CS260P PROJECT 2 REPORT

BIN PACKING

Contents:

THEORETICAL ANALYSIS

Bin Packing

I. Definition and Explanation

In the bin packing problem, objects of different volumes must be packed into a finite number of bins or containers each of volume V in a way that minimizes the number of bins used. In computational complexity theory, it is a combinatorial NP-hard problem. The decision problem (deciding if objects will fit into a specified number of bins) is NP-complete.

Since, Bin Packing is a NP-complete problem, there is no standard algorithm which is computationally better than the others, the various policies (which we will describe in this project later) performance is heavily dependent on the data that it is trying to fit, for different types of data different bin packing algorithms can show optimality.

With that said, the running time of these algorithms depends heavily on the policy used for fitting the data in the bins. The performance of a Bin Packing algorithm is generally based on the waste, W(A) that it produces. The better algorithm is the one which generates lower waste.

The general algorithm for Bin Packing is:

i. Decide on a policy which will be used for packing the bins.
ii. Once a suitable policy is decided, find the number of bins requires to pack the data based on the aforementioned policy.
iii. Calculate the waste W(A) of the Bin Packing algorithm, the waste is the difference between the sum of the data (numbers) and the number of bins used.

In this project we experimentally analyze running times using 5 Bin Packing algorithms. They are:

a) Next-fit (NF)
b) First-fit (FF)
c) Best-fit (BF)
d) First-fit Decreasing (FFD)
e) Best-fit Decreasing (BFD)

Therefore, we take a Bin Fitting algorithm one by one and do a comparative theoretical and experimental analysis.

II. Algorithm and Analysis

a) Next Fit

The general algorithm for generating the number of bins is as follows:

i. Check to see if the current item fits in the current bin. If so, then place it there, otherwise start a new bin.
ii. This algorithm generates the maximum amount of waste W(A).

For an input shown below and bins of size 1.0:

```
[0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6]
```

The Next Fit Bin Packing algorithm gives an output of 6 bins.

b) First Fit

The general algorithm for generating the number of bins is as follows:

i. Scan the bins in order and place the new item in the first bin that is large enough to hold it. A new bin is created only when an item does not fit in the previous bins.
ii. This algorithm generates less amount of waste W(A) than Next Fit Algorithm.

For an input shown below and bins of size 1.0:

```
[0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6]
```

The First Fit Bin Packing algorithm gives an output of 5 bins.

Extra Credit: In the implementation of the algorithm, I have used a TreeMap which is implemented as a Red-Black Tree by Java. The TreeMap stores the index of the bin in the key and the remaining value in the value. It is sorted in natural order of the index. This algorithm runs in **Average Case O(nlogn)**, and worst-case O(n^2).

c) Best Fit

The general algorithm for generating the number of bins is as follows:

i. New item is placed in a bin where it fits the tightest. If it does not fit in any bin, then start a new bin.
ii. This algorithm generates less amount of waste W(A) than First Fit Algorithm.

For an input shown below and bins of size 1.0:

```
[0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8]
```

The Best Fit Bin Packing algorithm gives an output of 4 bins.

Extra Credit: In the implementation of the algorithm, I have used a TreeMap which is implemented as a Red-Black Tree by Java. The TreeMap stores the remaining value of the bin as the key and the number of bins of that remaining key as the value.

The algorithm runs in **worst-case O(nlogn)**, since I have used the function containsKey() and higherKey(), for getting the required Bin.

d) First Fit Decreasing

The general algorithm for generating the number of bins is as follows:

i. Sort the numbers in decreasing order
ii. Scan the bins in order and place the new item in the first bin that is large enough to hold it. A new bin is created only when an item does not fit in the previous bins.
iii. This algorithm generates less amount of waste W(A) than Next Fit & First Fit Algorithm.

For an input shown below and bins of size 1.0:

```
[0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6]
```

The First Fit Decreasing Bin Packing algorithm gives an output of 4 bins.

e) Best Fit Decreasing

The general algorithm for generating the number of bins is as follows:

i. Sort the numbers in decreasing order
ii. New item is placed in a bin where it fits the tightest. If it does not fit in any bin, then start a new bin.
iii. This algorithm generates less amount of waste W(A) than First Fit, and Best Fit Algorithm.

For an input shown below and bins of size 1.0:

```
[0.2, 0.5, 0.4, 0.7, 0.1, 0.3, 0.8]
```

The Best Fit Bin Packing algorithm gives an output of 3 bins.

III. Worst-case running time

Since Bin Packing is a NP-complete problem, there is no real worst-case time, it is dependent on the kind of data being used. That being said, generally the **running time comparison** of the algorithms is:

Next Fit > First Fit > Best Fit > First Fit Decreasing > Best Fit Decreasing

So, Best Fit Decreasing generally gives the best results.

