Benjamin Yang
eternadj4@csu.fullerton.edu
Gurman Gill
gurman662@csu.fullerton.edu

Project 2

Greedy Algorithm Running Time Analysis

- Copying the foods vector: Creating a copy of the foods vector takes O(n) time, where n is the size of the vector.
- Sorting the sortedFoods vector: Sorting the vector using std::sort has a time complexity of O(nlgn), where n is the size of the vector.
- Building the result vector: Iterating over the sortedFoods vector and adding the food items to the result vector takes O(n) time since it visits each element once.

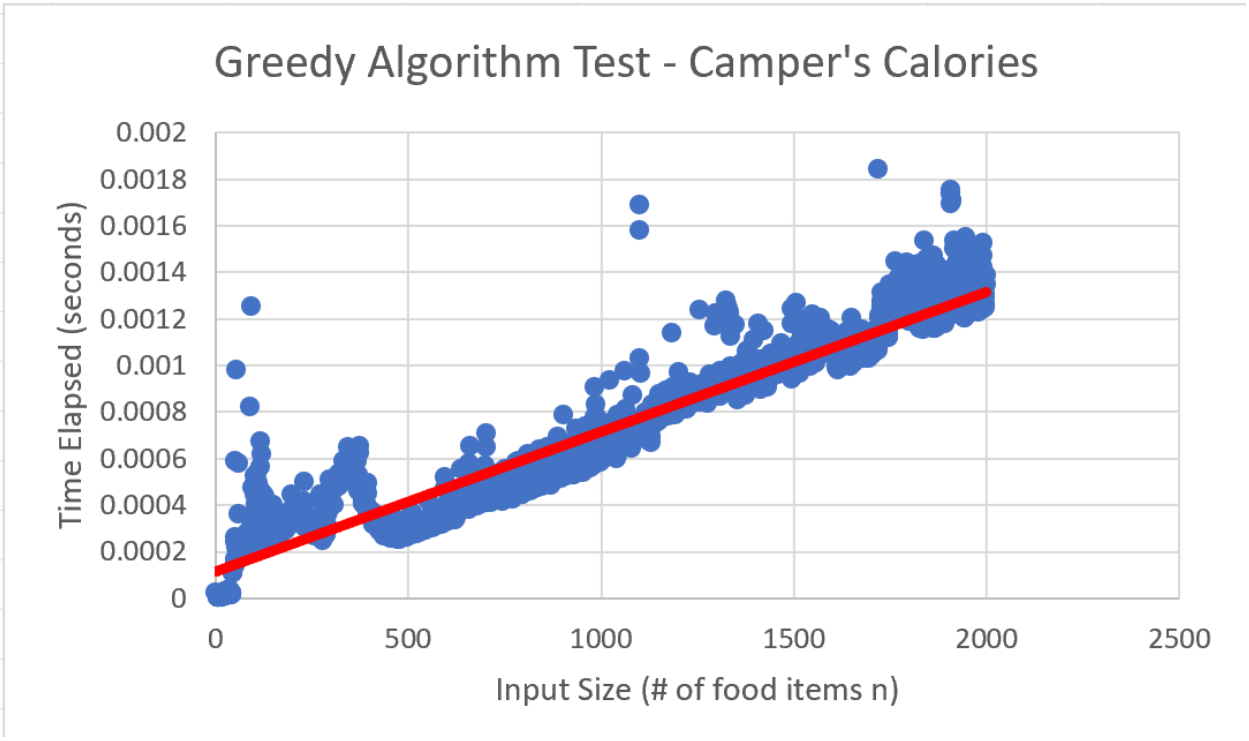
Therefore, the overall time complexity of the greedy_max_weight function is dominated by the sorting step, resulting in O(nlgn) complexity, where n is the size of the input vector foods.

