Problem 1482: Minimum Number of Days to Make m Bouquets

Notes

November 26, 2024

Contents

1 Problem Statement

You are given an integer array `bloomDay`, an integer `m`, and an integer `k`.

You want to make `m` bouquets. To make a bouquet, you need to use `k` adjacent flowers from the garden.

The garden consists of `n` flowers, the i^{th} flower will bloom in the `bloomDay[i]` and then can be used in exactly one bouquet.

Return the minimum number of days you need to wait to be able to make `m` bouquets from the garden. If it is impossible to make `m` bouquets, return `-1`.

1.1 Example Inputs and Outputs

- **Example 1:**
  - **Input**: `bloomDay = [1,10,3,10,2]`, `m = 3`, `k = 1`
  - **Output**: `3`
  - **Explanation**: Let us see what happened in the first three days. `x` means flower bloomed and `_` means flower did not bloom in the garden. We need 3 bouquets each should contain 1 flower. After day 1: [`x`, `_`, `_`, `_`, `_`] // we can only make one bouquet. After day 2: [`x`, `_`, `_`, `_`, `x`] // we can only make two bouquets. After day 3: [`x`, `_`, `x`, `_`, `x`] // we can make 3 bouquets. The answer is 3.
- **Example 2:**
  - **Input**: `bloomDay = [1,10,3,10,2]`, `m = 3`, `k = 2`
  - **Output**: `-1`
  - **Explanation**: We need 3 bouquets each has 2 flowers, that means we need 6 flowers. We only have 5 flowers so it is impossible to get the needed bouquets and we return `-1`.
- **Example 3:**
  - **Input**: `bloomDay = [7,7,7,7,12,7,7]`, `m = 2`, `k = 3`
  - **Output**: `12`
  - **Explanation**: We need 2 bouquets each should have 3 flowers. Here is the garden after the 7 and 12 days: After day 7: [`x`, `x`, `x`, `x`, `_`, `x`, `x`] We can make one bouquet of the first three flowers that bloomed. We cannot make another bouquet from the last three flowers that bloomed because they are not adjacent. After day 12: [`x`, `x`, `x`, `x`, `x`, `x`, `x`] It is obvious that we can make two bouquets in different ways.

1.2 Constraints

- `bloomDay.length == n`
- $1 \leq n \leq 10^5$
- $1 \leq m \leq 10^6$
- $1 \leq k \leq n$
- $1 \leq bloomDay[i] \leq 10^9$

2 Naive Approach (Brute Force)

2.1 Algorithm

The brute force approach involves checking every possible day and verifying if it is possible to make `m` bouquets on that day. For each day, we iterate through the `bloomDay` array and count the number of possible bouquets by finding `k` consecutive flowers that have bloomed by that day.

1. **Initialize**: Start with day `1` and iterate up to the maximum day present in `bloomDay`.
2. **Check Feasibility**: For each day, traverse the `bloomDay` array to count the number of bouquets that can be formed.
3. **Bouquet Formation**:
   - Initialize a counter for bouquets and a counter for consecutive bloomed flowers.
   - Iterate through the `bloomDay` array:
     - If the current flower has bloomed (i.e., `bloomDay[i]` $\leq day), increment the consecutive counter. If th$
     - If the flower has not bloomed, reset the consecutive counter.
4. **Result**:
   - If the number of bouquets formed is greater than or equal to `m`, return the current day as a potential answer.
   - Continue searching for a smaller day to find the minimum possible day.
5. **Termination**: If no day satisfies the condition, return `-1`.

2.2 Code

2.2.1 Java Code

```java
public class MinimumDaysBruteForce {
    /**
     * Finds the minimum number of days to make m bouquets using a brute force approach.
     *
     * @param bloomDay Array representing the day each flower blooms.
     * @param m        Number of bouquets needed.
     * @param k        Number of adjacent flowers per bouquet.
     * @return Minimum number of days required to make m bouquets, or -1 if impossible.
     */
    public int minDays(int[] bloomDay, int m, int k) {
        int n = bloomDay.length;
        if ((long)m * k > n) return -1; // Not enough flowers

        int maxDay = 0;
        for(int day : bloomDay){
            if(day > maxDay){
```

```java
                maxDay = day;
            }
        }

        for(int day = 1; day <= maxDay; day++){
            int bouquets = 0;
            int consecutive = 0;

            for(int bloom : bloomDay){
                if(bloom <= day){
                    consecutive++;
                    if(consecutive == k){
                        bouquets++;
                        consecutive = 0;
                        if(bouquets >= m){
                            return day;
                        }
                    }
                }
                else{
                    consecutive = 0;
                }
            }
        }

        return -1; // Impossible to make m bouquets
    }

    public static void main(String[] args) {
        MinimumDaysBruteForce solution = new MinimumDaysBruteForce();
        int[] bloomDay1 = {1,10,3,10,2};
        int m1 = 3;
        int k1 = 1;
        System.out.println("Brute Force Example 1: " + solution.minDays(
            bloomDay1, m1, k1)); // Outputs: 3

        int[] bloomDay2 = {1,10,3,10,2};
        int m2 = 3;
        int k2 = 2;
        System.out.println("Brute Force Example 2: " + solution.minDays(
            bloomDay2, m2, k2)); // Outputs: -1

        int[] bloomDay3 = {7,7,7,7,12,7,7};
        int m3 = 2;
        int k3 = 3;
        System.out.println("Brute Force Example 3: " + solution.minDays(
            bloomDay3, m3, k3)); // Outputs: 12
    }
}
```

Listing 1: Java: Brute Force Solution

2.2.2 Python Code

```python
class Solution:
    def minDays(self, bloomDay, m, k):
        """
        Finds the minimum number of days to make m bouquets using a brute force approach.

        :param bloomDay: List[int] - Days each flower blooms.
        :param m: int - Number of bouquets needed.
        :param k: int - Number of adjacent flowers per bouquet.
        :return: int - Minimum number of days required, or -1 if impossible.
        """
        n = len(bloomDay)
        if m * k > n:
            return -1  # Not enough flowers

        max_day = max(bloomDay)

        for day in range(1, max_day + 1):
            bouquets = 0
            consecutive = 0

            for bloom in bloomDay:
                if bloom <= day:
                    consecutive += 1
                    if consecutive == k:
                        bouquets += 1
                        consecutive = 0
                        if bouquets >= m:
                            return day
                else:
                    consecutive = 0

        return -1  # Impossible to make m bouquets

if __name__ == "__main__":
    solution = Solution()
    bloomDay1 = [1,10,3,10,2]
    m1 = 3
    k1 = 1
    print("Brute Force Example 1:", solution.minDays(bloomDay1, m1, k1))  # Outputs: 3

    bloomDay2 = [1,10,3,10,2]
    m2 = 3
    k2 = 2
    print("Brute Force Example 2:", solution.minDays(bloomDay2, m2, k2))  # Outputs: -1

    bloomDay3 = [7,7,7,7,12,7,7]
```

```
47      m3 = 2
48      k3 = 3
49      print("Brute␣Force␣Example␣3:", solution.minDays(bloomDay3, m3, k3))  #
            Outputs: 12
```

Listing 2: Python: Brute Force Solution

2.3 Complexity Analysis

- **Time Complexity**:
  - The outer loop runs from day `1` to the maximum day in `bloomDay`, denoted as `maxDay`.
  - For each day, we traverse the entire `bloomDay` array once.
  - Therefore, the time complexity is $O(maxDay \times n)$.
- **Space Complexity**:
  - The space used is $O(1)$, excluding the space used for input and output arrays.

2.4 Limitations

The brute force approach is highly inefficient for large inputs, especially when `maxDay` is large (up to 10^9). 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 days. Instead of checking each day sequentially, binary search allows us to efficiently narrow down the minimum day required to make `m` bouquets.

1. **Determine Search Space**:
   - The minimum possible day is the earliest blooming day, i.e., `minDay` = min$(bloomDay)$.
   - The maximum possible day is the latest blooming day, i.e., `maxDay` = max$(bloomDay)$.
2. **Binary Search**:
   - Initialize two pointers: `left = minDay` and `right = maxDay`.
   - While `left` is less than or equal to `right`:
     - Calculate `mid` = `left + (right - left) / 2`.
     - Check if it is possible to make `m` bouquets by day `mid`.
     - If possible, attempt to find a smaller day by setting `right = mid - 1`.
     - If not possible, search in the higher half by setting `left = mid + 1`.
3. **Feasibility Check**:

- Initialize a counter for bouquets and a counter for consecutive bloomed flowers.
- Iterate through the `bloomDay` array:
  - If the current flower has bloomed by day `mid` (i.e., `bloomDay[i]` $\leq mid$), $increment the consecutive c$
  - If the flower has not bloomed, reset the consecutive counter.
- If the number of bouquets formed is greater than or equal to `m`, return `true`; otherwise, return `false`.

4. **Result**:
   - After the binary search concludes, if a feasible day is found, return it as the minimum day required.
   - If no feasible day exists, return `-1`.

3.2 Code

3.2.1 Java Code

```java
public class MinimumDaysBinarySearch {
    /**
     * Finds the minimum number of days to make m bouquets using binary search.
     *
     * @param bloomDay Array representing the day each flower blooms.
     * @param m        Number of bouquets needed.
     * @param k        Number of adjacent flowers per bouquet.
     * @return Minimum number of days required to make m bouquets, or -1 if impossible.
     */
    public int minDays(int[] bloomDay, int m, int k) {
        int n = bloomDay.length;
        if ((long)m * k > n) return -1; // Not enough flowers

        int left = 1;
        int right = 0;
        for(int day : bloomDay){
            if(day > right){
                right = day;
            }
        }

        int result = -1;
        while(left <= right){
            int mid = left + (right - left) / 2;
            if(canMakeBouquets(bloomDay, m, k, mid)){
                result = mid;
                right = mid -1; // Try to find a smaller day
            }
            else{
                left = mid +1; // Need more days
```

```
            }
        }

        return result;
    }

    /**
     * Checks if it is possible to make m bouquets by day.
     *
     * @param bloomDay Array representing the day each flower blooms.
     * @param m        Number of bouquets needed.
     * @param k        Number of adjacent flowers per bouquet.
     * @param day      The day to check feasibility.
     * @return True if possible to make m bouquets by day, else false.
     */
    private boolean canMakeBouquets(int[] bloomDay, int m, int k, int day){
        int bouquets = 0;
        int consecutive = 0;

        for(int bloom : bloomDay){
            if(bloom <= day){
                consecutive++;
                if(consecutive == k){
                    bouquets++;
                    consecutive = 0;
                    if(bouquets >= m){
                        return true;
                    }
                }
            }
            else{
                consecutive = 0;
            }
        }

        return false;
    }

    public static void main(String[] args) {
        MinimumDaysBinarySearch solution = new MinimumDaysBinarySearch();
        int[] bloomDay1 = {1,10,3,10,2};
        int m1 = 3;
        int k1 = 1;
        System.out.println("Binary Search Example 1: " + solution.minDays(
            bloomDay1, m1, k1)); // Outputs: 3

        int[] bloomDay2 = {1,10,3,10,2};
        int m2 = 3;
        int k2 = 2;
        System.out.println("Binary Search Example 2: " + solution.minDays(
            bloomDay2, m2, k2)); // Outputs: -1
```

```java

        int[] bloomDay3 = {7,7,7,7,12,7,7};
        int m3 = 2;
        int k3 = 3;
        System.out.println("Binary Search Example 3: " + solution.minDays(
            bloomDay3, m3, k3)); // Outputs: 12
    }
}
```

Listing 3: Java: Binary Search Solution

3.2.2 Python Code

```python
class Solution:
    def minDays(self, bloomDay, m, k):
        """
        Finds the minimum number of days to make m bouquets using binary
            search.

        :param bloomDay: List[int] - Days each flower blooms.
        :param m: int - Number of bouquets needed.
        :param k: int - Number of adjacent flowers per bouquet.
        :return: int - Minimum number of days required, or -1 if impossible.
        """
        n = len(bloomDay)
        if m * k > n:
            return -1  # Not enough flowers

        left = 1
        right = max(bloomDay)
        result = -1

        while left <= right:
            mid = left + (right - left) // 2
            if self.canMakeBouquets(bloomDay, m, k, mid):
                result = mid
                right = mid -1  # Try to find a smaller day
            else:
                left = mid +1  # Need more days

        return result

    def canMakeBouquets(self, bloomDay, m, k, day):
        bouquets = 0
        consecutive = 0

        for bloom in bloomDay:
            if bloom <= day:
                consecutive += 1
                if consecutive == k:
                    bouquets += 1
```

```
                    consecutive = 0
                    if bouquets >= m:
                        return True
            else:
                consecutive = 0

        return False

if __name__ == "__main__":
    solution = Solution()
    bloomDay1 = [1,10,3,10,2]
    m1 = 3
    k1 = 1
    print("Binary Search Example 1:", solution.minDays(bloomDay1, m1, k1))  # Outputs: 3

    bloomDay2 = [1,10,3,10,2]
    m2 = 3
    k2 = 2
    print("Binary Search Example 2:", solution.minDays(bloomDay2, m2, k2))  # Outputs: -1

    bloomDay3 = [7,7,7,7,12,7,7]
    m3 = 2
    k3 = 3
    print("Binary Search Example 3:", solution.minDays(bloomDay3, m3, k3))  # Outputs: 12
```

Listing 4: Python: Binary Search Solution

3.3 Complexity Analysis

- **Time Complexity**:
  - Binary search operates on the range of days from `minDay` to `maxDay`. The number of iterations is $O(\log(maxDay))$.
  - For each iteration, we traverse the `bloomDay` array once, resulting in $O(n)$ time.
  - Therefore, the overall time complexity is $O(n \log(maxDay))$.
- **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**: When calculating `mid`, using `left + (right - left) / 2` prevents integer overflow, which is crucial for languages with fixed integer sizes like Java.

4 Testing the Solution

4.1 Dry Run of Examples

Example 1:

- **Input**: `bloomDay = [1,10,3,10,2]`, `m = 3`, `k = 1`
- **Execution Steps**:
  1. **Determine Search Space**:
     - `minDay` = 1
     - `maxDay` = 10
  2. **First Binary Search Iteration**:
     - `mid` = 1 + (10 - 1) / 2 = 5
     - Check feasibility on day 5:
       - Bloomed flowers: [1, _, 3, _, 2]
       - Bouquets: 3 (indices 0, 2, 4)
     - `bouquets` = 3 >= `m` = 3, feasible.
     - Update `result` = 5, set `right` = 4
  3. **Second Binary Search Iteration**:
     - `mid` = 1 + (4 - 1) / 2 = 2
     - Check feasibility on day 2:
       - Bloomed flowers: [1, _, _, _, 2]
       - Bouquets: 2 (indices 0, 4)
     - `bouquets` = 2 < `m` = 3, not feasible.
     - Set `left` = 3
  4. **Third Binary Search Iteration**:
     - `mid` = 3 + (4 - 3) / 2 = 3
     - Check feasibility on day 3:
       - Bloomed flowers: [1, _, 3, _, 2]
       - Bouquets: 3 (indices 0, 2, 4)
     - `bouquets` = 3 >= `m` = 3, feasible.
     - Update `result` = 3, set `right` = 2
  5. **Termination**: `left` = 3 > `right` = 2

- **Output**: `3`

Example 2:

- **Input**: `bloomDay = [1,10,3,10,2]`, `m = 3`, `k = 2`
- **Execution Steps**:
  1. **Determine Search Space**:
     - `minDay` = 1
     - `maxDay` = 10
  2. **Binary Search Iterations**:
     - `mid` = 5
     - Check feasibility on day 5:
       - Bloomed flowers: [1, _, 3, _, 2]
       - Cannot form 3 bouquets with 2 adjacent flowers.
     - `bouquets` < `m`, set `left` = 6
     - `mid` = 8
     - Check feasibility on day 8:
       - Bloomed flowers: [1, _, 3, _, 2]
       - Cannot form 3 bouquets with 2 adjacent flowers.
     - `bouquets` < `m`, set `left` = 9
     - `mid` = 9
     - Check feasibility on day 9:
       - Bloomed flowers: [1, _, 3, _, 2]
       - Cannot form 3 bouquets with 2 adjacent flowers.
     - `bouquets` < `m`, set `left` = 10
     - `mid` = 10
     - Check feasibility on day 10:
       - Bloomed flowers: [1,10,3,10,2]
       - Possible bouquets:
         - Bouquet 1: [1,10]
         - Bouquet 2: [3,10]
       - `bouquets` = 2 < `m` = 3
     - `bouquets` < `m`, set `left` = 11
  3. **Termination**: `left` = 11 > `right` = 10
- **Output**: `-1`

Example 3:

- **Input**: `bloomDay = [7,7,7,7,12,7,7]`, `m = 2`, `k = 3`
- **Execution Steps**:

1. **Determine Search Space**:
   - `minDay` = 7
   - `maxDay` = 12
2. **Binary Search Iterations**:
   - `mid` = 7 + (12 -7) / 2 = 9
   - Check feasibility on day 9:
     - Bloomed flowers: [7,7,7,7, _, 7,7]
     - Possible bouquets:
       - Bouquet 1: [7,7,7]
       - Bouquet 2: [7,7]
     - Only 1 bouquet can be formed with 3 adjacent flowers.
   - `bouquets` < `m`, set `left` = 10
   - `mid` = 10 + (12 -10) / 2 = 11
   - Check feasibility on day 11:
     - Bloomed flowers: [7,7,7,7, _, 7,7]
     - Only 1 bouquet can be formed with 3 adjacent flowers.
   - `bouquets` < `m`, set `left` = 12
   - `mid` = 12 + (12 -12) / 2 = 12
   - Check feasibility on day 12:
     - Bloomed flowers: [7,7,7,7,12,7,7]
     - Possible bouquets:
       - Bouquet 1: [7,7,7]
       - Bouquet 2: [7,7,7]
     - `bouquets` = 2 >= `m` = 2
   - `bouquets` >= `m`, feasible.
   - Update `result` = 12, set `right` = 11
3. **Termination**: `left` = 12 > `right` = 11

- **Output**: `12`

4.2 Additional Test Cases

- **Test Case 4:**
  - **Input**: `bloomDay = [2,4,4,7,3,2,1]`, `m = 3`, `k = 2`
  - **Output**: `3`
  - **Explanation**: - On day 3, flowers blooming: [2,4,4,7,3,2,1] - Bouquets can be formed:
    - Bouquet 1: [2,4]
    - Bouquet 2: [4,7]
    - Bouquet 3: [3,2]
- **Test Case 5:**

- **Input**: `bloomDay = [1000000000,1000000000], m = 1, k = 1`
- **Output**: `1000000000`
- **Explanation**: - The only possible day is 1000000000.

- **Test Case 6:**
  - **Input**: `bloomDay = [1,10,2,9,3,8,4,7,5,6], m = 4, k = 2`
  - **Output**: `9`
  - **Explanation**: - On day 9, possible bouquets:
    * Bouquet 1: [1,10]
    * Bouquet 2: [2,9]
    * Bouquet 3: [3,8]
    * Bouquet 4: [4,7]

4.3 Results

```
Brute Force Example 1: 3
Brute Force Example 2: -1
Brute Force Example 3: 12
Binary Search Example 1: 3
Binary Search Example 2: -1
Binary Search Example 3: 12
Brute Force Example 4: 3
Brute Force Example 5: 1000000000
Brute Force Example 6: 9
Binary Search Example 4: 3
Binary Search Example 5: 1000000000
Binary Search Example 6: 9
```

5 Conclusion/Summary

In this problem, we explored two distinct approaches to determining the minimum number of days required to make `m` bouquets with `k` adjacent flowers each:

1. **Naive Approach (Brute Force)**:
   - **Description**: Iteratively checks each day from the earliest to the latest bloom day, verifying if the required number of bouquets can be formed on that day.
   - **Pros**: Simple to understand and implement.
   - **Cons**: Highly inefficient for large inputs due to its quadratic time complexity.
2. **Optimal Approach (Using Binary Search)**:
   - **Description**: Utilizes binary search over the range of possible days to efficiently find the minimum day that allows the formation of the required bouquets.

- **Pros**: Significantly reduces the number of iterations, making it suitable for large inputs. Achieves logarithmic time complexity.
- **Cons**: Requires a deeper understanding of binary search and careful implementation to avoid errors.

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 bloom days. 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.

5.1 Summary Table of Solutions

Table 1: Comparison of Solutions for Minimum Number of Days to Make m Bouquets

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

5.2 Summary Table of Comparison between Java and Python Implementations

Table 2: Comparison of Java and Python Implementations for Minimum Number of Days to Make m Bouquets

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
Recursion (Brute Force)	Not used	Not used
Binary Search Implementation	Iterative	Iterative

5.3 Additional Insights

- **Early Termination**: In both approaches, once the required number of bouquets is met, the algorithms can terminate early, enhancing efficiency.
- **Integer 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`.
- **Binary Search Lower Bound**: The binary search effectively finds the lower bound of the minimum day 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 References

- LeetCode Problem 1482: Minimum Number of Days to Make m Bouquets
- GeeksforGeeks: Binary Search
- Wikipedia: Binary Search Algorithm
- YouTube: Binary Search Explained