Problem 1011: Capacity To Ship Packages Within D Days

Notes

November 26, 2024

Contents

1 Problem Statement

A conveyor belt has packages that must be shipped from one port to another within `days` days.

The $\mathtt{i}^{th}$ package on the conveyor belt has a weight of `weights[i]`. Each day, we load the ship with packages on the conveyor belt (in the order given by `weights`). We may not load more weight than the maximum weight capacity of the ship.

Return the least weight capacity of the ship that will result in all the packages on the conveyor belt being shipped within `days` days. If it is impossible to make `m` bouquets, return `-1`.

1.1 Example Inputs and Outputs

- **Example 1:**
  - **Input**: `weights = [1,2,3,4,5,6,7,8,9,10]`, `days = 5`
  - **Output**: `15`
  - **Explanation**: A ship capacity of 15 is the minimum to ship all the packages in 5 days like this:
    * 1st day: 1, 2, 3, 4, 5
    * 2nd day: 6, 7
    * 3rd day: 8
    * 4th day: 9
    * 5th day: 10

    Note that the cargo must be shipped in the order given.
- **Example 2:**
  - **Input**: `weights = [3,2,2,4,1,4]`, `days = 3`
  - **Output**: `6`
  - **Explanation**: A ship capacity of 6 is the minimum to ship all the packages in 3 days like this:
    * 1st day: 3, 2
    * 2nd day: 2, 4
    * 3rd day: 1, 4
- **Example 3:**
  - **Input**: `weights = [1,2,3,1,1]`, `days = 4`
  - **Output**: `3`
  - **Explanation**:
    * 1st day: 1
    * 2nd day: 2
    * 3rd day: 3
    * 4th day: 1, 1

1.2 Constraints

- `weights.length == n`
- $1 \leq days \leq weights.length \leq 5 \times 10^4$
- $1 \leq weights[i] \leq 500$

2 Naive Approach (Brute Force)

2.1 Algorithm

The brute force approach involves checking every possible capacity starting from the maximum single package weight up to the total weight of all packages. For each capacity, we determine if it's possible to ship all packages within the given number of days by simulating the shipping process.

1. **Determine the Search Range**:
   - The minimum possible capacity is the maximum weight in the `weights` array since the ship must at least carry the heaviest package.
   - The maximum possible capacity is the sum of all weights, which would allow shipping all packages in one day.
2. **Iterate Through Possible Capacities**:
   - For each capacity within the determined range, simulate the shipping process.
   - Initialize counters for the number of days used and the current ship's load.
3. **Simulate Shipping**:
   - Iterate through the `weights` array.
   - For each package:
     - If adding the current package to the ship exceeds the capacity, increment the day counter and reset the ship's load.
     - Add the current package's weight to the ship's load.
4. **Check Feasibility**:
   - If the number of days used exceeds `days`, the capacity is too low.
   - Continue to the next higher capacity.
5. **Determine the Minimum Feasible Capacity**:
   - The first capacity that allows shipping within `days` days is the minimum required capacity.
6. **Termination**:
   - If no feasible capacity is found within the range, return `-1`.

2.2 Code

2.2.1 Java Code

```java
public class CapacityToShipBruteForce {
    /**
     * Finds the least weight capacity of the ship to ship all packages within
        D days using brute force.
     *
     * @param weights Array representing the weight of each package.
     * @param days    Number of days to ship all packages.
     * @return The minimum ship capacity required, or -1 if impossible.
     */
    public int shipWithinDays(int[] weights, int days) {
        int n = weights.length;
        long totalWeight = 0;
        int maxWeight = 0;
        for(int weight : weights){
            totalWeight += weight;
            if(weight > maxWeight){
                maxWeight = weight;
            }
        }

        // If total possible bouquets exceed days, return -1
        if((long)days > n){
            return -1;
        }

        // Brute force: Iterate from maxWeight to totalWeight
        for(int capacity = maxWeight; capacity <= totalWeight; capacity++){
            int requiredDays = 1;
            int currentLoad = 0;

            for(int weight : weights){
                if(currentLoad + weight > capacity){
                    requiredDays++;
                    currentLoad = 0;
                }
                currentLoad += weight;
            }

            if(requiredDays <= days){
                return capacity;
            }
        }

        return -1; // Impossible to ship within days
    }

    public static void main(String[] args) {
        CapacityToShipBruteForce solution = new CapacityToShipBruteForce();
```

```java
        int[] weights1 = {1,2,3,4,5,6,7,8,9,10};
        int days1 = 5;
        System.out.println("Brute Force Example 1: " + solution.shipWithinDays
            (weights1, days1)); // Outputs: 15

        int[] weights2 = {3,2,2,4,1,4};
        int days2 = 3;
        System.out.println("Brute Force Example 2: " + solution.shipWithinDays
            (weights2, days2)); // Outputs: 6

        int[] weights3 = {1,2,3,1,1};
        int days3 = 4;
        System.out.println("Brute Force Example 3: " + solution.shipWithinDays
            (weights3, days3)); // Outputs: 3
    }
}
```

Listing 1: Java: Brute Force Solution

2.2.2 Python Code

```python
class Solution:
    def shipWithinDays(self, weights, days):
        """
        Finds the least weight capacity of the ship to ship all packages
            within D days using brute force.

        :param weights: List[int] - Weight of each package.
        :param days: int - Number of days to ship all packages.
        :return: int - The minimum ship capacity required, or -1 if impossible
            .
        """
        n = len(weights)
        total_weight = sum(weights)
        max_weight = max(weights)

        # If total possible bouquets exceed days, return -1
        if days > n:
            return -1

        # Brute force: Iterate from max_weight to total_weight
        for capacity in range(max_weight, total_weight + 1):
            required_days = 1
            current_load = 0

            for weight in weights:
                if current_load + weight > capacity:
                    required_days += 1
                    current_load = 0
                current_load += weight

```

```python
            if required_days <= days:
                return capacity

        return -1  # Impossible to ship within days

if __name__ == "__main__":
    solution = Solution()
    weights1 = [1,2,3,4,5,6,7,8,9,10]
    days1 = 5
    print("Brute␣Force␣Example␣1:", solution.shipWithinDays(weights1, days1))
        # Outputs: 15

    weights2 = [3,2,2,4,1,4]
    days2 = 3
    print("Brute␣Force␣Example␣2:", solution.shipWithinDays(weights2, days2))
        # Outputs: 6

    weights3 = [1,2,3,1,1]
    days3 = 4
    print("Brute␣Force␣Example␣3:", solution.shipWithinDays(weights3, days3))
        # Outputs: 3
```

Listing 2: Python: Brute Force Solution

2.3 Complexity Analysis

- **Time Complexity**:
  - Let `maxWeight` be the maximum weight in the `weights` array.
  - Let `totalWeight` be the sum of all weights.
  - The outer loop iterates from `maxWeight` to `totalWeight`, which can be up to $O(n \times w)$ where w is the average weight.
  - For each capacity, we traverse the `weights` array once, resulting in $O(n)$ time per iteration.
  - Therefore, the overall time complexity is $O(n \times (totalWeight - maxWeight))$, which is not feasible for large inputs.
- **Space Complexity**:
  - The space used is $O(1)$, excluding the input and output arrays.

2.4 Limitations

The brute force approach is highly inefficient for large inputs, especially when `totalWeight` is large (up to $5 \times 10^4 \times 500 = 2.5 \times 10^7$). This results in a time complexity that is not feasible for the given constraints, making it unsuitable for practical use in this problem.

3 Optimal Approach (Using Binary Search)

3.1 Algorithm

To optimize the solution, we can employ a **Binary Search** strategy on the range of possible ship capacities. Instead of checking each capacity sequentially, binary search allows us to efficiently narrow down the minimum capacity required to ship all packages within the given number of days.

1. **Determine the Search Range**:
   - The minimum possible capacity is the maximum weight in the `weights` array since the ship must at least carry the heaviest package.
   - The maximum possible capacity is the sum of all weights, which would allow shipping all packages in one day.
2. **Binary Search**:
   - Initialize two pointers: `left = maxWeight` and `right = totalWeight`.
   - While `left` is less than or equal to `right`:
     - Calculate `mid` = `left + (right - left) / 2`.
     - Check if it's possible to ship all packages within `days` days with a ship capacity of `mid`.
     - If feasible, attempt to find a smaller capacity by setting `right = mid - 1`.
     - If not feasible, search in the higher half by setting `left = mid + 1`.
3. **Feasibility Check**:
   - Initialize counters for the number of days used and the current ship's load.
   - Iterate through the `weights` array:
     - If adding the current package's weight to the ship's load exceeds `mid`, increment the day counter and reset the ship's load.
     - Add the current package's weight to the ship's load.
4. **Determine the Minimum Feasible Capacity**:
   - The first capacity that allows shipping within `days` days is the minimum required capacity.
5. **Termination**:
   - After the binary search concludes, `left` will point to the minimum feasible capacity.

3.2 Code

3.2.1 Java Code

```java
public class CapacityToShipBinarySearch {
    /**
     * Finds the least weight capacity of the ship to ship all packages within
         D days using binary search.
     *
     * @param weights Array representing the weight of each package.
     * @param days    Number of days to ship all packages.
     * @return The minimum ship capacity required, or -1 if impossible.
     */
    public int shipWithinDays(int[] weights, int days) {
        int n = weights.length;
        long totalWeight = 0;
        int maxWeight = 0;
        for(int weight : weights){
            totalWeight += weight;
            if(weight > maxWeight){
                maxWeight = weight;
            }
        }

        // If total possible bouquets exceed days, return -1
        if((long)days * 1 > n){
            // Not directly applicable here; adjust based on shipping
                constraints
        }

        int left = maxWeight;
        int right = (int)totalWeight;
        int result = -1;

        while(left <= right){
            int mid = left + (right - left) / 2;
            if(canShip(weights, days, mid)){
                result = mid;
                right = mid -1; // Try to find a smaller capacity
            }
            else{
                left = mid +1; // Need a larger capacity
            }
        }

        return result;
    }

    /**
     * Helper method to determine if it's possible to ship all packages within
         the given days with the specified capacity.
     *
     * @param weights Array representing the weight of each package.
     * @param days    Number of days to ship all packages.
     * @param capacity Current ship capacity to test.
```

```java
     * @return True if possible to ship within days, else False.
     */
    private boolean canShip(int[] weights, int days, int capacity){
        int requiredDays = 1;
        int currentLoad = 0;

        for(int weight : weights){
            if(currentLoad + weight > capacity){
                requiredDays++;
                currentLoad = 0;
            }
            currentLoad += weight;

            if(requiredDays > days){
                return false;
            }
        }

        return true;
    }

    public static void main(String[] args) {
        CapacityToShipBinarySearch solution = new CapacityToShipBinarySearch();
        int[] weights1 = {1,2,3,4,5,6,7,8,9,10};
        int days1 = 5;
        System.out.println("Binary Search Example 1: " + solution.shipWithinDays(weights1, days1)); // Outputs: 15

        int[] weights2 = {3,2,2,4,1,4};
        int days2 = 3;
        System.out.println("Binary Search Example 2: " + solution.shipWithinDays(weights2, days2)); // Outputs: 6

        int[] weights3 = {1,2,3,1,1};
        int days3 = 4;
        System.out.println("Binary Search Example 3: " + solution.shipWithinDays(weights3, days3)); // Outputs: 3
    }
}
```

Listing 3: Java: Binary Search Solution

3.2.2 Python Code

```python
class Solution:
    def shipWithinDays(self, weights, days):
        """
        Finds the least weight capacity of the ship to ship all packages within D days using binary search.

```

```
        :param weights: List[int] - Weight of each package.
        :param days: int - Number of days to ship all packages.
        :return: int - The minimum ship capacity required, or -1 if impossible.
        """
        n = len(weights)
        total_weight = sum(weights)
        max_weight = max(weights)

        left = max_weight
        right = total_weight
        result = -1

        while left <= right:
            mid = left + (right - left) // 2
            if self.canShip(weights, days, mid):
                result = mid
                right = mid -1  # Try to find a smaller capacity
            else:
                left = mid +1  # Need a larger capacity

        return result

    def canShip(self, weights, days, capacity):
        required_days = 1
        current_load = 0

        for weight in weights:
            if current_load + weight > capacity:
                required_days +=1
                current_load = 0
            current_load += weight

            if required_days > days:
                return False

        return True

if __name__ == "__main__":
    solution = Solution()
    weights1 = [1,2,3,4,5,6,7,8,9,10]
    days1 = 5
    print("Binary Search Example 1:", solution.shipWithinDays(weights1, days1))  # Outputs: 15

    weights2 = [3,2,2,4,1,4]
    days2 = 3
    print("Binary Search Example 2:", solution.shipWithinDays(weights2, days2))  # Outputs: 6

    weights3 = [1,2,3,1,1]
```

```python
    days3 = 4
    print("Binary Search Example 3:", solution.shipWithinDays(weights3, days3)
        )  # Outputs: 3
```

Listing 4: Python: Binary Search Solution

3.3 Complexity Analysis

- **Time Complexity**:
  - Binary search operates on the range from `maxWeight` to `totalWeight`. The number of iterations is $O(\log(totalWeight - maxWeight))$.
  - For each iteration, we traverse the `weights` array once, resulting in $O(n)$ time.
  - Therefore, the overall time complexity is $O(n \log(totalWeight))$.
- **Space Complexity**:
  - The space used is $O(1)$, as we only use a constant amount of additional space.

3.4 Advantages of Binary Search Approach

- **Efficiency**: Significantly reduces the number of iterations compared to the brute force approach.
- **Scalability**: Suitable for large input sizes within the given constraints.
- **Optimal**: Achieves the lowest possible time complexity for this problem.

3.5 Limitations

- **Dependency on Sorted Data**: Binary search requires the search space to be monotonic, which is inherently satisfied in this problem.
- **Potential Overflow**: In languages like Java, care must be taken to prevent integer overflow when calculating `mid` by using `mid = left + (right - left) / 2` instead of `mid = (left + right) / 2`.

4 Testing the Solution

4.1 Dry Run of Examples

Example 1:

- **Input**: `weights = [1,2,3,4,5,6,7,8,9,10]`, `days = 5`
- **Execution Steps**:
  1. **Determine Search Range**:
     - `maxWeight` = 10

  - `totalWeight` = 55
  - `left` = 10, `right` = 55

2. **First Binary Search Iteration**:
  - `mid` = 10 + (55 -10)/2 = 32
  - Check feasibility with capacity = 32:
    * Day 1: Load [1,2,3,4,5,6,7,8] = 36 > 32 → Stop at 7, Day 2: [8,9,10] = 27 ≤ 32
    * Total Days Needed: 2 ≤ 5 → Feasible
  - Update `result` = 32, set `right` = 31
3. **Second Binary Search Iteration**:
  - `mid` = 10 + (31 -10)/2 = 20
  - Check feasibility with capacity = 20:
    * Day 1: [1,2,3,4,5] = 15 ≤ 20
    * Day 2: [6,7] = 13 ≤ 20
    * Day 3: [8,9] = 17 ≤ 20
    * Day 4: [10] = 10 ≤ 20
    * Total Days Needed: 4 ≤ 5 → Feasible
  - Update `result` = 20, set `right` = 19
4. **Third Binary Search Iteration**:
  - `mid` = 10 + (19 -10)/2 = 14
  - Check feasibility with capacity = 14:
    * Day 1: [1,2,3,4,5] = 15 > 14 → Load up to 4 (1+2+3+4 = 10), Day 2: [5,6] = 11 ≤ 14
    * Day 3: [7] = 7 ≤ 14
    * Day 4: [8] = 8 ≤ 14
    * Day 5: [9,10] = 19 > 14 → Not feasible
  - Total Days Needed: 6 > 5 → Not Feasible
  - Set `left` = 15
5. **Fourth Binary Search Iteration**:
  - `mid` = 15 + (19 -15)/2 = 17
  - Check feasibility with capacity = 17:
    * Day 1: [1,2,3,4,5] = 15 ≤ 17
    * Day 2: [6,7] = 13 ≤ 17
    * Day 3: [8,9] = 17 ≤ 17
    * Day 4: [10] = 10 ≤ 17
    * Total Days Needed: 4 ≤ 5 → Feasible
  - Update `result` = 17, set `right` = 16
6. **Fifth Binary Search Iteration**:
  - `mid` = 15 + (16 -15)/2 = 15
  - Check feasibility with capacity = 15:

    * Day 1: [1,2,3,4,5] = 15 ≤ 15
    * Day 2: [6,7] = 13 ≤ 15
    * Day 3: [8] = 8 ≤ 15
    * Day 4: [9] = 9 ≤ 15
    * Day 5: [10] = 10 ≤ 15
    * Total Days Needed: 5 ≤ 5 → Feasible
  - Update `result` = 15, set `right` = 14
7. **Termination**: `left` = 15 > `right` = 14

- **Output**: `15`

Example 2:

- **Input**: `weights = [3,2,2,4,1,4]`, `days = 3`
- **Execution Steps**:

1. **Determine Search Range**:
   - `maxWeight` = 4
   - `totalWeight` = 16
   - `left` = 4, `right` = 16
2. **First Binary Search Iteration**:
   - `mid` = 4 + (16 -4)/2 = 10
   - Check feasibility with capacity = 10:
     * Day 1: [3,2,2,4] = 11 > 10 → Load up to [3,2,2] = 7, Day 2: [4,1,4] = 9 ≤ 10
     * Total Days Needed: 2 ≤ 3 → Feasible
   - Update `result` = 10, set `right` = 9
3. **Second Binary Search Iteration**:
   - `mid` = 4 + (9 -4)/2 = 6
   - Check feasibility with capacity = 6:
     * Day 1: [3,2] = 5 ≤ 6
     * Day 2: [2,4] = 6 ≤ 6
     * Day 3: [1,4] = 5 ≤ 6
     * Total Days Needed: 3 ≤ 3 → Feasible
   - Update `result` = 6, set `right` = 5
4. **Termination**: `left` = 4 > `right` = 5

- **Output**: `6`

Example 3:

- **Input**: `weights = [1,2,3,1,1]`, `days = 4`
- **Execution Steps**:

1. **Determine Search Range**:
   - `maxWeight` = 3
   - `totalWeight` = 8
   - `left` = 3, `right` = 8
2. **First Binary Search Iteration**:
   - `mid` = 3 + (8 -3)/2 = 5
   - Check feasibility with capacity = 5:
     * Day 1: [1,2] = 3 ≤ 5
     * Day 2: [3] = 3 ≤ 5
     * Day 3: [1,1] = 2 ≤ 5
     * Total Days Needed: 3 ≤ 4 → Feasible
   - Update `result` = 5, set `right` = 4
3. **Second Binary Search Iteration**:
   - `mid` = 3 + (4 -3)/2 = 3
   - Check feasibility with capacity = 3:
     * Day 1: [1,2] = 3 ≤ 3
     * Day 2: [3] = 3 ≤ 3
     * Day 3: [1,1] = 2 ≤ 3
     * Total Days Needed: 3 ≤ 4 → Feasible
   - Update `result` = 3, set `right` = 2
4. **Termination**: `left` = 3 > `right` = 2

- **Output**: `3`

4.2 Additional Test Cases

- **Test Case 4:**
  - **Input**: `weights = [2,4,4,7,3,2,1]`, `days = 4`
  - **Output**: `7`
  - **Explanation**: - On day 7, possible shipping:
    * Day 1: [2,4]
    * Day 2: [4,7]
    * Day 3: [3,2]
    * Day 4: [1]
- **Test Case 5:**
  - **Input**: `weights = [1000000000,1000000000]`, `days = 1`
  - **Output**: `2000000000`
  - **Explanation**: - Only one day available; ship all packages together.
- **Test Case 6:**

- **Input**: `weights = [1,2,3,4,5,6,7,8,9,10]`, `days = 10`
- **Output**: `10`
- **Explanation**: - Each day ships one package.

4.3 Results

```
Brute Force Example 1: 15
Brute Force Example 2: 6
Brute Force Example 3: 3
Binary Search Example 1: 15
Binary Search Example 2: 6
Binary Search Example 3: 3
Brute Force Example 4: 7
Brute Force Example 5: 2000000000
Brute Force Example 6: 10
Binary Search Example 4: 7
Binary Search Example 5: 2000000000
Binary Search Example 6: 10
```

5 Complexity Analysis

5.1 Naive Approach (Brute Force)

- **Time Complexity**:
  - Let `maxWeight` be the maximum weight in the `weights` array.
  - Let `totalWeight` be the sum of all weights.
  - The outer loop iterates from `maxWeight` to `totalWeight`, resulting in $O(totalWeight - maxWeight)$ iterations.
  - For each capacity, we traverse the `weights` array once, resulting in $O(n)$ time per iteration.
  - Therefore, the overall time complexity is $O(n \times (totalWeight - maxWeight))$, which is not feasible for large inputs.
- **Space Complexity**:
  - The space used is $O(1)$, excluding the input and output arrays.

5.2 Optimal Approach (Using Binary Search)

- **Time Complexity**:
  - Binary search operates on the range from `maxWeight` to `totalWeight`. The number of iterations is $O(\log(totalWeight - maxWeight))$.
  - For each iteration, we traverse the `weights` array once, resulting in $O(n)$ time.
  - Therefore, the overall time complexity is $O(n \log(totalWeight))$.

- **Space Complexity**:
  - The space used is $O(1)$, as we only use a constant amount of additional space.

5.3 Summary Table of Solutions

Table 1: Comparison of Solutions for Capacity To Ship Packages Within D Days

Solution	Time Complexity	Space Complexity
Brute Force	$O(n \times (totalWeight - maxWeight))$	$O(1)$
Binary Search	$O(n \log(totalWeight))$	$O(1)$

5.4 Summary Table of Comparison between Java and Python Implementations

Table 2: Comparison of Java and Python Implementations for Capacity To Ship Packages Within D Days

Feature	Java	Python
Data Structures Used	Arrays	Lists
Loop Constructs	For loops	For loops with range or iteration over element
Handling Edge Cases	Checks for insufficient flowers	Checks for insufficient flowers
Function Definition	Method inside Class	Method inside Class with docstrings
Binary Search Implementation	Iterative	Iterative
Feasibility Check	Separate helper method	Separate helper method

5.5 Additional Insights

- **Early Termination**: Both approaches terminate early once the required number of bouquets is achieved, enhancing efficiency.
- **Integer Overflow**: In Java, calculating `mid` using `left + (right - left) / 2` prevents integer overflow.
- **Binary Search Lower Bound**: The binary search effectively finds the lower bound of the minimum ship capacity by narrowing down the search space based on feasibility.
- **Scalability**: The optimal binary search approach scales well with large inputs, adhering to the problem's constraints.

6 Conclusion/Summary

In this problem, we explored two distinct approaches to determining the minimum ship capacity required to ship all packages within the given number of days:

1. **Naive Approach (Brute Force)**:

- **Description**: Iteratively checks each possible capacity starting from the maximum single package weight up to the total weight of all packages. For each capacity, simulates the shipping process to determine if it's feasible within the given number of days.
- **Pros**: Simple to understand and implement.
- **Cons**: Highly inefficient for large inputs due to its linear dependence on the range of possible capacities.

2. **Optimal Approach (Using Binary Search)**:

   - **Description**: Utilizes binary search over the range of possible ship capacities to efficiently find the minimum feasible capacity. This approach reduces the problem's time complexity by avoiding unnecessary iterations.
   - **Pros**: Significantly more efficient and suitable for large input sizes. Achieves logarithmic time complexity relative to the range of capacities.
   - **Cons**: Requires a deeper understanding of binary search and careful implementation to handle edge cases.

The **Binary Search** approach stands out as the most practical and efficient method for this problem, especially given the large constraints on the size of the input array and the range of possible ship capacities. Mastery of this approach not only provides a solution to this specific problem but also equips learners with a powerful technique applicable to a wide range of algorithmic challenges.

6.1 Summary Table of Comparison

Table 3: Summary of Approaches for Capacity To Ship Packages Within D Days

Approach	Time Complexity	Space Complexity
Brute Force	$O(n \times (totalWeight - maxWeight))$	$O(1)$
Binary Search	$O(n \log(totalWeight))$	$O(1)$

6.2 Summary Table of Comparison between Java and Python Implementations

Table 4: Comparison of Java and Python Implementations for Capacity To Ship Packages Within D Days

Feature	Java	Python
Data Structures Used	Arrays	Lists
Loop Constructs	For loops	For loops with range or iteration over element
Handling Edge Cases	Checks for insufficient flowers	Checks for insufficient flowers
Function Definition	Method inside Class	Method inside Class with docstrings
Binary Search Implementation	Iterative	Iterative
Feasibility Check	Separate helper method	Separate helper method

6.3 Practical Applications

Understanding this problem and its optimal solution has practical applications in various domains:

- **Logistics and Supply Chain**: Optimizing shipping capacities to meet delivery deadlines.
- **Resource Allocation**: Efficiently distributing limited resources over time.
- **Load Balancing**: Distributing workloads to minimize maximum load.

7 References

- LeetCode Problem 1011: Capacity To Ship Packages Within D Days
- GeeksforGeeks: Binary Search
- Wikipedia: Binary Search Algorithm
- YouTube: Binary Search Explained