1482. Minimum Number of Days to Make m Bouquets

Binary Search Medium (Array)

# In this problem, we have an integer array bloomDay where each element represents the day on which a particular flower will bloom.

create one bouquet). The goal is to find out the minimum number of days we need to wait until we can make exactly m bouquets using k adjacent flowers for each bouquet. The garden is comprised of n flowers, and the i-th flower will bloom on bloomDay[i]. We can use each flower in exactly one bouquet once it has bloomed.

Leetcode Link

Example: Given bloomDay = [1, 10, 3, 10, 2], m = 3, k = 1, the output is 3 because on the third day, three flowers are bloomed which is enough to make 3 bouquets as 'k' is 1.

Intuition

found, where left will eventually hold the minimum day required to make m bouquets.

bloomed flowers we are currently considering for a bouquet).

day day, and the function returns True. Otherwise, it returns False.

bloomDay (the latest possible day a flower can bloom).

required to make the bouquets.

Example Walkthrough

Step 2: Setting Up Binary Search

12.

Day 8 (First Iteration):

8).

Day 10 (Second Iteration):

now 10.

from typing import List

if m \* k > len(bloom\_day):

return -1

class Solution:

days.

10

11

12

13

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

44

45

46

6

15

16

17

18

19

20

21

22

23

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

43

44

45

46

48

49

50

10

11

12

13

14

15

16

17

18

19

20

21

22

23

30 }

31

34

35

36

37

38

40

41

42

43

44

45

46

47

48

49

50

52

51 }

};

return left;

Typescript Solution

return -1;

while (left < right) +

let minDay = Math.min(...bloomDay);

let maxDay = Math.max(...bloomDay);

// Initialize binary search bounds

let left = minDay, right = maxDay;

right = mid;

for (let bloom of bloomDay) {

bouquetsMade++;

return bouquetsMade >= bouquetCount;

Time and Space Complexity

If it is impossible to make m bouquets with the given constraints, the function should return -1.

The solution to this problem involves a binary search for the minimum number of days required to make m bouquets. Binary search is

chosen because we are dealing with an optimization problem where we want to minimize the number of days we have to wait.

Here's the intuition for why binary search can be applied: 1. We know the possible range of days is between the day the earliest flower blooms (minimum value in bloomDay) and the day the

With binary search, we can efficiently narrow down the range to find the minimum number of days required. The check function

step-by-step walk-through of how the algorithm works:

last flower blooms (maximum value in bloomDay).

- 2. If we can make m bouquets on a certain day 'x', it means we can also do it on any day after 'x' (as more flowers would be bloomed). 3. Inversely, if we cannot make m bouquets on a certain day 'y', we will not be able to do it on any day before 'y'.
- bloomDay and counting flowers bloomed. We increment the count cur for consecutive bloomed flowers and when cur equals k, we increment the number of bouquets cnt. If cnt meets or exceeds m, the function returns True; otherwise, it returns False. The main function then uses this check function to adjust the binary search boundaries (left and right) until the minimum day is

inside the Solution class is used to verify if a certain number of days day is sufficient to make m bouquets by scanning through

Solution Approach

The implementation of the solution makes use of a binary search algorithm and a counter pattern to verify the conditions. Below is a

1. We begin by checking if it is even possible to make m bouquets. Since each bouquet requires k flowers, if m \* k is greater than

o It initializes two counters: cnt (the number of bouquets that can be made by the day day) and cur (the number of adjacent

o If, by the end of the array, cnt is greater than or equal to m, it means we can make the required number of bouquets by the

While left is less than right, we calculate the midpoint mid using the expression (left + right) >> 1 which is equivalent

immediately return -1. 2. The check function is defined to determine if m bouquets can be made by a given day, say day. Here's how it works:

the total number of flowers available (len(bloomDay)), it is impossible to make the required number of bouquets and we

## The function goes through each flower's bloom day in bloomDay. For each flower, if it is bloomed by the day day (bd <= day), we increase cur by 1, signifying that this flower can be part of a</li>

current bouquet. o If cur becomes equal to k, then we have found k adjacent flowers, and therefore we increment cut by 1 (signifying one complete bouquet) and reset cur to 0 to start counting for the next bouquet.

