University of Nottingham
UK | CHINA | MALAYSIA

Coursework:
Optimizing Inventory Management with 1D Bin Packing

Yuyang ZHANG

20514470

scyyz26@nottingham.edu.cn

May 1, 2025

School of Computer Science
University of Nottingham Ningbo China

1 Introduction and Background

I do not use any AI in this Coursework.

The 1D Bin Packing Problem (1D BPP) is a combinatorial optimization problem that involves determining the most efficient way to pack a collection of items, each with a specified size, into a minimum number of uniform bins. The primary constraint is that the total size of the items assigned to any bin must not exceed the bin's predetermined capacity. The objective is to minimize the number of bins used while adhering to this capacity constraint (Martello & Toth, 1990).

The 1D Bin Packing Problem (1D BPP) plays a crucial role in optimizing inventory management, storage, and transportation by minimizing space wastage in warehouses and shipping containers. Delorme et al. (2016) underscore its significance in reducing material waste in manufacturing, such as in paper production, while Munien and Ezugwu (2021) highlight its broad industrial applications, including cutting, such as wood and glass, packing, such as transportation and warehousing, and supply chain management,such as vehicle/container loading, cutting stock, trim loss, packaging design, resource allocation, and project scheduling.

2 Methodology

2.1 Approach

To address the 1D BPP problem, my approach combines a random search with a heuristic algorithm, primarily an optimized genetic algorithm (GA). Genetic algorithms offer robust optimization by simulating natural selection through selection, crossover, and mutation processes, thereby exploring large solution spaces (Back, 1996). On top of GA, I leveraged the Greedy algorithm to enhance search efficiency (Michalewicz, 2013). When creating the population, there's 50% chance that the individuals are created using the greedy strategy(sort item size in descending order), and 50% chance of random arrangement. This hybrid approach effectively balances exploration and exploitation, leveraging both randomization and GA enhancements to deliver high-quality packing solutions that meet real-world inventory management needs.

2.2 Implementation

I implemented the solution in Python(CW.py). The script consumes instance configurations from CW_ins.json, which includes item sizes and fixed bin capacities. These instances are parsed through read_bin_packing_instances() and then passed to random_search_fit() where the items are first shuffled and passed to optimized_genetic_packing() which applies the genetic algorithm(creates a population

of size 100, assesses the fitness of the population(fewer bins = more fit), selects the elite individual items, and performs crossover and mutation to produce the next population) and returns a list of bins where each bin is a list of items. The solution is then compared to the current best solution, and if the current solution uses fewer bins, the current solution becomes the new best solution. We can customize the number of times we invoke optimized_genetic_packing() to get a solution(to optimize runtime, the iteration variable is currently set to 1). The final best solution is then parsed and saved in 20514470_yuyang_zhang.json.

3 Results and Analysis

3.1 Presentation of Results

The genetic algorithm I implemented obtained the following result on the given instances:

```
[scyyz26@CSLinux Desktop]$ python CW.py
Instance: instance_1
Bins Used:      52 (Time: 0.2552s)
Instance: instance_2
Bins Used:      59 (Time: 0.3046s)
Instance: instance_3
Bins Used:      24 (Time: 0.0742s)
Instance: instance_4
Bins Used:      27 (Time: 0.0843s)
Instance: instance_5
Bins Used:      47 (Time: 0.2162s)
Instance: instance_6
Bins Used:      49 (Time: 0.2416s)
Instance: instance_7
Bins Used:      36 (Time: 0.1169s)
Instance: instance_8
Bins Used:      52 (Time: 0.2486s)
Instance: instance_large_9
Bins Used:      417 (Time: 13.1149s)
Instance: instance_large_10
Bins Used:      373 (Time: 9.3788s)

--- Summary ---
Output saved to 20514470_yuyang_zhang.json
Total Used Bins: 1136
Total Execution Time: 24.0360s
```

Figure 1: *Results of the Bin Packing Algorithm*

The GA algorithm used a total of 1136 bins, **matching the best known results for all instances and surpassing *instance_large_10* by 2 bins**, which earned 3 bonus marks. The total run-time for all instances was 24.04 s, and the total scoring is 33/30. I also implemented five other traditional bin packing algorithms(Next-Fit, First-Fit, First-Fit Decreasing, Best-Fit, Worst-Fit). For the bin count, runtime, and scoring for each instance in each traditional algorithm, see the Appendix 1.

```
[scyyz26@CSLinux Desktop]$ python CW_marker.py
Instance: instance_1    Mark: 3         Bonus: 0        Bins used/Best known: 52/52
Instance: instance_2    Mark: 3         Bonus: 0        Bins used/Best known: 59/59
Instance: instance_3    Mark: 3         Bonus: 0        Bins used/Best known: 24/24
Instance: instance_4    Mark: 3         Bonus: 0        Bins used/Best known: 27/27
Instance: instance_5    Mark: 3         Bonus: 0        Bins used/Best known: 47/47
Instance: instance_6    Mark: 3         Bonus: 0        Bins used/Best known: 49/49
Instance: instance_7    Mark: 3         Bonus: 0        Bins used/Best known: 36/36
Instance: instance_8    Mark: 3         Bonus: 0        Bins used/Best known: 52/52
Instance: instance_large_9      Mark: 3         Bonus: 0        Bins used/Best known: 417/417
Instance: instance_large_10     Mark: 3         Bonus: 3        Bins used/Best known: 373/375

--- Summary ---
Total Bin: 1136
Run Time:  24.04 s
Bonus mark: 3
Total mark: 33 / 30
Passed
```