Now, we will move on to the experimental analysis of the sequences.

EXPERIMENTAL ANALYSIS

I. Types of data

We are using Randomly generated data. Java's Random class is used for generating this data. *Random.nextDouble()* function is used to generate the data.

Randomly Generated Data:

The data was kept at a 1 digit precision to facilitate simple operations. This was done using the *BigDecimal()* object for exact floating-point operations on the data.

The range of the data generated was [0.1-0.8].

In the test cases the algorithms were run on 10 different randomly generated sets of data, because one randomly generated set can be better than the other, and running it on many data points helps us get a general view of the runtime of the algorithm.

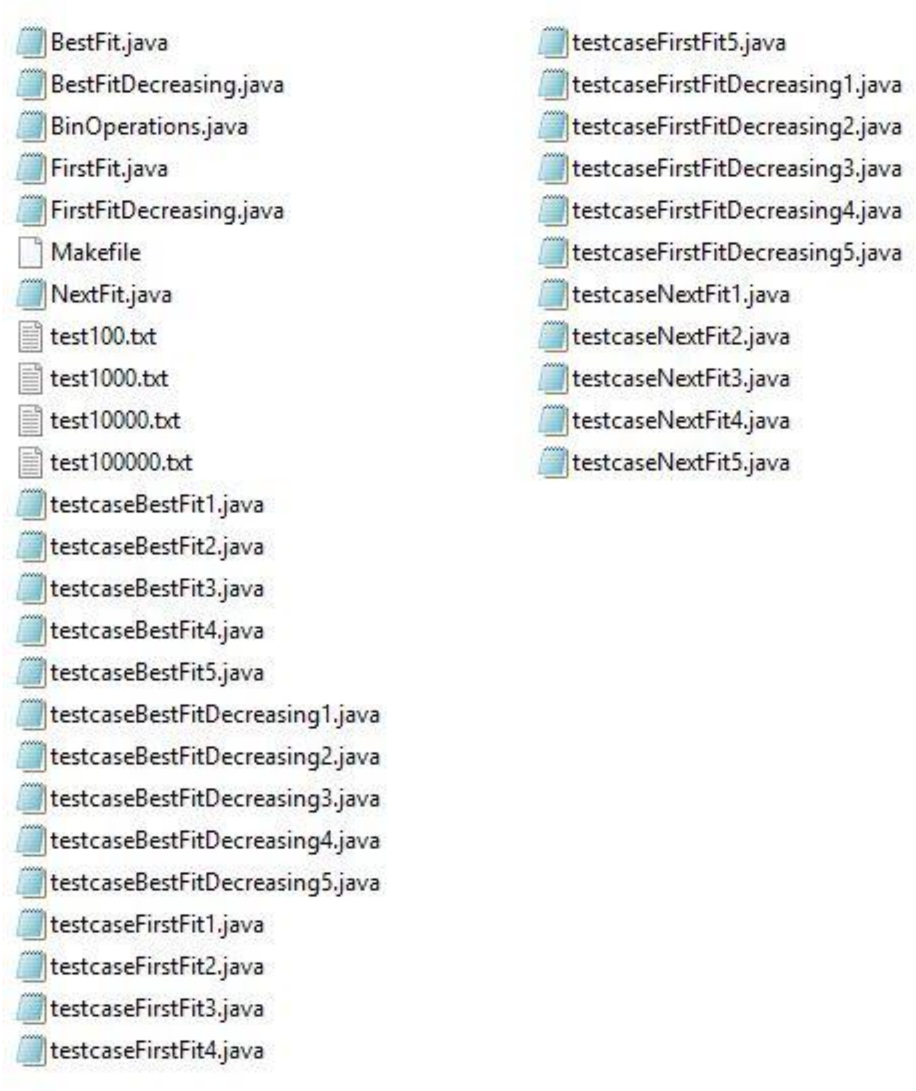
Also, static files were generated of various sizes and the algorithms were also run on that file to get a comparative measure of the waste W(A) from the average of the 10 datasets and the static file.

II. Running the testcases

Each algorithm was run on a size of 100, 1000, 10000, 100000.

The tree structure of the files of code is given on the next page.

There are a total of 5 test cases for each algorithm. The first test case check the correctness of the algorithm, and the next 4 test cases run the algorithm of the 10 randomly generated datasets and the static files and give us an average value for the waste W(A).



```
BestFit.java
BestFitDecreasing.java
BinOperations.java
FirstFit.java
FirstFitDecreasing.java
Makefile
NextFit.java
test100.txt
test1000.txt
test10000.txt
test100000.txt
testcaseBestFit1.java
testcaseBestFit2.java
testcaseBestFit3.java
testcaseBestFit4.java
testcaseBestFit5.java
testcaseBestFitDecreasing1.java
testcaseBestFitDecreasing2.java
testcaseBestFitDecreasing3.java
testcaseBestFitDecreasing4.java
testcaseBestFitDecreasing5.java
testcaseFirstFit1.java
testcaseFirstFit2.java
testcaseFirstFit3.java
testcaseFirstFit4.java
testcaseFirstFit5.java
testcaseFirstFitDecreasing1.java
testcaseFirstFitDecreasing2.java
testcaseFirstFitDecreasing3.java
testcaseFirstFitDecreasing4.java
testcaseFirstFitDecreasing5.java
testcaseNextFit1.java
testcaseNextFit2.java
testcaseNextFit3.java
testcaseNextFit4.java
testcaseNextFit5.java
```

To compile all the files, the *make* command can be used on the terminal.

The *.txt* files contain the static file data. The files are named as *test<input_size>.txt*

The testcases are named as *testcase<gap_sequence><testcase_number>*

Ex: test100.txt is randomly generated data of size 100.

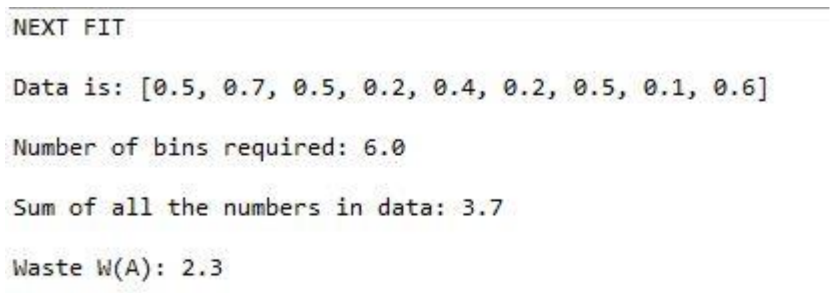
Ex: testcaseNextFit1.java is testcase 1 for Next Fit Bin Packing.

The testcases are designed as follows:

i. Testcase 1 : This testcase check for the correctness of the algorithm using the examples given in the slides in the class.

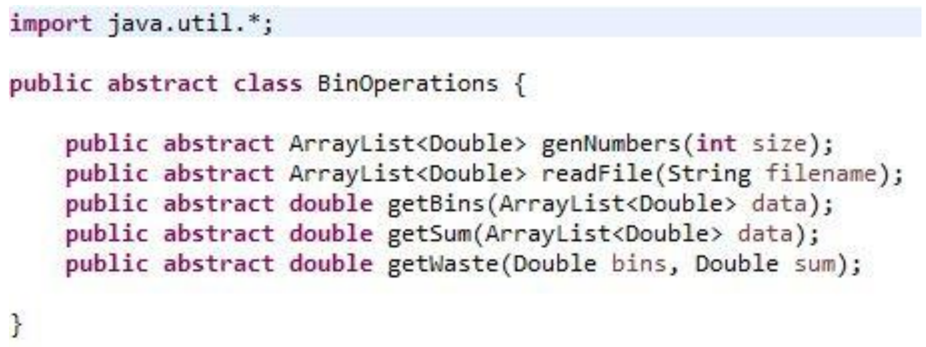
ii. Testcase 2 : This testcase computes the waste(A) of a dataset of 100 numbers for 10 randomly generated datasets and computes the average waste W(A) over all the datasets. Also it computes the W(A) for a static file of size 100.
iii. Testcase 3 : This testcase computes the waste(A) of a dataset of 1000 numbers for 10 randomly generated datasets and computes the average waste W(A) over all the datasets. Also it computes the W(A) for a static file of size 1000.
iv. Testcase 4 : This testcase computes the waste(A) of a dataset of 10000 numbers for 10 randomly generated datasets and computes the average waste W(A) over all the datasets. Also it computes the W(A) for a static file of size 10000.
v. Testcase 5 : This testcase computes the waste(A) of a dataset of 100000 numbers for 10 randomly generated datasets and computes the average waste W(A) over all the datasets. Also it computes the W(A) for a static file of size 100000.

