Dynamic Programming

University College Cork

Konrad Dagiel

April 2023

Abstract

A section on what this thesis is about

Declaration

Declaration of Originality.
In signing this declaration, you are conforming, in writing, that the submitted work is entirely your own original work, except where clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award.
I hereby declare that:
• this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;
• with respect to my own work: none of it has been submitted at any educational institution contributing in any way to an educational award;
• with respect to another's work: all text, diagrams, code, or ideas,
whether verbatim, paraphrased or otherwise modified or adapted, have
been duly attributed to the source in a scholarly manner, whether
from books, papers, lecture notes or any other student's work, whether
published or unpublished, electronically or in print.
Signed: Konrad Dagiel
Date: 17/04/2024

Acknowledgements

Dynamic Programming was originally proposed by Richard Bellman in the 1950s. In Bellman's dynamic programming, problems are typically represented using states, actions, and transitions between states. Each state represents a specific configuration or situation in the problem domain, and actions define the possible decisions or choices that can be made from each state. This representation allows for a more structured approach to problem-solving and optimization. Throughout this notebook when we refer to Dynamic Programming, we refer to a more general, modernized version of Dynamic Programming.

Contents

1 Introduction **5**

2 Analysis **6**

3 Design **7**

3.1 Introduction to Dynamic Programming 7

3.2 The Coin Change Problem 7

3.2.1 Greedy Approach to the Coin Change Problem 8

3.2.2 Optimality of the Greedy Approach to the Coin Change Problem 8

3.2.3 Correctness of the Greedy Approach to the Coin Change Problem 8

3.2.4 Brute Force Approach to the Coin Change Problem 9

3.2.5 Complexity Analysis of the Brute Force Approach to the Coin Change Problem . . 9

3.2.6 Memoization Approach to the Coin Change Problem 10

3.2.7 Complexity Analysis of the Memoization Approach to the Coin Change Problem . 10

3.2.8 Tabulation Approach to the Coin Change Problem 11

3.2.9 Complexity Analysis of the Tabulation Approach to the Coin Change Problem . . 12

3.3 Longest Increasing Subsequence 12

Chapter 1

Introduction

Chapter 2

Analysis

Chapter 3

Design

3.1 Introduction to Dynamic Programming

Dynamic Programming has many definitions, but can be summarized as method of breaking down a larger problem into sub-problems, so that if you work through the sub-problems in the right order, building each answer on the previous one, you eventually solve the larger problem. The two attributes a problem needs to have in order to be classified as a dynamic programming problem are as follows:

Definition 3.1.1 (Optimal Substructure)**.** A problem is said to have optimal substructure if an optimal solution to the problem can be deduced from optimal solutions of some or all of its subproblems.

Definition 3.1.2 (Overlapping Subproblems)**.** A problem is said to have overlapping subproblems if the problem can be broken down into subproblems which can be reused several times or a recursive algorithm would solve the same subproblem more than once resulting in repeated work. (If the subproblems do not overlap, the algorithm is categorized as a "divide and conquer" algorithm rather than a dynamic programming algorithm.)

Once we have deduced that a problem has both of these properties, we can use dynamic programming principles in order to solve the problem in an efficient manner. When solving a dynamic programming problem, it is common to start by implementing a brute force solution which explores all subproblems and returns a solution. We can then extend our solution to use a cache to store the results of any subproblems encountered, such that when the subproblem is encountered again we do not need to re-compute the result, instead we can simply look up the cache in constant time. This is known as "memoization", or "top-down dynamic programming". We can then look for any patterns in the cache table which, given an initialization (usually the base case of the recursive solution), would allow us to compute the values stored in the cache without ever traversing the decision tree of the problem itself. This is known as "tabulation", or "bottom-up dynamic programming". In order to demonstrate this, we will use a simple problem called The Fibonacci Problem.

The Fibonacci Problem

3.2 The Coin Change Problem

Problem Statement: Given a list of denominations of coins D and an integer amount a, compute the minimum amount of coins (where each coin's denomination $\in D$) needed to sum exactly to the given amount a.

