CS301 - ALGORITHMS — 2021-2022 FALL

Final Report

Group 13

Ahmet Ömer Kayabaşı - 27840

Albert Deniz Levi - 27801

Alperen Yıldız - 26758

Doğukan Yıldırım - 28364

Contents

I Bibliography

1 Problem Definition 7
1.1 The Problem 7
1.2 NP-Hardness 8

2 Algorithm Description 11
2.1 Brute Force Algorithm 11
2.1.1 Generating Permutations 11
2.1.2 Next-Fit Solver Function 12
2.1.3 Solution Phase 13
2.2 Greedy Algorithm 14
2.2.1 Best-Fit Algorithm 14
2.2.2 Ratio Bound 15

3 Algorithm Analysis 17
3.1 Brute Force Algorithm 17
3.1.1 Correctness of the Algorithm 17
3.1.2 Worst-Case Time Complexity 17
3.1.3 Space Complexity 18
3.2 Greedy Algorithm 18
3.2.1 Correctness of the Algorithm 18
3.2.2 Worst-Case Time Complexity 18
3.2.3 Space Complexity 18

4 Sample Generation & Brute Force Testing 19
4.1 Sample Generation 19
4.2 Complete Code of Brute Force Algorithm 19
4.3 Test Case Outputs of Brute Force Alogrithm 21

5 Experimental Analysis of The Performance 25
5.1 Initial Performance Testing 25
5.2 Bin Size Indifference 29
5.3 Approximation Performance 30

6 Experimental Analysis of the Correctness 31
6.1 Black Box Testing 31
6.2 White Box Testing 33

7 Discussion 37

8 Performance Testing Code 39
8.1 Performance Testing With Different Item Counts 39
8.2 Approximation Performance 41

Bibliography

1 Problem Definition 7
1.1 The Problem
1.2 NP-Hardness

2 Algorithm Description 11
2.1 Brute Force Algorithm
2.2 Greedy Algorithm

3 Algorithm Analysis 17
3.1 Brute Force Algorithm
3.2 Greedy Algorithm

4 Sample Generation & Brute Force Testing 19
4.1 Sample Generation
4.2 Complete Code of Brute Force Algorithm
4.3 Test Case Outputs of Brute Force Alogrithm

5 Experimental Analysis of The Performance 25
5.1 Initial Performance Testing
5.2 Bin Size Indifference
5.3 Approximation Performance

6 Experimental Analysis of the Correctness 31
6.1 Black Box Testing
6.2 White Box Testing

7 Discussion 37

8 Performance Testing Code 39
8.1 Performance Testing With Different Item Counts
8.2 Approximation Performance

1. Problem Definition

1.1 The Problem

Bin packing is the problem of finding the minimum number of B size bags that can house a list of n items with various sizes that individually do not exceed the size of B. We are given the bag size and the list of item sizes at the start of the solution. Objective is to partition the items such that the minimum number of bags are used. The output will not be the partition itself but the number of minimum bags.
A more formal way of portraying the problem is;

Definition 1.1.1 — Bin Packing. Given finite set U of m items; for each item u in U, a positive integer size $s(u)$, a positive integer B (called the bin capacity), what is minimum number k (of bins) such that items in U can be partitioned into k disjoint sets $U_i, ..., U_k$ where for each $U_i, (1 <= i <= k)$ the total sum of the sizes of the items in U_i does not exceed B?

On figure 1.1, an example of bin packing for the list of items with the weights 2, 2, 3, 4 and 4 into bins the size of $B = 13$ can be seen. This is one of the many optimal solutions for this input. We know this is an optimal solution as the minimum amount of bins required is 2 since total item volume requires at least 2 bins.

Proposition 1.1.1 — The lower-bound for bin-packing problem. Let m be the number of items in the set, w_i be the weight of an item, and B be the bin size. The minimum required bins, i.e. the lower-bound for bin-packing problem can be shown follows:

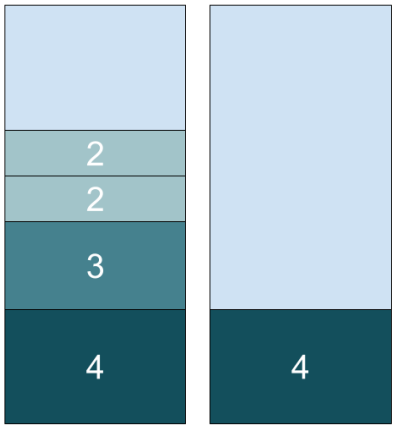
$$\left\lceil \frac{\sum_{i=1}^{m} w_i}{B} \right\rceil \tag{1.1}$$

According to the example in figure 1.1, the minimum required bins is:

$$\left\lceil \frac{\sum_{i=1}^{m} w_i}{B} \right\rceil = \left\lceil \frac{2+2+3+4+4}{13} \right\rceil = 2 \tag{1.2}$$

Moreover, the maximum number of bins that can be allocated for m items is $\boldsymbol{m}$, where m is the number of items in the set.

Figure 1.1: An Example



Bin-packing used to be a major problem in the shipping industry before the implementation of ISO Containers and can still be seen in post offices. An example of bin-packing problem in the shipping industry would be trying to minimize the number of trucks that should be loaded with packages where the bin size is the truck's weight capacity. It is an optimization problem and wherever there is a space that needs to be filled with regular rectangular objects, bin packing is the problem.

1.2 NP-Hardness

A problems' NP-Hard status can be proven if a problem previously proven to be NP-Hard can be reduced to it. We will prove Bin Packing is NP-Hard by reducing The Partition Problem to it, which we will show a proof of NP-hardness by the use of the Sub Reduction Problem.

This approach to prove the NP-Hard property of Bin packing was suggested by Dexter C. Kozen in his book "The Design and Application of Algorithms"[6]. We will write the proofs more explicitly.

First, let us define the partition problem and reduce the Subset Sum Problem into it. Given a finite set S and integer weight function $x : S \to H$, does there exist a subset

$$S' \subseteq S \tag{1.3}$$

such that,

$$\sum_{a \in S - S'} w(a) = \sum_{a \in S'} w(a) \tag{1.4}$$

Subset Sum can be reduced into this by taking SS, the sum of the subset to be equal to half the sum of all the elements in the set. More formally:

$$SS = \frac{1}{2} \sum_{a \in S'} w(a) \tag{1.5}$$

In other words, we are saying if we can find any size of a subset when we solve subset sum problem, then we should be able to find a subset that divides (partitions) the set right in the middle. Now we will reduce the partition problem into Bin Packing. Bin packing is no different from the Partition

Problem when $k = 2$ (the bin count is 2). We just need to repeat the previous method. Let us assume B is the bin size.

$$B = \frac{1}{2} \sum_{a \in S'} w(a) \tag{1.6}$$

This results in 2 subsets fitting in 2 bins. Partition fits in Bin Packing hence Bin packing Problem is NP-Hard.

2. Algorithm Description

Many algorithms can be found on the internet which provide a solution, but not always the best (exact) solution. In our case, we are trying to come up with the minimum number of bins given the bin size and the list of the items. Thus in this project, we are following a brute force approach to solve this problem. The algorithm we've designed consists of two steps:

2.1 Brute Force Algorithm

2.1.1 Generating Permutations

For an item list l, the permutation algorithm derives all the possible permutations of the given list. Here is an implementation of a function from the itertools[5] Python module that generates all the permutations of a given list.

```
def permutations(iterable, r=None):
    # permutations('ABCD', 2) --> AB AC AD BA BC BD CA CB CD DA DB DC
    # permutations(range(3)) --> 012 021 102 120 201 210
    pool = tuple(iterable)
    n = len(pool)
    r = n if r is None else r
    if r > n:
        return
    indices = range(n)
    cycles = range(n, n-r, -1)
    yield tuple(pool[i] for i in indices[:r])
    while n:
        for i in reversed(range(r)):
            cycles[i] -= 1
```

```
            if cycles[i] == 0:
                indices[i:] = indices[i+1:] + indices[i:i+1]
                cycles[i] = n - i
            else:
                j = cycles[i]
                indices[i], indices[-j] = indices[-j], indices[i]
                yield tuple(pool[i] for i in indices[:r])
                break
        else:
            return
```

Corollary 2.1.1 — Permutation Result. Given an array of strings [“a”,”b”,”c”], permutations function returns:

(“a”,”b”,”c”) (2.1)
(“a”,”c”,”b”) (2.2)
(“b”,”a”,”c”) (2.3)
(“b”,”c”,”a”) (2.4)
(“c”,”a”,”b”) (2.5)
(“c”,”b”,”a”) (2.6)

With the permutations function, we are trying to find all the possible item insertion orderings. The running time of the permutations function is $\theta(n!)$.

2.1.2 Next-Fit Solver Function

The pseudocode[1] for the Next-Fit algorithm is as follows:

Algorithm 1 Next-Fit

for *All objects i = 1,2,...,n* **do**
  if *Object i fits in current bin* **then**
    Pack object i in current bin
  else
    Create new bin, make it the current bin, and pack object i
  end
end

Packing an object takes constant time, and the algorithm is dominated by the for loop. Thus, the running time for the nextfit function is $\theta(n)$.

The nextfit function takes two parameters, weight is the list of the items we are trying to insert into the bins, and c is the bin size. The nextfit function returns two parameters: result of the algorithm which is the number of bins used for inserting the elements, and an array containing the item placements into the bins.[2]

```
def nextfit(weight,c):
    res = 0
```

```
    rem = c
    subArr = []
    temp = []
    for _ in range(len(weight)):
        if rem >= weight[_]:
            rem = rem - weight[_]
            temp.append(weight[_])
        else:

            subArr.append(temp)
            temp = []

            res += 1
            rem = c - weight[_]
            temp.append(weight[_])
    subArr.append(temp)
    return res + 1,subArr
```

2.1.3 Solution Phase

The Pseudo Code for the algorithm is as follows:

Algorithm 2 Solution

```
Create all the permutations of item list
for every permutation inside the permutations array do
    Perform Next-Fit function for permutation
    if the resulting number of bins is smaller than the current minimum then
        update the current minimum
    end
    Print the minimum
end
```

In the actual solution, given item list ‘items’ and bin size ‘binSize’, permutations function returns a list containing all the permutations of the items. And for every element inside the permutations array, Next-Fit algorithm is run.

```
    #SOLVING PHASE
    permutations = list(itertools.permutations(items))
    min = sys.maxsize # corresponds to INT_MAX
    minArr = []
    minPermutation = []
    print(f"TEST CASE\n")
    print(f"Bin Size: {binSize} - Items List: {items}\n")
    for elem in permutations:
```

```
9       res,subArr = nextfit(elem,binSize)
10      if res < min:
11        min = res
12        minArr = subArr
13        minPermutation = elem
14
15    print(f"Permutation: {minPermutation}")
16    print(f"Minimum number of bins required: {min}")
17    print(f"Resulting distribution: {minArr}")
18    print("\nAll permutations are checked for this test case")
19    print("\n-----------------------------------------------------\n")
```

If the result returned by the Next-Fit algorithm is smaller than the current minimum. The minimum number of bins is updated. After all the permutations are considered, the algorithm prints out the most efficient placement of the items inside the bins.

2.2 Greedy Algorithm

2.2.1 Best-Fit Algorithm

The greedy algorithm we have chosen is Best-Fit. The goal of Best-Fit is to put the following item in the **tightest** space possible. That is, place it in the bin with the least amount of vacant space possible. The pseudocode[1] for the Best-Fit algorithm is as follows:

Algorithm 3 Best-Fit

for *All objects i = 1,2,...,n* **do**
  for *All bins j = 1,2,... do* **do**
    if *Object i fits in bin j* **then**
      Calculate remaining capacity after the object has been added.
    end
  end
  Pack object i in bin j, where j is the bin with minimum remaining capacity after adding the object (i.e. the object "fits best").
  If no such bin exists, open a new one and add the object.
end

Best-Fit is a greedy algorithm, so although it doesn't provide the optimal solution all the time, it provides a much faster solution than the brute force approach. Below is the implementation[2] of the Best-Fit algorithm.

```
1  def bestFit(weight, n, c):
2          # Initialize result (Count of bins)
3          res = 0
4
5          # Create an array to store
6          # remaining space in bins
```

```
        # there can be at most n bins
        bin_rem = [0]*n

        # Place items one by one
        for i in range(n):

                # Find the first bin that
                # can accommodate
                # weight[i]
                j = 0

                # Initialize minimum space
                # left and index
                # of best bin
                min = c + 1
                bi = 0

                for j in range(res):
                        if (bin_rem[j] >= weight[i] and bin_rem[j] - weight[i] < min):
                                bi = j
                                min = bin_rem[j] - weight[i]

                # If no bin could accommodate weight[i],
                # create a new bin
                if (min == c + 1):
                        bin_rem[res] = c - weight[i]
                        res += 1

                else: # Assign the item to best bin
                        bin_rem[bi] -= weight[i]

        return res
```

2.2.2 Ratio Bound

Ratio bound of Best-Fit algorithm is found to be 1.7 · OPT (optimal). A brief proof of the ratio bound of the Best-Fit algorithm can be given with two lemmas and one theorem. Proof is taken from Optimal Analysis of Best Fit Bin Packing written by Dósa, G., & Sgall, J in 2014[4]:

Lemma 1 At any moment, in the BF packing the following holds:

1. The sum of levels of any two bins is greater than 1. In particular, there is at most one bin with level at most 1/2.
2. Any item a with level at most 1/2 (i.e., packed into the single bin with level at most 1/2)

does not fit into any bin open at the time of its arrival, except for the bin where the item a is packed.

3. If there are two bins B, B with level at most 2/3, in this order, then either B contains a single item or the first item in B is huge.

To illustrate the lemma, we now present a short proof of the asymptotic ratio 1.7 for Best-Fit. It uses the same weight function as the traditional analysis of Best-Fit. (In some variants the weight of an item is capped to be at most 1, which makes almost no difference in the analysis.) To use amortization, we split the weight of each item a into two parts, namely its bonus ϖ(a) and its scaled size ϖs(a), defined as

$$\varpi(a) = \begin{cases} 0 & \text{if } a \leq \frac{1}{6} \\ \frac{3}{5}(a - \frac{1}{6}) & \text{if } a \in (\frac{1}{6}, \frac{1}{3}) \\ 0.1 & \text{if } a \in [\frac{1}{3}, \frac{1}{2}] \\ 0.4 & \text{if } a > \frac{1}{2} \end{cases} \tag{2.7}$$

For every item a we define $\varpi s(a) = \frac{6}{5}a$ and its weight is $w(a) = \varpi s(a) + \varpi(a)$. For a set of items B,

$$\sum_{a \in B} \omega(a) \tag{2.8}$$

denotes the total weight, similarly for $\varpi(a)$ and $\varpi s(a)$ It is easy to observe that the weight of any bin B, i.e., of any set with $s(B) \leq 1$, is at most 1.7.

