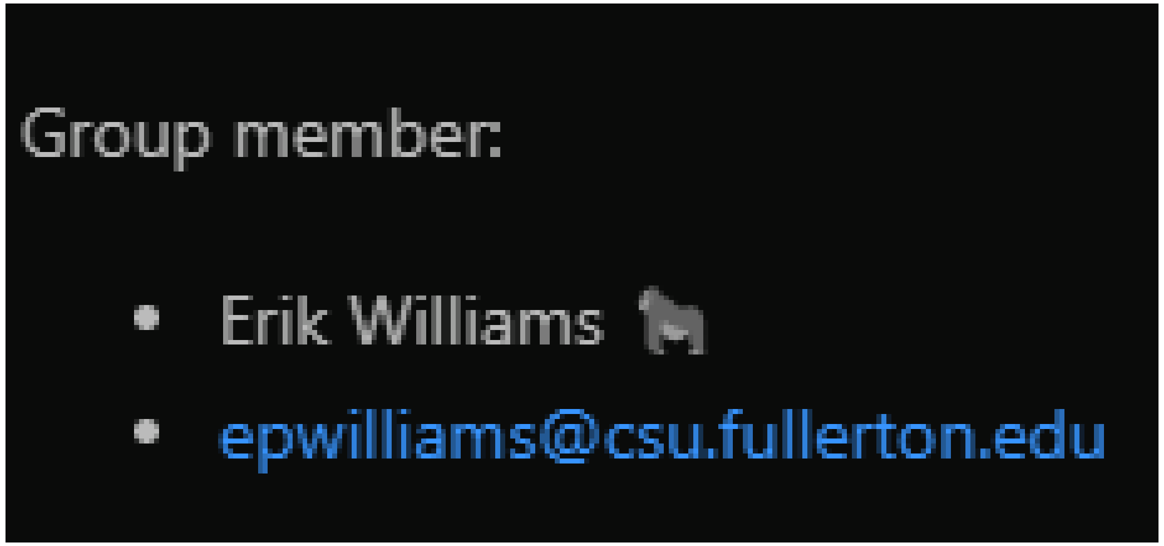
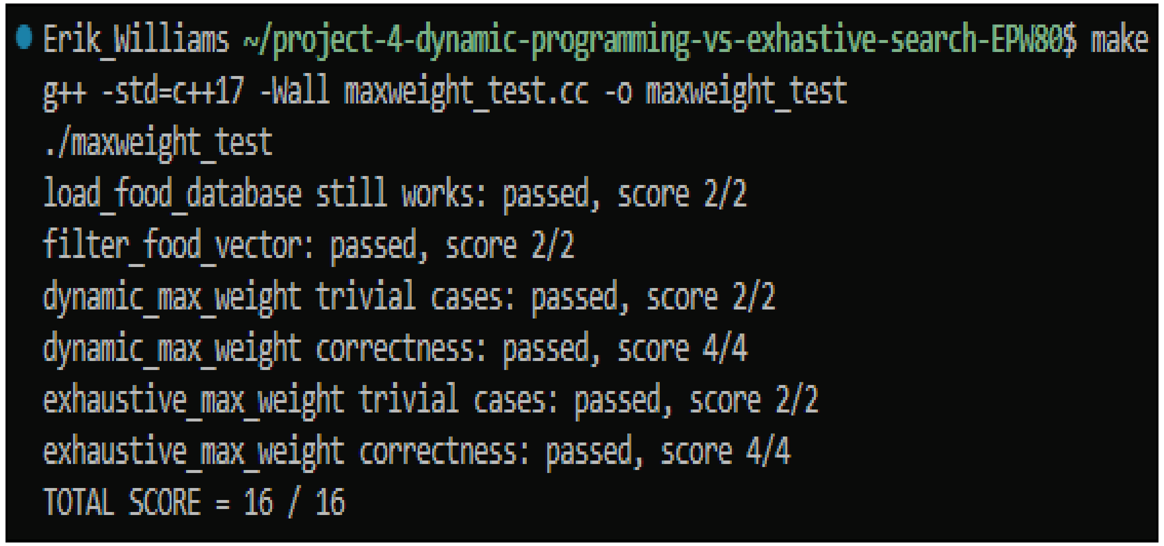
# Dynamic Programming VS Exhaustive Search



Group member:

- Erik Williams 🦍
- epwilliams@csu.fullerton.edu



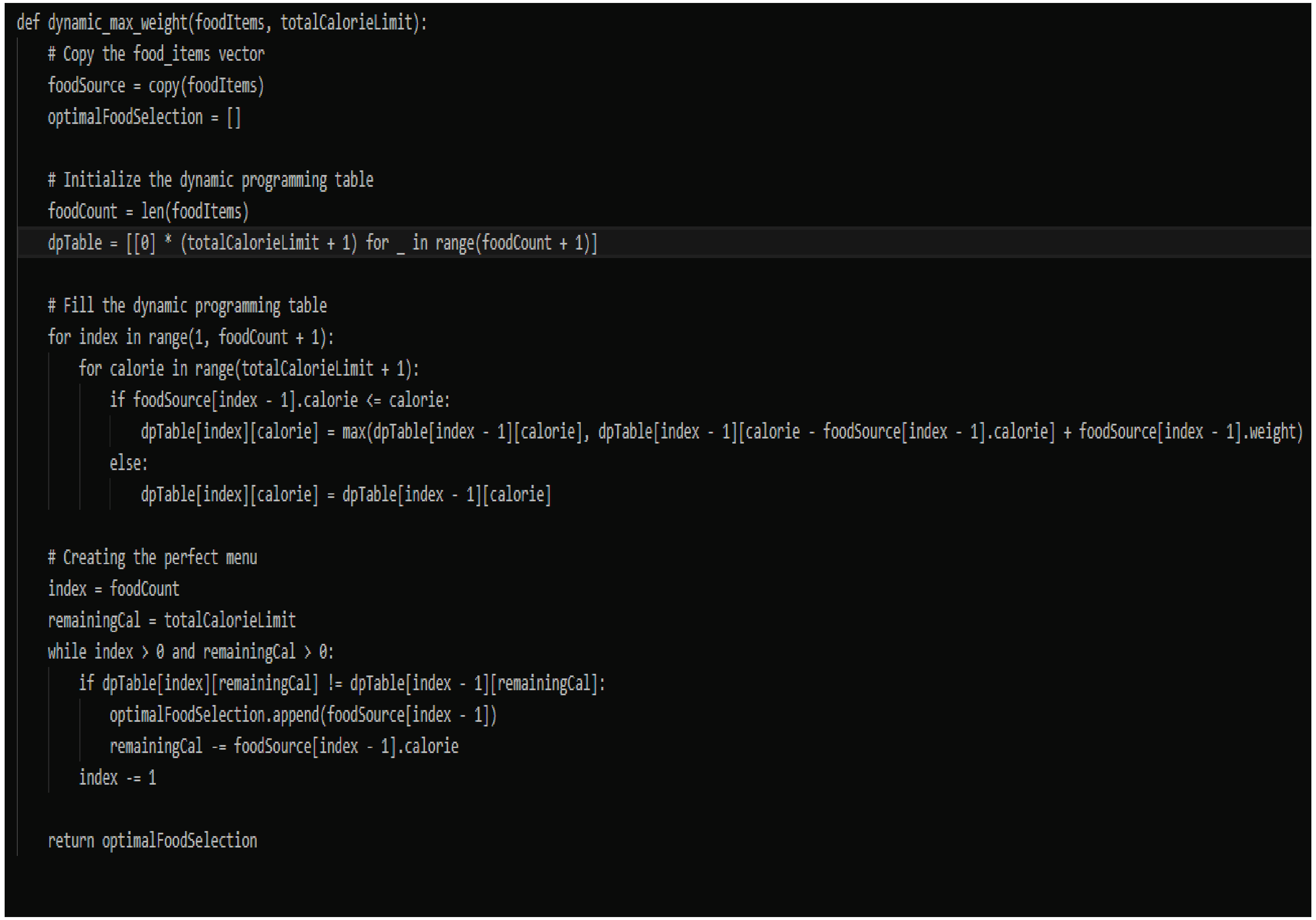
```
Erik_Williams ~/project-4-dynamic-programming-vs-exhastive-search-EPW80$ make
g++ -std=c++17 -Wall maxweight_test.cc -o maxweight_test
./maxweight_test
load_food_database still works: passed, score 2/2
filter_food_vector: passed, score 2/2
dynamic_max_weight trivial cases: passed, score 2/2
dynamic_max_weight correctness: passed, score 4/4
exhaustive_max_weight trivial cases: passed, score 2/2
exhaustive_max_weight correctness: passed, score 4/4
TOTAL SCORE = 16 / 16
```

