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published or unpublished, electronically or in print.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

Research on Dynamic Programming

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.

```
1    def fib_bf(n):
2        if n <=2: return 1
3        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.

```
1  def fib_memo(n, memo={}):
2      if n <= 2:
3          return 1
4      if n in memo:
5          return memo[n]
6      memo[n] = fib_memo(n-1, memo) + fib_memo(n-2, memo)
7      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 ← 0;
total ← 0;
while total < a do
    if |D| = 0 then
        return -1;
    if total + D[−1] > a then
        D.pop();
    else
        total ← total + D[−1];
        r ← 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.

```
1    def coin_change_bf(D, a):
2        def dfs(a):
3            if a == 0:
4                return 0
5            if a<0:
6                return float('inf')
7            return min([1+dfs(a-c) for c in D])
8        minimum = dfs(a)
9        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.

```
1     def coin_change_memo(D, a):
2     memo = {}
3     def dfs(a):
4         if a == 0:
5             return 0
6         if a < 0:
7             return float('inf')
8         if a in memo:
9             return memo[a]
10
11        memo[a] = min([1+dfs(a-c) for c in D])
12        return memo[a]
13
14    res = dfs(a)
15    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.

```
1    def coin_change_dp(D,a):
2        dp=[float('inf')] * (a + 1)
3        dp[0] = 0
4
5        for i in range(1, a+1):
6            for c in D:
7                t = i - c
8                if t >= 0:
9                    dp[i] = min(dp[i], 1+dp[t])
10
11       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 ≤ |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.

```
1   def lis_memo(nums):
2       if not nums:
3           return 0
4
5       memo = {}
6
7       def dfs(prev_index, current_index):
8           if current_index == len(nums):
9               return 0
10
11          if (prev_index, current_index) in memo:
12              return memo[(prev_index, current_index)]
13
14          exclude_current = dfs(prev_index, current_index + 1)
15
16          include_current = 0
17          if prev_index < 0 or nums[current_index] >
                nums[prev_index]:
18              include_current = 1 + dfs(current_index, current_index
                    + 1)
19
20          memo[(prev_index, current_index)] = max(include_current,
                exclude_current)
21
22          return memo[(prev_index, current_index)]
23
24      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.

```
1    def lis_dp(nums):
2        dp = [1] * len(nums)
3
4        for i in range(len(nums)-1,-1,-1):
5            for j in range(i+1,len(nums)):
6                if nums[i] < nums[j]:
7                    dp[i] = max(dp[i], 1+dp[j])
8
9        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 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 further 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.

```
1   def max_subarray_sum_kadanes(nums):
2       if not nums:
3           return 0
4
5       max_sum = current_sum = nums[0]
6
7       for num in nums[1:]:
8           if current_sum < 0:
9               current_sum = 0
10          current_sum += num
11          max_sum = max(max_sum, current_sum)
12
13      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

Problem Statement: Given an array, find the length of the longest alternating subsequence in the array[10]. A sequence $x1, x2, x3, x4...xn$ is alternating if its elements satisfy one of the following:

$x1 > x2 < x3 > x4 < ...$

$x1 < x2 > x3 < x4 > ...$

Input: An integer array $nums$.

Output: An integer max_length.

Example: For:
$nums = [1, 17, 5, 10, 13, 15, 10, 5, 16, 8]$
$max_length = 7$

Explanation: The longest alternating subsequence in $nums$ is $[1, 17, 5, 15, 5, 16, 8]$ which has length 7.

3.6.1 Brute Force Approach to Longest Alternating Subsequence

The naiive way to approach this problem is to generate every possible subsequence of nums. For each subsequence, we can check if it is alternating iteratively, and if it is, calculate its length. We keep track of the maximum length of all of the alternating subsequences.

An implementation of the brute force algorithm is given in Figure 3.13.

[10]Subsequences do not have to be continuous.

```
1     def is_alternating(sequence):
2         if len(sequence) < 3:
3             return True
4
5         for i in range(1, len(sequence) - 1):
6             if not ((sequence[i - 1] > sequence[i] < sequence[i + 1])
                  or
7                       (sequence[i - 1] < sequence[i] > sequence[i + 1])):
8                  return False
9         return True
10
11    def las_bf(nums):
12        if not nums:
13            return 0
14
15        n = len(nums)
16        max_length = 1
17
18        for i in range(1 << n):
19            subsequence = [nums[j] for j in range(n) if (i & (1 << j))
                  > 0]
20            if is_alternating(subsequence):
21                max_length = max(max_length, len(subsequence))
22
23        return max_length
```

Figure 3.13: Longest Alternating Subsequence Brute Force Python Implementation

In line 18 in Figure 3.13, The expression $1 << n$ represents a bitwise left shift operation. In Python, $<<$ is the left shift operator, and it shifts the binary representation of the number to the left by n positions. In the context of generating all possible subsequences, $1 << n$ is used to create a bitmask with the rightmost n bits set to 1. Each bit in the bitmask corresponds to whether the corresponding element in the array is included or excluded in the current subsequence. The line "for i in range($1 << n$)" generates all possible subsequences by iterating through all bitmasks from 0 to $2^n - 1$.

3.6.2 Complexity Analysis of the Brute Force Approach to Longest Alternating Subsequence

Let n be the length of nums.

Time Complexity: The complexity of generating every possible subsequence is $O(2^n)$. Checking if a sequence is alternating is $O(|sequence|)$, and getting the length of a subsequence, as well as comparing it to the max_length are $O(1)$. This is overall of order $O(2^n)$.

Space Complexity: The space complexity is $O(n)$, as we must store a single subarray at a time, which in the worst case is of length n.

Overall: Total:

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

Notice that for an aribitrary longest alternating subsequence of length k ending at index i, if there exists exactly one number which can extend the sequence, the length of the total longest alternating 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.

As using memoization on subsequence problems is not space efficient, we can use tabulation and only store the necessary information rather than an intermediate result for every subsequence. The following is known as the Auxiliary Arrays solution.

3.6.3 Auxiliary Arrays Solution for Longest Alternating Subsequence

Two auxiliary arrays *inc* and *dec*, of length $|nums|$ are initialized. $inc[i]$ contains the length of the longest alternating subsequence of $nums[0:i]$, where the last element of the subsequence is greater than the previous element. $dec[i]$ contains the length of the longest alternating subarray of $nums[0:i]$, where the last element of the subsequence is less than the previous element.

The algorithm iterates through *nums*, considering each element $nums[i]$ and updating the inc and dec arrays based on the following conditions:

1. If $nums[i]$ is greater than $nums[j]$ for some previous index j, it means the sequence can be extended in an increasing manner. In this case, $inc[i]$ is updated to be the maximum of its current value and the length of the longest decreasing subsequence ending at index $j+1$, found in $dec[j+1]$.

2. If $nums[i]$ is smaller than $nums[j]$ for some previous index j, it means the sequence can be extended in a decreasing manner. In this case, $dec[i]$ is updated to be the maximum of its current value and the length of the longest increasing subsequence ending at index $j+1$, found in $inc[j+1]$.

3. The length of the longest alternating subsequence is the maximum value in the *inc* and *dec* arrays.

A sample python implementation is shown in Figure 3.14.

```
1   def las_auxiliary(nums):
2       n = len(nums)
3
4       inc = [1] * n
5       dec = [1] * n
6
7       for i in range(1, n):
8           for j in range(i):
9               if nums[i] > nums[j]:
10                  inc[i] = max(inc[i], dec[j] + 1)
11              elif nums[i] < nums[j]:
12                  dec[i] = max(dec[i], inc[j] + 1)
13
14      return max(max(inc), max(dec))
```

Figure 3.14: Longest Alternating Subsequence Auxiliary Arrays Python Implementation

3.6.4 Complexity Analysis of the Auxiliary Arrays Approach to Longest Alternating Subsequence

Let n be the length of nums.

Time Complexity: We use a double nested for loop to iterate over *nums*. This gives a complexity of $O(n^2)$. All other operations are either constant time lookups, updates or $max(a, b)$, all of which are $O(1)$.