The key part is to show that, on average, the weight of each BF-bin is at least 1. Lemma 2 together with Lemma 1 implies that for almost all bins with two or more items, its scaled size plus the bonus of the following such bin is at least 1

Lemma 2 Let B be a bin such that $s(B) \geq 2/3$ and let c, c' be two items that do not fit into B, i.e, $c, c' > 1 - s(B)$. Then $\varpi s(B) + \varpi(c) + \varpi(c') \geq 1$.

This simple proof holds for a wide class of any-fit-type algorithms: Call an algorithm a RAAF (relaxed almost any fit) algorithm, if it uses the bin with level at most 1/2 only when the item does not fit into any previous bin (Lemma 2.1(i) implies that there is always at most one such bin). Our proof of the asymptotic ratio can be tightened so that the additive constant is smaller:

Theorem 2.2.1 For any RAAF algorithm A and any instance of bin packing we have $A \leq \lfloor 1.7 \cdot \mathrm{OPT} + 0.7 \rfloor \leq \lceil 1.7 \cdot \mathrm{OPT} \rceil$.

Thus as a result of Lemma 2.1 and Lemma 2.2, ratio bound of Best-Fit algorithm can briefly be given as $1.7 \cdot \mathrm{OPT}$.[4]

3. Algorithm Analysis

3.1 Brute Force Algorithm

3.1.1 Correctness of the Algorithm

Let us use mathematical induction to prove the correctness of this algorithm. Suppose we have k elements and bins with m capacity. Let us define S as the set of all possible correct solutions to this problem and S' as the set of solutions returned by our algorithm.

The base case for this algorithm would be $k = 1$. Since the first part of our solution will create all possible permutations, we can make sure that the first element placed in the first bin will definitely match with all solutions contained in S. Therefore, the base case is correct.

$$S'[1...1] \in S[1..1] \tag{3.1}$$

For the induction hypothesis, let us assume that S' contains all solutions in S for the first k elements.

$$S'[1...k] \in S[1..k] \tag{3.2}$$

For the $(k+1)$th element, because all correct placements for k elements will be contained in S, $(k+1)$th element will be able to placed on any valid place. Therefore, with the addition of $(k+1)$th element, S' will contain all possible correct permutations in $k+1$ elements:

$$S'[1...k+1] \in S[1..k+1] \tag{3.3}$$

Which concludes the mathematical induction and shows that our algorithm is correct.

3.1.2 Worst-Case Time Complexity

The time complexity of getting all the permutations of a list is $\theta(n!)$, and the time complexity of Next-Fit algorithm is $\theta(n)$ since it iterates over all the elements of the list in a single for loop. Firstly, all permutations will be generated, then each permutation will be given to the Next-Fit

algorithm, therefore the worst-case time complexity of our brute force algorithm will be:

$$T(n) = \text{Permutation generation} + \text{Next-Fit algorithm for each permutation} \tag{3.4}$$
$$T(n) = \theta(n!) + \theta(n!) \cdot \theta(n) \tag{3.5}$$
$$T(n) = \theta(n! \cdot n) \tag{3.6}$$

3.1.3 Space Complexity

The space complexity of all the permutations of a list is $\theta(n!)$ and the space complexity of Next-Fit algorithm is $\theta(n)$. In the worst case, each item in the items list will be put into a separate bin. Therefore for each permutation, there will be n bins in the worst case. Hence, the space complexity of our brute force algorithm will be:

$$S(n) = \text{All permutations are stored} + \text{For each permutation there are n bins} \tag{3.7}$$
$$S(n) = \theta(n!) + \theta(n!) \cdot \theta(n) \tag{3.8}$$
$$S(n) = \theta(n! \cdot n) \tag{3.9}$$

3.2 Greedy Algorithm

3.2.1 Correctness of the Algorithm

As explained in **Section 2.2.2** with ratio bounds, the Best-Fit algorithm always provides a solution, but the provided solution may not be the optimal one. This is due to the fact that the Best-Fit algorithm is a greedy algorithm, that it tries to reach a globally optimal solution by making a locally optimal choice. Despite that, the Best-Fit algorithm is a much faster algorithm compared to algorithms that take on a brute force approach. Moreover, one must mention that it is proven that it is not possible to get more than $1.7 \cdot \text{OPT}$ (optimal) number of bins with the Best-Fit algorithm.

3.2.2 Worst-Case Time Complexity

Initially, the Best-Fit algorithm creates an array to store n number of bins due to the fact that in the worst-case there can be a maximum number of n bins. At each iteration of the outer for loop, in other words, for each item in the items list, all the bins are checked in the inner for loop to find the bin with the minimum remaining capacity after adding the said item so tightest/best fit is achieved. Iterating over n items is $\theta(n)$, and iterating over all n bins is $\theta(n)$, therefore the worst-case time complexity of the Best-Fit algorithm is:

$$T(n) = \text{Iterate over n items} \cdot \text{Iterate over n bins} \tag{3.10}$$
$$T(n) = O(n \cdot n) \tag{3.11}$$
$$T(n) = O(n^2) \tag{3.12}$$

3.2.3 Space Complexity

In the worst-case, there can be a maximum number of n bins, therefore the worst-case space complexity of the Best-Fit algorithm is:

$$S(n) = \theta(n) \tag{3.13}$$

4. Sample Generation & Brute Force Testing

4.1 Sample Generation

The code below was used for generating input samples.

```
# GENERATING SAMPLES FOR THE ALGORITHM
testCount = 10
for _ in range(testCount):
  binSize = random.randint(4,15)
  itemCount = random.randint(5,10)
  items = []
  for x in range(itemCount):
    items.append(random.randint(1,binSize))
```

4.2 Complete Code of Brute Force Algorithm

The complete code of our brute force algorithm with test case generation and the output of the said code for 10 test cases can be found below.

```
import sys
import itertools
import random

def nextfit(weight,c):
    res = 0
    rem = c
```

```
    subArr = []
    temp = []
    for _ in range(len(weight)):
        if rem >= weight[_]:
            rem = rem - weight[_]
            temp.append(weight[_])
        else:

            subArr.append(temp)
            temp = []

            res += 1
            rem = c - weight[_]
            temp.append(weight[_])
    subArr.append(temp)
    return res + 1,subArr

#GENERATING TEST CASES
testCount = 10
for _ in range(testCount):
  binSize = random.randint(4,15)

  itemCount = random.randint(5,10)
  items = []
  for x in range(itemCount):
    items.append(random.randint(1,binSize))

  #SOLVING PHASE
  permutations = list(itertools.permutations(items))
  min = sys.maxsize # corresponds to INT_MAX
  minArr = []
  minPermutation = []
  print(f"TEST CASE\n")
  print(f"Bin Size: {binSize} - Items List: {items}\n")
  for elem in permutations:
    res,subArr = nextfit(elem,binSize)
    if res < min:
      min = res
      minArr = subArr
      minPermutation = elem

```

```
49    print(f"Permutation: {minPermutation}")
50    print(f"Minimum number of bins required: {min}")
51    print(f"Resulting distribution: {minArr}")
52    print("\nAll permutations are checked for this test case")
53    print("\n-----------------------------------------------------\n")
```

