Equilibrium index of an array

Equilibrium index of an array is an index such that the sum of elements at lower indexes is equal to the sum of elements at higher indexes. For example, in an array A:

**Example :**

***Input****: A[] = {-7, 1, 5, 2, -4, 3, 0}****Output****: 3  
3 is an equilibrium index, because:  
A[0] + A[1] + A[2] = A[4] + A[5] + A[6]*

***Input****: A[] = {1, 2, 3}****Output****: -1*

Write a function *int equilibrium(int[] arr, int n)*; that given a sequence arr[] of size n, returns an equilibrium index (if any) or -1 if no equilibrium indexes exist.

[**Recommended: Please solve it on “*PRACTICE* ” first, before moving on to the solution.**](https://practice.geeksforgeeks.org/problems/equilibrium-point/0)

**Method 1 (Simple but inefficient)**  
Use two loops. Outer loop iterates through all the element and inner loop finds out whether the current index picked by the outer loop is equilibrium index or not. Time complexity of this solution is O(n^2).

* C++
* C
* Java
* Python3
* C#
* PHP

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // C program to find equilibrium  // index of an array    #include <stdio.h>    int equilibrium(int arr[], int n)  {      int i, j;      int leftsum, rightsum;        /\* Check for indexes one by one until        an equilibrium index is found \*/      for (i = 0; i < n; ++i) {            /\* get left sum \*/          leftsum = 0;          for (j = 0; j < i; j++)              leftsum += arr[j];            /\* get right sum \*/          rightsum = 0;          for (j = i + 1; j < n; j++)              rightsum += arr[j];            /\* if leftsum and rightsum are same,             then we are done \*/          if (leftsum == rightsum)              return i;      }        /\* return -1 if no equilibrium index is found \*/      return -1;  }    // Driver code  int main()  {      int arr[] = { -7, 1, 5, 2, -4, 3, 0 };      int arr\_size = sizeof(arr) / sizeof(arr[0]);      printf("%d", equilibrium(arr, arr\_size));        getchar();      return 0;  } |

**Time Complexity:** O(n^2)

**Method 2 (Tricky and Efficient)**  
The idea is to get the total sum of the array first. Then Iterate through the array and keep updating the left sum which is initialized as zero. In the loop, we can get the right sum by subtracting the elements one by one. Thanks to Sambasiva for suggesting this solution and providing code for this.

1) Initialize leftsum as 0

2) Get the total sum of the array as *sum*

3) Iterate through the array and for each index i, do following.

a) Update *sum* to get the right sum.

*sum* = *sum* - arr[i]

// *sum* is now right sum

b) If leftsum is equal to *sum*, then return current index.

// update leftsum for next iteration.

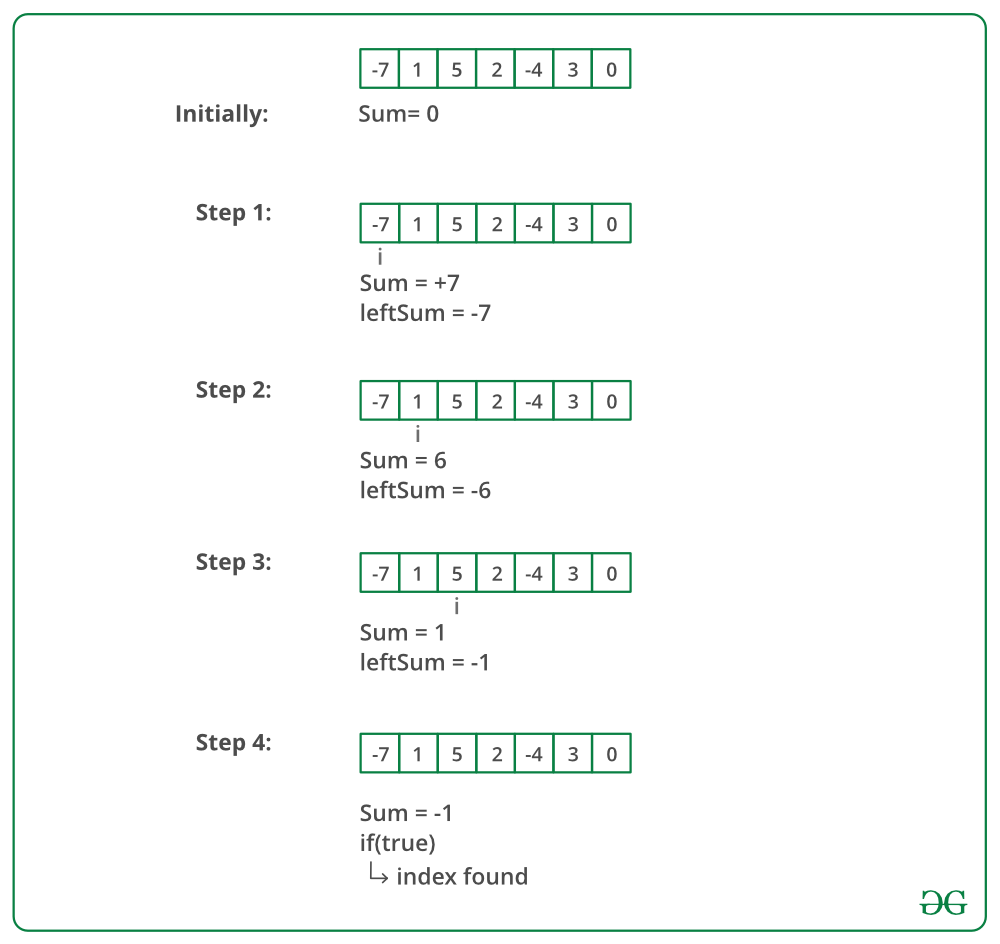
c) leftsum = leftsum + arr[i]

