CS301

2022-2023 Spring

Project Report

Group 158

::Group Members::

Berfin Sürücü 26530

Serhan Yorulmaz 26481

1. Problem Description

The bin packing problem, to put it intuitively, is the problem of placing the items in the most optimised and best way, not exceeding the capacity of the box, by leaving the least empty space inside a box and using the least number of boxes. Input is the bin's capacity and the item array list. Since the main objective (objective) is to achieve the minimum number of bin to put items, output does not give us the set of items of the partition, it returns the minimum number of bin.

Given n elements in a placed finite set U; for each item i in U, a positive integer size S(i), a positive integer C (called the bin capacity), what is min. number of k (of bins) such that elements in U could be partitioned into k disjoint sets Uj,...,Uk where for each Uj, (1 <= i <= k) the total sum of the sizes of the items in Uj does not exceed capacity C?

Bin packing is a problem that we can face in many real life problems. Since it is a problem that aims to allocate resources like space and time optimally, it can be used in industries like shipping, logistics, storing, manufacturing etc. More specifically, it can be a problem of fitting items in different shapes into containers, ships, trucks etc. Moreover, it is used in cutting stock that aims to minimise the waste by cutting raw materials into shapes and using leftover materials. Also, it is used in time management like scheduling events like meetings, sport events etc.

A problem could be shown to be NP-Hard if it may be reduced to another problem that has previously been demonstrated to be NP-Hard. By reducing The Partition Problem to it, the team will demonstrate that the Bin Packing problem is NP-Hard, and we will apply the Sub Reduction Problem to demonstrate this.

Dexter C. Kozen offered this method to demonstrate the Bin packing's NP-Hard property in his book "The Design and Application of Algorithms"[6]. The proof will be written more clearly.

The Sub Reduction Problem says that, given finite set S and positive integer weight subset there exists a subset which could be reduced into half. Other words, When we solve the subset sum problem, we are capable of finding subsets of any size, therefore if we can do that, we should be able to find a subset that splits (partitions) the set exactly in half.

Putting into this partition problem with bin packing problem, for example when B is the 4 (bin size) and bin count is 2 that is half of the bin size, the result will be 2 subsets which have put 2 bins inside of each other.

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

2. Algorithm Description

a. Brute Force Algorithm

At first, we started to study the algorithms in the resources, but of course we could not find the exact solution. For this, we decided to determine the objective at first and choose the algorithm accordingly. The aim is achieving a minimum number of bins with parameters of size of bin and item list. Therefore, in this part we're approaching with brute force to solve this bin packing problem. The approach consists of two steps.

2.1. Generating Permutations

The given list of items, the permutation algorithm derives all the possible permutations of the item list. The team used intertools[5] Python module to generate all the possible permutations. The permutations function takes **Θ**(n!) time.

```
1   def permutations(iterable, r=None):
2       # permutations('ABCD', 2) --> AB AC AD BA BC BD CA CB CD DA DB DC
3       # permutations(range(3)) --> 012 021 102 120 201 210
4       pool = tuple(iterable)
5       n = len(pool)
6       r = n if r is None else r
7       if r > n:
8           return
9       indices = range(n)
10      cycles = range(n, n-r, -1)
11      yield tuple(pool[i] for i in indices[:r])
12      while n:
13          for i in reversed(range(r)):
14              cycles[i] -= 1
15              if cycles[i] == 0:
16                  indices[i:] = indices[i+1:] + indices[i:i+1]
17                  cycles[i] = n - i
18              else:
19                  j = cycles[i]
20                  indices[i], indices[-j] = indices[-j], indices[i]
21                  yield tuple(pool[i] for i in indices[:r])
22                  break
23          else:
24              return
```

For example: Given an item list L = [‘x’,’y’, ‘z’] the permutations will be

(‘x’, ‘y’, ‘z’)

(‘x’, ‘z’, ‘y’)

(‘z’, ‘y’, ‘x’)

(‘z’, ‘x’, ‘y’)

(‘y’, ‘x’, ‘z’)

(‘y’, ‘z’, ‘x’)

2.2 Next-Fit Solver Function

The following pseudo code will be used for Next-Fit algorithm;

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
```

The time complexity of the Next-Fit Function is **Θ**(n). The algorithm consists of one for loop and packing and the object takes constant time.

```
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
```

The function of next-fit takes two parameters named as weight and c which is bin size. This function also returns two parameters; one is the list of item placements and the second one is the algorithm's result, which is the number of bins used for packing.

2.3 Solution Phase

The pseudocode of the algorithm is as follows;

Create item list's all possible permutations

For every element of the permutation list:

Do Next-Fit function

If resulting number of bins is smaller than the current minimum

Update the current minimum

Print minimum

Finish execution

In the code part, 'items' is the given item list, 'binSize' is carrying all the possible permutations. It has been mentioned that on the pseudocode part, every element in the permutations list 'binSize', the Next-Fit function will be applied.
If the returned result by Next-Fit is smaller than the current minimum, it will be updated with the returned result.

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

The running time complexity of the algorithm is;

$$T(n) = T_{permutations}(n) + n! * T_{next\text{-}fit}(n, binSize)$$
$$= T(n) = n! + n! * (n)$$
$$= n!(n+1)$$
$$= \Theta(n!)$$

b. Heuristic Algorithm

The Best-Fit heuristic algorithm is the one we've selected. To fit the following thing into the smallest area feasible is the aim of best-fit. That is, put it in the trash can with the smallest amount of empty space. The following is the Best-Fit algorithm's pseudocode:

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 that may be used for heuristic techniques, however even if it doesn't always provide the best result, it is much quicker than the brute force approach. The implementation of the Best-Fit algorithm is shown below.

```
def bestFit(weight, n, c):
    # Initialize result (Count of bins)
    res = 0

    # Create an array to store 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
```

Ratio Bound

We found that the ratio bound of the Best-Fit method is 1.7 OPT (optimal). Two lemmas and one theory can be used to give a short proof of the ratio bound of the Best-Fit algorithm. The proof comes from the 2014 paper Optimal Analysis of Best Fit Bin Packing by Dósa, G., and Sgall, J.:

Lemma 1 At any time, the following can be found in the BF packing:

1. The total of any two bins' numbers is more than 1. Specifically, there is no more than one bin with a level of no more than 1/2.

2. Any item a with a level of no more than 1/2 (that is, packed into the single bin with a level of no more than 1/2) does not fit into any open bin when it arrives, except the bin where it is packed.

3. If there are two bins B and B with a level of no more than 2/3, in this order, then either B only has one item or the first item in B is huge.

As an example of the lemma, we will now give a short proof that the asymptotic ratio for Best-Fit is 1.7. It uses the same weight function as the standard Best-Fit analysis. (In some versions, an item's weight can be no more than 1, which doesn't change the study much.)

As a result of Lemma 1, the ratio bound of the Best-Fit algorithm is 1.7 OPT.[4]

3. Algorithm Analysis

a. Brute Force Algorithm

3.a.1 Correctness of Algorithm

Correctness of this algorithm can be proved by using mathematical induction. To define the algorithm, assume that “n” is the number of items and the capacity of a bin is “m”.

Since k cannot be equal to 0, n=1 for our base case. In this situation, the first item n1 must be located in the first bin for all possible scenarios. So, the base case is true.

Next, the induction hypothesis suggests that our algorithm’s all possible outputs contain all possible outputs for the first n items.

For the (n+1)th item, since all correct placements for n items will be contained in the all possible correct solutions. Thus, this (k+1)th item can be placed to the appropriate position in any of those possible correct solution sets. So, after the placement of (n+1)th item, output solutions of our algorithm will contain all possible correct permutations in n +1 items.

Thus, mathematical induction for this algorithm is valid and theoretically this algorithm is correct.

3.a.2 Worst-Case Time Complexity

The running time of taking all the permutations of a list is **Θ**(n!). In addition, Next-fit iterates all the elements in one for-loop therefore time complexity of Next-Fit function is **Θ**(n). Thus, the worst-case time complexity of the brute-force algorithm will be **Θ**(n^2).

3.a.3 Space Complexity

The complexity of space will be determined by all of the permutations of a list which is **Θ**(n!). In the worst case, each item will be taken into separate bins. Thus, for n element, the bin size will be n also. The worst-case space complexity of the brute-force algorithm will be **Θ**(n^2).

b. Heuristic Algorithm

3.b.1 Correctness of Algorithm

The Best-Fit algorithm always offers a solution, however the offered solution may not be the best one, as was discussed in the section on ratio limits. This is because the Best-Fit algorithm is greedy, trying to arrive at a globally optimum solution by selecting a locally optimal one. However, when compared to algorithms that use a brute force method, the Best-Fit algorithm is substantially quicker. Furthermore, it should be noted that it has been shown that the Best-Fit algorithm cannot produce more than 1.7 OPT (optimal) number of bins.

3.b.2 Worst-Time Complexity

Because there can only be a maximum of n bins in the worst-case scenario, the Best-Fit algorithm first generates an array to contain n number of bins. Each time the outer for loop iterates, or for each item in the items list, all the bins are checked in the inner for loop to locate the bin with the least amount of space left over after the item is added, ensuring the tightest/best fit. The worst-case time complexity of the Best-Fit method is thus (n), where n is the number of items to iterate through and (n) is the number of bins to iterate over.

$$T(n) = (n \text{ items}) * (n \text{ bins})$$
$$T(n) = \Theta(n*n)$$
$$T(n) = \Theta(n^2)$$

3.b.3 Space Complexity

Since there can only be a maximum of n bins in the worst-case scenario, the Best-Fit algorithm's worst-case space complexity is as follows:

$$S(n) = \Theta(n)$$

4. Sample Generation (Random Instance Generator)

```python
import random

def generate_instances(num_tests, item_num_range, bin_capacity_range):

    instances = []
    for i in range(num_tests):
        bin_capacity = random.randint(bin_capacity_range[0], bin_capacity_range[1])
        max_item_size = bin_capacity
        num_items = random.randint(item_num_range[0], item_num_range[1])
        items = [random.randint(1, max_item_size) for j in range(num_items)]
        instances.append((bin_capacity, items))

    return instances

# Example usage:
instances = generate_instances(1, (5, 10), (4, 15))
for bin_capacity, items in instances:
    print("Items:", items,)
    print("Bin capacity:", bin_capacity,"\n")
```

The function above creates random samples that are suitable to use for bin packing algorithms. Firstly, the function decides bin capacity according to the range indicated in the parameter. After that, it sets the maximum item size to the bin capacity. Then, it generates an "item_num_range" number of items between the size value indicated in the third parameter of the function. Last but not least, the function runs this operation "num_tests" times. Since the "random" library is used, all those integer values generated are completely random. Border values of variables as well as the number of tests are not hard coded to run the function more flexibly and make the function reusable. Instead, they are given as parameters to the function called "generate_instances". There are 3 parameters of that function which are indicated below:

num_tests (int): number of test cases to generate
item_num_range (tuple): range of item numbers to generate, in the form of (min_num, max_num)(Notice that, it is a range)
bin_capacity_range (tuple): range of bin capacities to generate, in the form of (min_capacity, max_capacity)(Notice that, it is a range)

Last “for loop” prints a list of items generated by the function as well as the bin capacity for that instance. Notice that item sizes cannot exceed bin capacity. Any number of samples can be generated with this function.

Following image shows sample output that takes parameters from the first image.

```
Items: [8, 9, 2, 13, 4, 2, 9, 1, 10]
Bin capacity: 13
```

5. Algorithm Implementations

a. Brute Force Algorithm

The complete code for the brute force algorithm is provided below. This image includes sample generation as well. It is tested for 20 different samples. Outputs of those samples are listed below.

```python
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
def generate_instances(num_tests, item_num_range, bin_capacity_range):
    instances = []
    for i in range(num_tests):
        bin_capacity = random.randint(bin_capacity_range[0], bin_capacity_range[1])
        max_item_size = bin_capacity
        num_items = random.randint(item_num_range[0], item_num_range[1])
        items = [random.randint(1, max_item_size) for j in range(num_items)]
        instances.append((bin_capacity, items))

    return instances

instances = generate_instances(1, (5, 10), (4, 15))
for bin_capacity, items in instances:
    print("Sizes of items:", items)
    print("Number of items:", len(items))
    print("Bin capacity:", bin_capacity)

#SOLVING THE PROBLEM
permutations = list(itertools.permutations(items))
min = sys.maxsize
minArr = []
minPermutation = []

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

print(f"Permutation: {minPermutation}")
print(f"Minimum number of bins required(desired output): {min}")
print(f"Distribution of items into bins: {minArr}")
```

TEST CASE 1

Sizes of items: [3, 6, 8, 8, 3, 5, 2, 4, 10, 3]

Number of items: 10

Bin capacity: 10

Permutation: (3, 6, 8, 8, 3, 5, 2, 4, 3, 10)

Minimum number of bins required(desired output): 6

Distribution of items into bins: [[3, 6], [8], [8], [3, 5, 2], [4, 3], [10]]

TEST CASE 2

Sizes of items: [6, 5, 3, 5, 7, 4, 5]

Number of items: 7

Bin capacity: 7

Permutation: (6, 5, 3, 4, 5, 7, 5)

Minimum number of bins required(desired output): 6

Distribution of items into bins: [[6], [5], [3, 4], [5], [7], [5]]

TEST CASE 3

Sizes of items: [13, 15, 10, 3, 9, 1, 15, 1, 8]

Number of items: 9

Bin capacity: 15

Permutation: (13, 15, 10, 3, 9, 1, 15, 1, 8)

Minimum number of bins required(desired output): 6

Distribution of items into bins: [[13], [15], [10, 3], [9, 1], [15], [1, 8]]

TEST CASE 4

Sizes of items: [10, 4, 3, 1, 3, 5, 3, 10]

Number of items: 8

Bin capacity: 10

Permutation: (10, 4, 3, 3, 1, 5, 3, 10)

Minimum number of bins required(desired output): 4

Distribution of items into bins: [[10], [4, 3, 3], [1, 5, 3], [10]]

TEST CASE 5

Sizes of items: [5, 2, 7, 5, 8, 8]

Number of items: 6

Bin capacity: 8

Permutation: (5, 2, 7, 5, 8, 8)

Minimum number of bins required(desired output): 5

Distribution of items into bins: [[5, 2], [7], [5], [8], [8]]

TEST CASE 6

Sizes of items: [12, 9, 11, 12, 5, 6, 4, 7, 6]

Number of items: 9

Bin capacity: 14

Permutation: (12, 9, 5, 11, 12, 6, 4, 7, 6)

Minimum number of bins required(desired output): 6

Distribution of items into bins: [[12], [9, 5], [11], [12], [6, 4], [7, 6]]

TEST CASE 7

Sizes of items: [2, 5, 7, 6, 5]

Number of items: 5

Bin capacity: 11

Permutation: (2, 5, 7, 6, 5)

Minimum number of bins required(desired output): 3

Distribution of items into bins: [[2, 5], [7], [6, 5]]

TEST CASE 8

Sizes of items: [6, 3, 8, 5, 6, 6]

Number of items: 6

Bin capacity: 8

Permutation: (6, 3, 5, 8, 6, 6)

Minimum number of bins required(desired output): 5

Distribution of items into bins: [[6], [3, 5], [8], [6], [6]]

TEST CASE 9

Sizes of items: [10, 7, 3, 10, 9, 4, 6, 2, 8, 9]

Number of items: 10

Bin capacity: 11

Permutation: (10, 7, 3, 10, 9, 4, 6, 2, 8, 9)

Minimum number of bins required(desired output): 7

Distribution of items into bins: [[10], [7, 3], [10], [9], [4, 6], [2, 8], [9]]

TEST CASE 10

Sizes of items: [15, 14, 12, 12, 8, 7]

Number of items: 6

Bin capacity: 15

Permutation: (15, 14, 12, 12, 8, 7)

Minimum number of bins required(desired output): 5

Distribution of items into bins: [[15], [14], [12], [12], [8, 7]]

TEST CASE 11

Sizes of items: [3, 7, 6, 5, 5, 4]

Number of items: 6

Bin capacity: 8

Permutation: (3, 5, 7, 6, 5, 4)

Minimum number of bins required(desired output): 5

Distribution of items into bins: [[3, 5], [7], [6], [5], [4]]

TEST CASE 12

Sizes of items: [2, 5, 6, 6, 1, 6, 3, 3, 4, 8]

Number of items: 10

Bin capacity: 8

Permutation: (2, 6, 5, 3, 6, 1, 6, 3, 4, 8)

Minimum number of bins required(desired output): 6

Distribution of items into bins: [[2, 6], [5, 3], [6, 1], [6], [3, 4], [8]]

TEST CASE 13

Sizes of items: [4, 1, 3, 6, 6, 7]

Number of items: 6

Bin capacity: 7

Permutation: (4, 3, 1, 6, 6, 7)

Minimum number of bins required(desired output): 4

Distribution of items into bins: [[4, 3], [1, 6], [6], [7]]

TEST CASE 14

Sizes of items: [7, 2, 4, 5, 10, 9, 2, 4]

Number of items: 8

Bin capacity: 10

Permutation: (7, 2, 4, 5, 10, 9, 2, 4)

Minimum number of bins required(desired output): 5

Distribution of items into bins: [[7, 2], [4, 5], [10], [9], [2, 4]]

TEST CASE 15

Sizes of items: [2, 3, 1, 5, 2, 1, 1, 1]

Number of items: 8

Bin capacity: 7

Permutation: (2, 3, 1, 5, 2, 1, 1, 1)

Minimum number of bins required(desired output): 3

Distribution of items into bins: [[2, 3, 1], [5, 2], [1, 1, 1]]

TEST CASE 16

Sizes of items: [7, 1, 1, 3, 14, 9, 5, 2]

Number of items: 8

Bin capacity: 15

Permutation: (7, 1, 1, 3, 2, 14, 9, 5)

Minimum number of bins required(desired output): 3

Distribution of items into bins: [[7, 1, 1, 3, 2], [14], [9, 5]]

TEST CASE 17

Sizes of items: [2, 5, 5, 2, 1, 5, 5, 1, 3]

Number of items: 9

Bin capacity: 5

Permutation: (2, 2, 5, 5, 1, 1, 3, 5, 5)

Minimum number of bins required(desired output): 6

Distribution of items into bins: [[2, 2], [5], [5], [1, 1, 3], [5], [5]]

TEST CASE 18

Sizes of items: [4, 2, 5, 5, 7, 5]

Number of items: 6

Bin capacity: 12

Permutation: (4, 2, 5, 5, 7, 5)

Minimum number of bins required(desired output): 3

Distribution of items into bins: [[4, 2, 5], [5, 7], [5]]

TEST CASE 19

Sizes of items: [7, 2, 6, 11, 3, 6]

Number of items: 6

Bin capacity: 14

Permutation: (7, 2, 6, 6, 11, 3)

Minimum number of bins required(desired output): 3

Distribution of items into bins: [[7, 2], [6, 6], [11, 3]]

TEST CASE 20

Sizes of items: [1, 1, 3, 4, 2]

Number of items: 5

Bin capacity: 5

Permutation: (1, 1, 3, 4, 2)

Minimum number of bins required(desired output): 3

Distribution of items into bins: [[1, 1, 3], [4], [2]]

b. Heuristic Algorithm

The complete code for the heuristic algorithm is provided below. As mentioned earlier, a best-fit heuristic approach is used. This image includes sample generation as well. It is tested for 20 different samples. Outputs of those samples are listed below.

```python
import random

# Best Fit algorithm for bin packing
def best_fit(item_sizes, bin_capacity):
    bins = []  # List to store the bins and their current occupancy
    for size in item_sizes:
        # Find the bin with the smallest remaining capacity that can accommodate the current item
        best_bin_index = -1
        best_remaining_capacity = bin_capacity + 1  # Initialize with a value greater than the bin capacity

        for i in range(len(bins)):
            remaining_capacity = bin_capacity - sum(bins[i])
            if size <= remaining_capacity and remaining_capacity < best_remaining_capacity:
                best_bin_index = i
                best_remaining_capacity = remaining_capacity

        if best_bin_index == -1:
            # Create a new bin if no existing bin can accommodate the current item
            bins.append([size])
        else:
            # Add the item to the best bin found
            bins[best_bin_index].append(size)

    return bins

# Generating test cases
def generate_instances(num_tests, item_num_range, bin_capacity_range):
    instances = []
    for _ in range(num_tests):
        bin_capacity = random.randint(bin_capacity_range[0], bin_capacity_range[1])
        max_item_size = bin_capacity
        num_items = random.randint(item_num_range[0], item_num_range[1])
        items = [random.randint(1, max_item_size) for _ in range(num_items)]
        instances.append((bin_capacity, items))
    return instances

# Generate a single test case
instances = generate_instances(1, (5, 10), (4, 15))
for bin_capacity, items in instances:
    print("Sizes of items:", items)
    print("Number of items:", len(items))
    print("Bin capacity:", bin_capacity)
    bins = best_fit(items, bin_capacity)
    print("Minimum number of bins required (desired output):", len(bins))
    for i, bin_contents in enumerate(bins):
        print("Bin", i+1, ":", bin_contents)
```

TEST CASE 1

Sizes of items: [6, 9, 11, 7, 2]

Number of items: 5

Bin capacity: 12

Minimum number of bins required (desired output): 4

Bin 1 : [6]

Bin 2 : [9, 2]

Bin 3 : [11]

Bin 4 : [7]

TEST CASE 2

Sizes of items: [3, 3, 1, 10, 5, 3, 7, 3, 7, 11]

Number of items: 10

Bin capacity: 11

Minimum number of bins required (desired output): 6

Bin 1 : [3, 3, 1, 3]

Bin 2 : [10]

Bin 3 : [5]

Bin 4 : [7, 3]

Bin 5 : [7]

Bin 6 : [11]

TEST CASE 3

Sizes of items: [4, 2, 6, 5, 7, 2, 8]

Number of items: 7

Bin capacity: 9

Minimum number of bins required (desired output): 5

Bin 1 : [4, 2]

Bin 2 : [6]

Bin 3 : [5]

Bin 4 : [7, 2]

Bin 5 : [8]

TEST CASE 4

Sizes of items: [14, 1, 6, 7, 13, 8, 12, 14, 5, 4]

Number of items: 10

Bin capacity: 14

Minimum number of bins required (desired output): 7

Bin 1 : [14]

Bin 2 : [1, 6, 7]

Bin 3 : [13]

Bin 4 : [8, 5]

Bin 5 : [12]

Bin 6 : [14]

Bin 7 : [4]

TEST CASE 5

Sizes of items: [5, 13, 4, 6, 5, 9, 5, 10, 1, 6]

Number of items: 10

Bin capacity: 13

Minimum number of bins required (desired output): 6

Bin 1 : [5, 4]

Bin 2 : [13]

Bin 3 : [6, 5, 1]

Bin 4 : [9]

Bin 5 : [5, 6]

Bin 6 : [10]

TEST CASE 6

Sizes of items: [3, 6, 7, 7, 3, 7]

Number of items: 6

Bin capacity: 11

Minimum number of bins required (desired output): 4

Bin 1 : [3, 6]

Bin 2 : [7, 3]

Bin 3 : [7]

Bin 4 : [7]

TEST CASE 7

Sizes of items: [13, 1, 2, 15, 5, 4, 15, 5]

Number of items: 8

Bin capacity: 15

Minimum number of bins required (desired output): 5

Bin 1 : [13, 1]

Bin 2 : [2, 5, 4]

Bin 3 : [15]

Bin 4 : [15]

Bin 5 : [5]

TEST CASE 8

Sizes of items: [1, 5, 4, 2, 6, 4, 7, 7, 2, 3]

Number of items: 10

Bin capacity: 7

Minimum number of bins required (desired output): 7

Bin 1 : [1, 5]

Bin 2 : [4, 2]

Bin 3 : [6]

Bin 4 : [4, 2]

Bin 5 : [7]

Bin 6 : [7]

Bin 7 : [3]

TEST CASE 9

Sizes of items: [2, 9, 7, 11, 9, 3, 3]

Number of items: 7

Bin capacity: 14

Minimum number of bins required (desired output): 4

Bin 1 : [2, 9, 3]

Bin 2 : [7]

Bin 3 : [11, 3]

Bin 4 : [9]

TEST CASE 10

Sizes of items: [3, 3, 3, 2, 5, 2, 2]

Number of items: 7

Bin capacity: 6

Minimum number of bins required (desired output): 4

Bin 1 : [3, 3]

Bin 2 : [3, 2]

Bin 3 : [5]

Bin 4 : [2, 2]

TEST CASE 11

Sizes of items: [2, 3, 1, 3, 5, 1, 2, 6, 5]

Number of items: 9

Bin capacity: 6

Minimum number of bins required (desired output): 5

Bin 1 : [2, 3, 1]

Bin 2 : [3, 2]

Bin 3 : [5, 1]

Bin 4 : [6]

Bin 5 : [5]

TEST CASE 12

Sizes of items: [4, 1, 1, 4, 5]

Number of items: 5

Bin capacity: 5

Minimum number of bins required (desired output): 3

Bin 1 : [4, 1]

Bin 2 : [1, 4]

Bin 3 : [5]

TEST CASE 13

Sizes of items: [6, 4, 4, 11, 5, 12, 4, 8, 8]

Number of items: 9

Bin capacity: 12

Minimum number of bins required (desired output): 6

Bin 1 : [6, 4]

Bin 2 : [4, 5]

Bin 3 : [11]

Bin 4 : [12]

Bin 5 : [4, 8]

Bin 6 : [8]

TEST CASE 14

Sizes of items: [7, 14, 7, 4, 6, 2, 3]

Number of items: 7

Bin capacity: 14

Minimum number of bins required (desired output): 4

Bin 1 : [7, 7]

Bin 2 : [14]

Bin 3 : [4, 6, 2]

Bin 4 : [3]

TEST CASE 15

Sizes of items: [9, 4, 9, 12, 8]

Number of items: 5

Bin capacity: 12

Minimum number of bins required (desired output): 4

Bin 1 : [9]

Bin 2 : [4, 8]

Bin 3 : [9]

Bin 4 : [12]

TEST CASE 16

Sizes of items: [12, 10, 3, 15, 12, 11, 10, 6, 8, 11]

Number of items: 10

Bin capacity: 15

Minimum number of bins required (desired output): 8

Bin 1 : [12, 3]

Bin 2 : [10]

Bin 3 : [15]

Bin 4 : [12]

Bin 5 : [11]

Bin 6 : [10]

Bin 7 : [6, 8]

Bin 8 : [11]

TEST CASE 17

Sizes of items: [8, 4, 6, 3, 5, 2, 3, 3, 6]

Number of items: 9

Bin capacity: 11

Minimum number of bins required (desired output): 4

Bin 1 : [8, 3]

Bin 2 : [4, 6]

Bin 3 : [5, 2, 3]

Bin 4 : [3, 6]

TEST CASE 18

Sizes of items: [1, 8, 3, 8, 10]

Number of items: 5

Bin capacity: 10

Minimum number of bins required (desired output): 4

Bin 1 : [1, 8]

Bin 2 : [3]

Bin 3 : [8]

Bin 4 : [10]

TEST CASE 19

Sizes of items: [4, 5, 4, 5, 5]

Number of items: 5

Bin capacity: 5

Minimum number of bins required (desired output): 5

Bin 1 : [4]

Bin 2 : [5]

Bin 3 : [4]

Bin 4 : [5]

Bin 5 : [5]

TEST CASE 20

Sizes of items: [6, 4, 1, 12, 3]

Number of items: 5

Bin capacity: 15

Minimum number of bins required (desired output): 2

Bin 1 : [6, 4, 1]

Bin 2 : [12, 3]

6. Experimental Analysis of The Performance (Performance Testing)

The complexity of our heuristic approach, which uses the Best-Fit algorithm, is O(n2), where n is the number of items. Changing the number of items is important to test the time complexity of the algorithm. But we will also show the results in bin ranges, which are shown in code, to show that it doesn't matter.

For each of the test cases below, there are 100 repetitions from 0 to 2000. Based on the results of our speed tests, the calculations in the algorithm analysis part take exactly O(n2) polynomial time (with degree 2). In reality, the greedy algorithm Best-Fit seems to take as long as it would take in the worst case.

Num Runs	Processing Time	90% CI Lower Bound	90% CI Upper Bound	Standard Deviation	Standard Error
100	0.00081253051757812	0.06202608959839488	0.13797391040160512	1.5742327078102832e-05	1.5742327078102832e-06
200	0.0019366741180419922	0.09372385835613492	0.10627614164386508	2.5453262694323602e-06	1.799817465447879e-07
300	0.0035254955291748047	0.09621953916365719	0.1037804608363428	1.7693067257832044e-06	1.0215097144099484e-07
400	0.005652427673339844	0.09646435675465759	0.10353564324534241	1.9183860379639737e-06	9.591930189819868e-08
500	0.009825468063354492	0.09131485039840395	0.10868514960159606	8.269682163447305e-06	3.69831429395714e-07
600	0.013046741485595703	0.09695396364092611	0.10304603635907388	2.0437650863388793e-06	8.343636026075769e-08
700	0.016791105270385742	0.09717355639277755	0.10282644360722244	2.0408602626591677e-06	7.713726736614454e-08
800	0.021079063415527344	0.09742624546944297	0.10257375453055703	1.9907494151331774e-06	7.038362055419115e-08
900	0.029597043991088867	0.0970277149305824	0.10297228506941761	4.305759606518825e-06	1.4352532021729419e-07
1000	0.03497815132141113	0.09774369580381312	0.10225630419618688	1.958904366019722e-06	6.194599515070468e-08
1100	0.04093289375305176	0.09785401089900217	0.10214598910099784	1.9657943347101358e-06	5.927092930218033e-08
1200	0.0474240779876709	0.0979758729397733	0.1020241270602267	1.9351516835597555e-06	5.586301727129909e-08
1300	0.05449986457824707	0.09486384800969459	0.10513615199030539	5.142624865473509e-06	1.426307510995449e-07
1400	0.06192946434020996	0.09788981530819715	0.10211018469180286	2.137793723108393e-06	5.713494053905346e-08
1500	0.06990671157836914	0.09819949810802739	0.10180050189197261	1.8921036935476515e-06	4.885390729605064e-08
1600	0.07839488983154297	0.0981910349976352	0.1018089650023648	1.9585226267634495e-06	4.896306566908624e-08
1700	0.08789539337158203	0.09567751669263248	0.10432248330736753	5.0815876023476624e-06	1.232466025312271e-07
1800	0.09751582145690918	0.09836826588114174	0.10163173411885826	1.887780380274343e-06	4.449541027451986e-08
1900	0.10778450965881348	0.09793370929264293	0.10206629070735707	2.4835534886092247e-06	5.697662461960791e-08
2000	0.11938023567199707	0.09675272013916676	0.10324727986083324	4.295825138514667e-06	9.605757029171246e-08

Figure: Performance Testing 100 to 2000

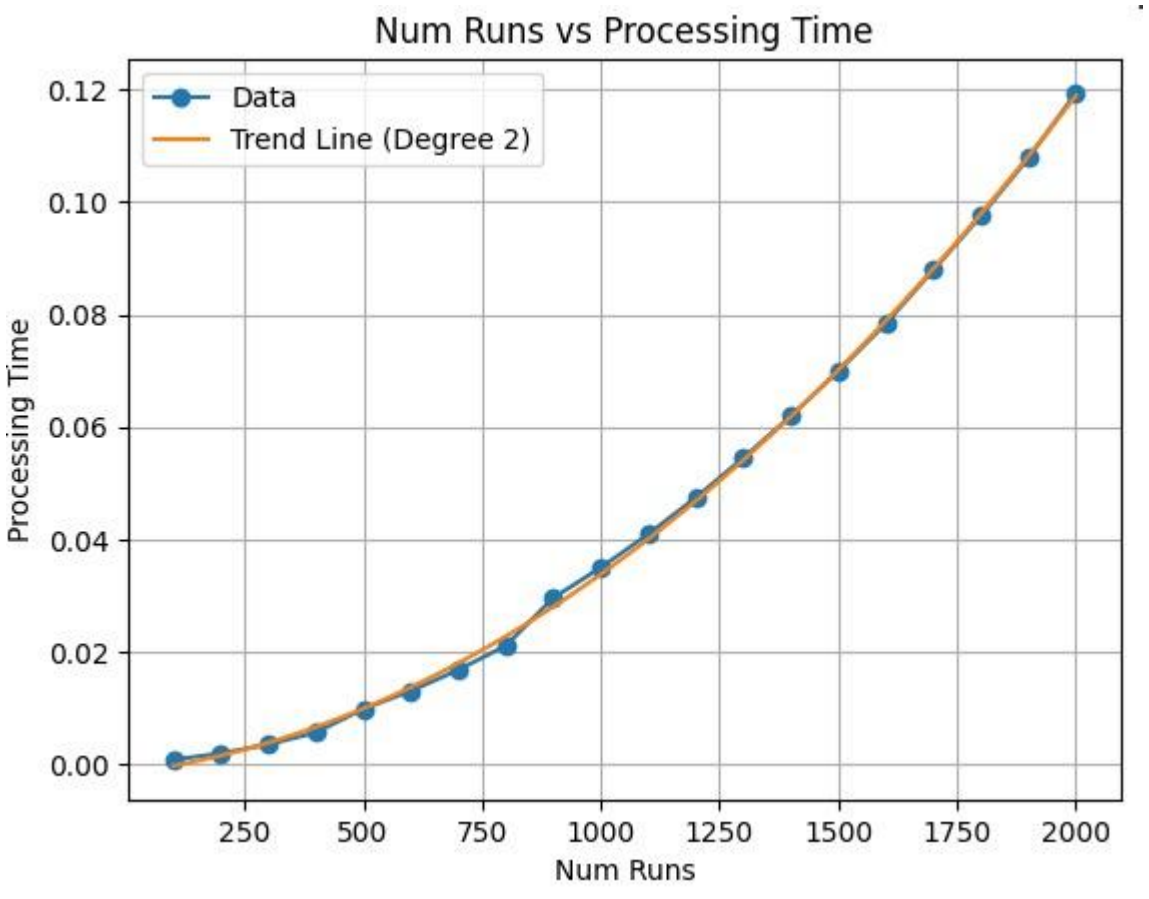

Table: Performance Testing 100 to 2000 Graph

7. Experimental Analysis of the Correctness (Functional Testing)

Unittest library offers a basic test suite in the form of a class, and each test case is expressed by a class method of the test suite. For Black-Box testing, two test suites were made: one for the right algorithm that was used before and one for the algorithm that works well. In an answer Since exhaustive testing is thought to be theoretically not possible, extreme cases and one normal case have been used. These include a case where the bins are empty and the sizes of the items are zero, a case where the sizes of the items are not zero but the binsize is zero, a case where the binsize is one but the weights are one, and a regular input.

White-Box Testing looked at each piece of code on its own. Each part has its own test suite, so there are a total of 18 tests in 4 test suites. You can see test packages that aren't on this list.