4.3 Test Case Outputs of Brute Force Alogrithm

TEST CASE

Bin Size: 11 – Items List: [7, 8, 7, 6, 1]

Permutation: (7, 8, 7, 6, 1)
Minimum number of bins required: 4
Resulting distribution: [[7], [8], [7], [6, 1]]

All permutations are checked **for** this test case

TEST CASE

Bin Size: 4 – Items List: [3, 4, 1, 2, 2, 2, 4, 2]

Permutation: (3, 1, 4, 2, 2, 2, 2, 4)
Minimum number of bins required: 5
Resulting distribution: [[3, 1], [4], [2, 2], [2, 2], [4]]

All permutations are checked **for** this test case

TEST CASE

Bin Size: 15 – Items List: [5, 12, 9, 13, 7, 4, 9]

Permutation: (5, 9, 12, 13, 7, 4, 9)
Minimum number of bins required: 5
Resulting distribution: [[5, 9], [12], [13], [7, 4], [9]]

All permutations are checked **for** this test case

TEST CASE

Bin Size: 13 – Items List: [4, 4, 13, 1, 5]

Permutation: (4, 4, 13, 1, 5)
Minimum number of bins required: 3
Resulting distribution: [[4, 4], [13], [1, 5]]

All permutations are checked **for** this test case

——

TEST CASE

Bin Size: 10 – Items List: [4, 10, 3, 9, 3, 1, 10]

Permutation: (4, 3, 3, 10, 9, 1, 10)
Minimum number of bins required: 4
Resulting distribution: [[4, 3, 3], [10], [9, 1], [10]]

All permutations are checked **for** this test case

——

TEST CASE

Bin Size: 14 – Items List: [1, 11, 5, 7, 13, 2]

Permutation: (1, 11, 5, 7, 2, 13)
Minimum number of bins required: 3
Resulting distribution: [[1, 11], [5, 7, 2], [13]]

All permutations are checked **for** this test case

——

TEST CASE

Bin Size: 5 – Items List: [5, 3, 2, 5, 2]

Permutation: (5, 3, 2, 5, 2)
Minimum number of bins required: 4
Resulting distribution: [[5], [3, 2], [5], [2]]

All permutations are checked **for** this test case

——

TEST CASE

Bin Size: 11 – Items List: [5, 6, 1, 7, 7, 10, 9, 4, 10, 4]

Permutation: (5, 6, 1, 10, 7, 4, 7, 4, 9, 10)
Minimum number of bins required: 6
Resulting distribution: [[5, 6], [1, 10], [7, 4], [7, 4], [9], [10]]

All permutations are checked **for** this test case

TEST CASE

Bin Size: 4 – Items List: [3, 3, 2, 2, 1, 4, 1, 2]

Permutation: (3, 3, 2, 2, 1, 1, 2, 4)
Minimum number of bins required: 5
Resulting distribution: [[3], [3], [2, 2], [1, 1, 2], [4]]

All permutations are checked **for** this test case

TEST CASE

Bin Size: 13 – Items List: [5, 6, 1, 7, 6]

Permutation: (5, 6, 1, 7, 6)
Minimum number of bins required: 2
Resulting distribution: [[5, 6, 1], [7, 6]]

All permutations are checked **for** this test case

---

5. Experimental Analysis of The Performance

5.1 Initial Performance Testing

Our heuristic approach (Best-Fit algorithm is used) has $O(n^2)$ complexity where n is the number of items, therefore altering the number of items is necessary in order to test the time complexity of the algorithm. But we will also be showing results with different bin sizes to prove it is indifferent.

For the cases below, there are 100 iterations for each test case. The results of our performance testing provide a precise polynomial time (with degree 2), namely $O(n^2)$, with respect to the calculations in the algorithm analysis **Section 3.2.2**. It seems that the worst-case time complexity of the greedy algorithm Best-Fit is what is displayed in practice.

Please see the following pages to observe the results of our performance testing in detail. The code of this Chapter is included in Chapter 8.

itemCount	binSize	Mean Time (s)	Standard Deviation	Standard Error	%90CL LB	%90CL UB	%95CL LB	%95CL UB
100	10	0.00015625	0.0015625	0.00015625	-0.0001007812	0.00041328125	-0.00015	0.0004625
200	10	0.0003125	0.002198520221	0.0002198520221	-4.92E-05	0.00067415657	-0.0001184099	0.0007434099
300	10	0.00140625	0.004494117051	0.0004494117051	0.0006669677	0.00214553225	0.00052540305	0.0022870969
400	10	0.0021875	0.00544898575	0.000544898575	0.0012911418	0.00308385815	0.0011194987	0.0032555012
500	10	0.00359375	0.006608634136	0.0006608634136	0.0025066296	0.00468087031	0.0022984577	0.0048890422
600	10	0.00515625	0.007384086916	0.0007384086916	0.0039415677	0.00637093229	0.0037089689	0.0066035310
700	10	0.0071875	0.007826691656	0.0007826691656	0.0059000092	0.00847499077	0.0056534684	0.0087215315
800	10	0.00921875	0.007723610475	0.0007723610475	0.0079482160	0.01048928392	0.0077049223	0.0107325776
900	10	0.011875	0.006706792125	0.0006706792125	0.0107717327	0.0129782673	0.0105604687	0.0131895312
1000	10	0.01484375	0.003422545524	0.0003422545524	0.0142807412	0.01540675874	0.0141729310	0.0155145689
1100	10	0.018125	0.005757077331	0.0005757077331	0.0171779607	0.01907203922	0.0169966128	0.0192533871
1200	10	0.0215625	0.0076223708	0.00076223708	0.02030862	0.02281638	0.02006851532	0.0230564846
1300	10	0.0246875	0.007750702559	0.0007750702559	0.0234125094	0.02596249057	0.0231683623	0.0262066377
1400	10	0.02953125	0.005831557495	0.0005831557495	0.0285719587	0.03049054121	0.0283882647	0.0306742352
1500	10	0.0340625	0.00603317062	0.000603317062	0.0330700434	0.03505495657	0.0328799985	0.0352450014
1600	10	0.03828125	0.008122024999	0.0008122024999	0.0369451768	0.03961732311	0.036689333	0.0398731669
1700	10	0.04203125	0.00726286248	0.000726286248	0.0408365091	0.04322599088	0.0406077289	0.0434547710
1800	10	0.0484375	0.004711114759	0.0004711114759	0.0476625216	0.04921247838	0.0475141215	0.0493608784
1900	10	0.05390625	0.0078125	0.00078125	0.0526210937	0.05519140625	0.052375	0.0554375
2000	10	0.059375	0.006281486345	0.0006281486345	0.0583416955	0.0604083045	0.0581438286	0.0606061713

