Divide and Conquer

[**Learn more about Divide and Conquer Algorithms in DSA Self Paced Course**](https://practice.geeksforgeeks.org/courses/dsa-self-paced?&utm_source=geeksforgeeks&utm_medium=articles+divnconquer_lp+header_link_click&utm_campaign=dsa+course+tracker)

[**Practice Problems on Divide and Conquer**](https://practice.geeksforgeeks.org/explore/?category%5B%5D=Divide%20and%20Conquer&page=1&category%5B%5D=Divide%20and%20Conquer&utm_source=geeksforgeeks&utm_medium=articles+divnconquer_lp+header_link_click&utm_campaign=practice+tracker)

[**Recent Articles on Divide and Conquer**](https://www.geeksforgeeks.org/category/algorithm/divide-and-conquer/)

**What is Divide and Conquer?**

*Divide and Conquer is an algorithmic paradigm in which the problem is solved using the Divide, Conquer, and Combine strategy.*

A typical Divide and Conquer algorithm solves a problem using following three steps:

1. **Divide:**This involves dividing the problem into smaller sub-problems.
2. **Conquer:**Solve sub-problems by calling recursively until solved.
3. **Combine:**Combine the sub-problems to get the final solution of the whole problem.

**Standard algorithms that follow Divide and Conquer algorithm**

The following are some standard algorithms that follow Divide and Conquer algorithm.

1. [**Quicksort**](https://www.geeksforgeeks.org/quick-sort/) is a sorting algorithm. The algorithm picks a pivot element and rearranges the array elements so that all elements smaller than the picked pivot element move to the left side of the pivot, and all greater elements move to the right side. Finally, the algorithm recursively sorts the subarrays on the left and right of the pivot element.
2. [**Merge Sort**](https://www.geeksforgeeks.org/merge-sort/) is also a sorting algorithm. The algorithm divides the array into two halves, recursively sorts them, and finally merges the two sorted halves.
3. [**Closest Pair of Points**](https://www.geeksforgeeks.org/closest-pair-of-points-using-divide-and-conquer-algorithm/) The problem is to find the closest pair of points in a set of points in the x-y plane. The problem can be solved in O(n^2) time by calculating the distances of every pair of points and comparing the distances to find the minimum. The Divide and Conquer algorithm solves the problem in O(N log N) time.
4. [**Strassen’s Algorithm**](https://www.geeksforgeeks.org/strassens-matrix-multiplication/) is an efficient algorithm to multiply two matrices. A simple method to multiply two matrices needs 3 nested loops and is O(n^3). Strassen’s algorithm multiplies two matrices in O(n^2.8974) time.
5. [**Cooley–Tukey Fast Fourier Transform (FFT) algorithm**](http://en.wikipedia.org/wiki/Cooley%E2%80%93Tukey_FFT_algorithm) is the most common algorithm for FFT. It is a divide and conquer algorithm which works in O(N log N) time.
6. [**Karatsuba algorithm for fast multiplication**](https://www.geeksforgeeks.org/karatsuba-algorithm-for-fast-multiplication-using-divide-and-conquer-algorithm/) does the multiplication of two *n*-digit numbers in at most

 single-digit multiplications in general (and exactly

when *n* is a power of 2). It is, therefore, faster than the [classical](http://en.wikipedia.org/wiki/Long_multiplication)algorithm, which requires *n*2 single-digit products. If *n* = 210 = 1024, in particular, the exact counts are 310 = 59, 049 and (210)2 = 1, 048, 576, respectively.

**Example of Divide and Conquer algorithm**

A classic example of Divide and Conquer is [Merge Sort](https://www.geeksforgeeks.org/merge-sort/) demonstrated below. In Merge Sort, we divide array into two halves, sort the two halves recursively, and then merge the sorted halves.

[](https://media.geeksforgeeks.org/wp-content/cdn-uploads/Merge-Sort-Tutorial.png)

**Topics:**

* [Introduction](https://www.geeksforgeeks.org/divide-and-conquer/#introduction)
* [Some Standard algorithms](https://www.geeksforgeeks.org/divide-and-conquer/#standard)
* [Binary Search based problems](https://www.geeksforgeeks.org/divide-and-conquer/#bsproblems)
* [Some other Practice problems](https://www.geeksforgeeks.org/divide-and-conquer/#practice)

**Introduction:**

1. [Intoduction to Divide and Conquer](https://www.geeksforgeeks.org/divide-and-conquer-introduction/)
2. [Dynamic Programming vs Divide-and-Conquer](https://www.geeksforgeeks.org/dynamic-programming-vs-divide-and-conquer/)
3. [Decrease and Conquer](https://www.geeksforgeeks.org/decrease-and-conquer/)
4. [Advanced master theorem for divide and conquer recurrences](https://www.geeksforgeeks.org/advanced-master-theorem-for-divide-and-conquer-recurrences/)

**Some Standard algorithms:**

1. [Binary Search](https://www.geeksforgeeks.org/binary-search/)
2. [Merge Sort](https://www.geeksforgeeks.org/merge-sort/)
3. [Quick Sort](https://www.geeksforgeeks.org/quick-sort/)
4. [Calculate pow(x, n)](https://www.geeksforgeeks.org/write-a-c-program-to-calculate-powxn/)
5. [Karatsuba algorithm for fast multiplication](https://www.geeksforgeeks.org/divide-and-conquer-set-2-karatsuba-algorithm-for-fast-multiplication/)
6. [Strassen’s Matrix Multiplication](https://www.geeksforgeeks.org/strassens-matrix-multiplication/)
7. [Convex Hull (Simple Divide and Conquer Algorithm)](https://www.geeksforgeeks.org/convex-hull-simple-divide-conquer-algorithm/)
8. [Quickhull Algorithm for Convex Hull](https://www.geeksforgeeks.org/quickhull-algorithm-convex-hull/)

**Binary Search based problems:**

1. [Find a peak element in a given array](https://www.geeksforgeeks.org/find-a-peak-in-a-given-array/)
2. [Check for Majority Element in a sorted array](https://www.geeksforgeeks.org/check-for-majority-element-in-a-sorted-array/)
3. [K-th Element of Two Sorted Arrays](https://www.geeksforgeeks.org/k-th-element-two-sorted-arrays/)
4. [Find the number of zeroes](https://www.geeksforgeeks.org/find-number-zeroes/)
5. [Find the Rotation Count in Rotated Sorted array](https://www.geeksforgeeks.org/find-rotation-count-rotated-sorted-array/)
6. [Find the point where a monotonically increasing function becomes positive first time](https://www.geeksforgeeks.org/find-the-point-where-a-function-becomes-negative/)
7. [Median of two sorted arrays](https://www.geeksforgeeks.org/median-of-two-sorted-arrays/)
8. [Median of two sorted arrays of different sizes](https://www.geeksforgeeks.org/median-of-two-sorted-arrays-of-different-sizes/)
9. [The painter’s partition problem using Binary Search](https://www.geeksforgeeks.org/the-painters-partition-problem-using-binary-search/)

**Some other Practic problems:**

1. [Square root of an integer](https://www.geeksforgeeks.org/square-root-of-an-integer/)
2. [Maximum and minimum of an array using minimum number of comparisons](https://www.geeksforgeeks.org/maximum-and-minimum-in-an-array/)
3. [Find frequency of each element in a limited range array in less than O(n) time](https://www.geeksforgeeks.org/find-frequency-of-each-element-in-a-limited-range-array-in-less-than-on-time/)
4. [Tiling Problem](https://www.geeksforgeeks.org/divide-and-conquer-set-6-tiling-problem/)
5. [Count Inversions](https://www.geeksforgeeks.org/counting-inversions/)
6. [The Skyline Problem](https://www.geeksforgeeks.org/divide-and-conquer-set-7-the-skyline-problem/)
7. [Search in a Row-wise and Column-wise Sorted 2D Array](https://www.geeksforgeeks.org/divide-conquer-set-6-search-row-wise-column-wise-sorted-2d-array/)
8. [Allocate minimum number of pages](https://www.geeksforgeeks.org/allocate-minimum-number-pages/)
9. [Modular Exponentiation (Power in Modular Arithmetic)](https://www.geeksforgeeks.org/modular-exponentiation-power-in-modular-arithmetic/)

**Quick Links :**

* [**Learn Data Structure and Algorithms | DSA Tutorial**](https://www.geeksforgeeks.org/learn-data-structures-and-algorithms-dsa-tutorial?utm_source=Website&utm_medium=Landing+Page+Click&utm_campaign=DSA+Page+Tracker&utm_id=DSA-Page-Tracker&utm_term=DSA+Page+Promo&utm_content=Course+Page)
* [‘Practice Problems’ on Divide and Conquer](https://practice.geeksforgeeks.org/topics/Divide-and-Conquer/)
* [‘Quizzes’ on Divide and Conquer](https://www.geeksforgeeks.org/divide-and-conquer/)

**Easy Questions:**

**Square root of an integer**

Given an integer **X**, find its square root. If **X** is not a perfect square, then return **floor(√x)**.

**Examples :**

***Input:****x = 4*

***Output:****2*

***Explanation:****The square root of 4 is 2.*

***Input:****x = 11*

***Output:****3*

***Explanation:****The square root of 11 lies in between 3 and 4 so floor of the square root is 3.*

Recommended Problem

Square root of a number

**Naive Approach:** To find the floor of the square root, try with all-natural numbers starting from 1. Continue incrementing the number until the square of that number is greater than the given number.

