DAA Assignment 4

Code 1:

IIT2022004_1.cpp U | IIT2022004_2.cpp U

IIT2022004_1.cpp > merge(vector<int>&, int, int, int)

```cpp

#include <bits/stdc++.h>
using namespace std;

int merge(vector<int> &arr, int low, int mid, int high)
{
    vector<int> temp;    // temporary array
    int left = low;      // starting index of left half of arr
    int right = mid + 1; // starting index of right half of arr

    int cnt = 0;

    while (left <= mid && right <= high)
    {
        if (arr[left] <= arr[right])
        {
            temp.push_back(arr[left]);
            left++;
        }
        else
        {
            temp.push_back(arr[right]);
            cnt += (mid - left + 1);
            right++;
        }
    }

    while (left <= mid)
    {
        temp.push_back(arr[left]);
        left++;
    }

    while (right <= high)
    {
        temp.push_back(arr[right]);
        right++;
    }

    for (int i = low; i <= high; i++)
    {
        arr[i] = temp[i - low];
    }

    return cnt; // Modification 3
}

```

```
46      }
47
48      int mergeSort(vector<int> &arr, int low, int high)
49      {
50          int cnt = 0;
51          if (low >= high)
52              return cnt;
53          int mid = (low + high) / 2;
54          cnt += mergeSort(arr, low, mid);
55          cnt += mergeSort(arr, mid + 1, high);
56          cnt += merge(arr, low, mid, high);
57          return cnt;
58      }
59
60      int numberOfInversions(vector<int> &a, int n)
61      {
62
63          // Count the number of pairs:
64          return mergeSort(a, 0, n - 1);
65      }
66
67      int main()
68      {
69          int n;
70          cout << "Number of elements: ";
71          cin >> n;
72          vector<int> a(n);
73          for (int i = 0; i < n; i++)
74          {
75              cin >> a[i];
76          }
77          int cnt = numberOfInversions(a, n);
78          cout << "The number of inversions are: "
79               << cnt << endl;
80          return 0;
81      }
82
```

Analysis:

Time Complexity:

The time complexity of this algorithm is O(n log n) due to the merge-sort algorithmstep.

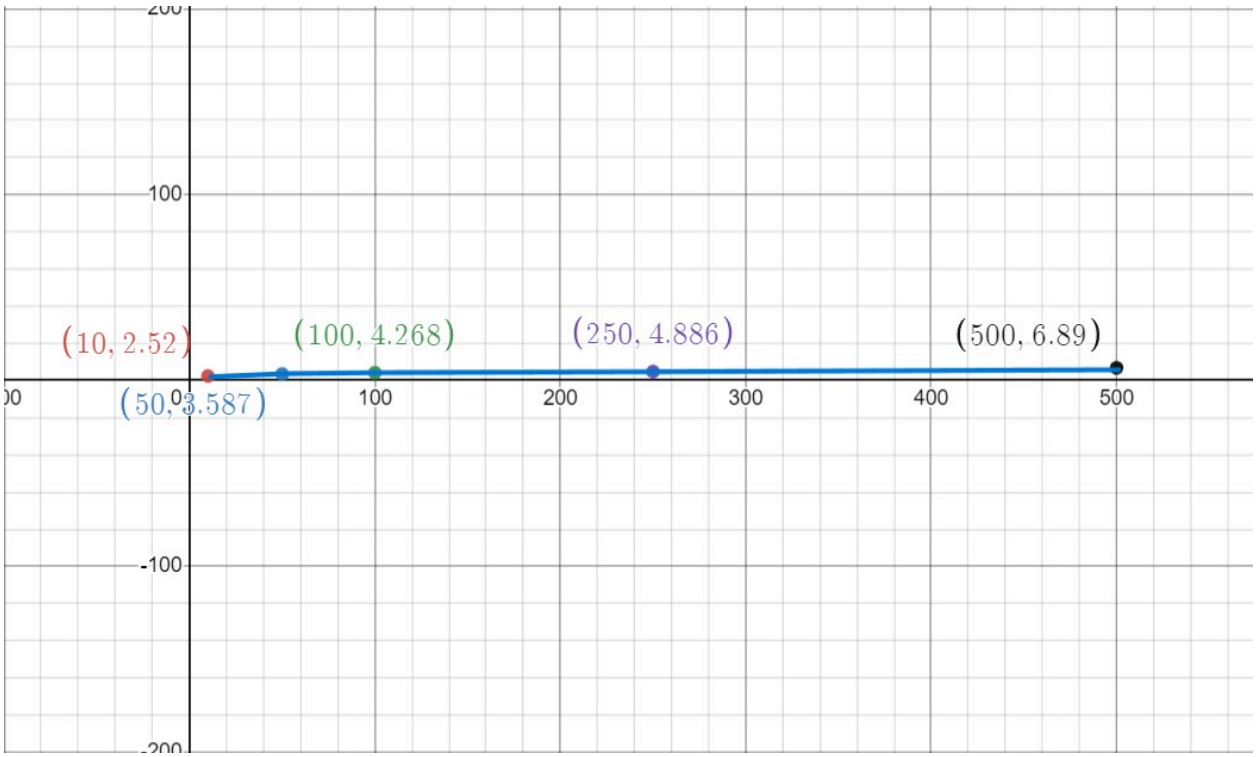
Space Complexity:

