Time Used: 9 hours"
   - "Available Budget: £200 | Budget Used: £190"
6. **Algorithm Comparison:**
   Your program should run both the brute-force and the dynamic programming algorithms on the same input, and report:
   - The results from each algorithm (which should be identical if both are implemented correctly).
   - A simple performance comparison, such as:
     - An informal measure of running time (e.g., "Brute force took approximately 2 seconds; dynamic programming took approximately 0.1 seconds").

## 6.2 Example Output Format

Below is an illustrative example of what the output might look like. You may format your output differently, as long as it is clear and contains all the required information. However, we **strongly suggest** you stay with the same output.

```
========================================
EVENT PLANNER - RESULTS
========================================

Input File: input_small.txt
Available Time: 10 hours
Available Budget: £200

--- BRUTE FORCE ALGORITHM ---
Selected Activities:
  - Game-Night (3 hours, £80, enjoyment 120)
  - Pizza-Workshop (2 hours, £60, enjoyment 100)
  - Hiking (5 hours, £30, enjoyment 140)

Total Enjoyment: 360
```

```
Total Time Used: 10 hours
Total Cost: £170

Execution Time: 0.002 seconds

--- DYNAMIC PROGRAMMING ALGORITHM ---
Selected Activities:
  - Game-Night (3 hours, £80, enjoyment 120)
  - Pizza-Workshop (2 hours, £60, enjoyment 100)
  - Hiking (5 hours, £30, enjoyment 140)

Total Enjoyment: 360
Total Time Used: 10 hours
Total Cost: £170

Execution Time: 0.001 seconds

========================================
```

# 7. Program Requirements

## 7.1 Baseline (Brute-Force) Algorithm

You must implement a correct brute-force algorithm that:

- **Generates all possible subsets** of the available activities. There are $2^n$ such subsets.
- For each subset, checks whether it satisfies the constraint(s) (time and/or budget).
- For each feasible subset, computes the total enjoyment value.
- **Keeps track of the best feasible solution** (the one with the highest enjoyment value).
- **Returns the optimal solution** (the set of activities, total enjoyment, time used, and cost).

Implementation Notes:

- You may generate subsets using any method you are comfortable with, such as:
  - Iterating through all integers from 0 to $2^{n-1}$ and treating each integer as a binary representation of a subset (bit manipulation).
  - Using recursive enumeration.
  - Using Python's `itertools.combinations` (or similar) to generate combinations of all possible sizes.
- **Documentation Requirement:**
  In your written document, you must explain clearly how your brute-force algorithm generates and evaluates combinations. Include pseudocode and discuss informally why the time complexity grows exponentially with *n*.

## 7.2 Improved Algorithm (Dynamic Programming)

You must implement at least one more efficient algorithm based on **dynamic programming** principles. This algorithm should:

- Systematically build up solutions to the problem by solving smaller subproblems.
- Avoid redundant computation by storing intermediate results (memoisation, or top-down dynamic programming).
- Produce the optimal solution (the same result as brute force, but more efficiently).

Guidance:

- Consider a dynamic programming approach where you maintain a table (e.g., a 2D array or list of lists) that represents the maximum enjoyment achievable for each possible "remaining capacity" (time or budget) and subset of items considered so far.
- The structure is similar to classical optimisation problems you may have encountered in other lectures or readings (though you should not name these explicitly in your code or documentation).
- Think about the problem in terms of decisions: for each activity, you decide whether to include it or not, and you combine the results of these decisions in an optimal way.

Documentation Requirement:

- Provide **clear pseudocode** for your dynamic programming algorithm in your written document.
- Explain the intuition: what subproblems are you solving, and how do you combine their solutions?
- Discuss informally why this approach is more efficient than brute force (You can focus on the idea that you avoid re-evaluating the same subproblems repeatedly. You can use big-O notation, but it not a requirement).

