# Given an array A[] and a number x, check for pair in A[] with sum as x

Write a C program that, given an array A[] of n numbers and another number x, determines whether or not there exist two elements in S whose sum is exactly x. 

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=552)

**METHOD 1 (Use Sorting)**

Algorithm:

hasArrayTwoCandidates (A[], ar\_size, sum)

1) Sort the array in non-decreasing order.

2) Initialize two index variables to find the candidate

elements in the sorted array.

(a) Initialize first to the leftmost index: l = 0

(b) Initialize second the rightmost index: r = ar\_size-1

3) Loop while l < r.

(a) If (A[l] + A[r] == sum) then return 1

(b) Else if( A[l] + A[r] < sum ) then l++

(c) Else r--

4) No candidates in whole array - return 0

Time Complexity: Depends on what sorting algorithm we use. If we use Merge Sort or Heap Sort then (-)(nlogn) in worst case. If we use Quick Sort then O(n^2) in worst case.  
Auxiliary Space : Again, depends on sorting algorithm. For example auxiliary space is O(n) for merge sort and O(1) for Heap Sort.

Example:  
Let Array be {1, 4, 45, 6, 10, -8} and sum to find be 16

Sort the array  
A = {-8, 1, 4, 6, 10, 45}

Initialize l = 0, r = 5  
A[l] + A[r] ( -8 + 45) > 16 => decrement r. Now r = 10  
A[l] + A[r] ( -8 + 10) < 2 => increment l. Now l = 1  
A[l] + A[r] ( 1 + 10) < 16 => increment l. Now l = 2  
A[l] + A[r] ( 4 + 10) < 14 => increment l. Now l = 3  
A[l] + A[r] ( 6 + 10) == 16 => Found candidates (return 1)

Note: If there are more than one pair having the given sum then this algorithm reports only one. Can be easily extended for this though.

# Majority Element

**Majority Element:** A majority element in an array A[] of size n is an element that appears more than n/2 times (and hence there is at most one such element).

Write a function which takes an array and emits the majority element (if it exists), otherwise prints NONE as follows:

I/P : 3 3 4 2 4 4 2 4 4

O/P : 4

I/P : 3 3 4 2 4 4 2 4

O/P : NONE

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=305)

**METHOD 1 (Basic)**  
The basic solution is to have two loops and keep track of maximum count for all different elements. If maximum count becomes greater than n/2 then break the loops and return the element having maximum count. If maximum count doesn’t become more than n/2 then majority element doesn’t exist.

**Time Complexity:** O(n\*n).  
**Auxiliary Space :** O(1).

**METHOD 2 (Using Binary Search Tree)**  
Thanks to Sachin Midha for suggesting this solution. Node of the Binary Search Tree (used in this approach) will be as follows.

|  |
| --- |
| struct tree  {    int element;    int count;  }BST; |

Run on IDE

Insert elements in BST one by one and if an element is already present then increment the count of the node. At any stage, if count of a node becomes more than n/2 then return.  
The method works well for the cases where n/2+1 occurrences of the majority element is present in the starting of the array, for example {1, 1, 1, 1, 1, 2, 3, 4}.  
 **Time Complexity:** If a binary search tree is used then time complexity will be O(n^2). If a [self-balancing-binary-search](http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree) tree is used then O(nlogn)  
**Auxiliary Space:**O(n)

**METHOD 3 (Using Moore’s Voting Algorithm)**  
This is a two step process.  
1. Get an element occurring most of the time in the array. This phase will make sure that if there is a majority element then it will return that only.  
2. Check if the element obtained from above step is majority element.

1. Finding a Candidate:  
The algorithm for first phase that works in O(n) is known as Moore’s Voting Algorithm. Basic idea of the algorithm is if we cancel out each occurrence of an element e with all the other elements that are different from e then e will exist till end if it is a majority element.

findCandidate(a[], size)

1. Initialize index and count of majority element

maj\_index = 0, count = 1

2. Loop for i = 1 to size – 1

(a) If a[maj\_index] == a[i]

count++

(b) Else

count--;

(c) If count == 0

maj\_index = i;

count = 1

3. Return a[maj\_index]

Above algorithm loops through each element and maintains a count of a[maj\_index], If next element is same then increments the count, if next element is not same then decrements the count, and if the count reaches 0 then changes the maj\_index to the current element and sets count to 1.  
First Phase algorithm gives us a candidate element. In second phase we need to check if the candidate is really a majority element. Second phase is simple and can be easily done in O(n). We just need to check if count of the candidate element is greater than n/2.

Example:  
A[] = 2, 2, 3, 5, 2, 2, 6  
Initialize:  
maj\_index = 0, count = 1 –> candidate ‘2?  
2, 2, 3, 5, 2, 2, 6

Same as a[maj\_index] => count = 2  
2, 2, 3, 5, 2, 2, 6

Different from a[maj\_index] => count = 1  
2, 2, 3, 5, 2, 2, 6

Different from a[maj\_index] => count = 0  
Since count = 0, change candidate for majority element to 5 => maj\_index = 3, count = 1  
2, 2, 3, 5, 2, 2, 6

Different from a[maj\_index] => count = 0  
Since count = 0, change candidate for majority element to 2 => maj\_index = 4  
2, 2, 3, 5, 2, 2, 6

Same as a[maj\_index] => count = 2  
2, 2, 3, 5, 2, 2, 6

Different from a[maj\_index] => count = 1

Finally candidate for majority element is 2.

First step uses Moore’s Voting Algorithm to get a candidate for majority element.

2.Check if the element obtained in step 1 is majority

printMajority (a[], size)

1. Find the candidate for majority

2. If candidate is majority. i.e., appears more than n/2 times.

Print the candidate

3. Else

Print "NONE"

**Implementation of method 3:**

# Find the Number Occurring Odd Number of Times

Given an array of positive integers. All numbers occur even number of times except one number which occurs odd number of times. Find the number in O(n) time & constant space.

**Example:**  
I/P = [1, 2, 3, 2, 3, 1, 3]  
O/P = 3

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=836)

A **Simple Solution** is to run two nested loops. The outer loop picks all elements one by one and inner loop counts number of occurrences of the element picked by outer loop. Time complexity of this solution is O(n2).

A **Better Solutio**n is to use Hashing. Use array elements as key and their counts as value. Create an empty hash table. One by one traverse the given array elements and store counts. Time complexity of this solution is O(n). But it requires extra space for hashing.

The **Best Solution** is to do bitwise XOR of all the elements. XOR of all elements gives us odd occurring element. Please note that XOR of two elements is 0 if both elements are same and XOR of a number x with 0 is x.

Below are implementations of this best approach.

**Program:**

# Largest Sum Contiguous Subarray

Write an efficient C program to find the sum of contiguous subarray within a one-dimensional array of numbers which has the largest sum.

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=106)

**Kadane’s Algorithm:**

Initialize:

max\_so\_far = 0

max\_ending\_here = 0

Loop for each element of the array

(a) max\_ending\_here = max\_ending\_here + a[i]

(b) if(max\_ending\_here < 0)

max\_ending\_here = 0

(c) if(max\_so\_far < max\_ending\_here)

max\_so\_far = max\_ending\_here

return max\_so\_far

**Explanation:**  
Simple idea of the Kadane's algorithm is to look for all positive contiguous segments of the array (max\_ending\_here is used for this). And keep track of maximum sum contiguous segment among all positive segments (max\_so\_far is used for this). Each time we get a positive sum compare it with max\_so\_far and update max\_so\_far if it is greater than max\_so\_far

Lets take the example:

{-2, -3, 4, -1, -2, 1, 5, -3}

max\_so\_far = max\_ending\_here = 0

for i=0, a[0] = -2

max\_ending\_here = max\_ending\_here + (-2)

Set max\_ending\_here = 0 because max\_ending\_here < 0

for i=1, a[1] = -3

max\_ending\_here = max\_ending\_here + (-3)

Set max\_ending\_here = 0 because max\_ending\_here < 0

for i=2, a[2] = 4

max\_ending\_here = max\_ending\_here + (4)

max\_ending\_here = 4

max\_so\_far is updated to 4 because max\_ending\_here greater

than max\_so\_far which was 0 till now

for i=3, a[3] = -1

max\_ending\_here = max\_ending\_here + (-1)

max\_ending\_here = 3

for i=4, a[4] = -2

max\_ending\_here = max\_ending\_here + (-2)

max\_ending\_here = 1

for i=5, a[5] = 1

max\_ending\_here = max\_ending\_here + (1)

max\_ending\_here = 2

for i=6, a[6] = 5

max\_ending\_here = max\_ending\_here + (5)

max\_ending\_here = 7

max\_so\_far is updated to 7 because max\_ending\_here is

greater than max\_so\_far

for i=7, a[7] = -3

max\_ending\_here = max\_ending\_here + (-3)

max\_ending\_here = 4

# Search an element in a sorted and rotated array

An element in a sorted array can be found in O(log n) time via binary search. But suppose we rotate an ascending order sorted array at some pivot unknown to you beforehand. So for instance, 1 2 3 4 5 might become 3 4 5 1 2. Devise a way to find an element in the rotated array in O(log n) time.

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=146)

**All solutions provided here assume that all elements in array are distinct.**

The idea is to find the pivot point, divide the array in two sub-arrays and call binary search.  
The main idea for finding pivot is – for a sorted (in increasing order) and pivoted array, pivot element is the only only element for which next element to it is smaller than it.  
Using above criteria and binary search methodology we can get pivot element in O(logn) time

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

Element to Search = 1

1) Find out pivot point and divide the array in two

sub-arrays. (pivot = 2) /\*Index of 5\*/

2) Now call binary search for one of the two sub-arrays.

(a) **If** element is greater than 0th element then

search in left array

(b) **Else** Search in right array

(1 will go in else as 1 < 0th element(3))

3) **If** element is found in selected sub-array then return index

**Else** return -1.

# Program for array rotation

Write a function rotate(ar[], d, n) that rotates arr[] of size n by d elements.  
  
Array

Rotation of the above array by 2 will make array

ArrayRotation1

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=360)

**METHOD 1 (Use temp array)**

Input arr[] = [1, 2, 3, 4, 5, 6, 7], d = 2, n =7

1) Store d elements in a temp array

temp[] = [1, 2]

2) Shift rest of the arr[]

arr[] = [3, 4, 5, 6, 7, 6, 7]

3) Store back the d elements

arr[] = [3, 4, 5, 6, 7, 1, 2]

**Time complexity** O(n)  
**Auxiliary Space:**O(d)

**METHOD 2 (Rotate one by one)**

leftRotate(arr[], d, n)

start

For i = 0 to i < d

Left rotate all elements of arr[] by one

end

To rotate by one, store arr[0] in a temporary variable temp, move arr[1] to arr[0], arr[2] to arr[1] …and finally temp to arr[n-1]

Let us take the same example arr[] = [1, 2, 3, 4, 5, 6, 7], d = 2  
Rotate arr[] by one 2 times  
We get [2, 3, 4, 5, 6, 7, 1] after first rotation and [ 3, 4, 5, 6, 7, 1, 2] after second rotation.

Implementation:

# Reversal algorithm for array rotation

Write a function rotate(arr[], d, n) that rotates arr[] of size n by d elements.

Example:

Input: arr[] = [1, 2, 3, 4, 5, 6, 7]

d = 2

Output: arr[] = [3, 4, 5, 6, 7, 1, 2]

Array

Rotation of the above array by 2 will make array

ArrayRotation1

**Method 4(The Reversal Algorithm)**  
Please read [this](http://geeksforgeeks.org/?p=2398)for first three methods of array rotation.

**Algorithm:**

rotate(arr[], d, n)

reverse(arr[], 1, d) ;

reverse(arr[], d + 1, n);

reverse(arr[], l, n);

Let AB are the two parts of the input array where A = arr[0..d-1] and B = arr[d..n-1]. The idea of the algorithm is:  
Reverse A to get ArB. /\* Ar is reverse of A \*/  
Reverse B to get ArBr. /\* Br is reverse of B \*/  
Reverse all to get (ArBr) r = BA.

For arr[] = [1, 2, 3, 4, 5, 6, 7], d =2 and n = 7  
A = [1, 2] and B = [3, 4, 5, 6, 7]  
Reverse A, we get ArB = [2, 1, 3, 4, 5, 6, 7]  
Reverse B, we get ArBr = [2, 1, 7, 6, 5, 4, 3]  
Reverse all, we get (ArBr)r = [3, 4, 5, 6, 7, 1, 2]

# Maximum sum such that no two elements are adjacent

Given an array of positive numbers, find the maximum sum of a subsequence with the constraint that no 2 numbers in the sequence should be adjacent in the array. So 3 2 7 10 should return 13 (sum of 3 and 10) or 3 2 5 10 7 should return 15 (sum of 3, 5 and 7).Answer the question in most efficient way.

Examples :

Input : arr[] = {5, 5, 10, 100, 10, 5}

Output : 110

Input : arr[] = {1, 2, 3}

Output : 4

Input : arr[] = {1, 20, 3}

Output : 20

## [We strongly recommend that you click here and practice it, before moving on to the solution.](http://www.practice.geeksforgeeks.org/problem-page.php?pid=531" \t "_blank)

   
**Algorithm:**  
Loop for all elements in arr[] and maintain two sums incl and excl where incl = Max sum including the previous element and excl = Max sum excluding the previous element.

Max sum excluding the current element will be max(incl, excl) and max sum including the current element will be excl + current element (Note that only excl is considered because elements cannot be adjacent).

At the end of the loop return max of incl and excl.

**Example:**

arr[] = {5, 5, 10, 40, 50, 35}

inc = 5

exc = 0

For i = 1 (current element is 5)

incl = (excl + arr[i]) = 5

excl = max(5, 0) = 5

For i = 2 (current element is 10)

incl = (excl + arr[i]) = 15

excl = max(5, 5) = 5

For i = 3 (current element is 40)

incl = (excl + arr[i]) = 45

excl = max(5, 15) = 15

For i = 4 (current element is 50)

incl = (excl + arr[i]) = 65

excl = max(45, 15) = 45

For i = 5 (current element is 35)

incl = (excl + arr[i]) = 80

excl = max(5, 15) = 65

And 35 is the last element. So, answer is max(incl, excl) = 80