Figure 5.1: 100 Tests each - Bin Size 10

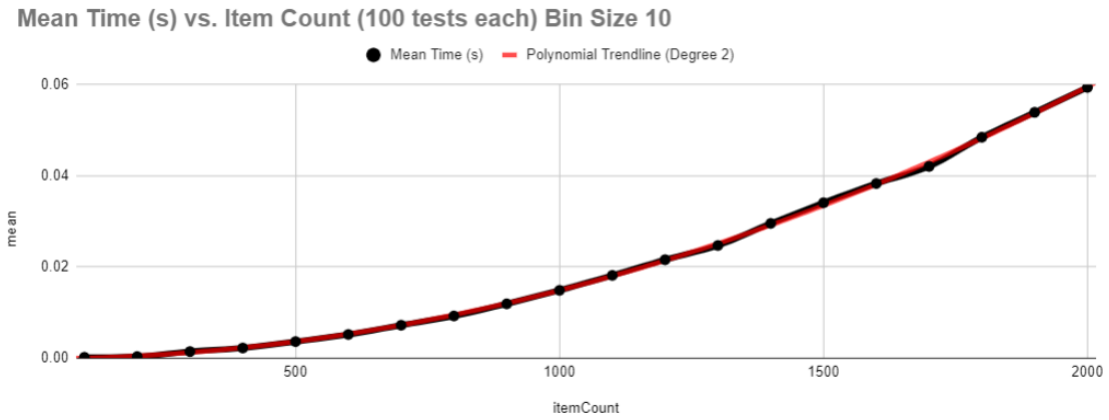


Figure 5.2: Time Complexity Plot - 100 Tests each - Bin Size 10

itemCount	binSize	Mean Time (s)	Standard Deviation	Standard Error	%90CL LB	%90CL UB	%95CL LB	%95CL UB
100	10	0.000203125	0.001770791931	5.60E-05	0.0001110093	0.0002952406	9.34E-05	0.0003128798
200	10	0.000734375	0.003308511596	0.0001046243231	0.0005622679	0.0009064820	0.0005293113	0.0009394386
300	10	0.00146875	0.004562105325	0.0001442664375	0.0012314317	0.0017060682	0.0011859877	0.0017515122
400	10	0.002328125	0.005566666407	0.0001760334482	0.0020385499	0.0026177000	0.0019830994	0.0026731505
500	10	0.003640625	0.006608652625	0.0002089839456	0.0032968464	0.0039844035	0.0032310164	0.0040502335
600	10	0.005421875	0.007441467853	0.0002353198755	0.0050347738	0.0058089761	0.0049606480	0.0058831019
700	10	0.00721875	0.007793802701	0.0002464616817	0.0068133205	0.0076241794	0.0067356851	0.0077018148
800	10	0.00940625	0.00765203704	0.0002419786579	0.0090081951	0.0098043048	0.0089319718	0.0098805281
900	10	0.011890625	0.006666970919	0.000210828132	0.0115438127	0.0122374372	0.0114774018	0.0123038481
1000	10	0.014890625	0.003453083987	0.0001091961035	0.0147109974	0.0150702525	0.0146766006	0.0151046493
1100	10	0.018015625	0.005627619198	0.0001779609447	0.0177228792	0.0183083707	0.0176668215	0.0183644284
1200	10	0.021734375	0.0113175451	0.0003578922003	0.0211456423	0.0223231076	0.0210329062	0.0224358437
1300	10	0.025046875	0.007680647235	0.0002428833917	0.0246473318	0.0254464181	0.0245708235	0.0255229264
1400	10	0.02909375	0.00561381451	0.0001775244021	0.0288017223	0.0293857776	0.0287458021	0.0294416978
1500	10	0.033359375	0.00534210254	0.0001689321152	0.0330814816	0.0336372683	0.0330282680	0.0336904819
1600	10	0.0378125	0.007715710495	0.0002439921893	0.0374111328	0.0382138671	0.0373342753	0.0382907246
1700	10	0.043296875	0.006679422358	0.0002112218811	0.0429494150	0.0436443349	0.0428828801	0.0437108698
1800	10	0.048796875	0.005546258702	0.0001753880999	0.0485083615	0.0490853884	0.0484531143	0.0491406356
1900	10	0.054046875	0.007914576513	0.000250280885	0.0536351629	0.0544585870	0.0535563244	0.0545374255
2000	10	0.060828125	0.0383228937	0.001211876306	0.0588345884	0.0628216615	0.0584528474	0.0632034025

Figure 5.3: 1000 Tests each - Bin Size 10

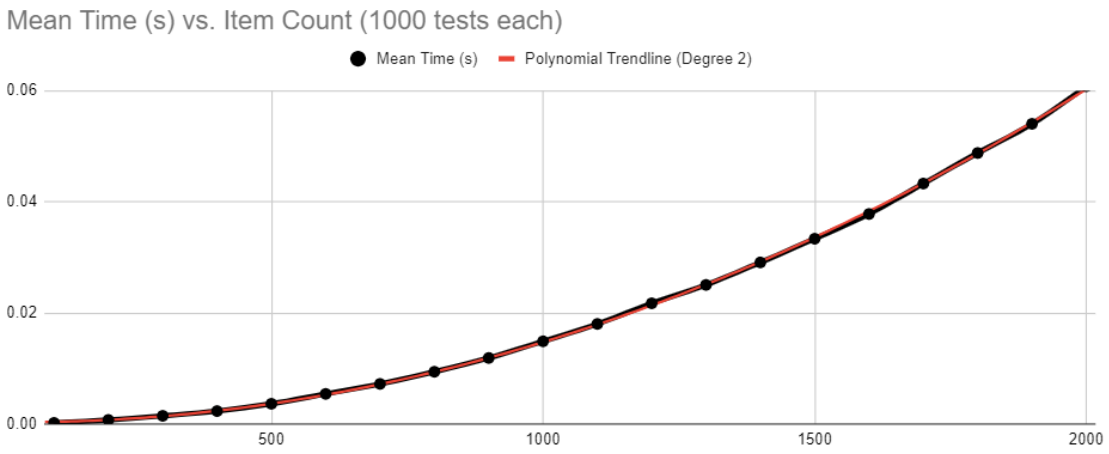


Figure 5.4: Time Complexity Plot - 1000 Tests each - Bin Size 10

itemCount	binSize	Mean Time (s)	Standard Deviatior	Standard Error	%90CL LB	%90CL UB	%95CL LB	%95CL UB
100	10	0.00015625	0.001554823361	2.20E-05	0.0001200788	0.0001924211	0.0001131524	0.0001993475
300	10	0.001378125	0.004431467273	6.27E-05	0.0012750321	0.0014812178	0.0012552909	0.0015009590
500	10	0.0036625	0.006619774961	9.36E-05	0.0035084987	0.0038165012	0.0034790092	0.0038459908
700	10	0.00700625	0.007771563332	0.000109906502	0.0068254538	0.0071870461	0.0067908332	0.0072216667
900	10	0.011728125	0.006932272239	9.80E-05	0.0115668539	0.0118893960	0.0115359722	0.0119202777
1100	10	0.01733125	0.005323944009	7.53E-05	0.0172073947	0.0174551052	0.0171836778	0.0174788222
1300	10	0.02424375	0.007777844972	0.000109955338	0.0240628076	0.0244246923	0.0240281591	0.0244593408
1500	10	0.03240625	0.008193506203	0.000115873676	0.0322156378	0.0325968622	0.0321791376	0.0326333624
1700	10	0.0417	0.01218377513	0.000172304600	0.0414165589	0.0419834410	0.0413622829	0.0420377170
1900	10	0.0521	0.007457925675	0.000105470996	0.0519265002	0.0522734997	0.0518932768	0.0523067231