## 7.3 Data Structures

Your implementation should make effective use of the fundamental data structures covered in the module:

- **Lists/Arrays:** To store the set of activities, their attributes, and intermediate results in the dynamic programming table.
- **Stacks or Queues (as appropriate):** If your algorithm involves iterative deepening, backtracking, or breadth-first enumeration, you may find stacks or queues useful. Justify your choice in your documentation.
- You may use **dictionaries** (hash maps) for convenient lookups if needed. While these may be covered later in the module, their basic use for this coursework is acceptable if it improves code clarity.

You should **not** use advanced data structures not yet covered in the module, such as graphs or complex tree structures.

## 8. Optional Extensions (Extra Credit)

The following extensions are **optional** and can earn you additional marks (up to 10% of the total coursework mark, distributed across extensions). These are intended for groups who finish the core requirements early and wish to demonstrate deeper understanding or tackle more advanced challenges.

**Extension 1: Multiple Constraints (Up to 5%)**

Modify your program to handle **both time and budget constraints simultaneously**. Instead of optimising with respect to a single constraint, your algorithms must ensure that the selected activities fit within both the available time **and** the available budget.

- Update your brute-force and dynamic programming algorithms accordingly.
- Provide test cases that demonstrate scenarios where both constraints are binding.
- Discuss in your document how handling multiple constraints affects the complexity and implementation.

**Extension 2: Advanced Performance Analysis and Visualization (Up to 3%)**

Create additional test inputs with $n$ = 10, 15, 20, 25, 30, … (for example), ensuring they are realistic scenarios with varied constraints. Then:

- **Generate performance plots** showing:
  - A line graph comparing the execution time of brute force vs. dynamic programming as $n$ increases (time on y-axis, $n$ on x-axis).
  - Optionally, a log-scale plot if the differences are very large.
  - A bar chart or table showing the "speedup factor" (brute force time ÷ dynamic programming time) for each input size.
- **Analyse scalability:** In your document, discuss your findings:
  - At what value of $n$ does brute force become impractical (e.g., taking more than 10 minutes)?
  - How does the dynamic programming execution time grow as $n$ increases?
  - Do your results match the theoretical complexity you would expect?
- **Present professionally:** Include well-labelled graphs with titles, axis labels, and legends. Graphs may be generated using Python libraries such as `matplotlib` or similar tools.

You may need to create your own input files for this extension to ensure a range of problem sizes and difficulty levels.

**Extension 3: Greedy Heuristic Comparison (Up to 2%)**

Implement a **greedy heuristic** that attempts to approximate the optimal solution quickly by selecting activities based on some ratio (e.g., enjoyment per hour, or enjoyment per pound). Compare the results of this heuristic to the optimal solutions produced by your brute-force and dynamic programming algorithms.

- Discuss in your document: How close does the greedy heuristic come to the optimal solution? Are there cases where it performs well, and cases where it performs poorly?

If you choose to pursue extensions, clearly mark them in your document and code. Extensions are optional and will be assessed based on correctness, clarity, and the depth of understanding demonstrated.

# 9. Deliverables

You must submit the following items by the deadline. All files should be packaged together in a **single ZIP archive**.

## 9.1 Written Document (PDF or Word)

A formal written report that includes:

1. **Introduction:**
   Briefly describe the problem you have implemented (the Event Planner scenario) and the objectives of the coursework.
2. **Design and Approach:**
   Explain your overall design decisions:
   - How you structured your program (e.g., modules, functions).
   - What data structures you used and why.
   - Any assumptions or simplifications you made.
3. **Algorithms:**
   For both the brute-force and dynamic programming algorithms:
   - Provide clear pseudocode.
   - Explain the logic and approach in plain language.
   - Discuss informally the time and space complexity (you may use terms like "exponential" or "polynomial" without formal Big-O notation if you prefer but aim to convey the relative efficiency).
4. **Testing and Results:**
   - Describe how you tested your program.
   - Present the results obtained for each of the three provided sample input files (or additional test cases if you created your own).
   - Compare the results and performance of the brute-force algorithm versus the dynamic programming algorithm. Include any timing information or counts of operations if available.
   - Discuss any limitations or edge cases you encountered.
5. **Optional Extensions (if attempted):**
   Describe any extensions you implemented, how they work, and what you learned from them.

6. **Generative AI Statement (if applicable):**
   If you used any Generative AI tools for assistance with the report (e.g., rephrasing sentences, generating ideas for structure), you must include a short statement declaring:
   - Which tool(s) you used.
   - For what purpose (e.g., "to help rephrase the introduction section").
   - A brief description of the prompts or assistance received.
7. **References:**
   Cite any external sources (textbooks, websites, articles) you consulted. Use a consistent referencing style (e.g., Harvard, IEEE).

The document should be professionally formatted with clear headings, numbered sections, and appropriate use of bullet points and figures/tables where helpful. Aim for approximately 8 pages (12pt) (excluding appendices and code listings, though you may include short code snippets for illustration).

## 9.2 Group Contribution Statement

A separate document (PDF or Word, 1–2 pages) that clearly outlines the contributions of each group member. Include:

- The name and student ID of each member.
- A breakdown of who was responsible for which parts of the project. For example:
  - Algorithm design (brute-force, dynamic programming).
  - Implementation (coding specific functions or modules).
  - Testing and debugging.
  - Documentation (writing sections of the report).
  - Presentation preparation.

Be honest and specific. If the workload was divided unevenly, state this clearly. This document helps the module leader understand individual contributions and may be used to adjust marks if there is evidence of unequal effort.

## 9.3 Source Code

All Python source files required to run your program. Your code should:

- Be well-organised and readable, with meaningful variable and function names.
- Include **comments** explaining key sections of the code, particularly the algorithmic logic.
- Be free of syntax errors and run correctly on a standard Python installation (Python 3.x).

Required files:

- Your main Python script(s) (e.g., `event_planner.py`).
- Any additional modules or helper files you created.
- A **README.txt** or **README.md** file that explains:

- How to run the program (e.g., python event_planner.py input_small.txt).
- Any dependencies or libraries required (though you should aim to use only standard Python libraries where possible).
- A brief overview of the file structure.

**Note:** Ensure your code is portable. The markers should be able to run your program on their own machines with minimal setup. Avoid hard-coded file paths or dependencies on specific operating systems.

## 9.4 Presentation

Your group will deliver a **10-minute presentation** during a scheduled workshop session. All group members must participate. The presentation should include:

1. **Introduction to the Problem (2 minutes):**
   Briefly explain the Event Planner scenario and the optimisation objective.
2. **Algorithm Explanations (3 minutes):**
   Describe your brute-force and dynamic programming approaches. Use slides with pseudocode or diagrams to aid understanding. Highlight the key differences and advantages of the dynamic programming approach.
3. **Demonstration (3 minutes):**
   Run your program live (or show a recorded demo) on at least one of the sample input files. Show the output clearly and explain what the results mean.
4. **Results and Discussion (2 minutes):**
   Discuss the performance comparison between the two algorithms. Mention any interesting findings, challenges you faced, or lessons learned.
5. **Questions (~2 minutes, not part of the presentation 10-minute limit):**
   Be prepared to answer questions from the module leader or peers about your implementation, design choices, or results.

**Deliverable:** Submit your presentation slides (PDF or PowerPoint) along with your other files. You do not need to submit a video recording unless otherwise instructed.

## 9.5 Required Folder Structure for Submission

To help you organise your submission, consider using the following folder structure inside your ZIP archive:

```
Group_X_ECM1414_Coursework.zip
│
├── Documents/
│   ├── Report.pdf
│   └── Group_Contribution_Statement.pdf
│
├── Code/
│   ├── event_planner.py
│   ├── README.txt
│   └── [any other .py files]
│
├── Input_Files/
```

```
│   ├── input_small.txt
│   ├── input_medium.txt
│   └── input_large.txt
│
└── Presentation/
    └── Presentation_Slides.pdf
```

Replace "Group_X" with your group identifier if assigned in the beginning of the term. If you have more sample files they should all be listed in the Input_files folder

# 10. Assessment and Marking Scheme

The coursework is worth **100%** of the module's coursework component and is marked as follows:

## 10.1 Documents Submitted (60%)

Marks are awarded for:

- Written Report (40%):
  - Clarity and completeness of the problem description, design, and algorithms.
  - Quality of pseudocode and explanations.
  - Depth and insight in the testing and results section.
  - Professional presentation and correct use of references.
- Source Code Quality and Correctness (15%):
  - Does the code compile and run without errors?
  - Are both algorithms implemented correctly and do they produce correct results?
  - Is the code well-organised, readable, and appropriately commented?
  - Does the code demonstrate effective use of data structures?
- Group Contribution Statement (5%):
  - Is the statement clear, detailed, and honest?
  - Does it provide sufficient information to understand each member's role?

Important Notes:

- Code that does not compile or run will incur a **substantial penalty** (potentially losing most of the code marks). Ensure you test your program thoroughly before submission.
- Partial credit may be awarded for a well-documented design and clear pseudocode even if the implementation is incomplete, but you should strive for a fully working solution.
- Submissions that do not include all required sample input files or that fail to produce the required output format will lose marks.

## 10.2 In-Class Presentation (40%)

Marks are awarded for:

- Content and Explanation (15%):
  - Clarity and accuracy in explaining the problem, objectives, and algorithms.
  - Demonstrated understanding of the code and the algorithmic principles.
- Demonstration (10%):
  - Quality and clarity of the live or recorded demo.
  - Ability to show the program running correctly on sample input(s).
  - Explanation of the output and what it means.
- Participation and Teamwork (10%):
  - All group members participate meaningfully in the presentation.
  - Effective coordination and division of speaking roles.
  - Ability to answer questions confidently and accurately.
- Presentation Skills (5%):
  - Use of clear, professional slides.
  - Good time management (staying within the 10–15 minute limit).
  - Engaging delivery and appropriate use of visuals or diagrams.

Important Notes:

- If a group member does not participate in the presentation, their individual mark for the presentation component may be reduced or set to zero, depending on the circumstances. Please inform the module leader in advance if there are any issues.
- Be prepared to answer questions about your code, your design choices, and your algorithms. The markers may ask you to explain specific parts of your implementation or to discuss alternative approaches.
- The presentations will be assessed jointly by the 3 academics involved in the delivery of the module. Groups will be presenting at different day (staring on March 3, 2026) and marks for the presentation will be released after the last group presents. PTAs may also be consulted for feedback and will be required to attend the presentations.
- Only the group presenting will be allowed in the room during the presentations to minimise any disadvantage of presenting earlier or late in the process.

## 10.3 Optional Extensions (Up to 10% Extra Credit)

If you attempt any of the optional extensions described in Section 8, you may earn up to an additional **10%** added to your overall coursework mark. This is assessed based on:

- Correctness and completeness of the extension.
- Clarity of explanation in your document.
- Demonstrated deeper understanding of the algorithmic or computational principles involved.

Extensions will be marked generously if they show genuine effort and insight, even if not perfect.

**Note:** Your total coursework mark **cannot exceed 100%**. Extensions only help students reach 100% if they lost marks elsewhere.

**Important Warning About Code Modularity:**

If your code is not modular and your extension attempts interfere with or break the core functionality required in Section 7, this **may negatively affect your marks for the core requirements**. You should:

- Implement extensions in separate, clearly labelled functions or modules where possible.
- Ensure that the core requirements (brute-force algorithm, dynamic programming algorithm, basic input/output) work correctly **independently** of any extensions.
- Test thoroughly to confirm that adding extension code does not introduce bugs into your baseline implementation.

**Students should carefully weigh the risks and benefits of attempting extensions.** If you are not confident in your ability to maintain clean, modular code, it may be safer to focus on perfecting the core requirements rather than risking marks by attempting extensions that could compromise your baseline implementation.

## 11. Deadline and Submission Instructions

**Deadline:** 24th February 2026 (the time is irrelevant as you are considered late as soon as the day is the 25th of February

Where to Submit:

- Submit your ZIP archive via the module's online submission portal: ELE page
- Ensure the filename of your ZIP archive includes your group identifier (e.g., `Group_10_ECM1414_Coursework.zip`).

Late Submissions:

- Late submissions will be subject to the university's standard penalties for late coursework.
- If you have extenuating circumstances (illness, personal issues), you must apply for an extension or mitigation through the appropriate university channels **before** the deadline.

Technical Issues:

- If you experience technical problems with submission (e.g., portal downtime), contact the HUB immediately. It is recommended you keep evidence (screenshots, error messages) and keep a timestamped copy of your files.

Checklist Before Submission:

☐ Written report (PDF or Word) included.

☐ Group contribution statement included.

☐ All source code files included and tested.

☐ README file with clear instructions.

☐ All sample input files included.

☐ Presentation slides included.

☐ ZIP archive named correctly.

☐ Code runs without errors on a clean Python installation.

# 12. Academic Integrity and Use of Generative AI

The University takes academic integrity very seriously. All work you submit must be your own group's work. The following rules apply:

## 12.1 Plagiarism and Collaboration

- **Plagiarism is strictly prohibited.** This includes copying text, code, or ideas from other groups. Online sources, textbooks, or any other source must have proper attribution.
- Collaboration between groups:
  - You may discuss high-level ideas, approaches, and concepts with students from other groups (e.g., "How did you think about the brute-force algorithm?" or "What data structure did you find useful?").
  - However, **you must not share code** between groups. Copying code, even with modifications, is considered plagiarism.
  - Each group must write their own implementation independently.
- Use of External Sources:
  - You may consult textbooks, online tutorials, and documentation to learn about algorithms and data structures.
  - If you adapt an idea, algorithm, or code snippet from an external source, you **must** cite it clearly in your report and in code comments. Provide the URL or bibliographic reference.
  - Simply changing variable names or reformatting code taken from elsewhere does not make it your own work.

## 12.2 Use of Generative AI (Large Language Models)

The use of Large Language Models (LLMs) such as ChatGPT, Claude, GitHub Copilot, or similar tools **is not permitted** for generating, debugging, or substantially altering your source code.

- What is not allowed:
  - Using an LLM to write functions, algorithms, or substantial portions of your code.
  - Using an LLM to debug your code by pasting in your code and asking it to fix errors.

    - Using an LLM to generate pseudocode that you then translate directly into code without understanding it.
- What is allowed (with declaration):
    - You may use LLMs for **non-coding tasks** such as:
        - Helping to rephrase or improve the clarity of sentences in your written report.
        - Generating ideas for how to structure your document.
        - Explaining general concepts (e.g., "What is dynamic programming?") as a learning aid. Note however that you must then write your own explanations and code.
    - **If you use an LLM for any purpose** (even these non-coding tasks), you **must** include a **"Generative AI Statement"** in your written report. This statement should include:
        - The name of the tool (e.g., "ChatGPT", "Claude").
        - A clear description of what you used it for (e.g., "to rephrase the introduction section", "to clarify the concept of dynamic programming before implementing our own version").
        - Representative examples of prompts or questions you asked, or a summary of the assistance received.
    - If you did not use LLMs you still need to declare clearly that LLMs were not used.
- **Why these restrictions exist?**
  The purpose of this coursework is for you to learn and demonstrate your own understanding of algorithms and data structures, you want to become a computing professional. Using an LLM to generate code for you undermines this learning and is not fair to other students. It also makes it difficult for us to assess your true understanding.

### 12.3 Consequences of Breaches

Any breach of these academic integrity rules will be treated as **academic misconduct** under the university's regulations.

If you are unsure whether something you are doing is acceptable, **ask the module leader before you do it**. It is always better to ask for clarification than to risk a misconduct allegation.

## Final Remarks

This coursework is an opportunity to apply what you have learned in a practical, meaningful way. Take the time to understand the problem deeply, design your solution carefully, and test thoroughly. Work collaboratively and communicate within your group. If you encounter difficulties, seek help early from the module leader or teaching assistants during office hours or lab sessions.

Good luck and enjoy the challenge of building your Event Planner system!