Follow the steps below to implement the above idea

1. Create a variable (counter)***i*** and take care of some base cases, (i.e when the given number is 0 or 1).
2. Run a loop until***i\*i <= n***, where n is the given number. Increment i by 1.
3. The floor of the square root of the number is *i – 1*

Below is the implementation of the above approach:

# Python3 program to find floor(sqrt(x)

# Returns floor of square root of x

**def** floorSqrt(x):

    # Base cases

**if** (x **==** 0 **or** x **==** 1):

**return** x

    # Starting from 1, try all numbers until

    # i\*i is greater than or equal to x.

    i **=** 1

    result **=** 1

**while** (result <**=** x):

        i **+=** 1

        result **=** i **\*** i

**return** i **-** 1

# Driver Code

x **=** 11

print(floorSqrt(x))

# This code is contributed by Smitha Dinesh Semwal.

**Output**

3

**Complexity Analysis:**

1. **Time Complexity:** O(√X). Only one traversal of the solution is needed, so the time complexity is O(√X).
2. **Auxiliary Space:** O(1).

*Thanks, Fattepur Mahesh for suggesting this solution.*

**Square root an integer using Binary search:**

*The idea is to find the largest integer****i****whose square is less than or equal to the given number. The values of****i \* i****is monotonically increasing, so the problem can be solved using binary search.*

**Below is the implementation of the above idea:**

1. Base cases for the given problem are when the given number is **0** or **1**, then return **X**;
2. Create some variables, for storing the lower bound say***l = 0,****and for upper bound****r = X / 2****(i.e, The floor of the square root of x cannot be more than x/2 when x > 1).*
3. Run a loop until***l <= r***, the search space vanishes
4. Check if the square of mid **(*mid = (l + r)/2*)**is less than or equal to **X**, If yes then search for a larger value in the second half of the search space, i.e**l = mid + 1**, update **ans = mid**
5. Else if the square of mid is more than **X** then search for a smaller value in the first half of the search space, i.e **r = mid – 1**
6. Finally, Return the **ans**

Below is the implementation of the above approach:

# Python 3 program to find floor(sqrt(x)

# Returns floor of square root of x

**def** floorSqrt(x):

    # Base cases

**if** (x **==** 0 **or** x **==** 1):

**return** x

    # Do Binary Search for floor(sqrt(x))

    start **=** 1

    end **=** x**//**2

**while** (start <**=** end):

        mid **=** (start **+** end) **//** 2

        # If x is a perfect square

**if** (mid**\***mid **==** x):

**return** mid

        # Since we need floor, we update

        # answer when mid\*mid is smaller

        # than x, and move closer to sqrt(x)

**if** (mid **\*** mid < x):

            start **=** mid **+** 1

            ans **=** mid

**else**:

            # If mid\*mid is greater than x

            end **=** mid**-**1

**return** ans

# driver code

x **=** 11

print(floorSqrt(x))

# This code is contributed by Nikita Tiwari.

**Output**

3

**Complexity Analysis:**

* **Time Complexity:** O(log(X)).
* **Auxiliary Space:** O(1).

Thanks to [Gaurav Ahirwar](http://qa.geeksforgeeks.org/user/Mr.Lazy) for suggesting the above method.

**Square root an integer using built-in functions:**

Below is the implementation for finding the square root using the built-in function.

**def** countSquares(x):

    sqrt **=** x**\*\***0.5

    result **=** int(sqrt)

**return** result

x **=** 9

print(countSquares(x))

**Output**

3

**Time Complexity:** O(log(X))

**Auxiliary Space:** O(1)

There can be many ways to solve this problem. For example, the [Babylonian Method](https://www.geeksforgeeks.org/square-root-of-a-perfect-square/) is one way.

**Another Approach to Solve This Problem Using Exponential Function:**

*The basic idea behind the method is to calculate the exponential of the logarithm of the integer divided by two.*

**Below are steps to implement the above approach:**

* Take the integer value as input and save it in a variable.
* Use the exponential function exp() and the logarithmic function log() from the <cmath> library to calculate the square root of the integer. exp(log(x) / 2) will give the square root of x.
* Use the floor() function to get the integer part of the result.
* Check whether the square of the floor result is equal to the input x.
* If the square of the floor result is equal to the input x, then return the floor result as it is the square root of x.
* If the square of the floor result is not equal to the input x, then return the floor result as the floor of the square root

**Below is the implementation of the above approach:**

# Python code to implement the above approach

**import** math

**def** findSquareRoot(x):

    # using exponential and logarithmic function to

    # calculate square root of x

    result **=** math.exp(math.log(x) **/** 2)

    # floor function to get integer part of the result

    floorResult **=** math.floor(result)

    # If the integer square of the floor result is equal to

    # the input x,

    # then x is a perfect square, and floor result is the

    # square root.

**if** floorResult **\*** floorResult **==** x:

**return** floorResult

**else**:

        # If not, then x is not a perfect square, and

        # floor result is the floor of the square root.

**return** floorResult

# Driver code

x **=** 11

# calling the findSquareRoot function to calculate the square root

squareRoot **=** findSquareRoot(x)

print(squareRoot)  # printing the result

**Output**

3

**Time Complexity:** O(1), The time complexity of the given approach is O(1) since it uses only one mathematical formula exp(log(x) / 2) which is constant time, and a few arithmetic operations, comparisons, and function calls that take constant time as well. Therefore, the time complexity of this algorithm is constant or O(1).

**Space Complexity:** O(1), The space complexity of the given approach is O(1) as it only uses a constant amount of extra space. It declares two integer variables, result and floorResult, which each take constant space, and there is no dynamic memory allocation or recursive calls. Therefore, the space complexity of this algorithm is constant or O(1).

**Maximum and minimum of an array using minimum number of comparisons**

Given an array of size **N.**The task is to find the maximum and the minimum element of the array using the minimum number of comparisons.

**Examples:**

***Input:****arr[] = {3, 5, 4, 1, 9}*

***Output:****Minimum element is: 1*

*Maximum element is: 9*

***Input:****arr[] = {22, 14, 8, 17, 35, 3}*

***Output:****Minimum element is: 3*

*Maximum element is: 35*

Recommended Problem

First of all, how do we return multiple values from a function? We can do it either using structures or pointers.

We have created a structure named pair (which contains min and max) to return multiple values.

# Python3 implementation

**class** pair:

**def** \_\_init\_\_(self):

        self.min **=** None

        self.max **=** None

# This code contributed by phasing17

**Maximum and minimum of an array using Linear search:**

*Initialize values of min and max as minimum and maximum of the first two elements respectively. Starting from 3rd, compare each element with max and min, and change max and min accordingly (i.e., if the element is smaller than min then change min, else if the element is greater than max then change max, else ignore the element)*

Below is the implementation of the above approach:

# Python program of above implementation

# structure is used to return two values from minMax()

**class** pair:

**def** \_\_init\_\_(self):

        self.min **=** 0

        self.max **=** 0

**def** getMinMax(arr: list, n: int) **-**> pair:

    minmax **=** pair()

    # If there is only one element then return it as min and max both

**if** n **==** 1:

        minmax.max **=** arr[0]

        minmax.min **=** arr[0]

**return** minmax

    # If there are more than one elements, then initialize min

    # and max

**if** arr[0] > arr[1]:

        minmax.max **=** arr[0]

        minmax.min **=** arr[1]

**else**:

        minmax.max **=** arr[1]

        minmax.min **=** arr[0]

**for** i **in** range(2, n):

**if** arr[i] > minmax.max:

            minmax.max **=** arr[i]

**elif** arr[i] < minmax.min:

            minmax.min **=** arr[i]

**return** minmax

# Driver Code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    arr **=** [1000, 11, 445, 1, 330, 3000]

    arr\_size **=** 6

    minmax **=** getMinMax(arr, arr\_size)

**print**("Minimum element is", minmax.min)

    print("Maximum element is", minmax.max)

# This code is contributed by

# sanjeev2552

**Output**

Minimum element is 1  
Maximum element is 3000

**Time Complexity:** O(n)

**Auxiliary Space:**O(1) as no extra space was needed.

In this method, the total number of comparisons is **1 + 2(n-2)** in the worst case and **1 + n – 2** in the best case.

In the above implementation, the worst case occurs when elements are sorted in descending order and the best case occurs when elements are sorted in ascending order.

**Maximum and minimum of an array using the tournament method:**

Divide the array into two parts and compare the maximums and minimums of the two parts to get the maximum and the minimum of the whole array.

*Pair MaxMin(array, array\_size)*

*if array\_size = 1*

*return element as both max and min*

*else if arry\_size = 2*

*one comparison to determine max and min*

*return that pair*

*else    /\* array\_size  > 2 \*/*

*recur for max and min of left half*

*recur for max and min of right half*

*one comparison determines true max of the two candidates*

*one comparison determines true min of the two candidates*

*return the pair of max and min*

Below is the implementation of the above approach:

# Python program of above implementation

**def** getMinMax(low, high, arr):

    arr\_max **=** arr[low]

    arr\_min **=** arr[low]

    # If there is only one element

**if** low **==** high:

        arr\_max **=** arr[low]

        arr\_min **=** arr[low]

**return** (arr\_max, arr\_min)

    # If there is only two element

**elif** high **==** low **+** 1:

**if** arr[low] > arr[high]:

            arr\_max **=** arr[low]

            arr\_min **=** arr[high]

**else**:

            arr\_max **=** arr[high]

            arr\_min **=** arr[low]

**return** (arr\_max, arr\_min)

**else**:

        # If there are more than 2 elements

        mid **=** int((low **+** high) **/** 2)

        arr\_max1, arr\_min1 **=** getMinMax(low, mid, arr)

        arr\_max2, arr\_min2 **=** getMinMax(mid **+** 1, high, arr)

**return** (max(arr\_max1, arr\_max2), min(arr\_min1, arr\_min2))

# Driver code

arr **=** [1000, 11, 445, 1, 330, 3000]

high **=** len(arr) **-** 1

low **=** 0

arr\_max, arr\_min **=** getMinMax(low, high, arr)

print('Minimum element is ', arr\_min)

**print**('nMaximum element is ', arr\_max)

# This code is contributed by DeepakChhitarka

**Output**

Minimum element is 1  
Maximum element is 3000

**Time Complexity:**O(n)

**Auxiliary Space:**O(log n) as the stack space will be filled for the maximum height of the tree formed during recursive calls same as a binary tree.

Total number of comparisons: let the number of comparisons be T(n). T(n) can be written as follows:

**Algorithmic Paradigm:** Divide and Conquer

T(n) = T(floor(n/2)) + T(ceil(n/2)) + 2  
T(2) = 1  
T(1) = 0

If n is a power of 2, then we can write T(n) as:

T(n) = 2T(n/2) + 2

After solving the above recursion, we get

T(n) = 3n/2 -2

Thus, the approach does 3n/2 -2 comparisons if n is a power of 2. And it does more than 3n/2 -2 comparisons if n is not a power of 2.

**Maximum and minimum of an array by comparing in pairs:**

If n is odd then initialize min and max as the first element.

If n is even then initialize min and max as minimum and maximum of the first two elements respectively.

For the rest of the elements, pick them in pairs and compare their

maximum and minimum with max and min respectively.

Below is the implementation of the above approach:

# Python3 program of above implementation

**def** getMinMax(arr):

    n **=** len(arr)

    # If array has even number of elements then

    # initialize the first two elements as minimum

    # and maximum

**if**(n **%** 2 **==** 0):

        mx **=** max(arr[0], arr[1])

        mn **=** min(arr[0], arr[1])

        # set the starting index for loop

        i **=** 2

    # If array has odd number of elements then

    # initialize the first element as minimum

    # and maximum

**else**:

        mx **=** mn **=** arr[0]

        # set the starting index for loop

        i **=** 1

    # In the while loop, pick elements in pair and

    # compare the pair with max and min so far

**while**(i < n **-** 1):

**if** arr[i] < arr[i **+** 1]:

            mx **=** max(mx, arr[i **+** 1])

            mn **=** min(mn, arr[i])

**else**:

            mx **=** max(mx, arr[i])

            mn **=** min(mn, arr[i **+** 1])

        # Increment the index by 2 as two

        # elements are processed in loop

        i **+=** 2

**return** (mx, mn)

# Driver Code

**if** \_\_name\_\_ **==**'\_\_main\_\_':

    arr **=** [1000, 11, 445, 1, 330, 3000]

    mx, mn **=** getMinMax(arr)

    print("Minimum element is", mn)

    print("Maximum element is", mx)

# This code is contributed by Kaustav

**Output**

Minimum element is 1  
Maximum element is 3000

**Time Complexity:** O(n)

**AuxiliarySpace:**O(1) as no extra space was needed.

The total number of comparisons: Different for even and odd n, see below:

If n is odd: 3\*(n-1)/2   
 If n is even: 1 Initial comparison for initializing min and max,   
 and 3(n-2)/2 comparisons for rest of the elements   
 = 1 + 3\*(n-2)/2 = 3n/2 -2

The second and third approaches make an equal number of comparisons when n is a power of 2.

In general, method 3 seems to be the best.

Please write comments if you find any bug in the above programs/algorithms or a better way to solve the same problem.

**Find frequency of each element in a limited range array in less than O(n) time**

Given a sorted array arr[]of positive integers, the task is to find the frequency for each element in the array. Assume all elements in the array are less than some constant **M**

**Note:**Do this without traversing the complete array. i.e. expected time complexity is less than O(n)

**Examples:**

***Input:****arr[] = [1, 1, 1, 2, 3, 3, 5, 5, 8, 8, 8, 9, 9, 10]*

***Output:***

*Element 1 occurs 3 times*

*Element 2 occurs 1 times*

*Element 3 occurs 2 times*

*Element 5 occurs 2 times*

*Element 8 occurs 3 times*

*Element 9 occurs 2 times*

*Element 10 occurs 1 times*

***Input:****arr[] = [2, 2, 6, 6, 7, 7, 7, 11]*

***Output:***

*Element 2 occurs 2 times*

*Element 6 occurs 2 times*

*Element 7 occurs 3 times*

*Element 11 occurs 1 times*

**Frequency of each element in a limited range array using linear search:**

To solve the problem follow the below idea:

*Traverse the input array and increment the frequency of the element if the current element and the previous element are the same, otherwise reset the frequency and print the element and its frequency*

Follow the given steps to solve the problem:

* Initialize frequency to 1 and index to 1.
* Traverse the array from the index position and check if the current element is equal to the previous element.
* If yes, increment the frequency and index and repeat step 2. Otherwise, print the element and its frequency and repeat step 2.
* At last(corner case), print the last element and its frequency.

Below is the implementation of the above approach:

# python3 program to count number of occurrences of

# each element in the array in O(n) time and O(1) space

**def** findFrequencies(ele, n):

    freq **=** 1

    idx **=** 1

    element **=** ele[0]

**while** (idx < n):

        # check if the current element is equal to

        # previous element.

**if** (ele[idx **-** 1] **==** ele[idx]):

            freq **+=** 1

            idx **+=** 1

**else**:

            print(element, " ", freq)

            element **=** ele[idx]

            idx **+=** 1

            # reset the frequency

            freq **=** 1

    # print the last element and its frequency

**print**(element, " ", freq)

# Driver code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

**print**("---frequencies in a sorted array----")

    arr **=** [10, 20, 30, 30, 30, 40, 50, 50, 50, 50, 70]

    n **=** len(arr)

    # Function call

    findFrequencies(arr, n)

# This code is contributed by shivanisinghss2110

**Output**

---frequencies in a sorted array----  
10 1  
20 1  
30 3  
40 1  
50 4  
70 1

**Time Complexity:** O(N)

**Auxiliary Space:** O(1)

**Frequency of each element in a limited range array using Hash-Map:**

To solve the problem follow the below idea:

*The idea is to traverse the input array and for each distinct element of the array, store its frequency in a*[*HashMap*](https://www.geeksforgeeks.org/java-util-hashmap-in-java/)*, and finally print the HashMap.*

Follow the given steps to solve the problem:

* Create a HashMap to map the frequency to the element, i.e to store the element-frequency pair.
* Traverse the array from start to end.
* For each element in the array update the frequency, i.e *hm[array[i]]++*
* Traverse the HashMap and print the element frequency pair

Below is the implementation of the above approach:

# Python program to count number of occurrences of

# each element in the array #include <iostream>

# It prints number of

# occurrences of each element in the array.

**def** findFrequency(arr, n):

    # HashMap to store frequencies

    mp **=** {}

    # traverse the array

**for** i **in** range(n):

        # update the frequency

**if** arr[i] **not in** mp:

            mp[arr[i]] **=** 0

        mp[arr[i]] **+=** 1

    # traverse the hashmap

**for** i **in** mp:

        print("Element", i, "occurs", mp[i], "times")

# Driver function

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    arr **=** [1, 1, 1, 2, 3, 3, 5, 5, 8, 8, 8, 9, 9, 10]

    n **=** len(arr)

    findFrequency(arr, n)

# This code is contributed by shubhamsingh10

**Output**

Element 10 occurs 1 times  
Element 2 occurs 1 times  
Element 9 occurs 2 times  
Element 1 occurs 3 times  
Element 8 occurs 3 times  
Element 3 occurs 2 times  
Element 5 occurs 2 times

**Time Complexity:**O(N), only one traversal of the array is needed.

**Auxiliary Space:**O(N), to store the elements in the HashMap O(N) extra space is needed.

**Frequency of each element in a limited range array using binary search:**

*The problem can be solved in less than O(n) time if all its elements are sorted, i.e. if similar elements exist in the array then the elements are in a contiguous subarray or it can be said that if the ends of a subarray are the same then all the elements inside the subarray are equal. So the count of that element is the size of the subarray and all the elements of that subarray need not be counted.*

Follow the given steps to solve the problem:

* Create a HashMap (*hm*) to store the frequency of elements.
* Create a recursive function that accepts an array and size.
* Check if the first element of the array is equal to the last element. If equal then all the elements are the same and update the frequency by *hm[array[0]+=size*
* Else divide the array into two equal halves and call the function recursively for both halves.
* Traverse the hashmap and print the element frequency pair.

Below is the implementation of the above approach:

# Python 3 program to count number of occurrences of

# each element in the array in less than O(n) time

# A recursive function to count number of occurrences

# for each element in the array without traversing

# the whole array

**def** findFrequencyUtil(arr, low, high, freq):

    # If element at index low is equal to element

    # at index high in the array

**if** (arr[low] **==** arr[high]):

        # increment the frequency of the element

        # by count of elements between high and low

        freq[arr[low]] **+=** high **-** low **+** 1

**else**:

        # Find mid and recurse for left

        # and right subarray

        mid **=** int((low **+** high) **/** 2)

        findFrequencyUtil(arr, low, mid, freq)

        findFrequencyUtil(arr, mid **+** 1, high, freq)

# A wrapper over recursive function

# findFrequencyUtil(). It print number of

# occurrences of each element in the array.

**def** findFrequency(arr, n):

    # create a empty vector to store frequencies

    # and initialize it by 0. Size of vector is

    # maximum value (which is last value in sorted

    # array) plus 1.

    freq **=** [0 **for** i **in** range(n **-** 1 **+** 1)]

    # Fill the vector with frequency

    findFrequencyUtil(arr, 0, n **-** 1, freq)

    # Print the frequencies

**for** i **in** range(0, arr[n **-** 1] **+** 1, 1):

**if** (freq[i] !**=** 0):

**print**("Element", i, "occurs",

                  freq[i], "times")

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    arr **=** [1, 1, 1, 2, 3, 3, 5,

           5, 8, 8, 8, 9, 9, 10]

    n **=** len(arr)

    # Function call

    findFrequency(arr, n)

# This code is contributed by

# Surendra\_Gangwar

**Output**

Element 1 occurs 3 times  
Element 2 occurs 1 times  
Element 3 occurs 2 times  
Element 5 occurs 2 times  
Element 8 occurs 3 times  
Element 9 occurs 2 times  
Element 10 occurs 1 times

**Time Complexity:** O(m log N). Where m is the number of distinct elements in the array of size N. Since m <= M (a constant) (elements are in a limited range), the time complexity of this solution is O(log N)

**Auxiliary Space:**O(N). To store the elements in the HashMap O(n) extra space is needed.

**Frequency of each element in a limited range array using the input array as a Hash-Map:**

In this method, we use the same array as the hash map by modifying its content:

Dry run of this approach:

***Input:****arr = { 1, 1, 1, 2, 3, 3, 5, 5, 8, 8, 8, 9, 9, 10 };*

***Step 1:****Subtract 1 from each element of the array*

*arr  = {0 ,0 ,0 ,1 ,2 ,2 ,4 ,4 ,7 ,7 ,7 ,8 ,8 ,9 }*

***Step 2:****Add n to the index at which the current array element points.*

*for example :-*

*when i=0, arr[arr[0]%n] = 0 adding n to the arr[0], arr[0] =  14;*

*when i=1, arr[arr[1]%n] = 14 adding n to arr[0] ,arr[0] = 28;*

*Similarly finding the modified array in the same way we will get array as*

*arr = {42 ,14 ,28 ,1 ,30, 2, 4, 46, 35, 21, 7, 8, 8, 9}*

***Step 3:****Now in step 2 if you have noticed we added the n value to the index at which a particular element points to. So if we have more than one time have a element that point to the same index then in that case the division of the modified number with the****n****gives us the frequency of the number.*

*for example*

*at i=0; arr[0] =42; arr[0] / n = 3 it means that 0  appeared three times in the modified array as you can see in the arr of step 1.*

*at i=1; arr[1] =14; arr[1]/14 = 1 it means that 1 appeared once in the modified array as you can see in the arr of step 1 .*

*and similarly for other values we can calculate.*

Below is the implementation of the above approach:

# Javascript program to count number of occurrences of

# each element in the array

# It prints number of

# occurrences of each element in the array.

**def** findFrequency(input, n):

**for** i **in** range(n):

        input[i] **-=** 1

**for** i **in** range(n):

        input[input[i] **%** n] **+=** n

**for** i **in** range(n):

**if** input[i] **//** n:

**print**("Element", i **+** 1, "occurs", input[i] **//** n, "times")

        # change element back to original value

        input[i] **=** input[i] **%** n **+** 1

# Driver code

**if** \_\_name\_\_ **==** "\_\_main\_\_":

    arr **=** [1, 1, 1, 2, 3, 3, 5, 5, 8, 8, 8, 9, 9, 10]

    n **=** len(arr)

    # Function call

    findFrequency(arr, n)

# This code is contributed by phasing17

**Output**

Element 1 occurs 3 times  
Element 2 occurs 1 times  
Element 3 occurs 2 times  
Element 5 occurs 2 times  
Element 8 occurs 3 times  
Element 9 occurs 2 times  
Element 10 occurs 1 times

**Time Complexity:** O(N)

**Auxiliary Space:** O(1)

**Tiling Problem using Divide and Conquer algorithm**

Given a n by n board where n is of form 2k where k >= 1 (Basically n is a power of 2 with minimum value as 2). The board has one missing cell (of size 1 x 1). Fill the board using L shaped tiles. A L shaped tile is a 2 x 2 square with one cell of size 1×1 missing.

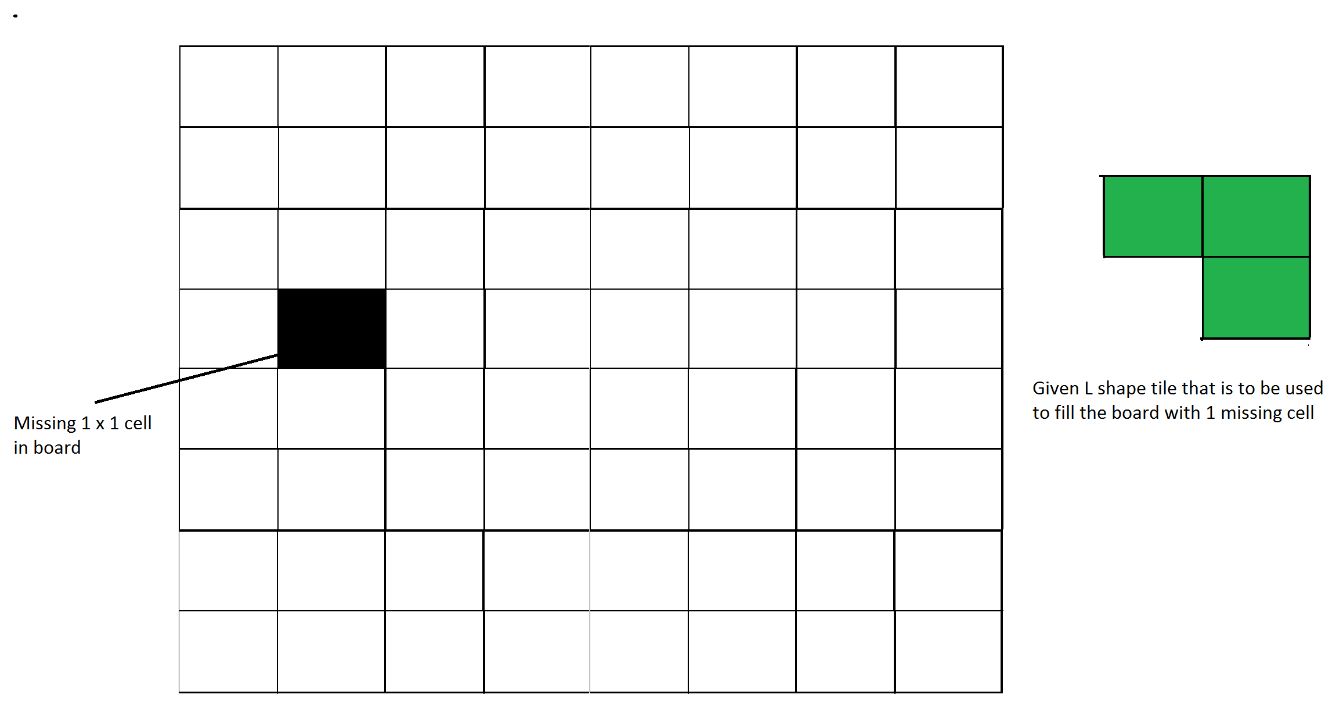


Figure 1: An example input

This problem can be solved using Divide and Conquer. Below is the recursive algorithm.

// n is size of given square, p is location of missing cell  
Tile(int n, Point p)

1) Base case: n = 2, A 2 x 2 square with one cell missing is nothing   
 but a tile and can be filled with a single tile.

2) Place a L shaped tile at the center such that it does not cover  
 the n/2 \* n/2 subsquare that has a missing square. **Now all four**   
 **subsquares of size n/2 x n/2 have a missing cell** (a cell that doesn't  
 need to be filled). See figure 2 below.

3) Solve the problem recursively for following four. Let p1, p2, p3 and  
 p4 be positions of the 4 missing cells in 4 squares.  
 a) Tile(n/2, p1)  
 b) Tile(n/2, p2)  
 c) Tile(n/2, p3)  
 d) Tile(n/2, p3)

The below diagrams show working of above algorithm

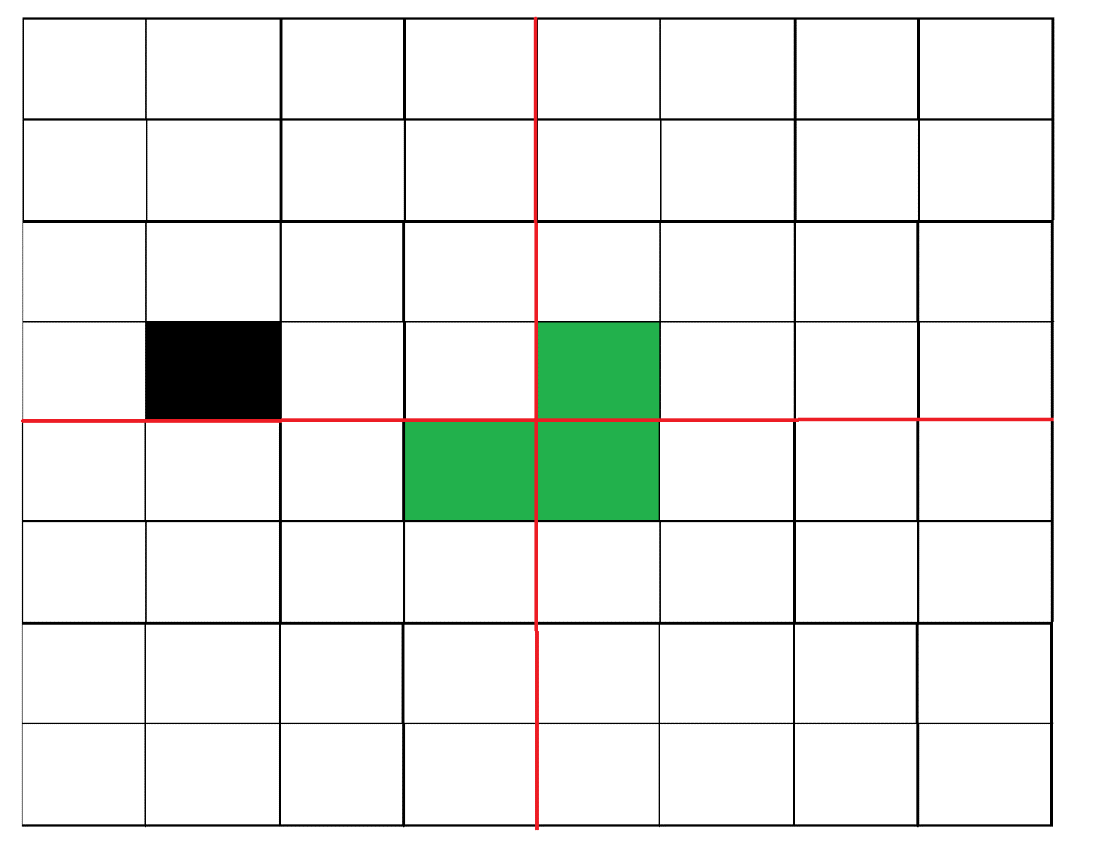


Figure 2: After placing the first tile

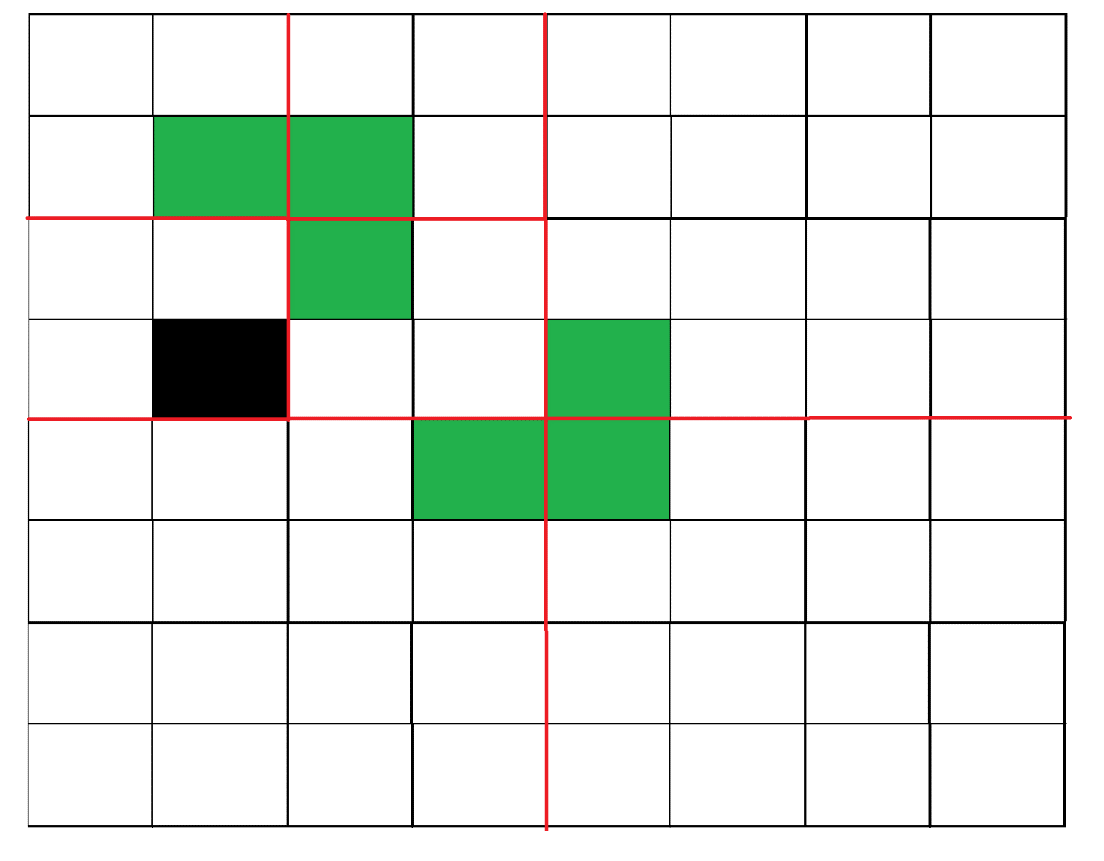


Figure 3: Recurring for the first subsquare.

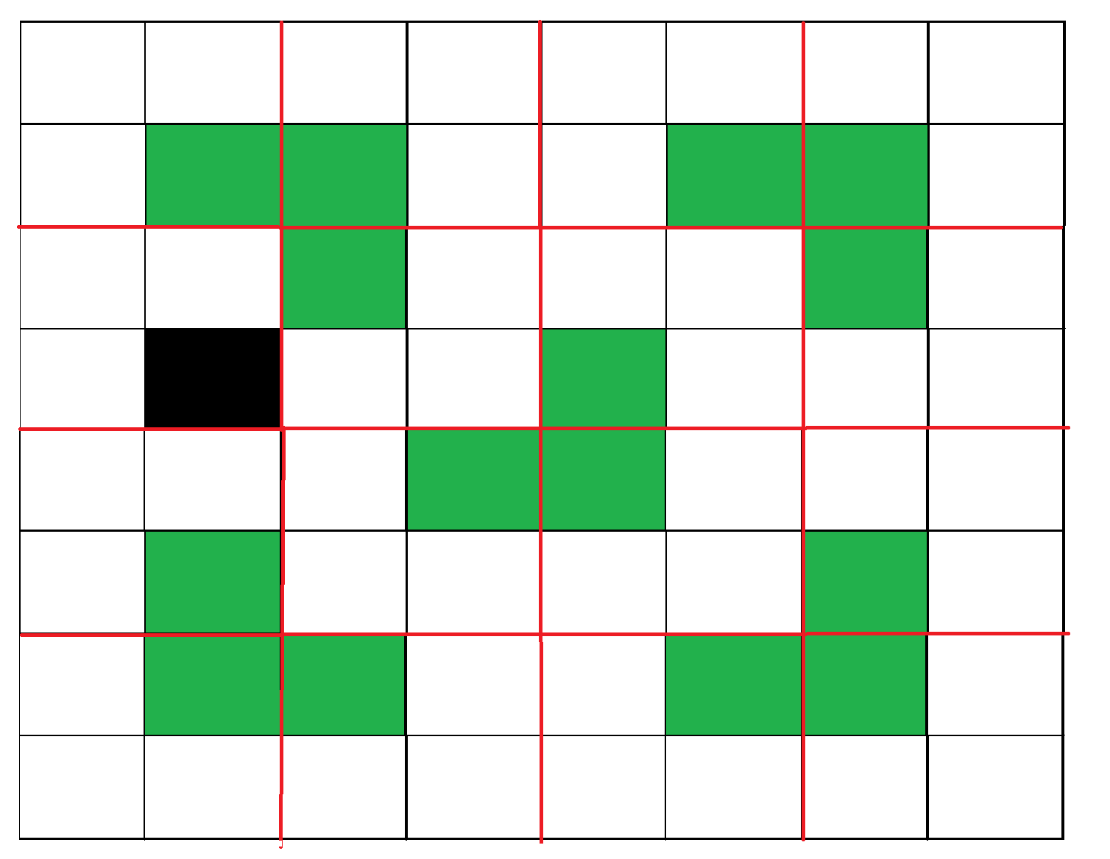


Figure 4: Shows the first step in all four subsquares.

**Examples:**

**Input :** size = 2 and mark coordinates = (0, 0)  
**Output :**   
-1 1  
1 1  
Coordinate (0, 0) is marked. So, no tile is there. In the remaining three positions,   
a tile is placed with its number as 1.  
**Input :** size = 4 and mark coordinates = (0, 0)  
**Output :**  
-1 3 2 2  
3 3 1 2  
4 1 1 5  
4 4 5 5

Below is the implementation of above idea:

# Python3 program to place tiles

size\_of\_grid **=** 0

b **=** 0

a **=** 0

cnt **=** 0

arr **=** [[0 **for** i **in** range(128)] **for** j **in** range(128)]

**def** place(x1, y1, x2, y2, x3, y3):

**global** cnt

    cnt **+=** 1

    arr[x1][y1] **=** cnt;

    arr[x2][y2] **=** cnt;

    arr[x3][y3] **=** cnt;

**def** tile(n, x, y):

**global** cnt

    r **=** 0

    c **=** 0

**if** (n **==** 2):

        cnt **+=** 1

**for** i **in** range(n):

**for** j **in** range(n):

**if**(arr[x **+** i][y **+** j] **==** 0):

                    arr[x **+** i][y **+** j] **=** cnt

**return** 0;

**for** i **in** range(x, x **+** n):

**for** j **in** range(y, y **+** n):

**if** (arr[i][j] !**=** 0):

                r **=** i

                c **=** j

**if** (r < x **+** n **/** 2 **and** c < y **+** n **/** 2):

        place(x **+** int(n **/** 2), y **+** int(n **/** 2) **-** 1, x **+** int(n **/** 2), y **+** int(n **/** 2), x **+** int(n **/** 2) **-** 1, y **+** int(n **/** 2))

**elif**(r >**=** x **+** int(n **/** 2) **and** c < y **+** int(n **/** 2)):

        place(x **+** int(n **/** 2) **-** 1, y **+** int(n **/** 2), x **+** int(n **/** 2), y **+** int(n **/** 2), x **+** int(n **/** 2) **-** 1, y **+** int(n **/** 2) **-** 1)

**elif**(r < x **+** int(n **/** 2) **and** c >**=** y **+** int(n **/** 2)):

        place(x **+** int(n **/** 2), y **+** int(n **/** 2) **-** 1, x **+** int(n **/** 2), y **+** int(n **/** 2), x **+** int(n **/** 2) **-** 1, y **+** int(n **/** 2) **-** 1)

**elif**(r >**=** x **+** int(n **/** 2) **and** c >**=** y **+** int(n **/** 2)):

        place(x **+** int(n **/** 2) **-** 1, y **+** int(n **/** 2), x **+** int(n **/** 2), y **+** int(n **/** 2) **-** 1, x **+** int(n **/** 2) **-** 1, y **+** int(n **/** 2) **-** 1)

    tile(int(n **/** 2), x, y **+** int(n **/** 2));

    tile(int(n **/** 2), x, y);

    tile(int(n **/** 2), x **+** int(n **/** 2), y);

    tile(int(n **/** 2), x **+** int(n **/** 2), y **+** int(n **/** 2));

**return** 0

size\_of\_grid **=** 8

a **=** 0

b **=** 0

arr[a][b] **= -**1

tile(size\_of\_grid, 0, 0)

**for** i **in** range(size\_of\_grid):

**for** j **in** range(size\_of\_grid):

**print**(arr[i][j], end**=**" ")

    print()

# This code is contributed by rag2127

**Output**

-1 9 8 8 4 4 3 3   
9 9 7 8 4 2 2 3   
10 7 7 11 5 5 2 6   
10 10 11 11 1 5 6 6   
14 14 13 1 1 19 18 18   
14 12 13 13 19 19 17 18   
15 12 12 16 20 17 17 21   
15 15 16 16 20 20 21 21

**Time Complexity:**

Recurrence relation for above recursive algorithm can be written as below. C is a constant.

