CPSC 335 - Project 2

Professor: Divyansh Mohan Rao
Course Meetings: M/W 11:30-12:45
Github Link: https://github.com/Vyadin/CPSC-335-Project-2

Group Members

Jacob Armstrong - jarmstrong31@csu.fullerton.edu
Phu Lam - lamphu75@csu.fullerton.edu
Haroutyun Chamelian - Harout.c@csu.Fullerton.edu

Time Complexity

Exhaustive Search: O(2^n)
Dynamic Programming: O(n)

Output Screenshots

```
============
= Sample 1 =
============

N: 4
Stocks and Values: [[1, 2], [4, 3], [5, 6], [6, 7]]
Amount: 12

<< Part A: Exhaustive Search >>
Maximum value: 11

<< Part B: Dynamic Programming >>
Maximum value: 11

============
= Sample 2 =
============

N: 4
Stocks and Values: [[3, 2], [4, 3], [5, 3], [6, 7]]
Amount: 10

<< Part A: Exhaustive Search >>
Maximum value: 12

<< Part B: Dynamic Programming >>
Maximum value: 12

============
= Sample 3 =
============

N: 5
Stocks and Values: [[1, 3], [10, 2], [4, 7], [7, 8], [9, 1]]
Amount: 12

<< Part A: Exhaustive Search >>
Maximum value: 26

<< Part B: Dynamic Programming >>
Maximum value: 26
```

```
============
= Sample 4 =
============

N: 6
Stocks and Values: [[2, 3], [5, 4], [6, 7], [7, 8], [8, 9], [9, 10]]
Amount: 14

<< Part A: Exhaustive Search >>
Maximum value: 14

<< Part B: Dynamic Programming >>
Maximum value: 14

============
= Sample 5 =
============

N: 4
Stocks and Values: [[2, 2], [4, 3], [6, 6], [6, 7]]
Amount: 12

<< Part A: Exhaustive Search >>
Maximum value: 12

<< Part B: Dynamic Programming >>
Maximum value: 12

============
= Sample 6 =
============

N: 3
Stocks and Values: [[1, 1], [5, 2], [10, 3]]
Amount: 5

<< Part A: Exhaustive Search >>
Maximum value: 15

<< Part B: Dynamic Programming >>
Maximum value: 15
```

```
============
= Sample 7 =
============

N: 2
Stocks and Values: [[100, 1], [60, 1]]
Amount: 1

<< Part A: Exhaustive Search >>
Maximum value: 100

<< Part B: Dynamic Programming >>
Maximum value: 100

============
= Sample 8 =
============

N: 5
Stocks and Values: [[5, 1], [3, 7], [8, 3], [50, 100], [3, 2]]
Amount: 6

<< Part A: Exhaustive Search >>
Maximum value: 16

<< Part B: Dynamic Programming >>
Maximum value: 16

============
= Sample 9 =
============

N: 4
Stocks and Values: [[2, 1], [3, 4], [6, 5], [7, 6]]
Amount: 12

<< Part A: Exhaustive Search >>
Maximum value: 15

<< Part B: Dynamic Programming >>
Maximum value: 15

=============
= Sample 10 =
=============

N: 3
Stocks and Values: [[10, 10], [5, 5], [4, 4]]
Amount: 10

<< Part A: Exhaustive Search >>
Maximum value: 10

<< Part B: Dynamic Programming >>
Maximum value: 10
```

Part A - Exhaustive Search

Algorithm Description: Determine maximum number of stocks that can be purchased within a specified budget. iterating through all possible combinations of available stocks and their values. For each combination, the function calculates the total value and compares it with the budget. If the total value is within the budget, the function updates the maximum number of stocks that can be bought. This method ensures that all possible stock combinations are considered, ensuring the most stocks are purchased without exceeding the budget.

Logic

1) Initialize Maximum Stock Count: Set stock_count to 0.

This represents the maximum number of stocks that can be purchased within the budget.

2) Iterate Over Possible Numbers of Stocks:

Loop through all numbers of stocks from 1 to n.

3) Generate Combinations of Stocks:

For each number of stocks, generate all possible combinations of that many stocks from the available stocks_and_values.

4) Calculate Total Value for Each Combination:

For each combination, calculate the total value of stocks in that combination.

5) Compare Total Value with Budget:

Check if the total value of the current combination is less than or equal to the budget.

6) Update Maximum Stock Count:

If the total value is within the budget, update stock_count with the number of stocks in the combination if it's higher than the current stock_count.

7) Return Maximum Stock Count:

After evaluating all combinations, return stock_count, which is the highest number of stocks purchasable within the given budget.

Part B - Dynamic Programming

Algorithm Description: Determine maximum value obtained with given money considering available stocks. List ‘dp’ of size amount + 1 stores maximum value for each amount. Iterate through sock and for each updated dp array to store maximum value obtained with current stock and money.

Logic

1) Initialize list +1 to display max stock to buy with different amount of money
2) Iterate through each stock item in stocks and values
3) Each item, iterate range of possible amount of money from amount down to cost of current stock item
4) Update dp array to find max value achievable with different amount of money
5) Return max value

Conclusion

Exhaustive search tends to be simpler to implement compared to the dynamic approach but it gets outweighed by the efficiency difference between the two. Exhaustive search is O(2^n), meaning that even with a sample size of 1, it still has a worse time complexity than dynamic programming. Given larger sample sizes, its time complexity grows exponentially while the time complexity of the dynamic programming implementation stays the same [O(n)]. With this information in mind, the dynamic programming implementation is better overall.