Bin Packing Problem using Greedy Algorithms

Name: Matthew Trang
Student PID: mattluutrang

Date: March 21, 2022
Project: 2

1 Optimization Problem and Description

In this problem, we are given n items of different weights and bins each with capacity c, with the restriction that all weights must be less than or equal to c. The list of items is given sequentially, thus cannot be sorted beforehand, and the items are not necessarily in any order. The goal is to place all the items in bins to minimize the amount of bins that are needed to hold all of the items. This situation holds real world parallels to numerous logistics problems, such as loading shipping containers, or transportation of luggage at an airport.

1.1 Input Parameters

The input to the problem is a list of n objects of different weights. The additional parameter for the problem is the capacity of the bins, c. The inputs are considered in a sequential order, which limits the ability to sort the input list.

1.2 Objective Function

The objective function for the problem can be written as the following minimization of the number of bins, B, that are needed to hold all of the items. Given a finite set of O objects, a bin capacity c, and a weight $w_o \leq c$ for each $o \in O$,

$$\text{minimize } B = \sum_{i=1}^{n} y_i \tag{1}$$

where, y_i is an indicator variable, $y_i = 1$ if the bin i is used.
While the exact number for the optimal solution would require solving the optimization problem, we are able to formulate a lower-bound approximation for the number of bins used by taking the sum of all objects and dividing by the bin capacity, c, or

$$\lim_{B} \geq \lceil \sum_{i=1}^{n} w_i / c \rceil \tag{2}$$

1.3 Expected Outputs

The expected output of the algorithm is the number of bins needed to pack all the objects. For example, when given the input weights of $4, 8, 1, 2, 4, 1$ and a bin capacity of 10, the algorithm should output 2 bins, as one form of the optimal solution would place the items into bins of $[8, 2], [4, 4, 1, 1]$.

1.4 Large Size Challenges

The problem is an NP hard problem, as in order to find the global optimum, we need to consider every combination of objects within bins. Thus, finding the exact minimum number of bins takes factorial time. Therefore, given a large enough input, finding the optimal solution could be almost impossible to calculate.

2 Greedy Algorithms

2.0.1 First Greedy Algorithm

The first algorithm for the problem uses an idea of greedily selecting the spot which it fits best in, where the heuristic for best is in terms of the remaining space in the bin. In other words, if an object could fit into two different bins, it would chose the bin that has the least space remaining. In the algorithm, we start with an input array of weights for the objects, O, and a bin capacity, c. Then, we create an array of initially empty bins, B, and set our number of bins used currently to 0. We iterate through each object, and for each iteration step, we check all of the currently used bins to see which one the object will fit in best. If the object does fit in any one bin, then the space taken up by the object is added to that heuristically selected bin. Otherwise, we put the item into a new bin and increment the number of bins used.

Algorithm 1: Greedy 1(O, c, n)

```
Data: O is an array of object costs, c is the capacity of the bins, and n is the number of
      objects
Result: the number of bins needed to pack all the objects
1  begin
2      total ← 0
3      initialize the array of bins to be empty
4      for obj in O:
5          rem ← c + 1
6          idx ← 0
7          for j in range total:
8              if obj ≤ bins[j] and bins[j] - obj < rem:
9                  rem ← bins[j] + O[object]
10                 idx ← j
11         if c + 1 == rem:
12             bins[total] ← c - obj
13             total ← total + 1
14         else: if it fits
15             bins[idx] ← bins[idx] - obj
16     return total
```

This first greedy algorithm has a time complexity of $O(n^2)$, where n is the number of objects. The first for loop iterates through each object, and the asymptotic behavior of the algorithm occurs where each object takes its own bin, resulting in the second iteration over the number of bins to also be an iteration of the number of objects. The other operations in the algorithm are all of constant time. The space complexity of the algorithm is linear, as the algorithm has to initialize

the array of bins to be empty using the worst case size of the bins, which would be n bins.

2.0.2 Second Greedy Algorithm

The second algorithm uses the idea of greedily selecting the spot which was last used to store an object in to store the next object as well. This algorithm is simpler than the first algorithm, but runs in linear time. The algorithm also is able to compute the local optimal bin number without the use of additional allocated space for an array of bins, thus achieving constant space complexity. The algorithm first begins with a count of the total used bins initialized to 0 and a remainder variable set to the capacity of the bins. Then the algorithm iterates through the objects, and for each object, if the object fits in the remaining space in the bin, then the remaining space variable is decremented by the weight of the object. If the object does not fit in the remaining space, then a new bin is created, and the remaining space is set to the value of the capacity minus that of the object which was just placed.

Algorithm 2: Greedy 2(O, c, n)

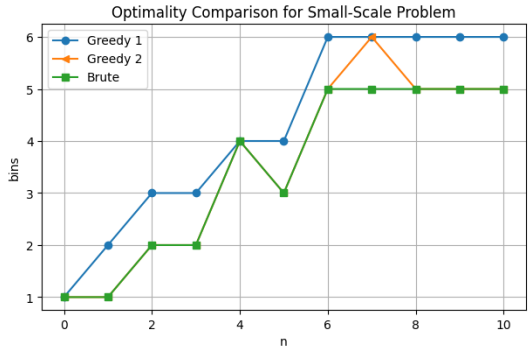
Data: O is a array of object costs, c is the capacity of the bins, and n is the number of objects
Result: the number of bins needed to pack all the objects

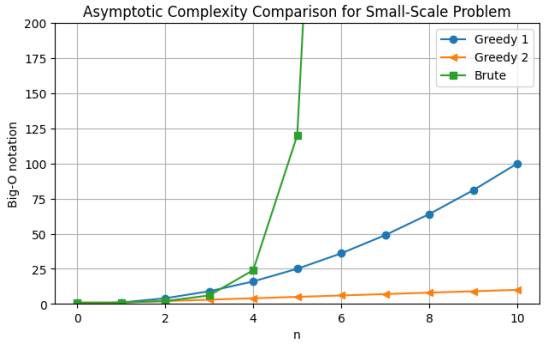
```
1 begin
2     total ← 0
3     rem ← c
4     for obj in O:
5         if obj ≤ rem:
6             rem ← rem - obj
7         else:  if it doesn't fit
8             total ← total + 1
9             rem ← c - obj
```

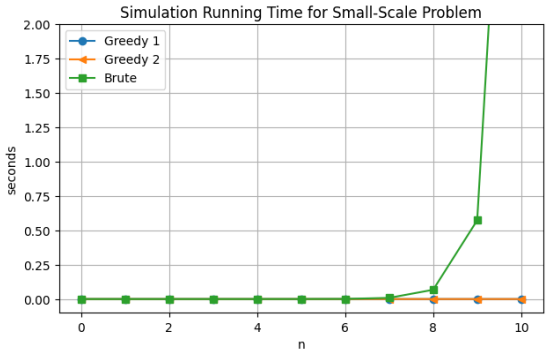
The second greedy algorithm has a time complexity of $O(n)$, where n is the number of objects. Due to the fact that the algorithm merely checks if it can fit in the previous bin, and not in any of the preceding bins, the algorithm only iterates through the objects once. The space complexity is also impacted by this, as there is no longer a need to store an array of n bins, and instead a single remaining capacity variable can be used. However, the optimality of this algorithm is significantly worse than that of the first greedy algorithm, as the bins which are passed over could have potentially been filled by another object.

3 Comparative Performance Analysis

3.1 Experimental Results - Small-size





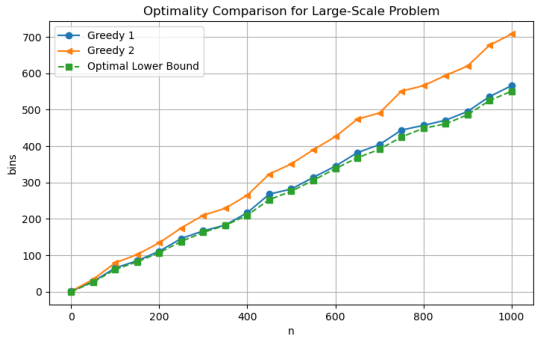


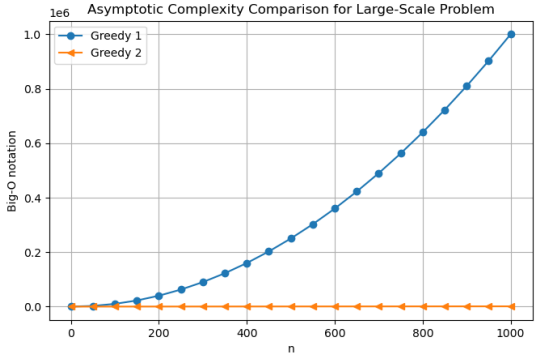
3.2 Small-size Results Analysis

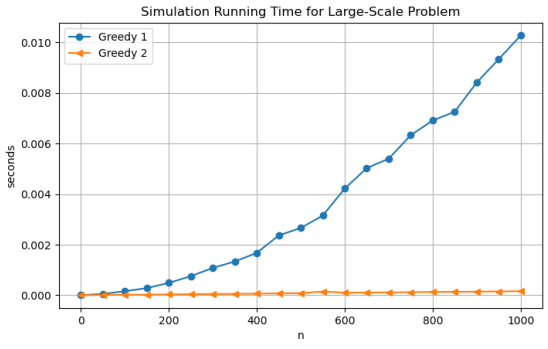
In the small-size experimental results, the number of bins that each algorithm used for the problem size n was compared. As expected, the green brute force method had the lowest number of bins across all n, as it achieves the optimal value. The first greedy algorithm unexpectedly performs worse than the second greedy algorithm, which is likely due to the fact that at a small problem size, there are less remaining objects that are able to perfectly fit the size of a leftover bin.

For the asymptotic complexity, the first greedy algorithm has a complexity of $O(n^2)$, while the other greedy algorithm is able to execute in $O(n)$ time, and the brute force algorithm is the worst with $O(n!)$ time. The simulation running time for the different algorithms also reflects this.

3.3 Experimental Results - Large-size







3.4 Large-size Results Analysis

In the large-size experimental results, it can be seen that the first greedy algorithm performs much closer to that of the theoretical optimal lower bound compared to that of the second greedy al-

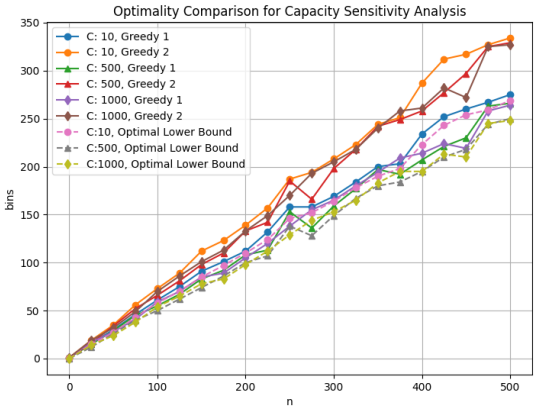
gorithm. As the value for n increases, the difference between greedy 1 and greedy 2 increases. Theoretically, as the limit of n approaches infinity, the algorithms will approach their true approximate optimal values.

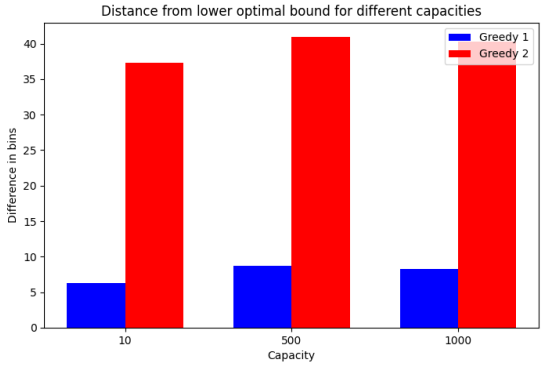
For the asymptotic complexity, the first greedy algorithm has a complexity of $O(n^2)$, while the other greedy algorithm is able to execute in $O(n)$ time, and the simulation time of the programs reflect this.

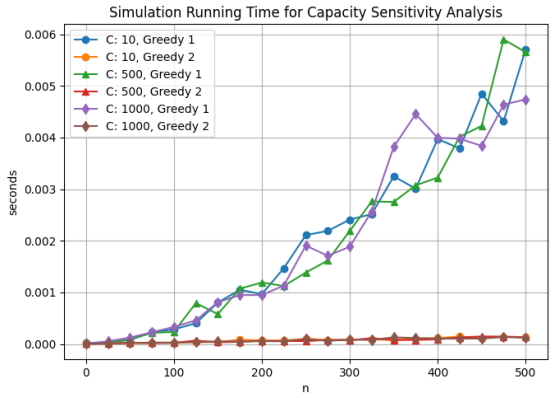
4 Parameters and Extensions

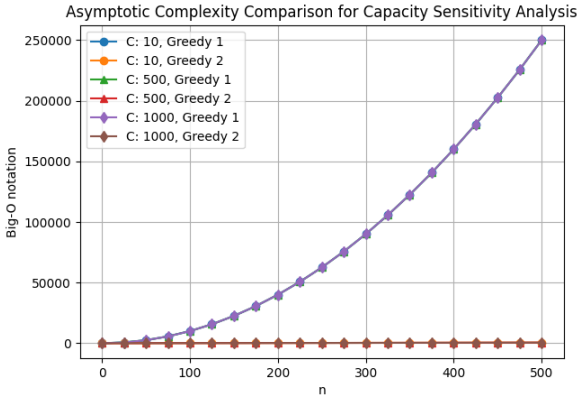
4.1 Parameter 1 - Capacity

In this section we first explore the effect of capacity of the bins. In the previous tests, our capacity was 10. For these parameter tests, we compare the results of capacities of 10, 500, and 1000.





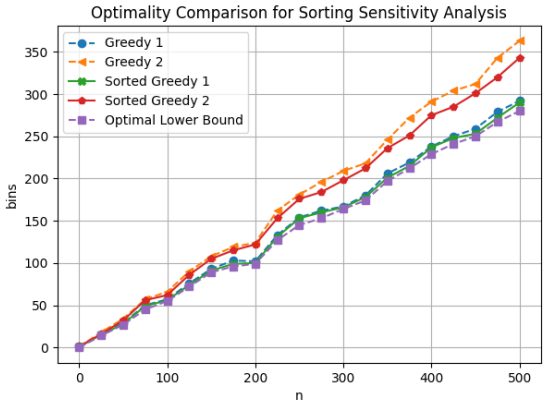


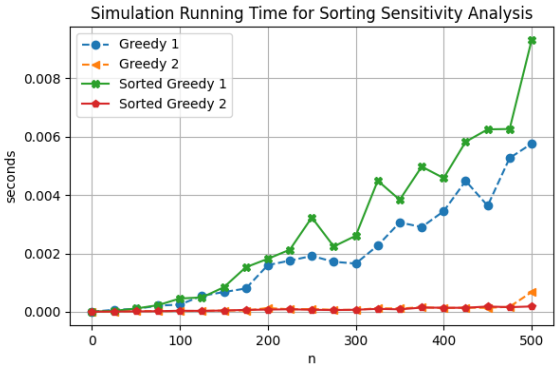


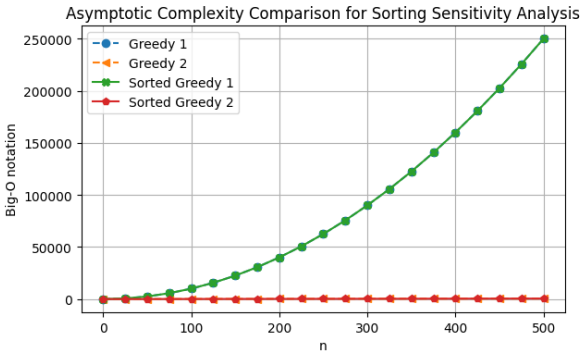
The change in capacity can be seen to be insignificant with respect to run time and complexity. This is likely because the capacity of the bins is not important for the runtime of the algorithm, and the complexities are still limited by the number of objects. However, there is a noticeable difference in terms of optimality, likely because with a larger capacity, the deviation of the weights of the generated objects increases, and it is slightly less likely that the objects will fit as optimally.

4.2 Parameter 2 - Sorted Input

In this section we first explore the effect of sorting the array of objects before passing the array to the algorithms. The arrays were sorted in decreasing order of their weights, and nothing was changed in either algorithm.







The sorted input did not affect the complexity of the algorithms, as the sorting was not considered part of the algorithm, therefore the asymptotic behavior of the algorithms did not change. It did however cause an increase in the simulation runtime for algorithm 1, and also a decrease in the amount of bins that both algorithms ended up using. The improvement in the optimality for the first algorithm makes sense, as the decreasing weights ensures that we are placing the smallest objects when there is the least space available. Similarly, for the second algorithm, once we reach the smallest objects, the sequential objects are more likely to also fit in the same bin, thus being more optimal. Run time increases for algorithm 1 because we fill bins faster in the beginning, thus