Figure 5.5: 5000 Tests each - Bin Size 10

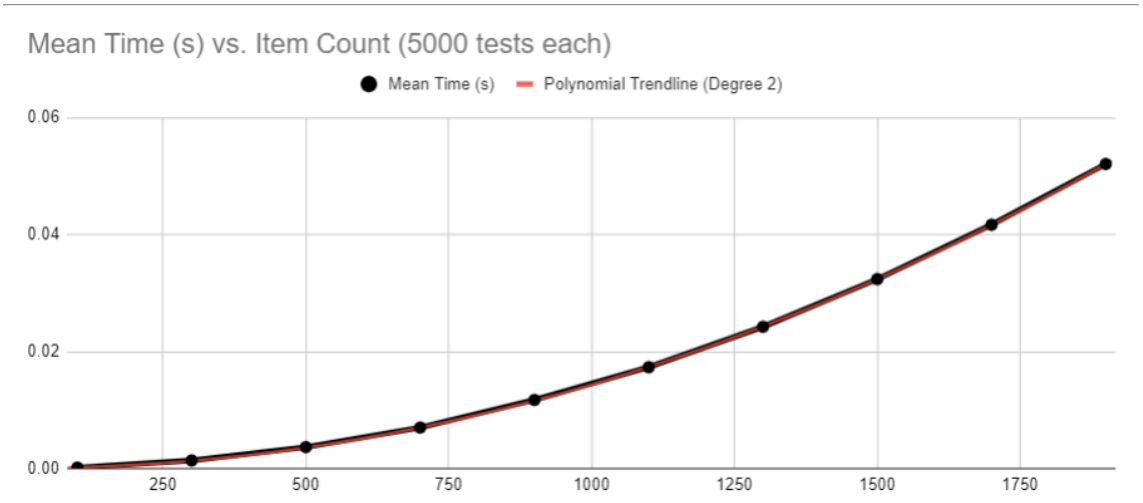


Figure 5.6: Time Complexity Plot - 5000 Tests each - Bin Size 10

5.2 Bin Size Indifference

Tests below are provided to observe the differences between different bin sizes with respect to time complexity plots.

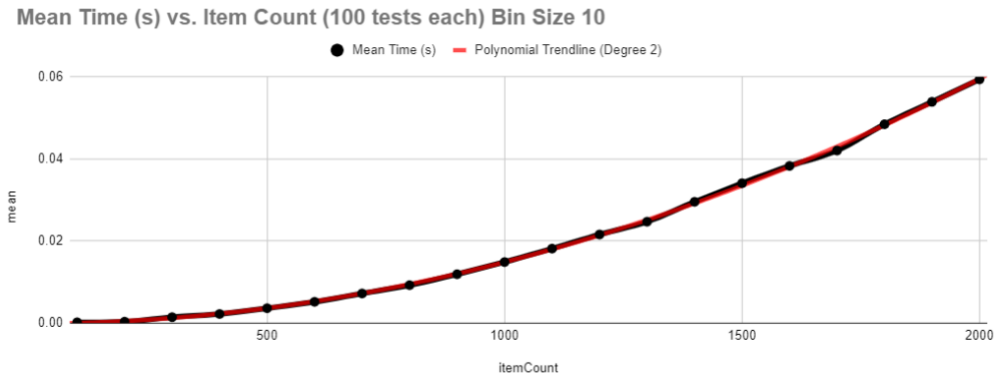


Figure 5.7: Time Complexity Plot - 100 Tests each - Bin Size 10

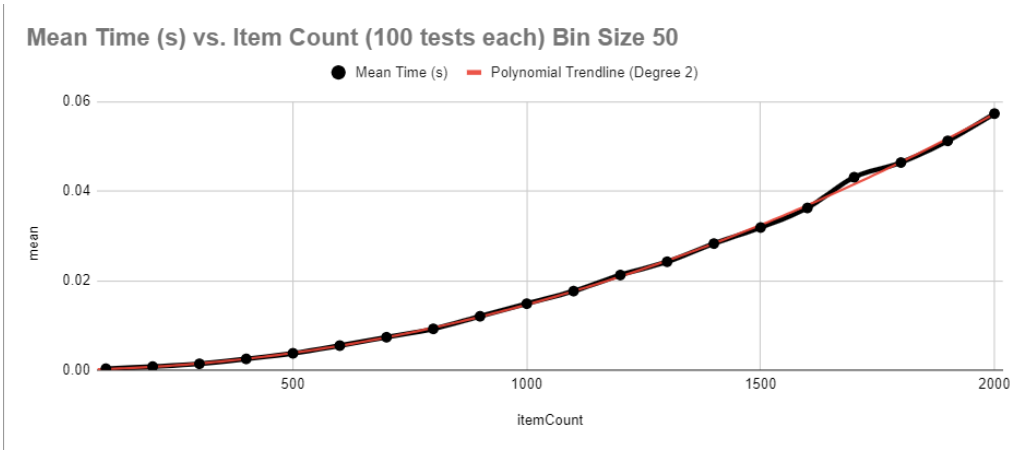


Figure 5.8: Time Complexity Plot - 100 Tests each - Bin Size 50

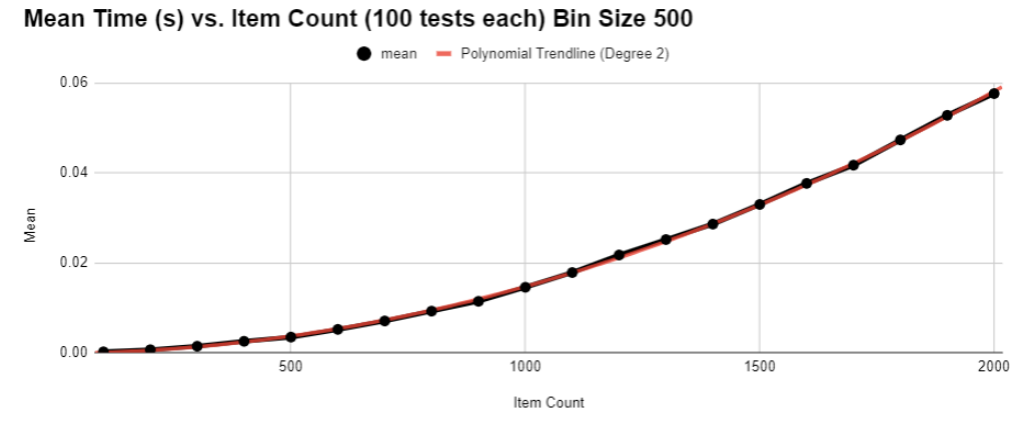


Figure 5.9: Time Complexity Plot - 100 Tests each - Bin Size 500

As can be seen in the plots, different bin sizes do not cause any difference in case of time complexity which supports our claims made in the algorithm analysis part.

5.3 Approximation Performance

In order to test approximation performance and compare it with the $1.7 \cdot \text{OPT}$ ratio bound given in the algorithm description part, a total of 400 tests were performed for 7, 8, 9, and 10 items with 100 iterations each. Tests were carried out by using the exact same test case for both greedy and brute force algorithms and calculating the result difference ratio.