Figure 2: *Comparison with Best Known Result*

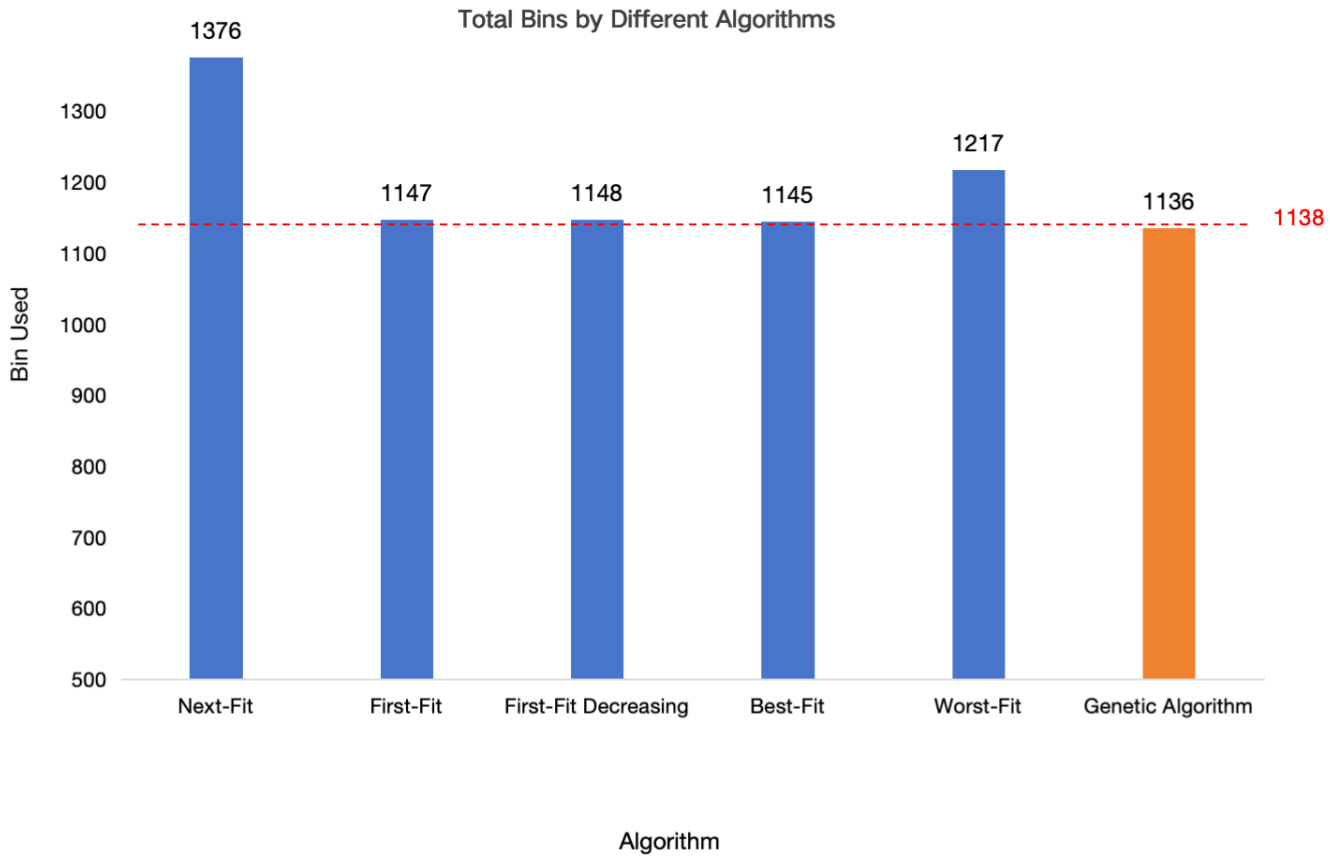

Figure 3: *Results of the Bin Packing Algorithm-Total Bins*

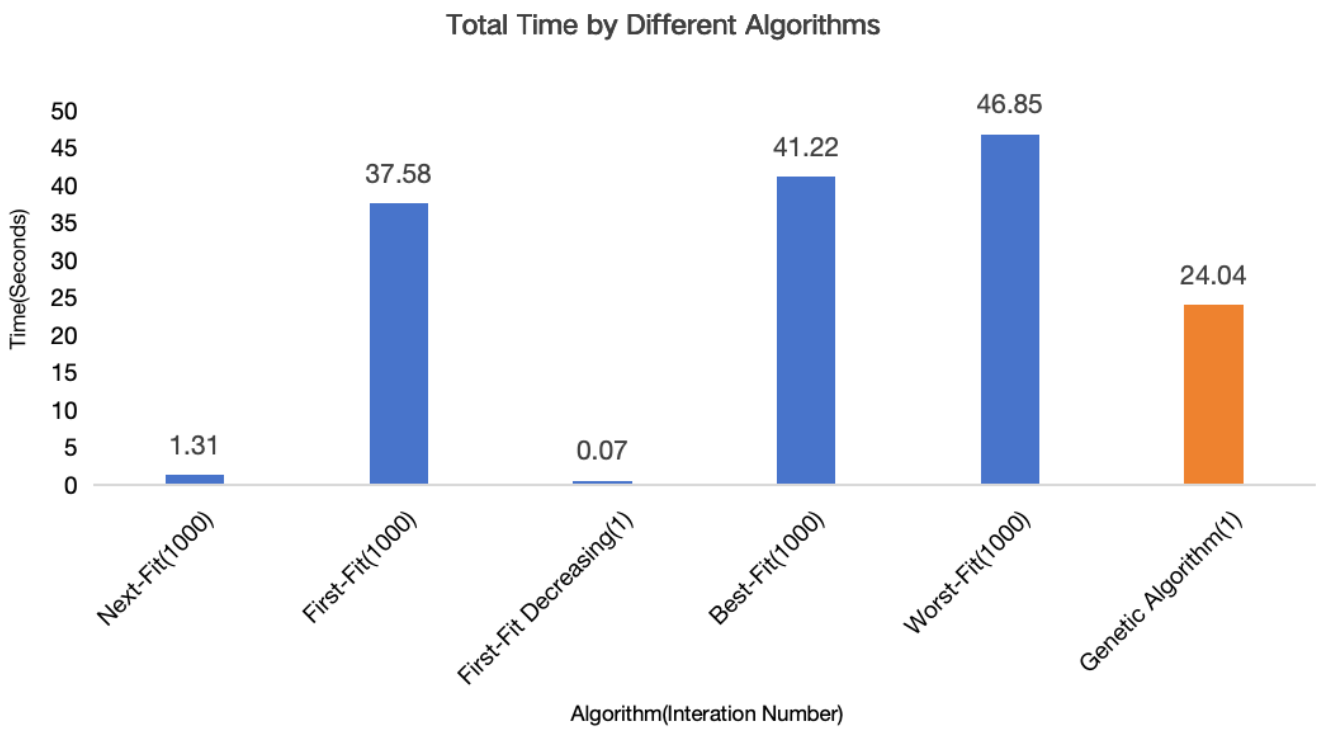

Figure 4: *Results of the Bin Packing Algorithm-Total Time*

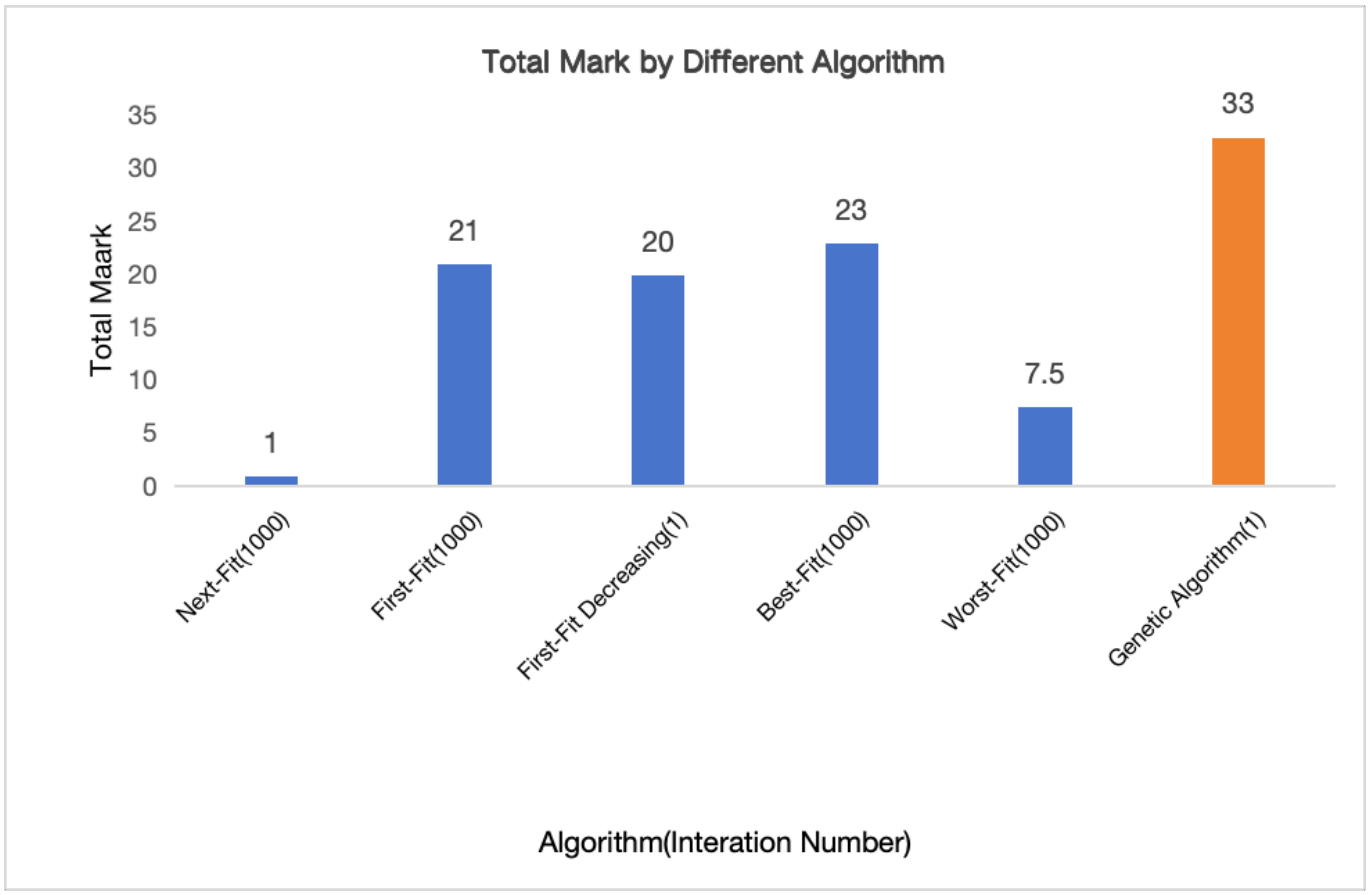

Figure 5: *Results of the Bin Packing Algorithm-Total Mark*

3.2 Critical Analysis

In this section I will analyze the bin count, runtime, and the pros and cons of the six algorithms I implemented. Each algorithm was executed for 1000 iterations using the *random_search_fit()* method to select the best solution, except First-Fit Decreasing and Genetic Algorithm, which were both executed for 1 iteration. For each iteration, the item order is randomly shuffled.

- **Next-Fit** This algorithm puts items into bins until the current bin is full, then moves on to the next empty bin.

  Pros: Next-fit is simple to implement and runs very quickly. It has a linear runtime ($O(n)$), with most instances running in less than 0.1 seconds, and instance_large_10 running in less than 0.5 seconds, making it the fastest algorithm.

  Cons: Next-fit generally performs the worst across all instances, especially for large datasets, for example, using 513/417 bins on *instance_large_9* and 461/375 bins on *instance_large_10*. Since there is no intelligence or estimation of the overall bin placement status, the bin utilization rate is far from ideal and should not be used for real-world applications.

- **First-Fit** This algorithm puts each item into the first bin that can hold it.

  Pros: Compared to Next-Fit, First-Fit utilizes space from previous bins that are not full, which achieves a better bin count result (1147 versus 1376 total bins).