Input: An integer array D of possible coin denominations, and an integer amount a.

Output: An integer r, which represents the minimum amount of coins with denominations $\in D$ needed in order to sum exactly to a. If this cannot be done, return -1.

Example: For:

$D = [1, 5, 10, 20]$

$a = 115$

$r = 7$

Explanation: The minimum amount of coins with denominations in D needed to sum to a is 7.

These coins are: $[20, 20, 20, 20, 20, 10, 5]$

3.2.1 Greedy Approach to the Coin Change Problem

Algorithm 1 shows a greedy approach to the coin change problem.

Algorithm 1: Greedy Approach to the Coin Change Problem

```
Input: List of denominations of coins $D$ and an amount $a$
Output: $r$, The minimum number of coins required to make change for $a$
Sort $D$ in ascending order;
$r \leftarrow 0$;
$total \leftarrow 0$;
while $total < a$ do
    if $|D| = 0$ then
        return -1;
    if $total + D[-1] > a$ then
        $D$.pop();
    else
        total $\leftarrow$ total $+ D[-1]$;
        $r \leftarrow r + 1$;
return $r$
```

In Algorithm 1, we choose the coin with the largest value which will not make the total exceed a.

3.2.2 Optimality of the Greedy Approach to the Coin Change Problem

This algorithm is not optimal, and we can prove this by counter-example. Take:

$$D = [5, 4, 3, 2, 1], a = 7$$

Given these inputs, the greedy result is: $r1 = 3$ ([5,1,1]).
The optimal solution for these inputs is: $r2 = 2$ ([4,3]).
We see that $r1 > r2$, meaning the greedy approach does not find the miminized solution.

3.2.3 Correctness of the Greedy Approach to the Coin Change Problem

The algorithm is also not correct, and we can prove this by another counter-example. Take:

$$D = [4, 3], a = 6$$

Given these inputs, the greedy result is: $r1 = -1$ ([4]).
The optimal solution for these inputs is: $r2 = 2$ ([3,3]).
We see that the greedy approach fails when a solution is indeed possible, as shown by $r2$.
Since we have shown that the greedy approach is neither correct nor optimal, we move on to the brute force solution.

3.2.4 Brute Force Approach to the Coin Change Problem

The brute force approach to the coin change problem involves generating all possible coin combinations, and checking if any of them sum exactly to a. Of the ones that do, we return the minimum length. To try all possible coin combinations, we can subtract each coin denomination $c \in D$ from a, as long as $a - c >= 0$. We can repeat this step for each result obtained from this calculation (replacing a with the intermediate result), until all possible coin combinations are explored. We can keep track of the shortest path through the resulting tree which has a leaf value of 0, to avoid storing the entire tree in memory. We return the length of the shortest path as r.

A sample python implementation is shown in Figure 3.1.

```python
def coin_change_bf(D, a):
    def dfs(a):
        if a == 0:
            return 0
        if a<0:
            return float('inf')
        return min([1+dfs(a-c) for c in D])
    minimum = dfs(a)
    return minimum if minimum < float("inf") else -1
```

Figure 3.1: Coin Change Brute Force Python Implementation

3.2.5 Complexity Analysis of the Brute Force Approach to the Coin Change Problem

Time Complexity: For the worst case scenario, let's assume each coin denomination $c \in D < a$ such that each node which is not a leaf node has $|D|$ children. This means we have $|D|$ recursive calls at the first level, $|D|^2$ at the second level, $|D|^n$ at the n'th level.

The total number of recursive calls in this scenario is $|D| + |D|^2 + ... + |D|^a$ which is $O(|D|^a)$.

Therefore the time complexity is $O(|D|^a)$. This is because at each step, there are $|D|$ choices (coin denominations) to consider, and the recursion depth is at most a.

Space Complexity: We do not store the entire tree in memory, only the current path.

The space complexity is determined by the maximum depth of the recursion stack. In the worst case, the recursion depth is equal to the target amount a. Therefore, the space complexity is $O(a)$.

