Cal State Fullerton
College of Engineering and Computer Science

Project 4:

Dynamic Programming versus Exhaustive Algorithm

Student: Holly Thuy Do

Class: CPSC 335-11

CWID: 884916222

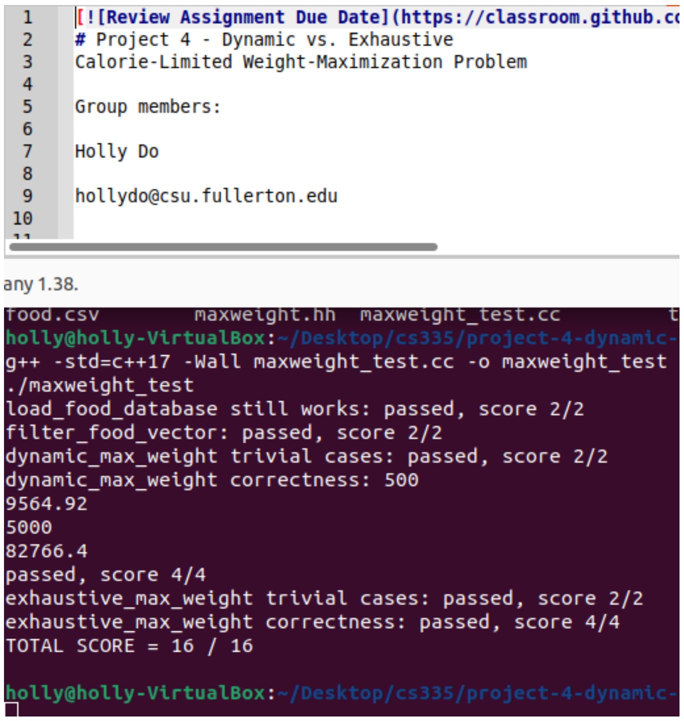
Date: 05/10/2024

Introduction

Calorie-limited weight-maximization problem:

Input: A positive number C representing calories and a vector V of n “food item” objects, containing one or more food items where each item a=(q, c) has a quantity in ounces q>=0 and number of calories c>0

Output: A vector K of food items drawn from V, such that the sum of total calories from K is within the prescribed total calories C and the sum of the items’ weight is maximized. In other words: (q, c)Vq C and the total sum of all items’ weights (q, c) q is maximized.

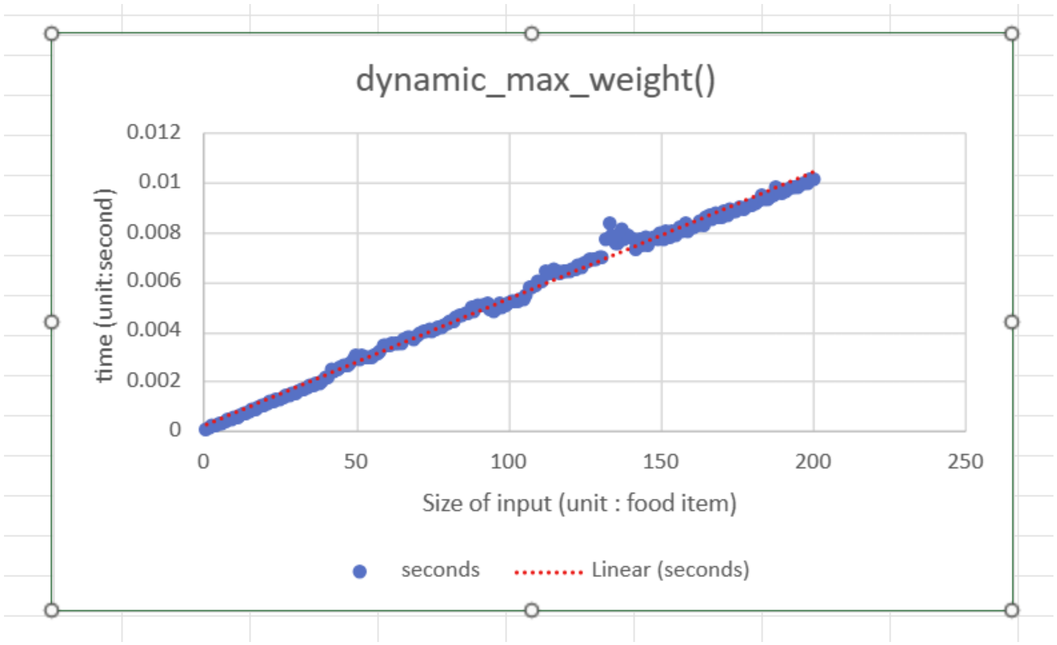


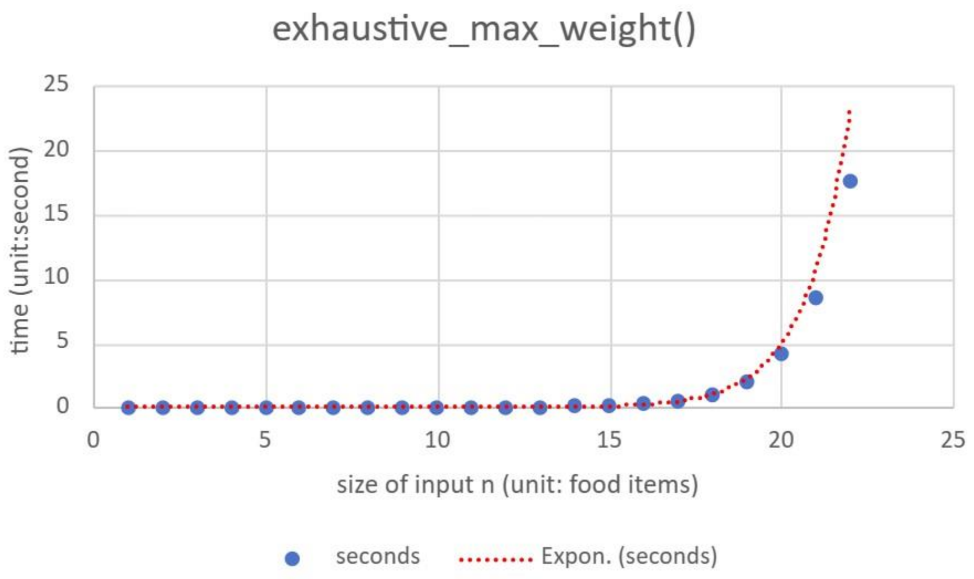
```
[![Review Assignment Due Date](https://classroom.github.co
# Project 4 - Dynamic vs. Exhaustive
Calorie-Limited Weight-Maximization Problem

Group members:

Holly Do

hollydo@csu.fullerton.edu
```

any 1.38.

```
food.csv        maxweight.hh  maxweight_test.cc        t
holly@holly-VirtualBox:~/Desktop/cs335/project-4-dynamic-
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: 500
9564.92
5000
82766.4
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

holly@holly-VirtualBox:~/Desktop/cs335/project-4-dynamic-
```

Scatter Plots





Complexity Analysis

Dynamic_max_weight()

```
 std::unique_ptr<FoodVector> dynamic_max_weight
(
const FoodVector& foods,
double total_calories
)
{

   // TODO: implement this function, then delete the return statement below
std::unique_ptr<FoodVector> result (new FoodVector); //1

int n = 0; //1
for (auto food : foods)
{
n++; //N
}


FoodVector foods_copy;

for (auto& food : foods) //N
{
foods_copy.push_back(std::shared_ptr<FoodItem>(
new FoodItem(
food->description() ,
food->calorie() ,
                    food->weight()
)));
}



std::vector< std::vector<double> > dynamicMatrix ( n+1, std::vector<double>(total_calories+1) );

for( int i = 0; i<=n; ++i ) //N+1
{
for( int j = 0; j<=total_calories; ++j ) //J+1
{

dynamicMatrix[i][j] = 0;
}
}
```

```
for( int i = 1; i<=n; ++i ) //N+1
{
for( int j = 1; j<=total_calories; ++j ) //J+1
{
int newj = j-foods_copy[i-1]->calorie(); //1

if(newj >=0) //1
{
double include = foods_copy[i-1]->weight() + dynamicMatrix[i-1][newj]; //1

if ( include > dynamicMatrix[i-1][j] ) //1
{
dynamicMatrix[i][j] = include; //1
}
else
{
dynamicMatrix[i][j] = dynamicMatrix[i-1][j]; //1
}

}
else
{
dynamicMatrix[i][j] = dynamicMatrix[i-1][j]; //1
}

}
}

//trace back from last [i][j]

double j = total_calories;

while (n > 0 && j >= 0)
    {
if( dynamicMatrix[n][j] != dynamicMatrix[n-1][j] ) //1
{
result->push_back(std::shared_ptr<FoodItem>(
new FoodItem(
foods_copy[n-1]->description() ,
foods_copy[n-1]->calorie() ,
             foods_copy[n-1]->weight()
))); //3

j = j-foods_copy[n-1]->calorie(); //2
}
n = n-1; //2
    }

  return result;
}
```

S.C = 1+1+N+3N+1+(N+1)(J+1) + NJ + max(2,1) + 1 + 3NJ +2 = O(n)

SO, THE TIME COMPLEXITY OF THIS ALGORITHM IS O(n).

Exhaustive_max_weight()

```
std::unique_ptr<FoodVector> exhaustive_max_weight
(
const FoodVector& foods,
double total_calorie
) {
int n = 0; //1

for (auto food : foods) //N
{
n++; //1
}

int total_todo = 1<<n; //2
std::unique_ptr<FoodVector> best (new FoodVector); //1 double
total_weight_best = 0; //1
for (int i = 0; i < total_todo; i++)  //2^N
{

std::unique_ptr<FoodVector> candidate (new FoodVector); //1

for (int j = 0; j <= (n-1); j++) //N
{
if ( ((1<<j) & i) != 0 ) //3
{
candidate->push_back(std::shared_ptr<FoodItem>( new
FoodItem(
foods[j]->description() ,
foods[j]->calorie() ,
                    foods[j]->weight()
))); //4
}
}  double total_weight_candidate = 0; //1

for (auto& food : (*candidate) ) //N
{
total_weight_candidate += food->weight(); //2
}


double total_calorie_candidate = 0; //1

for (auto& food : (*candidate) ) //N
{
total_calorie_candidate += food->calorie(); //2
}


if ( total_calorie_candidate <= total_calorie ) //2
{
if( total_weight_candidate > total_weight_best ) //1
{
best = std::move(candidate); //1
total_weight_best =  total_weight_candidate; //1
}
}
}

return best;

}
```

S.C = 1+N+2+1+1+ 2^N (1+ N(3+MAX(4,0)+1+2N+1+2N+2+MAX(2+1+MAX(2,0))
      = 5+N+ 2^N(1+ 7N + 4N + 4 + 5)
      = 5+N +2^N(11N+10)

SO, THE TIME COMPLEXITY OF THIS ALGORITHM IS O(2^n . n)

Question Answers

a. **Is there a noticeable difference in the performance of the two algorithms? Which is faster, and by how much? Does this surprise you?**

For dynamic search, the processing time tends to increase slowly like a linear line. For exhaustive searches, the processing time increases exponentially around input size of only 15. As the input size grows, the exhaustive search algorithm after input size 15's graph increases extremely fast by exponential processing time.
Thus, dynamic search is much faster. For example, input size of 21 will take exhaustive search close to 20 second while it only takes close to zero time if you use dynamic programming search.

The result is not surprising because it correlates to the mathematical analysis we did. Dynamic search algorithms are faster, especially for larger input sizes, due to their saved memory in the 2Dvector cache and more localized decision-making process.

b. **Are your empirical analyses consistent with your mathematical analyses? Justify your answer.**

My empirical analyses are consistent with my mathematical analyses. We can confirm this by looking at the graphs. The processing time of exhaustive search grows exponentially, which has exponential time complexity of O (2^n . n). On the other hand, the processing time of dynamic search grows like a linear function, which has O(n) time complexity. Overall, using both approaches verifies my analysis of each algorithm's efficiency.

c. **Is this evidence consistent or inconsistent with hypothesis 1? Justify your answer.**

Hypothesis 1 says exhaustive search algorithms are feasible to implement and produce correct outputs.

While exhaustive search might produce correct output with small sized input, it is not efficient because the algorithm's processing time grows extremely fast, takes up to 20 seconds with only input size of around 21. In real life, input size can grow much faster with larger amounts of data. So, while you can implement exhaustive search algorithms to produce correct results, they are impossible to work with very large input sizes.

d. **Is this evidence consistent or inconsistent with hypothesis 2? Justify your answer.**

Hypothesis 2 says algorithms with exponential running times are extremely slow, probably too slow to be of practical use.

Yes, the time versus input size chart for exhaustive search algorithm shows that the exhaustive algorithm increases very rapidly. If for input size of 21, exhaustive search costs 20 seconds for the program to run, it indicates that the exhaustive search approach becomes impractical for larger input sizes and much more quickly. In reality, input size can grow much faster with larger amounts of data. So, exhaustive search algorithms are mostly impractical to use with very large input sizes. Using dynamic search algorithms thus is extremely fast, linear time fast, comparing to exhautive search, which costs exponential running time.