UNIVERSITY COMPUTER SCIENCE • EXERCISE SHEET

10.4 Dynamic programming

Name: _____ Class: _____ Date: _____ **Total: 24 marks**

KEY WORDS DP table DP • dependency order

Objective

Build DP tables by hand and price the memoisation win.

1 Key knowledge and worked examples

■ **The design**

Recurrence + base cases + fill order. Coin table (1, 3, 4): $dp[i] = 1 + \min(dp[i - c])$ over usable coins ✓.

■ **Worked example: the win**

Naive fib(30): 2,692,537 calls; memoised: 29 subproblems ✓. Knapsack capacity 7, items (1,1)(3,4)(4,5)(5,7): $dp[7] = 9$ ✓.

2 Practice

2.1 Naive fib(5) makes how many calls (count the root)? [3]

2.2 Coin table (1, 3, 4): what is $dp[5]$? [3]

2.3 Name the three parts of a DP design. [3]

3 Exam-style questions

3.1 Coin system 1, 3, 4; targets 0 through 6.

Build the full table $dp[0..6]$, cell by cell, naming the chosen coin at each step. State $dp[6]$ and the coins used. Contrast with greedy's answer and name which design is correct here. [8]

3.2 Knapsack, capacity 7, items (1,1), (3,4), (4,5), (5,7).

Build the capacity table after each item (four rows of 8 cells, or the final row with derivations). State $dp[7]$ and the item set. Explain why iterating capacity downward is required for 0/1 knapsack. State the naive recursion's complexity and the table's. [7]

Solutions

2.1

- $C(5) = C(4) + C(3) + 1$ with $C(0) = C(1) = 1$: $C(5) = 15$ ✓.

2.2

- $dp[5] = 2$ (4+1) ✓.

2.3

- The recurrence, the base cases, the fill (dependency) order ✓.

3.1

- $dp[0] = 0$; $dp[1] = 1$ (coin 1); $dp[2] = 2$ (1+1); $dp[3] = 1$ (coin 3) ✓.
- $dp[4] = 1$ (coin 4); $dp[5] = 2$ (4+1); $dp[6] = 2$ (3+3) ✓.
- $dp[6] = 2$ with coins 3, 3 ✓.
- Greedy pays 3 (4+1+1) – the table is correct; no exchange argument exists for this coin system ✓.

3.2

- After (1,1): $dp = [0, 1, 1, 1, 1, 1, 1, 1]$ ✓.
- After (3,4): $dp = [0, 1, 1, 4, 5, 5, 5, 5]$ – taking both items at capacity 4 gives value $1 + 4 = 5$; after (4,5): $dp[7] = 4 + 5 = 9$ ✓.
- Final: $dp[7] = 9$ – items (3,4) and (4,5): weight 7, value 9 ✓.
- Downward iteration prevents reusing an item within the same pass (upward would allow the item twice – that is unbounded knapsack) ✓.
- Naive recursion: $\Theta(2^n)$ (include/exclude per item); the table: $\Theta(n \times \text{capacity})$ ✓.