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published or unpublished, electronically or in print.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

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.

3.1.1 Demonstration of Dynamic Programming Principles using the Fibonacci Problem

Problem Statement: Compute the n'th number in the Fibonacci sequence.[1]

Input: A positive integer n.

[1]Where $fib(1) = 1$, $fib(2) = 1$ and $fib(n) = fib(n-1) + fib(n-2)$

Output: An positive integer $fib(n)$.

Example: For:

$n = 7$

$fib(n) = 13$

Explanation: The Fibonacci sequence is as follows: 1,1,2,3,5,8,13,...
We can see that the 7th number in the sequence is 13.

3.1.2 Brute Force Approach to The Fibonacci Problem

We start with a brute force approach which will use recursion. The base cases are $fib(1) = 1$ and $fib(2) = 1$. By definition, the recursive case is $fib(n) = fib(n-1) + fib(n-2)$. An implementation of the brute force solution is given below. A sample python implementation is shown in Figure 3.1.

```python
    def fib_bf(n):
        if n <=2: return 1
        return fib_bf(n-1) + fib_bf(n-2)
```

Figure 3.1: Fibonacci Brute Force Python Implementation

In order to understand just how inefficent this approach is, consider the calculation of fib(20). The brute force approach will split this calculation into the calculation of fib(19) + fib(18). Now, to calculate fib(19), we split it into fib(18) + fib(17), and to calculate fib(18) we split it into fib(17) + fib(16). Since the original problem was to calculate fib(19) + fib(18), and we need fib(18) to calculate fib(19), the calculation of fib(18) is repeated.

3.1.3 Complexity Analysis of the Brute Force Approach to the Fibonacci Problem

Time Complexity: At each step in the calculation of $fib(n)$, we make two 'branches', where one calculates $fib(n-1)$ and the other calculates $fib(n-2)$. This branching factor leads to an exponential growth in the number of function calls. The number of function calls grows exponentially with n, as each level of the tree doubles the number of function calls. Therefore the time complexity of this approach is $2 + 2^2 + 2^3 + ... + 2^n$ which is $O(2^n)$.

Space Complexity: In the brute force approach to compute Fibonacci numbers, the space complexity is influenced by the recursive calls, each of which adds a frame to the call stack. However, as the recursion progresses, some of these frames can be discarded once their corresponding Fibonacci values have been computed. Specifically, at any point during the recursion, we only need to keep track of the previous two Fibonacci numbers. Therefore, the maximum depth of the call stack at any point is at most n due to the recursion. This means that the space complexity of the brute force approach to compute Fibonacci numbers is $O(n)$.

Overall: Total:
Time Complexity: $O(2^n)$
Space Complexity: $O(n)$

3.1.4 Memoization Approach to The Fibonacci Problem

Because we have optimal substructure[2] and overlapping subproblems[3], we can make this calculation more efficient through the use of memoization. This simple adjustment involves storing a $(key, value)$ table called *memo*, where the key is an intermediate subproblem and the value is the intermediate result of that subproblem. Now, for any subproblem, we first check if the result is in memo and if it is, we return the result of that calculation in constant time. If the subproblem is not in *memo*, we calculate the intermediate result and cache it in *memo*. A sample python implementation is shown in Figure 3.2.

```python
def fib_memo(n, memo={}):
    if n <= 2:
        return 1
    if n in memo:
        return memo[n]
    memo[n] = fib_memo(n-1, memo) + fib_memo(n-2, memo)
    return memo[n]
```

Figure 3.2: Fibonacci Memoization Python Implementation

3.1.5 Complexity Analysis of the Memoization Approach to The Fibonacci Problem

Time Complexity: Since each subproblem is only ever computed once, and any repeated subproblems are handled with a constant time table lookup, the time complexity depends only on the amount of subproblems. Since there are n possible subproblems for any given input n, the time complexity is reduced to $O(n)$

Space Complexity: The space complexity remains determined by the recursion call stack, at $O(n)$. We also have to store the memo table, which contains an integer solution to each of the n subproblems. This is also $O(n)$, giving us a total $O(2n)$ space complexity. This can be simplified to $O(n)$.

Overall: Total:
Time Complexity: $O(n)$
Space Complexity: $O(n)$

3.1.6 Tabulation Approach to the Fibonacci Problem

With the memoization approach, we saw how to compute the solution top-down, starting at $fib(n-1)+fib(n-2)$, arriving at the base cases, and working up from there. Notice that this step is unnecessary. If we can deduce $fib(3)$ from $fib(2)+fib(1)$ (both of which are given in the base case), and $fib(4)$ from $fib(3)$ and $fib(2)$, we can work bottom-up until we arrive at $fib(n)$. This reduces the space complexity from $O(2n)$ to $O(n)$, as all we need to do is store the table. It is common practice to refer to the table as *dp* in tabulation approaches.

A sample python implementation is shown in Figure 3.3.

[2]The optimal solution to fib(n-1) + the optimal solution to fib(n-2) will always be the optimal solution to fib(n).
[3]The calculation of fib(n-1) contains the calculation of fib(n-2).

```python
def fib_dp(n):
    if n <= 2:
        return 1

    dp = [0] * (n + 1)
    dp[1] = 1

    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]

    return dp[n]
```

Figure 3.3: Fibonacci Tabulation Python Implementation

Space Optimized Approach to the Fibonacci Problem

We can often save space with the tabulation approach by releasing parts of the dp table which are not in use from memory. In this case, notice that we only need $fib(n-1) and fib(n-2)$ to deduce the result of $fib(n)$. The rest of the table does not need to be stored. We can achieve this by storing just two variables, $prev$ and $curr$. For an arbitrary value k, $curr$ represents the value of $fib(k)$, $prev$ represents the value of $fib(k-1)$. We can calculate the result of $fib(n+1)$ from $prev$ and $curr$, then update $curr$ to the result, and $prev$ to what $curr$ was. Starting at $curr = 1$ and $prev = 0$ and repeating this $n-1$ times will make $curr = fib(n)$.

A sample python implementation is shown in Figure 3.4.

```python
def fib_optimized(n):
    if n <= 1:
        return n

    prev, curr = 0, 1
    for _ in range(n-1):
        prev, curr = curr, prev + curr

    return curr
```

Figure 3.4: Fibonacci Optimized Python Implementation

3.1.7 Complexity Analysis of the Tabulation Approach to the Fibonacci Problem

Time Complexity: The time complexity remains unchanged at $O(n)$.

Space Complexity: Since we are only storing two variables of constant size at a time, and there is no recursion, the space complexity of this optimized version is $O(1)$.

Overall: Total:
Time Complexity: $O(n)$
Space Complexity: $O(1)$

3.2 Dynamic Programming Summary

In summary, the "dynamic programming way of thinking" involves:

1. Creating a brute force solution.
2. Figuring out if the optimal substructure property holds.
3. Identifying the repeating and overlapping subproblems.
4. Introducing memoization to the brute force solution to eliminate repeated work.
5. Using tabulation to try to deduce the memoization table bottom-up rather than top-down.
6. Looking for ways to optimize space in the tabulation approach by reducing the size of the table.

Using the Fibonacci example, we have demonstrated the way of thinking about a problem which is dynamic programming. We have went from an $O(2^n)$ time and $O(n)$ space complexity recursive solution to an $O(n)$ time and $O(1)$ space complexity solution using dynamic programming principles. The Project Notebook contains an in depth analysis of 8 well known dynamic programming problems, followed by my own original dynamic programming problems and solutions. Where applicable, the problems analyzed contain:

1. The problem statement, and a deep explanation of the problem with examples. This may contain example greedy algorithms and proofs of why they do not actually work for the given problem.
2. A comprehensive brute force algorithm, with an implementation and complexity analysis.
3. A short informal proof of why the optimal substructure and overlapping subproblems attributes hold.
4. An explanation of how memoization is used in the problem, with an implementation and complexity analysis.
5. An explanation of how tabulation is used in the problem, with an implementation and complexity analysis.
6. An explanation of how we can space optimize the tabulation solution, with an implementation and complexity analysis.

In the Project Notebook, the tabulation approach of each of the problems comes with a *printTable* flag which, when set to *True*, displays the table which has been calculated for the specific problem.

3.3 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: $[20, 20, 20, 20, 20, 10, 5]$

3.3.1 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.

3.3.2 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.

3.3.3 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.

3.3.4 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 3.5.

```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 3.5: Coin Change Brute Force Python Implementation

3.3.5 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: $O(a)$

3.3.6 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 3.3.7.
A sample python implementation is shown in figure 3.6.

```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 3.6: Coin Change Memoization Python Implementation

3.3.7 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[4]. As there are $|D| * a$ unique subproblems in the worst case[5], 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: $O(a)$

3.3.8 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,

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

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 3.7.

```
    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 3.7: Coin Change Tabulation Python Implementation

3.3.9 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: $O(a)$

3.4 Longest Increasing Subsequence

Problem Statement: Given an array, return the length of the longest strictly increasing subsequence in the array. A sequence is said to be increasing if and only if $x1 < x2 < x3 < ...xn$. A subsequence does not have to be contiguous.

Input: An integer array $nums$.

Output: An integer lis, the length of the longest increasing subsequence of $nums$.

Example: For:

$nums = [2, 5, 3, 7, 101, 18]$

$lis = 4$

Explanation: The subsequence $[2, 5, 7, 101]$ is the longest increasing subsequence in *nums*, and has length 4.

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

3.4.1 Greedy Approach to Longest Increasing Subsequence

Algorithm 2: Greedy Approach to Longest Increasing Subsequence

```
Input: An integer array $nums$.
Output: An integer $lis$, the longest increasing subsequence in $nums$.
$lis := 0$;
$index := 0$;
$cur := nums[0]$;
while $index \leq |nums|$ do
    if $nums[index] > cur$ then
        $lis += 1$;
        $cur := nums[index]$;
    $index += 1$;
return $lis$
```

In Algorithm 2 we iterate through nums keeping track of the current max value enountered, incrementing our result each time a new larger value is encountered.

3.4.2 Optimality of the Greedy Approach to Longest Increasing Subsequence

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

$$nums = [10, 9, 2, 5, 3, 7, 101, 18]$$

Given these inputs, the greedy result is: $r1 = 2([10, 101])$.
An optimal solution for these inputs is: $r2 = 4([2, 3, 7, 101])$.
We see that $r1 > r2$, meaning the greedy approach does not find the maximised solution.

We therefore need a more sophisticated approach.

3.4.3 Brute Force Approach to Longest Increasing Subsequence

We can try a brute force approach, where we start at $nums[0]$, and for each $i \in nums$ generate two subsequences, one where we exclude it from the subsequence and one where we include it in the subsequence if $nums[current\_index] > nums[prev\_index]$ (So that we ensure each subsequence generated is strictly increasing). We keep track of the *prev_index*, which represents the last index we included in the result, and the *current_index*, which is the index for which we are making the choice. This will generate all possible increasing subsequenes. We keep track of the length of the longest increasing subsequence, and return it once all increasing subsequences are explored.

A sample python implementation is shown in Figure 3.8.

```python
def 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)
```

Figure 3.8: Longest Increasing Subsequence Brute Force Python Implementation

3.4.4 Complexity Analysis of the Brute Force Approach to Longest Increasing Subsequence

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 decision, one where we include the current index, and one where we do not. This brings the time complexity to $O(2^n)$.

Space Complexity: The space complexity is determined by the recursion depth, as once we explore a path in the recursion tree, we can release it from memory when we go to the next path. Therefore the space complexity is $O(n)$.

Overall: Total:
Time Complexity: $O(2^n)$
Space Complexity: $O(n)$

3.4.5 Memoization Approach to Longest Increasing Subsequence

Notice that for an aribitrary longest increasing subsequence of length k ending at index i, if there exists exactly one number which can extend the sequence, the length of the total subsequence is guaranteed to be $k + 1$. This shows the optimal substructure property. Also, when looking for subsequences at index i, we must re-compute all of the subsequences at index $i - 1$. This shows the overlapping subproblems property. We can use memoization to avoid repeating subproblems, such as when we are deciding wether the next element should be added or not for multiple subsequences ending in the same element. The memoization approach goes as follows: We initialize an empty dictionary called *memo*, the keys of which are constructed by making a tuple of $(prev\_index, current\_index)$. Before proceeding with the recursive calls, the function checks if the result for the current combination of $prev\_index$ and $current\_index$ is already computed and stored in the *memo* dictionary. If it is, the stored result is returned immediately. This optimization allows the algorithm to avoid repeating work, speeding up the runtime significantly. Notice that memoization is not a space efficient approach to solving subsequence problems, as there is usually a lot of subproblems to store. In the case of longest increasing subsequence, memoization actually has a worse space complexity than the brute force approach.

A sample python implementation is shown in Figure 3.9.

