A

CIA REPORT

ON

BIN PACKING PROBLEM

Submitted by,

29 Chaudhari Omkar

37 Deshmukh Shivraj

38 Deshmukh Tushar

52 Gore Abhishek

(TY BTECH COMPUTER)

[DIVISION A]

Guided by,

Prof. P.B. Dhanwate

Subject- Design and Analysis of Algorithms

In Academic Year 2024-25

Department of Computer Engineering

Sanjivani College of Engineering, Kopargaon

Sanjivani College of Engineering, Kopargaon

CERTIFICATE

This is to certify that

Chaudhari Omkar

Deshmukh Shivraj

Deshmukh Tushar

Gore Abhishek

(T.Y. Computer)

Successfully completed their CIA Report on

BIN PACKING PROBLEM

Towards the partial fulfilment of

Bachelor's Degree in Computer Engineering

During the academic year 2022-23

Prof. P.B.Dhanwate
[Guide]

Dr. D. B. KSHIRSAGAR
[H.O.D. Comp Engg]

Dr. A. G. THAKUR

[Director]

CONTENTS

Sr. No.	Chapter	Page No.
1.	Introduction	04
2.	Problem Statement & description	05
3.	Problem Pre requisite	06
4.	Requirement analysis of problem	08
5.	Solved example of given application	10
6.	Control abstraction of the Algorithm: Flowchart	14
7.	Explanation of algorithmic steps or code of entire program	15
8.	Implementation	18
9.	Time and space complexity of problem statement	20
10.	Conclusion	22
11.	References	23

1. Introduction

The **Bin Packing Problem** is a classic optimization problem in computer science and operations research. The problem entails allocating a set of in items with varying weights into bins of fixed capacity ccc, such that the total number of bins used is minimized. This problem finds applications in various real-world scenarios, such as resource allocation, storage management, and scheduling, where the efficient use of space or resources is a priority. The challenge lies in distributing the items optimally to minimize waste while adhering to the bin capacity constraints

A key assumption in the Bin Packing Problem is that the weight of each item is less than or equal to the capacity of a single bin. This guarantees that all items can be accommodated within the available bins, ensuring feasibility. The challenge lies in arranging the items strategically to optimize bin usage while satisfying the constraints.

A fundamental assumption in the problem is that each item's weight is smaller than or equal to the bin's capacity. This guarantees that every item can fit into at least one bin, making the problem solvable. Despite this, the problem's complexity arises from the exponential number of possible ways to allocate items into bins. As a result, the Bin Packing Problem is classified as NP-hard, meaning there is no known polynomial-time algorithm to find an exact solution for all instances.

2. Problem Statement & Description

Problem Statement:

Given n items of different weights and bins each of capacity c, assign each item to a bin such that number of total used bins is minimized. It may be assumed that all items have weights smaller than bin capacity.
Example:
Input: weight[] = {4, 8, 1, 4, 2, 1}
Bin Capacity c = 10 Output: 2

Description:

The given problem is a classic example of the **Bin Packing Problem**, where the task is to assign n items, each with a specified weight, to a minimum number of bins, each with a fixed capacity ccc. The objective is to minimize the number of bins used while ensuring that the total weight of items in any bin does not exceed the bin's capacity.

This problem models practical scenarios like packing goods into containers, managing memory in computer systems, or scheduling tasks with resource constraints. The challenge is to determine the optimal arrangement efficiently, especially for larger datasets, where brute force approaches become computationally infeasible. Various heuristic and exact algorithms, such as First-Fit, Best-Fit, or branch-and-bound, can be employed to tackle this problem effectively.

Problem Constraints:

1. **Each bin has a fixed capacity ccc, and no bin can exceed this capacity.**
2. **All item weights are smaller than or equal to the bin capacity, ensuring that every item can fit into at least one bin.**
3. **Items must be placed into bins in such a way that the number of bins used is minimized.**