```
import unittest
from bestFit import bestFit
class bestFitTest(unittest.TestCase):
    def test_empty(self):
        self.assertEqual(bestFit([0], 0, 0), 0)
    def test_regularInputTest(self):
        weights = [1,2,1]
        result = bestFit(weights, len(weights), 3)
        expected = 2
        self.assertEqual(result, expected)
    def test_zeroBinSize(self):
        weights = [1,2,1]
        result = bestFit(weights, len(weights), 0)
        expected = 3
        self.assertEqual(result, expected)
    def test_oneBinSize(self):
        weights = [1,2,1]
        result = bestFit(weights, len(weights), 1)
        expected = 3
        self.assertEqual(result, expected)
```

(Heuristic Black-Box Testing)

```python
#WHITE BOX
import unittest
from bin import Bin
from bin import getFilledBinsCount
from bin import bins
class binTest(unittest.TestCase):
    def test_put_half(self):
        bin = Bin(10)
        self.assertTrue(bin.put(5))
    def test_fill(self):
        bin = Bin(10)
        self.assertTrue(bin.put(5))
    def test_put_too_much(self):
        bin = Bin(10)
        self.assertFalse(bin.put(11))
    def test_remove(self):
        bin = Bin(10)
        bin.put(5)
        bin.remove(5)
        self.assertEqual(len(bin.contents), 0)
    def test_getFilledBinsCount(self):
        self.assertEqual(getFilledBinsCount(),0)
    def test_getFilledBinsCountFull(self):
        bins[0].put(9)
        self.assertEqual(getFilledBinsCount(),1)
```

(Heuristic White-Box Testing)

8. Discussion

To initiate a discourse on our results, let us commence with an examination of the experimental analysis pertaining to the section on accuracy. The study exhibited the flawless nature of the brute force algorithm and the Best-Fit algorithm implementations through the utilization of functional testing techniques such as Black Box Testing and White Box Testing. Furthermore, the group has previously demonstrated through algorithmic analysis that both approaches are precise in terms of the specific actions taken to resolve the matter under consideration.

Subsequently, our experimental investigation has demonstrated that the actual performance of the Best-Fit algorithm is consistent with our theoretical analysis of its worst-case time complexity in Algorithm analysis. To clarify, the findings of our research indicate that the Best-Fit algorithm exhibits a time complexity of n2 in the worst-case scenario.

Furthermore, as indicated in the experimental analysis of performance, it was discovered that the ratio bound observed in practical application aligns with the proven theoretical ratio bound, specifically, the ratio bound in practical application is 1.7 times the optimal solution. The rationale behind our study was to evaluate the approximation efficacy of the Best-Fit algorithm with respect to both optimal and approximation values. Furthermore, the ratio limit that was empirically observed was approximately 1.23 OPT, surpassing the ratio constraint of 1.7 OPT.

References

- *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.
- *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.