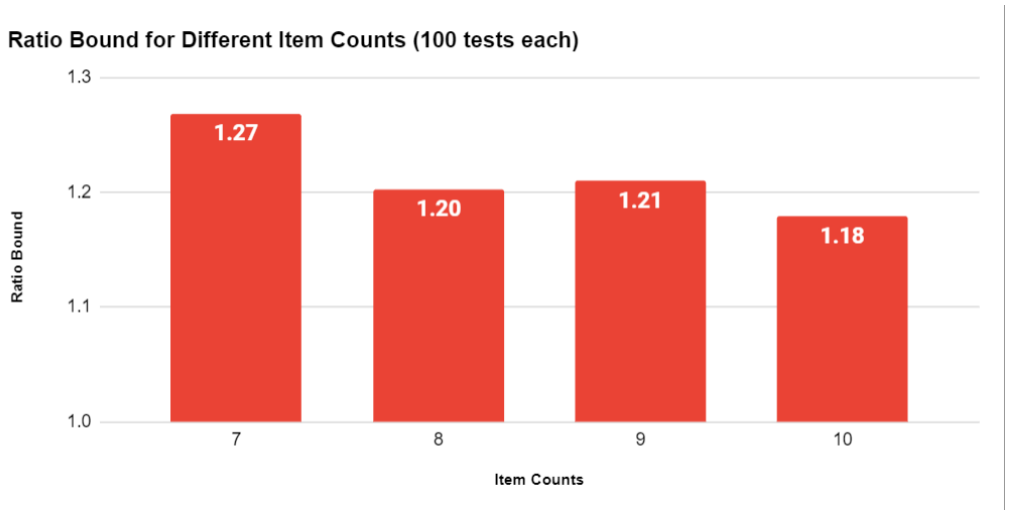


Figure 5.10: Ratio Bounds

Our results showed that the ratio bound is around $1.23 \cdot \text{OPT}$, which is in line with the proven ratio bound of $BF(L) \leq \lfloor 1.7 \cdot \text{OPT} \rfloor$. Regardless, one can say that the ratio bound of around $1.23 \cdot \text{OPT}$ observed in practice is better than the ratio bound of $1.7 \cdot \text{OPT}$.

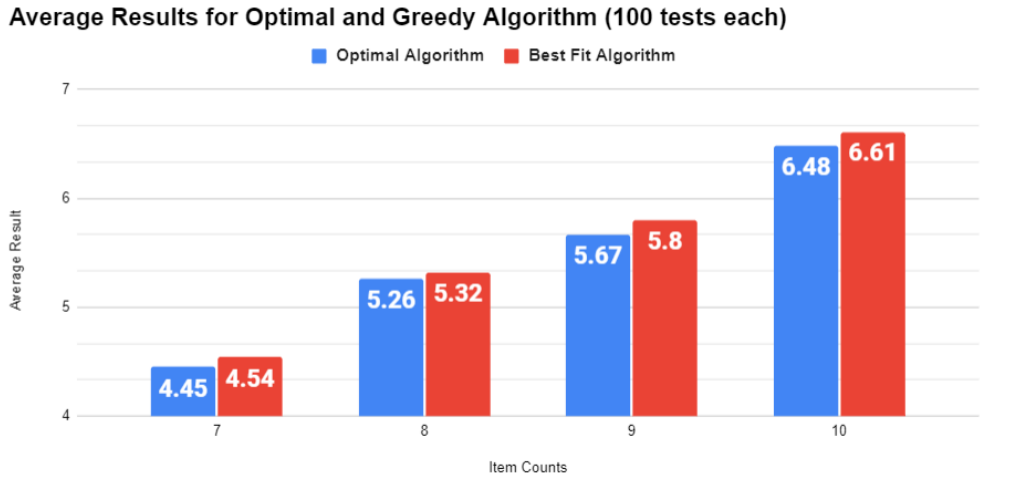


Figure 5.11: Average Results

6. Experimental Analysis of the Correctness

Functional testing has been carried out using the built-in unit testing framework in Python, know as unittest library [7]. Unittest library provides basic test suite in the form of a class and each test case are represented by a class method of the test suite.

6.1 Black Box Testing

Two test suites were created, one for the previous correct algorithm and one for the efficient algorithm.

```
1  import unittest
2  from bestFit import bestFit
3
4  class bestFitTest(unittest.TestCase):
5      def test_empty(self):
6          self.assertEqual(bestFit([0], 0, 0), 0)
7
8      def test_regularInputTest(self):
9          weights = [1,2,1]
10         result = bestFit(weights, len(weights), 3)
11         expected = 2
12         self.assertEqual(result, expected)
13
14     def test_zeroBinSize(self):
15         weights = [1,2,1]
16         result = bestFit(weights, len(weights), 0)
```

```
17          expected = 3
18          self.assertEqual(result, expected)
19
20      def test_oneBinSize(self):
21          weights = [1,2,1]
22          result = bestFit(weights, len(weights), 1)
23          expected = 3
24          self.assertEqual(result, expected)
```

Since exhaustive testing is assumed to be theoretically impossible, extreme cases and one regular case has been used; namely a case where the bins are empty and item sizes are zero, a case where item sizes are nonzero but the binsize is zero, a case where bin sizes are one but weights are one and finally a non-extreme input.

For the bruteForce approach, a similar test suite was created. Due to the implementation, bruteForce utilizes some global variables. These global variables create a shared state between tests and can result in unexpected behaviour (dependency between tests). In order to prevent this, a setUp and tearDown methods were also introduced to breakup any shared state between tests.

```
1  from bruteForce import bruteforce
2  import unittest
3  import bin as bin
4
5
6  class bruteForceTest(unittest.TestCase):
7      def setUp(self) -> None:
8          super().setUp()
9          bin.currentBestSolution = 0
10         bin.currentBestBins = []
11         bin.size = 9
12
13     def tearDown(self) -> None:
14         super().tearDown()
15         bin.currentBestSolution = 0
16         bin.currentBestBins = []
17         bin.size = 9
18
19     def test_one_bin_size(self):
20         items = [3, 5, 2, 5, 3, 2, 4, 3]
21         bruteforce(items, 1)
22         self.assertEqual(bin.currentBestSolution, 0)
23
24     def test_regular_input(self):
```

```
25          items = [3, 5, 2, 5, 3, 2, 4, 3]
26          bruteforce(items, 0)
27          self.assertEqual(bin.currentBestSolution, 0)
28
29      def test_zero_bin_size_input(self):
30          items = [3, 5, 2, 5, 3, 2, 4, 3]
31          bruteforce(items, 0)
32          bin.size = 0
33          self.assertEqual(bin.currentBestSolution, 0)
34
35      def test_empty_input(self):
36          items = []
37          bruteforce(items, 0)
38          self.assertEqual(bin.currentBestSolution, 0)
```

As expected, the test execution was successful which proves our algorithm is not only theoretically correct as proven in **Section 3.1.1**, but the implementation of the algorithm does not pose any impediment to its correctness.

6.2 White Box Testing

In this phase of functional testing, each component of the implementation were considered separately. Each component has a test suite corresponding to them which adds-up to be 4 test suites and 18 tests. Test suites other than the ones mentioned below can be seen below.

```
1  import unittest
2  from bin import Bin
3  from bin import getFilledBinsCount
4  from bin import bins
5
6  class binTest(unittest.TestCase):
7
8      def test_put_half(self):
9          bin = Bin(10)
10         self.assertTrue(bin.put(5))
11
12     def test_fill(self):
13         bin = Bin(10)
14         self.assertTrue(bin.put(5))
15
16     def test_put_too_much(self):
17         bin = Bin(10)
18         self.assertFalse(bin.put(11))
```

```
19
20      def test_remove(self):
21          bin = Bin(10)
22          bin.put(5)
23          bin.remove(5)
24          self.assertEqual(len(bin.contents), 0)
25
26      def test_getFilledBinsCount(self):
27          self.assertEqual(getFilledBinsCount(),0)
28
29      def test_getFilledBinsCountFull(self):
30          bins[0].put(9)
31          self.assertEqual(getFilledBinsCount(),1)
```

```
1   import unittest
2   from nextFit import nextfit
3
4   class nextFitTest(unittest.TestCase):
5
6       def test_empty(self):
7           self.assertEqual(nextfit([0], 0), (1, [[0]]))
8
9       def test_regularInputTest(self):
10          weights = [1,2,1]
11          result = nextfit(weights, 3)
12          expected = (2, [[1, 2], [1]])
13          self.assertEqual(result, expected)
14
15      def test_zeroBinSize(self):
16          weights = [1,2,1]
17          result = nextfit(weights, 0)
18          expected = (4, [[], [1], [2], [1]])
19          self.assertEqual(result, expected)
20
21      def test_oneBinSize(self):
22          weights = [1,2,1]
23          result = nextfit(weights, 1)
24          expected = (3, [[1], [2], [1]])
25          self.assertEqual(result, expected)
```

Each test suite were ran using a coverage tool in python known as Coverage.py [3] This tool enables us to see percentage of statement coverage and percentage of tests executed. The test suites were executed using command coverage run -m unittest discover coverage report -m and the logged output can be seen below.

.....

Ran 18 tests **in** 16.413s

OK
Name Stmts Miss Cover Missing
----
bestFit.py 16 0 100%
bin.py 22 0 100%
bruteForce.py 16 0 100%
nextFit.py 18 0 100%
test_bestFit.py 20 0 100%
test_bin.py 24 0 100%
test_bruteForce.py 31 0 100%
test_nextFit.py 20 0 100%

TOTAL 167 0 100%

As it can be seen through the logged information, execution of the test suites were successful ("OK" indicates that no test has failed) and for each test suite 100% statement coverage was achieved.

7. Discussion

To discuss our results; firstly, in **Chapter 6**, we have established that the implementations of our brute force algorithm and the Best-Fit algorithm works correctly by doing functional testing with Black Box Testing and White Box Testing techniques, and showed that there are no defects in implementation of the both algorithms. Moreover, we had already shown in **Chapter 3** that both algorithms are correct in terms of the actual steps taken to solve the actual problem.

Secondly, in **Chapter 5**, we have shown that in terms of time complexity, the performance of the Best-Fit algorithm in practice is identical to our theoretical, worst-case analysis in **Chapter 3**. In other words, the worst-case time complexity of the Best-Fit algorithm, $\theta(n^2)$, is what was observed in our experiments. In addition, as stated in **Chapter 5**, when we tested the approximation performance of the Best-Fit algorithm in terms of approximation and optimal values, we have found that the observed ratio bound in practice is consistent with the theoretical ratio bound proven in **Section 2.2.2**, that is, the ratio bound in practice is $\leq \lfloor 1.7 \cdot \mathrm{OPT} \rfloor$. Moreover, the observed ratio bound in practice was around $1.23 \cdot \mathrm{OPT}$, which one can say that it is better than the ratio bound of $1.7 \cdot \mathrm{OPT}$.

8. Performance Testing Code

8.1 Performance Testing With Different Item Counts

The following code was used to do the initial performance testing. **testCount** variable was the independent variable in our tests, i.e. it was the main variable that was altered between visualizations.

```
import sys
import itertools
import random
import time
import statistics
import math
import csv

#GENERATING TEST CASES
testCount = 100
binSize = 10
batchResults = {}

for itemCount in range(100, 2100, 100):
    results = []

    for _ in range(testCount):
        start = time.process_time()
        items = []
```

```
20          for x in range(itemCount):
21              items.append(random.randint(1,binSize))
22
23          #SOLVING PHASE
24          n = bestFit(items, len(items), binSize)
25          end = time.process_time()
26          results.append(end-start)
27
28      batchResults[itemCount] = results
29      #print(f"Results:\n{results}")
30
31  header = ['itemCount', 'binSize', 'mean', 'sd','error','90clLower','90clUpper','95clLower'
32
33  with open('output.csv', 'w', encoding='UTF8') as f:
34      writer = csv.writer(f)
35
36      # write the header
37      writer.writerow(header)
38
39      # write the data
40      for itemCount,results in batchResults.items():
41          N = len(results)
42          totalTime = sum(results)
43
44          sd = statistics.stdev(results)
45
46          m = totalTime / N
47
48          tval90 = 1.645
49          tval95 = 1.96
50
51          sm = sd / math.sqrt(N)
52
53          upperMean90 = m + tval90 * sm
54          lowerMean90 = m - tval90 * sm
55
56          upperMean95 = m + tval95 * sm
57          lowerMean95 = m - tval95 * sm
58          data = [itemCount,binSize,m,sd,sm,lowerMean90,upperMean90,lowerMean95,upperMean95]
59          writer.writerow(data)
```

8.2 Approximation Performance

The following code was used for testing the approximation performance.

```
import sys
import itertools
import random
import time
import statistics
import math
import csv

#GENERATING TEST CASES
testCount = 100
binSize = 10
batchResults = {}

for itemCount in range(7,11):
    print(f"Testing for {itemCount} items, bin size is {binSize}.")
    results = []

    for _ in range(testCount):
        print(".",end='')
        items = []
        for x in range(itemCount):
            items.append(random.randint(1,binSize))

        permutations = list(itertools.permutations(items))
        min = sys.maxsize # corresponds to INT_MAX
        minArr = []
        minPermutation = []

        for elem in permutations:
            res,subArr = nextfit(elem,binSize)
            if res < min:
                min = res
                minArr = subArr
                minPermutation = elem

        bfres = bestFit(items,len(items),binSize)
        results.append([bfres,min])
    print()
```

```
39      batchResults[itemCount] = results
40      #print(f"Results:\n{results}")
41
42  header = ['itemCount', 'binSize', 'optResult','bfResult','diff']
43
44  with open('output.csv', 'w',newline='') as f:
45      writer = csv.writer(f)
46
47      # write the header
48      writer.writerow(header)
49
50      # write the data
51      for itemCount,results in batchResults.items():
52          for test in results:
53              data = [itemCount,binSize,test[1],test[0],test[0] - test[1]]
54              writer.writerow(data)
```

Bibliography

[1] *Basic Analysis of Bin-Packing Heuristics*. URL: https://bastian.rieck.me/research/Note_BP.pdf.

[2] *Bin Packing Problem (Minimize number of used Bins)*. URL: https://www.geeksforgeeks.org/bin-packing-problem-minimize-number-of-used-bins/.

[3] *coverage.py*. URL: https://coverage.readthedocs.io.

[4] *Dósa, G., Sgall, J. (2014). Optimal Analysis of Best Fit Bin Packing. Lecture Notes in Computer Science, 429–441.* URL: https://link.springer.com/chapter/10.1007/978-3-662-43948-7_36.

[5] *Functions creating iterators for efficient looping*. URL: https://docs.python.org/3/library/itertools.html#itertools.permutationsf.

[6] *Kozen, D., 1992. The design and analysis of algorithms. New York: Springer-Verlag.* URL: https://link.springer.com/book/10.1007/978-1-4612-4400-4.

[7] *unittest - unit testing framework - python 3.10.1 documentation*. URL: https://docs.python.org/3/library/unittest.html.