1011. Capacity To Ship Packages Within D Days

Medium Array Binary Search [Leetcode Link]

Problem Description

In this problem, we are tasked with finding the minimum weight capacity of a ship that is required to ship all packages on a conveyor belt within a specified number of days. Each package has a certain weight, and the ship cannot carry more than its maximum weight capacity. Therefore, the goal is to determine the smallest maximum weight that the ship can handle such that all the packages can still be shipped on time.

To accomplish this task, the packages must be loaded onto the ship in the same order as they are placed on the conveyor belt. This restriction adds complexity since we cannot reorder the packages to better fit the ship's weight capacity on a given day. Additionally, there is a limit to the number of days available to ship all packages.

Put simply, we need to discover the lightest possible maximum weight of the ship that allows all packages to be shipped within the given time frame. This weight determines how many packages can be loaded each day without exceeding the ship's weight limit, thus ensuring timely delivery.

Intuition

The solution to this problem relies on binary search, an efficient algorithm for finding an item in a sorted array. However, the direct application is not obvious because the weights are not sorted, and we are also dealing with an optimization problem, not just a simple search.

We use binary search to narrow down the possible range of the ship's weight capacity. The intuition here is that there is a lower bound, which is at least as large as the heaviest package (since the ship must be able to carry the heaviest package), and an upper bound, which is the sum of all package weights (since the ship can carry all packages at once).

The binary search will repeatedly split the range in half. At each step, it will guess a middle value and use a helper function `check` to determine if it is possible to ship all packages within 'days' days, without exceeding the guess weight capacity. The helper function will simulate the process of loading packages onto the ship:

- It sums the weights of the packages until adding another package would exceed the assumed maximum capacity.
- When the maximum is reached, it simulates the start of a new day, incrementing a day counter.
- If the day counter exceeds the allowed number of days, the guess is too low and we must try a larger capacity.

The binary search is constrained within the range between the heaviest package and the total weight using the `bisect_left` function. `bisect_left` will find the position to insert the lower bound of the ship's capacity in such a way that all packages are shipped within the time frame. At the end of the binary search, the minimum weight capacity of the ship is found, which is the goal of the problem.

Solution Approach

The solution approach revolves around binary search, and here's a step-by-step breakdown of how the provided solution in Python implements it:

1. **Define the `check` Function**:
   - The helper function `check(mx)` simulates the loading of packages onto the ship to determine if it is possible to ship all packages within `days` days, given a maximum shipping weight of `mx`.
   - It iterates through each weight in `weights` and aggregates them until the current load exceeds `mx`.
   - Once the aggregated weight exceeds `mx`, it simulates starting a new day by resetting the load to the current weight and incrementing the day count.
   - After iterating through all weights, if the number of days needed is less than or equal to `days`, it means the `mx` is a viable capacity; otherwise, it is not.
2. **Determine the Search Range**:
   - The lower bound `left` is set to the maximum single package weight (`max(weights)`). This is the absolute minimum capacity the ship must have to ensure even the heaviest package can be shipped.
   - The upper bound `right` is set to the sum of all weights plus one (`sum(weights) + 1`), the plus one being a non-inclusive upper bound for the binary search.
3. **Perform Binary Search**:
   - `bisect_left` function is used to perform the binary search. This is a function provided by Python's standard library, which uses binary search to find the insertion point for a given element `x` in a sorted array to maintain the array's sorted order. Here, it's used to find the minimum valid capacity.
   - The range from `left` to `right` is provided along with a key function `check` which `bisect_left` uses to determine the validity of the capacity. `True` represents the element to be inserted, corresponding to the condition of finding a workable capacity.
4. **Return the Calculated Capacity**:
   - The minimum weight capacity is found when `bisect_left` concludes the search. It's computed as `left + position` where `position` is the insertion point returned by `bisect_left`. Since `left` is the starting point of the search range, adding the position gives us the exact weight capacity.

By leveraging binary search, the solution minimizes the number of times it needs to check for a viable shipping plan, which drastically reduces the time complexity compared to linearly searching through all possible capacities. The use of `bisect_left` provides a concise and efficient way to perform the search. The `check` function acts as a simulation/verification step that guarantees that the found capacity is indeed the lowest that can still ship all packages within the provided number of days.

Example Walkthrough

Let's assume we have the following input where `weights = [3, 2, 2, 4, 1]` represent the weights of the packages, and we need to ship all packages within `days = 3`.

We first determine the search range for the ship's capacity:

- The heaviest package is `4`, so our lower bound `left` is `4`.
- The sum of all package weights is `3+2+2+4+1 = 12`, so our upper bound `right` is `13` (one more than the sum for binary search).

Now, we apply the binary search approach:

1. We start with the middle of our range. The initial guess for the ship's capacity will be `(left + right) / 2` which is `(4 + 13) / 2 = 8.5`. We round this down to `8` (since ship capacity must be an integer and binary search takes the lower mid).

2. With a capacity of `8`, we simulate the loading using the `check` function:

   - Day 1: We load `3 + 2 + 2` because adding the next package would exceed the capacity.
   - Day 2: We start with the next package and load `4`.
   - Day 3: The last package `1` is loaded.

   We managed to load all the packages in `3` days which is within our allowed `days`. This indicates capacity `8` is possible.

3. Since `8` worked, we now try to find if there is a smaller possible capacity. We adjust our search range to `left` remaining `4` and `right` becoming the current guess `8`.

4. Repeat the binary search process:

   - New middle is `(4 + 8) / 2 = 6`. So we check capacity `6`.
   - Day 1: We load `3 + 2` because adding another package would exceed the capacity.
   - Day 2: We continue with `2 + 4`, we reach the capacity exactly.
   - Day 3: We load the last package `1`.

   Again, we have managed to load all the packages in `3` days, so capacity `6` seems to work as well.

5. We continue the binary search with the new range `left` is `4` and `right` is now `6`.

6. New middle is `(4 + 6) / 2 = 5`. So we check capacity `5`.

   - Day 1: We load `3` and cannot add `2` because it would exceed capacity, so we load just `3`.
   - Day 2: We load `2 + 2` and stop there, as adding `4` would exceed the capacity.
   - Day 3: We load `4`, but now we cannot load the last package as it would require an additional day.

   In this case, we failed to load all packages within `3` days using capacity `5`. This tells us the capacity of `5` is too low.

7. Now adjust the range with `left` becoming `6`, since `5` didn't work, and `right` staying `6`.

   Given that `left` and `right` are now the same, the search ends, and we have found that the minimum capacity needed is `6`.

By using binary search and simulating the loading of packages, we determined that the minimum capacity of the ship to load all the packages within `3` days is `6`. This approach is efficient because instead of trying every capacity between the heaviest package and the total weight, we eliminate half of the possibilities with each step.

Python Solution

```python
from typing import List
from bisect import bisect_left

class Solution:
    def shipWithinDays(self, weights: List[int], days: int) -> int:
        # Helper function to check if the given capacity 'max_capacity' is enough
        # to ship all the packages within 'days' days
        def can_ship_with_capacity(max_capacity):
            current_weight, day_count = 0, 1  # Initialize current total weight and day count
            # Loop through the weights of the packages
            for weight in weights:
                current_weight += weight  # Add package weight to current load
                # If the current load exceeds the max capacity, start a new day
                # and reset the current load to the current package's weight
                if current_weight > max_capacity:
                    day_count += 1
                    current_weight = weight
            # The capacity is enough if we can ship within required days
            return day_count <= days

        # Define the search space between largest single package and total weight
        left, right = max(weights), sum(weights)

        # Use binary search to find the minimum capacity needed to ship within 'days' days
        # The search will find the first value where 'can_ship_with_capacity' returns True
        min_capacity_needed = left + bisect_left(range(left, right), True, key=can_ship_with_capacity)

        return min_capacity_needed

# Example usage:
# solution = Solution()
# min_capacity = solution.shipWithinDays([1,2,3,4,5,6,7,8,9,10], 5)
# print(min_capacity)  # Output: 15

```

Java Solution

```java
class Solution {
    public int shipWithinDays(int[] weights, int days) {
        // Setting initial left boundary to the largest weight
        // and right boundary to the sum of all weights
        int leftBoundary = 0, rightBoundary = 0;
        for (int weight : weights) {
            leftBoundary = Math.max(leftBoundary, weight);
            rightBoundary += weight;
        }

        // Binary search to find the minimum capacity of the ship that will work within given days
        while (leftBoundary < rightBoundary) {
            // Midpoint to test if it's a valid ship capacity
            int midCapacity = (leftBoundary + rightBoundary) >> 1;
            // Check if midCapacity is enough, if so, look for a smaller potential capacity
            if (canShip(weights, midCapacity, days)) {
                rightBoundary = midCapacity;
            } else {
                // Otherwise, discard midCapacity and look for a larger one
                leftBoundary = midCapacity + 1;
            }
        }
        // Return the minimum valid ship capacity
        return leftBoundary;
    }

    // Helper method to determine if the given capacity can ship all weights within the given days
    private boolean canShip(int[] weights, int capacity, int days) {
        int currentLoad = 0;   // Current weight load of the ship
        int dayCount = 1;      // Start with the first day

        // Iterating over each weight to simulate the shipping process
        for (int weight : weights) {
            currentLoad += weight;
            // If adding weight exceeds capacity, ship current load and increment day counter
            if (currentLoad > capacity) {
                currentLoad = weight; // Reset load with the current weight
                dayCount++;           // Move to the next day
            }
        }

        // If the number of shipping days needed is within the allowed range, return true
        return dayCount <= days;
    }
}

```

C++ Solution

```cpp
#include <vector>
#include <algorithm> // For max()
using std::vector;
using std::max;

class Solution {
public:
    // Function to find the minimum capacity of a ship that can ship all the packages within 'days' days
    int shipWithinDays(vector<int>& weights, int days) {
        int minCapacity = 0; // Lower boundary of binary search - max weight in the shipment
        int maxCapacity = 0; // Upper boundary of binary search - sum of all weights

        // Calculate the minimum and maximum capacity
        for (int weight : weights) {
            minCapacity = max(minCapacity, weight);
            maxCapacity += weight;
        }

        // Lambda function to check if mid value can work as the capacity
        auto canShipWithCapacity = [&](int capacity) -> bool {
            int currentWeightSum = 0; // Current weight sum of the ongoing shipment
            int dayCount = 1; // Counter for the number of days needed

            for (int weight : weights) {
                currentWeightSum += weight;

                // Check if adding the current weight exceeds capacity, if so, reset for the next day
                if (currentWeightSum > capacity) {
                    currentWeightSum = weight;
                    dayCount++;
                }
            }

            // Check if we can ship within the given number of days with the current capacity
            return dayCount <= days;
        };

        // Binary search to find the minimum capacity required
        while (minCapacity < maxCapacity) {
            int midCapacity = (minCapacity + maxCapacity) / 2;

            // Check if the midCapacity can be a possible solution, adjust search boundaries based on it
            if (canShipWithCapacity(midCapacity)) {
                maxCapacity = midCapacity;
            } else {
                minCapacity = midCapacity + 1;
            }
        }

        // Return the minimum capacity needed to ship within 'days' days
        return minCapacity;
    }
};

```

Typescript Solution

```typescript
function shipWithinDays(weights: number[], days: number): number {
    // Initialize the lower and upper bounds for the binary search
    let lowerBound = 0;
    let upperBound = 0;

    // Calculate the initial bounds for the capacity of the ship
    for (const weight of weights) {
        lowerBound = Math.max(lowerBound, weight); // The ship's capacity must be at least as much as the heaviest package
        upperBound += weight; // The maximum capacity is the sum of all weights, i.e., shipping all at once
    }

    // Function to determine if it's possible to ship all packages within 'days' given a maximum capacity 'maxCapacity'
    const canShipInDays = (maxCapacity: number): boolean => {
        let currentWeightSum = 0; // Current total weight in the current shipment
        let requiredDays = 1;     // Start with 1 day, the minimum possible

        for (const weight of weights) {
            currentWeightSum += weight;

            // If adding the current weight exceeds max capacity, need a new shipment (next day)
            if (currentWeightSum > maxCapacity) {
                currentWeightSum = weight; // Reset the currentWeightSum with the current weight as the start for the next day
                ++requiredDays;            // Increment the day counter as we move to the next day
            }
        }

        // Return true if the number of required days is less than or equal to the given days, false otherwise
        return requiredDays <= days;
    };

    // Perform a binary search to find the minimum capacity needed to ship within 'days'
    while (lowerBound < upperBound) {
        const midCapacity = Math.floor((lowerBound + upperBound) / 2); // Mid-point of the current bounds

        // If it's possible to ship with this capacity, reduce the upper bound to midCapacity
        if (canShipInDays(midCapacity)) {
            upperBound = midCapacity;
        } else {
            // Otherwise, increase the lower bound just above midCapacity
            lowerBound = midCapacity + 1;
        }
    }

    // The lower bound at the end of the binary search will be the minimum capacity needed to meet the requirement
    return lowerBound;
}

```

Time and Space Complexity

The time complexity of the code is `O(n * log(S))`, where `n` is the number of elements in `weights`, and `S` is the sum of weights minus the maximum weight in `weights`. This is because the binary search is performed over a range with `left` and `right` as lower and upper bounds, respectively. The `check` function, called at each step of the binary search, runs in `O(n)` time since it iterates through all the elements of `weights`. Since the binary search narrows the range by half each time, it will run `log(S - max(weights))` times.

The space complexity of the code is `O(1)` as it uses a constant amount of space. The `check` function uses variables to store the current weight sum and the count of days, both of which do not depend on the input size.