1 Introduction

The timber problem involves a log of wood that is divided into smaller logs of varying sizes. The goal is to find the maximum amount of timber that can be obtained by an individual. In its most basic form, the problem can be solved using a recursive algorithm. However, the recursive algorithm has an exponential time complexity of $O(2^n)$, making it inefficient for large input sizes. To address this issue, a dynamic programming (DP) algorithm can be used to solve the timber problem in $\Theta(n^2)$ time complexity. This report presents the theoretical analysis of the timber problem, the implementation of the recursive and DP algorithms, and an experimental analysis to validate the theoretical results.

2 Basic Recursive Algorithm

```python
def timber_recursive(log_sizes):
    # Base Case
    if len(log_sizes) == 1:
        return log_sizes[0]

    # Recursive Case
    return sum(log_sizes) - min(timber_recursive(log_sizes[1:]),
                                  timber_recursive(log_sizes[:-1]))
```

The timber problem is solvable using a recursive algorithm. With each recursive call, 2 subproblems are created, one with the first log removed and one with the last log removed. Thus the total operations for the recursive algorithm is: $O(2^n)$.

$$1 + 2 + 2^2 + 2^3 + ... + 2^n = \Theta(2^n) \tag{1}$$

3 DP Algorithm (Bottom-Up)

```
def timber_bottom_up(log_sizes):
    '''
    This function solves the timber problem using a bottom-up approach
    :param log_sizes: A list of log sizes
    :return: The maximum value that can be obtained from cutting the logs
    '''
    n = len(log_sizes)
    # Prefix sum of the log sizes
    sums = [0] * n
    for i in range(n):
        sums[i] = log_sizes[i] + (sums[i - 1] if i > 0 else 0)

    # Create a table to store the results and fill in the base case (i == j)
    table = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        table[i][i] = log_sizes[i]

    # Fill in the table using the bottom-up approach (traverse diagonally)
    for diag in range(1, n):
        for i in range(n - diag):
            j = i + diag
            table[i][j] = sums[j] - (sums[i - 1] if i > 0 else 0) - \
                min(table[i + 1][j], table[i][j - 1])

    return table[0][n - 1]
```

The DP bottom-up algorithm solves the timber problem by creating a table to store the results of subproblems. The algorithm fills in the table using a bottom-up approach, starting with the base case (i == j) and then traversing diagonally to fill in the remaining entries. The time complexity of the DP bottom-up algorithm is $\Theta(n^2)$.

At the start of execution, a prefix sum of the log sizes is calculated in $O(n)$ time complexity. The table is then initialized with the base case in $O(n)$ time complexity. The table is filled in using a nested loop that traverses diagonally, with each entry taking $O(1)$ time complexity to calculate and a total of $O(\frac{n^2}{2} - n)$ operations. Thus, the overall time complexity of the DP bottom-up algorithm is $\Theta(n^2)$.

$$1 + 2 + 3 + ... + n = \Theta(n^2) \tag{2}$$

4 Experimental Analysis

To validate the theoretical analysis, the DP bottom-up algorithm presented in Section 3 was implemented in Python [1] and tested with various input sizes, $1 \leq n \leq 2000$. The python random library was used to generate random log sizes for each input size ranging from 1 to 1000. The algorithm was run 100 times for each size and the average time taken to solve the timber problem was recorded for each input size in Table 1 [2].

4.1 Experimental Results

Table 1: Average time taken to solve the timber problem using the DP bottom-up algorithm

Size	Average Time (ms)	Size	Average Time (ms)
100	0.838	1100	130.936
200	3.253	1200	155.425
300	7.784	1300	185.394
400	14.526	1400	214.237
500	24.288	1500	247.649
600	35.379	1600	283.764
700	49.560	1700	318.818
800	66.870	1800	360.112
900	85.758	1900	402.864
1000	106.546	2000	442.341

[1] Simulation ran on Apple M2 Pro with 16GB unified RAM
[2] Only some of the results are present in the table. The actual simulation tested in intervals of 10

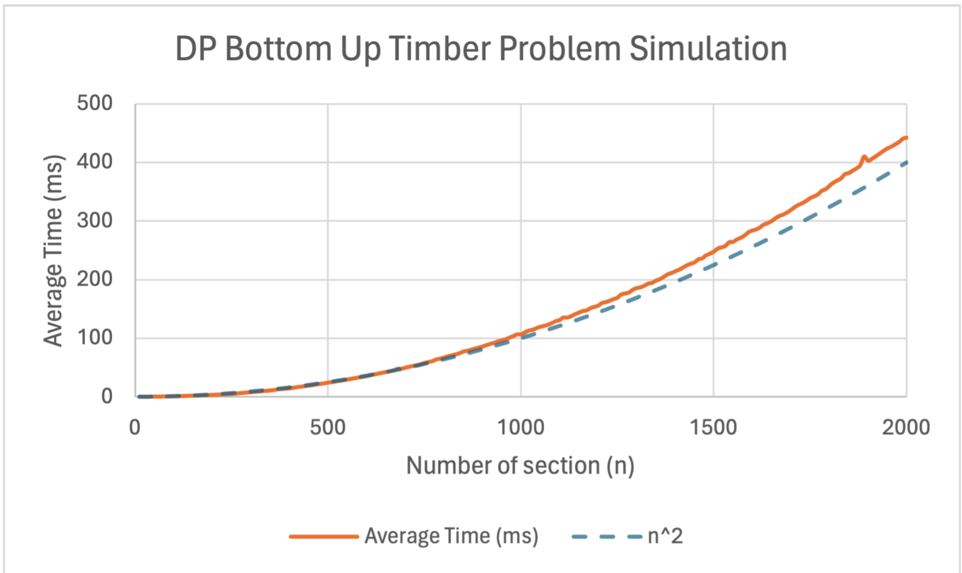

Figure 1: Average time taken to solve the timber problem using the DP bottom-up algorithm

4.2 Analysis

The experimental results depicted in Table 1 and Figure 1 showcase a clear trend: the average time taken to solve the timber problem using the DP bottom-up algorithm grows quadratically with the input size.