Space Complexity: The space complexity is $O(2n)$, which is of order $O(n)$. This is because we must store the *inc* array and the *dec* array, each of which have the same length as *nums*. All other variables are stored with constant space complexity.

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

3.6.5 Optimized Soluiton to Longest Alternating Subsequence

Observe that at each step, the max of *inc* and *dec* can be found at the current index of the arrays, so we do not need to store the entire *inc* and *dec* arrays. Instead, we can use an *inc* and *dec* variable which will hold the value stored at $inc[i]$ and $dec[i]$ respectively. This is because a maximum alternating subsqeuence of $nums[0 : a]$ will never be of lower length than a maximum alternating subsequence of $nums[0 : b]$ if $a > b$.

We can do a single iteration over nums where:

1. *inc* should be set to $dec + 1$, if and only if the last element in the alternating sequence was less than its previous element.
2. *dec* should be set to $inc + 1$, if and only if the last element in the alternating sequence was greater than its previous element.

A sample python implementation is shown in Figure 3.15.

```
1   def las_optimized(nums):
2       n = len(nums)
3
4       inc = 1
5       dec = 1
6
7       for i in range(1, n):
8           if nums[i] > nums[i - 1]:
9               inc = dec + 1
10          elif nums[i] < nums[i - 1]:
11              dec = inc + 1
12
13      result = max(inc, dec)
14      return result
```

Figure 3.15: Longest Alternating Subsequence Optimzied Python Implementation

3.6.6 Complexity Analysis of the Optimized Approach to Longest Alternating Subsequence

Let n be the length of nums.

Time Complexity: We use a single for loop to iterate over *nums*. This gives a complexity of $O(n)$. All other operations are either constant time lookups, updates or $max(a, b)$, all of which are $O(1)$.

Space Complexity: The space complexity is $O(1)$ as we only need to store a single value for *inc* and *dec*.

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

3.7 Binomial Coefficients

Problem Statement: Given as input positive integers n and k where $n \geq k$, calculate how many different ways you can select k unique examples from a set of size n. In other words, calculate $C(n, k)$ where:

$$C(n, k) = n!/k!((n - k)!)$$

Input: Two positive integers $C(n, k)$

Output: A single positive integer $C(n, k)$

Example: For:
$n = 4$
$k = 2$
$C(n, k) = 6$

Explanation: There are 6 unique ways you can choose 2 different examples from a set of size 4.

We know that by definition:

$$C(n, k) = C(n - 1, k - 1) + C(n - 1, k)$$

$$C(n, 0) = 1$$

$$C(n, n) = 1$$

We can use this as a recursive case and base cases in a brute force solution.

3.7.1 Brute Force Approach to Binomial Coefficients

We can use recursion to arrive at our answer, using the definition as our recursive case and our base cases.

A sample python implementation is shown in Figure 3.16.

```python
def C_bf(n,k):
    if k == 0 or k == n: return 1
    return C_bf(n-1,k-1) + C_bf(n-1,k)
```

Figure 3.16: Binomial Coefficients Brute Force Python Implementation

3.7.2 Complexity Analysis of the Brute Force Approach to Binomial Coefficients

Time Complexity: For our analysis, in the worst case k is equal to $n/2$. This means that at each recursion, we create two subproblems. We do this n times, giving a total time complexity of $O(2^n)$.

Space Complexity: The space complexity is determined by the depth of recursion, which is $O(n)$.

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

3.7.3 Memoization Approach to Binomial Coefficients

If we look at the trace of calculating $C(4,2)$ in our recursive algorithm, we can see that it is guaranteed to be the sum of $C(3,1)$ and $C(3,2)$, as by definition $C(n,k) = C(n-1,k-1) + C(n-1,k)$. This shows the problem has the optimal substructure property. To calculate $C(3,1)$ we must know $C(2,1)$. This is also true when calculating $C(3,2)$. This means there is repeated calculations of $C(2,1)$, showing the overlapping subproblems property. In larger problems, the number of repeated calculations is vast. We can use memoization to store each calculation we do in a table called *memo* to avoid repeating subproblems. At each step, before continuing with recursion, we check *memo* to see if the calculation has been done already, and if so, we take the value from the *memo* table instead of making a recursive call.

A sample python implementation is shown in Figure 3.17.

```python
def C_memo(n,k,memo={}):
    if k == 0 or k == n: return 1

    if (n,k) in memo:
        return memo[(n,k)]

    result = C_memo(n-1,k-1,memo) + C_memo(n-1,k,memo)
    memo[(n,k)] = result
    return result
```

Figure 3.17: Binomial Coefficeints Memoization Python Implementation

3.7.4 Complexity Analysis of the Memoization Approach to Binomial Coefficients

Time Complexity: The memoization table ensures that each unique subproblem is solved only once. In the case of "n choose k," there are $O(n * k)$ unique subproblems because the parameters n and

k can take values from 0 to n. The rest of the table lookups are expected constant time as we are using a dictionary for the *memo* table.

Space Complexity: We must store the *memo* table, which is a dictionary of size $O(n * k)$ as n and k can take values from 0 to n.

Overall: Total:

Time Complexity: $O(n * k)$

Space Complexity: $O(n * k)$

3.7.5 Tabulation Approach to Binomial Coefficients

Instead of recursively creating the table by searching the entire tree of possible n and k values, we can use tabulation to fill in a table from which we can extract our answer. Consider a table *dp* with dimensions $(n + 1)$ x $(k + 1)$, where *dp*[*i*][*j*] in the table contains the result of $C(i, j)$ We could build this table up starting from our base cases:

$$C(n, 0) = 1$$

$$C(n, n) = 1$$

, until we have a solution to $C(n, k)$.

So for the calculation of $C(4, 2)$ for example, we would initialize the following table:

	0	**1**	**2**
0	1	0	0
1	1	1	0
2	1		1
3	1		
4	1		

Now, we can use $C(n, k) = C(n - 1, k - 1) + C(n - 1, k)$ to fill the table.

For example, to get *dp*[2][1] ($C(2, 1)$), we take *dp*[1][0] + *dp*[1][1] = 2.

Do this for the entire table as follows:

	0	**1**	**2**
0	1	0	0
1	1	1	0
2	1	2	1
3	1	3	3
4	1	4	6

Until we arrive at *dp*[*n*][*k*] which will give the answer of $C(n, k)$.

A sample python implementation is shown in Figure 3.18.

```
    def C_dp(n,k):
        dp = [[0] * (k+1) for _ in range(n+1)]

        #Fill in the base cases
        for i in range(n+1):
            dp[i][0] = 1
            dp[i][min(i,k)]=1

        #Fill in the rest of the table using the definition of C(n,k)
        for i in range(1,n+1):
            for j in range(1,min(i,k)+1):
                dp[i][j] = dp[i-1][j-1] + dp[i-1][j]

        return dp[n][k]
```

Figure 3.18: Binomial Coefficients Tabulation Python Implementation

We do not fill the table where $k > n$. This because it is impossible to choose k unique examples from a set of size n. This is done by only iterating to $min(i, k)$ in the inner loop.

3.7.6 Complexity Analysis of the Tabulation Approach to Binomial Coefficients

Time Complexity: We have to fill in a table of size $O(n * k)$ with values, and each value is obtained through a constant time lookup and addition.

Space Complexity: We must store the *dp* table, which is a 2D array of size $O(n * k)$ as n and k can take values from 0 to n.

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

3.7.7 Optimzied Tabulation Approach to Binomial Coefficeints

Notice that we can calculate the values in any row r using the values in row $r-1$. Hence, we do not need to store the entire *dp* table in memory, only two rows at a time. The current row r, and the previous row $r-1$. This reduces space complexity from $O(n*k)$ to $O(k)$. Notice also that $C(n,k) = C(n,n-k)$. We can therefore set $k = n-k$ if $k > n-k$ and $n-k$ is positive before we start our algorithm, and our result will be the same. This reduces the time complexity from $O(n*k)$ to $O(n*min(k,n-k))$.

A sample python implementation is shown in Figure 3.19.

```
    def C_optimized(n,k):
        if k > n-k and n-k >= 0:
            k = n-k
        oldRow = [0] * (k+1)
        oldRow[0] = 1

        for i in range(1,n+1):
            newRow = [0] * (k+1)
            newRow[0] = 1
            for j in range(1,min(i,k)+1):
                newRow[j] = oldRow[j-1] + oldRow[j]

            oldRow = newRow

        return newRow[k]
```

Figure 3.19: Binomial Coefficients Optimized Python Implementation

3.8 Longest Common Subsequence

Problem Statement: Given two strings, return the length of their longest common subsequence[11].

Input: Two strings $text1$ and $text2$.

Output: An integer lcs, the longest common subsequence of $text1$ and $text2$.

Example: For:
$text1 =$ "$abcde$"
$text2 =$ "ace"
$lcs = 3$

Explanation: The longest common subsequence of "$abcde$" and "ace" is "ace", which is of length 3.

3.8.1 Longest Common Subsequence Brute Force

The naiive way to find the longest common subsequence of two strings is to generate all possible subsequences of both strings. We initialize a lcs variable to store the length of the longest common subsequence. We then iterate over each list of subsequences and report any common subsequences we find, updating lcs accordingly. Finally we return lcs.

3.8.2 Complexity Analysis of the Brute Force Approach to Longest Common Subsequence

For the calculation of worst case time and space complexity, we assume that both strings are equal in length, and that length is denoted by n.

Time Complexity: The time complexity of generating every subsequence of a string of length n is $O(2^n)$, and doing this for both strings is $O(2 * (2^n))$ which is $O(2^n + 1)$, as for each character we

[11] Subsequences do not have to be contiguous.

decide if we include it in or exclude it from the substring. The time complexity of iterating over two lists of substrings both of size $O(2^n)$ and looking for matches is $O((2^n)^2)$ which is equal to $O(2^{2n})$. This is still of order $O(2^n)$.

Space Complexity: The space complexity of storing every subsequence of a string of length n is $O(2^n)$. As we must do this for two strings, the total space complexity required is $O(2 * 2^n)$ which is of order $O(2^n)$.

Overall: Total:

Time Complexity: $O(2^n)$

Space Complexity: $O(2^n)$

Notice that:

OBSERVATION 1: If we have a common character at the current position, such as position 1 in:

$$text1 = abcde, text2 = ace$$

The solution will be $1+lcs(bcde, ce)$[12]. This shows the problem has the optimal substructure property.

OBSERVATION 2: If there is no common character at the current position such as in:

$$text1 = bcde, text2 = ce$$

The solution will be $max(lcs(cde, ce), lcs(bcde, e))$[13]. As we have the chance to recurse on the same two substrings multiple times, the problem has the overlapping subproblems property.

3.8.3 Tabulation Approach to Longest Common Subsequence

We can use these two cases to solve this problem by tabulation. We create a table called dp with $i + 1$ rows and $j + 1$ columns, where i and j are the lengths of $text1$ and $text2$ respectively. Each row and column of dp represents a character in $text1$ and $text2$.

	a	**c**	**e**
a			
b			
c			
d			
e			

In this table, for example, $dp[0][0]$ will represent $lcs(abcde, ace) \rightarrow$ the solution. $dp[3][1]$ will represent $lcs(de, ce)$ ect.. i.e. $dp[i][j]$ represents the longest common subseqence of $text1[i :]$ and $text2[j :]$.

We fill the table starting from the bottom right as follows:

1. If the row and column have the same label, we have found a common character.

   As per OBSERVATION 1, we fill the space with $1 + dp[i + 1][j + 1]$.

2. If the row and col do not have the same label, we must use OBSERVATION 2:

   Fill the space with $max(dp[i][j + 1], dp[i + 1][j])$.

3. If we go out of bounds, we simply take zero[14].

[12] Remove the common character from both texts, add 1 and recurse. The reason we add 1 is because each common character contributes 1 length to our final output.

[13] Remove the first character from either the first or second text, recurse on both cases.

[14] For ease of code, the grid is $(i + 1)$ x $(j + 1)$ and initialized to all zeros, so we don't have to check for going out of bounds.

Our solution will be located at $dp[0][0]$, which represents $lcs(text1, text2)$.

	a	**c**	**e**
a	3	2	1
b	2	2	1
c	2	2	1
d	1	1	1
e	1	1	1

A sample python implementation is shown in Figure 3.20.

```
1   def lcs(text1, text2):
2       dp=[[0 for j in range(len(text2)+1)] for i in
            range(len(text1)+1)]
3
4       for i in range(len(text1)-1,-1,-1):
5           for j in range(len(text2)-1,-1,-1):
6               if text1[i] == text2[j]:
7                   dp[i][j] = 1 + dp[i+1][j+1]
8               else:
9                   dp[i][j] = max(dp[i][j+1], dp[i+1][j])
10
11      return dp[0][0]
```

Figure 3.20: Longest Common Subsequence Tabulation Python Implementation

3.8.4 Complexity Analysis of the Tabulation Approach to Longest Common Subsequence

Let i be the length of $text1$, and j be the length of $text2$.

Time Complexity: The time complexity of filling an $i*j$ table with values, where each value is derived from a constant time lookup and a constant time addition or max function is $O(i*j)$.

Space Complexity: The space complexity of storing an $i*j$ table is $O(i*j)$.

Overall: Total:
Time Complexity: $O(i*j)$
Space Complexity: $O(i*j)$

NOTE THAT SINCE EACH ROW OF THIS TABLE CAN BE COMPUTED USING THE PREVIOUS ROW ONLY, THE SAME TWO-ROW METHOD CAN BE USED AS IN BINOMIAL COEFFICIENTS

3.9 Longest Palindromic Subsequence

Problem Statement: Given a string, return the length of the longest subsequence[15] of the string which is a palindrome[16].

[15] Subsequences do not have to be continuous.
[16] A palindrome is a string which is identical to itself when reversed (Example: "racecar").

Input: A string s.

Output: An integer lps which represents the longest palondromic subsequence of s.

Example: For:
$s =$ "$babbb$"
$lps = 4$

Explanation: "$bbbb$" is the longest palindromic subsequence of s. It has length 4.

3.9.1 Tabulation Approach to Longest Palindromic Subsequence

This problem is simply a special case of Longest Common Subsequence. The longest palindromic subsequence of s is simply the longest common subsequence of s and $reverse(s)$. Therefore the same logic applies. See Section 3.8.

A sample python implementation is shown in Figure 3.21. For the implementation of the lcs subroutine, see Figure 3.20.

```
1    def lps(s):
2        return lcs(s,s[::-1])
```

Figure 3.21: Longest Palindromic Subsequence Python Implementation

3.10 Longest Contiguous Palindromic Substring

Problem Statement: Given a string, return the longest palindromic substring[17] of the string.

Input: A string s.

Output: A string $lpcs(s)$, which is the longest palindromic substring of s.

Example: For:
$s =$ "$aaaabbaa$"
$lpcs(s) =$ "$aabbaa$"

Explanation: The longest palindromic substring of "$aaaabbaa$" is "$aabbaa$". Note that here we are asked for the substring itself, rather than the length of the substring.

3.10.1 Brute Force Approach to Longest Contiguous Palindromic Substring

If we were to tackle this problem the brute force way, we would have to iterate over all substrings of s, filtering out non palindromes along the way, and keeping track of the longest palindromic substring found so far. We would then return the longest palindromic substring of s.

3.10.2 Complexity Analysis of the Brute Force Approach to Longest Contiguous Palindromic Substring

Let n be the length of s.

[17] Substrings must be contiguous.

Time Complexity: Iterating over all substrings of a string of length n has a time complexity of $O(n^2)$. Checking if a substring is a palindrome has a time complexity of $O(n)$. This gives the brute force approach an overall time complexity of $O(n^3)$.

Space Complexity: The space complexity of generating all substrings and storing them in a list is also $O(n^3)$ because we have $O(n^2)$ substrings, each of which has a maximum length of $O(n)$. However, in our algorithm it is possible to iterate over the subsrtings of s in place using a double for loop [18], we can store one substring at a time, which has a maximum length of n. This brings the space complexity down to $O(n)$.

