GREEDY VS EXHAUSTIVE SEARCH

Group member:

- Erik Williams
- epwilliams@csu.fullerton.edu

```
Erik_Williams ~/project-2-spring-camp-food-erik-williams$ 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
greedy_max_weight trivial cases: passed, score 2/2
greedy_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 Greedy Algorithm

The first algorithm uses the greedy pattern. The greedy heuristic is always to choose the "best" food item that fits within the limit and keep selecting the following best food items until you do not have enough calories to add more food items. The time complexity of the greedy algorithm depends on the data structures used to implement it. A naive approach using unsorted vectors and sequential search to find "a" takes $O(n^2)$time. This is acceptable but not ideal.

GREEDY ALGORITHM PSEUDOCODE

```
Function greedy_max_weight(total_calorie, food_items):
    selected_items <- Empty List         // 1 step
    current_calories <- 0                // 1 step

    While food_items is not empty && current_calories <= total_calorie: // Average O(n/2) steps
        best_item <- null                 // 1 step
        best_ratio <- 0                  // 1 step

        For each item in food_items:      // n steps
            item_ratio <- item.weight / item.calorie // 2 steps
            If item_ratio > best_ratio && current_calories + item.calorie <= total_calorie: // 4 steps
                best_item <- item        // 1 step
                best_ratio <- item_ratio  // 1 step

        If best_item is not null:
            selected_items.add(best_item) // 1 step
            current_calories += best_item.calorie // 2 steps
            food_items.remove(best_item) //  1 step
        Else:
            Break // Exit if no suitable item found 1 step
    Return selected_items
End Function
```

Big O = $1 + 1 + \sum_{n=1}^{n/2} 2 + \sum_{n=1}^{n} 2 + 4 + 1 + 1 + \max(4, 1)$

=> $2 + \sum_{n=1}^{n/2} 2 + \sum_{n=1}^{n} 12$

=> 2 + n + 12n^2

=> 12n^2 + n + 2

=> O(n^2)

Big-O for Greedy Algorithm

Initialization: The creation of an empty list and setting of initial variables is O(1).

While Loop: The loop runs as long as items are left in food_items and the current_calories is less than or equal to total_calorie. The worst-case scenario for this loop to run is n times, where n is the number of items in food_items. However, since items are removed from food_items in each iteration where a suitable item is found, the number of iterations will often be less than n.

For Each Loop Inside While Loop: Inside the while loop, there's a for-each loop that iterates over each item in food_items to find the item with the best weight-per-calorie ratio that fits within the remaining calorie limit. This nested loop means that for each iteration of the while loop, up to n comparisons might be made. In the worst case, O(n^2)comparisons might be necessary.

Selection and Removal of Best Item: If a suitable "best item" is found, it's added to selected_items, and its calories are added to current_calories. The item is then removed from food_items. The addition operation is O(1) but removing an item from a list could be O(n) in the worst case since it may require shifting elements.

Worst-case time complexity of this algorithm can be approximated as O(n^2). This quadratic time arises because, in each iteration of the while loop, the algorithm examines each item in the list to find the best item, and removing an item from a list can be O(n) time in the list.

Analysis of Exhaustive Search

The second uses an exhaustive search. For the implementation to function correctly, the variable 'bits' used in the loop must be capable of representing the value $2^n - 1$. To achieve this, the most suitable approach is to utilize the largest standard integer data type available in C++, which is 'uint64_t'. This data type provides 64 bits of storage. Theory suggests that the exhaustive search algorithm, due to its $O(2^n \cdot n)$ complexity, will be significantly slower than the greedy algorithm.

EXHAUSTIVE SEARCH PSEUDOCODE

```
Function exhaustive_max_weight(foods, total_calorie):
best_subset <- Create Empty FoodVector // 1 step
best_weight <- 0.0 // 1 step
subsetCount <- 2 raised to the power n

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

    For j from 0 to number of items in foods - 1: // n steps
        If jth bit in i is set:
            Add foods[j] to current_subset // 1 step
            Add weight of foods[j] to current_weight // 1 step
            Add calories of foods[j] to current_calories // 1 step

    If current_calories is within total_calorie && current_weight is greater than best_weight:
        best_weight <- current_weight // 1 step
        best_subset <- current_subset// 1 step

Return best_subset
```

Big O = $1 + 1 + 2^n + \sum_{n=1}^{n} 3 + \sum_{n=1}^{n} 3 + 1 + 1$

=> 1 + 1 + 2^n + $\sum_{n=1}^{n} 3$ + $\sum_{n=1}^{n} 3 + 2$

=> 1 + 1 + 2^n + 3n + 5n

=> 8n + 2^n + 2

=> O(2^n*n)

Big-O for Exhaustive Search

Initialization: Creating a new empty FoodVector (best_subset) and setting best_weight to 0.0 are constant time operations, O(1).

Subset Count Calculation: The total number of possible subsets, which is 2^n where n is the number of food items, can be done in O(n) time because it involves left-shifting 1 n times.

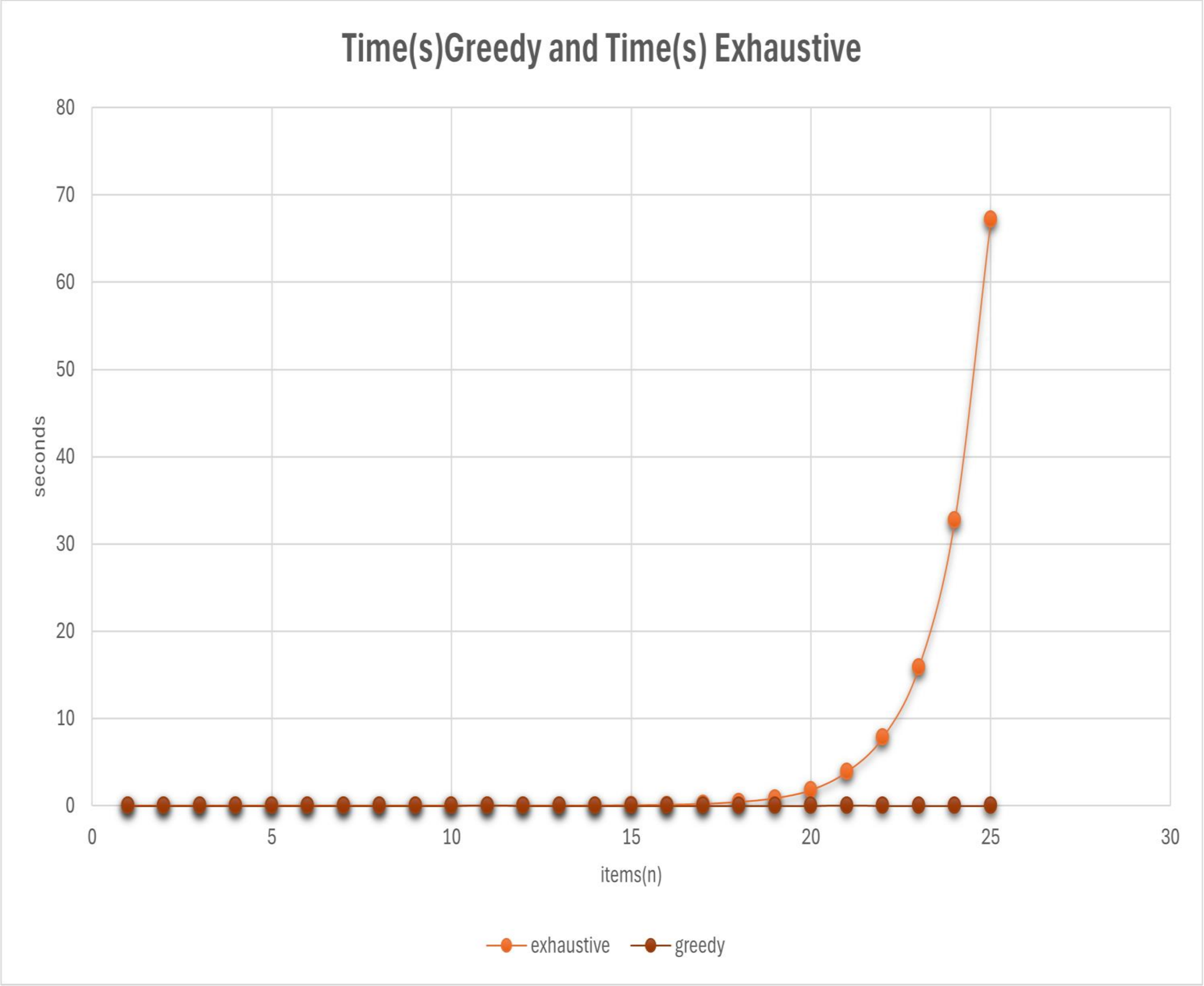
Subset Loop: The loop runs for each possible subset, resulting in 2^n iterations. Each iteration creates a new empty vector and initializes variables to track the weight and calories of the current subset, which are constant time operations, O(1). However, this loop's total running time is O(2^n) due to the exponential number of subsets.

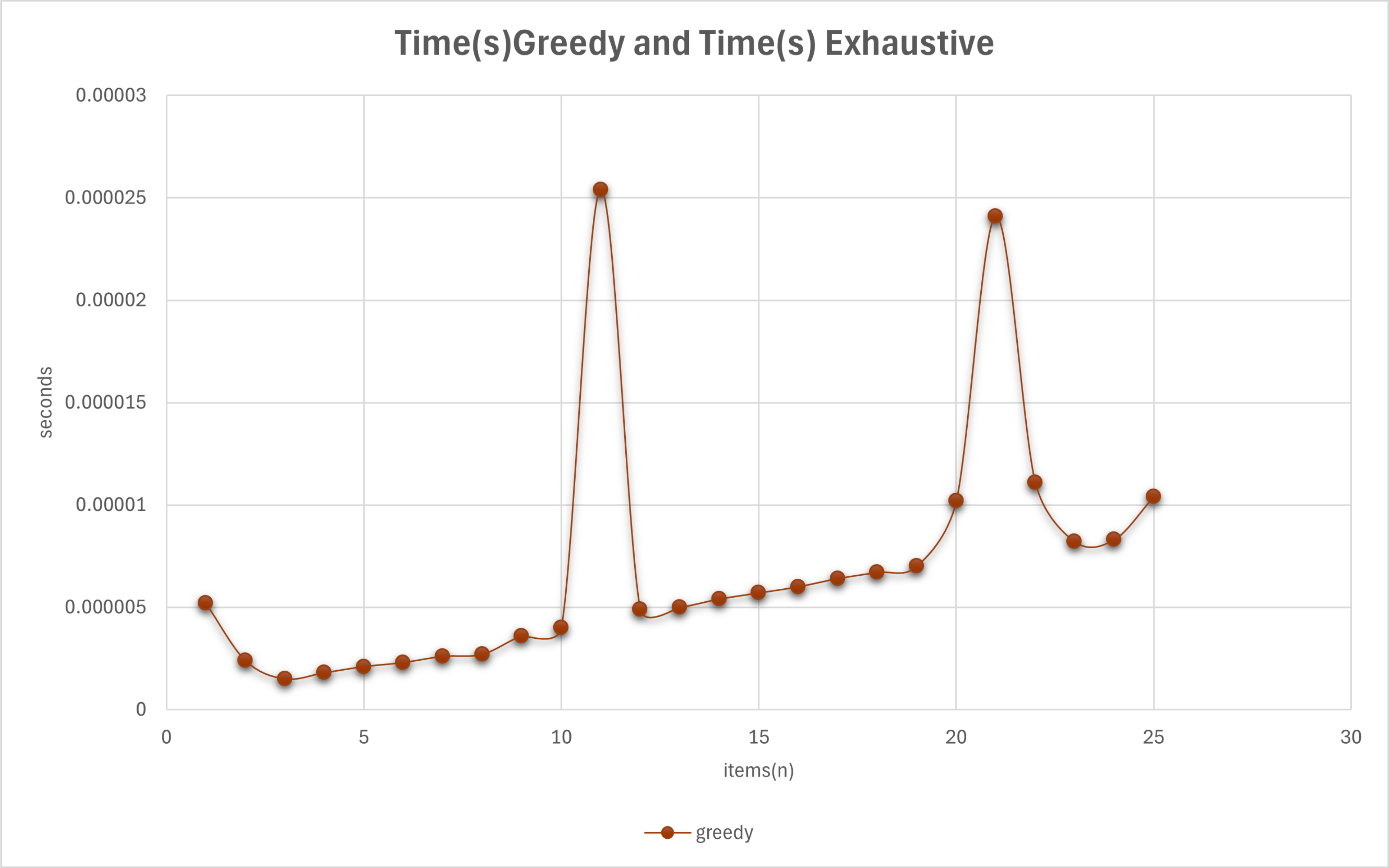
Inner Loop for Subset Construction: There is a loop over n items for each subset to construct the subset by checking if the j-th bit is set and then adding the corresponding food item to the current subset. Each check and addition is O(1), but over n items and 2^n subsets, this results in O(n*2^n) time.

Comparison and Update: The comparison is done to check if the current subset's calories are within the total calorie limit and if its weight is greater than the best_weight, which is O(1). Updating the best_weight and reassigning the best_subset are also O(1) operations.

Return: Returning the best subset is a constant time operation, O(1).
Overall, the worst-case time complexity of the exhaustive search algorithm can be approximated as O(n*2^n), which considers both the number of subsets (2^n) and the processing for each subset (O(n)). The most time-consuming part is iterating over all possible subsets and constructing each one to evaluate its total weight and calorie content.

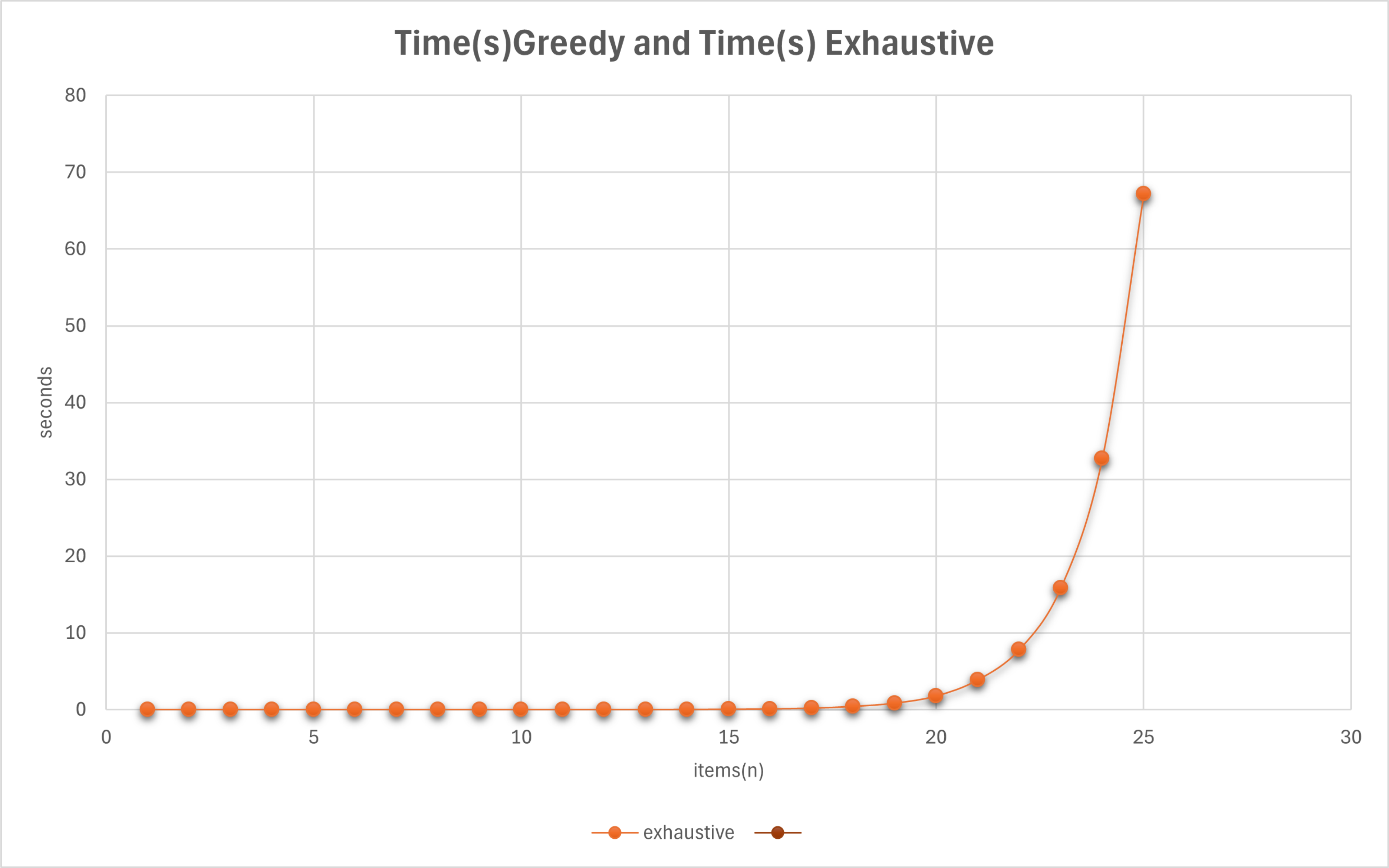
- **X-axis (items(n))**: the size of the input, n. It shows how the execution time changes as the number of items increases.
- **Y-axis (seconds)**: Represents the time in seconds it took for each algorithm to complete. It's the metric used to compare the performance of the algorithms.
- The scatterplot compares the execution times of greedy and exhaustive search algorithms.
- Both algorithms perform similarly for small input sizes.
- The greedy algorithm maintains low and stable execution times even as the input size increases.
- The exhaustive search algorithm's execution time remains low initially but dramatically increases after 20 items.
- The steep rise in execution time for the exhaustive search indicates a much higher time complexity, likely due to its combinatorial nature.
- The graph demonstrates that the greedy algorithm is more scalable for larger input sizes than the exhaustive search.





Time(s)Greedy and Time(s) Exhaustive
seconds
0.00003
0.000025
0.00002
0.000015
0.00001
0.000005
0
0
5
10
15
20
25
30
items(n)
greedy

- **Hypothesis 1:** The greedy algorithm will perform faster than the exhaustive search algorithm for all sizes of input n.
- **Answer:**The data supports Hypothesis 1, showing the greedy algorithm is much faster than the exhaustive search across all n values. The greedy algorithm scales linearly or logarithmically, while the exhaustive search's time grows exponentially with n. This is expected, as the greedy algorithm makes a single pass, choosing the best option at each step, whereas the exhaustive search considers all item combinations, significantly increasing with n.


Time(s)Greedy and Time(s) Exhaustive
seconds
items(n)
80
70
60
50
40
30
20
10
0
0
5
10
15
20
25
30
exhaustive

- **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).

In conclusion, the greedy algorithm demonstrates superior performance over the exhaustive search, maintaining relatively stable execution times even as the number of items, n, increases. In contrast, the exhaustive search's execution time escalates exponentially with larger n values. The widening performance gap affirms the greedy algorithm's effectiveness and efficiency, attributable to its design principles prioritizing speed, unlike the exhaustive search that sacrifices speed for thoroughness.