Allocate Minimum Number of Pages

Problem Statement: Given an array ‘arr of integer numbers, ‘ar[i]’ represents the number of pages in the ‘i-th’ book. There are a ‘m’ number of students, and the task is to allocate all the books to the students.
Allocate books in such a way that:

1. Each student gets at least one book.
2. Each book should be allocated to only one student.
3. Book allocation should be in a contiguous manner.

You have to allocate the book to ‘m’ students such that the maximum number of pages assigned to a student is minimum. If the allocation of books is not possible. return -1

Example 1:

Input Format: n = 4, m = 2, arr[] = {12, 34, 67, 90}

Result: 113

Explanation: The allocation of books will be 12, 34, 67 | 90. One student will get the first 3 books and the other will get the last one.

Example 2:

Input Format: n = 5, m = 4, arr[] = {25, 46, 28, 49, 24}

Result: 71

Explanation: The allocation of books will be 25, 46 | 28 | 49 | 24.

We can allocate books in several ways but it is clearly said in the question that we have to allocate the books in such a way that the maximum number of pages received by a student should be minimum.

Assume the given array is {25 46 28 49 24} and number of students, M = 4. Now, we can allocate these books in different ways. Some of them are the following:

- 25 | 46 | 28 | 49, 24 → Maximum no. of pages a student receive = 73
- 25 | 46 | 28, 49 | 24 → Maximum no. of pages a student receive = 77
- 25 | 46, 28 | 49 | 24 → Maximum no. of pages a student receive = 74
- 25, 46 | 28 | 49 | 24 → Maximum no. of pages a student receive = 71

From the above allocations, we can clearly observe that the minimum possible maximum number of pages is 71.

When it is impossible to allocate books:

When the number of books is lesser than the number of students, we cannot allocate books to all the students even if we give only a single book to each student. So, ***if m > n, we should return -1.***

Observations:

- **Minimum possible answer:** We will get the minimum answer when we give n books of the array to n students(*i.e. Each student will receive 1 book*). Now, in this case, the maximum number of pages will be the maximum element in the array. So, the minimum possible answer is max(arr[]).
- **Maximum possible answer:** We will get the maximum answer when we give all n books to a single student. The maximum no. of pages he/she will receive is the summation of array elements i.e. sum(arr[]). So, the maximum possible answer is sum(arr[]).

From the observations, it is clear that our answer lies in the **range [max(arr[]), sum(arr[])].**

How to calculate the number of students to whom we can allocate the books if one can receive at most 'pages' number of pages:

In order to calculate the number of students we will write a function, **countStudents()**. This function will take the array and 'pages' as parameters and return the number of students to whom we can allocate the books.

countStudents(arr[], pages):

1. We will first declare two variables i.e. 'students'(*stores the no. of students*), and pagesStudent(*stores the number of pages of a student*). As we are starting with the first student, 'students' should be initialized with 1.
2. We will start traversing the given array.
3. **If pagesStudent + arr[i] <= pages:** If upon adding the pages with the existing number of pages does not exceed the limit, we can allocate this i-th book to the current student.
4. **Otherwise,** we will move to the next student(*i.e. students += 1*) and allocate the book.

Finally, we will return the value of 'students'.

Naive Approach:

The extremely naive approach is to check all possible pages from max(arr[]) to sum(arr[]). The minimum pages for which we can allocate all the books to M students will be our answer.

Algorithm:

1. **If m > n:** In this case, book allocation is not possible and so, we will return -1.
2. Next, we will find the maximum element and the summation of the given array.
3. We will use a loop(say **pages**) to check all possible pages from max(arr[]) to sum(arr[]).

4. Next, inside the loop, we will send each ‘pages’, to the function **countStudents()** function to get the number of students to whom we can allocate the books.
    1. The first number of pages, ‘pages’, for which the number of students will be equal to ‘m’, will be our answer. So, we will return that particular ‘pages’.
5. Finally, if we are out of the loop, we will return max(arr[]) as there cannot exist any answer smaller than that.

```
#include <bits/stdc++.h>

using namespace std;


int countStudents(vector<int> &arr, int pages) {

    int n = arr.size(); //size of array.

    int students = 1;

    long long pagesStudent = 0;

    for (int i = 0; i < n; i++) {

        if (pagesStudent + arr[i] <= pages) {

            //add pages to current student

            pagesStudent += arr[i];
```

```
        }
        else {
            //add pages to next student
            students++;
            pagesStudent = arr[i];
        }
    }
    return students;
}

int findPages(vector<int>& arr, int n, int m) {
    //book allocation impossible:
    if (m > n) return -1;

    int low = *max_element(arr.begin(), arr.end());
    int high = accumulate(arr.begin(), arr.end(), 0);

    for (int pages = low; pages <= high; pages++) {
        if (countStudents(arr, pages) == m) {
```

```
            return pages;
        }
    }
    return low;
}

int main()
{
    vector<int> arr = {25, 46, 28, 49, 24};
    int n = 5;
    int m = 4;
    int ans = findPages(arr, n, m);
    cout << "The answer is: " << ans << "\n";
    return 0;
}
```