A sample of testcase 1 output for Next Fit is given below:



```
NEXT FIT

Data is: [0.5, 0.7, 0.5, 0.2, 0.4, 0.2, 0.5, 0.1, 0.6]

Number of bins required: 6.0

Sum of all the numbers in data: 3.7

Waste W(A): 2.3
```

An abstract class (BinOperation.java) is created which implements the same functions for various Bin Fit algorithms. It is given below:



```
import java.util.*;

public abstract class BinOperations {

    public abstract ArrayList<Double> genNumbers(int size);
    public abstract ArrayList<Double> readFile(String filename);
    public abstract double getBins(ArrayList<Double> data);
    public abstract double getSum(ArrayList<Double> data);
    public abstract double getWaste(Double bins, Double sum);

}
```

genNumbers() – generates random floating point values of given size

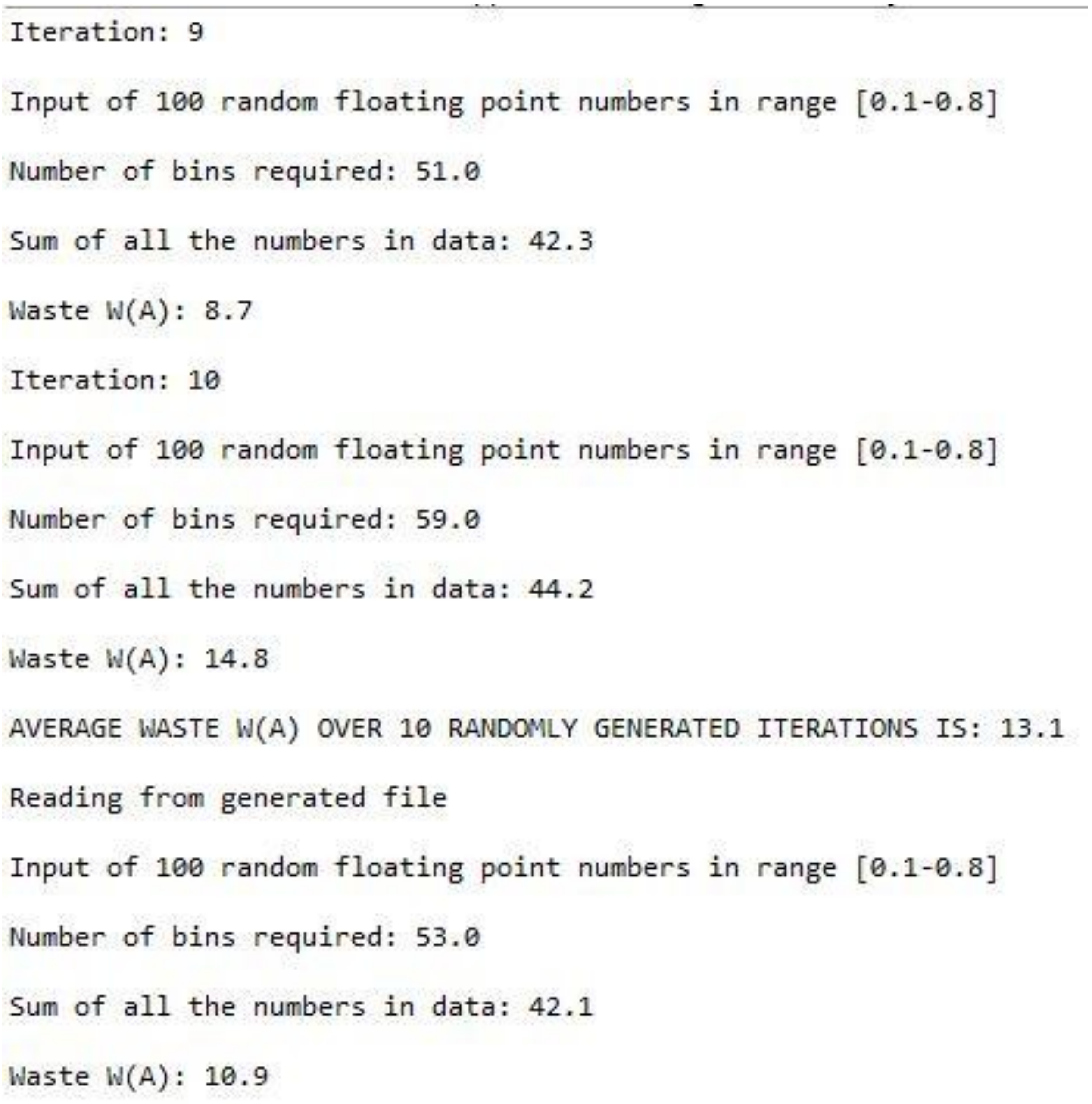
readFile() – reads uniformly distributed data from a file

getBins() – computes number of bins for an algorithm getSum() – returns sum of all the data values

getWaste() - returns waste W(A) of a given algorithm

To run a testcase (after compilation) – *java testcaseShellNormal1*

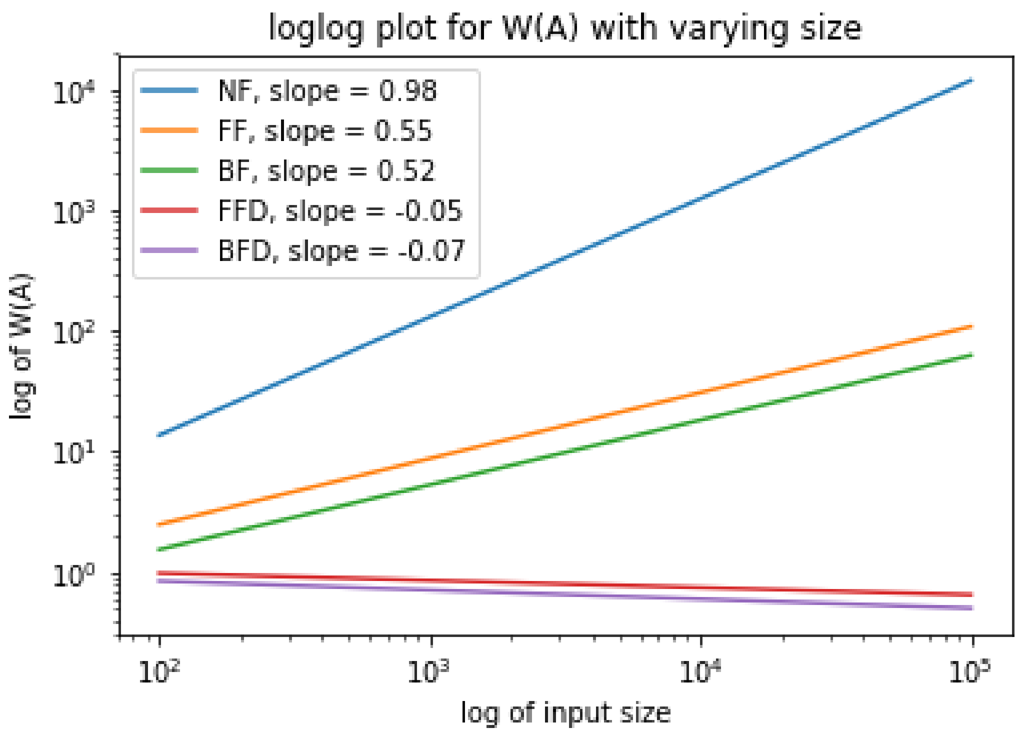
A sample of testcase 2 output for Next Fit is given below:



```
Iteration: 9

Input of 100 random floating point numbers in range [0.1-0.8]

Number of bins required: 51.0

Sum of all the numbers in data: 42.3

Waste W(A): 8.7

Iteration: 10

Input of 100 random floating point numbers in range [0.1-0.8]

Number of bins required: 59.0

Sum of all the numbers in data: 44.2

Waste W(A): 14.8

AVERAGE WASTE W(A) OVER 10 RANDOMLY GENERATED ITERATIONS IS: 13.1

Reading from generated file

Input of 100 random floating point numbers in range [0.1-0.8]

Number of bins required: 53.0

Sum of all the numbers in data: 42.1

Waste W(A): 10.9
```

III. Experimental Analysis

i. Graph 1: Comparison Bin Fitting Algorithms



The graph shown above is a loglog plot for all the Bin Fitting algorithms run of randomly generated data.

We observe from the graph that the Best Fit Decreasing algorithm performs the best of all the Bin Fitting algorithms.

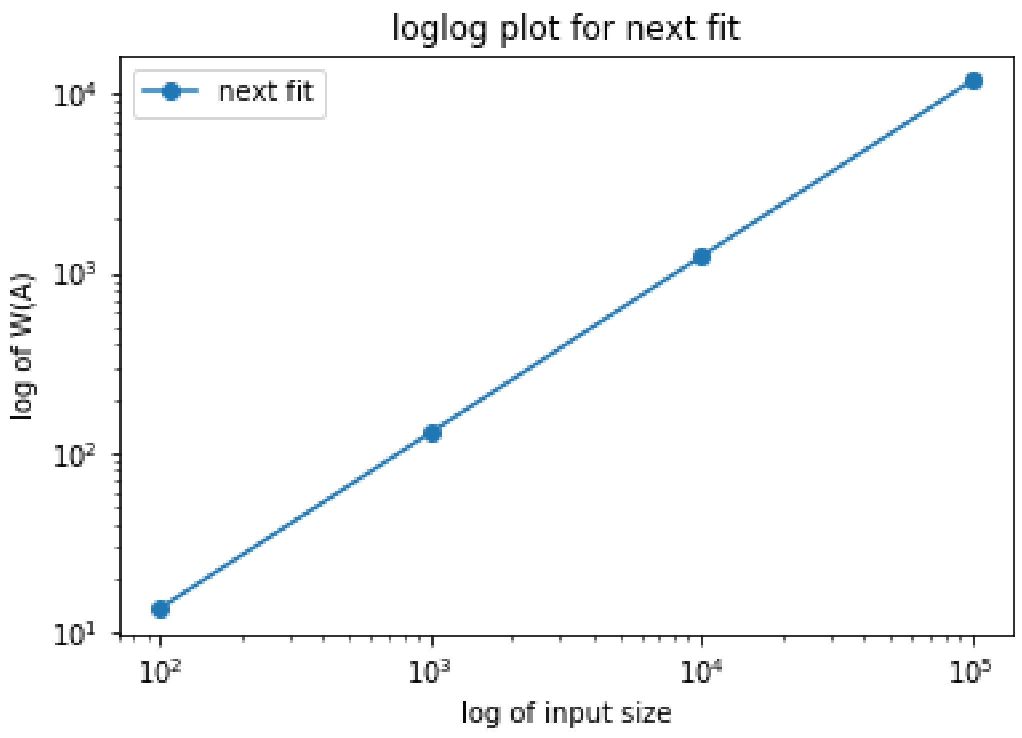
The Next Fit Algorithm performs way worse than all the other algorithms.

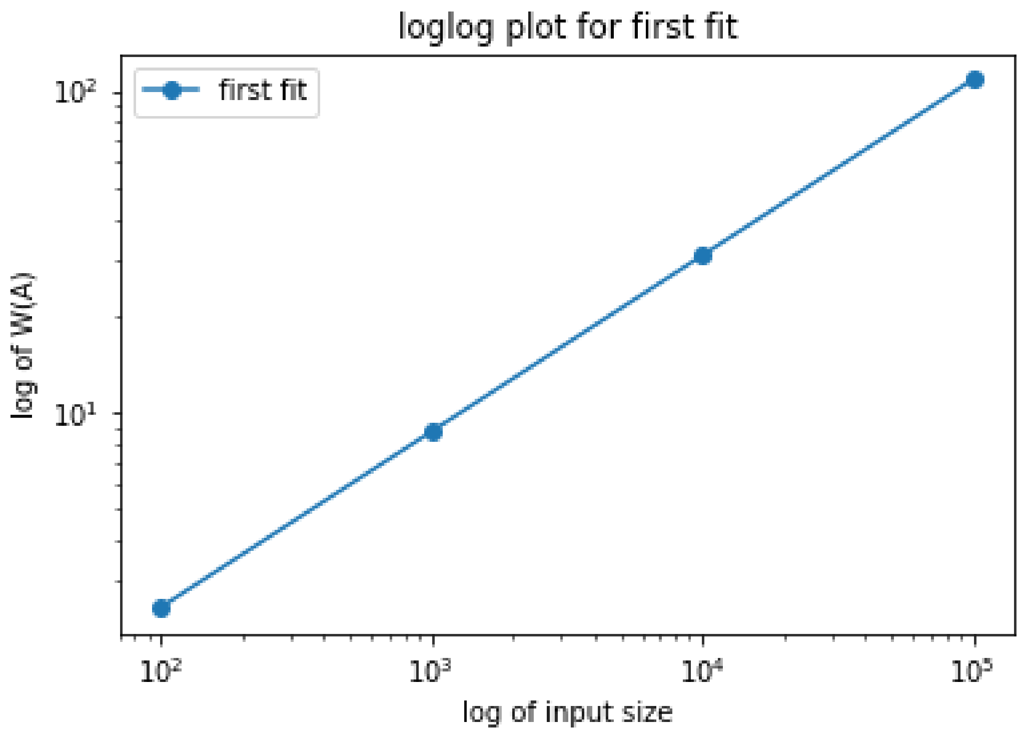
Surprisingly, for First-fit decreasing and Best-fit decreasing algorithms, we get a negative slope, that means sometimes it performs better on data of greater size.

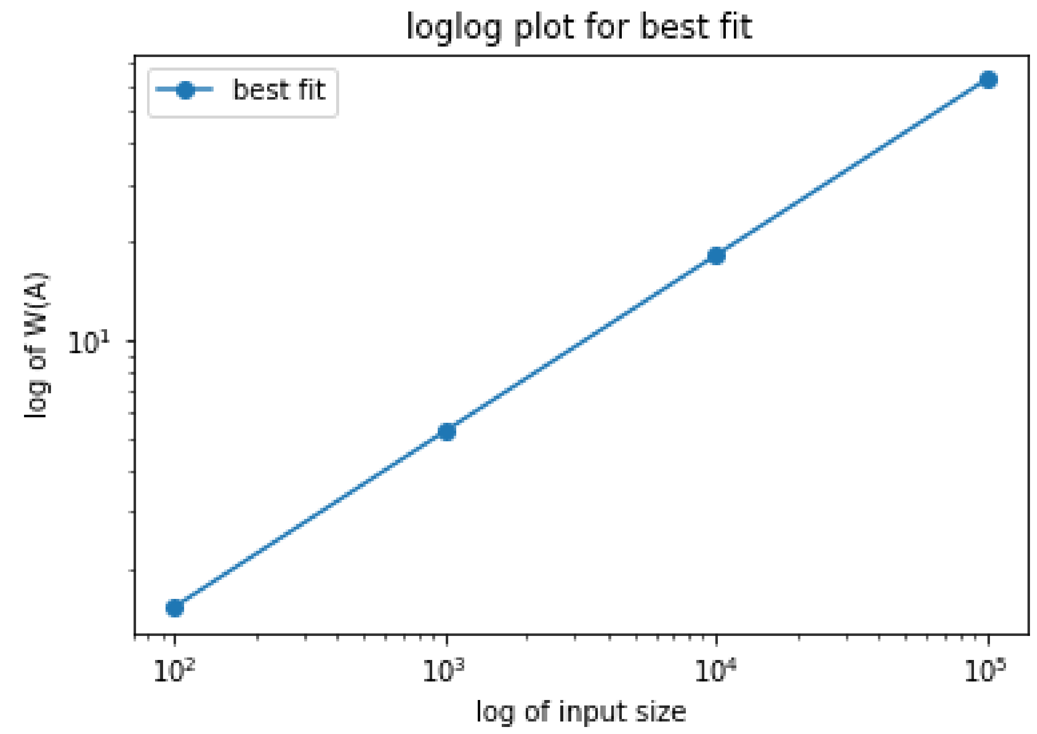
The reason for negative slope (of very less value) can be:

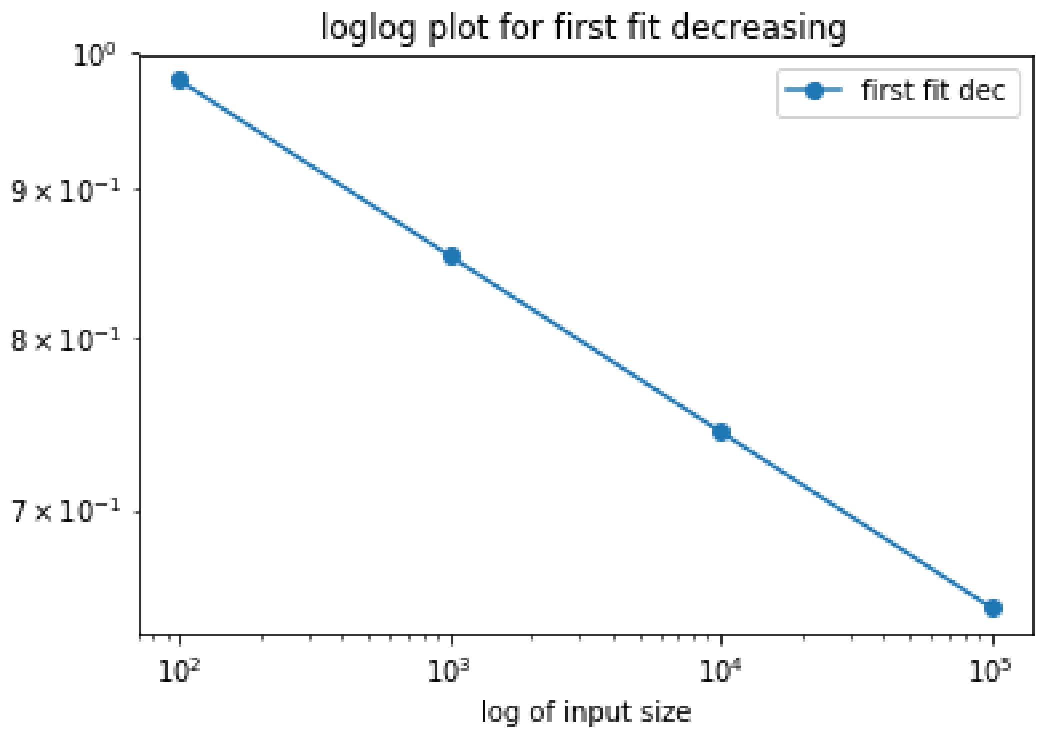
a. Since the precision of floating point values are 1, we can be fitting the data perfectly in some cases.
b. In some cases the randomly generated data must be a good fit and the average comes out to be a negative slope.