- 3. After defining the check function, binary search is employed to find the minimum number of days required to make m bouquets. Here's how: Set left to the minimum value in bloomDay (the earliest possible day a flower can bloom) and right to the maximum value in
- to (left + right) // 2. • Utilize the check function on mid. If it returns True, it indicates that we can potentially make the bouquets even earlier, so we

set right to mid. Otherwise, if check returns False, it means we need more time, so we update left to mid + 1.

With each iteration, we effectively halve the search space, leading to a time complexity of O(log(max(bloomDay) - min(bloomDay))) \* O(n), where O(n) is the time complexity of the check function since it scans through the entire bloomDay array.

This process continues to narrow down the range until left equals right, which would be the minimum number of days

bouquets with 2 flowers each. Step 1: Preliminary Check

First, we verify whether it is possible to make m bouquets from the available number of flowers. We have m \* k = 2 \* 2 = 4 flowers

needed and there are 5 flowers in total (len(bloomDay) = 5). Since we have more flowers than needed, it's possible to proceed.

Let's go through an example to illustrate the solution approach with bloomDay = [7, 12, 9, 5, 4], m = 2 bouquets, and k = 2

adjacent flowers needed for each bouquet. We want to find out the minimum number of days we need to wait to make exactly 2

Step 3: Binary Search and the check Function

■ Flower 4 (bloomDay [3]): Blooms by day 8? Yes. cur is now 1 (reset because the previous flower wasn't bloomed by day

 $\circ$  We have finished checking all flowers and found cnt = 1, which is less than m = 2. Therefore, we cannot make enough

Now, we prepare for binary search. The earliest a flower can bloom is day 4, and the latest is day 12. So we set left to 4 and right to

#### bouquets by day 8. Since check returned False, we cannot make 2 bouquets by day 8, so we move our left boundary up. left is now 9.

Function check with day = 10:

Flower 2 blooms by day 10? No.

 Loop through bloomDay: ■ Flower 1 blooms by day 10? Yes. cur is 1.

New midpoint: mid = (left + right) // 2 = (9 + 12) // 2 = 10

We will use a binary search along with the check function to narrow down the days.

Midpoint: mid = (left + right) // 2 = (4 + 12) // 2 = 8

Starting from the first flower, we loop through bloomDay:

Flower 2 (bloomDay[1]): Blooms by day 8? No.

Flower 3 (bloomDay [2]): Blooms by day 8? No.

Function check: bloomDay = [7, 12, 9, 5, 4], day = 8

We use the check function to see if we can make 2 bouquets by day 8.

■ Flower 1 (bloomDay [0]): Blooms by day 8? Yes. cur is now 1.

■ Flower 5 (bloomDay [4]): Blooms by day 8? Yes. cur is now 2.

Now cur equals k, we have a complete bouquet. So cnt is now 1 and cur resets to 0.

 Flower 3 blooms by day 10? Yes. cur is 1 (reset because the previous flower wasn't bloomed by day 10). Flower 4 blooms by day 10? Yes. cur is now 2 (we have the second bouquet).

We can make the required number of bouquets by day 10, so check returns True.

We increment cnt to 2 since we have another complete bouquet.

def min\_days(self, bloom\_day: List[int], m: int, k: int) -> int:

# by checking the array using a sliding window technique

bouquets = 0 # number of bouquets that can be made

# Helper function to check if it's possible to make m bouquets in 'day'

flowers\_in\_row = 0 # number of adjacent flowers blooming in a row

# If we have enough flowers in a row, we can make a bouquet

flowers\_in\_row = 0 # Reset counter if the flower won't bloom

# Start by setting the search space between the earliest and the latest bloom day

# When the search space is reduced to a single element, it's the minimum day

// Determines the minimum number of days required to get m bouquets of k consecutive flowers

// If there are not enough flowers to make the required bouquets, return -1

// Check if it's possible to make the bouquets by this day

// At this point, left is the minimum day on which we can make the bouquets

int bouquetsMade = 0; // Bouquets made so far

if (flowersCollected == flowersPerBouquet) {

for (int bloom : bloomDay) {

bouquetsMade++;

flowersCollected = 0;

return bouquetsMade >= bouquetCount;

int flowersCollected = 0; // Flowers in the current bouquet

// Helper function to verify if we can make the required number of bouquets by a given day

// Collect flower if bloomed by 'day' or reset the current bouquet progress

// If we have enough flowers for a bouquet, increment the count and reset flowersCollected

// Array of bloom days for individual flowers, number of bouquets to make, and number of flowers per bouquet

function minDays(bloomDay: number[], bouquetCount: number, flowersPerBouquet: number): number {

// Determine the lowest and highest bloom day to establish the binary search range

flowersCollected = (bloom <= day) ? (flowersCollected + 1) : 0;

// Check if we have made at least the required number of bouquets

// Cannot make the required number of bouquets with the available flowers

// Check if we can make the required bouquets by this midpoint day

// Can't make bouquets, seek a later day in the right half

if (canMakeBouquets(bloomDay, bouquetCount, flowersPerBouquet, mid)) {

// Helper function checks if the required number of bouquets can be made by a specific day

let flowersCollected = 0; // Count of flowers collected towards the current bouquet

// If flower has bloomed by the given day, add to current bouquet; otherwise, reset count

// Once enough flowers are collected for a bouquet, increment count and reset for next bouquet

function canMakeBouquets(bloomDay: number[], bouquetCount: number, flowersPerBouquet: number, day: number): boolean {

// Possible to make bouquets, try finding an earlier day in the left half

if (bouquetCount \* flowersPerBouquet > bloomDay.length) {

// Find the midpoint in the current search range

let mid = left + Math.floor((right - left) / 2);

let bouquetsMade = 0; // Count of bouquets successfully made

if (flowersCollected === flowersPerBouquet) {

flowersCollected = (bloom <= day) ? (flowersCollected + 1) : 0;

// Return true if we can make at least the required number of bouquets

flowersCollected = 0; // Start collecting for the next bouquet

check function has a time complexity of O(n), where n is the number of elements in bloomDay.

Therefore, the overall time complexity of the algorithm is 0(n \* log(max(bloomDay)) - min(bloomDay)).

// Iterate through the array of bloom days

bool canMakeBouquets(vector<int>& bloomDay, int bouquetCount, int flowersPerBouquet, int day) {

flowers\_in\_row = 0 # Reset for the next potential bouquet

# If the flower bloom day is less than or equal to the chosen day, it will bloom

# If it's impossible to make m bouquets, return -1

def can\_make\_bouquets(day: int) -> bool:

# Loop through each bloom day

flowers\_in\_row += 1

if flowers\_in\_row == k:

# Check if we can make at least m bouquets

# Continue searching while the search space is valid

# Take the mid-point of the search space

# Use binary search to find the minimum day to make all m bouquets

bouquets += 1

left, right = min(bloom\_day), max(bloom\_day)

public int minDays(int[] bloomDay, int m, int k) {

// Use binary search to find the minimum day

if (canMakeBouquets(bloomDay, m, k, mid)) {

int left = minDays, right = maxDays;

int mid = (left + right) >>> 1;

if (m \* k > bloomDay.length) {

return -1;

while (left < right) {</pre>

} else {

right = mid;

left = mid + 1;

for bloom in bloom\_day:

if bloom <= day:</pre>

else:

while left < right:</pre>

return left

return bouquets >= m

mid = (left + right) // 2

Python Solution

Since check returned True, we can potentially make the bouquets even earlier, so we move our right boundary down. right is

Since left is now equal to right which is 10, we have found the minimum number of days required to make m bouquets which is 10

38 # Check if we can make the required number of bouquets by this day if can\_make\_bouquets(mid): 39 right = mid # If yes, try to find a smaller day 40 else: 41 left = mid + 1 # If not, discard the left half of the search space 42 43

```
// Initialize the minimum and maximum days from the bloomDay array
9
           int minDays = Integer.MAX_VALUE, maxDays = Integer.MIN_VALUE;
10
            for (int day : bloomDay) {
11
12
               minDays = Math.min(minDays, day);
13
               maxDays = Math.max(maxDays, day);
14
```

Java Solution

class Solution {

```
24
25
26
           // The left boundary of binary search will be the answer
27
           return left;
28
29
       // Helper method to check if it's possible to make the required number of bouquets by day 'currentDay'
30
       private boolean canMakeBouquets(int[] bloomDay, int numBouquets, int bouquetSize, int currentDay) {
31
32
            int bouquetsMade = 0, flowersInBouquet = 0;
33
            for (int day : bloomDay) {
34
               // If the flower is bloomed by 'currentDay', include it in the current bouquet
35
                if (day <= currentDay) {</pre>
36
                    flowersInBouquet++;
37
                    // If we've gathered enough flowers for a bouquet, increase the count and reset
38
                    if (flowersInBouquet == bouquetSize) {
39
                        bouquetsMade++;
40
                        flowersInBouquet = 0;
41
42
                } else {
                    // Flower is not bloomed, reset the count for the current bouquet
43
                    flowersInBouquet = 0;
44
45
46
47
           // Check if we have made at least 'numBouquets' bouquets
48
            return bouquetsMade >= numBouquets;
49
50 }
51
C++ Solution
   class Solution {
   public:
        int minDays(vector<int>& bloomDay, int bouquetCount, int flowersPerBouquet) {
           // If we cannot make the required number of bouquets, return -1
            if (bouquetCount * flowersPerBouquet > bloomDay.size()) {
                return -1;
           // Set initial search range for binary search
           int minDay = *min_element(bloomDay.begin(), bloomDay.end());
10
            int maxDay = *max_element(bloomDay.begin(), bloomDay.end());
11
12
13
            int left = minDay, right = maxDay;
           while (left < right) {</pre>
14
               // Find the mid value of our search range
15
                int mid = left + (right - left) / 2;
16
17
18
               // Check if it is possible to make the required bouquets by mid day
               if (canMakeBouquets(bloomDay, bouquetCount, flowersPerBouquet, mid)) {
19
                    right = mid; // It is possible, look for a smaller day in the left half
20
                } else {
21
22
                    left = mid + 1; // Not possible, look for a possible day in the right half
23
24
25
```

#### 24 left = mid + 1;25 26 27 28 // 'left' is the earliest day we can create the required bouquets 29 return left;

} else {

The code presented implements a binary search algorithm to find the minimum day on which we can obtain m bouquets of flowers, each with k blooms.

### The primary operation in this algorithm is the binary search, which operates between the range of the minimum and maximum bloom day. This operation takes O(log(max(bloomDay) - min(bloomDay))). Inside each iteration of the binary search, the check function is called, which iterates over all the bloom days to determine if it's possible to make m bouquets. The iteration over bloom days in the

Time Complexity

**Space Complexity** The space complexity of this algorithm is determined primarily by the variables used for keeping track of the binary search bounds and the count variables within the check function. Since the algorithm only uses a constant amount of extra space, regardless of the input size, the space complexity is 0(1).

```
We are also given two integers m (the number of bouquets we want to create) and k (the number of adjacent flowers needed to
```

**Problem Description**