As shown in Figure 1, the experimental data closely follows the theoretical analysis of $\Theta(n^2)$[3]. Despite minor fluctuations in runtime, likely attributable to system variations and other factors, the overall trend exhibits a clear quadratic growth pattern.

[3] To provide a more illustrative comparison, a scaled line representing the theoretical complexity ($\Theta(n^2)$) is included in Figure 1. This scaled line, reduced by a factor of 10,000.

5 Appendix - Python Code

```python
import sys
import random
import time


def timber_recursive(log_sizes):
    '''
    This function solves the timber problem using a recursive approach
    :param log_sizes: A list of log sizes
    :return: The maximum value that can be obtained from cutting the logs
    '''
    # Base Case
    if len(log_sizes) == 1:
        return log_sizes[0]

    # Recursive Case
    return sum(log_sizes) - min(timber_recursive(log_sizes[1:]),
                                timber_recursive(log_sizes[:-1]))


def timber_bottom_up(log_sizes):
    '''
    This function solves the timber problem using a bottom-up approach
    :param log_sizes: A list of log sizes
    :return: The maximum value that can be obtained from cutting the logs
    '''
    n = len(log_sizes)
    # Prefix sum of the log sizes
    sums = [0] * n
    for i in range(n):
        sums[i] = log_sizes[i] + (sums[i - 1] if i > 0 else 0)

    # Create a table to store the results and fill in the base case (i == j)
```

```
    table = [[0 for _ in range(n)] for _ in range(n)]
    for i in range(n):
        table[i][i] = log_sizes[i]

    # Fill in the table using the bottom-up approach (traverse diagonally)
    for diag in range(1, n):
        for i in range(n - diag):
            j = i + diag
            table[i][j] = sums[j] - (sums[i - 1] if i > 0 else 0) - \
                min(table[i + 1][j], table[i][j - 1])

    return table[0][n - 1]


def command_line_input():
    '''
    This function reads the input file from the command line and runs
    the timber_bottom_up function
    :return: None
    '''
    # Get the first argument as the input file
    input_file = sys.argv[1]

    # Read the input file
    with open(input_file, 'r') as f:
        lines = f.readlines()
    f.close()

    # Get the sizes of the logs from the second line and run the algorithm
    log_sizes = list(map(int, lines[1].split()))
    print(timber_bottom_up(log_sizes))


def synthetic_test(max_size=20, test_count=100):
    '''
```

```
    This function runs a synthetic test on the timber_recursive function
    The results are written to an output file
    :param range: The range of log sizes to test from 1 to range (default is 20)
    :return: None
    '''
    with open("output_recursive.csv", "w") as f:
        for i in range(1, max_size + 1):
            print("Testing log size: ", i)
            total_time = 0
            for _ in range(test_count):
                # Use a random number generator to generate the log sizes
                log_sizes = [random.randint(1, 1001) for _ in range(i)]
                # Get the start time of the algorithm
                start_time = time.time()
                # Run the algorithm
                timber_recursive(log_sizes)
                # Get the end time of the algorithm
                end_time = time.time()
                # Calculate the time taken in milliseconds
                total_time += (end_time - start_time) * 1000
            # Write the results to the output file
            f.write(f"{i}, {total_time / test_count}\n")
    f.close()


def synthetic_test_bottom_up(max_size=2000, test_count=100):
    '''
    This function runs a synthetic test on the timber_bottom_up function
    The results are written to an output file
    :param range: The range of log sizes to test (default is 2000)
    :return: None
    '''
    with open("output_bottom_up.csv", "w") as f:
        for i in range(1, max_size + 1):
            print("Testing log size: ", i)
```

```
            total_time = 0
            for _ in range(test_count):
                # Use a random number generator to generate the log sizes
                log_sizes = [random.randint(1, 1001) for _ in range(i)]
                # Get the start time of the algorithm
                start_time = time.time()
                # Run the algorithm
                timber_bottom_up(log_sizes)
                # Get the end time of the algorithm
                end_time = time.time()
                # Calculate the time taken in milliseconds
                total_time += (end_time - start_time) * 1000
            # Write the results to the output file
            f.write(f"{i}, {total_time / test_count}\n")
    f.close()


def validate_methods():
    '''
    This function validates the timber_recursive and timber_bottom_up
    functions by comparing the results for random log sizes
    :return: None
    '''
    # Test random log lengths and random log sizes for 1000 iterations
    for _ in range(1000):
        if _ % 100 == 0:
            print(f"Testing iteration: {_}")
        log_sizes = [random.randint(1, 1001)
                     for _ in range(random.randint(1, 21))]
        # print if the results are not the same
        if timber_recursive(log_sizes) != timber_bottom_up(log_sizes):
            print("Results are not the same")
            print(log_sizes)
            print(timber_recursive(log_sizes))
            print(timber_bottom_up(log_sizes))
```

```
            return


if __name__ == "__main__":
    # Uncomment the line below to run the command line input
    command_line_input()  # This one is used for grading
    # synthetic_test()
    # synthetic_test_bottom_up()
    # validate_methods()
```