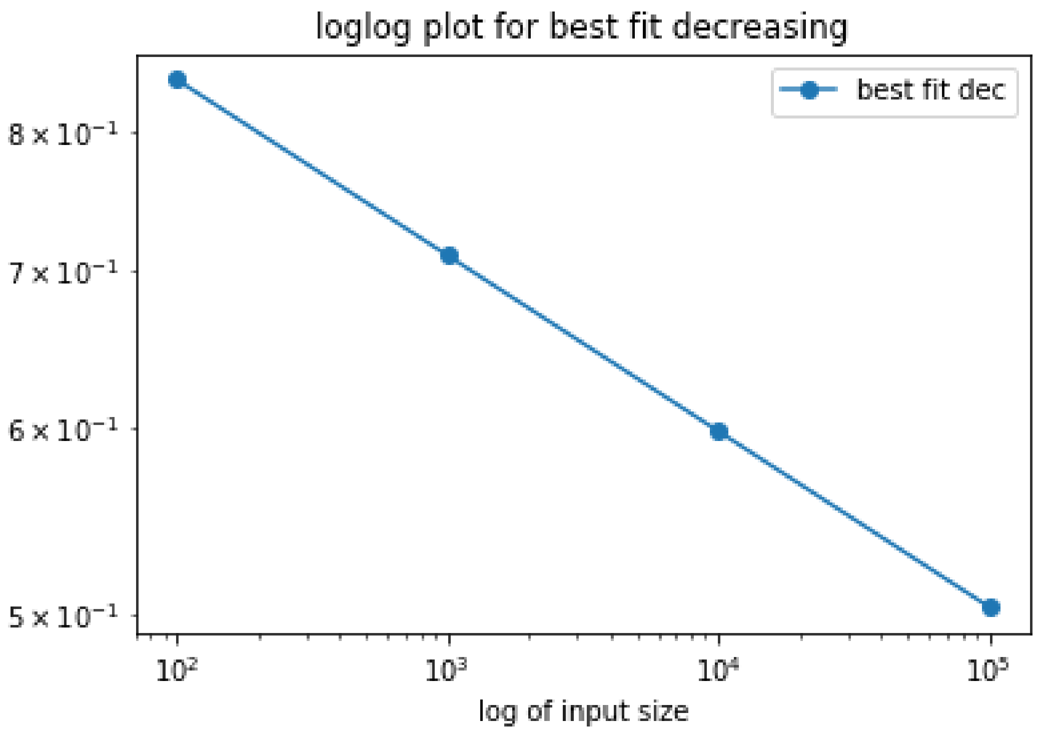
ii. Series of graphs comparing the running times for uniformly distributed and almost sorted data for each algorithm






loglog plot for best fit
best fit
log of W(A)
10^1
10^2
10^3
10^4
10^5
log of input size


loglog plot for first fit decreasing
first fit dec
10^0
9×10^{-1}
8×10^{-1}
7×10^{-1}
10^2
10^3
10^4
10^5
log of input size



From the series of graphs above, we can come to the conclusion that the Best Fit Decreasing algorithm performs the best.

One peculiar thing that we can observe is the negative slope for First-fit decreasing and Best-fit decreasing.

These graphs were plotted by taking the line estimate of the points on the loglog scale. The line was estimated using Python's *numpy.polyfit()* function.

To experimentally derive the complexity of the algorithms, we can calculate the slope of the lines on the loglog scale and n to the power of the slope.

The value of slope therefore gives us a metric of how well the algorithm scales.

The comparative analysis is given on the next page.

IV. Comparative Analysis

i. Randomly generated data

From the graphs the derived complexity is as follows (from highest to lowest)

Gap Sequence	Slope	Complexity
Next Fit	0.98	$n^{0.98}$
First Fit	0.55	$n^{0.55}$
Best Fit	0.52	$n^{0.52}$
First Fit Decreasing	-0.05	$\frac{1}{n^{0.05}}$
Best Fit Decreasing	-0.07	$\frac{1}{n^{0.07}}$

V. General Conclusions

i. The experimental complexities differ from what is theoretically predicted as the complexity.

ii. In general the Best Fit Decreasing algorithm works very well for randomly generated data, both theoretically and experimentally.

iii. Surprisingly both the First-fit decreasing and the Best-fit decreasing algorithms give us a negative slope, that is, they work better for larger data size.

iv. This negative slope can be due to the fact that we use 1 digit precision which helps it get a better fit on the data, and the randomly generated data may work in favor of the algorithm so that the overall average is negative.

v. The Bin Packing problem is NP-complete and there is still active research in the area of improving the algorithms.