Overall: Total:
Time Complexity: $O(n^3)$
Space Complexity: $O(n)$

Notice that if a string p is a palindrome, and x is a character, the string xpx is guaranteed to be a palindrome. This is proof of optimal substructure. The problem of determining whether a substring is a palindrome or not can be reduced to smaller subproblems. For example, when checking if $s[i:j]$ is a palindrome, we often need to check if $s[i+1:j-1]$ is a palindrome, which overlaps with other similar subproblems. Therefore, we can use dynamic programming principles to optimize the solution to this problem.

3.10.3 Tabulation Approach to Longest Contiguous Palindromic Substring.

We can create a table dp where $dp[i][j]$ is 1 if $s[i:j+1]$ is a palindrome, else 0. (eg: if $dp[1][3] = 1$, $s[1:4]$("aaa" in our example string) is a palindrome.) We start with a table dp which is a 2D matrix of size $|s| * |s|$. We can initialize all fields where $i == j$ to 1, as single letters are palindromes.

	a	**a**	**a**	**a**	**b**	**b**	**a**	**a**
a	1							
a		1						
a			1					
a				1				
b					1			
b						1		
a							1	
a								1

We can initialize all fields where $j = i + 1$ and $s[i] = s[j]$ to 1 as all pairs of the same letter are palindromes. This handles the case where the palindrome has an even amount of characters (meaning the center of the palindrome is a character pair).

	a	**a**	**a**	**a**	**b**	**b**	**a**	**a**
a	1	1						
a		1	1					
a			1	1				
a				1	0			
b					1	1		
b						1	0	
a							1	1
a								1

[18] See Figure 3.11 for an example of how to do this.

Now, notice that if a string p is a palindrome, and x is a character, the string xpx is guaranteed to be a palindrome. Using this, for all substrings s' of length 3, we check if the start and end are the same letter, and the middle is a palindrome (we know from the existing entries in the table). If so, we know that s' is a palindrome, so we can set $dp[i][j]$ to 1. We can put a 1 in $dp[i][j]$ the table if and only if:

1. If $s[i] == s[j]$ (the starting character is equal to the ending character).

   AND

2. If $dp[i+1][j-1] == 1$ (the string p in between i and j is already known to be a palindrome).

We will end up with a table as follows:

	a	**a**	**a**	**a**	**b**	**b**	**a**	**a**
a	1	1	1					
a		1	1	1				
a			1	1	0			
a				1	0	0		
b					1	1	0	
b						1	0	0
a							1	1
a								1

Do this for all lengths from $3 \rightarrow len(s) - 1$. We will end up with the following table.

	a	**a**	**a**	**a**	**b**	**b**	**a**	**a**
a	1	1	1	1	0	0	0	0
a		1	1	1	0	0	0	0
a			1	1	0	0	0	1
a				1	0	0	1	0
b					1	1	0	0
b						1	0	0
a							1	1
a								1

From the table we can deduce all palindromic substrings, including the longest palindromic substring. To get the longest palindromic substring from the table, we track *start* and *max_length*, which get updated every time a new palindrome is found. This works because palindromes are found from shortest to longest, so every new palindrome found is the maximum length palindrome. We can then simply return a slice of the input string as follows: $s[start, start + max_length]$. This will yield the maximum length palindromic substring.

A sample python implementation is shown in Figure 3.22.

```python
def lpcs(s):
    n = len(s)
    # Single character strings are palindromes, so we can return s
    if n <=1: return s

    # Create a n*n table of zeros
    dp = [[0] * n for _ in range(n)]

    # Initialize all single character substrings to 1

    # Single character substrings start and end at the same index,
       so we locate them with dp[i][i]
    for i in range(n):
        dp[i][i] = 1

    # Since we are looking for the longest palindromic substring
       itself and not the length,
    # we can track the starting position and max_length for
       convenience
    start = 0
    max_length = 1

    # Initialize all pairs of identical characters to 1
    for i in range(n-1):
        if s[i] == s[i+1]:
            dp[i][i+1] = 1
            start, max_length = i,2

    # For all substrings with length >=3, starting at length 3,
    # set dp[i][j] to 1 if start and end are the same letter
    # and the middle is a palindrome

    for length in range(3, n+1):
        for i in range(n - length + 1):
            j = i + length - 1
            if dp[i+1][j-1] and s[i] == s[j]:
                dp[i][j] = 1
                start, max_length = i, length

    return s[start:start+max_length]
```

Figure 3.22: Longest Contiguous Palindromic Substring Tabulation Python Implementation

3.10.4 Complexity Analysis of the Tabulation Approach to Longest Contiguous Palindromic Subsequence

Let n be the length of s.

Time Complexity: The time complexity to build an $n * n$ table, where the values are deduced from a constant time check and a constant time lookup is $O(n^2)$.

Space Complexity: The space complexity to store an $n * n$ table is $O(n^2)$.

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

3.11 The Needleman-Wunsch Algorithm

This famous dynamic programming algorithm is used for global alignment of DNA and protein sequences.

Problem Statement: Given two character sequences $seq1$ and $seq2$, place zero or more 'gap's in $seq1$ and/or $seq2$ such that the maximum number of characters in $seq1$ and $seq2$ are 'aligned'. Characters X and Y are 'aligned' when $X == Y$ and the index of X in $seq1$ is exactly equal to the index of Y in $seq2$.

Input: Two strings $seq1$ and $seq2$.

Output: Two strings $align1$ and $align2$.

Example: For:
$seq1 =$ "$ATGCT$"
$seq2 =$ "$AGCT$"
$align1 =$ "$ATGCT$"
$align2 =$ "$A - GCT$"

Explanation: Globally Aligned sequences:
A T G C T
A - G C T

The trick is to find an efficient way to place gaps in the sequences to maximise the amount of globally aligned letters. We can do this with tabulation:
Create a matrix of size: $(len(seq1) + 1)$ x $(len(seq2) + 1)$ as follows:

		A	**T**	**G**	**C**	**T**
A						
G						
C						
T						

Now use the following scheme to fill in the table:
$Match : 1$
$Mismatch : -1$
$GAP : -2$

3.11.1 Initialization of the Needleman-Wunsch Table

Starting with 0 at $(0, 0)$, fill the extra row and column with progressive GAP penalties as follows:

		A	**T**	**G**	**C**	**T**
	0	-2	-4	-6	-8	-10
A	-2					
G	-4					
C	-6					
T	-8					

3.11.2 Filling the Needleman-Wunsch Table

Now starting at $(1, 1)$, and going row-wise, fill each cell with the max of the following:

1. Value from left + GAP
2. Value from above + GAP
3. Value from diagonal + ($Match$ or $Mismatch$) [depending on wether the row and column label are the same letter]

So, for the $(1, 1)$ cell, we fill it with $max(-4, -4, 1) = 1$.
And for the $(1, 2)$ cell, we fill it with $max(-1, -6, -3) = -1$.
And so on...
Until we get:

		A	**T**	**G**	**C**	**T**
	0	-2	-4	-6	-8	-10
A	-2	1	-1	-3	-5	-7
G	-4	-1	0	0	-2	-4
C	-6	-3	-2	-1	1	-1
T	-8	-5	-2	-3	-1	2

3.11.3 Traceback in the Needleman-Wunsch Table

Starting from the bottom-right (which will always be the highest value in the matrix), continue the following until $(0, 0)$ is reached:

- If *row* and *col* labels match, go diagonally top-left.
- Else, go to $max(left, above, diagonal)$.
- Each time we go diagonally, we can align the *row* and *col* labels.
- Each time we go left, we put a gap in $seq2$ at that index.
- Each time we go up, we put a gap in $seq1$ at that index.

Note that the aligned sequences will be written from right to left.
A sample python implementation is given in Figure 3.23

```python
def needleman_wunsch(seq1, seq2, match=1, mismatch=-1, gap=-2):
    # Initialize the scoring matrix
    m, n = len(seq2), len(seq1)
    score = [[0] * (n + 1) for _ in range(m + 1)]

    # Initialize the first row and column
    for i in range(m + 1):
        score[i][0] = i * gap
    for j in range(n + 1):
        score[0][j] = j * gap

    # Fill in the scoring matrix
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            match_mismatch = match if seq2[i - 1] == seq1[j - 1]
                else mismatch
            diagonal = score[i - 1][j - 1] + match_mismatch
            horizontal = score[i][j - 1] + gap
            vertical = score[i - 1][j] + gap
            score[i][j] = max(diagonal, horizontal, vertical)

    # Traceback to find the alignment
    align2, align1 = "", ""
    i, j = m, n
    while i > 0 or j > 0:
        if i > 0 and j > 0 and score[i][j] == score[i - 1][j - 1]
            + (match if seq2[i - 1] == seq1[j - 1] else mismatch):
            align2 = seq2[i - 1] + align2
            align1 = seq1[j - 1] + align1
            i -= 1
            j -= 1
        elif i > 0 and score[i][j] == score[i - 1][j] + gap:
            align2 = seq2[i - 1] + align2
            align1 = "-" + align1
            i -= 1
        else:
            align2 = "-" + align2
            align1 = seq1[j - 1] + align1
            j -= 1

    return align1, align2

sequence1 = "ATGCT"
sequence2 = "AGCT"
align1, align2 = needleman_wunsch(sequence1, sequence2)
```

Figure 3.23: The Needleman Wunsch Algorithm Python Implementation

3.12 The Smith-Waterman Algorithm

This famous dynamic programming algorithm is used for local alignment of DNA and protein sequences.

Problem Statement: Given two character sequences $seq1$ and $seq2$, remove characters from the beginning or end of $seq1$ and/or $seq2$ such that the maximum number of characters in $seq1$ and $seq2$ are 'aligned'. Characters X and Y are 'aligned' when $X == Y$ and the index of X in $seq1$ is exactly equal to the index of Y in $seq2$.

Input: Two strings $seq1$ and $seq2$.

Output: Two strings $align1$ and $align2$.

Example: For:

$seq1 =$ "$ATGCT$"
$seq2 =$ "$AGCT$"
$align1 =$ "GCT"
$align2 =$ "GCT"

Explanation: Locally Aligned sequences:

G C T
G C T

The trick is to find an efficient way to trim the sequences, such that the number of aligned letters is maximised. In this case it is 3.

This is very similar to Needleman-Wunsch. The algorithm is the same as above, however all negative values become zero as follows:

Create a matrix of size: $(len(seq1) + 1)$ x $(len(seq2) + 1)$ as follows:

		A	**T**	**G**	**C**	**T**
A						
G						
C						
T						

Now use the following scheme to fill in the table:

$Match : 1$
$Mismatch : -1$
$GAP : -2$

3.12.1 Initialization of the Smith-Waterman Table

Starting with 0 at $(0, 0)$, fill the extra row and column with progressive GAP penalties as follows [In the Smith-Waterman Algorithm, all negative numbers become 0]:

		A	**T**	**G**	**C**	**T**
	0	0	0	0	0	0
A	0					
G	0					
C	0					
T	0					

3.12.2 Filling the Smith-Waterman Table

Now starting at $(1, 1)$, and going row-wise, fill each cell with the max of the following, converting any negative values to 0:

1. Value from left + GAP
2. Value from above + GAP
3. Value from diagonal + ($Match$ or $Mismatch$) [depending on wether the row and column label are the same letter]

So, for the $(1, 1)$ cell, we fill it with $max(-4, -4, 1) = 1$.
And for the $(1, 2)$ cell, we fill it with $max(-1, -6, -3) = -1 \rightarrow 0$.
And so on...
Until we get:

		A	**T**	**G**	**C**	**T**
	0	0	0	0	0	0
A	0	1	0	0	0	0
G	0	0	0	1	0	0
C	0	0	0	0	2	0
T	0	0	1	0	0	3

3.12.3 Traceback in the Smith-Waterman Table

Starting from the highest value in the matrix, move diagonally backwards until any 0 is reached.
For each diagonal movement, align the coresponding characters.
Note, the characters are aligned in reverse.
A sample python implementation is given in Figure 3.24

```python
def smith_waterman(seq1, seq2, match=1, mismatch=-1, gap=-2):
    m, n = len(seq2), len(seq1)
    score = [[0] * (n + 1) for _ in range(m + 1)]

    # Initialize the first row and column
    for i in range(m + 1):
        score[i][0] = 0
    for j in range(n + 1):
        score[0][j] = 0

    # Fill in the scoring matrix
    max_score = 0
    max_position = (0, 0)

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            match_mismatch = match if seq2[i - 1] == seq1[j - 1]
                else mismatch
            diagonal = score[i - 1][j - 1] + match_mismatch
            horizontal = score[i][j - 1] + gap
            vertical = score[i - 1][j] + gap

            score[i][j] = max(0, diagonal, horizontal, vertical)

            if score[i][j] > max_score:
                max_score = score[i][j]
                max_position = (i, j)

    # Traceback to find the alignment
    align1, align2 = "", ""
    i, j = max_position

    while i > 0 and j > 0 and score[i][j] > 0:
        if score[i][j] == score[i - 1][j - 1] + (match if seq2[i -
            1] == seq1[j - 1] else mismatch):
            align2 = seq2[i - 1] + align2
            align1 = seq1[j - 1] + align1
            i -= 1
            j -= 1
        elif score[i][j] == score[i][j - 1] + gap:
            align2 = seq2[i - 1] + align2
            align1 = "-" + align1
            j -= 1
        else:
            align2 = "-" + align2
            align1 = seq1[j - 1] + align1
            i -= 1

    return align1, align2
```


Figure 3.24: The Needleman Wunsch Algorithm Python Implementation

Chapter 4

Benchmarking of the Researched Algorithms

This section presents a comparative analysis of dynamic programming algorithms, focusing on their computational efficiency. Using Python's timeit module, we conduct timed experiments to measure the execution performance of these algorithms across varying input sizes and scenarios. We aim to evaluate the scalability and practical effectiveness of these algorithms in solving real-world computational problems.

The Timeit Module

This module provides a simple way to time small bits of Python code. It has both a Command-Line Interface as well as a callable one. It avoids a number of common traps for measuring execution times. See also Tim Peters' introduction to the "Algorithms" chapter in the second edition of Python Cookbook, published by O'Reilly.
The following are some of the common traps for measuring execution time that timeit avoids:

- Warm-up Time: The Timeit module runs the code multiple times (by default, 1 million times) to ensure that any overhead related to JIT compilation or other one-time operations is amortized and doesn't skew the timing results. This helps ensure that the measured time reflects the actual performance of the code snippet, rather than the overhead of the Python interpreter starting up or other initialization tasks.

- External Factors: Timeit executes the code in a controlled environment, minimizing the impact of external factors such as other processes running on the system, I/O operations, or system load. This helps ensure that the timing measurements are as consistent and reliable as possible.

- Garbage Collection: The Timeit module disables the garbage collector during timing measurements to prevent it from interfering with the results. Garbage collection can introduce variability in execution times, especially for short code snippets, so disabling it helps ensure more consistent timing measurements.

- Timer Resolution: Timeit uses high-resolution timers to measure execution times accurately, taking into account the system's timer resolution. This helps avoid inaccuracies that can arise from low-resolution timers, especially when measuring very short code snippets.

- Compilation: For code snippets that are executed multiple times, timeit compiles the code to bytecode to speed up execution. This compilation step is done automatically and transparently, helping to reduce overhead and improve the accuracy of timing measurements.

4.0.1 Discussion of Benchmarking Methods of Algorithms with Inputs of Size N

When benchmarking algorithms with a variable input size, we cannot simply run the code multiple times on the same fixed input, as we cannot be sure if this input accurately reflects the average runtime of the algorithm. Instead, on each run, the input size is fixed but the input itself is randomized. If we do this enough times, we should converge on the actual average runtime of the algorithm. To help demonstrate this point, we can look at the worst case runtime of Quicksort $O(n^2)$, compared to its average case runtime $O(nlogn)$. If we were to test Quicksort vs Mergesort on a list that's sorted in reverse order, Mergesort would almost certainly come out on top, which is not indicative of the actual average runtimes of the algorithms. Therefore, we must somehow randomize the input on each test run, and get an average performance instead. When benchmarking the researched algorithms, we will use the following approaches in an attempt to get a more accurate view of the actual average runtimes on random examples.

4.0.2 Approach to Benchmarking the Fibonacci Problem

In order to benchmark the Fibonacci Problem, we do not need any randomization. We can simply input n, the index of the fibonacci number to be computed, and calculate the amount of time each of the brute force, memoization and tabulation approaches take to solve for $fib(n)$. The code used for benchmarking the different implementations of this problem is shown in Figure 4.1.

```
1    def benchmark_fib(funcs, n, repetitions):
2        res = [-1] * len(funcs)
3        res_idx = 0
4        for func in funcs:
5            total_time = 0
6            for _ in range(repetitions):
7
8                execution_time = timeit.timeit(lambda: func(n),
                     number=1000)
9
10               total_time += execution_time
11
12           average_time = total_time / repetitions
13           res[res_idx] = average_time
14           res_idx += 1
15       return res
```

Figure 4.1: Python Code Used for Benchmarking the Fibonacci Implementations

4.0.3 Results of Benchmarking the Fibonacci Problem

n=	5	10	25	35	50	1000
Brute Force	0.000092	0.012308	1.365216	145.007681	*	*
Memoization	0.000218	0.000154	0.000231	0.000238	0.000275	0.000303
Tabulation	0.000867	0.001471	0.003473	0.004803	0.005291	0.124944
Optimized	0.000519	0.000534	0.001210	0.001620	0.002763	0.053696

Table 4.1: Results of Benchmarking the Fibonacci Problem

*=over 10 mins to run on an MSI GF63 SCXR.
Results are rounded to 6 decimal places.

4.0.4 Approach to Benchmarking the Coin Change Problem

In order to benchmark the Coin Change Problem, we must randomize a list of coin denominations D of fixed size, and a total amount a, in order to get an accurate insight into the average runtime. The maximum and minimum size of the coin denominations and the total amount should be kept consistent throughout test runs to reduce the amount of variance. Ideally, the inputs to each of the brute force, memoization and tabulation algorithms should be the same on each run to avoid the case where one of the algorithms "gets lucky" and consistently gets "easier" inputs than the others. For simplicity, we se the target amount a to the length of D.

NOTE: Make sure D_min is NOT 0, as having a zero size coin will make the brute force and memoization solution non-terminable. The code used for benchmarking the different implementations of this problem is shown in Figure 4.2.

```
    def benchmark_coin_change(funcs, D_len, D_min, D_max, a_min,
        a_max, repetitions):
         res = [-1] * len(funcs)
         res_idx = 0
         for func in funcs:
             total_time = 0
             for _ in range(repetitions):

                 random_D = [random.randint(D_min, D_max) for _ in
                     range(D_len)]

                 random_a = random.randint(a_min, a_max)

                 execution_time = timeit.timeit(lambda: func(random_D,
                     random_a), number=1)

                 total_time += execution_time

             average_time = total_time / repetitions
             res[res_idx] = average_time
             res_idx += 1
         return res
```

Figure 4.2: Python Code Used for Benchmarking the Coin Change Implementations

4.0.5 Results of Benchmarking the Coin Change Problem

—D—=	3	5	10	15	17	1000
Brute Force	0.000026	0.000129	0.002109	2.391904	*	*
Memoization	0.000013	0.000024	0.000070	0.000040	2.391904	0.329376
Tabulation	0.000017	0.000012	0.000033	0.000016	0.000060	0.243787

Table 4.2: Results of Benchmarking the Coin Change Problem

*=over 10 mins to run on an MSI GF63 Thin SCXR.
Results in seconds rounded to 6 decimal places.

4.0.6 Approach to Benchmarking Problems With Single List Inputs

In order to benchmark problems with single list inputs, we must randomize a list nums of fixed size in order to get an accurate insight into the average runtime. The maximum and minimum size of the elements of *nums* should be kept consistent throughout test runs to reduce the amount of variance. Ideally, the inputs to each of the brute force, memoization and tabulation algorithms should be the same on each run to avoid the case where one of the algorithms "gets lucky" and consistently gets "easier" inputs than the others. For simplicity, the max number size in *nums* is set to $|nums|$, and the minimum is set to 1 or $-1 * |nums|$ (depending on the constraints of the problem). The code used for benchmarking the different implementations of problems of this type is shown in Figure 4.3.

```python
    def benchmark_subsequence(funcs, nums_len, nums_min, nums_max,
        repetitions):
        res = [-1] * len(funcs)
        res_idx = 0
        for func in funcs:
            total_time = 0
            for _ in range(repetitions):

                random_nums = [random.randint(nums_min, nums_max) for
                    _ in range(nums_len)]

                execution_time = timeit.timeit(lambda:
                    func(random_nums), number=1)

                total_time += execution_time

            average_time = total_time / repetitions
            res[res_idx] = average_time
            res_idx += 1
        return res
```

Figure 4.3: Python Code Used for Benchmarking Subsequence Problem Implementations

4.0.7 Results of Benchmarking Longest Increasing Subsequence

—nums—=	5	10	25	50	100	1000
Brute Force	0.000008	0.000071	0.001841	0.118709	53.019671	*
Memoization	0.000012	0.000040	0.000182	0.000714	0.002965	0.461804
Tabulation	0.000006	0.000011	0.000043	0.000162	0.000664	0.059541

Table 4.3: Results of Benchmarking Longest Increasing Subsequence

*=over 10 mins to run on an MSI GF63 Thin SCXR.
Results in seconds rounded to 6 decimal places.

4.0.8 Results of Benchmarking Max Subarray Sum

—nums—=	5	25	100	1000	10000	1000000
Brute Force	0.000007	0.000071	0.001073	0.076034	7.504696	*
Kadanes	0.000002	0.000005	0.000027	0.000158	0.001591	0.016662

Table 4.4: Results of Benchmarking Max Subarray Sum

*=over 10 mins to run on an MSI GF63 Thin SCXR.
Results in seconds rounded to 6 decimal places.

4.0.9 Results of Benchmarking Longest Alternating Subsequence

—nums—=	10	20	30	50	1000	10000
Brute Force	0.001798	2.723978	*	*	*	*
Auxiliary	0.000013	0.000043	0.000146	0.000379	0.106903	11.012398
Optimized	0.000003	0.000003	0.000008	0.000018	0.000106	0.001133

Table 4.5: Results of Benchmarking Longest Alternating Subsequence

*=over 10 mins to run on an MSI GF63 Thin SCXR.
Results in seconds rounded to 6 decimal places.

4.0.10 Approach to Benchmarking the Binomial Coefficients Problem

In order to benchmark the Binomial Coefficients Problem, we do not need any randomization. We can simply input n and k, and calculate the average time each of the brute force, memoization and tabulation approaches take to solve for $C(n, k)$. For simplicity, we take k as $n/2$. The code used for benchmarking the different implementations of this problem is shown in Figure 4.4.

```
1   def benchmark_binomial_coefficients(funcs, n,k, repetitions):
2       res = [-1] * len(funcs)
3       res_idx = 0
4       for func in funcs:
5           total_time = 0
6           for _ in range(repetitions):
7
8               execution_time = timeit.timeit(lambda: func(n,k),
                    number=1)
9
10              total_time += execution_time
11
12          average_time = total_time / repetitions
13          res[res_idx] = average_time
14          res_idx += 1
15      return res
```

Figure 4.4: Python Code Used for Benchmarking the Binomial Coefficients Implementations

4.0.11 Results of Benchmarking Binomial Coefficients

n=	10	20	50	1000	10000
Brute Force	0.000056	0.034847	*	*	+
Memoization	0.000001	0.000001	0.000061	0.020484	+
Tabulation	0.000013	0.000030	0.000212	0.078734	+
Optimized	0.000015	0.000020	0.000118	0.033395	+

Table 4.6: Results of Benchmarking Binomial Coefficients

*=over 10 mins to run on an MSI GF63 Thin SCXR.
+=maximum recursion depth reached/crash.
Results in seconds rounded to 6 decimal places.

Chapter 5

Multi Dimensional Dynamic Programming Algorithms

5.1 Unique Paths

Problem Statement: A robot who's location on a grid is denoted R is located at the top-left corner of a m x n grid. The robot is trying to reach the finish square at the bottom right corner of the grid donoted as F. How many possible unique paths to F exist starting at R, given the constraint that the robot can only move right one square, or down one square at any given point in time?

Input: Two integers m and n, which are the dimensions of the grid.

Output: An integer $unique\_paths(m, n)$.

Example: For:

$m = 3$, $n = 7$

$unique\_paths(m, n) = 28$

Explanation: There are 28 unique ways the robot can reach F from R. Example Path:

R	>	V				
		>	>	>	>	V
						F

5.1.1 Tabulation Approach to Unique Paths

Notice that wherever the robot lands on the grid, it is possible to reach the finish square, so we do not have to worry about cases where the robot cannot reach the finish square (backtracking). Notice also that the number of paths from any square is the sum of the number paths to the right, and the number of paths if you go down. We can use tabulation to solve the problem, by creating an m x n table called dp where the value at $dp[i][j]$ represents the number of unique paths from that square to F. The value at $dp[0][0]$ will hence contain the solution to the problem. We can then initialize $dp[F]$ to 1, as there is one path from F to itself (the path contains zero moves). The square directly above and to the left of $dp[F]$ can also be initialized to 1, as there is one path from them to F, going down or right respectively. This logic follows for all of the bottom row of the grid, and the last column of the grid.

						1
						1
	1	1	1	1	1	1

Now we simply fill the grid from the bottom right to the top left, where the value of each square is the sum of the value below it and to its right.

28	21	15	10	6	3	1
7	6	5	4	3	2	1
1	1	1	1	1	1	1

The top left square is where the robot was, so that is the number of paths from the robot to the finish square.

A sample Python implementation is given in 5.1

```
1   def unique_paths(m,n):
2
3       # Initialize a m x n table of 1s
4       dp = [[1] * n for _ in range(m)]
5
6       # Fill in the table in a bottom-up manner
7       for i in range(m-1,-1,-1):
8           for j in range(n-1,-1,-1):
9               # Leave last row and column as 1s
10              if i == m-1 or j == n-1:
11                  continue
12              # Fill all other squares with the sum of the square
                    below and to the right
13              else:
14                  dp[i][j] = dp[i+1][j] + dp[i][j+1]
15
16      return dp[0][0]
```

Figure 5.1: Unique Paths Python Implementation

5.1.2 Complexity Analysis of Unique Paths

Time Complexity: The time complexity of filling an m x n table with values, where each value is calculated by two constant time lookups and a constant time addition, is $O(m * n)$.

Space Complexity: The space complexity of storing an m x n table where each field in the table contains a single integer is $O(m * n)$.

Overall: Total:

Time Complexity: $O(m * n)$

Space Complexity: $O(m * n)$

5.1.3 Optimization of the Unique Paths Problem

Notice that for a given value in row r in the table dp, v can be calculated using just row r and row $r-1$. Since we compute the values in dp rowwise from the bottom up, we do not actually need to store the entire dp table in memory as the values which are two or more rows below the row we are computing at any given time do not contribute to our calculation. Storing only two rows of the dp table at a time reduces the space complexity from $O(m * n)$ to just $O(m)$.

A Python implementation of this optimization is given in Figure 5.2.

```
1   def unique_paths_optimized(m,n):
2       # Initialize the bottom row of 1s
3       oldRow = [1] * n
4
5       # For each subsequent row in the grid
6       for _ in range(m-1):
7           # Create a new row which is initialized to all 1s
8           newRow = [1] * n
9           # Traverse the new row in reverse, without the last element
10          for j in range(n - 2, -1, -1):
11              # Each value in the new row is the sum of the value to
                    the right and below
12              newRow[j] = newRow[j+1] + oldRow[j]
13          oldRow = newRow
14
15      return oldRow[0]
```

Figure 5.2: Unique Paths Optimized Python Implementation

5.2 Minimum Path Sum

Problem Statement: Given a 2D array filled with non-negative numbers representing the 'cost' at that field in the input matrix, find the cost of the path from the top-left corner (denoted R) to the bottom-right corner (denoted F) which minimizes the sum of numbers along the path. (minimizes path cost). You can only move down or to the right at any point in time (no path can make negative progress).

Input: A 2D array, called $cost$.

Output: An integer $min_path_sum(cost)$, which represents the cost of the minimum path from R to F.

Example: For:

$cost =$

1	3	1
1	5	1
4	2	1

$min_path_sum(cost) = 7$

Explanation: Explanation: $1 + 3 + 1 + 1 + 1 = 7$ is the minimum path sum.

R	>	V
		V
		F

5.2.1 Tabulation Approach to Minimum Path Sum

We can solve this problem in a very similar manner to Unique Paths. We start by creating a table dp with the same dimensions as the input matrix, where $dp[i][j]$ represents the minimum path cost from $dp[i][j]$ to F. We can initialize F to be it's own cost in the input matrix (the cost of moving from F to F is $cost[F]$).

		1

Now we can iterate through the input matrix backwards using the same logic as in Unique Paths, but this time:

1. If we are on the last column: $dp[i][j] = dp[i][j+1] + cost[i][j]$. [This is because there is no more squares to go right, so we must go down and incur that cost]
2. If we are on the last row: $dp[i][j] = dp[i+1][j] + cost[i][j]$. [This is because there is no more squares to go down, so we must go right and incur that cost]
3. Otherwise, $dp[i][j] = cost[i][j]+$ the minimum cost of going down, or going right.

The top left of the grid will give us the minimum cost of reaching the bottom right position.

7	6	3
8	7	2
7	3	1

A sample Python Implementation of this is given in Figure 5.3

```python
def min_path_sum(cost):
    rows, cols = len(cost), len(cost[0])

    # Initialize a table to store minimum path sums
    dp = [[0] * cols for _ in range(rows)]

    # Fill in the table in a bottom-up manner
    for i in range(rows-1,-1,-1):
        for j in range(cols-1,-1,-1):
            # Initialize the bottom right square to its own cost
            if i == rows-1 and j == cols-1:
                dp[i][j] = cost[i][j]

            # When we are on the last row or col
            elif i == rows-1:
                dp[i][j] = dp[i][j+1] + cost[i][j]
            elif j == cols-1:
                dp[i][j] = dp[i+1][j] + cost[i][j]

            # Otherwise, choose the minimum cost path and add its
               cost
            # to the cost of this square.
            else:
                dp[i][j] = cost[i][j] + min(dp[i+1][j], dp[i][j+1])

    return dp[0][0]
```

Figure 5.3: Min Path Sum Python Implementation

5.2.2 Complexity Analysis of Min Path Sum

Let m be the number of rows in *cost* (the input matrix), and n be the number of columns in *cost*.

Time Complexity: The time complexity of filling an m x n table with values, where each value is calculated by two constant time lookups and a constant time addition, is $O(m * n)$.

Space Complexity: The space complexity of storing an m x n table where each field in the table contains a single integer is $O(m * n)$.

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

5.2.3 Optimization of Minimum Path Sum

We can use the exact same optimization as we did with Unique Paths, as each value in row r can be calculated from just the values in row r and row $r - 1$. Storing only two rows at a given time will reduce the Space complexity to $O(m)$.

A sample Python implementation is given in Figure 5.4

```python
    def min_path_sum_optimized(cost):
        rows, cols = len(cost), len(cost[0])

        # Initialize the bottom row of 0s
        oldRow = [0] * cols

        # For each subsequent row in the grid
        for i in range(rows-1, -1, -1):
            # Create a new row which is initialized to all 0s
            newRow = [0] * cols
            # Traverse the new row in reverse, same logic as before.
            for j in range(cols - 1, -1, -1):
                if i == rows-1 and j == cols-1:
                    newRow[j] = cost[i][j]
                elif i == rows-1:
                    newRow[j] = newRow[j+1] + cost[i][j]
                elif j == cols-1:
                    newRow[j] = oldRow[j] + cost[i][j]
                else:
                    newRow[j] = cost[i][j] + min(oldRow[j],
                        newRow[j+1])
            oldRow = newRow

        return oldRow[0]
```

Figure 5.4: Min Path Sum Optimized Python Implementation

5.2.4 Note on Optimization

This two-row framework can be used to solve any pathing problem which is constrained such that negative progress is not possible. This optimization can be taken even further. We have solved both problems rowwise starting from the bottom. This problem can equally be solved columnwise, using the same optimization except this time storing two columns instead of two rows. If we solve these problems rowwise in the case of $m \leq n$, and columnwise if $m > n$, we actually reduce the space complexity even further to $O(min(m, n))$.

5.3 Extending Unique Paths and Min Path Sum to 3 Dimensions

So far we have seen that Unique Paths and Min Path Sum can be solved for 2D array inputs. These problems can also be trivially solved for 1D array inputs, and single scalar inputs. But what if the input to the problem is not a 2D array but a 3D array? (But the constraint which prevents negative progress is still in place).

5.3.1 Tabulation Approach to 3D-Unique Paths

Consider an m x n x k array called *input*, where we start at the top left at index $(0, 0, 0)$ (denoted R), and our target is in the bottom right at index $(m-1, n-1, k-1)$ (denoted F). We can only move down at the current depth, right at the current depth, or "in" (meaning we increase the depth we are at). The problem, like before, is to find the number of distinct paths from R to F. We can use a similar approach as with the 2D problem, but this time dp will need to be a 3D array to store the intermediate values. Instead of summing the value to the right and below to get our current value, we sum the value to the right, below, and at the next depth level. The value at index $dp[i][j][t]$ represents the number of unique paths from $input[i][j][t]$ to $input[m-1][n-1][k-1]$. We will still initialize $dp[m-1][n-1][k-1]$ to 1. To calculate the value at $dp[i][j][t]$, we will sum up $dp[i+1][j][t]$, $dp[i][j+1][t]$ and $dp[i][j][t+1]$, unless the value is at a border, in which case we we will exclude that border from the sum, because going further in that direction is not possible[1]. The value at $dp[0][0][0]$ will be our solution.

A sample python implementation is shown in Figure 5.5.

```
1   def unique_paths_3d(m, n, k):
2       dp = [[[0 for _ in range(k)] for _ in range(n)] for _ in
            range(m)]
3
4       dp[m - 1][n - 1][k - 1] = 1
5
6       for i in range(m - 1, -1, -1):
7           for j in range(n - 1, -1, -1):
8               for t in range(k - 1, -1, -1):
9                   if i + 1 < m:
10                      dp[i][j][t] += dp[i + 1][j][t]
11                  if j + 1 < n:
12                      dp[i][j][t] += dp[i][j + 1][t]
13                  if t + 1 < k:
14                      dp[i][j][t] += dp[i][j][t + 1]
15
16      return dp[0][0][0]
```

Figure 5.5: 3D Unique Paths Python Implementation

5.3.2 Complexity Analysis of 3D-Unique-Paths

Time Complexity: We have proved that the time complexity of 2D Unique Paths is $O(m*n)$ in Section 5.1.2. Similarly, we can see that the time complexity of 3D Unique Paths is $O(m*n*k)$.

Space Complexity: We have proved that the space complexity of 2D Unique Paths is $O(m*n)$ in Section 5.1.2. Similarly, we can see that the space complexity of 3D Unique Paths is $O(m*n*k)$.

Overall: Total:

Time Complexity: $O(m*n*k)$

Space Complexity: $O(m*n*k)$

[1] eg, if $i+1=m$, we exclude $dp[i+1][j][t]$ from the sum as $dp[i+1][j][t]$ is out of bounds.

5.3.3 Note on this Approach

We can trivially convert this 3D Unique Paths algorithm into a 3D Min Path Sum solution, and in fact we can convert any 2D problem with the constraint that negative progress is not allowed into a 3D problem using this approach.

5.4 A Framework for Solving N-Dimensional Pathfinding Problems

We can see that extending a K-Dimensional pathfinding problem where no negative progress is allowed (similar to Unique Paths or Min Path Sum), we can extend it to a K+1-Dimensional problem by simply considering a single new 'direction' in our calculation. Because of this, we can solve any pathfinding problem from any corner R of a rank N tensor to the opposite corner F (where no negative progress is allowed), by simply initializing an N dimensional table dp, initializing $dp[F]$ and each field on a border in dp to an appropriate value[2], and deducing the value of an arbitrary field v by doing a constant time operation (such as sum or max) of each 'surrounding' value in a direction opposite to the direction you are allowed to travel in that dimension.

5.4.1 N-Dimensional Unique Paths Python Implementation Details

This implemetation is tricky as we must simulate n for loops where n is the dimension count. Numpy was used to access, store data in, and manipulate the input tensor as well as the dp table. This choice was made because numpy allows us to access elements in n-rank tensors by providing a tuple of length n as an index. Numpy also allows us to easily initialialize n-rank tensors through the use of its $np.zeros(shape)$ function, rather than having to use something similar to Figure 5.6:

```
1    def initialize_array(dimensions):
2        if len(dimensions) == 1:
3            return [0] * dimensions[0]
4        else:
5            return [initialize_array(dimensions[1:]) for _ in
                 range(dimensions[0])]
```

Figure 5.6: Example code to generate an n-rank tensor of zeros without Numpy

Numpy's $np.ndindex()$ is a function that provides an iterator yielding tuples of indices for a given shape. It is particularly useful for iterating over the indices of n-dimensional arrays. This function returns an object that can be iterated over, generating all possible index tuples for a specified shape.

The pseudocode of the algorithm using numpy is as follows:

1. Initialize an n-dimensional array of zeros called dp to store the number of paths for each cell, with $dp[R]$ set to 1.
2. The $np.ndindex(shape)$ generates an iterator over all possible indices in the n-dimensional array.
3. For each index, represented by the $current\_cell$:
4. Iterate over each dimension using for i in $range(num_dimensions)$.

[2] 1 in the case of Unique Paths, or $cost[i]$ in Min Path Sum

5. Check if the current cell's index in the current dimension ($current\_cell[i]$) is greater than 0. If true, it means there is a valid cell to move from in that dimension.

6. Create a copy of the current cell (denoted $prev\_cell$) and decrement the index in the current dimension ($prev\_cell[i] -= 1$). This represents the cell from which we are coming.

7. Add the number of paths from the previous cell to the current cell in the dp array ($dp[index] += dp[tuple(prev\_cell)]$).

5.4.2 Python Implementation of N-Dimensional Unique Paths

Figure 5.7 shows a Python Implementation of N-Dimensional Unique Paths using the framework discussed above. The code is heavily commented so we can see the framework in action. Note that this approach can be trivially converted to calculate the Min Path Sum, or any other pathfinding problem where negative progress is not allowed.

```python
import numpy as np

def unique_paths_nd(dimensions):
    # Determine the number of dimensions
    num_dimensions = len(dimensions)

    # Initialize an n-dimensional array to store the number of paths for each cell
    shape = tuple(dimensions)
    dp = np.zeros(shape, dtype=int)

    # Set the number of paths for the starting cell to 1
    dp[(0,) * num_dimensions] = 1

    # Calculate the number of paths for each cell in the matrix
    for index in np.ndindex(shape):
        # Get the current index as an array
        current_cell = np.array(index)
        # Iterate over the array
        for i in range(num_dimensions):
            # If the current cell is not on the edge
            if current_cell[i] > 0:
                # Add the previous cell's paths to the current cell [this happens from each valid direction in each dimension]
                prev_cell = current_cell.copy()
                prev_cell[i] -= 1
                dp[index] += dp[tuple(prev_cell)]

    # The result is stored in the last cell of the matrix
    return dp[tuple(np.array(dimensions) - 1)]

# Example usage:
dimensions = (2,7,5)
print(unique_paths_nd(dimensions))
```

Figure 5.7: Python Implementation of N-Dimensional Unique Paths