4) return -1

// If we come out of loop without returning then

// there is no equilibrium index

The image below shows the dry run of the above approach:



Below is the implementation of the above approach:

* C++
* C
* Java
* Python3
* C#
* PHP

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // C program to find equilibrium  // index of an array    #include <stdio.h>    int equilibrium(int arr[], int n)  {      int sum = 0; // initialize sum of whole array      int leftsum = 0; // initialize leftsum        /\* Find sum of the whole array \*/      for (int i = 0; i < n; ++i)          sum += arr[i];        for (int i = 0; i < n; ++i) {          sum -= arr[i]; // sum is now right sum for index i            if (leftsum == sum)              return i;            leftsum += arr[i];      }        /\* If no equilibrium index found, then return 0 \*/      return -1;  }    // Driver code  int main()  {      int arr[] = { -7, 1, 5, 2, -4, 3, 0 };      int arr\_size = sizeof(arr) / sizeof(arr[0]);      printf("First equilibrium index is %d",                   equilibrium(arr, arr\_size));        getchar();      return 0;  } |

**Output:**

First equilibrium index is 3

**Time Complexity:** O(n)

Find if there is a subarray with 0 sum

Given an array of positive and negative numbers, find if there is a subarray (of size at-least one) with 0 sum.

**Examples :**

Input: {4, 2, -3, 1, 6}

Output: true

There is a subarray with zero sum from index 1 to 3.

Input: {4, 2, 0, 1, 6}

Output: true

There is a subarray with zero sum from index 2 to 2.

Input: {-3, 2, 3, 1, 6}

Output: false

There is no subarray with zero sum.

[**Recommended: Please solve it on “*PRACTICE* ” first, before moving on to the solution.**](https://practice.geeksforgeeks.org/problems/subarray-with-0-sum/0)

A **simple solution** is to consider all subarrays one by one and check the sum of every subarray. We can run two loops: the outer loop picks a starting point i and the inner loop tries all subarrays starting from i (See [this](https://www.geeksforgeeks.org/find-subarray-with-given-sum/)for implementation). Time complexity of this method is O(n2).

We can also **use hashing**. The idea is to iterate through the array and for every element arr[i], calculate sum of elements form 0 to i (this can simply be done as sum += arr[i]). If the current sum has been seen before, then there is a zero sum array. Hashing is used to store the sum values, so that we can quickly store sum and find out whether the current sum is seen before or not.

Example :

arr[] = {1, 4, -2, -2, 5, -4, 3}

If we consider all prefix sums, we can

notice that there is a subarray with 0

sum when :

1) Either a prefix sum repeats or

2) Or prefix sum becomes 0.

Prefix sums for above array are:

**1**, 5, 3, **1**, 6, 2, 5

Since prefix sum 1 repeats, we have a subarray

with 0 sum.

Following is implementation of the above approach.

* C++
* Java
* Python3
* C#

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // A C++ program to find if there is a zero sum  // subarray  #include <bits/stdc++.h>  using namespace std;    bool subArrayExists(int arr[], int n)  {      unordered\_set<int> sumSet;        // Traverse through array and store prefix sums      int sum = 0;      for (int i = 0 ; i < n ; i++)      {          sum += arr[i];            // If prefix sum is 0 or it is already present          if (sum == 0 || sumSet.find(sum) != sumSet.end())              return true;            sumSet.insert(sum);      }      return false;  }    // Driver code  int main()  {      int arr[] =  {-3, 2, 3, 1, 6};      int n = sizeof(arr)/sizeof(arr[0]);      if (subArrayExists(arr, n))          cout << "Found a subarray with 0 sum";      else          cout << "No Such Sub Array Exists!";      return 0;  } |

**Output :**

No Such Sub Array Exists!

Time Complexity of this solution can be considered as O(n) under the assumption that we have good hashing function that allows insertion and retrieval operations in O(1) time.

Maximum subarray size, such that all subarrays of that size have sum less than k

Given an array of **n** positive integers and a positive integer **k**, the task is to find the maximum subarray size such that all subarrays of that size have sum of elements less than k.

**Examples :**

**Input :**  arr[] = {1, 2, 3, 4} and k = 8.

**Output :** 2

Sum of subarrays of size 1: 1, 2, 3, 4.

Sum of subarrays of size 2: 3, 5, 7.

Sum of subarrays of size 3: 6, 9.

Sum of subarrays of size 4: 10.

So, maximum subarray size such that all subarrays

of that size have sum of elements less than 8 is 2.

**Input :**  arr[] = {1, 2, 10, 4} and k = 8.

**Output :** -1

There is an array element with value greater than k,

so subarray sum cannot be less than k.

**Input :**  arr[] = {1, 2, 10, 4} and K = 14

**Output :** -2

[**Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.**](https://ide.geeksforgeeks.org/)

First of all, required subarray size must lie between 1 to n.

Now, since all the array element are positive integers, we can say that the prefix sum of any subarray shall be strictly increasing.  
Thus, we can say that

if arr[i] + arr[i + 1] + ..... + arr[j - 1] + arr[j] <= K

then arr[i] + arr[i + 1] + ..... + arr[j - 1] <= K, as

arr[j] is a positive integer.

So, we perform [Binary Search](http://quiz.geeksforgeeks.org/binary-search/) over the range 1 to n and find the highest subarray size such that all the subarrays of that size have sum of elements less than k.

Below is implementation of this approach.

* C/C++
* Java
* Python3
* C#
* PHP

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // C++ program to find maximum  // subarray size, such that all  // subarrays of that size have  // sum less than K.  #include<bits/stdc++.h>  using namespace std;    // Search for the maximum length of  // required subarray.  int bsearch(int prefixsum[], int n,                               int k)  {      // Initialize result      int ans = -1;        // Do Binary Search for largest      // subarray size      int left = 1, right = n;      while (left <= right)      {          int mid = (left + right) / 2;            // Check for all subarrays after mid          int i;          for (i = mid; i <= n; i++)          {              // Checking if all the subarrays              //  of a size less than k.              if (prefixsum[i] - prefixsum[i - mid] > k)                  break;          }            // All subarrays of size mid have          // sum less than or equal to k          if (i == n + 1)          {              left = mid + 1;              ans = mid;          }            // We found a subrray of size mid          // with sum greater than k          else              right = mid - 1;      }      return ans;  }    // Return the maximum subarray size,  // such that all subarray of that size  // have sum less than K.  int maxSize(int arr[], int n, int k)  {      // Initialize prefix sum array as 0.      int prefixsum[n + 1];      memset(prefixsum, 0, sizeof(prefixsum));        // Finding prefix sum of the array.      for (int i = 0; i < n; i++)          prefixsum[i + 1] = prefixsum[i] +                             arr[i];        return bsearch(prefixsum, n, k);  }    // Driver code  int main()  {      int arr[] = {1, 2, 10, 4};      int n = sizeof(arr) / sizeof(arr[0]);      int k = 14;      cout << maxSize(arr, n, k) << endl;      return 0;  } |

**Output :**

2

**Time Complexity :**O(n log n).

Find the prime numbers which can written as sum of most consecutive primes

Given an array of limits. For every limit, find the prime number which can be written as the sum of the most consecutive primes smaller than or equal to limit.

The maximum possible value of a limit is 10^4.

Example:

Input : arr[] = {10, 30}

Output : 5, 17

Explanation : There are two limit values 10 and 30.

Below limit 10, 5 is sum of two consecutive primes,

2 and 3. 5 is the prime number which is sum of largest

chain of consecutive below limit 10.

Below limit 30, 17 is sum of four consecutive primes.

2 + 3 + 5 + 7 = 17

[**Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.**](https://ide.geeksforgeeks.org/)

Below are steps.

1. Find all prime numbers below a maximum limit (10^6) using [Sieve of Sundaram](https://www.geeksforgeeks.org/sieve-sundaram-print-primes-smaller-n/) and store them in primes[].
2. Construct a prefix sum array **prime\_sum[]** for all prime numbers in primes[]  
   prime\_sum[i+1] = prime\_sum[i] + primes[i].  
   Difference between two values in prime\_sum[i] and prime\_sum[j] represents sum of consecutive primes from index i to index j.
3. Traverse two loops , outer loop from i (0 to limit) and inner loop from j (0 to i)
4. For every i, inner loop traverse (0 to i), we check if current sum of consecutive primes (**consSum** = prime\_sum[i] – prime\_sum[j]) is prime number or not (we search consSum in prime[] using [Binary search](http://quiz.geeksforgeeks.org/binary-search/)).
5. If consSum is prime number then we update the result if the current length is more than length of current result.

Below is implementation of above steps.

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // C++ program to find Longest Sum of consecutive  // primes  #include<bits/stdc++.h>  using namespace std;  const int MAX  = 10000;    // utility function for sieve of sundaram  void sieveSundaram(vector <int> &primes)  {      // In general Sieve of Sundaram, produces primes smaller      // than (2\*x + 2) for a number given number x. Since      // we want primes smaller than MAX, we reduce MAX to half      // This array is used to separate numbers of the form      // i+j+2ij from others where 1 <= i <= j      bool marked[MAX/2 + 1] = {0};        // Main logic of Sundaram. Mark all numbers which      // do not generate prime number by doing 2\*i+1      for (int i=1; i<=(sqrt(MAX)-1)/2; i++)          for (int j=(i\*(i+1))<<1; j<=MAX/2; j=j+2\*i+1)              marked[j] = true;        // Since 2 is a prime number      primes.push\_back(2);        // Print other primes. Remaining primes are of the      // form 2\*i + 1 such that marked[i] is false.      for (int i=1; i<=MAX/2; i++)          if (marked[i] == false)              primes.push\_back(2\*i + 1);  }    // function find the prime number which can be written  // as the sum of the most consecutive primes  int LSCPUtil(int limit, vector<int> &prime, long long int sum\_prime[])  {      // To store maximum length of consecutive primes that can      // sum to a limit      int max\_length = -1;        // The prime number (or result) that can be reprsented as      // sum of maximum number of primes.      int prime\_number = -1;        // Conisder all lengths of consecutive primes below limit.      for (int i=0; prime[i]<=limit; i++)      {          for (int j=0; j<i; j++)          {              // if we cross the limit, then break the loop              if (sum\_prime[i] - sum\_prime[j] > limit)                  break;                // sum\_prime[i]-sum\_prime[j] is prime number or not              long long int consSum  = sum\_prime[i] - sum\_prime[j];                // Check if sum of current length of consecutives is              // prime or not.              if (binary\_search(prime.begin(), prime.end(), consSum))              {                  // update the length and prime number                  if (max\_length < i-j+1)                  {                      max\_length = i-j+1;                      prime\_number = consSum;                  }              }          }      }        return prime\_number;  }    // Returns the prime number that can written as sum  // of longest chain of consecutive primes.  void LSCP(int arr[], int n)  {      // Store prime number in vector      vector<int> primes;      sieveSundaram(primes);        long long int sum\_prime[primes.size() + 1];        // Calculate sum of prime numbers and store them      // in sum\_prime array. sum\_prime[i] stores sum of      // prime numbers from primes[0] to primes[i-1]      sum\_prime[0] = 0;      for (int i = 1 ; i <= primes.size(); i++)          sum\_prime[i] = primes[i-1] + sum\_prime[i-1];        // Process all queries one by one      for (int i=0; i<n; i++)        cout << LSCPUtil(arr[i], primes, sum\_prime) << " ";  }    // Driver program  int main()  {      int arr[] = {10, 30, 40, 50, 1000};      int n = sizeof(arr)/sizeof(arr[0]);      LSCP(arr, n);      return 0;  } |

Output:

5 17 17 41 953

Longest Span with same Sum in two Binary arrays

Given two binary arrays arr1[] and arr2[] of same size n. Find length of the longest common span (i, j) where j >= i such that arr1[i] + arr1[i+1] + …. + arr1[j] = arr2[i] + arr2[i+1] + …. + arr2[j].

Expected time complexity is Θ(n).

**Examples :**

Input: arr1[] = {0, 1, 0, 0, 0, 0};

arr2[] = {1, 0, 1, 0, 0, 1};

Output: 4

The longest span with same sum is from index 1 to 4.

Input: arr1[] = {0, 1, 0, 1, 1, 1, 1};

arr2[] = {1, 1, 1, 1, 1, 0, 1};

Output: 6

The longest span with same sum is from index 1 to 6.

Input: arr1[] = {0, 0, 0};

arr2[] = {1, 1, 1};

Output: 0

Input: arr1[] = {0, 0, 1, 0};

arr2[] = {1, 1, 1, 1};

Output: 1

**[We strongly recommend that you click here and practice it, before moving on to the solution.](https://practice.geeksforgeeks.org/problem-page.php?pid=188" \t "_blank)**

**Method 1 (Simple Solution)**  
One by one by consider same subarrays of both arrays. For all subarrays, compute sums and if sums are same and current length is more than max length, then update max length. Below is C++ implementation of simple approach.

* C++
* Java
* Python3
* C#
* PHP

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // A Simple C++ program to find longest common  // subarray of two binary arrays with same sum  #include<bits/stdc++.h>  using namespace std;    // Returns length of the longest common subarray  // with same sum  int longestCommonSum(bool arr1[], bool arr2[], int n)  {      // Initialize result      int maxLen = 0;        // One by one pick all possible starting points      // of subarrays      for (int i=0; i<n; i++)      {         // Initialize sums of current subarrays         int sum1 = 0, sum2 = 0;           // Conider all points for starting with arr[i]         for (int j=i; j<n; j++)         {             // Update sums             sum1 += arr1[j];             sum2 += arr2[j];               // If sums are same and current length is             // more than maxLen, update maxLen             if (sum1 == sum2)             {               int len = j-i+1;               if (len > maxLen)                  maxLen = len;             }         }      }      return maxLen;  }    // Driver program to test above function  int main()  {      bool  arr1[] = {0, 1, 0, 1, 1, 1, 1};      bool  arr2[] = {1, 1, 1, 1, 1, 0, 1};      int n = sizeof(arr1)/sizeof(arr1[0]);      cout << "Length of the longest common span with same "              "sum is "<< longestCommonSum(arr1, arr2, n);      return 0;  } |

**Output :**

Length of the longest common span with same sum is 6

**Time Complexity :** O(n2)  
**Auxiliary Space :** O(1)

**Method 2 (Using Auxiliary Array)**  
The idea is based on below observations.

1. Since there are total n elements, maximum sum is n for both arrays.
2. Difference between two sums varies from **-n** to **n**. So there are total 2n + 1 possible values of difference.
3. If differences between prefix sums of two arrays become same at two points, then subarrays between these two points have same sum.

Below is Complete Algorithm.

1. Create an auxiliary array of size 2n+1 to store starting points of all possible values of differences (Note that possible values of differences vary from -n to n, i.e., there are total 2n+1 possible values)
2. Initialize starting points of all differences as -1.
3. Initialize **maxLen** as 0 and prefix sums of both arrays as 0, **preSum1** = 0, **preSum2** = 0
4. Traverse both arrays from i = 0 to n-1.
   1. Update prefix sums: preSum1 += arr1[i], preSum2 += arr2[i]
   2. Compute difference of current prefix sums: **curr\_diff**= preSum1 – preSum2
   3. Find index in diff array: **diffIndex** = n + curr\_diff // curr\_diff can be negative and can go till -n
   4. **If** curr\_diff is 0, then i+1 is maxLen so far
   5. **Else If** curr\_diff is seen first time, i.e., starting point of current diff is -1, then update starting point as i
   6. **Else** (curr\_diff is NOT seen first time), then consider i as ending point and find length of current same sum span. If this length is more, then update maxLen
5. Return maxLen

Below is the implementation of above algorithm.

* C++
* Java
* Python
* C#

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // A O(n) and O(n) extra space C++ program to find  // longest common subarray of two binary arrays with  // same sum  #include<bits/stdc++.h>  using namespace std;    // Returns length of the longest common sum in arr1[]  // and arr2[]. Both are of same size n.  int longestCommonSum(bool arr1[], bool arr2[], int n)  {      // Initialize result      int maxLen = 0;        // Initialize prefix sums of two arrays      int preSum1 = 0, preSum2 = 0;        // Create an array to store staring and ending      // indexes of all possible diff values. diff[i]      // would store starting and ending points for      // difference "i-n"      int diff[2\*n+1];        // Initialize all starting and ending values as -1.      memset(diff, -1, sizeof(diff));        // Traverse both arrays      for (int i=0; i<n; i++)      {          // Update prefix sums          preSum1 += arr1[i];          preSum2 += arr2[i];            // Comput current diff and index to be used          // in diff array. Note that diff can be negative          // and can have minimum value as -1.          int curr\_diff = preSum1 - preSum2;          int diffIndex = n + curr\_diff;            // If current diff is 0, then there are same number          // of 1's so far in both arrays, i.e., (i+1) is          // maximum length.          if (curr\_diff == 0)              maxLen = i+1;            // If current diff is seen first time, then update          // starting index of diff.          else if ( diff[diffIndex] == -1)              diff[diffIndex] = i;            // Current diff is already seen          else          {              // Find length of this same sum common span              int len = i - diff[diffIndex];                // Update max len if needed              if (len > maxLen)                  maxLen = len;          }      }      return maxLen;  }    // Driver code  int main()  {      bool  arr1[] = {0, 1, 0, 1, 1, 1, 1};      bool  arr2[] = {1, 1, 1, 1, 1, 0, 1};      int n = sizeof(arr1)/sizeof(arr1[0]);      cout << "Length of the longest common span with same "              "sum is "<< longestCommonSum(arr1, arr2, n);      return 0;  } |

**Output:**

Length of the longest common span with same sum is 6

Time Complexity: Θ(n)  
Auxiliary Space: Θ(n)

**Method 3 (Using Hashing)**

1. Find difference array arr[] such that arr[i] = arr1[i] – arr2[i].
2. [Largest subarray with equal number of 0s and 1s](https://www.geeksforgeeks.org/largest-subarray-with-equal-number-of-0s-and-1s/) in the difference array.

* C++
* Java

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // C++ program to find largest subarray  // with equal number of 0's and 1's.  #include <bits/stdc++.h>  using namespace std;    // Returns largest common subarray with equal  // number of 0s and 1s in both of t  int longestCommonSum(bool arr1[], bool arr2[], int n)  {      // Find difference between the two      int arr[n];      for (int i=0; i<n; i++)        arr[i] = arr1[i] - arr2[i];        // Creates an empty hashMap hM      unordered\_map<int, int> hM;        int sum = 0;     // Initialize sum of elements      int max\_len = 0; // Initialize result        // Traverse through the given array      for (int i = 0; i < n; i++)      {          // Add current element to sum          sum += arr[i];            // To handle sum=0 at last index          if (sum == 0)              max\_len = i + 1;            // If this sum is seen before,          // then update max\_len if required          if (hM.find(sum) != hM.end())            max\_len = max(max\_len, i - hM[sum]);            else // Else put this sum in hash table              hM[sum] = i;      }        return max\_len;  }    // Driver progra+m to test above function  int main()  {      bool  arr1[] = {0, 1, 0, 1, 1, 1, 1};      bool  arr2[] = {1, 1, 1, 1, 1, 0, 1};      int n = sizeof(arr1)/sizeof(arr1[0]);      cout << longestCommonSum(arr1, arr2, n);      return 0;  } |

**Output:**

6

Maximum subarray sum modulo m

Given an array of **n** elements and an integer m. The task is to find the maximum value of the sum of its subarray modulo m i.e find the sum of each subarray mod m and print the maximum value of this modulo operation.

**Examples:**

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

m = 7

Output : 6

All sub-arrays and their value:

{ 9 } => 9%7 = 2

{ 3 } => 3%7 = 3

{ 5 } => 5%7 = 5

{ 9, 5 } => 14%7 = 2

{ 9, 9 } => 18%7 = 4

{ 3, 9 } => 12%7 = 5

{ 3, 3 } => 6%7 = 6

{ 3, 9, 9 } => 21%7 = 0

{ 3, 3, 9 } => 15%7 = 1

{ 9, 9, 5 } => 23%7 = 2

{ 3, 3, 9, 9 } => 24%7 = 3

{ 3, 9, 9, 5 } => 26%7 = 5

{ 3, 3, 9, 9, 5 } => 29%7 = 1

Input : arr[] = {10, 7, 18}

m = 13

Output : 12

The subarray {7, 18} has maximum sub-array

sum modulo 13.

[**Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.**](https://ide.geeksforgeeks.org/)

**Method 1 (Brute Force):**  
Use brute force to find all the subarrays of the given array and find sum of each subarray mod m and keep track of maximum.

**Method 2 (efficient approach):**  
The idea is to compute prefix sum of array. We find maximum sum ending with every index and finally return overall maximum. To find maximum sum ending at index at index, we need to find the starting point of maximum sum ending with i. Below steps explain how to find the starting point.

Let prefix sum for index i be prefixi, i.e.,

prefixi = (arr[0] + arr[1] + .... arr[i] ) % m

Let maximum sum ending with i be, maxSumi.

Let this sum begins with index j.

maxSumi = (prefixi - prefixj + m) % m

From above expression it is clear that the

value of maxSumi becomes maximum when

prefixj is greater than prefixi

and closest to prefixi

We mainly have two operations in above algorithm.

1. Store all prefixes.
2. For current prefix, prefixi, find the smallest value greater than or equal to prefixi + 1.

For above operations, a self-balancing-binary-search-trees like AVL Tree, Red-Black Tree, etc are best suited. In below implementation we use [set in STL](http://quiz.geeksforgeeks.org/set-associative-containers-the-c-standard-template-library-stl/) which implements a self-balancing-binary-search-tree.

Below is the implementation of this approach:

* C++
* Python3

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // C++ program to find sub-array having maximum  // sum of elements modulo m.  #include<bits/stdc++.h>  using namespace std;    // Return the maximum sum subarray mod m.  int maxSubarray(int arr[], int n, int m)  {      int x, prefix = 0, maxim = 0;        set<int> S;      S.insert(0);        // Traversing the array.      for (int i = 0; i < n; i++)      {          // Finding prefix sum.          prefix = (prefix + arr[i])%m;            // Finding maximum of prefix sum.          maxim = max(maxim, prefix);            // Finding iterator pointing to the first          // element that is not less than value          // "prefix + 1", i.e., greater than or          // equal to this value.          auto it = S.lower\_bound(prefix+1);            if (it != S.end())              maxim = max(maxim, prefix - (\*it) + m );            // Inserting prefix in the set.          S.insert(prefix);      }        return maxim;  }    // Driver Program  int main()  {      int arr[] = { 3, 3, 9, 9, 5 };      int n = sizeof(arr)/sizeof(arr[0]);      int m = 7;      cout << maxSubarray(arr, n, m) << endl;      return 0;  } |

**Output:**

6

Maximum subarray size, such that all subarrays of that size have sum less than k

Given an array of **n** positive integers and a positive integer **k**, the task is to find the maximum subarray size such that all subarrays of that size have sum of elements less than k.

**Examples :**

**Input :**  arr[] = {1, 2, 3, 4} and k = 8.

**Output :** 2

Sum of subarrays of size 1: 1, 2, 3, 4.

Sum of subarrays of size 2: 3, 5, 7.

Sum of subarrays of size 3: 6, 9.

Sum of subarrays of size 4: 10.

So, maximum subarray size such that all subarrays

of that size have sum of elements less than 8 is 2.

**Input :**  arr[] = {1, 2, 10, 4} and k = 8.

**Output :** -1

There is an array element with value greater than k,

so subarray sum cannot be less than k.

**Input :**  arr[] = {1, 2, 10, 4} and K = 14

**Output :** -2

[**Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.**](https://ide.geeksforgeeks.org/)

First of all, required subarray size must lie between 1 to n.

Now, since all the array element are positive integers, we can say that the prefix sum of any subarray shall be strictly increasing.  
Thus, we can say that

if arr[i] + arr[i + 1] + ..... + arr[j - 1] + arr[j] <= K

then arr[i] + arr[i + 1] + ..... + arr[j - 1] <= K, as

arr[j] is a positive integer.

So, we perform [Binary Search](http://quiz.geeksforgeeks.org/binary-search/) over the range 1 to n and find the highest subarray size such that all the subarrays of that size have sum of elements less than k.

Below is implementation of this approach.

* C/C++
* Java
* Python3
* C#
* PHP

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // C++ program to find maximum  // subarray size, such that all  // subarrays of that size have  // sum less than K.  #include<bits/stdc++.h>  using namespace std;    // Search for the maximum length of  // required subarray.  int bsearch(int prefixsum[], int n,                               int k)  {      // Initialize result      int ans = -1;        // Do Binary Search for largest      // subarray size      int left = 1, right = n;      while (left <= right)      {          int mid = (left + right) / 2;            // Check for all subarrays after mid          int i;          for (i = mid; i <= n; i++)          {              // Checking if all the subarrays              //  of a size less than k.              if (prefixsum[i] - prefixsum[i - mid] > k)                  break;          }            // All subarrays of size mid have          // sum less than or equal to k          if (i == n + 1)          {              left = mid + 1;              ans = mid;          }            // We found a subrray of size mid          // with sum greater than k          else              right = mid - 1;      }      return ans;  }    // Return the maximum subarray size,  // such that all subarray of that size  // have sum less than K.  int maxSize(int arr[], int n, int k)  {      // Initialize prefix sum array as 0.      int prefixsum[n + 1];      memset(prefixsum, 0, sizeof(prefixsum));        // Finding prefix sum of the array.      for (int i = 0; i < n; i++)          prefixsum[i + 1] = prefixsum[i] +                             arr[i];        return bsearch(prefixsum, n, k);  }    // Driver code  int main()  {      int arr[] = {1, 2, 10, 4};      int n = sizeof(arr) / sizeof(arr[0]);      int k = 14;      cout << maxSize(arr, n, k) << endl;      return 0;  } |

**Output :**

2

**Time Complexity :**O(n log n).

Maximum occurred integer in n ranges

Given **n** ranges of the form **L** and **R**, the task is to find the maximum occurred integer in all the ranges. If more than one such integer exists, print the smallest one.  
0 <= Li, Ri < 1000000.

**Examples :**

**Input :** L1 = 1 R1 = 15

L2 = 4 R2 = 8

L3 = 3 R3 = 5

L3 = 1 R3 = 4

**Output :** 4

**Input :** L1 = 1 R1 = 15

L2 = 5 R2 = 8

L3 = 9 R3 = 12

L4 = 13 R4 = 20

L5 = 21 R5 = 30

**Output :** 5

Numbers having maximum occurrence i.e 2 are 5, 6,

7, 8, 9, 10, 11, 12, 13, 14, 15. The smallest number

among all are 5.

[**Recommended: Please solve it on “*PRACTICE* ” first, before moving on to the solution.**](https://practice.geeksforgeeks.org/problems/maximum-occured-integer/0)

A **simple solution** is to use hash table to store counts of all numbers. We traverse every interval from Li to Ri and increment count of all numbers present in every interval. Finally we traverse the hash table to find the number with maximum count. Time complexity of this solution is O(n \* MAX\_INTERVAL) where MAX\_INTERVAL is maximum number of elements in an interval.

An **efficient solution** requires linear time. We create an array arr[] of size 1000000 (limit given on maximum value of an interval). The idea is to add +1 to each Li index and -1 to corresponding Ri index in arr[]. After this, find the prefix sum of the array. Adding +1 at Li shows the starting point of ith Range and adding -1 shows the ending point of ith range. Finally we return the array index that has maximum prefix sum

Algorithm to solve the problem:

1. Initialize an array arr[] to 0.
2. For each range i, add +1 at Li index and -1 at Ri of the array.
3. Find the prefix sum of the array and find the smallest index having maximum prefix sum.

Below is the implementation of this approach:

* C++
* Java
* Python3
* C#
* PHP

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // C++ program to find maximum occurred element in  // given N ranges.  #include <bits/stdc++.h>  #define MAX 1000000  using namespace std;    // Return the maximum occurred element in all ranges.  int maximumOccurredElement(int L[], int R[], int n)  {      // Initalising all element of array to 0.      int arr[MAX];      memset(arr, 0, sizeof arr);        // Adding +1 at Li index and substracting 1      // at Ri index.      int maxi=-1;      for (int i = 0; i < n; i++) {          arr[L[i]] += 1;          arr[R[i] + 1] -= 1;          if(R[i]>maxi){              maxi=R[i];          }      }        // Finding prefix sum and index having maximum      // prefix sum.      int msum = arr[0],ind;      for (int i = 1; i < maxi+1; i++) {          arr[i] += arr[i - 1];          if (msum < arr[i]) {              msum = arr[i];              ind = i;          }      }        return ind;  }    // Driven Program  int main()  {      int L[] = { 1, 4, 9, 13, 21 };      int R[] = { 15, 8, 12, 20, 30 };      int n = sizeof(L) / sizeof(L[0]);        cout << maximumOccurredElement(L, R, n) << endl;      return 0;  } |

**Output:**

4

**Time Complexity:** O(n + MAX)

**Exercise:** Try for 0 <= Li, Ri <= 1000000000. (Hint: Use stl map).

Minimum cost for acquiring all coins with k extra coins allowed with every coin

You are given a list of N coins of different denominations. you can pay an amount equivalent to any 1 coin and can acquire that coin. In addition, once you have paid for a coin, we can choose at most K more coins and can acquire those for free. The task is to find the minimum amount required to acquire all the N coins for a given value of K.

**Examples :**

Input : coin[] = {100, 20, 50, 10, 2, 5},

k = 3

Output : 7

Input : coin[] = {1, 2, 5, 10, 20, 50},

k = 3

Output : 3

[**Recommended: Please try your approach on *{IDE}* first, before moving on to the solution.**](https://ide.geeksforgeeks.org/)

As per question, we can see that at a cost of 1 coin, we can acquire at most K+1 coins. Therefore, in order to acquire all the n coins, we will be choosing ceil(n/(k+1)) coins and the cost of choosing coins will be minimum if we choose smallest ceil(n/(k+1)) ( Greedy approach). Smallest ceil(n/(k+1)) coins can be found by simply sorting all the N values in increasing order.  
If we should check for time complexity (n log n) is for sorting element and (k) is for adding the total amount. So, finally Time Complexity : O(n log n).

* C++
* Java
* Python3
* C#
* PHP

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // C++ program to acquire all n coins  #include<bits/stdc++.h>  using namespace std;    // function to calculate min cost  int minCost(int coin[], int n, int k)  {      // sort the coins value      sort(coin, coin + n);        // calculate no. of      // coins needed      int coins\_needed = ceil(1.0 \* n /                              (k + 1));        // calculate sum of      // all selected coins      int ans = 0;      for (int i = 0; i <= coins\_needed - 1;                                        i++)          ans += coin[i];        return ans;  }    // Driver Code  int main()  {      int coin[] = {8, 5, 3, 10,                    2, 1, 15, 25};      int n = sizeof(coin) / sizeof(coin[0]);      int k = 3;      cout << minCost(coin, n, k);      return 0;  } |

**Output :**

3

Note that there are more efficient approaches to find given number of smallest values. For example, method 6 of [m largest(or smallest) elements in an array](https://www.geeksforgeeks.org/k-largestor-smallest-elements-in-an-array/) can find m’th smallest element in (n-m) Log m + m Log m).

**How to handle multiple queries for a single predefined array?**  
In the case, if you are asked to find the above answer for many different values of K, you have to compute it fast and our time complexity got increased as per number of queries for k. For the purpose to serve, we can maintain a prefix sum array after sorting all the N values and can answer queries easily and quickly.  
Suppose

* C++
* Java
* Python3
* C#
* PHP

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| // C++ program to acquire all  // n coins at minimum cost  // with multiple values of k.  #include<bits/stdc++.h>  using namespace std;    // Converts coin[] to prefix sum array  void preprocess(int coin[], int n)  {      // sort the coins value      sort(coin, coin + n);        // Maintain prefix sum array      for (int i = 1; i <= n - 1; i++)          coin[i] += coin[i - 1];  }    // Function to calculate min  // cost when we can get k extra  // coins after paying cost of one.  int minCost(int coin[], int n, int k)  {      // calculate no. of coins needed      int coins\_needed = ceil(1.0 \* n / (k + 1));        // return sum of from prefix array      return coin[coins\_needed - 1];  }    // Driver Code  int main()  {      int coin[] = {8, 5, 3, 10,                    2, 1, 15, 25};      int n = sizeof(coin) / sizeof(coin[0]);      preprocess(coin, n);      int k = 3;      cout << minCost(coin, n, k) << endl;      k = 7;      cout << minCost(coin, n, k) << endl;      return 0;  } |

**Output :**

3

1

After preprocessing, every query for a k takes O(1) time.

Random number generator in arbitrary probability distribution fashion

Given n numbers, each with some frequency of occurrence. Return a random number with probability proportional to its frequency of occurrence.

Example:

Let following be the given numbers.

arr[] = {10, 30, 20, 40}

Let following be the frequencies of given numbers.

freq[] = {1, 6, 2, 1}

The output should be

10 with probability 1/10

30 with probability 6/10

20 with probability 2/10

40 with probability 1/10

It is quite clear that the simple random number generator won’t work here as it doesn’t keep track of the frequency of occurrence.

We need to somehow transform the problem into a problem whose solution is known to us.

One simple method is to take an auxiliary array (say aux[]) and duplicate the numbers according to their frequency of occurrence. Generate a random number(say r) between 0 to Sum-1(including both), where Sum represents summation of frequency array (freq[] in above example). Return the random number aux[r] (Implementation of this method is left as an exercise to the readers).

The limitation of the above method discussed above is huge memory consumption when frequency of occurrence is high. If the input is 997, 8761 and 1, this method is clearly not efficient.

How can we reduce the memory consumption? Following is detailed algorithm that uses O(n) extra space where n is number of elements in input arrays.

**1.** Take an auxiliary array (say prefix[]) of size n.  
**2.** Populate it with prefix sum, such that prefix[i] represents sum of numbers from 0 to i.  
**3.** Generate a random number(say r) between 1 to Sum(including both), where Sum represents summation of input frequency array.  
**4.** Find index of Ceil of random number generated in step #3 in the prefix array. Let the index be index**c**.  
**5.** Return the random number arr[indexc], where arr[] contains the input n numbers.

  Before we go to the implementation part, let us have quick look at the algorithm with an example:  
      arr[]: {10, 20, 30}  
      freq[]: {2, 3, 1}  
      Prefix[]: {2, 5, 6}  
  Since last entry in prefix is 6, all possible values of r are [1, 2, 3, 4, 5, 6]  
         1: Ceil is 2. Random number generated is 10.  
         2: Ceil is 2. Random number generated is 10.  
         3: Ceil is 5. Random number generated is 20.  
         4: Ceil is 5. Random number generated is 20.  
         5: Ceil is 5. Random number generated is 20.  
         6. Ceil is 6. Random number generated is 30.  
  In the above example  
      10 is generated with probability 2/6.  
      20 is generated with probability 3/6.  
      30 is generated with probability 1/6.

**How does this work?**  
Any number input[i] is generated as many times as its frequency of occurrence because there exists count of integers in range(prefix[i – 1], prefix[i]] is input[i]. Like in the above example 3 is generated thrice, as there exists 3 integers 3, 4 and 5 whose ceil is 5.

* C++
* C

filter\_none

edit

play\_arrow

brightness\_4

|  |
| --- |
| //C program to generate random numbers according to given frequency distribution  #include <stdio.h>  #include <stdlib.h>    // Utility function to find ceiling of r in arr[l..h]  int findCeil(int arr[], int r, int l, int h)  {      int mid;      while (l < h)      {           mid = l + ((h - l) >> 1);  // Same as mid = (l+h)/2          (r > arr[mid]) ? (l = mid + 1) : (h = mid);      }      return (arr[l] >= r) ? l : -1;  }    // The main function that returns a random number from arr[] according to  // distribution array defined by freq[]. n is size of arrays.  int myRand(int arr[], int freq[], int n)  {      // Create and fill prefix array      int prefix[n], i;      prefix[0] = freq[0];      for (i = 1; i < n; ++i)          prefix[i] = prefix[i - 1] + freq[i];        // prefix[n-1] is sum of all frequencies. Generate a random number      // with value from 1 to this sum      int r = (rand() % prefix[n - 1]) + 1;        // Find index of ceiling of r in prefix arrat      int indexc = findCeil(prefix, r, 0, n - 1);      return arr[indexc];  }    // Driver program to test above functions  int main()  {      int arr[]  = {1, 2, 3, 4};      int freq[] = {10, 5, 20, 100};      int i, n = sizeof(arr) / sizeof(arr[0]);        // Use a different seed value for every run.      srand(time(NULL));        // Let us generate 10 random numbers accroding to      // given distribution      for (i = 0; i < 5; i++)        printf("%d\n", myRand(arr, freq, n));        return 0;  } |

**Output:**May be different for different runs

4

3

4

4

4