Won Joon Lee
15647075
CS165
Due 5/20/19

Project 2 Report

Introduction:

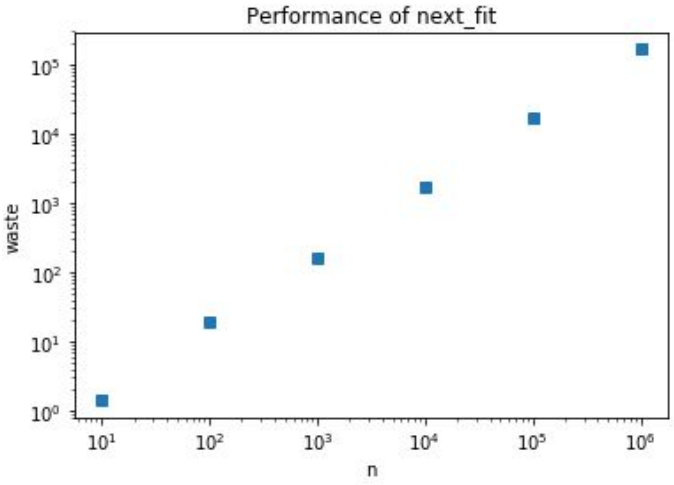
Inputs: For all of these bin packing algorithms, the input sizes used were 10^1 to 10^6, increased by a factor of 10 each time.

Note: all items (and free_space) are rounded to the seventh decimal place to avoid subtraction precision issues..

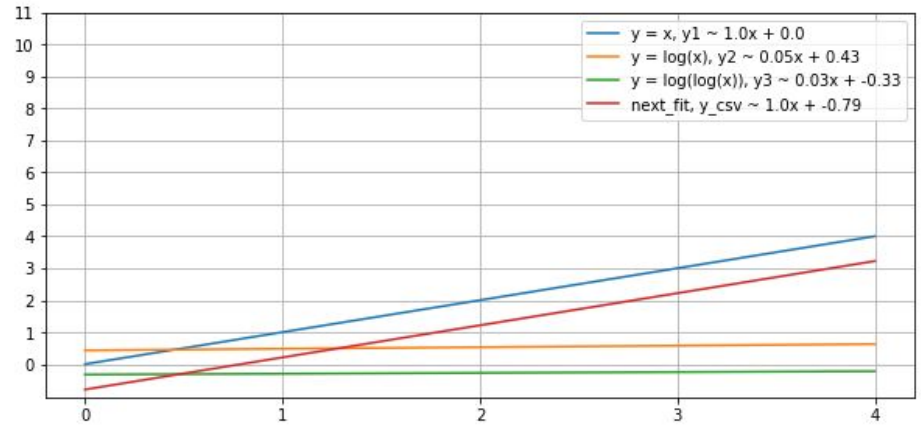
Algorithms:

Next Fit:

Data Graph:



Regression Graph:



Data:

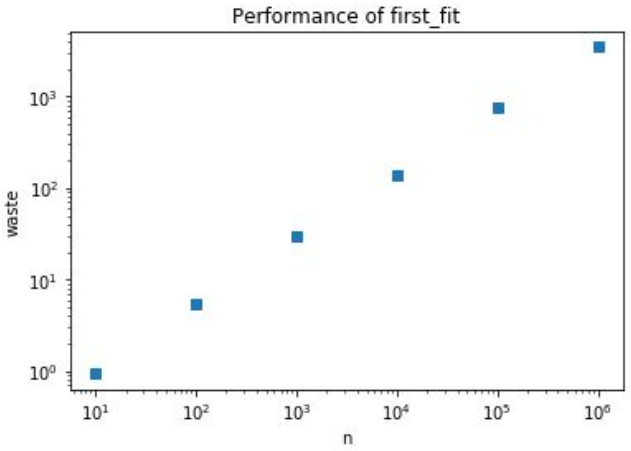
Size (n)	Waste (W)
10	1.42345
100	19.5135
1,000	163.325
10,000	1,681.39
100,000	16,690
1,000,000	166,737

Description:

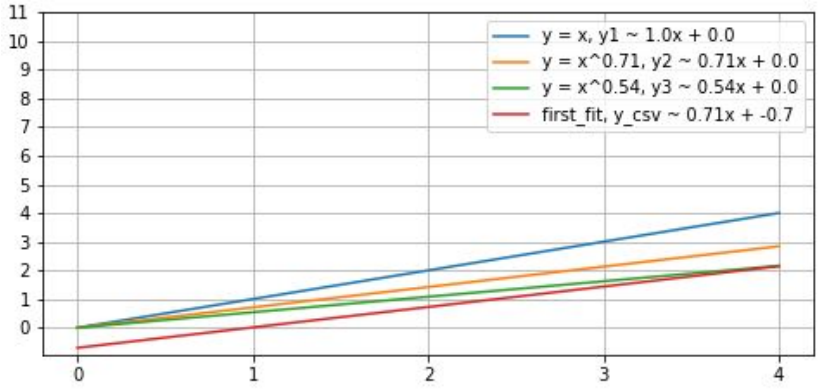
Next Fit does bin packing as follows: for each item, it checks if the last bin can fit the item. If it can, it puts it into the last bin. Else, it creates a new bin and puts the item in question in the new bin. Obviously, this algorithm is horrible for optimizing W, because it only checks if the last bin can fit the incoming item, rather than the entire set of bins. For example, if you have 2 bins with their free space [0.7, 0.5] (with the bin with 0.5 free space being the 'last' bin), and an item of size 0.6 is incoming, the Next Fit algorithm will only check whether the bin with free space of 0.5 can fit the item, which it can't, then it will make a new bin, when, optimally, the item should be placed in the first bin, thus increasing W. This is evident in the data, as for n = 1,000,000, W reached more than one-tenth of n (at 166,737). As for the regression graph, the slope is the same as y = x, so **W(A)** for Next Fit as a function of n would be **W(A) = n + C**.

First Fit:

Data Graph:



Regression Graph:



Data:

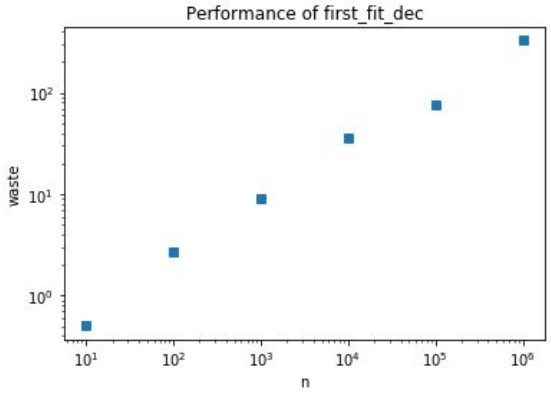
Size (n)	Waste (W)
10	0.94235
100	5.5265
1,000	29.4321
10,000	139.192
100,000	750.638
1,000,000	3,483.78

Description:

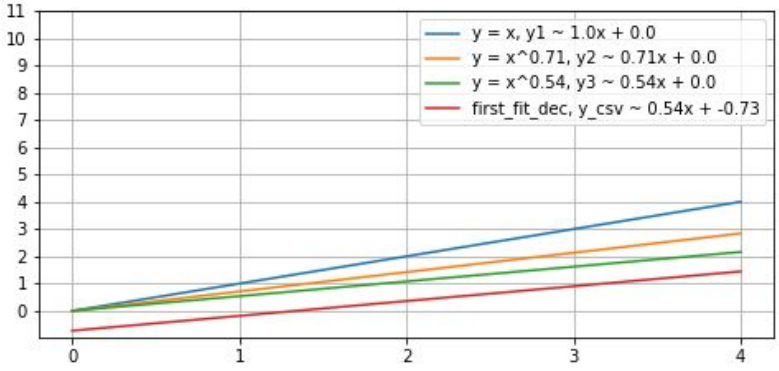
First Fit does just what its name suggests; for each item, it checks all the pre-existing bins and putting the item in the first bin that can fit it. Only when NO bins can fit the incoming item does it create a new bin and put the item there. For example, if you have free_space [0.3, 0.5, 0.4], and you are trying to put in an item of size 0.4, First Fit will put it into the bin with free space 0.5. However, the problem now is that it is still not optimized. Note that there was a bin (after the bin with free space 0.5 that the item was put into) with exactly 0.4 free space, yet First Fit chose to put the item into the first bin it found that can fit it. If another item of size 0.5 came after the first item, then First Fit will create a new bin to fit the incoming item because the bin with free space (originally) 0.5 now only has 0.1 free space, and none of the bins now ([0.3, 0.1, 0.4]) can fit an item of size 0.5. Still, it performs miles better than Next Fit, and it's evident because at n = 1,000,000, Next Fit has W= 166,737, whereas First Fit has W = 3483.78. According to the regression graph, **W(A)** for First Fit **= 0.71n + C.**

First Fit Decreasing:

Data Graph:



Regression Graph:



Data:

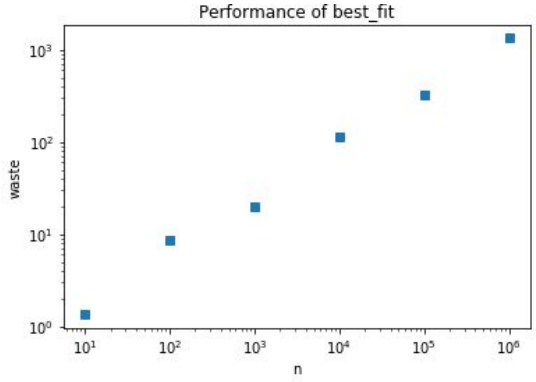
Size (n)	Waste (W)
10	0.50725
100	2.66225
1,000	9.1807
10,000	35.5469
100,000	76.9398
1,000,000	326.764

Description:

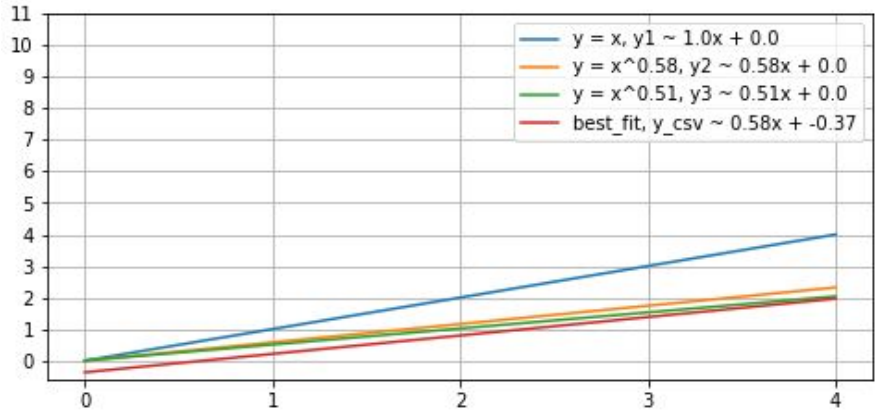
A lasting problem with First Fit (and by extension, Best Fit), is that larger items later down the line make it hard to optimize the Waste. For example, suppose there is an item list of [0.3, 0.2, 0.8, 0.7]. Following the logic of First Fit, it would put 0.3 and 0.2 in the same bin (free_space = [0.5]), then make a new bin for both 0.8 and 0.7, creating 2 bins. (the end result will be free_space = [0.5, 0.2, 0.3]). But we know that we could have done better; we could have fit the items in just two bins perfectly. (0.2 and 0.8, 0.3 and 0.7). To alleviate this issue, First Fit Decreasing first sorts the items in descending order, so that bigger items come first, making it less likely for such a situation to occur. For example, for the same list of items, First Fit Decreasing would sort the items in descending order, resulting in [0.8, 0.7, 0.3, 0.2]. Then, it would fit in the items using regular First Fit, and would end up with free_space = [0, 0]. Sorting before running First Fit proves to be effective, as for n = 1,000,000, First Fit gets W = 3,483.78, whereas First Fit Decreasing gets W = 326.764. **W(A) = 0.54n + C.**

Best Fit:

Data Graph:



Regression Graph:



Data:

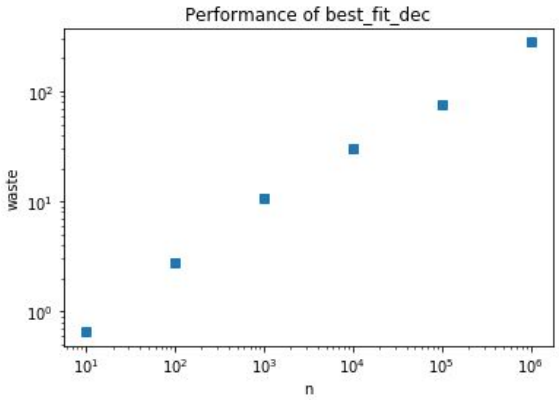
Size (n)	Waste (W)
10	1.3515
100	8.64665
1,000	19.5767
10,000	113.171
100,000	320.931
1,000,000	1,333.47

Description:

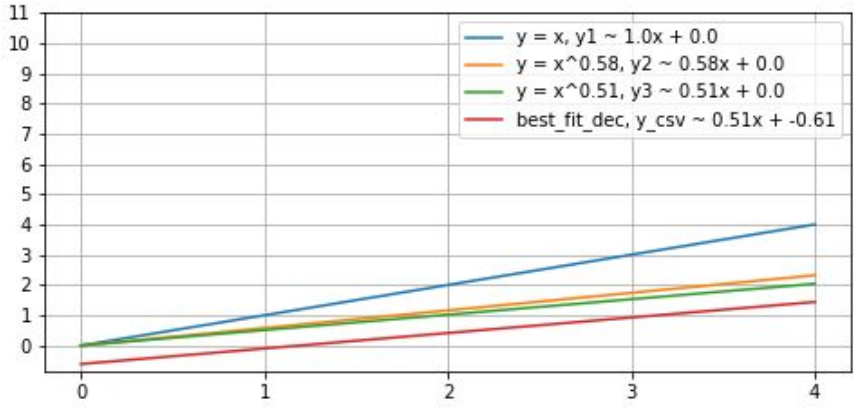
Best Fit is also self-explanatory in its name; for each item, Best Fit puts it into the bin that fits it ‘best’. That is, it will iterate through all the pre-existing bins, and put the item into the bin whose free_space is closest to the size of the item (if none of the bins fit, then it will make a new bin, of course). For example, given free_space = [0.3, 0.5, 0.4] (same as the example given in First Fit), when trying to put in an item of size 0.4. Best Fit, unlike First Fit, will choose the last bin, because it’s the ‘tightest’ bin that can fit the item. Best Fit makes sure that the bins will be used the most efficiently out of Next Fit, First Fit, and Best Fit, but still struggles with larger items towards the end of the item list, which is why we have Best Fit Decreasing. Performance-wise, it’s slightly better than First Fit, at W = 1,333.47 at n = 1,000,000 compared to First Fit’s 3,483.78. **W(A) = 0.58n + C**.

Best Fit Decreasing:

Data Graph:



Regression Graph:



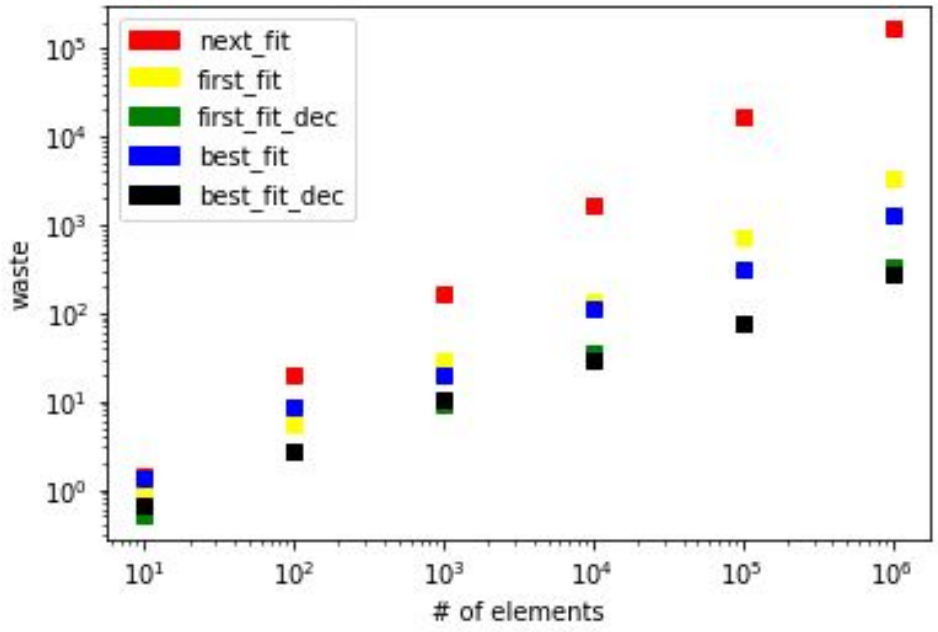
Data:

Size (n)	Waste (W)
10	0.6558
100	2.7664
1,000	10.5846
10,000	30.1232
100,000	76.7776
1,000,000	279.864

Description:

Best Fit Decreasing follows the same logic as First Fit Decreasing: Sort in reverse, then perform Best Fit rather than First Fit. Because Best Fit is essentially a better version of First Fit, it's safe to assume that Best Fit Decreasing would perform better than First Fit Decreasing. **W(A) = 0.51n + C**.

Compilation:



Comparing all 5 of the bin packing algorithms together, **Next Fit** definitely performs the worst in terms of W, as it creates too many unnecessary bins and does not use them optimally. Then, **First Fit** and **Best Fit** both perform better, (with **Best Fit** slightly performing better than **First Fit**, naturally), but still waste bins if larger elements come later in the sequence of items. **First Fit Dec** and **Best Fit Dec** fix this issue by sorting the array of items in descending order before running the respective algorithms, which is why those two perform better than their original versions for the cost of O (n log n) time to sort beforehand.

Conclusion:

All in all, the two algorithms that sort the items first in descending order, First Fit Decreasing and Best Fit Decreasing, seem to perform the best because they avoid the issue of a large item coming in later in the algorithm. Between First Fit and Best Fit, naturally Best Fit performs better. And, of course, Next Fit performs the worst. However, in terms of time

complexity, only Next Fit is O(n), while all the others are O(n^2), so we are sacrificing time for optimization, which, given the discrepancy of W between Next Fit and the other algorithms, seem worth.