```python
def lis_memo(nums):
    if not nums:
        return 0

    memo = {}

    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)
```

Figure 3.9: Longest Increasing Subsequence Memoization Python Implementation

3.4.6 Complexity Analysis of the Memoization Approach to Longest Increasing Subsequence

Let n be the length of nums.

Time Complexity: For each unique combination of $(prev\_index, current\_index)$, the algorithm either calculates the result or looks it up in the *memo* table. Each subproblem is calculated only once. The algorithm explores all combinations of $prev\_index$ and $current\_index$. There are at most n choices for $current\_index$ and, in the worst case, n choices for $prev\_index$ for each $current\_index$. Therefore, the total number of unique subproblems is $O(n^2)$.

Space Complexity: The space complexity is increased to $O(n^2)$, as the *memo* table needs to store all n^2 combinations of $prev\_index$ and $current\_index$ in the worst case.

Overall: Total:
Time Complexity: $O(n^2)$
Space Complexity: $O(n^2)$

3.4.7 Tabulation Approach to Longest Increasing Subsequence

We can use tabulation to build a table from which we can deduce the result, similar to the Coin Change Problem. We know that a subsequence which consists of only the last index will result in an increasing subsequence of length 1. We can work backwards, for the second last, third last ect.. deciding if including that element will result in a longer increasing subsequence or not, and storing the longest possible increasing subsequence starting at each index until we reach index 0. We create a table called dp of size $|nums|$, where $dp[i]$ represents the longest increasing subsequence starting at index i in nums. Lets take the example $nums = [1, 2, 4, 3]$. We initialize $dp[3] \leftarrow 1$, as the longest increasing subsequence starting at index 3 is 1. Now, consider $nums[2] = 4$ We can either take $nums[2]$ as a subsequence by itself, or include $nums[2]$ in any subsequence at any index that comes after it (as long as it maintains the property of an increasing subsequence). Including it would make $dp[2] = 1 + dp[3]$, Excluding it would make $dp[2] = 1$. Since including it would not result in an increasing subsequence[6], 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. Including it would make $dp[1] \leftarrow max(1 + dp[2], 1 + dp[3])$, and taking it by itself would make $dp[1] = 1$. We choose the option which maximizes the value of $dp[1]$, which is $1 + dp[2]$[7] $= 2$. So for $dp[i]$, by the same logic, we simply put $max(1, 1 + dp[j1], 1 + dp[j2], 1 + dp[j3]...)$[8]. (only include $1 + dp[jx]$ in the max function if $nums[i] < nums[jx]$, to maintain increasing subsequence property). This will give us a table where $dp[i]$ contains the longest increasing subsequence of $nums$ where the first number is $nums[i]$. We can then simply return $max(dp)$.

A sample python implementation is shown in Figure 3.10.

```python
def lis_dp(nums):
    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])

    return max(dp)
```

Figure 3.10: Longest Increasing Subsequence Tabulation Python Implementation

[6]The resulting sequence $[4, 3]$ is not increasing.
[7]Or, equally, $1 + dp[3]$.
[8]Where jx is the index which comes x places after i.

3.4.8 Complexity Analysis of the Tabulation Approach to Longest Increasing Subsequence

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, so the time complexity is of order $O(n^2)$.

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: Total:
Time Complexity: $O(n^2)$
Space Complexity: $O(n)$

3.5 Max Subarray Sum

Problem Statement: Given an integer array, return the largest value that a subarray[9] of the array sums to.

Input: An integer array *nums*.

Output: An integer *max_sum*.

Example: For:
$nums = [-2, 1, -3, 4, -1, 2, 1, -5, 4]$
$max\_sum = 6$

Explanation: $[4, -1, 2, 1]$ is the subarray of *nums* that has the largest sum, 6.

3.5.1 Brute Force Approach to Max Subarray Sum

The naiive way to solve this problem is to generate every possible subarray of *nums* starting with the subarray containing only *nums*[0], and ending with the entirety of *nums*. Then sum each subarray and get the maximum of these sums. Instead of generating every subarray, we can instead iterate over each subarray in place by iterating over *nums* with a *start* pointer marking the start of the subarray, and iterate over the rest of the array with an *end* pointer marking the end of the subarray. This reduces the space complexity from O(n) to O(1) (see section 3.5.2). We can improve this by keeping track of a *current_sum* and *max_sum* and updating them dynamically as we execute the *end* loop rather than summing each subarray after it is generated, reducing the time complexity from $O(n^3)$ to $O(n^2)$ (see section 3.5.2).

A sample python implementation is shown in Figure 3.11.

[9]A subarray is a contiguous subsequence. This means all elements of the subsequence are strictly consecutive in the original sequence.

```python
def max_subarray_sum_bf(nums):
    if not nums:
        return 0

    n = len(nums)
    max_sum = float('-inf')

    for start in range(n):
        current_sum = 0
        for end in range(start, n):
            current_sum += nums[end]
            max_sum = max(max_sum, current_sum)

    return max_sum
```

Figure 3.11: Max Subarray Sum Brute Force Python Implementation.

3.5.2 Complexity Analysis of the Brute Force Approach to Max Subarray Sum

Let n be the length of nums.

Time Complexity: Due to the use of a double nested for loop, generating all subarrays of *nums* has a time complexity of $O(n^2)$. All other operations such as adding to the *current_sum*, resetting the *current_sum* and getting the max of *current_sum* and *max_sum* are constant time. If we were to sum each subarray individually rather than keeping a *current_sum*, which would be an $O(n)$ operation, the total complexity would be increased to $O(n^3)$.

Space Complexity: All other variables stored such as *max_sum* and *current_sum* are constant space. The number of variables stored does not increase as n increases, hence the space complexity is $O(1)$. If instead of iterating over each subarray in place, we stored the current subarray separately, the space complexity would be increased to $O(n)$.

Overall: Total:
Time Complexity: $O(n^2)$
Space Complexity: $O(1)$

3.5.3 Kadanes Algorithm for Max Subarray Sum

There is a way to find the max subarray sum in a single iteration of *nums*. This is done using the famous Kadane's algorithm. Kadane's algorithm is shown in Algorithm 3:

Algorithm 3: Kadane's Algorithm

```
Input: An integer array $nums$
Output: An integer $max\_sum$, the max subarray sum of $nums$
$max\_sum = 0$;
$current\_sum = 0$;
foreach $i \in nums$ do
    if $current\_sum < 0$ then
        $current\_sum := 0$;
    $current\_sum += i$;
    if $current\_sum > max\_sum$ then
        $max\_sum := current\_sum$;
return $max\_sum$
```

Explanation of Kadanes Algorithm

In Kadane's algorithm, we iterate over *nums*, and at each step increment *current_sum* by the value of the current element. If *current_sum* becomes negative, *current_sum* is set to the value of current element. The *max_sum* is updated to *current_sum* whenever *current_sum* becomes greater than *max_sum*. If the array consists entirely of negative numbers, the algorithm will return 0 for the maximum subarray sum. If you want to modify the algorithm such that empty subarrays are not allowed, you can initialize *max_sum* and *current_sum* to the first element of the array instead of 0. This way, the algorithm will return the largest single negative element if the array consists entirely of negative numbers. Kadane's algorithm is considered a dynamic programming algorithm even though it does not use a table, because it satisfies the necessary criteria:

Optimal Substructure: The problem of finding the maximum subarray can be divided into smaller subproblems, where the solution to the problem at each index depends on the solution to the problem at the previous index.

Overlapping Subproblems: The subproblems in Kadane's algorithm overlap, as the solution to the problem at each index relies on the solution to the problem at the previous index.

A sample python implementation is shown in Figure 3.12.

```python
    def max_subarray_sum_kadanes(nums):
        if not nums:
            return 0

        max_sum = current_sum = nums[0]

        for num in nums[1:]:
            if current_sum < 0:
                current_sum = 0
            current_sum += num
            max_sum = max(max_sum, current_sum)

        return max_sum
```

Figure 3.12: Kadane's Algorithm Python Implementation

3.5.4 Complexity Analysis of Kadane's Algorithm

Let n be the length of nums.

Time Complexity: This algorithm consists of a single iteration of nums, which is $O(n)$. All other operations such as updating the $current\_sum$, max_sum are constant time.

Space Complexity: All variables stored such as max_sum and $current\_sum$ are constant space, and we do not store more information as n grows. Hence the space complexity is $O(1)$.

Overall: Total:
Time Complexity: $O(n)$
$SpaceComplexity$:O(1)

3.6 Longest Alternating Subsequence

3.7 Binomial Coefficients

3.8 Longest Common Subsequence

3.9 Longest Palindromic Subsequence

3.10 Longest Contiguous Palindromic Substring

3.11 The Needleman-Wunsch Algorithm

3.12 The Smith-Waterman Algorithm