The space complexity is O(n) due to the additional space required for the temporary arrays during the merge operation.

Explanation of Efficiency:
The key idea is that during the merge step, when merging two sorted subarrays, we can efficiently count the number of inversions by observing that if an element from the right subarray is smaller than an element from the left subarray, it is also smaller than all the remaining elements in the left subarray. This observation helps in counting inversions without comparing all pairs, leading to an efficient O(n log n) solution.

Using brute-force, we'd have to use an O(n^2) algorithm to check the number of inversions for each number in the array.

Graph:



Code 2:

IIT2022004_1.cpp U IIT2022004_2.cpp U

IIT2022004_2.cpp > isPossible(int [], int, int, int)

```cpp
#include <bits/stdc++.h>
using namespace std;

bool isPossible(int arr[], int n, int m, int minOfMax)
{
    int stuReq = 1;
    int curr_sum = 0;

    for (int i = 0; i < n; i++)
    {
        if (arr[i] > minOfMax)
            return false;

        if (curr_sum + arr[i] > minOfMax)
        {
            stuReq++;

            curr_sum = arr[i];
            if (stuReq > m)
                return false;
        }
        else
            curr_sum += arr[i];
    }
    return true;
}

int findPages(int arr[], int n, int m)
{
    long long sum = 0;
```

```
int findPages(int arr[], int n, int m)
{
    long long sum = 0;
    if (n < m)
        return -1;
    int mx = INT_MIN;
    for (int i = 0; i < n; i++)
    {
        sum += arr[i];
        mx = max(mx, arr[i]);
    }
    int start = mx, end = sum;
    int result = INT_MAX;
    while (start <= end)
    {
        int mid = (start + end) / 2;
        if (isPossible(arr, n, m, mid))
        {
            result = mid;
            end = mid - 1;
        }
        else
            start = mid + 1;
    }
    return result;
}

int main()
{
    int n;
    cout << "Number of books: ";
    cin >> n;
    int m;
    cout << "Number of students: ";
    cin >> m;
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        cin >> arr[i];
    }
    cout << "Minimum number of pages = "
         << findPages(arr, n, m) << endl;
    return 0;
}
```

Analysis

Time Complexity:

The function isPossible has a time complexity of O(n), where n is the number of books. It iterates over all the books once.
The findPages function performs binary search on the range of possible minimum pages, which takes O(log(sum of pages)) time.
Overall, the time complexity of the algorithm is O(n log(sum of pages)).
Space Complexity:
The space complexity of the algorithm is O(1) because it uses a constant amount of extra space irrespective of the input size. The algorithm does not use any additional data structures that scale with the input size.

Efficiency:
The binary search approach is efficient for this problem because it allows us to quickly narrow down the search space for the minimum number of pages. The algorithm sorts the array of pages, which takes O(n log n) time, but this sorting step is outweighed by the subsequent binary search.

Compared to other potential methods, this algorithm is relatively efficient. Sorting the array and then applying binary search is a common and effective technique for optimization problems like this one.

Graph: