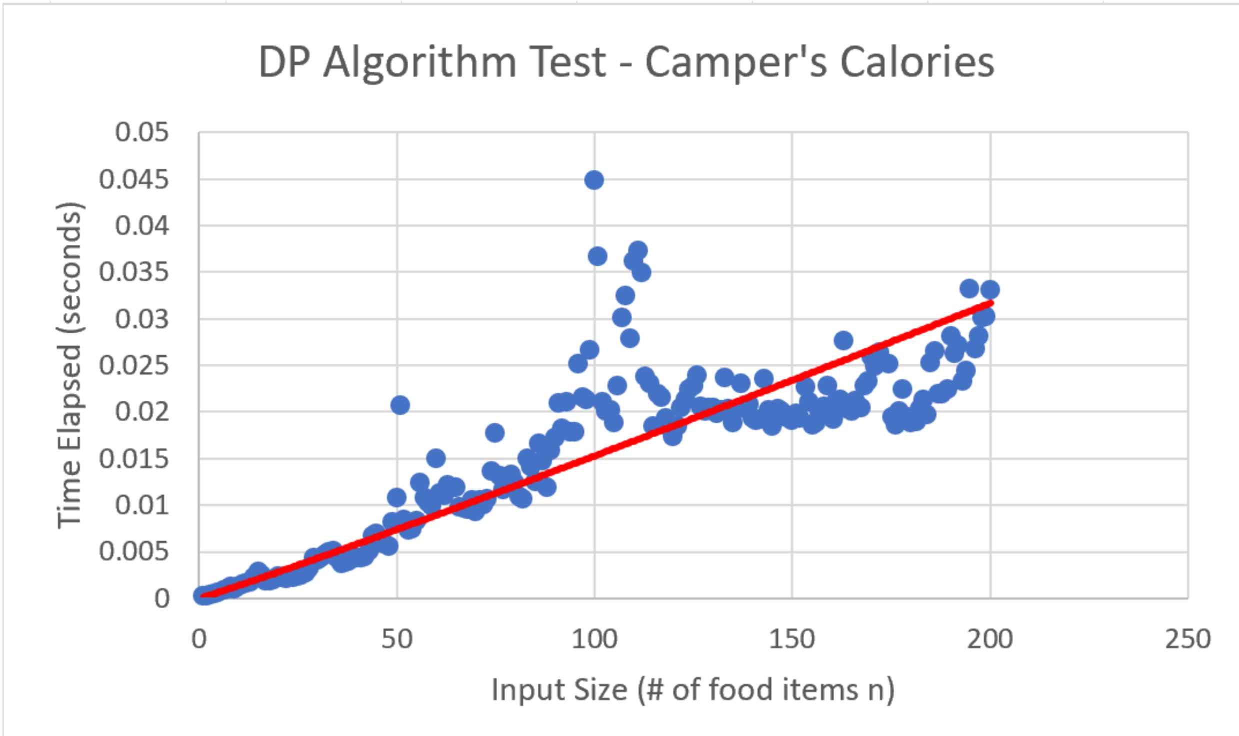
Benjamin Yang
Gurman Gill

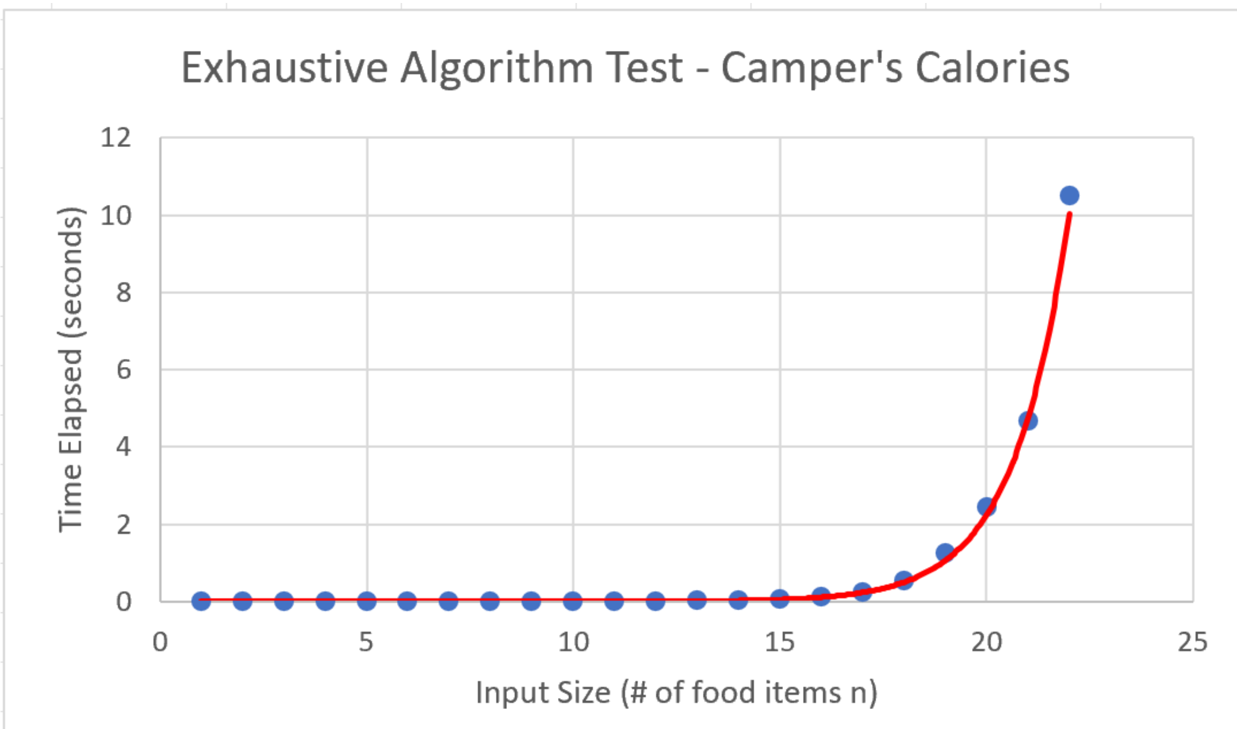
Project 4

Dynamic Programming Algorithm Running Time Analysis
Initialization: Initializing variables n and total_calories_int takes constant time.
DP Table Initialization: Creating a 2D vector dp with size (n + 1) x (total_calories_int + 1) takes O(n * total_calories) time, where n is the size of the foods vector and total_calories is the total allowed calorie value.
Building the DP Table: The function iterates over i from 0 to n and j from 0 to total_calories_int. Each iteration takes constant time, so the overall time complexity of this step is O(n * total_calories).
Backtracking to Find Chosen Foods: The function iterates over i from n to 1 and weight from total_calories_int to 1. Each iteration takes constant time, so the overall time complexity of this step is O(n * total_calories).
Therefore, the overall time complexity of the dynamic_max_weight function is dominated by the time complexity of building the DP table, resulting in O(n * total_calories) complexity, where n is the size of the input vector foods and total_calories is the total allowed calorie value.

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


DP Algorithm Test - Camper's Calories
Time Elapsed (seconds)
0.05
0.045
0.04
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
0
50
100
150
200
250
Input Size (# of food items n)


Exhaustive Algorithm Test - Camper's Calories
Time Elapsed (seconds)
12
10
8
6
4
2
0
0
5
10
15
20
25
Input Size (# of food items n)

Conclusion
The DP algorithm operates in O(n * total_calories) time complexity where n is the size of the input. This is because of the double for loop when building the table. The actual selection of items is a linear operation when backtracking through the table. On the other hand, the exhaustive search operates in O(2^n * n) time complexity which is exponential.

The difference was empirically noticeable; with 200 items, the greedy algorithm was around 0.03 seconds, while the exhaustive was already over 1 second with 19 items. The dynamic algorithm is faster, especially for larger inputs. The empirical analyses are consistent with the mathematical analyses. Both trend lines generally fit the mathematical counterparts. Dynamic was like the graph of O(n) or linear since the total calories were relatively low and consistent, 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 dynamic:

```
dynamic_max_weight(foods, total_calories):        //nlogn
   n = size of foods vector              //n
   total_calories_int = integer value of total_calories

   // DP table initialization
   create dp table with size (n + 1) x (total_calories_int + 1)    //3

   // Build DP table
   for i = 0 to n:              //1
      for j = 0 to total_calories_int:
         if foods[i].calorie > j:          //1
            dp[i + 1][j] = dp[i][j]      //2
         else:
            dp[i + 1][j] = max(dp[i][j], dp[i][j - foods[i].calorie] + foods[i].weight)   //3

   // Backtrack to find chosen foods
   create an empty vector called best

   weight = total_calories_int   //1
   for i = n to 1:                  //n
      if dp[i][weight] != dp[i - 1][weight]:   //1
         add foods[i - 1] to best vector    //1
         weight -= foods[i - 1].calorie   //1

   return best
```

Time complexity: O(nlogn)

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

lim 5n+nlogn+8/ =L'hopital lim
n to infinity nlogn n to infinity

0 0
1 + lim 5/ln n + lim 8/nlnn = 1
n to infinity n to infinity

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

   best = null   //1
   bestWeight = 0     //1

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

```
            candidateCalorie += foods[j].calorie()   //1
                                                             //n
            candidateWeight += foods[j].weight()    //1

        if candidateCalorie <= total_calorie and (best == null or candidateWeight > bestWeight):
            best = candidate    //1
            bestWeight = candidateWeight    //1
    return best
                    Time Complexity: O(2^n)
```

Algorithms:

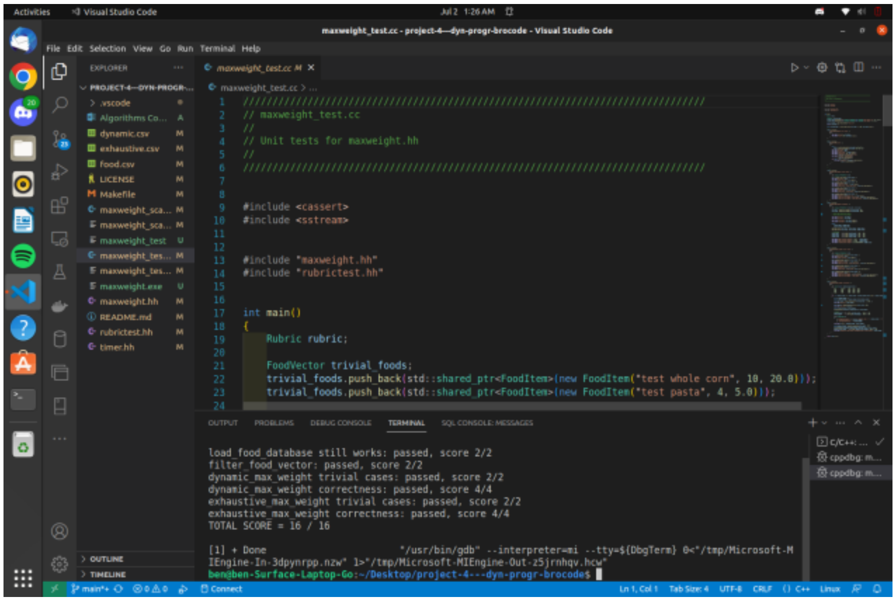
```
// Compute the optimal set of food items with dynamic programming.
// Specifically, among the food items that fit within a total_calories,
// choose the foods whose weight-per-calorie is largest.
// Repeat until no more food items can be chosen, either because we've
// run out of food items, or run out of space.
std::unique_ptr<FoodVector> dynamic_max_weight
(
    const FoodVector& foods,
    double total_calories
)
{
    // std::unique_ptr<FoodVector> source(new FoodVector(foods));
    // std::unique_ptr<FoodVector> best(new FoodVector);
    // print_food_vector(*todo);

    // TODO: implement this function, then delete the return statement below
    int n = foods.size();
    int total_calories_int = static_cast<int>(total_calories);

    // DP table
    std::vector<std::vector<double>> dp(n + 1, std::vector<double>(total_calories_int + 1, 0.0));

    // build DP table
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j <= total_calories_int; ++j) {
            if (foods[i]->calorie() > j) {
                dp[i + 1][j] = dp[i][j];
            } else {
                // choose whichever is higher
                dp[i + 1][j] = std::max(dp[i][j], dp[i][j - static_cast<int>(foods[i]->calorie())] + foods[i]-
>weight());
            }
        }
    }
```

```
    // backtrack to find chosen foods
    std::unique_ptr<FoodVector> best(new FoodVector);
    int weight = total_calories_int;
    for (int i = n; i > 0 && weight > 0; --i) {
        // the food was chosen so add it
        if (dp[i][weight] != dp[i - 1][weight]) {
            best->push_back(foods[i - 1]);
            weight -= static_cast<int>(foods[i - 1]->calorie());
        }
    }

    return best;
}



// 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_calories
(
    const FoodVector& foods,
    double total_calorie
)
{
    // TODO: implement this function, then delete the return statement below
    // prevent overflow
    assert(foods.size() < 64);

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

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

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

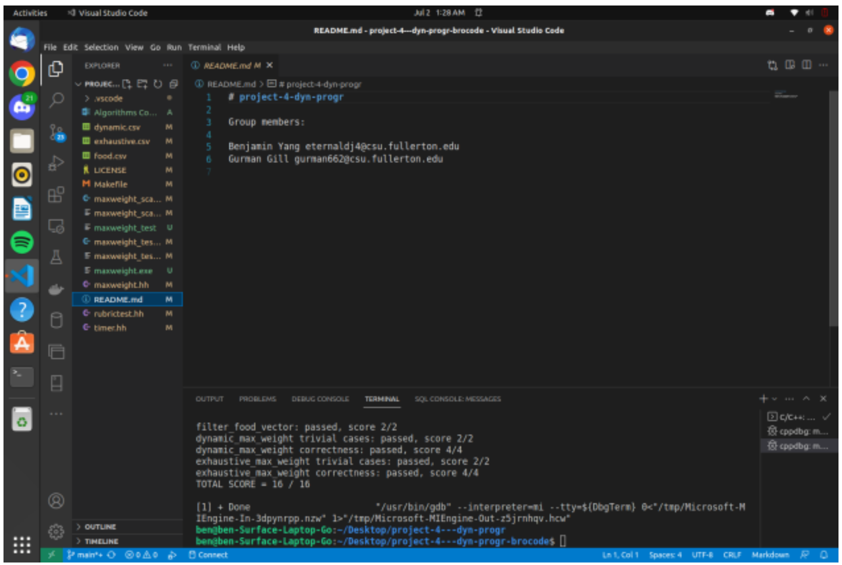
```
            candidateCalorie += foods[j]->calorie();
            candidateWeight += foods[j]->weight();
          }
        }

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

      return best;
    }
```

Step count: 8+5n+nlogn

output



readme:



Algorithms execution: