COMP3121/9101
Algorithm Design

Tutorial 6
Dynamic Programming

Before the Tutorial

Before coming to the tutorial, try and answer these discussion questions yourself to get a firm understanding of how you're pacing in the course. No solutions to these discussion points will be officially released, but feel free to discuss your thoughts on the forum. Your tutor will give you some comments on your understanding.

- For a problem to be solved effectively with dynamic programming, there are primarily two characteristics that a problem should exhibit.
  - The first characteristic is that the problem should have *overlapping* subproblems. Briefly outline why such a problem would require overlapping subproblems for dynamic programming to an efficient paradigm to solve the problem.
  - The second characteristic is that the problem should exhibit an optimal substructure. Briefly explain why such a problem would also require the property of optimal substructure.
- There are typically two ways to construct the recursion to a dynamic programming solution: the *top-down* approach and the *bottom-up* approach. Outline how each of these techniques are used to solve the subproblems.
- What are the primary differences between a greedy and a dynamic programming solution?
- Read through the problem prompts and think about how you would approach the problems.

Tutorial Problems

Problem 1. A subset $I \subseteq \{1, \ldots, n\}$ is said to be *independent* if I does not contain any pair of consecutive integers. Let A be an array of n distinct positive integers. Our task is to find the maximum sum of the values that come from an *independent subset* I of indices.

For example, consider the array $A = [2, 4, 3, 5]$. An example of an independent subset of indices is $\{1, 4\}$ with sum $A[1] + A[4] = 2 + 5 = 7$, while the independent subset of indices with maximum sum is $\{2, 4\}$ with sum $A[2] + A[4] = 4 + 5 = 9$. Therefore, we return 9.

(a) Consider the following greedy method: *choose the index corresponding to the maximum element and remove its neighbours. Repeat this process until there are no more indices left to choose. Return the sum of the values at these indices.*

Provide an array with suitable *distinct* values that show that this greedy method does not always return the maximum sum.

(b) We will now design a dynamic programming algorithm to solve this problem. Formulate a suitable subproblem.

(c) State and justify the recurrence relation.

- Based on your subproblem formulation, how do previously defined subproblems help to solve the current subproblem?

(d) State the base case(s) and the final solution.

- In what order should we be evaluating our subproblems?
- What is the final solution?

(e) Analyse the running time of your dynamic programming algorithm.

Problem 2. In this problem, we will design an algorithm that counts the number of ways to form a bit string of length n such that the 0 bits are always in groups of length k. To understand this problem statement, if $n = 3$ and $k = 2$, then possible strings are 111, 100, 001 since each string has the 0 bits occurring in groups of 2. Another example is: $n = 5$, $k = 2$, with the possible strings being 11111, 11100, 11001, 10011, 00111, 10000, 00100, 00001.

Design an $O(n)$ algorithm that counts the number of strings such that the 0 bits are in groups of length k.

- Follow the scaffold from problem 1.
- You may have multiple base cases. Carefully consider how these base cases are constructed.
- How many subproblems do you have? What is the time complexity to compute each subproblem?

Problem 3. You are emailed a text file containing important meeting notes. Unfortunately, the text file was corrupted and all of the white spaces of the text file has been removed. Therefore, all you see in the text file is a string of characters without any spaces. Fortunately, your company is very good at keeping tabs with words that are mentioned during meetings and have devised a lookup function. The lookup function acts as an oracle, and returns "true" if and only if the string is a valid word; that is, given a word `w`, `lookup(w)` returns `true` if and only if `w` is a valid word.

Given a string s of n characters, design an $O(n^2)$ algorithm that determines whether s forms a valid text by placing spaces in between characters to form a string of valid words.

After the Tutorial

After your allocated tutorial (or after having done the tutorial problems), review the discussion points. Reflect on how your understanding has changed (if at all).

- In your own time, try and attempt some of the practice problems marked [K] for further practice. Attempt the [H] problems once you're comfortable with the [K] problems. All practice problems will contain fully-written solutions.
- If time permits, try and implement one of the algorithms from the tutorial in your preferred language. How would you translate high-level algorithm design to an implementation setting?