3. Problem Prerequisite

- **Basic Understanding of Algorithms:**
  - **Greedy Algorithms**: Many solutions to the bin packing problem are based on greedy strategies, such as First-Fit or Best-Fit algorithms.
  - **Dynamic Programming**: Some approaches to solve variations of the bin packing problem use dynamic programming for optimal solutions.
  - **Backtracking**: Used in cases where exhaustive search is necessary.
- **Mathematical Foundations:**
  - Basic arithmetic and summation are used to calculate total capacities, item sizes, and determine fitment.
  - Understanding of inequalities to check constraints (e.g., ensuring the sum of weights in a bin does not exceed its capacity).
- **Combinatorics:**
  - Concepts of permutations and combinations are useful for understanding how items can be grouped or placed into bins.
- **Graph Theory (Optional):**
  - Some advanced formulations involve graph representations, where items and bins are treated as vertices and edges.
- **Problem-Specific Knowledge:**
  - **Weights and Capacities**: Familiarity with scenarios where weights/items are to be distributed across containers with limited capacity.
  - **Types of Bin Packing Problems**: Know the variations:
    - 1D Bin Packing: Items and bins have a single dimension (e.g., volume or weight).
    - 2D or 3D Bin Packing: Items and bins have multiple dimensions (e.g., width, height, depth).
    - Online vs. Offline Bin Packing: In the online version, items are placed as they arrive without prior knowledge.
- **Programming Skills:**
  - Knowledge of at least one programming language to implement algorithms (Python, C++, Java, etc.).
  - Familiarity with **data structures** like arrays, lists, heaps, and priority queues.
- **Optimization Techniques:**
  - Linear Programming (LP): Some formulations involve using LP to optimize the number of bins used.
  - Approximation Algorithms: Understand algorithms that provide near-optimal solutions within

guaranteed bounds.

- **Real-World Applications:**
  - Logistics and Transportation: Packing goods into containers.
  - Cloud Computing: Assigning tasks to servers with limited capacity.
  - Warehouse Management: Efficient space utilization.
- **Familiarity with Heuristics:**
  - Examples: First-Fit, Best-Fit, Worst-Fit, Next-Fit algorithms.
- **Understanding Problem Complexity:**
  - Bin Packing is **NP-Hard**; therefore, achieving an exact solution for large instances may be computationally infeasible. Approximation algorithms or heuristics are often used.

4. Requirement Analysis of Problem

Requirement analysis involves understanding the problem, identifying the objectives, constraints, inputs, and outputs, and determining the resources needed for the solution. Here's a breakdown for the **Bin Packing Problem**:

1.Problem Statement

- **Objective**: Minimize the number of bins required to pack a given set of items while ensuring that no bin exceeds its capacity.
- **Constraints**: Each bin has a fixed capacity, and the sum of item weights in any bin cannot exceed this capacity.
- **Variations**:
    - **1D Bin Packing**: Only the weight or volume of items is considered.
    - **2D/3D Bin Packing**: Items have dimensions, and spatial constraints must be managed.
    - **Online vs. Offline**: In online, items arrive sequentially, and decisions must be made immediately. In offline, all items are known beforehand.

2. Functional Requirements

- **Input**:
    - List of item sizes/weights (e.g., [w1, w2, ..., wn]).
    - Bin capacity (e.g., C).
- **Processing**:
    - Apply an algorithm to pack items into bins optimally (e.g., First-Fit, Best-Fit, Dynamic Programming).
    - Ensure the sum of item weights in each bin does not exceed the capacity.
- **Output**:
    - Number of bins used.
    - The assignment of items to bins.

3. Non-Functional Requirements

- **Performance**:
    - Should handle large datasets efficiently.
    - Approximation algorithms may be required for scalability since the problem is NP-Hard.

- **Scalability**:
  - Should scale with the number of items and the bin capacity.
- **Robustness**:
  - Handle edge cases such as no items, items too large for any bin, or all items fitting into one bin.
- **Usability**:
  - Easy to configure for different types of inputs and bin capacities.
- **Accuracy**:
  - For exact solutions, ensure correctness; for approximate solutions, provide a guaranteed bound.

5. Solved example of given application

Let's solve the **bin packing problem** for the given input step by step using the **First-Fit Algorithm** (one of the simplest and most commonly used approaches).

- **First-Fit Approach**

Input:

- **Weights:** weight[]={4,8,1,4,2,1
- **Bin Capacity (c)**: 10

First-Fit Approach

Steps:

1. Create bins as needed to accommodate the items.
2. Place each item in the first bin that has enough remaining capacity.
3. If no existing bin can accommodate the item, create a new bin.

Solution Process

Step 1: Initialize empty bins

We start with no bins. As we process the weights, we will create bins dynamically.

Step 2: Process each weight

- **Weight = 4**:
    - There are no bins yet, so create **Bin 1** and place 4 in it.
        - **Bin 1:** [4] (remaining capacity = 10−4=6)
- **Weight = 8**:
    - 8>68 ,so it cannot fit into **Bin 1**. Create **Bin 2** and place 8 in it.
        - **Bin 1:** [4] (remaining capacity = 6)
        - **Bin 2:** [8] (remaining capacity = 10−8=2)
- **Weight = 1**:
    - 1≤61 , so place it in **Bin 1**.
        - **Bin 1:** [4, 1] (remaining capacity = 6−1=5)
        - **Bin 2:** [8] (remaining capacity = 2)
- **Weight = 4**:

- $4 \leq 54$, so place it in **Bin 1**.
  - **Bin 1:** [4, 1, 4] (remaining capacity = $5-4=1$)
  - **Bin 2:** [8] (remaining capacity = 2)

- **Weight = 2**:
  - $2 > 12$, so it cannot fit into **Bin 1**.
  - $2 \leq 22$, so place it in **Bin 2**.
    - **Bin 1:** [4, 1, 4] (remaining capacity = 1)
    - **Bin 2:** [8, 2] (remaining capacity = $2-2=0$)
- **Weight = 1**:
  - $1 \leq 11$, so place it in **Bin 1**.
    - **Bin 1:** [4, 1, 4, 1] (remaining capacity = $1-1=0$)
    - **Bin 2:** [8, 2] (remaining capacity = 0)

Step 3: Final Result

- Total bins used: **2**
- Bin configurations:
  - **Bin 1:** [4, 1, 4, 1]
  - **Bin 2:** [8, 2]

Output:

The minimum number of bins required is **2**.

➢ **Best-Fit Algorithm**.

This algorithm places each item in the bin with the *least remaining capacity* that can still accommodate the item.

Input:

- **Weights:** weight[]={4,8,1,4,2,1
- **Bin Capacity (c)**: 10

Algorithm: Best-Fit Approach

Steps:

1. Create bins as needed to accommodate the items.
2. For each item, check all bins and choose the one with the **least remaining capacity** that can still

fit the item.

3. If no existing bin can accommodate the item, create a new bin.

Solution Process

Step 1: Initialize empty bins

We start with no bins. As we process the weights, we will create bins dynamically.

Step 2: Process each weight

- **Weight = 4**:
  - There are no bins yet, so create **Bin 1** and place 4 in it.
    - **Bin 1:** [4] (remaining capacity = 10−4=6)
- **Weight = 8**:
  - 8>68, so it cannot fit into **Bin 1**. Create **Bin 2** and place 8 in it.
    - **Bin 1:** [4] (remaining capacity = 6)
    - **Bin 2:** [8] (remaining capacity = 10−8=2)
- **Weight = 1**:
  - 1≤61 (Bin 1's remaining capacity)
  - 1≤21 (Bin 2's remaining capacity), but Bin 2 has **less remaining capacity**, so place 1 in **Bin 2**.
    - **Bin 1:** [4] (remaining capacity = 6)
    - **Bin 2:** [8, 1] (remaining capacity = 2−1=1)
- **Weight = 4**:
  - 4≤64 (Bin 1's remaining capacity)
  - 4>14 > 14>1 (Bin 2's remaining capacity), so place 4 in **Bin 1**.
    - **Bin 1:** [4, 4] (remaining capacity = 6−4=2)
    - **Bin 2:** [8, 1] (remaining capacity = 1)
- **Weight = 2**:
  - 2≤22 (Bin 1's remaining capacity)
  - 2>12 (Bin 2's remaining capacity), so place 222 in **Bin 1**.
    - **Bin 1:** [4, 4, 2] (remaining capacity = 2−2=0)
    - **Bin 2:** [8, 1] (remaining capacity = 1)
- **Weight = 1**:
  - 1>01 (Bin 1's remaining capacity)
  - 1≤11 (Bin 2's remaining capacity), so place 111 in **Bin 2**.

- **Bin 1:** [4, 4, 2] (remaining capacity = 0)
- **Bin 2:** [8, 1, 1] (remaining capacity = 1−1=0)

Step 3: Final Result

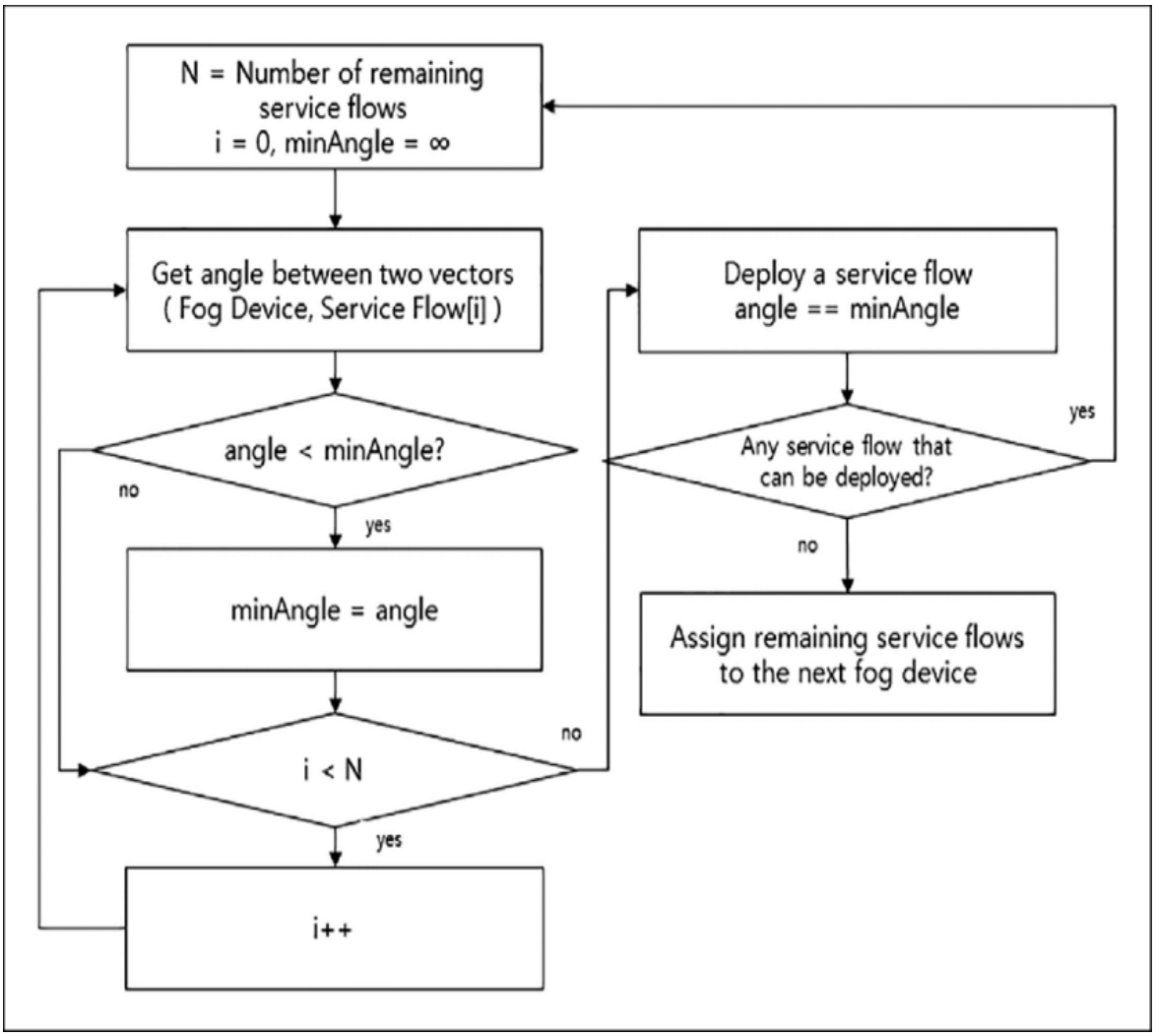
- Total bins used: **2**
- Bin configurations:
  - **Bin 1:** [4, 4, 2]
  - **Bin 2:** [8, 1, 1]

Output:

The minimum number of bins required is **2**.

6. Control abstraction of the Algorithm:

Flowchart for the problem statement:



7. Explanation of algorithmic steps or code of entire program

Algorithm for the considered problem statement is as follows:

1. **Initialize**:
    - Create an empty list bins [] to store the bins and their remaining capacities.
    - Each bin is represented as an object (or array) with:
        - remaining_capacity: The remaining space in the bin.
        - items[]: The items placed in the bin.

2. **Iterate Through Each Item**:
    - For each weight in weights[]:
        1. **Find the Best Bin**:
            - Check all bins in bins[] to find the one where:
                - The remaining capacity is large enough to accommodate the weight.
                - The remaining capacity after adding the weight is minimized.
        2. **Decision**:
            - **If a Suitable Bin is Found**:
                - Place the weight in the selected bin.
                - Update the remaining_capacity of the selected bin.
            - **Otherwise**:
                - Create a new bin.
                - Add the weight to the new bin.
                - Append the new bin to bins[].

3. **Output the Result**:
    - After processing all items, the total number of bins is the size of bins[].
    - Display the contents and remaining capacities of each bin.

- **Pseudocode**

```
FUNCTION BestFit(weights[], capacity):
  INITIALIZE bins[] AS empty list
```

```
FOR each weight IN weights[]:
    best_bin_index = -1
    best_fit = INFINITY

    FOR i IN range(0, length(bins)):
        IF bins[i].remaining_capacity >= weight:
            fit = bins[i].remaining_capacity - weight
            IF fit < best_fit:
                best_fit = fit
                best_bin_index = i

    IF best_bin_index == -1:
        # No suitable bin found, create a new bin
        new_bin = {
            remaining_capacity: capacity - weight,
            items: [weight]
        }
        ADD new_bin TO bins
    ELSE:
        # Place the item in the best-fit bin
        bins[best_bin_index].items.ADD(weight)
        bins[best_bin_index].remaining_capacity -= weight

RETURN length(bins), bins
```

1. **Initialization**:

- bins[] is an empty list that will store all bins and their details.
- Each bin is represented as an object with properties for tracking its remaining_capacity and the items[] it contains.

2. **Finding the Best Bin**:

- For each item (weight), loop through all bins to find the one with the smallest remaining capacity that can still accommodate the item.
- Use the variable best_bin_index to track the index of the selected bin and best_fit to track the smallest remaining capacity after placing the item.

3. **Decision**:

- If no suitable bin is found (best_bin_index == -1), create a new bin and add the item to it.
- Otherwise, place the item in the selected bin and update the bin's remaining_capacity.

4. **Output**:

- The algorithm returns the total number of bins used (length(bins)) and the details of all bins (items and capacities).

8. Implementation

```java
package Recursion.Leetcode;
import java.util.ArrayList;
import java.util.Collections;
public class BinPacking {

    // Method to implement First Fit Decreasing (FFD) algorithm
    public static ArrayList<ArrayList<Integer>> firstFitDecreasing(int[] items, int binCapacity) {  1 usage
        // Step 1: Sort items in descending order
        ArrayList<Integer> itemList = new ArrayList<>();
        for (int item : items) {
            itemList.add(item);
        }
        Collections.sort(itemList, Collections.reverseOrder());

        // Step 2: Initialize bins
        ArrayList<ArrayList<Integer>> bins = new ArrayList<>();

        // Step 3: Place each item in the first bin that can accommodate it
        for (int item : itemList) {
            boolean placed = false;
            for (ArrayList<Integer> bin : bins) {
                int currentBinSum = bin.stream().mapToInt(Integer::intValue).sum();
                if (currentBinSum + item <= binCapacity) {
                    bin.add(item);
                    placed = true;
                    break;
                }
            }
```

```java
public class BinPacking {
    public static ArrayList<ArrayList<Integer>> firstFitDecreasing(int[] items, int binCapacity) {  1 usage
            }
            if (!placed) {
                // Create a new bin if no existing bin can accommodate the item
                ArrayList<Integer> newBin = new ArrayList<>();
                newBin.add(item);
                bins.add(newBin);
            }
        }

        return bins;
    }

    public static void main(String[] args) {
        // Example input
        int[] items = {4, 8, 1, 4, 2, 1, 7, 6, 3}; // Item sizes
        int binCapacity = 10; // Capacity of each bin

        // Solve the problem using First Fit Decreasing
        ArrayList<ArrayList<Integer>> bins = firstFitDecreasing(items, binCapacity);

        // Output the results
        System.out.println("Number of bins used: " + bins.size());
        for (int i = 0; i < bins.size(); i++) {
            System.out.println("Bin " + (i + 1) + ": " + bins.get(i));
        }
    }
}
```

```
"C:\Program Files\Java\jdk-22\bin\java.exe" "-
Number of bins used: 4
Bin 1: [8, 2]
Bin 2: [7, 3]
Bin 3: [6, 4]
Bin 4: [4, 1, 1]

Process finished with exit code 0
```

