Allocate Minimum Pages

Given a number of pages in **N** different books and **M** students. The books are arranged in ascending order of the number of pages. Every student is assigned to read some consecutive books. The task is to assign books in such a way that the maximum number of pages assigned to a student is minimum.

Example :

```
Input : pages[] = {12, 34, 67, 90} , m = 2

Output : 113

Explanation: There are 2 number of students. Books can be distributed in following fashion :
1) [12] and [34, 67, 90]
Max number of pages is allocated to student ‘2’ with 34 + 67 + 90 = 191 pages
2) [12, 34] and [67, 90] Max number of pages is allocated to student ‘2’ with 67 + 90 = 157 pages
3) [12, 34, 67] and [90] Max number of pages is allocated to student ‘1’ with 12 + 34 + 67 = 113 pages
Of the 3 cases, Option 3 has the minimum pages = 113.
```

Naive Approach:

C++ Java

```
import java.util.*;
import java.io.*;
import java.lang.*;

class GFG {

    public static void main(String args[])
    {
        int arr[]={10,20,10,30};
        int n=arr.length;
        int k=2;

        System.out.print(minPages(arr,n,k));
    }

    public static int sum(int arr[],int b, int e){
        int s=0;
        for(int i=b;i<=e;i++)
            s+=arr[i];
        return s;
    }

    public static int minPages(int arr[],int n, int k){
        if(k==1)
            return sum(arr,0,n-1);
        if(n==1)
            return arr[0];
        int res=Integer.MAX_VALUE;
```

```
        for(int i=1;i<n;i++){
            res=Math.min(res,Math.max(minPages(arr,i,k-1),sum(arr,i,n-1)));
        }
        return res;
    }
}
```


Output:

```
40
```

Approach: A Binary Search method for solving the book allocation problem:

Case 1: When no valid answer exists.

> *If the number of students is greater than the number of books **(i.e, M > N)**, In this case at least 1 student will be left to which no book has been assigned.*

Case 2: When a valid answer exists.

> *The maximum possible answer could be when there is only one student. So, all the book will be assigned to him and the result would be the sum of pages of all the books.*
>
> *The minimum possible answer could be when number of student is equal to the number of book (i.e, M == N) , In this case all the students will get at most one book. So, the result would be the maximum number of pages among them **(i.e, minimum(pages[])).***

> *Hence, we can apply binary search in this given range and each time we can consider the mid value as the maximum limit of pages one can get. And check for the limit if answer is valid then update the limit accordingly.*

Below is the implementation of the above idea:

- Initialise the **start** to **minimum(pages[])** and **end** = sum of **pages[]**,
- Do while start <= end
  - Calculate the mid and check if **mid** number of pages can assign any student by satisfying the given condition such that all students will get at least one book. Follow the steps to check for validity.
    - Initialise the **studentsRequired = 1** and **curr_sum = 0** for sum of consecutive pages of book
    - Iterate over all books or say pages[]
      - Add the pages to curr_sum and check **curr_sum > curr_min** then increment the count of **studentRequired** by 1.
      - Check if the studentRequired > M, return false.
    - Return true.
  - If mid is valid then, update the **result** and move the **end = mid – 1**
  - Otherwise, move the **start = mid + 1**
- Finally, return the **result**.

Below is the implementation of the above approach:

C++ Java

```
// Java program for optimal allocation of pages

public class GFG {
    // Utility method to check if current minimum value
    // is feasible or not.
```

```
static boolean isPossible(int arr[], int n, int m,
                          int curr_min)
{
    int studentsRequired = 1;
    int curr_sum = 0;

    // iterate over all books
    for (int i = 0; i < n; i++) {
        curr_sum += arr[i];
        if (curr_sum > curr_min) {
            studentsRequired++; // increment student
                                // count

            curr_sum = arr[i]; // update curr_sum
        }
    }
```



```
// method to find minimum pages
static int findPages(int arr[], int n, int m)
{
    int sum = 0;

    // return -1 if no. of books is less than
    // no. of students
    if (n < m)
        return -1;
```

```
// Count total number of pages
for (int i = 0; i < n; i++)
    sum += arr[i];

// initialize start as arr[n-1] pages(minimum answer
// possible) and end as total pages(maximum answer
// possible)
int start = arr[n - 1], end = sum;
int result = Integer.MAX_VALUE;

// traverse until start <= end
while (start <= end) {
    // check if it is possible to distribute
    // books by using mid is current minimum
    int mid = start + (end - start) / 2;
    if (isPossible(arr, n, m, mid)) {
        // update result to current distribution
        // as it's the best we have found till now.
        result = mid;

        // as we are finding minimum so,
        end = mid - 1;
    }

    else
        // if not possible, means pages should be
        // increased ,so update start = mid + 1
        start = mid + 1;
```

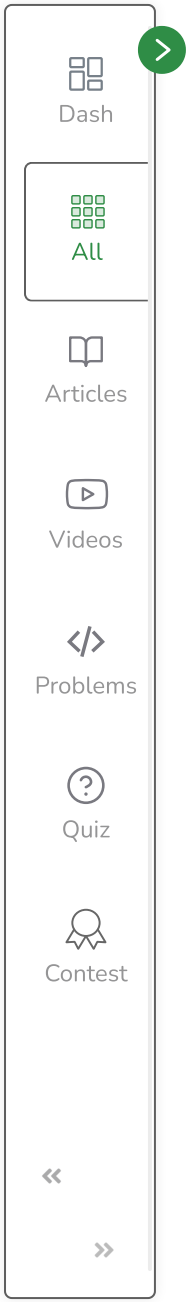
```
        }

        // at-last return minimum no. of pages
        return result;
    }

    // Driver Method
    public static void main(String[] args)
    {

        int arr[] = { 12, 34, 67,
                      90 }; // Number of pages in books

        int m = 2; // No. of students

        System.out.println("Minimum number of pages = "
                           + findPages(arr, arr.length, m));
    }
}
```

Output

```
Minimum number of pages = 113
```

Time Complexity: O(N*log(N)), Where N is the total number of pages in the book.
Auxiliary Space: O(1)

Marked as Read



Report An Issue

If you are facing any issue on this page. Please let us know.