My Project: Trying to Pack Bins

1. What I Tried to Do

1.1 The Problem

This project is about bin packing. You have a bunch of items, each with a size, and you want to put them into bins. All bins have the same size limit (capacity). The main goal is to use the fewest bins possible to hold all the items.

I worked on the 1D version, where items just have one size number. So, figure out the minimum number of bins needed.

1.2 Why Bother?

Bin packing is useful for stuff like:

- Loading things into trucks.
- Cutting materials without wasting too much.
- Storing files.

It saves space and money. People say it's a hard problem to solve perfectly, especially if you have lots of items, so usually, we just try to find a pretty good answer quickly.

2. How I Did It

2.1 My Basic Idea

I thought I could mix a few things:

1. **Start Simple**: Use some basic packing methods (like First Fit Decreasing, Best Fit Decreasing) first.
2. **Try Moving Things**: Use a "local search" to see if moving items between bins helps use fewer bins.
3. **Combine Bins**: If some bins are almost empty, maybe combine them.
4. **Try Random Orders**: Pack the items in different random orders and run the simple methods again, keeping the best result found.

I hoped combining these would work better than just one thing. I also needed to make sure it finished quickly enough.

2.2 The Code I Wrote (`CW_exp.py`)

My final code used a few main parts:

Simple Packing Methods:

- **First Fit Decreasing (FFD)**: Sort items big to small. Put each one in the first bin it fits in.
- **Best Fit Decreasing (BFD)**: Sort items big to small. Put each one in the bin it fits *most snugly* into.
- **Best Fit (BF)**: Like BFD, but don't sort the items first.

Trying to Improve:

- **Local Search (`local_search`)**: This function randomly picks two bins and tries moving an item from bin A to bin B. If it works and bin A becomes empty, great! One less bin. It does this a bunch of times but stops if it hasn't found an improvement for a while.
- **Post-Processing / Bin Merging (`post_process_optimization`)**: After packing, this checks if any two bins can be dumped together into one bin without going over the limit. I made it keep checking until it couldn't merge any more.

Putting It Together (`advanced_fit`):

- This function first runs FFD, BFD, and BF and picks the best result out of those three.
- Then, it runs `local_search` on that result to try and improve it.
- Finally, it runs `post_process_optimization` to merge any possible bins.

Adding Randomness (`random_search_fit`):

- This was the main function I called. It runs `advanced_fit` many times.
- Before each run, it shuffles the list of items in different ways:
  - Sometimes completely random shuffle.
  - Sometimes sort them and then shuffle small groups of items.
  - Sometimes sort them and then randomly swap some items that are next to each other.
  - Sometimes group items by size (like all items between 10-19, 20-29 etc.) and shuffle the groups and items within groups.
- It tries these different shuffles to hopefully find an order that lets `advanced_fit` get a really good result.

- **Important for speed**: It doesn't always run the same number of times. If there are lots of items (like > 500), it runs fewer times (maybe only 1000). If there are fewer items, it runs more times (up to 15000). This helps it finish faster on the big problems.
- It keeps track of the best solution (fewest bins) found across all the random tries.

Speeding Up Notes:

- In `local_search`, I tried to update the list directly instead of making full copies all the time.

3. What Happened (Results)

I ran my final code (`CW_exp.py`) on the 10 test problems. The marker script said it passed and got 30/30. Here are the results:

Instance	Items	Capacity	Bins I Used	Known Best	Notes
instance_1	50	100	52	52	Matched
instance_2	100	100	59	59	Matched
instance_3	200	100	24	24	Matched
instance_4	500	100	27	27	Matched
instance_5	1000	100	47	47	Matched
instance_6	50	150	49	49	Matched
instance_7	100	150	36	36	Matched
instance_8	200	150	52	52	Matched
instance_large_9	10000	100	417	417	Matched
instance_large_10	10000	150	375	375	Matched

It worked! My code got the best known result for all of them. The total time was **26.2 seconds**, which was really fast.

4. Thinking About It

4.1 Different Code Versions

I made a few different versions while working on this:

1. **Early Tries**: Some early versions were simpler but didn't get the best scores, maybe missing by one bin on some problems.
2. **Too Slow Tries**: I tried adding more complex things (like other algorithms I read about or doing the local search/merging *inside* the random loop more often), and sometimes they got the right answers but took way too long (> 300s).
3. **Final Code (`CW_exp.py`)**: The one that worked best combined the simple FFD/BFD/BF with local search and merging (`advanced_fit`), and then used the random search (`random_search_fit`) with different shuffling methods and the dynamic number of tries. This combination was fast enough *and* got the best scores.

The dynamic number of tries in `random_search_fit` (fewer tries for more items) was probably key to making it fast enough for instances 9 and 10.

4.2 What Seemed to Help in the Final Code?

- Using FFD and BFD as starting points inside `advanced_fit` seemed good.
- `local_search` helped clean things up by moving items.
- `post_process_optimization` (merging) caught some simple improvements at the end.
- The different shuffling methods in `random_search_fit` must have found good item orders for `advanced_fit` to work on.
- Adjusting the number of random tries based on the number of items was crucial for speed on the big problems.

4.3 Good vs. Bad

Good:

- It got the best scores (30/30).
- It was very fast (26.2 seconds total).
- It worked for small and very large (10,000 items) problems.
- The number of random tries changes automatically based on item count, which seems like a good general idea.

Bad:

- Randomness means the exact time or maybe even the result could change slightly if run again (though `random.seed(0)` helps make it repeatable).
- Maybe uses more memory than needed when shuffling lists?
- It's still a bit complex with all the different shuffling methods; maybe not all of them were necessary.

- Hard to be 100% sure *why* a particular random shuffle leads to the best answer.

5. The End

5.1 Summary

So, my final code (`CW_exp.py`) worked pretty well.

- It mixed simple packing methods (like FFD, BFD) with steps to improve the solution (moving items, merging bins).
- Trying random orders for the items helped find good results.
- Changing how many random tries it did based on the number of items made it fast enough, especially for the big problems.

It got the best possible score on all the test problems and finished quickly.

5.2 Ideas for Later (Future Work)

If I had more time, maybe I could try:

- **Making it Faster**: See if there are ways to speed up the shuffling or the local search part even more.
- **Trying Other Algorithms**: Maybe look into other packing algorithms I didn't use, like Genetic Algorithms, and see if they work better or faster.
- **2D Packing**: Try to make something that works for packing squares into a bigger square, not just lines.
- **Smarter Adjustments**: Improve how the code decides how many random tries to do, maybe make it adapt even better without hardcoding thresholds based on item count.

This project was a good challenge. I learned that combining different ideas and thinking about speed is important for these kinds of problems.