  Cons: Because First-Fit needs to iterate bins from the beginning for each item, the runtime is close to quadratic ($O(n^2)$) and thus much slower than Next-Fit. Overall, First-Fit checks the status of other bins, but its performance can be drastically affected by the sequence of item sizes.

- **First-Fit Decreasing** This algorithm uses the same underlying logic as First-Fit, except the items are sorted in decreasing order(bigger item in the front) instead of being randomly shuffled.

  Pros: The algorithm is much faster (0.07s total runtime) than the unoptimized version, because it only requires 1 iteration since the item sequence is fixed in a particular order. Although sorting in Python takes about $O(nlog(n))$, it is still the fastest algorithm among all and is the only one that gives deterministic output.

  Cons: For each instance, there is still a gap(off by 1) with the Best known result. In real-world applications, however, First-Fit Decreasing might be one of the best solutions since it is the most cost-efficient in terms of runtime and result.

- **Best-Fit** This algorithm places items into the bin with the least free space. If there's no space in any of the current bins, a new bin is created.

  Pros: Best-Fit considers all current bin status and chooses the best one, which leads to overall

better results(second best scoring among six algorithms).

Cons: with 1,000 iterations, the runtime is much longer ($O(n^2)$, 41.22 s total) compared to Next-Fit and First-Fit Decreasing. Overall, this method can be considered if the user wants to chase better placement results and is okay with sacrificing runtime.

- **Worst-Fit** This algorithm is the opposite of Best-Fit, where it puts items into the bin with the most free space.

  Pros: The bin used result is slightly better than Next-Fit because it revisits previous bins that are not full.

  Cons: The bin count results get worse from the best known result, the more items the instance has. The runtime is also the longest among the six algorithms ($O(n^2)$). This is because a bin with the most free space is irrelevant to the size of the item, so it does not contribute to space optimization. It is very possible that a small item might be placed in a mostly empty bin. Overall, I would not recommend using this method for bin placement

- **Genetic Algorithm (My Approach)** This algorithm first creates a population of size 100, assesses the fitness of the population(fewer bins = more fit), selects the elite individual items, and performs crossover and mutation to produce the next population. When creating the population, there's 50% chance that the individuals are created using the greedy strategy(sort item size in descending order), and 50% chance of random arrangement.

  Pros: The algorithm leverages heuristics to have fine-grained intelligence on placement optimization. As a result, it yields the best bin count among all and a stable output, regardless of the number of iterations. This means we can execute just one iteration to shorten the runtime.

  Cons: The runtime is the second longest due to the potential sorting and repeated selection process, and the longest if we count by per iteration($O(G * P * n^2)$ where G is generation number($G = 6$), P is population size($P = 100$) and n is number of items). However, due to the higher intelligence of this approach, Genetic Algorithm's advantage will shine brighter when the number of items increases on a large scale.

4 Conclusion and Recommendations

4.1 Summary of Key Findings

By running multiple bin packing algorithms on the given instances, I found that the genetic algorithm gives the best result in terms of finding the least number of bins(Bin used/Best known: 1136/1138, runtime 24.04s), matching all best known results, and even surpassing the best known result for instance 10. For smaller instances (1-8), the genetic algorithm finishes in under 0.3 seconds for each, but for larger instances, the time increases to around 10 seconds. Nevertheless, the result is still far better than other traditional algorithms (next-fit, first-fit, first-fit decreasing, best-fit,

worst-fit), which not only cost similar or even more runtime but also result in a larger number of bins.

The only concern is that the runtime of the genetic algorithm appears to be increasing exponentially with the number of items, which is not ideal for real-world applications where the inventory may contain thousands or more items to sort through. One potential solution is for the user to break down the list of items into smaller groups and run the algorithm concurrently on each group. For example, instead of calculating bin placement for 10,000 items into 1,000 bins at once, split the items into 10 groups and calculate placing 1,000 items into 100 bins. Therefore, future work can focus on improving the execution efficiency of the algorithm on large-scale datasets and adopting more efficient optimization strategies.

4.2 Future Work and Recommendations

- **Algorithm optimization**

  Although the genetic algorithm yields optimal results, its execution efficiency needs to be further optimized for larger-scale instances. We can consider using parallel processing or more advanced caching strategies to reduce time complexity and improve the algorithm's scalability.

- **Hybrid Algorithm**

  In the future, genetic algorithms can be combined with simulated annealing to quickly explore the solution space using genetic algorithms and optimize local solutions through simulated annealing, thereby avoiding local optimality and improving algorithm efficiency and solution quality, especially when processing large-scale data.

- **Adaptive population size**

  In genetic algorithms, the population size has a significant impact on the convergence speed of the algorithm and the quality of the solution. Future work could explore methods for adaptive population size, dynamically adjusting the population size based on the algorithm's progress to optimize resource usage.

1 Appendix

```
[scyyz26@CSLinux Desktop]$ python next_fit.py
Instance: instance_1
Bins Used:      62 (Time: 0.0557s)
Instance: instance_2
Bins Used:      68 (Time: 0.0638s)
Instance: instance_3
Bins Used:      26 (Time: 0.0255s)
Instance: instance_4
Bins Used:      31 (Time: 0.0286s)
Instance: instance_5
Bins Used:      55 (Time: 0.0505s)
Instance: instance_6
Bins Used:      58 (Time: 0.0536s)
Instance: instance_7
Bins Used:      41 (Time: 0.0358s)
Instance: instance_8
Bins Used:      61 (Time: 0.0538s)
Instance: instance_large_9
Bins Used:      513 (Time: 0.4833s)
Instance: instance_large_10
Bins Used:      461 (Time: 0.4427s)

--- Summary ---
Output saved to 20514470_yuyang_zhang.json
Total Used Bins: 1376
Total Execution Time: 1.2938s
```

```
[scyyz26@CSLinux Desktop]$ python CW_marker.py
Instance: instance_1    Mark: 0         Bonus: 0        Bins used/Best known: 62/52
Instance: instance_2    Mark: 0         Bonus: 0        Bins used/Best known: 68/59
Instance: instance_3    Mark: 1         Bonus: 0        Bins used/Best known: 26/24
Instance: instance_4    Mark: 0         Bonus: 0        Bins used/Best known: 31/27
Instance: instance_5    Mark: 0         Bonus: 0        Bins used/Best known: 55/47
Instance: instance_6    Mark: 0         Bonus: 0        Bins used/Best known: 58/49
Instance: instance_7    Mark: 0         Bonus: 0        Bins used/Best known: 41/36
Instance: instance_8    Mark: 0         Bonus: 0        Bins used/Best known: 61/52
Instance: instance_large_9      Mark: 0         Bonus: 0        Bins used/Best known: 513/417
Instance: instance_large_10     Mark: 0         Bonus: 0        Bins used/Best known: 461/375

--- Summary ---
Total Bin: 1376
Run Time:  1.29 s
Bonus mark: 0
Total mark: 1 / 30
Passed
```

Figure 6: *The Result of Next-Fit*

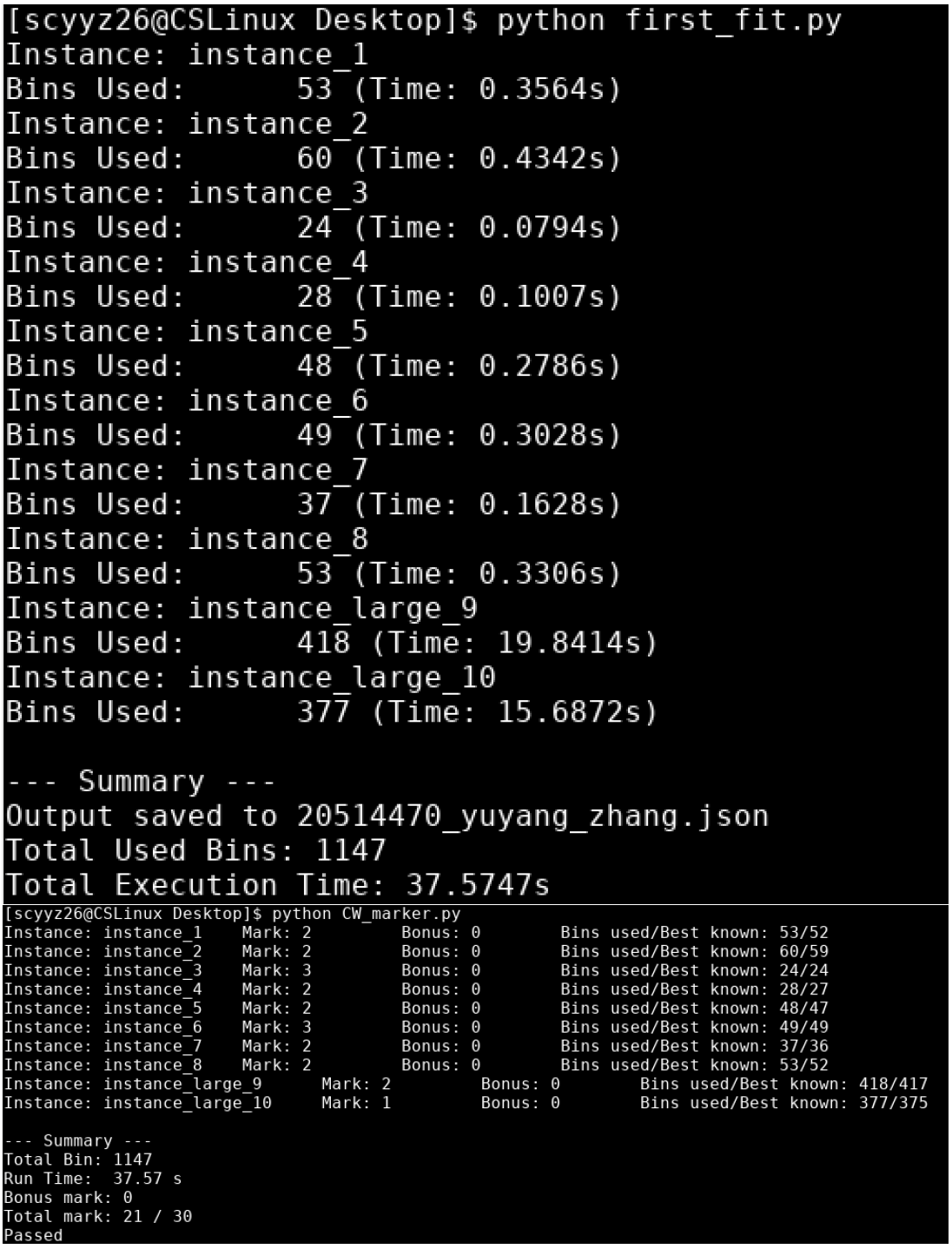

Figure 7: *The Result of First-Fit*

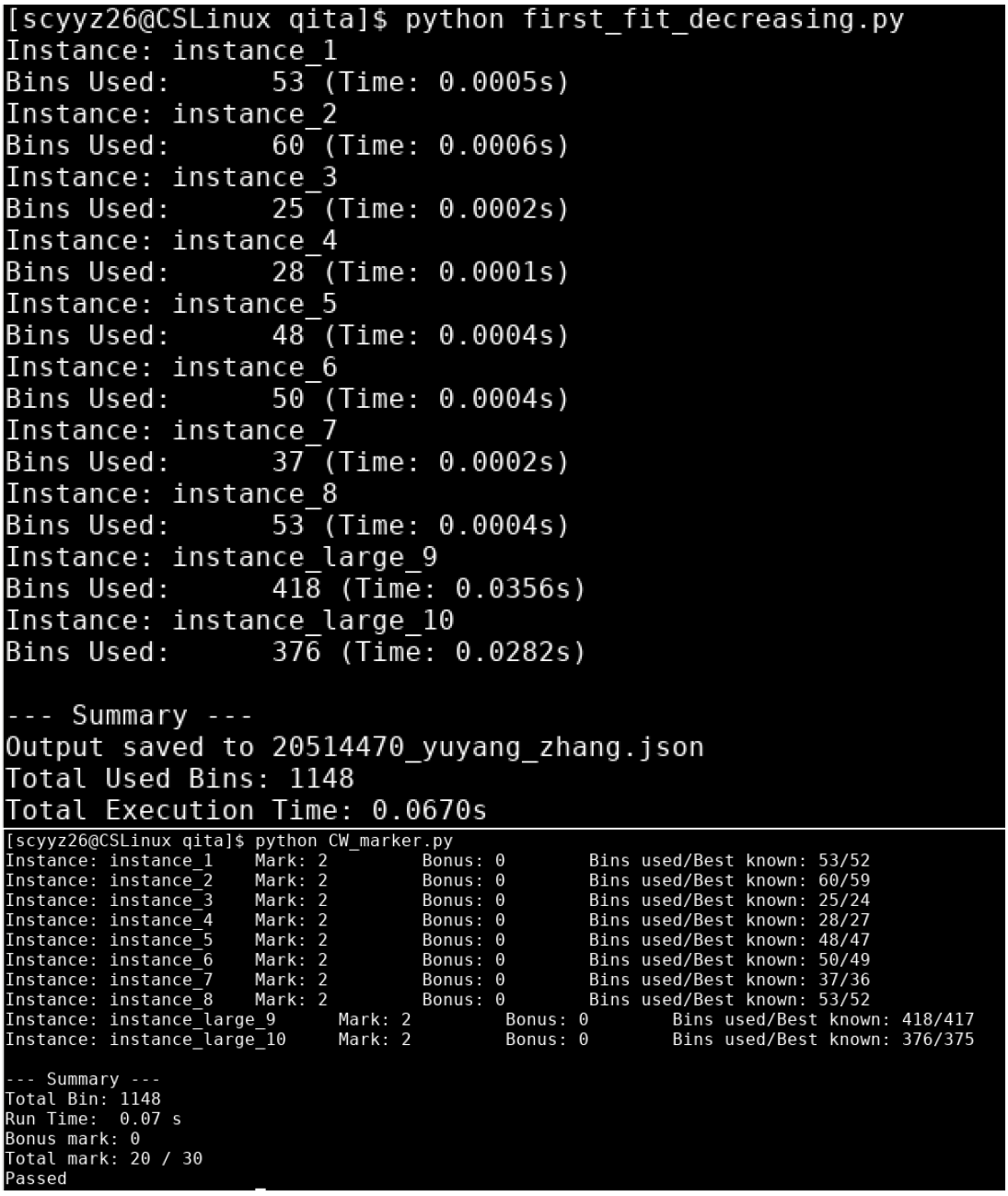

Figure 8: *The Result of First-Fit Decreasing*

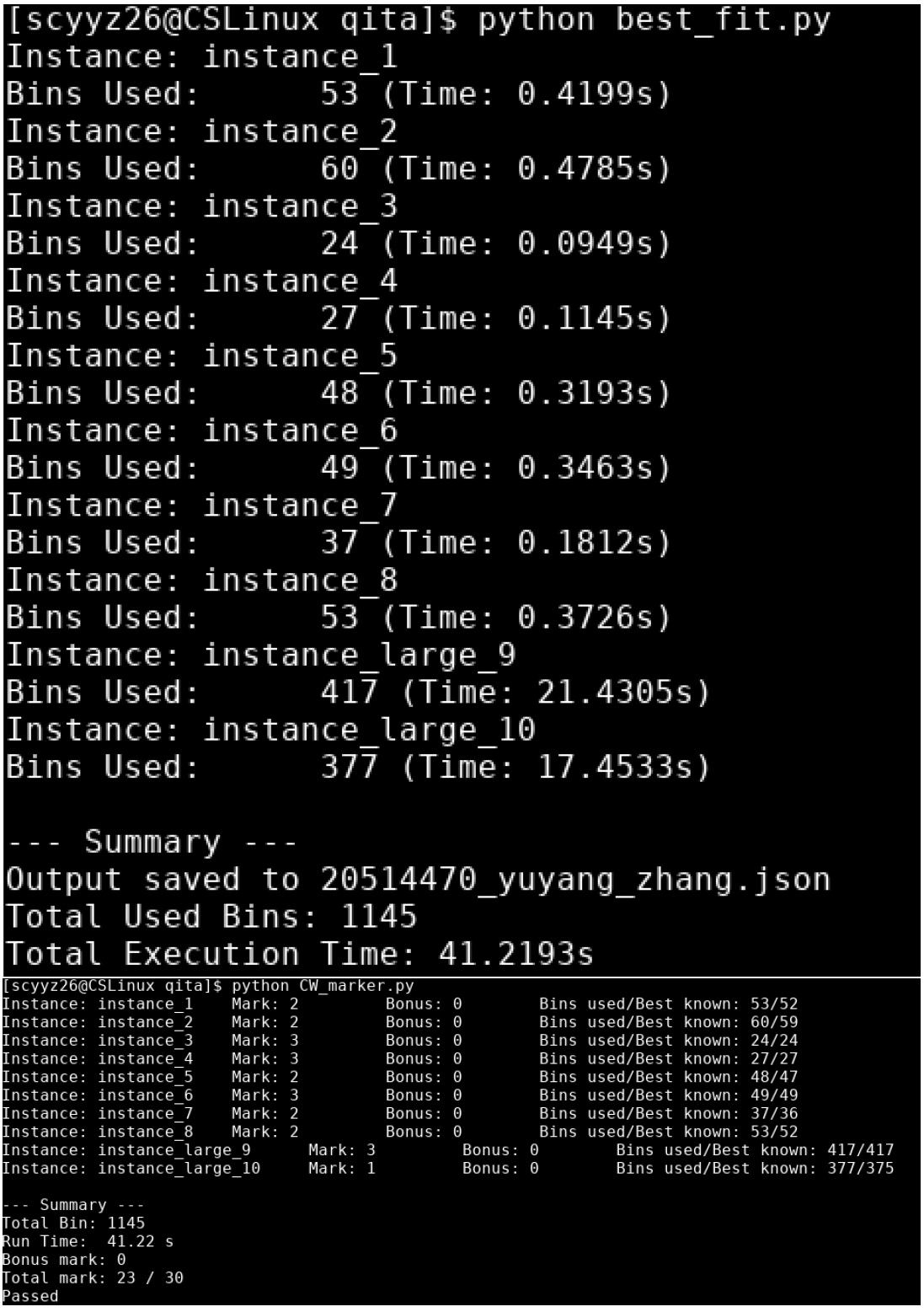

Figure 9: *The Result of Best-Fit*

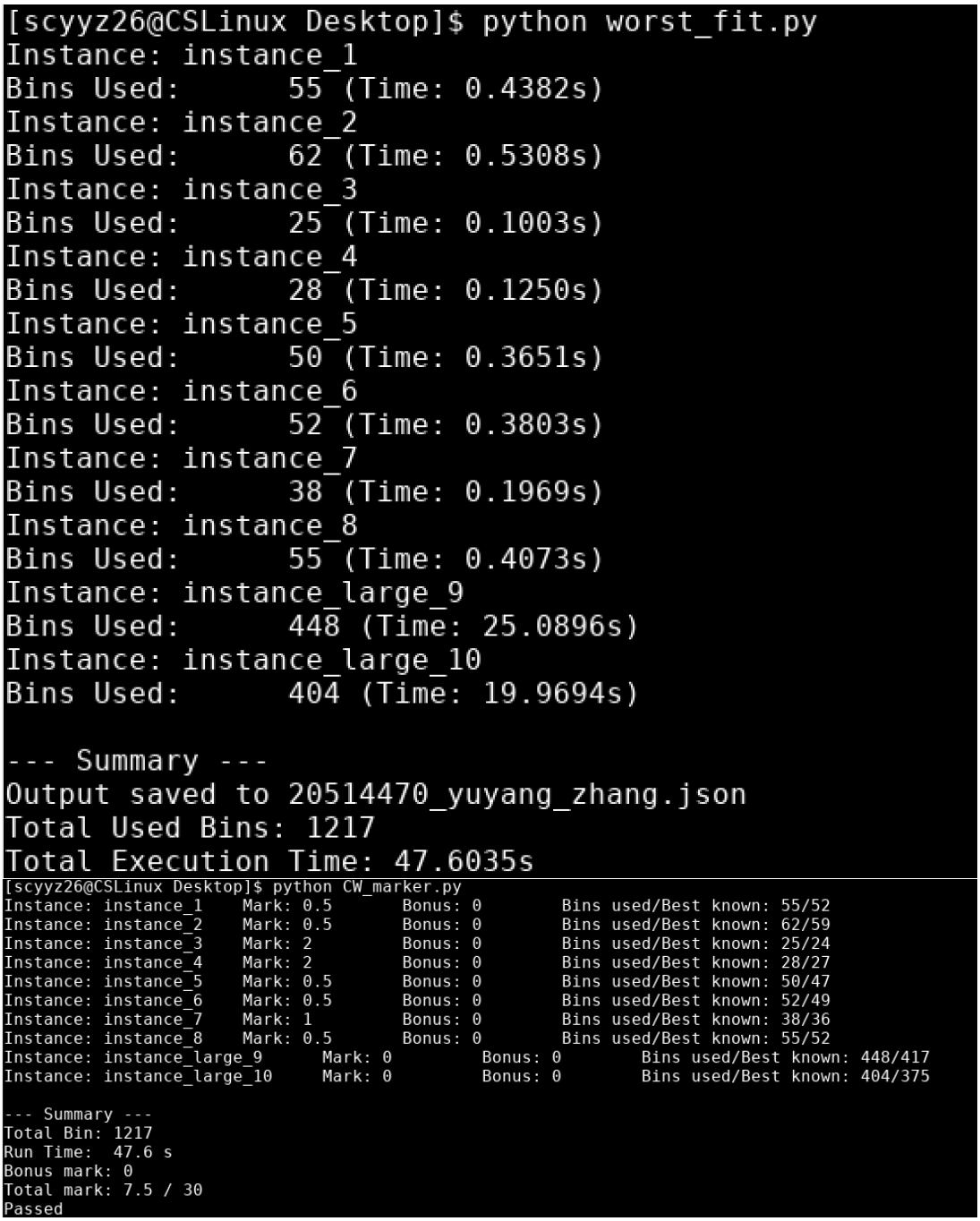

Figure 10: *The Result of Worst-Fit*

References

Back, T. (1996). *Evolutionary algorithms in theory and practice: Evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press.

Delorme, M., Iori, M., & Martello, S. (2016). Bin packing and cutting stock problems: Mathematical models and exact algorithms. *European Journal of Operational Research*, *255*(1), 1–20.

Martello, S., & Toth, P. (1990). Lower bounds and reduction procedures for the bin packing problem. *Discrete applied mathematics*, *28*(1), 59–70.

Michalewicz, Z. (2013). *Genetic algorithms+ data structures= evolution programs*. Springer Science & Business Media.

Munien, C., & Ezugwu, A. E. (2021). Metaheuristic algorithms for one-dimensional bin-packing problems: A survey of recent advances and applications. *Journal of Intelligent Systems*, *30*(1), 636–663.