Overall: Total:

Time Complexity: $O(|D|^a)$

Space Complexity: $O(a)$

3.2.6 Memoization Approach to the Coin Change Problem

We can trivially see that the problem has the optimal substructure property. In the brute force algorithm, we have a chance to arrive at a value multiple times. This is because an intermediate value can be made up of different combinations of coins (eg, 3 can be made up of (2,1) or (1,1,1)). This demonstrates the overlapping subproblems property. Therefore, we can use memoization to prevent repeated calculations of the optimal number of coins needed to make up a given sub-amount.
For every path in the search tree, we can store intermediate results in a table, so that the next time we arrive at a value, eg. 3, we don't have to repeat the work in finding the minimum amount of extra coins needed to sum to a. Instead we can simply look in the table with a constant time lookup. This optimization reduces search time greatly, as seen in subsection 3.2.7.
A sample python implementation is shown in figure 3.2.

```python
def coin_change_memo(D, a):
memo = {}
def dfs(a):
    if a == 0:
        return 0
    if a < 0:
        return float('inf')
    if a in memo:
        return memo[a]

    memo[a] = min([1+dfs(a-c) for c in D])
    return memo[a]

res = dfs(a)
return res if res < float("inf") else -1
```

Figure 3.2: Coin Change Memoization Python Implementation

3.2.7 Complexity Analysis of the Memoization Approach to the Coin Change Problem

Time Complexity: Each unique subproblem is evaluated once, and the next time it is encountered it is retrieved from the memoization table with a constant time lookup[1]. As there are $|D| * a$ unique subproblems in the worst case[2], the time complexity to solve all of them is $O(|D| * a)$.

Space Complexity: We need to store the *memo* table in memory. The memoization table is represented by a lookup data structure where the keys range from 0 to a, representing the solution to each unique subproblem. Hence, the memory required to store the table is of order $O(a)$.

Overall: Total:

Time Complexity: $O(|D| * a)$

Space Complexity: $O(a)$

[1]Python dictionary lookups have an expected $O(1)*$ time complexity.
[2]$|D|$ constant time subtractions from any intermediate value v where $0 \leq v \leq a$.

3.2.8 Tabulation Approach to the Coin Change Problem

Instead of doing a dfs to fill in the memo table, we can calculate the values in the memo table directly, and extract the answer from there. We will call the *memo* table dp, as we are no longer doing memoization, but tabulation. $dp[i]$ represents the minimum amount of coins needed to get the amount i. Consider the example:

$$D = [5, 4, 3, 1], a = 7$$

We initialize each $dp[i]$ to contain infinity. We know that $dp[0] = 0$ as it takes 0 coins to add up to an amount of 0. We can initialize this in our table. Now we can deduce $dp[1], dp[2], ...dp[a]$. $dp[a]$ will contain r. To get $dp[i]$, we will look at each coin $c \in D$ in sequence. For each $c \in D$, we take $i - c$ to get t, and look for $dp[t]$ if it exists. Our intermediate result is $1 + dp[t]$. If this result is less than the current $dp[i]$ and is not negative, we update $dp[i] \leftarrow 1 + dp[t]$.
The logic of this is that the amount of coins it takes to make the amount $dp[i]$ is the amount of coins it takes to make the amount $dp[t]$ plus one. The logic is demonstrated with the examples:

Example 1: Calculating $dp[1]$

$$dp[0] = 0$$

$$dp[1] = \infty$$

$$dp[2] = \infty$$

$$dp[3] = \infty$$

$$dp[4] = \infty$$

$$dp[5] = \infty$$

$$dp[6] = \infty$$

$$dp[7] = \infty$$

To calculate $dp[1](i = 1)$:
For $c \in D = [5, 4, 3, 1]$
$t = i - c = -4$, ignore because negative.
$t = i - c = -3$, ignore because negative.
$t = i - c = -2$, ignore because negative.
$t = i - c = 0$
Look up the value of $dp[t] = 0$.
Now we take $1 + dp[0] = 1$.
This means a possible solution to $dp[1]$ is 1.
Since $1 < \infty$, we update $dp[1] \leftarrow 1$

