Project 2

Bin Packing

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. There are many variations of this problem, such as 2D packing, linear packing, packing by weight, packing by cost, and so on and the efficiency of a bin-packing algorithm depends on the variation it uses for placing the objects in the bin.

Pseudo-code of Bin Packing algorithm:

```
def binpacking(objects, capacity_of_bin):
        n = len(objects)
        remaining[n]
        for i,object in objects:
                if object fits in the current bins available:
                        remaining[i] = remaining[i] - volume(object)
                else:
                        create new bin and add that object in new bin
        return number_of_bins
```

Design Choices and Data Structure Used

In the bin-packing method, we have used an array to store the remaining volume of every bin and we loop over the objects and place the object that fits in the bin that has same or more volume available in it. The objects are placed in the bin based on different algorithms and the inputs of these methods are the list of objects and the capacity of each bin. The list of objects is stored in an array of size n. We generate the list of objects, each object having a volume between 0 and 0.8, by a method whose input is again the size (n) of the array and insert n random floating numbers in the array. The sizes vary for different variety of test cases. We then test this randomly generated list of volumes of objects on various bin-packing methods.

Different versions of Bin Packing

Here are the 5 different versions of Bin Packing algorithm.

1. Next Fit (NF).
2. First Fit (FF).
3. Best Fit (BF).
4. First Fit Decreasing (FFD).

5. Best Fit Decreasing (BFD).

For each bin-packing algorithm we'll find the waste, W(A) for any given list of objects, which is the number of bins used by the algorithm A minus the total size (i.e., the sum) of all the items in the list. We'll generate the list of volumes of objects by Uniformly Distributed Random Permutations with multiple runs for each problem size, for increasing problem sizes.

Uniformly Distributed Permutations:

Here we randomly generate a list of volumes of objects of a particular size and run the bin-packing algorithm for analyzing the waste, W(A) for different sizes of lists. We also perform multiple runs for a particular size.

```
def uniformPerm(size):
    arr[size]
    for i in range(size):
        arr[i] = random(0,0.8) //between 0 and 0.8
    ret arr
```

Experimental Analysis:

Next Fit (NF):

Pseudo-code:

```
def nextfit(objects, capacity):
    bins = 1
    remaining = capacity
    for i in range (len(objects)):
        if objects[i] <= remaining:
            remaining  = remaining - objects[i]
        else:
            bins++
            remaining = 1 - objects[i]
    return bins
```

Time complexity: O(n), where n is the number of objects.

Uniformly Distributed Permutations:

We run the uniform distributed permutations on different sizes of the arrays and run with 5 different permutations for a particular size of array by randomly generating the list every time. We then take the average of 5 different permutations run for a particular size.

size = 64	Bins = 37	W(A) = 9.6
size = 128	Bins = 68	W(A) = 17.36

size = 256	Bins = 143	W(A) = 39.54
size = 512	Bins = 283	W(A) = 80.12
size = 1024	Bins = 552	W(A) = 144.97
size = 2048	Bins = 1120	W(A) = 301.03
size = 4096	Bins = 2220	W(A) = 584.36
size = 8192	Bins = 4460	W(A) = 1185.46
size = 16384	Bins = 8930	W(A) = 2367.04
size = 32768	Bins = 17864	W(A) = 4737.81
size = 65536	Bins = 35676	W(A) = 9457.34
size = 131072	Bins = 71272	W(A) = 18890.82
size = 262144	Bins = 142927	W(A) = 37921.87

First Fit (FF):

Pseudo-code:

```
def firstfit(objects, capacity):
    bins = 1
    n = len(objects)
    remaining[n]
    for i in range(n): remaining[i] = 1
    for i in range(n):
        for j in range(bins):
            if remaining[j] >= items[i]:
                remaining[j] = remaining[j] - items[i]
                break
        if j == bins:
            remaining[j] = remaining[j] - items[i]
            bins++
    return bins
```

Time complexity: O(n^2), where n is the number of objects.