9. Time and space complexity of problem statement

Let's dive into the **time and space complexity** of the **First Fit Decreasing (FFD)** and **Best Fit (BF)** algorithms for the Bin Packing Problem.

1. First Fit Decreasing (FFD)

Time Complexity

1. **Sorting the Items**:
    - FFD starts by sorting the items in decreasing order. Sorting an array of n items takes: O(nlogn)
2. **Placing Each Item**:
    - For each item, we iterate through the bins to find the first bin where it fits.
    - In the worst case, there can be n bins (when each item requires a new bin), so checking bins takes O(n) item.
    - For n items, the placement operation takes:
    - O(n×n)=O(n2)

Overall Time Complexity:

O(nlogn+n2)=O(n2)

Since n2 dominates nlogn the final complexity is O(n^2)

Space Complexity

1. **Bins**:
    - At most n bins are needed (if every item requires a new bin). Each bin stores items, and the total space for bins is proportional to the number of items.
    - Space required for bins: O(n)
2. **Sorted Items**:
    - Sorting creates an auxiliary array of n items.
    - Space required for sorting: O(n)

Overall Space Complexity:

O(n)

2. Best Fit (BF)

Time Complexity

1. **Placing Each Item**:

- For each item, Best Fit iterates through all bins to find the one that leaves the least remaining space after accommodating the item.
- Checking all bins takes O(m), where mmm is the number of bins (at most n).
- For n items, the placement operation takes: O(n×m)= O(n2)O(n \times m)

2. **Sorting** (if required):
   - If sorting is performed before placement (e.g., Best Fit Decreasing), the sorting step adds: O(nlogn)

Overall Time Complexity:

O(n2+nlogn)

Sorting is not always part of the pure Best Fit algorithm but is common in variations like Best Fit Decreasing.

Space Complexity

1. **Bins**:
   - Similar to FFD, at most n bins may be required, storing O(n)O(n)O(n) items in total.
   - Space required for bins: O(n)O(n)O(n)
2. **Auxiliary Storage**:
   - Best Fit does not inherently require additional storage beyond bins and items, unlike sorting which may require an auxiliary array of size n.

Overall Space Complexity:

O(n)O(n)O(n)

Comparison of Complexities

Algorithm	**Time Complexity**	**Space Complexity**
First Fit	O(n^2)O(n2)	O(n)
First Fit Decreasing	O(n^2)	O(n)
Best Fit	O(n^2)	O(n)
Best Fit Decreasing	O(n2)	O(n)

10. Conclusion

The First Fit Decreasing (FFD) and Best Fit (BF) algorithms are efficient, greedy approaches for solving the Bin Packing Problem, offering approximate solutions with similar time complexities of $O(n^2)$ and space complexities of $O(n)$. Variants like FFD and Best Fit Decreasing include a sorting step with an additional $O(n \log n)$ time complexity, which improves packing efficiency without significantly impacting overall performance for large n. FFD is often faster in practice, as it stops checking bins once a suitable one is found, while BF checks all bins to find the most space-efficient option, potentially resulting in better space utilization. FFD is simpler to implement and works well for general use cases, whereas BF can be beneficial in scenarios where optimizing space usage is critical, even at the cost of additional computations. These algorithms are widely used in practical applications such as container packing, memory management, and resource scheduling, where achieving a near-optimal solution quickly is more important than exact optimization.

11. References

1. https://en.wikipedia.org/wiki/Bin_packing_problem
2. https://en.wikipedia.org/wiki/Bin_packing_problem
3. https://www.geeksforgeeks.org/minimum-number-of-bins-required-to-place-n-items-using-best-fit-algorithm/
4. https://stackoverflow.com/questions/8310385/bin-packing-brute-force-recursive-solution-how-to-make-it-faster
5. https://codeincomplete.com/articles/bin-packing/