# School of Information Technology, Deakin University

# SIT320 — Advanced Algorithms

## Module Six — Dynamic Programming

### ChatGPT Link

### GitHub Link

### Summary of Learning:

- Dynamic programming is a way to solve problems by **breaking them down into smaller subproblems**. The main idea is to identify the subproblems, solve them, and store the solutions. The solutions to the subproblems are then used to solve the original problem. It is well suited to solving optimisation probelms.
- Dynaic programming takes advantage of two properties:
  - **Overlapping subproblems**
    - A property where the same subproblems are solved multiple times. In DP, solutions to these are stored to avoid redundant computations.
  - **Optimal substructure**
    - A property where the optimal solution of a problem can be constructed from the optimal solutions of its subproblems.
- This can be acheived witha top-down or bottom-up approach
  - **Top-down(Memoization)** is the approach that begins with the complex problem, breaking it into subproblems, and storing solutions.
    - Storing these solution is called memoization and is acheived by storing the results of function calls so that they can be **used again later**
  - **Bottom-up(Tabulation)** is the approach that starts from the simplest subproblem and builds up the solution.
- **Advantages of DP:**
  - Efficiency: Solves problems faster by avoiding redundant calculations.
  - Versatility: Can be applied to a wide range of problems.
- **Disadvantages of DP:**
  - Memory Usage: Storing solutions to subproblems can be memory-intensive.
  - Complexity: Implementation can be complex and requires a deep understanding of the problem.
- In this module we solved three problems; Running Up Stairs, the Longest Common Subsequence (LCS) and the 0/1 Knapsack Problem.

## Running Up Stairs

- The Running Up Stairs problem is a problem where one has a staircase with n steps and can hop 1, 2, or 3 steps at a time. The goal is to find how many possible ways you can run up the stairs.
- This was a good example of a toy problem that can be solved with DP.
- I quickly realised that even a modest number of steps would result in a large number of possible combinations. 10 stairs had over 700 combinations.
- Starting with a recursive solution laid the groundwork for the DP solution. The recursive solution was slow and inefficient, but it was easy to see how it could be improved.
- **Subproblems:**
  - Given n steps, find the number of possible combinations if the last step is 1, 2, or 3.
- **Optimal Solution:**
  - The number of possible combinations is the sum of the number of possible combinations if the last step is 1, 2, or 3.
- I used a very simple algorithm to solve this problem. It was very close to the recursive solution but used a list to store the solutions to the subproblems. The list was checked to see if the solution had already been calculated. If it had, the solution was returned. If not, the solution was calculated and stored in the list.

## Longest Common Subsequence

- A classical problem in computer science and bioinformatics. A few common examples are spell checkers, plagiarism checkers, and DNA sequence analysis.
- Steps taken to solve the problem:
  - Understand the problem: What are the subproblems? What is the optimal solution?
    - **Subproblems:**
      - Given prefixes of the two strings, find the longest common subsequence.
      - Given prefixes of the two strings, find the length of the longest common subsequence if the last charascter is the same.
      - Given prefixes of the two strings, find the length of the longest common subsequence if the last charascter is different.
    - **Optimal Solution:**
      - The longest common subsequence of two strings is the longest sequence of characters that appear in the same order in both strings.
  - Choose DP approach: Top-down or bottom-up?

        - **Top-down:** this problem leant itself to a top-down approach. The problem was broken down into subproblems and the solutions were stored.
    - Construct solution:
        - **Backtracking** was used to construct the solution. The matrix was traversed from the bottom right to the top left. If the characters were the same, the character was added to the solution and the matrix was traversed diagonally. If the characters were different, the matrix was traversed either left or up, depending on which value was larger.
- **Complexity:**
    - Time: O(mn) where m and n are the lengths of the two strings.
    - Space: O(mn) where m and n are the lengths of the two strings.
- **Testing**
    - The algorithm was tested with various test cases. The results were as expected.

#### 0/1 Knapsack Problem

- The 0/1 Knapsack Probelm takes a set of n items with a wieght and value and aims to **find the maximum value** that can be put into a knapsack of **capacity W**.
- This problem is a good example of a real-world problem that can be solved with DP. My imagination took me to thinking of a **thief trying to steal the most valuable items** from a house, but only being able to carry a certain weight - the size of their knapsack.
- Ultimately the knapsack problem solves the **scarse resource allocation problem**. Other such applications are simple household budgeting, where one decides how to allocate their money to various expenses.
- It uses the same basic ideas of a **top-down** dynamic programming approach as the LCS problem.
- **Sub Problems**
    - Compute the value of the optimum solution with the first i items and capacity W.
- **Optimal Sub-Structure**
    - **Case 1**: The optimal solution excludes the last item. The optimal solution is the maximum value that can be put into a knapsack of capacity W using the first n-1 items.
    - **Case 2**: The optimal solution includes the last item. The optimal solution is the maximum value that can be put into a knapsack of capacity W using the first n-1 items plus the value of the last item. This is sligtly more complicated as the weight of the last item must be less than or equal to the capacity of the knapsack.
- **Memoization & Reconstruction of the Solution**
    - The solutions to the subproblems are **stored in a matrix**.
    - The matrix is traversed from the **bottom right to the top left**. (If

    the matrix if filled top left to bottom right.)
  - If the item was **included**, the item was **added to the solution** and the matrix was **traversed diagonally**.
  - If the item was **not included**, the matrix was **traversed up**.
- **Complexity**
  - Time: **O(nW)** where n is the number of items and W is the capacity of the knapsack.
  - Space: **O(nW)** where n is the number of items and W is the capacity of the knapsack.
- **Testing**
  - The algorithm was tested with various test cases. The results were as expected.

## Conclusion

- Dynamic programming uses a three step proicess to solve problems:
  - **Identify subproblems**
  - **Solve and store subproblems**
  - **Reconstruct the final solution**
- Reflecting back on my coding I realised the solutions were not long, in terms of lines of code. However, they were complex and required a deep understanding of the problem. I found it difficult to get started with the problems, but once I had a basic understanding of the problem, I was able to break them down into subproblems and come up with a solution. I used ChatGPT to help me form my understanding of the problems and come up with a framework to solve them. I have noticed myself becoming more optimistic about learning complex topics. It was interesting to notice that when I first opened up the readings I was somewhat confused, but as I read more, coded partial solutions, tested and refactored, ideas began to form and solutions crystallised. I am looking forward to learning more about dynamic programming and applying it to more complex problems.

## Reading

- **Algorithms Illuminated - Part 3**: Greedy Algorithms and Dynamic Programming
  - Chapter 13: Introduction to Greedy Algorithms
  - Chapter 16: Introduction to Dynamic Programming
- **Introduction to Algorithms**: Cormen, Leiserson, Rivest, Stein
  - Chapter 15: Dynamic Programming
  - Chapter 16: Greedy Algorithms