INFO0902 - Data Structures and Algorithms
Project 2 - Bin Packing

DUY VU DINH (S2401627)

1 THEORETICAL ANALYSIS

1.1 Give a rough idea of the complexity of a solution to the problem by exhaustive search (as a function of N)

The off-line bin packing problem asks for the minimum number of fixed-capacity disks (bins) required to store N files, each i-th file with a known size s_i ($1 \leq i \leq N$ and $0 < s_i \leq B$), such that the sum of file sizes assigned to each disk does not exceed the disk capacity B. An exhaustive search strategy attempts to find the optimal solution by enumerating and evaluating all possible valid configurations of file-to-disk assignments.

The exhaustive search proceeds recursively: each file can be placed into any of the currently open disks (provided there is enough space), or into a newly created disk. This results in a branching decision tree:

- File 1: Must be placed in the first disk (a new bin). This gives 1 option.
- File 2: Can be placed in the first disk (if space allows), or in a new second disk. This gives at most 2 options.
- File 3: Can be placed in the first or second disk (if space allows), or in a new third disk. This gives at most 3 options.
- ...
- File N: Can be placed in any of the existing disks (if space allows) or in a new N^{th} disk. This results in up to N placement options.

In the worst case, each file introduces one more possible bin assignment, leading to a total number of configurations bounded by:

$$1 \times 2 \times 3 \times \cdots \times N = O(N!)$$

Therefore, the time complexity of a basic exhaustive search for the bin packing problem is in the order of $O(N!)$.

Alternatively, a looser upper bound for the number of configurations is $O(N^N)$, which assumes that each of the N files could be placed into any of the N possible bins without considering the recursive structure or capacity constraints. While this bound is more pessimistic, it still reflects the exponential nature of exhaustive search.

1.2 Prove, by giving a counterexample, that at least two of the four proposed greedy solutions are not optimal

Let $N = 12, \quad B = 100$

File list: $L = \{50,\ 3,\ 48,\ 53,\ 53,\ 4,\ 3,\ 41,\ 23,\ 20,\ 52,\ 49\}$

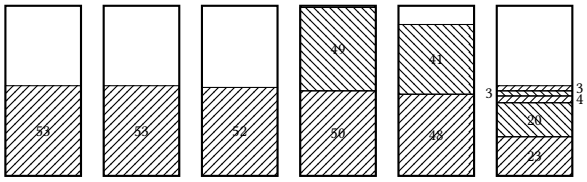
Sorted file list by size: $L_{sorted} = \{53,\ 53,\ 52,\ 50,\ 49,\ 48,\ 41,\ 23,\ 20,\ 4,\ 3,\ 3\}$

Figure 1 compares how different heuristics perform on this input. All bins have a fixed capacity of 100.

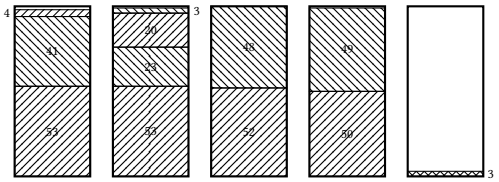
As shown in Figure 1:

- The Next-Fit algorithm uses 6 disks (Figure 1a).
- The First-Fit algorithm uses 5 disks (Figure 1b).
- The optimal solution uses only 4 disks (Figure 1c).

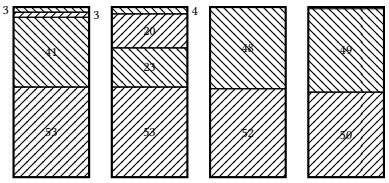
Therefore, both Next-Fit and First-Fit are not guaranteed to produce optimal results, since the use more disks than the optimal packing.



(a) Next-Fit packing: `{53}, {53}, {52}, {50, 49}, {48, 41}, {23, 20, 4, 3, 3}` - **6 disks**



(b) First-Fit packing: `{53, 41, 4}, {53, 23, 20, 3}, {52, 48}, {50, 49}, {3}` - **5 disks**



(c) Optimum packing: `{53, 41, 3, 3}, {53, 23, 20, 4}, {52, 48}, {50, 49}` - **4 disks**

Fig. 1. Packing results for Next-Fit (6 bins), First-Fit (5 bins), and Optimal solution (4 bins).

1.3 Give the pseudo-code of a function `Tree-Search-BF(T, k)` finding, in a binary search tree represented as in the lecture, the smallest key of the tree greater than or equal to k. Analyze its worst-case time complexity as a function of N

Pseudocode.

TREE_SEARCH_BF(T, k)

```
1  x = T.root.left // Start at the real root (ignore dummy root)
2  best = NIL
3  while x ≠ NIL
4      if x.space ≥ k
5          best = x // Candidate found, but continue to find a smaller one
6          x = x.left // Maybe even better on the left
7      else
8          x = x.right // Need larger space, go right
9  return best
```

In the implementation of the Best-Fit heuristic, an AVL tree is used in which the nodes are ordered by the amount of remaining free space on each disk. This ordering enables efficient search for the disk with the smallest free space that is still sufficient to store a given file.

`Tree-Search-BF(T, k)` returns a pointer to the node in a binary search tree that contains the smallest key greater than or equal to a given value k. In the context of the Best-Fit bin packing heuristic, the key corresponds to the `space` value of a disk node, and the binary search tree is structured as an AVL tree for balance.

The function follows a standard lower-bound search logic: starting at the root of the tree, it moves left whenever it encounters a value $\geq k$ (to find a smaller valid candidate), and moves right if the current node does not have enough space. The algorithm keeps track of the best candidate found so far and returns it at the end.

Worst-case time complexity. Assuming the binary search tree is an AVL tree (i.e., it is height-balanced), the height of the tree is bounded by $O(\log N)$, where N is the number of nodes in the tree. In the worst case, the search may traverse from root to leaf, examining at most one node per level. Therefore, the time complexity of `Tree-Search-BF` is:

$$O(\log N)$$

1.4 Give the pseudo-code of a function `Tree-Search-FF(T, size)` finding, in a binary search tree as represented in figure 2, the node corresponding to the first disk that can contain a file of size SIZE. Let `x` be a node. `x.key`, `x.space`, `x.maxspace` contain, respectively, the disk number, the free space on this disk, and the maximum space among the disks in the subtree rooted by `x`. Analyze its worst-case time complexity as a function of N

Pseudocode.

TREE-SEARCH-FF(T, $size$)

```
1  x = root of T
2  while x ≠ NIL
3      if x.left ≠ NIL and x.left.maxspace ≥ size
4          x = x.left // Go left if possible (prefer earlier disks)
5      elseif x.space ≥ size
6          return x // Current disk fits
7      elseif x.right ≠ NIL and x.right.maxspace ≥ size
8          x = x.right // Try right if it may help
9      else
10         return NIL // No disk found
11 return NIL
```

In the implementation of the First-Fit heuristic, an AVL tree is used in which the nodes are ordered by the disk index (insertion order). Each node is augmented with a `maxspace` field that stores the maximum free space among all disks in its subtree, allowing efficient pruning during the search for the first disk that can fit a given file.

`Tree-Search-FF(T, size)` returns a pointer to the node corresponding to the first disk with sufficient remaining space to store a file of size SIZE. The binary search tree is structured as an AVL tree and is augmented to support the First-Fit heuristic efficiently. Each node stores:

- `x.key`: the disk index (insertion order),
- `x.space`: the remaining free space on that disk,
- `x.maxspace`: the maximum free space value in the subtree rooted at `x`.

The function follows a modified in-order traversal logic: it first explores the left subtree if there is any chance a suitable disk exists there (i.e., `maxspace` is sufficient). If not, it checks the current node, and if still unsuccessful, it explores the right subtree under the same condition.

Worst-case time complexity. As in the Best-Fit case, assuming the binary search tree is height-balanced (AVL), its height is bounded by $O(\log N)$, where N is the number of disks stored. At each level, the algorithm performs constant-time checks and potentially one subtree descent. In the worst case, the function descends from root to leaf, resulting in:

$$O(\log N)$$

1.5 Analyze the worst-case time complexity of the four algorithms as a function of the number of files N, taking into account (and specifying in the report) your implementation choices

The worst-case time complexity of each bin packing strategy implemented in this project is analyzed below. The complexity is expressed as a function of N, the number of files to be packed. Let D denote the number of disks used during execution. In the worst-case scenario (e.g., one file per disk), we have $D = N$.

1.5.1 Next-Fit. In this strategy, a simple pointer is used to keep track of the currently open disk. Each file is either added to the current disk or causes the creation of a new disk. No sorting, searching, or traversal operations are required. As a result, each file is processed in constant time. Therefore, the worst-case complexity is: $O(N)$

1.5.2 Worst-Fit. A priority queue is used to store the disks, prioritized by their remaining free space. For each file:

- The disk with the most remaining free space is extracted: $O(\log D)$
- If the file fits, the disk is updated and reinserted into the heap: $O(\log D)$
- If it does not fit, a new disk is created and inserted: $O(\log D)$

Therefore, the worst-case complexity is: $O(N \log N)$

1.5.3 Best-Fit. The Best-Fit strategy is implemented using an AVL tree, where each node represents a disk and is ordered by its remaining free space. For each file:

- `Tree-Search-FF(T, size)` is used to find the disk with the smallest free space that can fit the file: $O(\log D)$
- The corresponding node is removed, the disk is updated, and then reinserted: $O(\log D)$ each

Therefore, the worst-case complexity is: $O(N \log N)$

1.5.4 First-Fit. An AVL tree ordered by disk index is used to implement the First-Fit strategy. Each node stores:

- `space`: the remaining free space on the disk,
- `maxspace`: the maximum remaining space within its subtree.

For each file:

- `Tree-Search-FF` is used to find the first disk that can store the file (pruning subtrees using `maxspace`): $O(\log D)$
- If a suitable disk is found, its `space` is updated and changes in `maxspace` are propagated: $O(\log D)$
- If no fit is found, a new disk node is inserted: $O(\log D)$

Therefore, the worst-case complexity is: $O(N \log N)$

2 EXPERIMENTS

2.1 Using the provided file `main.c`, plot the waste generated by each solution as a function of N on a graph. Waste is defined as the difference between the sum of the disk sizes used and the total file size. You can set the disk size to 1,000,000. Average curves for at least 5 experiments. Briefly discuss the differences between the methods

We evaluate the average waste generated by each of the four bin packing algorithms (Next-Fit, Worst-Fit, Best-Fit, and First-Fit) as a function of the number of files N. Waste is defined as the difference between the total disk space allocated and the total size of all files:

$$\text{Waste} = D \cdot B - \sum_{i=1}^{N} s_i$$

where D is the number of disks used, $B = 1{,}000{,}000$ is the fixed disk size, and s_i is the size of file i.

For each value of N, we ran 5 experiments with random file sizes and plotted the average waste. The values of N tested were: $N = \{100,\ 500,\ 1{,}000,\ 5{,}000,\ 10{,}000,\ 50{,}000,\ 100{,}000,\ 500{,}000,\ 1{,}000{,}000\}$

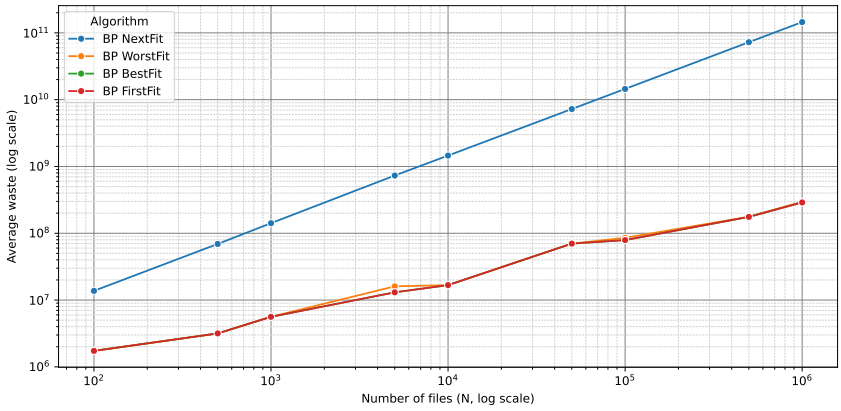


Fig. 2. Average waste over number of files for different algorithms (averaged over 5 experiments per N).

Figure 2 clearly shows that Next-Fit results in significantly more waste compared to the other three algorithms. Its waste grows rapidly and almost linearly with N, making it the least efficient strategy in terms of space utilization.

In contrast, Best-Fit, First-Fit, and Worst-Fit all produce much less waste. Best-Fit and First-Fit perform nearly identically at all scales, with indistinguishable curves. Worst-Fit performs similarly at small values of N, but shows slightly higher waste at larger scales (e.g., for $N = 10^6$).

The results indicate that more sophisticated heuristics, particularly Best-Fit and First-Fit, significantly reduce disk space waste compared to the simpler Next-Fit algorithm. Among the more efficient methods, Best-Fit and First-Fit offer comparable performance, while Worst-Fit lags slightly behind as N increases (Table A.1).

2.2 For the same experiments, also plot the average computation times as a function of N. Discuss the agreement between these curves and the theoretical analysis in the previous section

We evaluate the average CPU time required by each of the four bin packing algorithms, including Next-Fit, Worst-Fit, Best-Fit, and First-Fit, over the same set of experiments used in the waste analysis. Each data point represents the average time taken to process N files, with disk size fixed at 1,000,000.

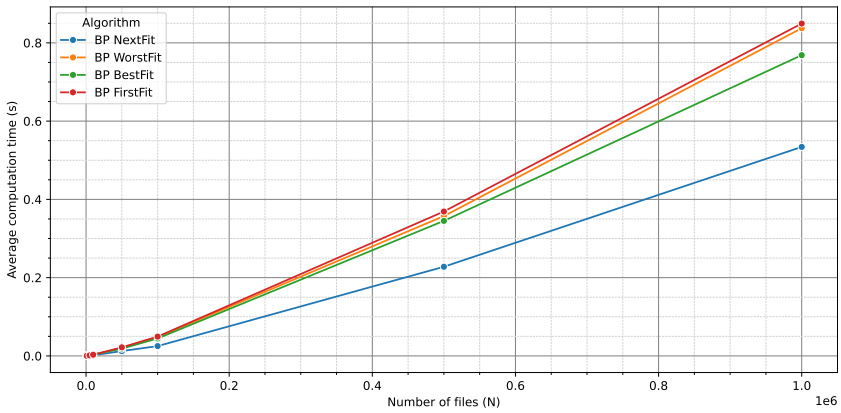


Fig. 3. Average computation time over number of files for different algorithms (averaged over 5 experiments per N).

Figure 3 shows that Next-Fit is by far the fastest algorithm, with computation time scaling linearly with N. For instance, at $N = 1{,}000{,}000$, Next-Fit completes in under 0.6 seconds. This is consistent with its theoretical time complexity of $O(N)$, since it performs no sorting or tree operations.

In contrast, the other three algorithms, including Worst-Fit, Best-Fit, and First-Fit, exhibit superlinear growth in computation time. This matches their theoretical $O(N \log N)$ complexity, due to the use of balanced AVL trees and priority queues for searching, inserting, and updating disk structures.

Among the three $O(N \log N)$ strategies, Best-Fit is slightly faster than First-Fit at higher values of N, likely due to the more involved tree updates required for First-Fit's augmented structure. Worst-Fit also performs competitively, though it is slightly slower than Best-Fit at moderate scales.

The experimental data closely matches the theoretical expectations. Next-Fit provides the fastest execution time with a simple linear scaling behavior, while Best-Fit, First-Fit, and Worst-Fit follow the predicted $O(N \log N)$ performance due to their reliance on ordered or priority-based data structures.

A APPENDIX: EXPERIMENT RESULTS

Table A.1. Experiment results.

Algorithm	Next-Fit		Worst-Fit		Best-Fit		First-Fit	
# files, N	Waste	Comp. time (s)	Waste	Comp. time (s)	Waste	Comp. time (s)	Waste	Comp. time (s)
100	13734912	1.22e-05	1734912	2.04e-05	1734912	2.74e-05	1734912	2.46e-05
500	69164807	6.06e-05	3164807	1.12-04	3164807	1.22e-04	3164807	1.178e-04
1000	141615153	1.312e-04	5615153	2.57e-04	5615153	2.60e-04	5615153	2.608e-04
5000	732051250	8.186e-04	16051250	0.0015	13051250	0.0015	13051250	0.0014
10000	1455735160	0.0016	16735160	0.0032	16735160	0.0032	16735160	0.0031
50000	7245006777	0.0124	70006777	0.0198	70006777	0.0187	70006777	0.0220
100000	14482434017	0.0251	85434017	0.0479	79034017	0.0446	79034017	0.0494
500000	72532546426	0.2278	176746426	0.3572	175946426	0.3451	175946426	0.3690
1000000	144991230860	0.5341	296630860	0.8374	289430860	0.7685	289430860	0.8493