Example 2: Calculating $dp[7]$

$$dp[0] = 0$$

$$dp[1] = 1$$

$$dp[2] = 2$$

$$dp[3] = 1$$

$$dp[4] = 1$$

$$dp[5] = 1$$

$$dp[6] = 2$$

$$dp[7] = \infty$$

To calculate $dp[7](i = 7)$:
For each $c \in D = [5, 4, 3, 1]$:
$t = i - c = 2$, $1 + dp[2] = 3$, $3 < \infty$, update $dp[7] \leftarrow 3$
$t = i - c = 3$, $1 + dp[3] = 2$, $2 < 3$, update $dp[7] \leftarrow 2$
$t = i - c = 4$, $1 + dp[4] = 2$, $2 = 2$, ignore.
$t = i - c = 6$, $1 + dp[6] = 3$, $3 > 2$, ignore.
We conclude that the minimum solution to $dp[7]$ is 2, achieved by adding a 4 coin to $dp[3]$, which is achieved by adding a 3 coin to $dp[0]$
A sample python implementation is shown in Figure 3.3.

```python
    def coin_change_dp(D,a):
        dp=[float('inf')] * (a + 1)
        dp[0] = 0

        for i in range(1, a+1):
            for c in D:
                t = i - c
                if t >= 0:
                    dp[i] = min(dp[i], 1+dp[t])

        return dp[a] if dp[a] != float('inf') else -1
```

Figure 3.3: Coin Change Tabulation Python Implementation

3.2.9 Complexity Analysis of the Tabulation Approach to the Coin Change Problem

Time Complexity: For the worst case scenario, we need to iterate for all $0 \leq i \leq a$. And for each i, we need to iterate over each coin $c \in D$. All other operations within the loops are constant time lookups and subtractions, so the time complexity is $O(|D| * a)$

Space Complexity: The space complexity is determined by the size of the dp array. This array is always of size $a + 1$. Therefore the space complexity is $O(a)$

Overall: Total:

Time Complexity: $O(|D| * a)$

Space Complexity: $O(a)$

3.3 Longest Increasing Subsequence

```
# Longest Increasing Subsequence

Given an array nums, return the length of the longest strictly increasing subsequence. A subsequence
```

Example: nums = [2,5,3,7,101,18]

Output: 4

Explanation: The subsequence [2,5,7,101] is the longest increasing subsequence, with length 4.

Much like coin change, this problem appears trivial at first glance. One may attempt to be greedy as

## Longest Increasing Subsequence Greedy Approach

```
Input: nums

r := 0

index := 0

cur := nums[0]

While index < |nums|:

    if nums[index] > cur:

        r += 1

        cur := nums[index]

    index +=1

Return r
```

In this greedy algorithm we iterate through nums keeping track of the current max value enountered,

### Optimality of the Greedy approach

This algorithm is not optimal however, and we can prove this by counter-example.

A counter example is nums = [10,9,2,5,3,7,101,18]

Given these inputs, the greedy result is: r1 = 2 ([10,101])

An optimal solution for these inputs is: r2 = 4 ([2,3,7,101,18])

We see that r1 > r2, meaning the greedy approach does not find the maximised solution.

We therefore need a more sophisticated approach.

## Longest Increasing Subsequence Brute Force

We can try a brute force approach, where we start at index 0, and for each index choose wether we sh

This will generate all possible increasing subsequenes.

We keep track of the longest increasing subsequence length, and return it.

Below is an implementation of this algorithm.

```
def length_of_lis_bf(nums):
    def dfs(prev_index, current_index):
        # Base case: reached the end of the sequence
        if current_index == len(nums):
            return 0

        # Case 1: Exclude the current element
        exclude_current = dfs(prev_index, current_index + 1)

        # Case 2: Include the current element if it is greater than the previous one
        include_current = 0
        if prev_index < 0 or nums[current_index] > nums[prev_index]:
            include_current = 1 + dfs(current_index, current_index + 1)

        # Return the maximum length of the two cases
        return max(exclude_current, include_current)

    # Start the recursion with initial indices (-1 represents no previous index)
    return dfs(-1, 0)

print(length_of_lis_bf([10,9,2,5,3,7,101,18]))
```

## Longest Increasing Subsequence Brute Force Complexity Analysis

Let n be the length of nums.

Time Complexity:

For the worst case scenario, There are n indices to consider. There are two subtrees at each decisio

This brings the time complexity to O(2^n)

Space Complexity:

The space complexity is determined by the recursion depth.

Therefore the space complexity is O(n)

Overall:

Time Complexity: O(2^n)

Space Complexity: O(n)

## Longest Increasing Subsequence Memoization

We can use memoization to avoid repeating subproblems, such as when we are deciding wether the next

Before proceeding with the recursive calls, the function checks if the result for the current combin

```
def length_of_lis_memo(nums):
    if not nums:
        return 0

    memo = {}  # Memoization dictionary to store computed results

    def dfs(prev_index, current_index):
        if current_index == len(nums):
            return 0

        if (prev_index, current_index) in memo:
            return memo[(prev_index, current_index)]

        exclude_current = dfs(prev_index, current_index + 1)

        include_current = 0
        if prev_index < 0 or nums[current_index] > nums[prev_index]:
            include_current = 1 + dfs(current_index, current_index + 1)


        # Save the result in the memoization dictionary
        memo[(prev_index, current_index)] = max(include_current, exclude_current)

        return memo[(prev_index, current_index)]

    return dfs(-1, 0)

print(length_of_lis_memo([10,9,2,5,3,7,101,18]))
```

## Longest Increasing Subsequence Memoization Complexity Analysis

Let n be the length of nums.

Time Complexity:

For each unique combination of (prev_index, current_index), the algorithm either calculates the resu

The algorithm explores all combinations of prev_index and current_index. There are at most n choices

Space Complexity:

The space complexity is increased to O(n^2), as the memo table needs to store all n^2 combinations i

Overall:

Time Complexity: O(n^2)

Space Complexity: O(n^2)

## Longest Increasing Subsequence Tabulation

We can use tabulation to build a table from which we can deduce the result, similar to the coin chan

We know that starting at the last index will result in an increasing subsequence of length 1. We can

We create a table called dp of size len(nums), where dp[i] represents the longest increasing subsequ

Lets take the example nums = [1,2,4,3]

We initialize dp[3] to 1, as the longest increasing subsequence starting at index 3 is 1.

Consider nums[2] = 4

We can either take nums[2] by itself, or include nums[2] in any subsequence at any index that comes

Since including it would not result in an increasing subsequence, we must exclude it, so dp[2] = 1

Now Conisder nums[1] = 2

We can either take it by itself or include it in any subsequence at any index that comes after it. I

We choose the option which maximizes the value of dp[1], which is 1+dp[2] (or equally 1+dp[3]) = 2

So for dp[i], by the same logic, we simply put max(1,1+dp[j1],1+dp[j2],1+dp[j3]...) (only include 1+

```
def length_of_lis_dp(nums, printTable = False):
    dp = [1] * len(nums)

    for i in range(len(nums)-1,-1,-1):
        for j in range(i+1,len(nums)):
            if nums[i] < nums[j]:
                dp[i] = max(dp[i], 1+dp[j])

    if printTable:
        print(dp)

    return max(dp)

print(length_of_lis_dp([10,9,2,5,3,7,101,18], printTable=True))
```

## Longest Increasing Subsequence Tabulation Complexity Analysis

Let n be the length of nums.

Time Complexity:

For the worst case scenario, we need to perform a double nested iteration over nums.

All other operations within the loops are constant time lookups and max(a,b), so the time complexity

Space Complexity:

The space complexity is determined by the size of the dp array. This array is always of size n.

Therefore the space complexity is O(n)

Overall:

Time Complexity: O(n^2)

Space Complexity: O(n)