Project 2: Greedy versus Exhaustive

By:
Cristopher Hernandez CPSC 335-02 (Bernstein)
cristopherh@csu.fullerton.edu

Carmine Infusino CPSC 335-04 (Wortman)
c.infusino@csu.fullerton.edu

Christian Medina CPSC 335-02 (Bernstein)
christian.medina@csu.fullerton.edu

The Hypotheses

This experiment will test the following hypotheses:

1. Exhaustive search algorithms are feasible to implement, and produce correct outputs.
2. Algorithms with exponential running times are extremely slow, probably too slow to be of practical use.

Mathematical Analysis of Pseudocode

The following is the problem we are trying to examine along with two algorithms that would solve the problem:

high-protein diet problem
input: a positive calorie budget C; and a vector V of food objects, where each food $f = (c, p)$ has an integer amount of calories $c > 0$ and protein $p \geq 0$ **output:** a vector K of food objects drawn from V, such that the sum of all calorie values is within the food budget i.e. $\sum_{(c,p) \in V} c \leq C$; and the sum of all protein values $\sum_{(c,p)} p$ is maximized

Greedy Algorithm:

```
greedy_max_protein(C, foods):
      todo = foods
      result = empty vector
      result_cal = 0
      while todo is not empty:
            Find the food f in todo of maximum protein.
            Remove f from todo.
            Let c be f's calories.
            if (result_cal + c) <= C:
                  result.add_back(f)
                  result_cal += c
      return result


      def merge(L, R):
```

```
 S = Vector()
 li = ri = 0
 while li < len(L) and ri < len(R):
       if L[li] <= R[ri]:
            S.add_back(L[li]) li += 1
       else:
           S.add_back(R[ri])
           ri += 1
 for i in range(li, len(L)):
           S.add_back(L[i])
for i in range(ri, len(R)):
            S.add_back(R[i])
return S

def merge(L, R):
      S = Vector()
      li = ri = 0
      while li < len(L) and ri < len(R):
      if L[li] <= R[ri]:
           S.add_back(L[li])
           li += 1
      else:
           S.add_back(R[ri])
           ri += 1
      for i in range(li, len(L)):
           S.add_back(L[i])
      for i in range(ri, len(R)):
           S.add_back(R[i])
      return S
```

The given efficiency for the greedy algorithm was either $O(n^2)$ or $O(n * logn)$.

Our implementation contains helper functions to sort the input so that we can speed up the algorithm to $O(n * logn)$. These helper functions are used to `Find the food f in todo of maximum protein.`

The `merge_sort` algorithm described above contains a time complexity of $O(n * logn)$.

Proofs for `merge_sort` and `merge` can be found on page 199 of *Algorithm Design in Three Acts.*

Analyzing our implementation, we can get a step count to ensure that it conforms to this expectation:

$f(n) = 6n + n * logn + 4$

Trying to prove: $f(n) \in O(n * logn)$

Let

$g(n) = n * logn \rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} \rightarrow \lim_{n\to\infty} \frac{6n+n*logn+4}{n*logn} \rightarrow \lim_{n\to\infty} \frac{6n}{n*logn} + \frac{n*logn}{n*logn} + \frac{4}{n*logn} \rightarrow \lim_{n\to\infty} 0 + 1 + 0$

= 1

The result is a constant, therefore f(n) = g(n) as $n \rightarrow \infty$. Therefore, $f(n) \in O(n * logn)$

Exhaustive Algorithm:

```
exhaustive_max_protein(C, foods):
     n = |foods|
     best = None
     for bits from 0 to (2^n -1):
          candidate = empty vector
          for j from 0 to n-1:
               if ((bits >> j) & 1) == 1:
                    candidate.add_back(foods[j])

          if total_calories(candidate) <= C:
               if best is None or
                  total_protein(candidate) > total_protein(best):
                    best = candidate
     return best
```

The given efficiency for the exhaustive algorithm was $O(2^n * n)$. Analyzing our implementation, we can get a step count to ensure that it conforms to this expectation:

$f(n) = 2^n * n + 8$

Trying to prove: $f(n) \in O(2^n * n)$

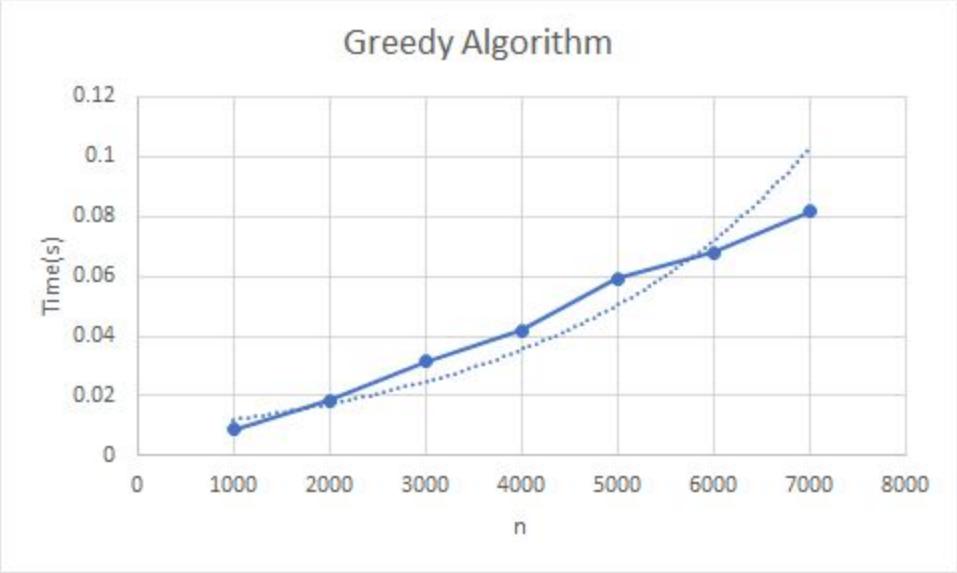
Let $g(n) = 2^n * n \rightarrow \lim_{n\to\infty} \frac{f(n)}{g(n)} \rightarrow \lim_{n\to\infty} \frac{2^n*n+8}{2^n*n} \rightarrow \lim_{n\to\infty} \frac{2^n*n}{2^n*n} + \frac{8}{2^n*n} \rightarrow \lim_{n\to\infty} 1 + 0 \rightarrow 1$

The result is a constant, therefore f(n) = g(n) as $n \rightarrow \infty$. Therefore, $f(n) \in O(2^n * n)$.

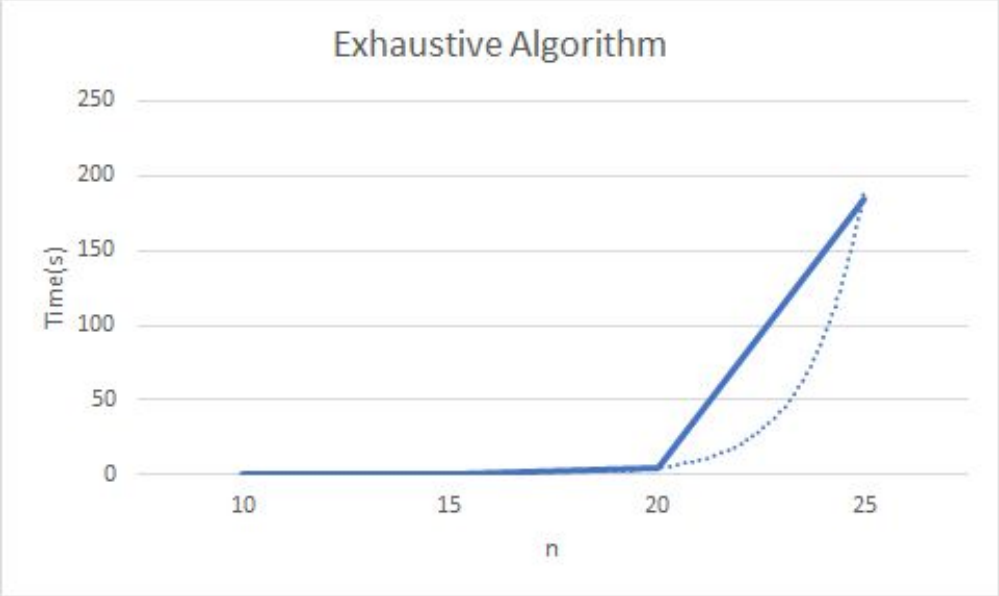
Empirical Analysis: Scatterplots of Implemented Functions

Each implementation used a kcal size of 2000.

Greedy n	Time(seconds)
1000	0.00874371
2000	0.0183258
3000	0.0316041
4000	0.0417203
5000	0.0592033
6000	0.068031
7000	0.0815632



Exhaustive n	Time(seconds)
10	0.00260952
15	0.121594
20	4.77656
25	185.109



Questions

a. **Is there a noticeable difference in the performance of the two algorithms? Which is faster, and by how much? Does this surprise you?**
   - The greedy algorithm is significantly faster. It was able to take in larger sizes of *n* and finish within less than 1 second every time. The exhaustive algorithm took significantly longer for *n* sizes smaller than 25. Anything after *n* = 30 was too long to quantify. This does not surprise us since the greedy Algorithm has a much faster efficiency class than the exhaustive algorithm.

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

- After plotting our points on the scatterplots, we noticed that the trendline for the exhaustive algorithm was definitely exponential. The time required to complete the algorithm begins with very small values. However, these values rapidly expand in an exponential manner. Trying larger sizes for *n* makes the algorithm extremely slow. The trendline for the greedy algorithm appears to be close to what one might expect from an $n * logn$ trendline. The growth of the greedy algorithm is more linear by comparison to the exhaustive algorithm. Essentially, its growth is far more gradual than that of the exhaustive algorithm, which grew more rapidly as *n* increased.

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

- The evidence produced during these experiments does seem to be consistent with the first hypothesis. The implementation of the exhaustive algorithm involved relatively few difficulties and bugs. As such, the process of getting the algorithm functioning was reasonable. Further, the output the algorithm produces is often more optimal than the greedy algorithm's solution. For example, with an *n* value of 15 the greedy algorithm produced a 118g protein diet under a 2000 kcal limit. However, the exhaustive algorithm produced a 141g protein diet under the same 2000 kcal limit, clearly a more optimal solution. If we increase the kcal limit to 3000 the same trend is found, with the results being a 190g and 202g protein diet, for the greedy and exhaustive algorithms respectively. In essence, the exhaustive algorithm was not wholly difficult to implement and certainly produces optimal solutions.

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

- The evidence we gathered appeared consistent with the second hypothesis. While the running times for small *n* values were negligible this very quickly changed as the value increased. For instance, at an *n* size of 10 the algorithm took roughly 0.003 seconds, a fast running time. However, by only adding 15 to that initial *n* value the running time balloons to 185 seconds. In fact, it was originally planned to test the algorithm at *n* = 30. However, it took so long that there were too many time limitations to get the experiment finished. The change in time requirements is so large for such a small difference. Comparatively, the greedy algorithm ran in roughly 0.0087 seconds for an *n* value of 1000. Further, the greedy algorithm increased by significantly smaller amounts with an *n* of 2000 taking only roughly 0.02 seconds. Clearly, this evidence is in favor of the second hypothesis. With such significant slowing in speed for small *n* increases, the exhaustive algorithm quickly becomes less useful.