Complexity Analysis

Time Complexity: O(N * (sum(arr[])-max(arr[])+1)), where N = size of the array, sum(arr[]) = sum of all array elements, max(arr[]) = maximum of all array elements.
Reason: We are using a loop from max(arr[]) to sum(arr[]) to check all possible numbers of pages. Inside the loop, we are

calling the countStudents() function for each number. Now, inside the countStudents() function, we are using a loop that runs for N times.

Space Complexity: O(1) as we are not using any extra space to solve this problem.

Algorithm / Intuition

Optimal Approach:

We are going to use the Binary Search algorithm to optimize the approach.

The primary objective of the Binary Search algorithm is to efficiently determine the appropriate half to eliminate, thereby reducing the search space by half. It does this by determining a specific condition that ensures that the target is not present in that half.

Upon closer observation, we can recognize that our answer space, represented as [max(arr[]), sum(arr[])], is actually sorted. Additionally, we can identify a pattern that allows us to divide this space into two halves: one consisting of potential answers and the other of non-viable options. So, we will apply binary search on the answer space.

Algorithm:

1. **If m > n:** In this case, book allocation is not possible and so, we will return -1.
2. **Place the 2 pointers i.e. low and high:** Initially, we will place the pointers. The pointer low will point to max(arr[]) and the high will point to sum(arr[]).
3. **Calculate the 'mid':** Now, inside the loop, we will calculate the value of 'mid' using the following formula:
   mid = (low+high) // 2 ('//' refers to integer division)

4. **Eliminate the halves based on the number of students returned by countStudents():**
   We will pass the potential number of pages, represented by the variable 'mid', to the '**countStudents()**' function. This function will return the number of students to whom we can allocate the books.
    1. **If students > m:** On satisfying this condition, we can conclude that the number 'mid' is smaller than our answer. So, we will eliminate the left half and consider the right half(i.e. low = mid+1).
    2. **Otherwise,** the value mid is one of the possible answers. But we want the minimum value. So, we will eliminate the right half and consider the left half(i.e. high = mid-1).
5. Finally, outside the loop, we will return the value of low as the pointer will be pointing to the answer.

The steps from 3-4 will be inside a loop and the loop will continue until low crosses high.

Note: *Please make sure to refer to the video and try out some test cases of your own to understand, how the pointer 'low' will be always pointing to the answer in this case. This is also the reason we have not used any extra variable here to store the answer.*

```
#include <bits/stdc++.h>

using namespace std;


int countStudents(vector<int> &arr, int pages) {
    int n = arr.size(); //size of array.
    int students = 1;
    long long pagesStudent = 0;
    for (int i = 0; i < n; i++) {
        if (pagesStudent + arr[i] <= pages) {
            //add pages to current student
```

```
            pagesStudent += arr[i];
        }
        else {
            //add pages to next student
            students++;
            pagesStudent = arr[i];
        }
    }
    return students;
}

int findPages(vector<int>& arr, int n, int m) {
    //book allocation impossible:
    if (m > n) return -1;

    int low = *max_element(arr.begin(), arr.end());
    int high = accumulate(arr.begin(), arr.end(), 0);
    while (low <= high) {
        int mid = (low + high) / 2;
        int students = countStudents(arr, mid);
        if (students > m) {
            low = mid + 1;
        }
        else {
            high = mid - 1;
        }
    }
    return low;
}
```

```
int main()
{
   vector<int> arr = {25, 46, 28, 49, 24};
   int n = 5;
   int m = 4;
   int ans = findPages(arr, n, m);
   cout << "The answer is: " << ans << "\n";
   return 0;
}
```

Complexity Analysis

Time Complexity: O(N * log(sum(arr[])-max(arr[])+1)), where N = size of the array, sum(arr[]) = sum of all array elements, max(arr[]) = maximum of all array elements.
Reason: We are applying binary search on [max(arr[]), sum(arr[])]. Inside the loop, we are calling the countStudents() function for the value of 'mid'. Now, inside the countStudents() function, we are using a loop that runs for N times.

Space Complexity: O(1) as we are not using any extra space to solve this problem.