## Analysis of Dynamic Programming

The dynamic_max_weight function takes a list of food items, each with a weight, calorie count, and calorie limit. It creates a table where each cell represents the maximum weight that can be achieved with a certain number of food items without exceeding the calorie limit.
The function fills this table by comparing each food item's calorie count to each possible calorie limit. If the food item's calorie count is less than or equal to the calorie limit, the function updates the cell with the maximum value between the current cell value and the sum of the current food item's weight and the value of the cell that represents the remaining calorie limit after including the current food item. If the food item's calorie count is greater than the calorie limit, the function just copies the value from the cell above.

After filling the table, the function constructs the optimal food selection. Starting from the bottom right corner of the table, it moves up, adding any food item that was included in the optimal selection to a list and subtracting its calorie count from the remaining calorie limit. This process continues until it reaches the top row or runs out of calories. The function then returns the list of selected food items.

## Dynamic Max Weight Pseudocode



```
def dynamic_max_weight(foodItems, totalCalorieLimit):
    # Copy the food_items vector
    foodSource = copy(foodItems)
    optimalFoodSelection = []

    # Initialize the dynamic programming table
    foodCount = len(foodItems)
    dpTable = [[0] * (totalCalorieLimit + 1) for _ in range(foodCount + 1)]

    # Fill the dynamic programming table
    for index in range(1, foodCount + 1):
        for calorie in range(totalCalorieLimit + 1):
            if foodSource[index - 1].calorie <= calorie:
                dpTable[index][calorie] = max(dpTable[index - 1][calorie], dpTable[index - 1][calorie - foodSource[index - 1].calorie] + foodSource[index - 1].weight)
            else:
                dpTable[index][calorie] = dpTable[index - 1][calorie]

    # Creating the perfect menu
    index = foodCount
    remainingCal = totalCalorieLimit
    while index > 0 and remainingCal > 0:
        if dpTable[index][remainingCal] != dpTable[index - 1][remainingCal]:
            optimalFoodSelection.append(foodSource[index - 1])
            remainingCal -= foodSource[index - 1].calorie
        index -= 1

    return optimalFoodSelection
```

## Big-O for Dynamic Programming Algorithm

```
def dynamic_max_weight(foodItems, totalCalorieLimit):
   # Copy the food_items vector
   foodSource = copy(foodItems) // 1
   optimalFoodSelection = [] // 1

   # Initialize the dynamic programming table
   foodCount = len(foodItems) // 1
   dpTable = [[0] * (totalCalorieLimit + 1) for _ in range(foodCount + 1)] // 2

   # Fill the dynamic programming table
   for index in range(1, foodCount + 1):  n+1
     for calorie in range(totalCalorieLimit + 1):  n+1
       if foodSource[index - 1].calorie <= calorie:  2
         dpTable[index][calorie] = max(dpTable[index - 1][calorie], dpTable[index - 1][calorie -
foodSource[index - 1].calorie] + foodSource[index - 1].weight) // 2
       else:                                                    max(4,1)
         dpTable[index][calorie] = dpTable[index - 1][calorie] // 1

   # Creating the perfect menu
   index = foodCount // 1
   remainingCal = totalCalorieLimit // 1
   while index > 0 and remainingCal > 0: // 2
     if dpTable[index][remainingCal] != dpTable[index - 1][remainingCal]: // 1
       optimalFoodSelection.append(foodSource[index - 1])
       remainingCal -= foodSource[index - 1].calorie // 2
     index -= 1 // 2

   return optimalFoodSelection // 0
```

$$5 + \sum_{i=0}^{n+1} + \sum_{j=0}^{n+1} + \max(4,1) + 2 + k + 1 + \max(5,0)$$

$$5 + \sum_{i=0}^{n+1} + \sum_{j=0}^{n+1} + 4 + 2 + k + 1 + 5$$

$$10 + \sum_{i=0}^{n+1} + \sum_{j=0}^{n+1} + 6 + k + 1$$

$$16 + \sum_{i=0}^{n+1} + \sum_{j=0}^{n+1} + k + 1$$

$$17 + 4k + 4kn + kn^2$$

$$O(n^2 . k)$$

Time Complexity of this algorithm is $O(n^2)$

- Big-O for Dynamic Programming Algorithm

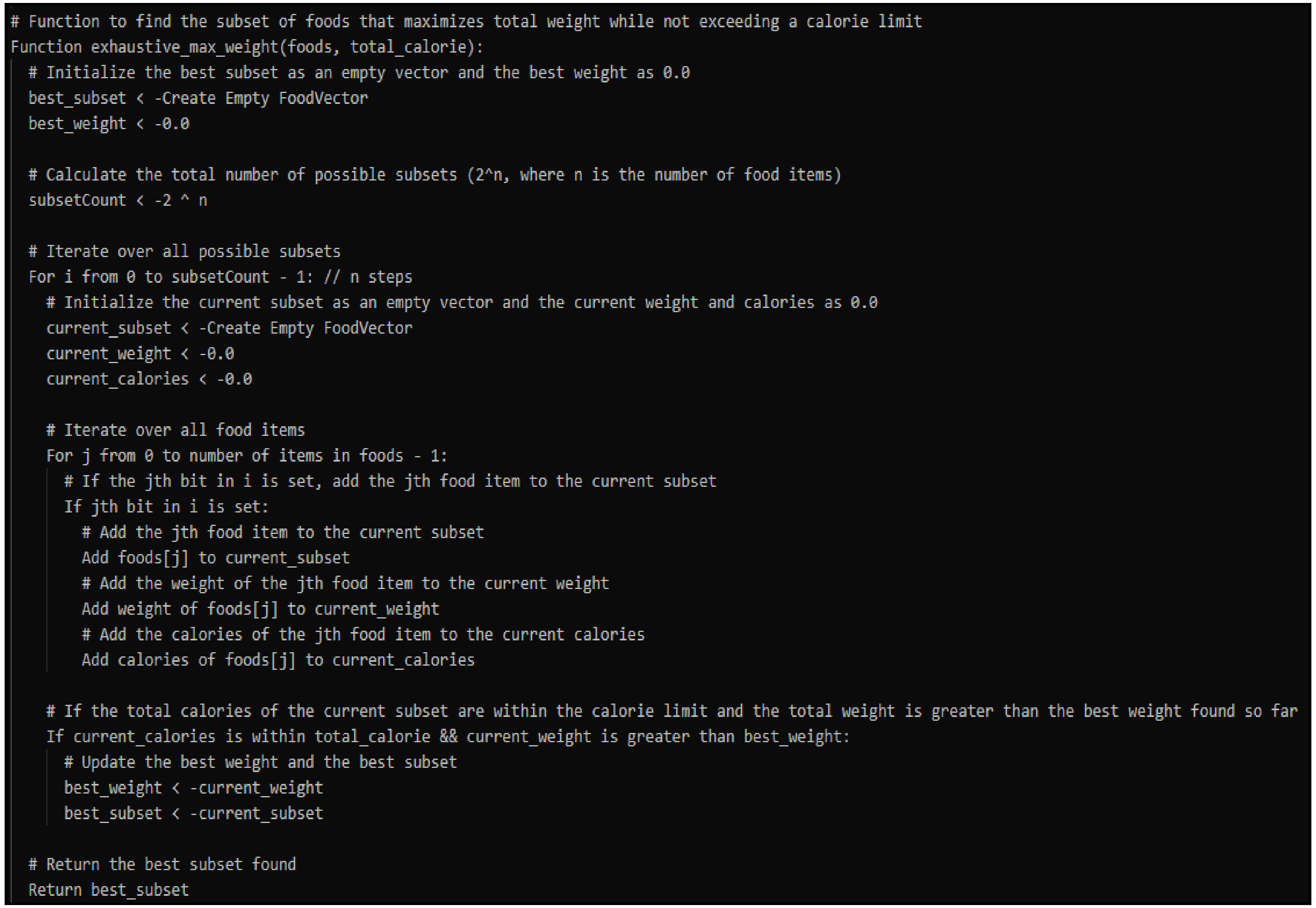
## Analysis of Exhaustive Search

The following exhaustive_max_weight C++ function takes a vector of foods and an overall calorie limit. It is to find which combination of foods yields the greatest overall weight under the limit of calories. It works in several steps. The process begins by initializing the vector best subset to an empty vector while the scalar best weight to zero; these two variables store the best-found combination of food items and the total weight, respectively. Next, the subset generation function generates all possible subsets of the food items. It iterates from 0 to 2^n-1, where n is the number of food items. The variable res is the subset of that iteration. The value of the food's total weight and calories is calculated for each subset. If the particular food was taken, the corresponding bit in the subset mask is set, and the food's weight and calories are added.

The function then checks if the total calories of that current subset are less than the calorie limit and if the total weight of that subset is greater than the best weight found, in which case it goes ahead to update best weight and best subset. The function ultimately returns a unique pointer to the best subset of food items found.

This is implemented as a naive search algorithm, which means it considers all the possible combinations of food items. Nevertheless, due to the time complexity of this algorithm being on the order of O(2n·n), it might get slow in extreme cases of large inputs. Notice that the function gives an optimal solution, although it's computationally inefficient in terms of time.

## Exhaustive Search Pseudocode



```
# Function to find the subset of foods that maximizes total weight while not exceeding a calorie limit
Function exhaustive_max_weight(foods, total_calorie):
  # Initialize the best subset as an empty vector and the best weight as 0.0
  best_subset < -Create Empty FoodVector
  best_weight < -0.0

  # Calculate the total number of possible subsets (2^n, where n is the number of food items)
  subsetCount < -2 ^ n

  # Iterate over all possible subsets
  For i from 0 to subsetCount - 1: // n steps
    # Initialize the current subset as an empty vector and the current weight and calories as 0.0
    current_subset < -Create Empty FoodVector
    current_weight < -0.0
    current_calories < -0.0

    # Iterate over all food items
    For j from 0 to number of items in foods - 1:
      # If the jth bit in i is set, add the jth food item to the current subset
      If jth bit in i is set:
        # Add the jth food item to the current subset
        Add foods[j] to current_subset
        # Add the weight of the jth food item to the current weight
        Add weight of foods[j] to current_weight
        # Add the calories of the jth food item to the current calories
        Add calories of foods[j] to current_calories

    # If the total calories of the current subset are within the calorie limit and the total weight is greater than the best weight found so far
    If current_calories is within total_calorie && current_weight is greater than best_weight:
      # Update the best weight and the best subset
      best_weight < -current_weight
      best_subset < -current_subset

  # Return the best subset found
  Return best_subset
```

## Big-O for Exhaustive Search

```
Function exhaustive_max_weight(foods, total_calorie):
 best_subset <- Create Empty FoodVector  // 1
 best_weight <- 0.0  // 1
 subsetCount <- 2 ^ n  // 1

 For i from 0 to subsetCount - 1:  // n+1
  current_subset <- Create Empty FoodVector  // 1
  current_weight <- 0.0  // 1
  current_calories <- 0.0  // 1

  # Iterate over all food items
  For j from 0 to number of items in foods - 1:  n+1
   If jth bit in i is set:  max( 3, 0 )
    Add foods[j] to current_subset  // 1
    Add weight of foods[j] to current_weight  // 1
    Add calories of foods[j] to current_calories  // 1

  # If the total calories of the current subset are within the calorie limit and the total weight
is greater than the best weight found so far
  If current_calories is within total_calorie && current_weight is greater than best_weight: max(2,0)
   best_weight <- current_weight  // 1
   best_subset <- current_subset  // 1

 Return best_subset  // 0
```

$$3 + \sum_{i=0}^{n+1} 3 + \sum_{j=0}^{n+1} 3 + \max(2,0)$$

$$3 + \sum_{i=0}^{n+1} 3 + \sum_{j=0}^{n+1} 3 + 2$$

$$6 + \sum_{i=0}^{n+1} 3 + \sum_{j=0}^{n+1} 3$$

$$6 + 3(n+2) + 3(n+2)$$

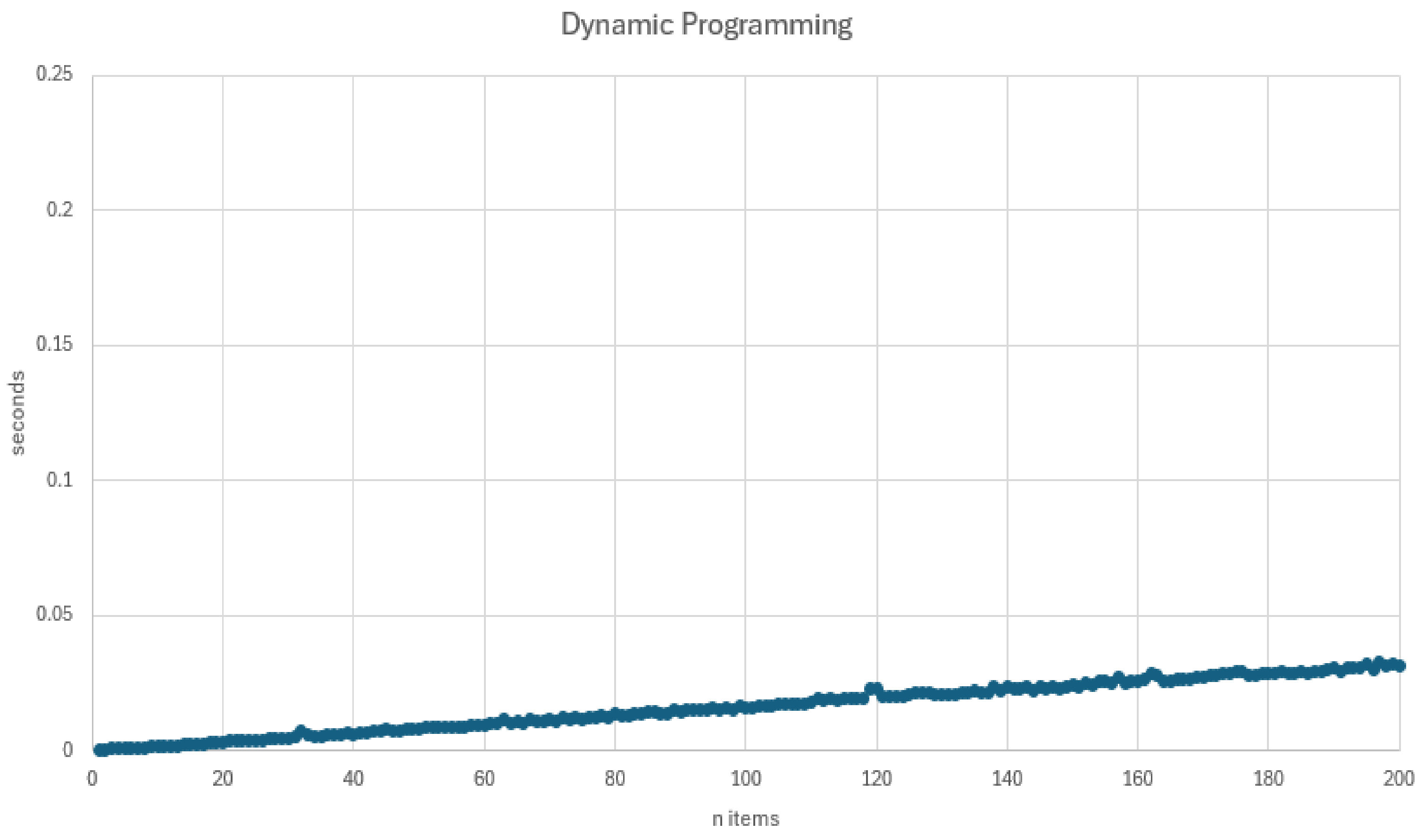
$$6 + 3n + 6 + 3n + 6$$

$$18 + 6n$$

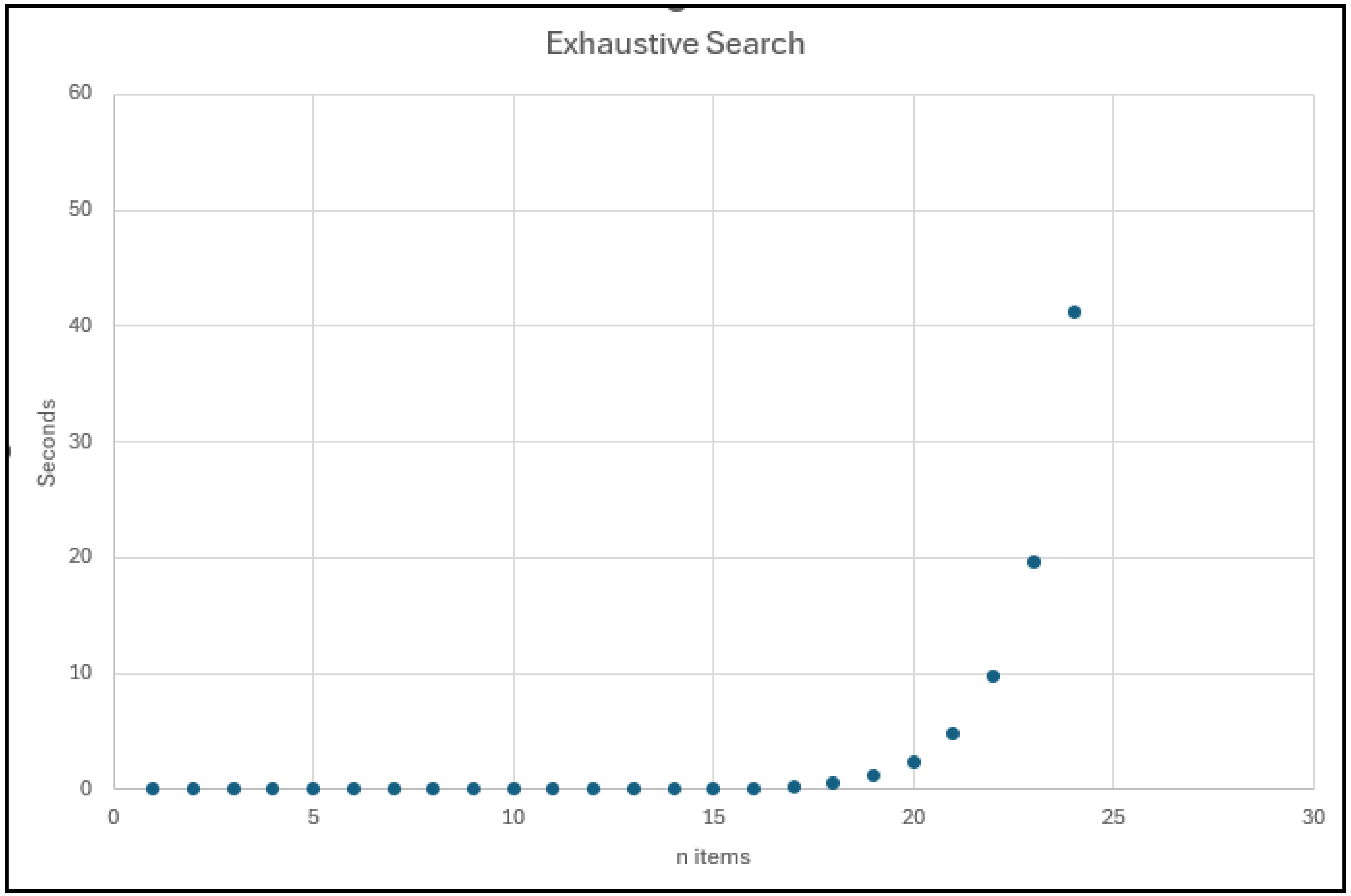
Outerloop runs at $2^n$

Time Complexity of this algorithm $O(2^n \cdot n)$

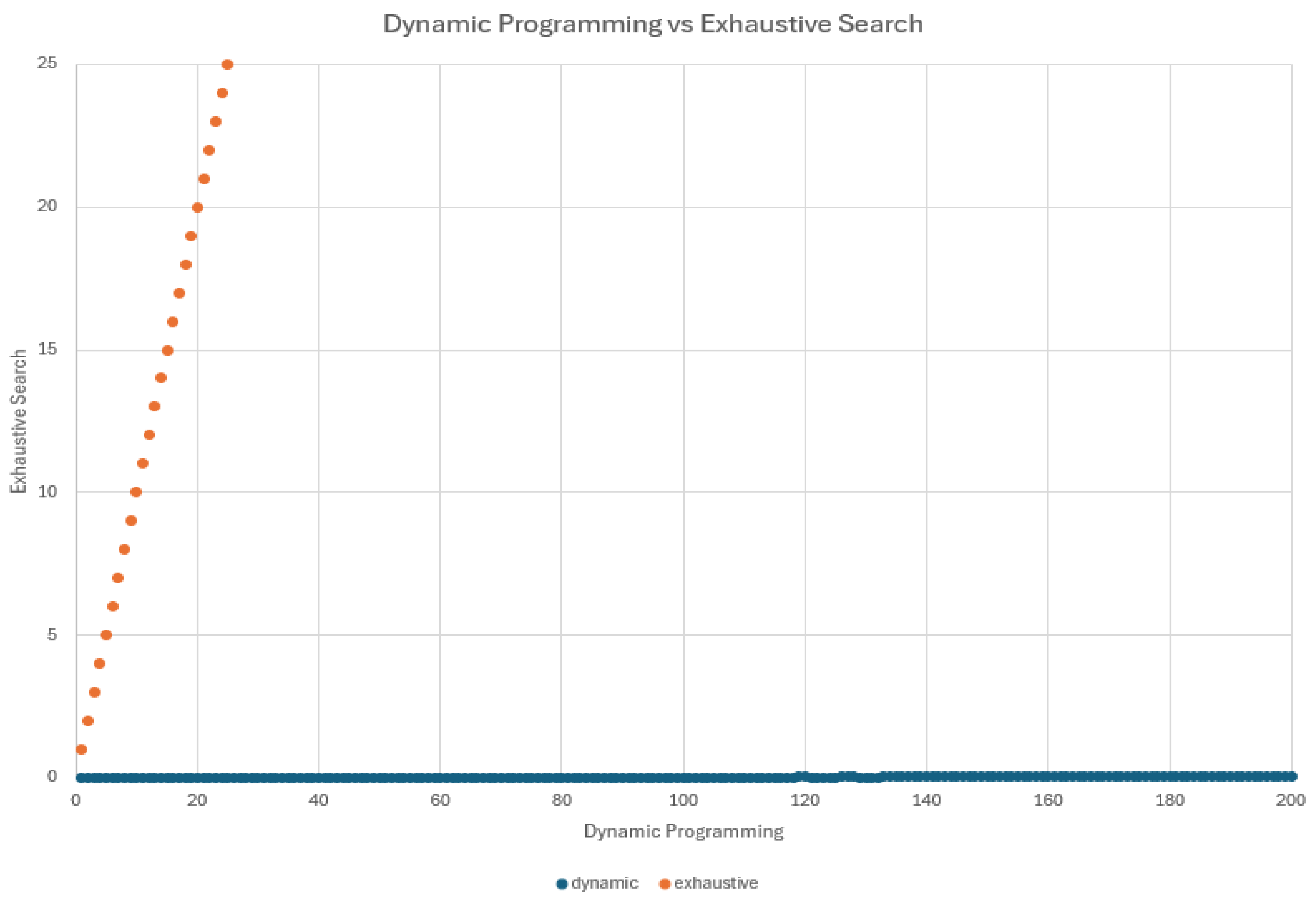
- **Big-O for Exhaustive Search**


Dynamic Programming
0.25
0.2
0.15
0.1
0.05
0
seconds
0
20
40
60
80
100
120
140
160
180
200
n items

- **Hypothesis 1:** The dynamic programming algorithm will perform faster than the exhaustive search algorithm for all sizes of input n.
- **Answer:** The data supports this hypothesis, demonstrating that the dynamic programming algorithm is much faster than the exhaustive search across all $n$n values. Dynamic programming scales much better because it optimizes the decision-making process by breaking down the problem into manageable subproblems and storing the results to avoid redundant calculations. In contrast, the exhaustive search’s execution time grows exponentially with $n$ as it considers all possible combinations of items, leading to a dramatic increase in computation time as $n$ increases.


Exhaustive Search
Seconds
n items
0
10
20
30
40
50
60
0
5
10
15
20
25
30

- **Hypothesis 2:** The exhaustive search algorithm will demonstrate a polynomial time complexity as n increases.
- **Answer:**The data refutes Hypothesis 2, indicating the exhaustive search likely has exponential time complexity. The execution time increases more than a polynomial rate, suggesting it doubles with each item, typical of O(2^n) complexity from evaluating all combinations, unlike a polynomial O(n^k).


Dynamic Programming vs Exhaustive Search
Exhaustive Search
Dynamic Programming
0
5
10
15
20
25
0
20
40
60
80
100
120
140
160
180
200
● dynamic
● exhaustive