Uniformly Distributed Permutations:

We run the uniform distributed permutations on different sizes of the arrays and run with 5 different permutations for a particular size of array by randomly generating the list every time. We then take the average of 5 different permutations run for a particular size.

size = 64	Bins = 31	W(A) = 3.6

size = 128	Bins = 56	W(A) = 4.76
size = 256	Bins = 111	W(A) = 6.94
size = 512	Bins = 215	W(A) = 10.12
size = 1024	Bins = 423	W(A) = 16.77
size = 2048	Bins = 844	W(A) = 25.24
size = 4096	Bins = 1679	W(A) = 43.36
size = 8192	Bins = 3351	W(A) = 76.86
size = 16384	Bins = 6699	W(A) = 135.87
size = 32768	Bins = 13392	W(A) = 265.81
size = 65536	Bins = 26727	W(A) = 507.94
size = 131072	Bins = 53375	W(A) = 993.82
size = 262144	Bins = 106999	W(A) = 1993.87

Best Fit (BF):

Pseudo-code:

```
def bestfit(objects, capacity):
    bins = 1
    n = len(objects)
    remaining[n]
    for i in range(n): remaining[i] = 1
    for i in range(n):
        index = -1
        min = 1.1
        for j in range(bins):
            if remaining[j] >= items[i]:
                if min > remaining[j] - items[i]:
                    min = remaining[j] - items[i]
                    index = j
        if min < 1.1:
            remaining[index] -= items[i]
        else:
            remaining[bins] -= items[i]
            bins++
    return bins
```

Time complexity: O(n^2), where n is the number of objects.

Uniformly Distributed Permutations:

We run the uniform distributed permutations on different sizes of the arrays and run with 5 different permutations for a particular size of array by randomly generating the list every time. We then take the average of 5 different permutations run for a particular size.

size = 64	Bins = 30	W(A) = 2.2
size = 128	Bins = 55	W(A) = 3.2
size = 256	Bins = 109	W(A) = 4.14
size = 512	Bins = 211	W(A) = 5.9
size = 1024	Bins = 419	W(A) = 8.6
size = 2048	Bins = 835	W(A) = 12.83
size = 4096	Bins = 1666	W(A) = 20.82
size = 8192	Bins = 3328	W(A) = 36.65
size = 16384	Bins = 6660	W(A) = 66.07
size = 32768	Bins = 13324	W(A) = 114.81
size = 65536	Bins = 26593	W(A) = 205.54
size = 131072	Bins = 53121	W(A) = 396.02
size = 262144	Bins = 106494	W(A) = 782.87

First Fit Decreasing (FFD):

Pseudo-code:

```
def decfirstfit(objects, capacity):
    bins = 1
    n = len(objects)
    sort(objects, reverse)
    remaining[n]
    for i in range(n): remaining[i] = 1
    for i in range(n):
        for j in range(bins):
            if remaining[j] >= items[i]:
                remaining[j] = remaining[j] - items[i]
                break
        if j == bins:
            remaining[j] = remaining[j] - items[i]
            bins++
    return bins
```

Time complexity: O(n^2), where n is the number of objects.

Uniformly Distributed Permutations:

We run the uniform distributed permutations on different sizes of the arrays and run with 5 different permutations for a particular size of array by randomly generating the list every time. We then take the average of 5 different permutations run for a particular size.

size = 64	Bins = 27	W(A) = 0.8
size = 128	Bins = 51	W(A) = 0.95
size = 256	Bins = 104	W(A) = 1.1
size = 512	Bins = 204	W(A) = 1.21
size = 1024	Bins = 407	W(A) = 1.28
size = 2048	Bins = 820	W(A) = 1.36
size = 4096	Bins = 1637	W(A) = 1.48
size = 8192	Bins = 3276	W(A) = 1.58
size = 16384	Bins = 6565	W(A) = 1.82
size = 32768	Bins = 13302	W(A) = 2.02
size = 65536	Bins = 26221	W(A) = 2.24
size = 131072	Bins = 52385	W(A) = 2.52
size = 262144	Bins = 105010	W(A) = 2.87

Best Fit Decreasing (BF):

Pseudo-code:

```
def decbestfit(objects, capacity):
    bins = 1
    n = len(objects)
    sort(objects, reverse)
    remaining[n]
    for i in range(n): remaining[i] = 1
    for i in range(n):
        index = -1
        min = 1.1
        for j in range(bins):
            if remaining[j] >= items[i]:
                if min > remaining[j] - items[i]:
                    min = remaining[j] - items[i]
                    index = j
        if min < 1.1:
            remaining[index] -= items[i]
        else:
            remaining[bins] -= items[i]
            bins++
    return bins
```

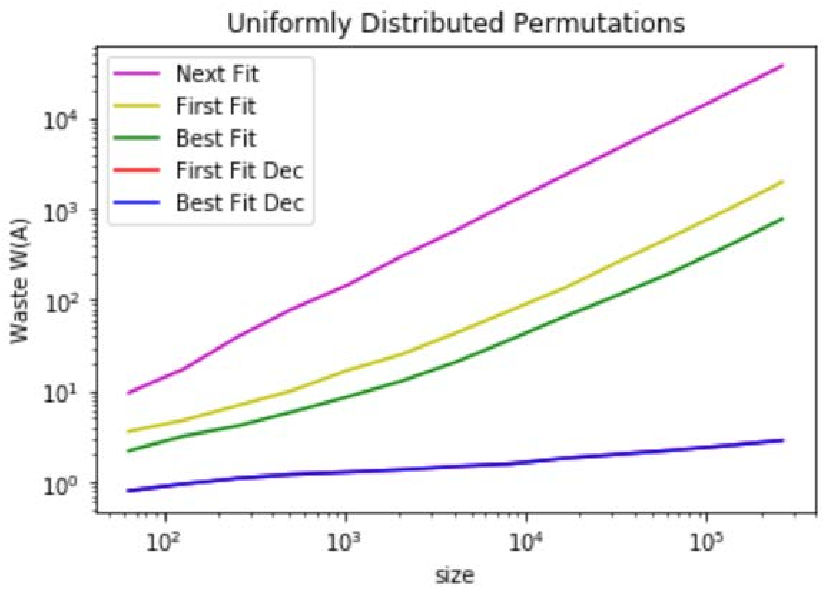
Time complexity: O(n^2), where n is the number of objects.

Uniformly Distributed Permutations:

We run the uniform distributed permutations on different sizes of the arrays and run with 5 different permutations for a particular size of array by randomly generating the list every time. We then take the average of 5 different permutations run for a particular size.

size = 64	Bins = 27	W(A) = 0.8
size = 128	Bins = 51	W(A) = 0.95
size = 256	Bins = 104	W(A) = 1.1
size = 512	Bins = 204	W(A) = 1.21
size = 1024	Bins = 407	W(A) = 1.28
size = 2048	Bins = 820	W(A) = 1.36
size = 4096	Bins = 1637	W(A) = 1.48
size = 8192	Bins = 3276	W(A) = 1.58
size = 16384	Bins = 6565	W(A) = 1.82
size = 32768	Bins = 13302	W(A) = 2.02
size = 65536	Bins = 26221	W(A) = 2.24
size = 131072	Bins = 52385	W(A) = 2.52
size = 262144	Bins = 105010	W(A) = 2.87

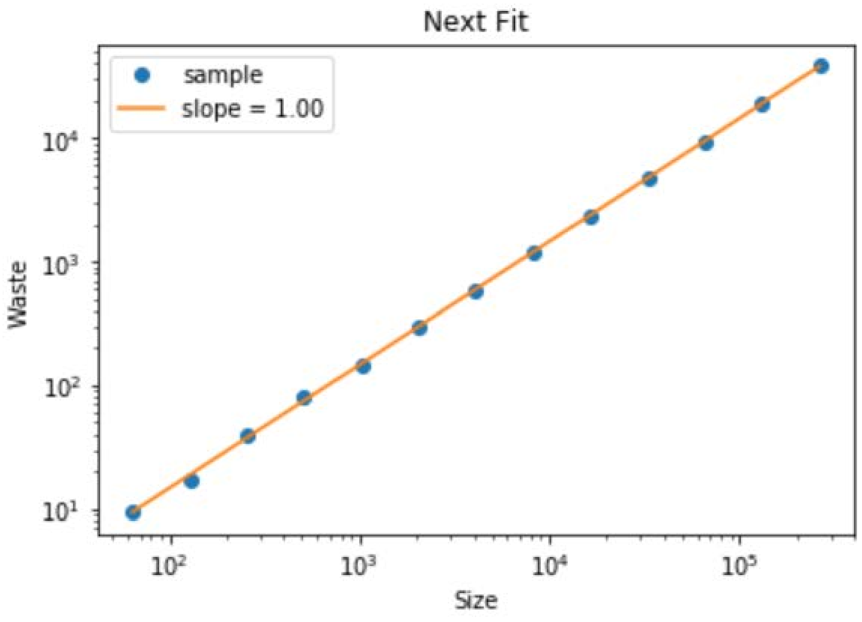
<u>Wastage Comparison:</u>



Here, the First Fit Decreasing and Best Fit Decreasing has the same waste factor for all the sizes.

<u>Observations:</u>

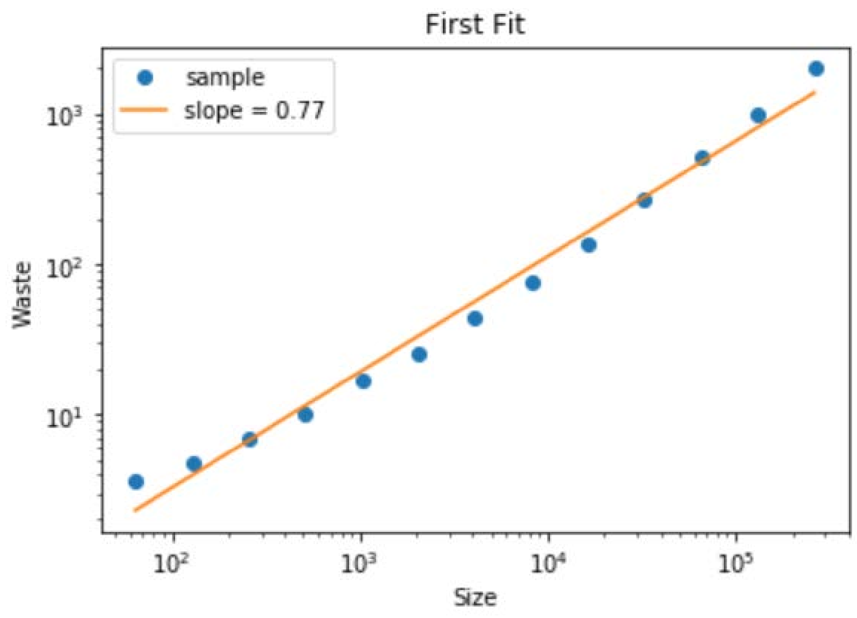
Next Fit Wastage Function:



Wastage = n^1

Here, the slope is 1 which means the wastage is proportional to n, where n is the number of objects.

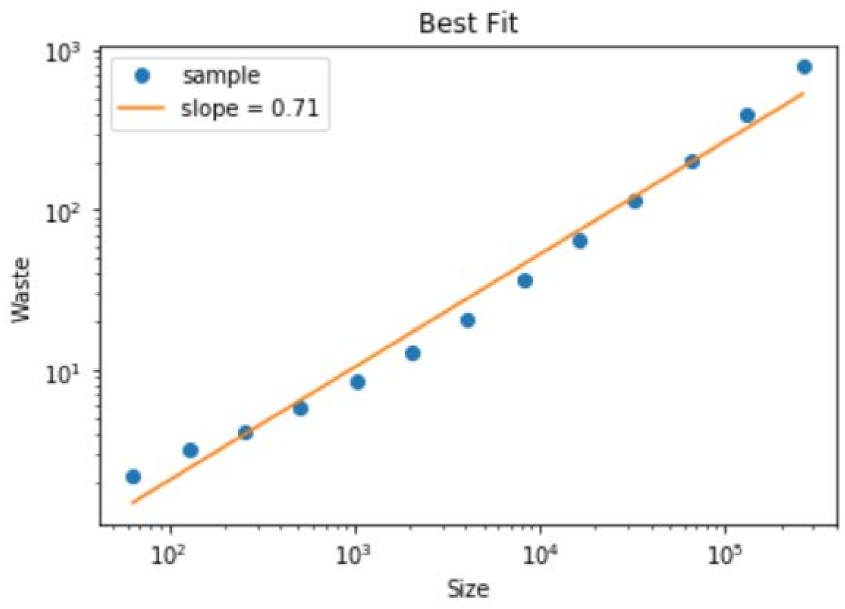
First Fit Wastage Function:



Wastage = $n^{0.77}$

Here, the slope is 0.77 which means the wastage is proportional to $n^{0.77}$, where n is the number of objects.

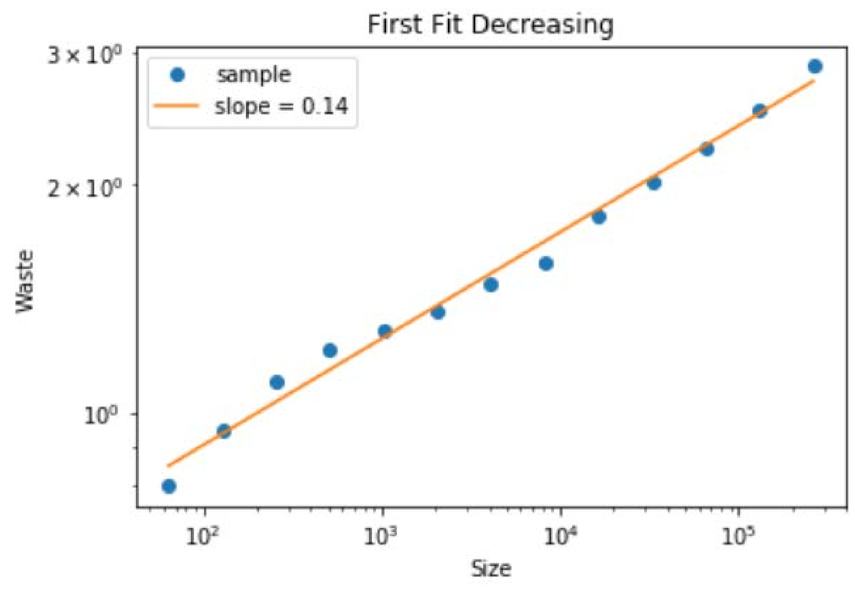
Best Fit Wastage Function:



Wastage = $n^{0.71}$

Here, the slope is 0.71 which means the wastage is proportional to $n^{0.71}$, where n is the number of objects.

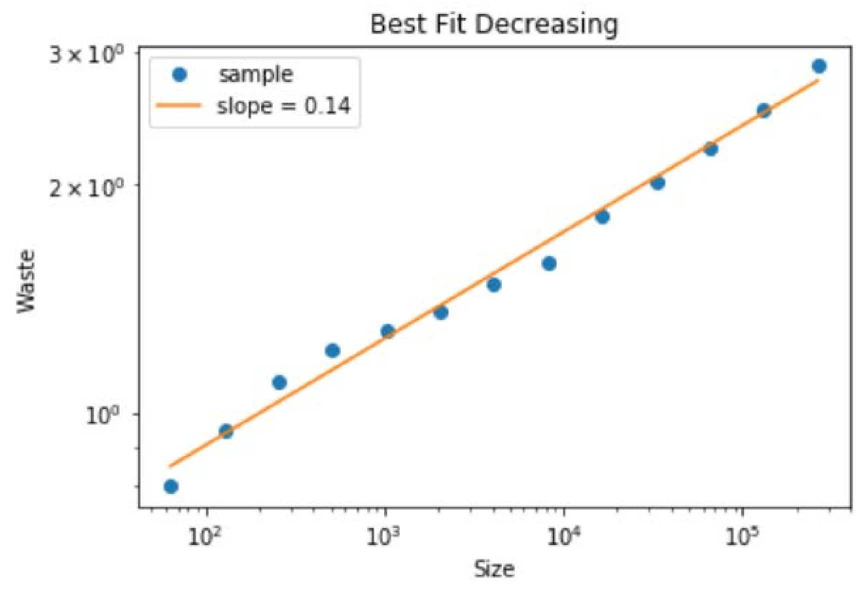
First Fit Decreasing Wastage Function:



Wastage = $n^{0.14}$

Here, the slope is 0.14 which means the wastage is proportional to $n^{0.14}$, where n is the number of objects.

Best Fit Decreasing Wastage Function:



Wastage = $n^{0.14}$

Here, the slope is 0.14 which means the wastage is proportional to $n^{0.14}$, where n is the number of objects.

Conclusion:

So, First Fit Decreasing and Best Fit Decreasing are the best bin-packing algorithms which have the lowest wastage function among the 5 bin-packing algorithms observed above. Best Fit and First Fit have almost similar wastage function but Best Fit slightly performs better than First Fit. Next Fit algorithm is worst among all and causes the maximum wastage.