T(n) = 4T(n/2) + C

The above recursion can be solved using [Master Method](https://www.geeksforgeeks.org/analysis-algorithm-set-4-master-method-solving-recurrences/)and time complexity is O(n2)

**Space Complexity:** O(n)

**How does this work?**

The working of Divide and Conquer algorithm can be proved using Mathematical Induction. Let the input square be of size 2k x 2k where k >=1.

Base Case: We know that the problem can be solved for k = 1. We have a 2 x 2 square with one cell missing.

Induction Hypothesis: Let the problem can be solved for k-1.

Now we need to prove  that the problem can be solved for k if it can be solved for k-1. For k, we put a L shaped tile in middle and we have four subsquares with dimension 2k-1 x 2k-1 as shown in figure 2 above. So if we can solve 4 subsquares, we can solve the complete square.

**References:**

<http://www.comp.nus.edu.sg/~sanjay/cs3230/dandc.pdf>

**Inversion count in Array using Merge Sort**

**Inversion Count**for an array indicates – how far (or close) the array is from being sorted. If the array is already sorted, then the inversion count is 0, but if the array is sorted in reverse order, the inversion count is the maximum.

Given an array **a[]**. The task is to find the inversion count of **a[]**. Where two elements a[i] and a[j] form an inversion if a[i] > a[j] and i < j.

**Examples:**

***Input:****arr[] = {8, 4, 2, 1}*

***Output:****6*

***Explanation:****Given array has six inversions: (8, 4), (4, 2), (8, 2), (8, 1), (4, 1), (2, 1).*

***Input:****arr[] = {1, 20, 6, 4, 5}*

***Output:****5*

***Explanation:****Given array has five inversions: (20, 6), (20, 4), (20, 5), (6, 4), (6, 5).*

Recommended Problem

Count Inversions

**Naive Approach:**

*Traverse through the array, and for every index, find the number of smaller elements on its right side of the array. This can be done using a nested loop. Sum up the counts for all indices in the array and print the sum.*

Follow the below steps to Implement the idea:

* Traverse through the array from start to end
* For every element, find the count of elements smaller than the current number up to that index using another loop.
* Sum up the count of inversion for every index.
* Print the count of inversions.

Below is the Implementation of the above approach:

# Python3 program to count

# inversions in an array

**def** getInvCount(arr, n):

    inv\_count **=** 0

**for** i **in** range(n):

**for** j **in** range(i **+** 1, n):

**if** (arr[i] > arr[j]):

                inv\_count **+=** 1

**return** inv\_count

# Driver Code

arr **=** [1, 20, 6, 4, 5]

n **=** len(arr)

print("Number of inversions are",

      getInvCount(arr, n))

# This code is contributed by Smitha Dinesh Semwal

**Output**

Number of inversions are 5

**Time Complexity:** O(N2), Two nested loops are needed to traverse the array from start to end.

**Auxiliary Space:**O(1), No extra space is required.

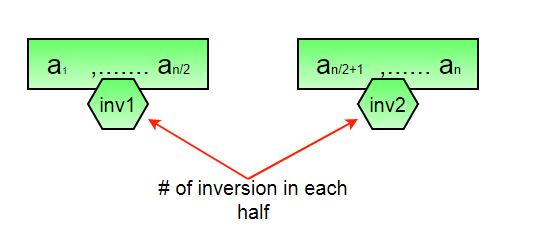
**Count Inversions in an array using** [Merge Sort](https://www.geeksforgeeks.org/merge-sort/)**:**

Below is the idea to solve the problem:

*Use*[***Merge sort***](https://www.geeksforgeeks.org/merge-sort/)*with modification that every time an unsorted pair is found increment****count****by one and return count at the end.*

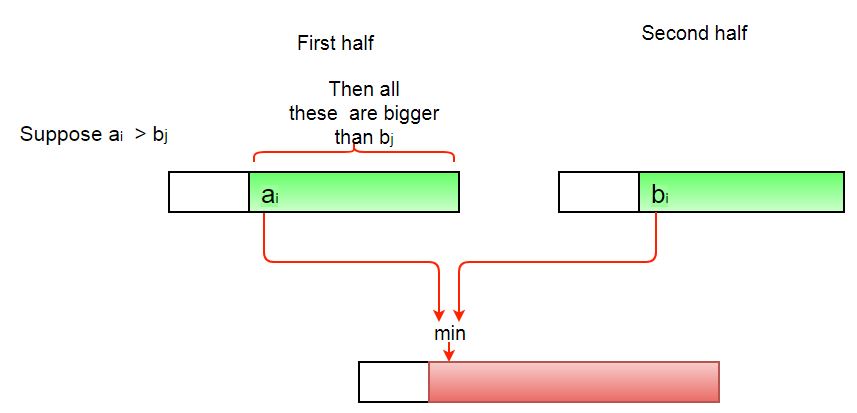
**Illustration:**

*Suppose the number of inversions in the left half and right half of the array (let be inv1 and inv2); what kinds of inversions are not accounted for in Inv1 + Inv2? The answer is – the inversions that need to be counted during the merge step. Therefore, to get the total number of inversions that needs to be added are the number of inversions in the left subarray, right subarray, and merge().*

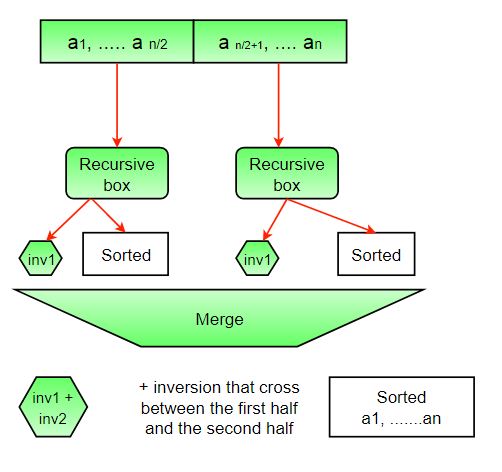


***How to get****the****number of inversions in merge()?***

*In merge process, let i is used for indexing left sub-array and j for right sub-array. At any step in merge(), if a[i] is greater than a[j], then there are (mid – i) inversions. because left and right subarrays are sorted, so all the remaining elements in left-subarray (a[i+1], a[i+2] … a[mid]) will be greater than a[j]*



***The complete picture:***



Follow the below steps to Implement the idea:

* The idea is similar to merge sort, divide the array into two equal or almost equal halves in each step until the base case is reached.
* Create a function merge that counts the number of inversions when two halves of the array are merged,
* Create two indices i and j, i is the index for the first half, and j is an index of the second half.
* If a[i] is greater than a[j], then there are (mid – i) inversions because left and right subarrays are sorted, so all the remaining elements in left-subarray (a[i+1], a[i+2] … a[mid]) will be greater than a[j].
* Create a recursive function to divide the array into halves and find the answer by summing the number of inversions in the first half, the number of inversions in the second half and the number of inversions by merging the two.
* The base case of recursion is when there is only one element in the given half.
* Print the answer.

Below is the Implementation of the above approach:

# Python 3 program to count inversions in an array

# Function to Use Inversion Count

**def** mergeSort(arr, n):

    # A temp\_arr is created to store

    # sorted array in merge function

    temp\_arr **=** [0]**\***n

**return** \_mergeSort(arr, temp\_arr, 0, n**-**1)

# This Function will use MergeSort to count inversions

**def** \_mergeSort(arr, temp\_arr, left, right):

    # A variable inv\_count is used to store

    # inversion counts in each recursive call

    inv\_count **=** 0

    # We will make a recursive call if and only if

    # we have more than one elements

**if** left < right:

        # mid is calculated to divide the array into two subarrays

        # Floor division is must in case of python

        mid **=** (left **+** right)**//**2

        # It will calculate inversion

        # counts in the left subarray

        inv\_count **+=** \_mergeSort(arr, temp\_arr,

                                left, mid)

        # It will calculate inversion

        # counts in right subarray

        inv\_count **+=** \_mergeSort(arr, temp\_arr,

                                mid **+** 1, right)

        # It will merge two subarrays in

        # a sorted subarray

        inv\_count **+=** merge(arr, temp\_arr, left, mid, right)

**return** inv\_count

# This function will merge two subarrays

# in a single sorted subarray

**def** merge(arr, temp\_arr, left, mid, right):

    i **=** left     # Starting index of left subarray

    j **=** mid **+** 1  # Starting index of right subarray

    k **=** left     # Starting index of to be sorted subarray

    inv\_count **=** 0

    # Conditions are checked to make sure that

    # i and j don't exceed their

    # subarray limits.

**while** i <**=** mid **and** j <**=** right:

        # There will be no inversion if arr[i] <= arr[j]

**if** arr[i] <**=** arr[j]:

            temp\_arr[k] **=** arr[i]

            k **+=** 1

            i **+=** 1

**else**:

            # Inversion will occur.

            temp\_arr[k] **=** arr[j]

            inv\_count **+=** (mid**-**i **+** 1)

            k **+=** 1

            j **+=** 1

    # Copy the remaining elements of left

    # subarray into temporary array

**while** i <**=** mid:

        temp\_arr[k] **=** arr[i]

        k **+=** 1

        i **+=** 1

    # Copy the remaining elements of right

    # subarray into temporary array

**while** j <**=** right:

        temp\_arr[k] **=** arr[j]

        k **+=** 1

        j **+=** 1

    # Copy the sorted subarray into Original array

**for** loop\_var **in** range(left, right **+** 1):

        arr[loop\_var] **=** temp\_arr[loop\_var]

**return** inv\_count

# Driver Code

# Given array is

arr **=** [1, 20, 6, 4, 5]

n **=** len(arr)

result **=** mergeSort(arr, n)

print("Number of inversions are", result)

# This code is contributed by ankush\_953

**Output**

Number of inversions are 5

**Time Complexity:** O(n \* log n), The algorithm used is divide and conquer i.e. merge sort whose complexity is O(n log n).

**Auxiliary Space:** O(n), Temporary array.

**Note:**The above code modifies (or sorts) the input array. If we want to count only inversions, we need to create a copy of the original array and call mergeSort() on the copy to preserve the original array’s order.

**Count Inversions in an array using**[Heapsort](https://www.geeksforgeeks.org/heap-sort/)**and**[Bisection](https://www.geeksforgeeks.org/program-for-bisection-method/)**:**

Follow the below steps to Implement the idea:

* Create a heap with new pair elements,  (element, index).
* After sorting them, pop out each minimum sequentially and create a new sorted list with the indexes.
* Calculate the difference between the original index and the index of bisection of the new sorted list.
* Sum up the difference.

Below is the idea to Implement the above approach:

**from** heapq **import** heappush, heappop

**from** bisect **import** bisect, insort

**def** getNumOfInversions(A):

    N **=** len(A)

**if** N <**=** 1:

**return** 0

    sortList **=** []

    result **=** 0

    # Heapsort, O(N\*log(N))

**for** i, v **in** enumerate(A):

        heappush(sortList, (v, i))

    # Create a sorted list of indexes

    x **=** []

**while** sortList:

        # O(log(N))

        v, i **=** heappop(sortList)

        # Find the current minimum's index

        # the index y can represent how many minimums on the left

        y **=** bisect(x, i)

        # i can represent how many elements on the left

        # i - y can find how many bigger nums on the left

        result **+=** i **-** y

        insort(x, i)

**return** result

# Driver Code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    A **=** [1, 20, 6, 4, 5]

    result **=** getNumOfInversions(A)

**print**(f'Number of inversions are {result}')

**Output**

Number of inversions are 5

**Time Complexity:** O(n \* log n). Both heapsort and bisection can perform sorted insertion in (log n) in each element.

**Auxiliary Space:** O(n). A heap and a new list are the same length as the original array.

**The Skyline Problem using Divide and Conquer algorithm**

Given n rectangular buildings in a 2-dimensional city, computes the skyline of these buildings, eliminating hidden lines. The main task is to view buildings from a side and remove all sections that are not visible.  All buildings share common bottom and every **building**is represented by triplet (left, ht, right)

* ‘left’: is x coordinated of left side (or wall).
* ‘right’: is x coordinate of right side
* ‘ht’: is height of building.

A **skyline**is a collection of rectangular strips. A rectangular **strip**is represented as a pair (left, ht) where left is x coordinate of left side of strip and ht is height of strip. **Examples:**

Input: Array of buildings  
 { (1, 11, 5), (2, 6, 7), (3, 13, 9), (12, 7, 16), (14, 3, 25),  
 (19, 18, 22), (23, 13, 29), (24, 4, 28) }  
Output: Skyline (an array of rectangular strips)  
 A strip has x coordinate of left side and height   
 (1, 11), (3, 13), (9, 0), (12, 7), (16, 3), (19, 18),   
 (22, 3), (25, 0)  
Below image is for input 1 :

Consider following as another example when there is only one  
building  
Input: {(1, 11, 5)}  
Output: (1, 11), (5, 0)

A **Simple Solution** is to initialize skyline or result as empty, then one by one add buildings to skyline. A building is added by first finding the overlapping strip(s). If there are no overlapping strips, the new building adds new strip(s). If overlapping strip is found, then height of the existing strip may increase. Time complexity of this solution is O(n2) We can find Skyline in Θ(nLogn) time using [**Divide and Conquer**](https://www.geeksforgeeks.org/divide-and-conquer-set-1-find-closest-pair-of-points/). The idea is similar to [Merge Sort](http://geeksquiz.com/merge-sort/), divide the given set of buildings in two subsets. Recursively construct skyline for two halves and finally merge the two skylines. **How to Merge two Skylines?**The idea is similar to merge of merge sort, start from first strips of two skylines, compare x coordinates. Pick the strip with smaller x coordinate and add it to result. The height of added strip is considered as maximum of current heights from skyline1 and skyline2. **Example to show working of merge:**

Height of new Strip is always obtained by takin maximum of following  
 (a) Current height from skyline1, say 'h1'.   
 (b) Current height from skyline2, say 'h2'  
 h1 and h2 are initialized as 0. h1 is updated when a strip from  
 SkyLine1 is added to result and h2 is updated when a strip from   
 SkyLine2 is added.  
   
 Skyline1 = {(1, 11), (3, 13), (9, 0), (12, 7), (16, 0)}  
 Skyline2 = {(14, 3), (19, 18), (22, 3), (23, 13), (29, 0)}  
 Result = {}  
 h1 = 0, h2 = 0  
   
 Compare (1, 11) and (14, 3). Since first strip has smaller left x,  
 add it to result and increment index for Skyline1.   
 h1 = 11, New Height = max(11, 0)   
 Result = {(1, 11)}

Compare (3, 13) and (14, 3). Since first strip has smaller left x,  
 add it to result and increment index for Skyline1  
 h1 = 13, New Height = max(13, 0)  
 Result = {(1, 11), (3, 13)}   
   
 Similarly (9, 0) and (12, 7) are added.  
 h1 = 7, New Height = max(7, 0) = 7  
 Result = {(1, 11), (3, 13), (9, 0), (12, 7)}

Compare (16, 0) and (14, 3). Since second strip has smaller left x,   
 it is added to result.  
 h2 = 3, New Height = max(7, 3) = 7  
 Result = {(1, 11), (3, 13), (9, 0), (12, 7), (14, 7)}

Compare (16, 0) and (19, 18). Since first strip has smaller left x,   
 it is added to result.  
 h1 = 0, New Height = max(0, 3) = 3  
 Result = {(1, 11), (3, 13), (9, 0), (12, 7), (14, 7), (16, 3)}

Since Skyline1 has no more items, all remaining items of Skyline2   
are added   
 Result = {(1, 11), (3, 13), (9, 0), (12, 7), (14, 7), (16, 3),   
 (19, 18), (22, 3), (23, 13), (29, 0)}

One observation about above output is, the strip (14, 7) is redundant  
(There is already an strip of same height). We remove all redundant   
strips.   
 Result = {(1, 11), (3, 13), (9, 0), (12, 7), (16, 3), (19, 18),   
 (22, 3), (23, 13), (29, 0)}

In below code, redundancy is handled by not appending a strip if the   
previous strip in result has same height.

Below is C++ implementation of above idea.

**class** Building:

**def** \_\_init\_\_(self, left, ht, right):

        self.left **=** left

        self.ht **=** ht

        self.right **=** right

**class** Strip:

**def** \_\_init\_\_(self, left**=**0, ht**=**0):

        self.left **=** left

        self.ht **=** ht

**class** SkyLine:

**def** \_\_init\_\_(self, cap):

        self.arr **=** []

        self.capacity **=** cap

        self.n **=** 0

**def** count(self):

**return** self.n

**def** merge(self, other):

        res **=** SkyLine(self.n **+** other.n)

        h1, h2, i, j **=** 0, 0, 0, 0

**while** i < self.n **and** j < other.n:

**if** self.arr[i].left < other.arr[j].left:

                x1, h1 **=** self.arr[i].left, self.arr[i].ht

                maxh **=** max(h1, h2)

                res.append(Strip(x1, maxh))

                i **+=** 1

**else**:

                x2, h2 **=** other.arr[j].left, other.arr[j].ht

                maxh **=** max(h1, h2)

                res.append(Strip(x2, maxh))

                j **+=** 1

**while** i < self.n:

            res.append(self.arr[i])

            i **+=** 1

**while** j < other.n:

            res.append(other.arr[j])

            j **+=** 1

**return** res

**def** append(self, st):

**if** self.n > 0 **and** self.arr[self.n**-**1].ht **==** st.ht:

**return**

**if** self.n > 0 **and** self.arr[self.n**-**1].left **==** st.left:

            self.arr[self.n**-**1].ht **=** max(self.arr[self.n**-**1].ht, st.ht)

**return**

        self.arr.append(st)

        self.n **+=** 1

**def** print\_skyline(self):

**print**("Skyline for given buildings is")

**for** i **in** range(self.n):

            print(" ({}, {}),".format(self.arr[i].left, self.arr[i].ht), end**=**"")

        print()

**def** find\_skyline(arr, l, h):

**if** l **==** h:

        res **=** SkyLine(2)

        res.append(Strip(arr[l].left, arr[l].ht))

        res.append(Strip(arr[l].right, 0))

**return** res

    mid **=** (l **+** h) **//** 2

    sl **=** find\_skyline(arr, l, mid)

    sr **=** find\_skyline(arr, mid**+**1, h)

    res **=** sl.merge(sr)

**return** res

arr **=** [Building(1, 11, 5), Building(2, 6, 7), Building(3, 13, 9), Building(12, 7, 16), Building(14, 3, 25), Building(19, 18, 22), Building(23, 13, 29), Building(24, 4, 28)]

n **=** len(arr)

skyline **=** find\_skyline(arr, 0, n**-**1)

skyline.print\_skyline()

**Output:**

Skyline for given buildings is  
 (1, 11), (3, 13), (9, 0), (12, 7), (16, 3), (19, 18),   
 (22, 3), (23, 13), (29, 0),

Time complexity of above recursive implementation is same as Merge Sort. T(n) = T(n/2) + Θ(n) Solution of above recurrence is Θ(nLogn) **References:**

* <http://faculty.kfupm.edu.sa/ics/darwish/stuff/ics353handouts/Ch4Ch5.pdf>
* [www.cs.ucf.edu/~sarahb/COP3503/Lectures/DivideAndConquer.ppt](http://www.cs.ucf.edu/~sarahb/COP3503/Lectures/DivideAndConquer.ppt)

**Search in a Row-wise and Column-wise Sorted 2D Array using Divide and Conquer algorithm**

Given an n x n matrix, where every row and column is sorted in increasing order. Given a key, how to decide whether this key is in the matrix.

[A linear time complexity is discussed in the previous post.](https://www.geeksforgeeks.org/search-in-row-wise-and-column-wise-sorted-matrix/) This problem can also be a very good example for [divide and conquer algorithms](https://www.geeksforgeeks.org/tag/divide-and-conquer/). Following is divide and conquer algorithm.

1) Find the middle element.

2) If middle element is same as key return.

3) If middle element is lesser than key then

….3a) search submatrix on lower side of middle element

….3b) Search submatrix on right hand side.of middle element

4) If middle element is greater than key then

….4a) search vertical submatrix on left side of middle element

….4b) search submatrix on right hand side.



Following implementation of above algorithm.

# Python3 program for implementation of

# divide and conquer algorithm to find

# a given key in a row-wise and column-wise

# sorted 2D array a divide and conquer method

# to search a given key in mat in rows from

# fromRow to toRow and columns from fromCol to

# toCol

**def** search(mat, fromRow, toRow, fromCol, toCol, key):

    # Find middle and compare with middle

    i **=** fromRow **+** (toRow **-** fromRow) **//** 2;

    j **=** fromCol **+** (toCol **-** fromCol) **//** 2;

**if** (mat[i][j] **==** key): # If key is present at middle

        print("Found " , key , " at " , i , " " , j);

**else**:

        # right-up quarter of matrix is searched in all cases.

        # Provided it is different from current call

**if** (i !**=** toRow **or** j !**=** fromCol):

            search(mat, fromRow, i, j, toCol, key);

        # Special case for iteration with 1\*2 matrix

        # mat[i][j] and mat[i][j+1] are only two elements.

        # So just check second element

**if** (fromRow **==** toRow **and** fromCol **+** 1 **==** toCol):

**if** (mat[fromRow][toCol] **==** key):

                print("Found " , key , " at " , fromRow , " " , toCol);

        # If middle key is lesser then search lower horizontal

        # matrix and right hand side matrix

**if** (mat[i][j] < key):

            # search lower horizontal if such matrix exists

**if** (i **+** 1 <**=** toRow):

                search(mat, i **+** 1, toRow, fromCol, toCol, key);

        # If middle key is greater then search left vertical

        # matrix and right hand side matrix

**else**:

            # search left vertical if such matrix exists

**if** (j **-** 1 >**=** fromCol):

                search(mat, fromRow, toRow, fromCol, j **-** 1, key);

# Driver code

**if** \_\_name\_\_ **==** '\_\_main\_\_':

    mat **=** [[ 10, 20, 30, 40],

           [15, 25, 35, 45],

           [27, 29, 37, 48],

           [32, 33, 39, 50]];

    rowcount **=** 4; colCount **=** 4; key **=** 50;

**for** i **in** range(rowcount):

**for** j **in** range(colCount):

            search(mat, 0, rowcount **-** 1, 0, colCount **-** 1, mat[i][j]);

# This code is contributed by 29AjayKumar

**Output:**

Found 10 at 0 0  
Found 20 at 0 1  
Found 30 at 0 2  
Found 40 at 0 3  
Found 15 at 1 0  
Found 25 at 1 1  
Found 35 at 1 2  
Found 45 at 1 3  
Found 27 at 2 0  
Found 29 at 2 1  
Found 37 at 2 2  
Found 48 at 2 3  
Found 32 at 3 0  
Found 33 at 3 1  
Found 39 at 3 2  
Found 50 at 3 3

**Time complexity:**

We are given a n\*n matrix, the algorithm can be seen as recurring for 3 matrices of size n/2 x n/2. Following is recurrence for time complexity

T(n) = 3T(n/2) + O(1)

**Space Complexity:** O(log(n))

The solution of recurrence is O(n1.58) using [Master Method](https://www.geeksforgeeks.org/analysis-algorithm-set-4-master-method-solving-recurrences/).

But the actual implementation calls for one submatrix of size n x n/2 or n/2 x n, and other submatrix of size n/2 x n/2.

This article is contributed by **Kaushik Lele**. Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above

**Allocate minimum number of 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.*

Recommended Problem

Allocate minimum number of pages

**Approach:** A [Binary Search](https://www.geeksforgeeks.org/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, maximum(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 **maximum(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:

# Python3 program for optimal allocation of pages

# Utility function to check if

# current minimum value is feasible or not.

**def** isPossible(arr, n, m, curr\_min):

    studentsRequired **=** 1

    curr\_sum **=** 0

    # iterate over all books

**for** i **in** range(n):

        # check if current number of pages are

        # greater than curr\_min that means

        # we will get the result after

        # mid no. of pages

**if** (arr[i] > curr\_min):

**return** False

        # count how many students are required

        # to distribute curr\_min pages

**if** (curr\_sum **+** arr[i] > curr\_min):

            # increment student count

            studentsRequired **+=** 1

            # update curr\_sum

            curr\_sum **=** arr[i]

            # if students required becomes greater

            # than given no. of students, return False

**if** (studentsRequired > m):

**return** False

        # else update curr\_sum

**else**:

            curr\_sum **+=** arr[i]

**return** True

# function to find minimum pages

**def** findPages(arr, n, m):

    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** i **in** range(n):

        sum **+=** arr[i]

    # initialize start as 0 pages and

    # end as total pages

    start, end **=** 0, sum

    result **=** 10**\*\***9

    # traverse until start <= end

**while** (start <**=** end):

        # check if it is possible to distribute

        # books by using mid as current minimum

        mid **=** (start **+** end) **//** 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 and books

            # are sorted so reduce end = mid -1

            # that means

            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 Code

# Number of pages in books

arr **=** [12, 34, 67, 90]

n **=** len(arr)

m **=** 2   # No. of students

**print**("Minimum number of pages = ",

      findPages(arr, n, m))

# This code is contributed by Mohit Kumar

**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)

**Modular Exponentiation (Power in Modular Arithmetic)**

Given three numbers x, y and p, compute (xy) % p.

**Examples :**

**Input:** x = 2, y = 3, p = 5  
**Output:** 3  
**Explanation:** 2^3 % 5 = 8 % 5 = 3.

**Input:** x = 2, y = 5, p = 13  
**Output:** 6  
**Explanation:** 2^5 % 13 = 32 % 13 = 6.

Recommended Problem

Modular Exponentiation for large numbers

We have discussed [recursive](https://www.geeksforgeeks.org/write-a-c-program-to-calculate-powxn/) and [iterative](https://www.geeksforgeeks.org/write-an-iterative-olog-y-function-for-powx-y/) solutions for power.

Below is discussed iterative solution.

* C++14
* Java
* Python3
* Javascript
* C#

# Iterative Function to calculate (x^y)%p in O(log y)

**def** power(x, y, p):

    # Initialize result

    res **=** 1

**while** (y > 0):

        # If y is odd, multiply x with result

**if** ((y & 1) !**=** 0):

            res **=** res **\*** x

        # y must be even now

        y **=** y >> 1  # y = y/2

        x **=** x **\*** x  # Change x to x^2

**return** res **%** p

  # Driver Code

x **=** 2

y **=** 5

p **=** 13

print("Power is ", power(x, y, p))

# This code is contributed by Khushboogoyal499

**Output**

Power is 6

***Time Complexity:*** O(log2y), where y represents the value of the given input.

***Auxiliary Space:***O(1), no extra space is required, so it is a constant.

**Efficient Approach:**

The problem with the above solutions is, overflow may occur for large values of n or x. Therefore, power is generally evaluated under the modulo of a large number.

Below is the fundamental modular property that is used for efficiently computing power under modular arithmetic.

(ab) mod p = ( (a mod p) (b mod p) ) mod p

For example a = 50, b = 100, p = 13  
50 mod 13 = 11  
100 mod 13 = 9

(50 \* 100) mod 13 = ( (50 mod 13) \* (100 mod 13) ) mod 13   
or (5000) mod 13 = ( 11 \* 9 ) mod 13  
or 8 = 8

Below is the implementation based on the above property.

# Iterative Python3 program

# to compute modular power

# Iterative Function to calculate

# (x^y)%p in O(log y)

**def** power(x, y, p) :

    res **=** 1     # Initialize result

    # Update x if it is more

    # than or equal to p

    x **=** x **%** p

**if** (x **==** 0) :

**return** 0

**while** (y > 0) :

        # If y is odd, multiply

        # x with result

**if** ((y & 1) **==** 1) :

            res **=** (res **\*** x) **%** p

        # y must be even now

        y **=** y >> 1      # y = y/2

        x **=** (x **\*** x) **%** p

**return** res

# Driver Code

x **=** 2; y **=** 5; p **=** 13

print("Power is ", power(x, y, p))

# This code is contributed by Nikita Tiwari.

**Output**

Power is 6

**Time** **Complexity:** O(Log y), where y represents the value of the given input.

**Auxiliary Space:**O(1), as we are not using any extra space.

**Medium Questions:**

**Hard Questions:**