# CS320 Homework 5

Dustin Randall

November 12, 2025

## 1 Describe how to find the minimum weight path from (0,0) to (m,n)

### 1.1 Describe a recursive formula to compute the minimum weight.

If we were to start with (0,0), it's minimum weight adds it's own number (5) to the minimum total weights of its neighbors. Each neighbor needs to do the same thing before it can be considered in the minimum comparison. The base case is when we reach (m,n), where the weight is it's own value.

$$m[i,j] = \begin{cases} w[i,j] & \text{if } i == m \text{ and } j == n \\ w[i,j] + m[i+1,j] & \text{if } i < m \text{ and } j == n \\ w[i,j] + m[i,j+1] & \text{if } i == m \text{ and } j < n \\ w[i,j] + \min(m[i+1,j], m[i,j+1], m[i+1,j+1]) & \text{if } i < m \text{ and } j < n \end{cases}$$

### 1.2 Describe the dynamic programming algorithm, and determine its runtime.

Similar to the matrix chain multiplication problem, we'll need to cache 2 pieces of information for each cell. We need to store the minimum weight to reach (m,n) from a given cell, and we need to store the next cell in the path. We start by initializing a 2D array cost[][] to be match the weight matrix. We then iterate from the bottom right corner (m,n) to the top left corner (0,0), and update cost[row][col] to be the cost[row][col] + min(cost[row+1][col], cost[row][col+1], cost[row+1][col+1]). The next array is also updated to the cell that provided the minimum cost. The runtime should be $\Theta(mn)$ because each node is visted exactly once, and the work done at each node is constant time.

### 1.3 Write an algorithm to find the actual path.

Given that we have stored cost[][] and next[][], the cost of reaching (m,n) from (0,0) is simply cost[0][0]. To create the actual path, we start at (0,0) and look up our next cell at next[0][0].

```
function findPath(next, m, n) {
    i = j = 0
    while(i != m and j != n) {
        print(i, j)
        (i, j) = next[i][j]
    }
}
```

The runtime of this algorithm is linear to the number of steps in the path. The upper bound is the manhattan distance from (0,0) to (m,n) which is m+n. So we can say the runtime is $O(m + n)$.

# 2 Find 3 dynamic programming problems not discussed in class.

## 2.1 Change Making Problem.

### 2.1.1 Description

How many ways can we make change for $1,000,000 using [.01, .05, .1, .25, .5, 1, 5, 10, 20]?

### 2.1.2 Why recursive is inefficient

The recursive solution would try every possible combination of coins, and would have exponential time complexity.

### 2.1.3 Why does dynamic programming work

Dynamic programming is viable because the majority of the work is repeated calculations of subproblems.

### 2.1.4 Sources

- https://www.geeksforgeeks.org/dsa/coin-change-dp-7/

## 2.2 Longest Common Substring.

### 2.2.1 Description

Given two strings, find the longest common substring.

### 2.2.2 Why recursive is inefficient

Checking every possible substring combination takes exponential time.

#### 2.2.3 Why does dynamic programming work

The problem can be broken down into smaller subproblems, where we build up larger and larger substrings from smaller ones.

#### 2.2.4 Sources

- https://www.geeksforgeeks.org/dsa/longest-common-substring-dp-29/

### 2.3 Painter's Partitioning Problem.

#### 2.3.1 Description

Given n boards of different lengths, and k painters, find the minimum time to paint all of the boards where each painter can only paint adjacent boards.

#### 2.3.2 Why recursive is inefficient

The recursive solution would try every possible partitioning of boards among painters, leading to exponential time complexity.

#### 2.3.3 Why does dynamic programming work

The problem can be broken down into smaller subproblems by considering the optimal way to partition the first i boards among j painters.

#### 2.3.4 Sources

- https://www.geeksforgeeks.org/dsa/painters-partition-problem/