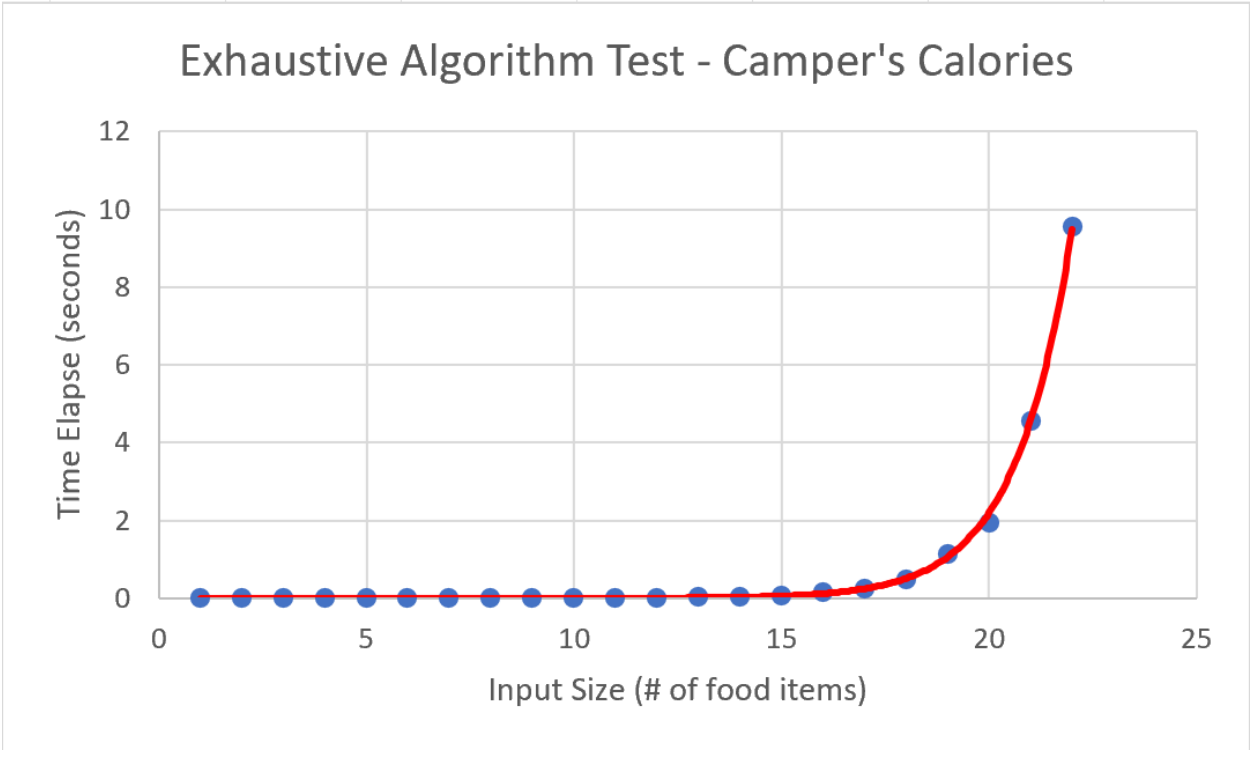
Exhaustive Algorithm Running Time Analysis

- Assert & Initialization: Assert statement and initializing the variables best and bestWeight takes constant time.
- Subset Generation: The function uses a loop to generate all possible subsets of the foods vector. The loop iterates from 0 to 2^n - 1, where n is the size of the foods vector. Generating all possible subsets using bitwise operations has a time complexity of O(2^n).
- Building candidate subsets: For each subset, the function creates a new candidate vector and computes its calorie and weight by iterating over the foods vector. Since each food item is considered once for each subset, this step has a time complexity of O(n * 2^n).
- Updating the best subset: The function checks if the candidate subset satisfies the calorie constraint and has a higher weight than the current best subset. This step takes constant time.

Considering the above steps, the overall time complexity of the exhaustive_max_weight function is O(n * 2^n), where n is the size of the input vector foods. This complexity grows exponentially with the number of food items, making the function computationally expensive for large inputs.

Scatterplots





Conclusion

The greedy algorithm operates in O(n log n) time complexity where n is the size of the input. This is because the most time-consuming operation in the algorithm is sorting the items based on their weight-to-calorie ratio. The actual selection of items is a linear operation which is typically faster than the sorting operation. On the other hand, the exhaustive search operates in O(2^n * n) time complexity which is exponential.

The difference was empirically quite surprising; even with over 2000 items, the greedy algorithm was mere milliseconds while the exhaustive was already over 1 second with 19 items. The greedy algorithm is faster, especially for larger inputs. The empirical analyses are consistent with the mathematical analyses. Both trend lines generally fit the mathematical counterparts. Greedy was like the graph of O(nlgn), while exhaustive was clearly exponential.

The evidence is consistent with hypothesis 1. Exhaustive search algorithms are indeed feasible to implement, and they produce correct outputs. However, their feasibility strongly depends on the size of the input. For small inputs, an exhaustive search is quite manageable. But as the size of the input grows, the exhaustive search quickly becomes infeasible due to its exponential time complexity.

The evidence is consistent with hypothesis 2. Algorithms with exponential running times, like the exhaustive search algorithm, are extremely slow for large inputs. This can make them impractical for real-world use where the input size can be large. The speed of such algorithms can be improved using various techniques such as dynamic programming or greedy algorithms, but those come with their own trade-offs.

Pseudocode for greedy:

```
function greedy_max_weight(foods, total_calorie):
    sortedFoods = copy of foods vector, sorted in descending order based on calorie ratio // nlogn

    result = new empty FoodVector // 1
    result_calories = 0.0 // 1

    for each food in sortedFoods: // n
        if (result_calories + food.calorie() <= total_calorie): // 1
            add food to result // 1
            result_calories += food.calorie() // 1
    return result
```

Time complexity: O(n log n)

f(n)= 5n+ n logn+ 8 and g(n)= n log n

$$\lim_{n \to \infty} \frac{5n+n\log n+8}{n\log n} \overset{\text{L'hopital}}{=} \lim_{n \to \infty} \frac{0}{0}$$

$$1 + \lim_{n \to \infty} \frac{5}{\ln n} + \lim_{n \to \infty} \frac{8}{n \ln n} = 1$$

Since here the output is greater than 0 and a constant, 5n+nlogn+8 belongs to O(nlogn)

Pseudocode for exhaustive:

```
function exhaustive_max_weight(foods, total_calorie):
   assert (foods.size() < 64)  // prevent overflow  // 1
                                                                  ] 3
   best = null   // 1
   bestWeight = 0  // 1

   n = foods.size()  // 1
   for bits = 0 to (2^n - 1):  // 2^n
      candidate = new empty FoodVector  // 1
      candidateCalorie = 0  // 1                                  ] 3(2^n)
      candidateWeight = 0  // 1

      for j = 0 to (n - 1):  // n
         if ((bits >> j) & 1) == 1:  // if the j-th bit is set  // 1
            add foods[j] to candidate  // 1                     ] n
            candidateCalorie += foods[j].calorie()  // 1
            candidateWeight += foods[j].weight()  // 1

      if candidateCalorie <= total_calorie and (best == null or candidateWeight > bestWeight):  // 1
         best = candidate  // 1
         bestWeight = candidateWeight // 1
   return best
```

Time Complexity: $O(2^n)$

```cpp
// Compute the optimal set of food items with a greedy algorithm.
// Specifically, among the food items that fit within a total_calorie,
// choose the foods whose weight-per-calorie is greatest.
// Repeat until no more food items can be chosen, either because we've
// run out of food items, or run out of space.
std::unique_ptr<FoodVector> greedy_max_weight
(
    const FoodVector& foods,
    double total_calorie
)
{
    // TODO: implement this function, then delete the return statement below
    // make a copy of foods vector to be sorted
    FoodVector sortedFoods = foods;  // 1

    // sort foods descending based on calorie ratio
    std::sort(sortedFoods.begin(), sortedFoods.end(), CompareFoodItems()); // nlogn

    std::unique_ptr<FoodVector> result(new FoodVector); // 1
    double result_calories = 0.0; // 1

    // iterate over sorted foods and add while foods in list and less than total
cals
    for (auto& food : sortedFoods) {  // n+1
        if ((result_calories + food->calorie()) <= total_calorie) { // 2 ] 5n+5
            result->push_back(food);  // 1
            result_calories += food->calorie(); // 2
        }
    }

    return result;
}
```

Step Count: 8+ 5n+nlogn

```cpp
// Compute the optimal set of food items with a exhaustive search algorithm.
// Specifically, among all subsets of food items, return the subset
// whose weight in ounces fits within the total_weight one can carry and
// whose total calories is greatest.
// To avoid overflow, the size of the food items vector must be less than 64.
std::unique_ptr<FoodVector> exhaustive_max_weight
(
    const FoodVector& foods,
    double total_calorie
)
{
    // TODO: implement this function, then delete the return statement below
    // prevent overflow
    assert(foods.size() < 64); // 1

    std::unique_ptr<FoodVector> best = nullptr;  //1
    double bestWeight = 0; //1

    // loop over all subsets
    uint64_t n = foods.size(); //1
    for (uint64_t bits = 0; bits < (1ull << n); ++bits) {// 2^n+1
        std::unique_ptr<FoodVector> candidate(new FoodVector);//1
        double candidateCalorie = 0;  //1
        double candidateWeight = 0;   //1

        for (uint64_t j = 0; j < n; ++j) { //n+1
            // if the j-th bit is set
            if (((bits >> j) & 1) == 1) {  //3
                candidate->push_back(foods[j]); //1
                candidateCalorie += foods[j]->calorie(); //1
                candidateWeight += foods[j]->weight();  //1
            }
        }

        // update best if this subset is better
        if (candidateCalorie <= total_calorie &&
            (best == nullptr || candidateWeight > bestWeight)) { //5
            best = std::move(candidate);  //1
            bestWeight = candidateWeight; //1
        }
    }

    return best;
}
```

3(2^n)+3

6n+6

7

Step Count: $18n + 18 \cdot (2^n) + 39(2^n) + 43$

Readme

README.md > # project-2

```
# project-2
summer camp food

Group members:

Benjamin Yang eternaldj4@csu.fullerton.edu
Gurman Gill gurman662@csu.fullerton.edu

```

Output: