

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:

$maj_index = 0, count = 1 \rightarrow$ candidate '2'?

2, 2, 3, 5, 2, 2, 6

Same as $a[maj_index] \Rightarrow count = 2$

2, 2, 3, 5, 2, 2, 6

Different from $a[maj_index] \Rightarrow count = 1$

2, 2, 3, 5, 2, 2, 6

Different from $a[maj_index] \Rightarrow count = 0$

Since count = 0, change candidate for majority element to 5 $\Rightarrow maj_index = 3, count = 1$

2, 2, 3, 5, 2, 2, 6

Different from $a[maj_index] \Rightarrow count = 0$

Since count = 0, change candidate for majority element to 2 $\Rightarrow maj_index = 4$

2, 2, 3, 5, 2, 2, 6

Same as $a[maj_index] \Rightarrow count = 2$

2, 2, 3, 5, 2, 2, 6

Different from $a[maj_index] \Rightarrow count = 1$

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

We need second phase for the case when there is no majority element

like:

1,2,3,4,5,6,7,8,9,0

First Phase will return 0

Second Phase will say that it is not a majority element

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

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 $O(n^2)$.

A **Better Solution** 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 .

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.

Limitation: When all elements are negative
Then we will have to check first
if all negative
then return the maximum element in array
if not
apply Kadane

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`

Collect

Check if sum less than zero

Check if new maximum sum sub array found

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`

Notes:

Algorithm doesn't work for all negative numbers. It simply returns 0 if all numbers are negative. For handling this we can add an extra phase before actual implementation. The phase will look if all numbers are negative, if they are it will return maximum of them (or smallest in terms of absolute value). There may be other ways to handle it though.

Above program can be optimized further, if we compare `max_so_far` with `max_ending_here` only if `max_ending_here` is greater than 0.

Find the Missing Number

You are given a list of $n-1$ integers and these integers are in the range of 1 to n . There are no duplicates in list. One of the integers is missing in the list. Write an efficient code to find the missing integer.

We strongly recommend that you click [here](#) and practice it, before moving on to the solution.

Example:

I/P [1, 2, 4, ,6, 3, 7, 8]

O/P 5

METHOD 1(Use sum formula)

Algorithm:

1. Get the sum of numbers

$$\text{total} = n*(n+1)/2$$

- 2 Subtract all the numbers from sum and you will get the missing number.

METHOD 2(Use XOR)

- 1) XOR all the array elements, let the result of XOR be $X1$.
- 2) XOR all numbers from 1 to n , let XOR be $X2$.
- 3) XOR of $X1$ and $X2$ gives the missing number.

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**.

3	4	5	1	2
---	---	---	---	---



We strongly recommend that you click [here](#) and practice it, before moving on to the solution.

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 element for which next element to it is smaller than it**

Using above criteria and binary search methodology we can get pivot element in $O(\log n)$ 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 $1 < 0\text{th element}(3)$)
- 3) If element is found in selected sub-array then return index
Else return -1.



Implementation:

```
/* Program to search an element in a sorted and pivoted array*/
#include <stdio.h>

int findPivot(int[], int, int);
int binarySearch(int[], int, int, int);

/* Searches an element key in a pivoted sorted array arr[]
of size n */
int pivotedBinarySearch(int arr[], int n, int key)
{
    int pivot = findPivot(arr, 0, n-1);

    // If we didn't find a pivot, then array is not rotated at all
    if (pivot == -1)
        return binarySearch(arr, 0, n-1, key);

    // If we found a pivot, then first compare with pivot and then
    // search in two subarrays around pivot
    if (arr[pivot] == key)
        return pivot;
    if (arr[0] <= key)
        return binarySearch(arr, 0, pivot-1, key);
    return binarySearch(arr, pivot+1, n-1, key);
}

/* Function to get pivot. For array 3, 4, 5, 6, 1, 2 it returns
3 (index of 6) */
int findPivot(int arr[], int low, int high)
{
    // base cases
    if (high < low) return -1;
    if (high == low) return low;

    int mid = (low + high)/2; /*low + (high - low)/2;*/
    if (mid < high && arr[mid] > arr[mid + 1])
        return mid;
    if (mid > low && arr[mid] < arr[mid - 1])
        return (mid-1);
    if (arr[low] >= arr[mid])
        return findPivot(arr, low, mid-1);
    return findPivot(arr, mid + 1, high);
}

/* Standard Binary Search function*/
int binarySearch(int arr[], int low, int high, int key)
{
    if (high < low)
        return -1;
    int mid = (low + high)/2; /*low + (high - low)/2;*/
    if (key == arr[mid])
        return mid;
    if (key > arr[mid])
        return binarySearch(arr, (mid + 1), high, key);
    return binarySearch(arr, low, (mid - 1), key);
}

/* Driver program to check above functions */
int main()
{
    // Let us search 3 in below array
    int arr1[] = {5, 6, 7, 8, 9, 10, 1, 2, 3};
    int n = sizeof(arr1)/sizeof(arr1[0]);
    int key = 3;
    printf("Index: %d\n", pivotedBinarySearch(arr1, n, key));
    return 0;
}
```

[Run on IDE](#)

Output:

Index of the element is 8

Method 2 (By comparing the medians of two arrays)

This method works by first getting medians of the two sorted arrays and then comparing them.

Let ar1 and ar2 be the input arrays.

Algorithm:

- 1) Calculate the medians m1 and m2 of the input arrays ar1[] and ar2[] respectively.
- 2) If m1 and m2 both are equal then we are done.
return m1 (or m2)
- 3) If m1 is greater than m2, then median is present in one of the below two subarrays.
 - a) From first element of ar1 to m1 (ar1[0...|_n/2_|])
 - b) From m2 to last element of ar2 (ar2[|_n/2_|...n-1])
- 4) If m2 is greater than m1, then median is present in one of the below two subarrays.
 - a) From m1 to last element of ar1 (ar1[|_n/2_|...n-1])
 - b) From first element of ar2 to m2 (ar2[0...|_n/2_|])
- 5) Repeat the above process until size of both the subarrays becomes 2.
- 6) If size of the two arrays is 2 then use below formula to get the median.
$$\text{Median} = (\max(\text{ar1}[0], \text{ar2}[0]) + \min(\text{ar1}[1], \text{ar2}[1]))/2$$

Example

```
ar1[] = {1, 12, 15, 26, 38}
ar2[] = {2, 13, 17, 30, 45}
```

For above two arrays m1 = 15 and m2 = 17

For the above ar1[] and ar2[], m1 is smaller than m2. So median is present in one of the following two subarrays.

[15, 26, 38] and [2, 13, 17]

Let us repeat the process for above two subarrays:

m1 = 26 m2 = 13.

m1 is greater than m2. So the subarrays become

[15, 26] and [13, 17]

Now size is 2, so median = $(\max(\text{ar1}[0], \text{ar2}[0]) + \min(\text{ar1}[1], \text{ar2}[1]))/2$
$$= (\max(15, 13) + \min(26, 17))/2$$
$$= (15 + 17)/2$$
$$= 16$$

Rotate the array by d

METHOD 3 (A Juggling Algorithm)

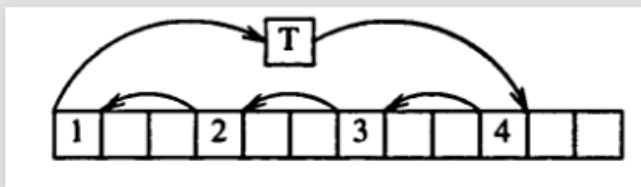
This is an extension of method 2. Instead of moving one by one, divide the array in different sets where number of sets is equal to GCD of n and d and move the elements within sets.

If GCD is 1 as is for the above example array ($n = 7$ and $d = 2$), then elements will be moved within one set only, we just start with $\text{temp} = \text{arr}[0]$ and keep moving $\text{arr}[i+d]$ to $\text{arr}[i]$ and finally store temp at the right place.

Here is an example for $n = 12$ and $d = 3$. GCD is 3 and

Let $\text{arr}[]$ be {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

a) Elements are first moved in first set - (See below diagram for this movement)



$\text{arr}[]$ after this step --> {4 2 3 7 5 6 10 8 9 1 11 12}

b) Then in second set.

$\text{arr}[]$ after this step --> {4 5 3 7 8 6 10 11 9 1 2 12}

c) Finally in third set.

$\text{arr}[]$ after this step --> {4 5 6 7 8 9 10 11 12 1 2 3}



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]`

1	2	3	4	5	6	7
---	---	---	---	---	---	---

Rotation of the above array by 2 will make array

3	4	5	6	7	1	2
---	---	---	---	---	---	---

Method 4(The Reversal Algorithm)

Please read [this](#) for first three methods of array rotation.

Algorithm:

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

Block swap algorithm for array rotation

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

1	2	3	4	5	6	7
---	---	---	---	---	---	---

Rotation of the above array by 2 will make array

3	4	5	6	7	1	2
---	---	---	---	---	---	---

Algorithm:

Initialize `A = arr[0..d-1]` and `B = arr[d..n-1]`

1) Do following until size of A is equal to size of B

- If A is shorter, divide B into `B1` and `Br` such that `Br` is of same length as A. Swap A and `Br` to change `AB1Br` into `BrB1A`. Now A is at its final place, so recur on pieces of B.
- If A is longer, divide A into `A1` and `An` such that `A1` is of same length as B. Swap `A1` and B to change `A1ArB` into `BArA1`. Now B is at its final place, so recur on pieces of A.

2) Finally when A and B are of equal size, block swap them.

Maximum sum such that no two elements are adjacent

Question: 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.

We strongly recommend that you click here and practice it, before moving on to the solution.

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}
```

```
incl = 5
```

```
excl = 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`

Implementation:

```
#include<stdio.h>

/*Function to return max sum such that no two elements
are adjacent */
int FindMaxSum(int arr[], int n)
{
    int incl = arr[0];
    int excl = 0;
    int excl_new;
    int i;

    for (i = 1; i < n; i++)
    {
        /* current max excluding i */
        excl_new = (incl > excl)? incl: excl;

        /* current max including i */
        incl = excl + arr[i];
        excl = excl_new;
    }

    /* return max of incl and excl */
    return ((incl > excl)? incl : excl);
}

/* Driver program to test above function */
int main()
{
    int arr[] = {5, 5, 10, 100, 10, 5};
    printf("%d \n", FindMaxSum(arr, 6));
    getchar();
    return 0;
}
```

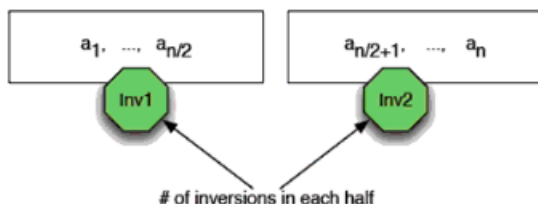
[Run on IDE](#)

Time Complexity: $O(n)$

Number of Inversions

METHOD 2(Enhance Merge Sort)

Suppose we know 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 we have to count during the merge step. Therefore, to get number of inversions, we need to add number of inversions in left subarray, right subarray and merge().

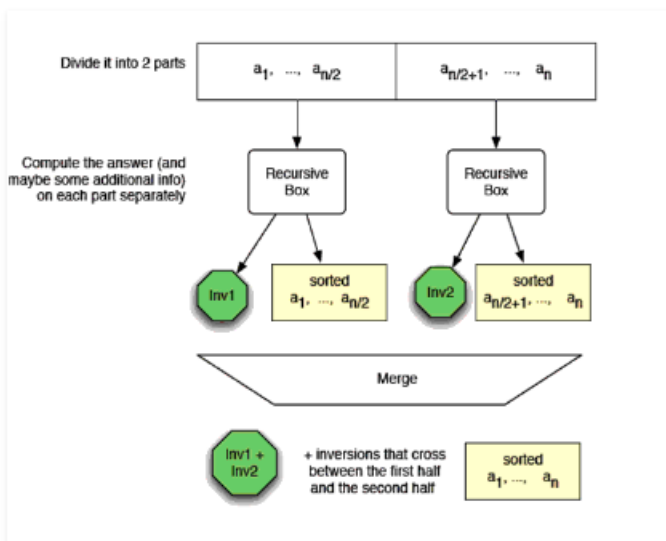


How to get number of inversions in merge()?

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 ($a[i+1], a[i+2] \dots a[mid]$) will be greater than $a[j]$.



The complete picture:



Check for Majority Element in a sorted array

Question: Write a C function to find if a given integer x appears more than $n/2$ times in a sorted array of n integers.

Basically, we need to write a function say `isMajority()` that takes an array (`arr[]`), array's size (n) and a number to be searched (x) as parameters and returns true if x is a majority element (present more than $n/2$ times).

Examples:

Input: `arr[] = {1, 2, 3, 3, 3, 3, 10}`, $x = 3$

Output: True (x appears more than $n/2$ times in the given array)

Input: `arr[] = {1, 1, 2, 4, 4, 4, 6, 6}`, $x = 4$

Output: False (x doesn't appear more than $n/2$ times in the given array)

Input: `arr[] = {1, 1, 1, 2, 2}`, $x = 1$

Output: True (x appears more than $n/2$ times in the given array)

METHOD 1 (Using Linear Search)

Linearly search for the first occurrence of the element, once you find it (let at index i), check element at index $i + n/2$.

If element is present at $i+n/2$ then return 1 else return 0.

C Java

```
/* C Program to check for majority element in a sorted array */
#include <stdio.h>
#include <stdbool.h>
```

```
bool isMajority(int arr[], int n, int x)
{
    int i;

    /* get last index according to n (even or odd) */
    int last_index = n%2? (n/2+1): (n/2);

    /* search for first occurrence of x in arr[] */
    for (i = 0; i < last_index; i++)
    {
        /* check if x is present and is present more than n/2
        times */
        if (arr[i] == x && arr[i+n/2] == x)
            return 1;
    }
    return 0;
}
```

```
/* Driver program to check above function */
int main()
{
    int arr[] = {1, 2, 3, 4, 4, 4, 4};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 4;
    if (isMajority(arr, n, x))
        printf("%d appears more than %d times in arr[]",
            x, n/2);
    else
        printf("%d does not appear more than %d times in arr[]",
            x, n/2);

    return 0;
}
```

Run on IDE

Output:

4 appears more than 3 times in arr[]

Time Complexity: $O(n)$

METHOD 2 (Using Binary Search)

Use binary search methodology to find the first occurrence of the given number. The criteria for binary search is important here.

C Java

```
/* Program to check for majority element in a sorted array */
#include <stdio.h>
#include <stdbool.h>

/* If x is present in arr[low...high] then returns the index of
first occurrence of x, otherwise returns -1 */
int _binarySearch(int arr[], int low, int high, int x);

/* This function returns true if the x is present more than n/2
times in arr[] of size n */
bool isMajority(int arr[], int n, int x)
{
    /* Find the index of first occurrence of x in arr[] */
    int i = _binarySearch(arr, 0, n-1, x);

    /* If element is not present at all, return false*/
    if (i == -1)
        return false;

    /* check if the element is present more than n/2 times */
    if (((i + n/2) <= (n - 1)) && arr[i + n/2] == x)
        return true;
    else
        return false;
}

/* If x is present in arr[low...high] then returns the index of
first occurrence of x, otherwise returns -1 */
int _binarySearch(int arr[], int low, int high, int x)
{
    if (high >= low)
    {
        int mid = (low + high)/2; /*low + (high - low)/2*/

        /* Check if arr[mid] is the first occurrence of x.
        arr[mid] is first occurrence if x is one of the following
        is true:
        (i) mid == 0 and arr[mid] == x
        (ii) arr[mid-1] < x and arr[mid] == x
        */
        if ((mid == 0 || x > arr[mid-1]) && (arr[mid] == x))
            return mid;
        else if (x > arr[mid])
            return _binarySearch(arr, (mid + 1), high, x);
        else
            return _binarySearch(arr, low, (mid - 1), x);
    }

    return -1;
}

/* Driver program to check above functions */
int main()
{
    int arr[] = {1, 2, 3, 3, 3, 3, 10};
    int n = sizeof(arr)/sizeof(arr[0]);
    int x = 3;
    if (isMajority(arr, n, x))
        printf("%d appears more than %d times in arr[]",
            x, n/2);
    else
        printf("%d does not appear more than %d times in arr[]",
            x, n/2);
    return 0;
}
```

[Run on IDE](#)

Output:

```
3 appears more than 3 times in arr[]
```

Time Complexity: $O(\log n)$

Algorithmic Paradigm: Divide and Conquer

Maximum and minimum of an array using minimum number of comparisons

Write a C function to return minimum and maximum in an array. Your program should make minimum number of comparisons.

We strongly recommend that you click here and practice it, before moving on to the solution.

First of all, how do we return multiple values from a C 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.

```
struct pair
{
    int min;
    int max;
};
```

[Run on IDE](#)

And the function declaration becomes: `struct pair getMinMax(int arr[], int n)` where `arr[]` is the array of size `n` whose minimum and maximum are needed.

METHOD 1 (Simple 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)

```
/* structure is used to return two values from minMax() */
#include<stdio.h>
struct pair
{
    int min;
    int max;
};

struct pair getMinMax(int arr[], int n)
{
    struct pair minmax;
    int i;

    /*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 = 2; i < n; i++)
    {
        if (arr[i] > minmax.max)
            minmax.max = arr[i];

        else if (arr[i] < minmax.min)
            minmax.min = arr[i];
    }

    return minmax;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {1000, 11, 445, 1, 330, 3000};
    int arr_size = 6;
    struct pair minmax = getMinMax (arr, arr_size);
    printf("\nMinimum element is %d", minmax.min);
    printf("\nMaximum element is %d", minmax.max);
    getchar();
}
```

[Run on IDE](#)

Time Complexity: $O(n)$

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

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

METHOD 2 (Tournament Method)

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

```
Pair MaxMin(array, array_size)
```

```
    if array_size = 1
```

```
        return element as both max and min
```

```
    else if array_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
```



Time Complexity: $O(n)$

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

Algorithmic Paradigm: Divide and Conquer

$$T(n) = T(\text{floor}(n/2)) + T(\text{ceil}(n/2)) + 2$$

$$T(2) = 1$$

$$T(1) = 0$$

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

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

After solving above recursion, we get

$$T(n) = 3/2n - 2$$

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



METHOD 3 (Compare in Pairs)

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

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

For rest of the elements, pick them in pairs and compare their maximum and minimum with max and min respectively.



Time Complexity: $O(n)$

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
 $= 1 + 3*(n-2)/2 = 3n/2 - 2$

Second and third approaches make equal number of comparisons when n is a power of 2.

In general, method 3 seems to be the best.



Segregate 0s and 1s in an array

Asked by kapil.

You are given an array of 0s and 1s in random order. Segregate 0s on left side and 1s on right side of the array. Traverse array only once.

```
Input array   = [0, 1, 0, 1, 0, 0, 1, 1, 1, 0]
Output array  = [0, 0, 0, 0, 0, 1, 1, 1, 1, 1]
```

Method 1 (Count 0s or 1s)

Thanks to Naveen for suggesting this method.

- 1) Count the number of 0s. Let count be C.
- 2) Once we have count, we can put C 0s at the beginning and 1s at the remaining $n - C$ positions in array.

Time Complexity: $O(n)$

The method 1 traverses the array two times. Method 2 does the same in a single pass.

Method 2 (Use two indexes to traverse)

Maintain two indexes. Initialize first index *left* as 0 and second index *right* as $n-1$.

Do following while *left* < *right*

- a) Keep incrementing index *left* while there are 0s at it
- b) Keep decrementing index *right* while there are 1s at it
- c) If *left* < *right* then exchange *arr*[*left*] and *arr*[*right*] Implementation.



Maximum difference between two elements such that larger element appears after the smaller number

Given an array `arr[]` of integers, find out the difference between any two elements such that larger element appears after the smaller number in `arr[]`.

Examples: If array is [2, 3, 10, 6, 4, 8, 1] then returned value should be 8 (Diff between 10 and 2). If array is [7, 9, 5, 6, 3, 2] then returned value should be 2 (Diff between 7 and 9)

We strongly recommend that you click [here](#) and practice it, before moving on to the solution.

Method 1 (Simple)

Use two loops. In the outer loop, pick elements one by one and in the inner loop calculate the difference of the picked element with every other element in the array and compare the difference with the maximum difference calculated so far.

```
#include<stdio.h>

/* The function assumes that there are at least two
elements in array.
The function returns a negative value if the array is
sorted in decreasing order.
Returns 0 if elements are equal */
int maxDiff(int arr[], int arr_size)
{
    int max_diff = arr[1] - arr[0];
    int i, j;
    for(i = 0; i < arr_size; i++)
    {
        for(j = i+1; j < arr_size; j++)
        {
            if(arr[j] - arr[i] > max_diff)
                max_diff = arr[j] - arr[i];
        }
    }
    return max_diff;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {1, 2, 90, 10, 110};
    printf("Maximum difference is %d", maxDiff(arr, 5));
    getchar();
    return 0;
}
```

Run on IDE

Time Complexity: $O(n^2)$

Auxiliary Space: $O(1)$



Method 2 (Tricky and Efficient)

In this method, instead of taking difference of the picked element with every other element, we take the difference with the minimum element found so far. So we need to keep track of 2 things:

- 1) Maximum difference found so far (max_diff).
- 2) Minimum number visited so far (min_element).

```
#include<stdio.h>

/* The function assumes that there are at least two
elements in array.
The function returns a negative value if the array is
sorted in decreasing order.
Returns 0 if elements are equal */
int maxDiff(int arr[], int arr_size)
{
    int max_diff = arr[1] - arr[0];
    int min_element = arr[0];
    int i;
    for(i = 1; i < arr_size; i++)
    {
        if (arr[i] - min_element > max_diff)
            max_diff = arr[i] - min_element;
        if (arr[i] < min_element)
            min_element = arr[i];
    }
    return max_diff;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {1, 2, 6, 80, 100};
    int size = sizeof(arr)/sizeof(arr[0]);
    printf("Maximum difference is %d", maxDiff(arr, size));
    getchar();
    return 0;
}
```


Run on IDE

Time Complexity: $O(n)$

Auxiliary Space: $O(1)$

Like min element, we can also keep track of max element from right side. See below code suggested by

A Product Array Puzzle



Given an array `arr[]` of n integers, construct a Product Array `prod[]` (of same size) such that `prod[i]` is equal to the product of all the elements of `arr[]` except `arr[i]`. Solve it without division operator and in $O(n)$.

Example:

`arr[] = {10, 3, 5, 6, 2}`

`prod[] = {180, 600, 360, 300, 900}`

Algorithm:

- 1) Construct a temporary array `left[]` such that `left[i]` contains product of all elements on left of `arr[i]` excluding `arr[i]`.
- 2) Construct another temporary array `right[]` such that `right[i]` contains product of all elements on right of `arr[i]` excluding `arr[i]`.
- 3) To get `prod[]`, multiply `left[]` and `right[]`.

Segregate Even and Odd numbers



Given an array $A[]$, write a function that segregates even and odd numbers. The functions should put all even numbers first, and then odd numbers.

Example

Input = {12, 34, 45, 9, 8, 90, 3}

Output = {12, 34, 8, 90, 45, 9, 3}

In the output, order of numbers can be changed, i.e., in the above example 34 can come before 12 and 3 can come before 9.


The problem is very similar to our old post [Segregate 0s and 1s in an array](#), and both of these problems are variation of famous [Dutch national flag problem](#).

Algorithm: segregateEvenOdd()

1) Initialize two index variables left and right:

left = 0, right = size - 1

- 2) Keep incrementing left index until we see an odd number.
- 3) Keep decrementing right index until we see an even number.
- 4) If left < right then swap arr[left] and arr[right]

imp  Using index values to mark presence of element

Find duplicates in $O(n)$ time and $O(1)$ extra space

Given an array of n elements which contains elements from 0 to $n-1$, with any of these numbers appearing any number of times. Find these repeating numbers in $O(n)$ and using only constant memory space.

For example, let n be 7 and array be {1, 2, 3, 1, 3, 6, 6}, the answer should be 1, 3 and 6.

This problem is an extended version of following problem.

Find the two repeating elements in a given array

Method 1 and Method 2 of the above link are not applicable as the question says $O(n)$ time complexity and $O(1)$ constant space. Also, Method 3 and Method 4 cannot be applied here because there can be more than 2 repeating elements in this problem. Method 5 can be extended to work for this problem. Below is the solution that is similar to the Method 5.

Algorithm:

```
traverse the list for i= 0 to n-1 elements
{
    check for sign of A[abs(A[i])] ;
    if positive then
        make it negative by A[abs(A[i])]=-A[abs(A[i])];
    else // i.e., A[abs(A[i])] is negative
        this element (ith element of list) is a repetition
}
```

Implementation:

```
#include <stdio.h>
#include <stdlib.h>

void printRepeating(int arr[], int size)
{
    int i;
    printf("The repeating elements are: \n");
    for (i = 0; i < size; i++)
    {
        if (arr[abs(arr[i])] >= 0)
            arr[abs(arr[i])] = -arr[abs(arr[i])];
        else
            printf(" %d ", abs(arr[i]));
    }
}

int main()
{
    int arr[] = {1, 2, 3, 1, 3, 6, 6};
    int arr_size = sizeof(arr)/sizeof(arr[0]);
    printRepeating(arr, arr_size);
    getchar();
    return 0;
}
```

Run on IDE

Note: The above program doesn't handle 0 case (If 0 is present in array). The program can be easily modified to handle that also. It is not handled to keep the code simple.

Output:

The repeating elements are:

1 3 6

Time Complexity: $O(n)$

Auxiliary Space: $O(1)$

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:

$A[0] = -7, A[1] = 1, A[2] = 5, A[3] = 2, A[4] = -4, A[5] = 3, A[6] = 0$

3 is an equilibrium index, because:

$$A[0] + A[1] + A[2] = A[4] + A[5] + A[6]$$

6 is also an equilibrium index, because sum of zero elements is zero, i.e., $A[0] + A[1] + A[2] + A[3] + A[4] + A[5] = 0$

7 is not an equilibrium index, because it is not a valid index of array A.

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.

Method 2 (Tricky and Efficient)

The idea is to get total sum of array first. Then iterate through the array and keep updating the left sum which is initialized as zero. In the loop, we can get 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.
 - c) `leftsum = leftsum + arr[i]` // update leftsum for next iteration.
- 4) return -1 // If we come out of loop without returning then
// there is no equilibrium index

Find the smallest missing number

Given a **sorted** array of n integers where each integer is in the range from 0 to $m-1$ and $m > n$. Find the smallest number that is missing from the array.

Examples

Input: $\{0, 1, 2, 6, 9\}$, $n = 5$, $m = 10$

Output: 3

Input: $\{4, 5, 10, 11\}$, $n = 4$, $m = 12$

Output: 0

Input: $\{0, 1, 2, 3\}$, $n = 4$, $m = 5$

Output: 4

Input: $\{0, 1, 2, 3, 4, 5, 6, 7, 10\}$, $n = 9$, $m = 11$

Output: 8

Thanks to [Ravichandra](#) for suggesting following two methods.

Method 1 (Use Binary Search)

For $i = 0$ to $m-1$, do binary search for i in the array. If i is not present in the array then return i .

Time Complexity: $O(m \log n)$

Method 2 (Linear Search)

If $arr[0]$ is not 0 , return 0 . Otherwise traverse the input array starting from index 1 , and for each pair of elements $a[i]$ and $a[i+1]$, find the difference between them. if the difference is greater than 1 then $a[i]+1$ is the missing number.

Time Complexity: $O(n)$

Method 3 (Use Modified Binary Search)

Thanks to yasein and Jams for suggesting this method.

In the standard Binary Search process, the element to be searched is compared with the middle element and on the basis of comparison result, we decide whether to search is over or to go to left half or right half.

In this method, we modify the standard Binary Search algorithm to compare the middle element with its index and make decision on the basis of this comparison.

- ...1) If the first element is not same as its index then return first index
- ...2) Else get the middle index say mid
 -a) If arr[mid] greater than mid then the required element lies in left half
 -b) Else the required element lies in right half

```
#include<stdio.h>
```

```
int findFirstMissing(int array[], int start, int end) {
```

```
    if (start > end)
        return end + 1;
```

All elements are present except the last one

```
    if (start != array[start])
        return start;
```

```
    int mid = (start + end) / 2;
```

```
    if (array[mid] > mid)
        return findFirstMissing(array, start, mid);
    else
        return findFirstMissing(array, mid + 1, end);
```

```
}
```

```
// driver program to test above function
```

```
int main()
```

```
{
```

```
    int arr[] = {0, 1, 2, 3, 4, 5, 6, 7, 10};
```

```
    int n = sizeof(arr)/sizeof(arr[0]);
```

```
    printf(" First missing element is %d",
           findFirstMissing(arr, 0, n-1));
```

```
    getchar();
```

```
    return 0;
```

```
}
```

Run on IDE

Note: This method doesn't work if there are duplicate elements in the array.

Time Complexity: $O(\log n)$



Count the number of occurrences in a sorted array

Given a sorted array `arr[]` and a number `x`, write a function that counts the occurrences of `x` in `arr[]`. Expected time complexity is $O(\log n)$

Examples:

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

Output: 4 // `x` (or 2) occurs 4 times in `arr[]`

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

Output: 1

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

Output: 2

Input: `arr[] = {1, 1, 2, 2, 2, 2, 3,}`, `x = 4`

Output: -1 // 4 doesn't occur in `arr[]`

Method 1 (Linear Search)



Linearly search for `x`, count the occurrences of `x` and return the count.

Time Complexity: $O(n)$

Method 2 (Use Binary Search)



- 1) Use Binary search to get index of the first occurrence of `x` in `arr[]`. Let the index of the first occurrence be `i`.
- 2) Use Binary search to get index of the last occurrence of `x` in `arr[]`. Let the index of the last occurrence be `j`.
- 3) Return `(j - i + 1)`.

Find the minimum distance between two numbers

Given an unsorted array `arr[]` and two numbers `x` and `y`, find the minimum distance between `x` and `y` in `arr[]`. The array might also contain duplicates. You may assume that both `x` and `y` are different and present in `arr[]`.

Examples:

Input: `arr[] = {1, 2}`, `x = 1`, `y = 2`

Output: Minimum distance between 1 and 2 is 1.

Input: `arr[] = {3, 4, 5}`, `x = 3`, `y = 5`

Output: Minimum distance between 3 and 5 is 2.

Input: `arr[] = {3, 5, 4, 2, 6, 5, 6, 6, 5, 4, 8, 3}`, `x = 3`, `y = 6`

Output: Minimum distance between 3 and 6 is 4.

Input: `arr[] = {2, 5, 3, 5, 4, 4, 2, 3}`, `x = 3`, `y = 2`

Output: Minimum distance between 3 and 2 is 1.

Method 1 (Simple)

Use two loops: The outer loop picks all the elements of `arr[]` one by one. The inner loop picks all the elements after the element picked by outer loop. If the elements picked by outer and inner loops have same values as `x` or `y` then if needed update the minimum distance calculated so far.

$O(n^2)$



~~★~~ imp

Method 2 (Tricky)

- 1) Traverse array from left side and stop if either x or y is found. Store index of this first occurrence in a variable say $prev$
- 2) Now traverse $arr[]$ after the index $prev$. If the element at current index i matches with either x or y then check if it is different from $arr[prev]$. If it is different then update the minimum distance if needed. If it is same then update $prev$ i.e., make $prev = i$.

★ imp

Find the repeating and the missing | Added 3 new methods

Given an unsorted array of size n . Array elements are in range from 1 to n . One number from set $\{1, 2, \dots, n\}$ is missing and one number occurs twice in array. Find these two numbers.

Examples:

```
arr[] = {3, 1, 3}
```

```
Output: 2, 3 // 2 is missing and 3 occurs twice
```

```
arr[] = {4, 3, 6, 2, 1, 1}
```

```
Output: 5, 1 // 5 is missing and 1 occurs twice
```

Method 3 (Use elements as Index and mark the visited places)

Traverse the array. While traversing, use absolute value of every element as index and make the value at this index as negative to mark it visited. If something is already marked negative then this is the repeating element. To find missing, traverse the array again and look for a positive value.

```
#include<stdio.h>
#include<stdlib.h>

void printTwoElements(int arr[], int size)
{
    int i;
    printf("\n The repeating element is");

    for(i = 0; i < size; i++)
    {
        if(arr[abs(arr[i])-1] > 0)
            arr[abs(arr[i])-1] = -arr[abs(arr[i])-1];
        else
            printf(" %d ", abs(arr[i]));
    }

    printf("\nand the missing element is ");
    for(i=0; i<size; i++)
    {
        if(arr[i]>0)
            printf("%d",i+1);
    }
}

/* Driver program to test above function */
int main()
{
    int arr[] = {7, 3, 4, 5, 5, 6, 2};
    int n = sizeof(arr)/sizeof(arr[0]);
    printTwoElements(arr, n);
    return 0;
}
```



Run on IDE

Time Complexity: $O(n)$

Method 4 (Make two equations)

Let x be the missing and y be the repeating element.

1) Get sum of all numbers.

Sum of array computed $S = n(n+1)/2 - x + y$

2) Get product of all numbers.

Product of array computed $P = 1*2*3*...*n * y / x$

3) The above two steps give us two equations, we can solve the equations and get the values of x and y .

Time Complexity: $O(n)$



Method 5 (Use XOR)

Let x and y be the desired output elements.

Calculate XOR of all the array elements.

```
xor1 = arr[0]^arr[1]^arr[2].....arr[n-1]
```

XOR the result with all numbers from 1 to n

```
xor1 = xor1^1^2^.....^n
```

In the result $xor1$, all elements would nullify each other except x and y . All the bits that are set in $xor1$ will be set in either x or y . So if we take any set bit (We have chosen the rightmost set bit in code) of $xor1$ and divide the elements of the array in two sets – one set of elements with same bit set and other set with same bit not set. By doing so, we will get x in one set and y in another set. Now if we do XOR of all the elements in first set, we will get x , and by doing same in other set we will get y .

Median in a stream of integers (running integers)

Given that integers are read from a data stream. Find median of elements read so far in efficient way. For simplicity assume there are no duplicates. For example, let us consider the stream 5, 15, 1, 3 ...

```
After reading 1st element of stream - 5 -> median - 5
After reading 2nd element of stream - 5, 15 -> median - 10
After reading 3rd element of stream - 5, 15, 1 -> median - 5
After reading 4th element of stream - 5, 15, 1, 3 -> median - 4, so on...
```

Making it clear, when the input size is odd, we take the middle element of sorted data. If the input size is even, we pick average of middle two elements in sorted stream.

Note that output is *effective median* of integers read from the stream so far. Such an algorithm is called **online algorithm**. Any algorithm that can guarantee output of i -elements after processing i -th element, is said to be **online algorithm**. Let us discuss three solutions for the above problem.

Method 1: Insertion Sort

If we can sort the data as it appears, we can easily locate median element. *Insertion Sort* is one such online algorithm that sorts the data appeared so far. At any instance of sorting, say after sorting i -th element, the first i elements of array are sorted. The insertion sort doesn't depend on future data to sort data input till that point. In other words, insertion sort considers data sorted so far while inserting next element. This is the key part of insertion sort that makes it an online algorithm.

However, insertion sort takes $O(n^2)$ time to sort n elements. Perhaps we can use *binary search* on *insertion sort* to find location of next element in $O(\log n)$ time. Yet, we can't do data movement in $O(\log n)$ time. No matter how efficient the implementation is, it takes polynomial time in case of insertion sort.

Interested reader can try implementation of Method 1.

Method 2: Augmented self balanced binary search tree (AVL, RB, etc...)

At every node of BST, maintain number of elements in the subtree rooted at that node. We can use a node as root of simple binary tree, whose left child is self balancing BST with elements less than root and right child is self balancing BST with elements greater than root. The root element always holds *effective median*.

If left and right subtrees contain same number of elements, root node holds average of left and right subtree root data. Otherwise, root contains same data as the root of subtree which is having more elements. After processing an incoming element, the left and right subtrees (BST) are differed utmost by 1.

Self balancing BST is costly in managing balancing factor of BST. However, they provide sorted data which we don't need. We need median only. The next method make use of Heaps to trace median.

Method 3: Heaps

Similar to balancing BST in Method 2 above, we can use a max heap on left side to represent elements that are less than *effective median*, and a min heap on right side to represent elements that are greater than *effective median*.

After processing an incoming element, the number of elements in heaps differ utmost by 1 element. When both heaps contain same number of elements, we pick average of heaps root data as *effective median*. When the heaps are not balanced, we select *effective median* from the root of heap containing more elements.

Maximum Length Bitonic Subarray

Given an array $A[0 \dots n-1]$ containing n positive integers, a subarray $A[i \dots j]$ is bitonic if there is a k with $i \leq k \leq j$ such that $A[i] \leq A[i+1] \dots \leq A[k] \geq A[k+1] \geq \dots A[j-1] \geq A[j]$. Write a function that takes an array as argument and returns the length of the maximum length bitonic subarray.

Expected time complexity of the solution is $O(n)$

Simple Examples

- 1) $A[] = \{12, 4, 78, 90, 45, 23\}$, the maximum length bitonic subarray is $\{4, 78, 90, 45, 23\}$ which is of length 5.
- 2) $A[] = \{20, 4, 1, 2, 3, 4, 2, 10\}$, the maximum length bitonic subarray is $\{1, 2, 3, 4, 2\}$ which is of length 5.

Extreme Examples

- 1) $A[] = \{10\}$, the single element is bitonic, so output is 1.
- 2) $A[] = \{10, 20, 30, 40\}$, the complete array itself is bitonic, so output is 4.
- 3) $A[] = \{40, 30, 20, 10\}$, the complete array itself is bitonic, so output is 4.

Solution

Let us consider the array $\{12, 4, 78, 90, 45, 23\}$ to understand the solution.

- 1) Construct an auxiliary array $inc[]$ from left to right such that $inc[i]$ contains length of the nondecreasing subarray ending at $arr[i]$.

For $A[] = \{12, 4, 78, 90, 45, 23\}$, $inc[]$ is $\{1, 1, 2, 3, 1, 1\}$

- 2) Construct another array $dec[]$ from right to left such that $dec[i]$ contains length of nonincreasing subarray starting at $arr[i]$.

For $A[] = \{12, 4, 78, 90, 45, 23\}$, $dec[]$ is $\{2, 1, 1, 3, 2, 1\}$.

- 3) Once we have the $inc[]$ and $dec[]$ arrays, all we need to do is find the maximum value of $(inc[i] + dec[i] - 1)$.

For $\{12, 4, 78, 90, 45, 23\}$, the max value of $(inc[i] + dec[i] - 1)$ is 5 for $i = 3$.

Find subarray with given sum

Given an unsorted array of nonnegative integers, find a continuous subarray which adds to a given number.

Examples:

```
Input: arr[] = {1, 4, 20, 3, 10, 5}, sum = 33
```

```
Output: Sum found between indexes 2 and 4
```

```
Input: arr[] = {1, 4, 0, 0, 3, 10, 5}, sum = 7
```

```
Output: Sum found between indexes 1 and 4
```

```
Input: arr[] = {1, 4}, sum = 0
```

```
Output: No subarray found
```

There may be more than one subarrays with sum as the given sum. The following solutions print first such subarray.

Source: Google Interview Question

Method 2 (Efficient)

Initialize a variable `curr_sum` as first element. `curr_sum` indicates the sum of current subarray. Start from the second element and add all elements one by one to the `curr_sum`. If `curr_sum` becomes equal to `sum`, then print the solution. If `curr_sum` exceeds the `sum`, then remove trailing elements while `curr_sum` is greater than `sum`.

Following is C implementation of the above approach.

```
/* An efficient program to print subarray with sum as given sum */
#include<stdio.h>

/* Returns true if there is a subarray of arr[] with sum equal to 'sum'
   otherwise returns false. Also, prints the result */
int subArraySum(int arr[], int n, int sum)
{
    /* Initialize curr_sum as value of first element
       and starting point as 0 */
    int curr_sum = arr[0], start = 0, i;

    /* Add elements one by one to curr_sum and if the curr_sum exceeds the
       sum, then remove starting element */
    for (i = 1; i <= n; i++)
    {
        // If curr_sum exceeds the sum, then remove the starting elements
        while (curr_sum > sum && start < i-1)
        {
            curr_sum = curr_sum - arr[start];
            start++;
        }

        // If curr_sum becomes equal to sum, then return true
        if (curr_sum == sum)
        {
            printf("Sum found between indexes %d and %d", start, i-1);
            return 1;
        }

        // Add this element to curr_sum
        if (i < n)
            curr_sum = curr_sum + arr[i];
    }

    // If we reach here, then no subarray
    printf("No subarray found");
    return 0;
}

// Driver program to test above function
int main()
{
    int arr[] = {15, 2, 4, 8, 9, 5, 10, 23};
    int n = sizeof(arr)/sizeof(arr[0]);
    int sum = 23;
    subArraySum(arr, n, sum);
    return 0;
}
```

Run on IDE

Output:

Sum found between indexes 1 and 4

Time complexity of method 2 looks more than $O(n)$, but if we take a closer look at the program, then we can figure out the time complexity is $O(n)$. We can prove it by counting the number of operations performed on every element of `arr[]` in worst case. There are at most 2 operations performed on every element: (a) the element is added to the `curr_sum` (b) the element is subtracted from `curr_sum`. So the upper bound on number of operations is $2n$ which is $O(n)$.

Dynamic Programming | Set 14 (Maximum Sum Increasing Subsequence)

Given an array of n positive integers. Write a program to find the sum of maximum sum subsequence of the given array such that the integers in the subsequence are sorted in increasing order. For example, if input is {1, 101, 2, 3, 100, 4, 5}, then output should be 106 (1 + 2 + 3 + 100), if the input array is {3, 4, 5, 10}, then output should be 22 (3 + 4 + 5 + 10) and if the input array is {10, 5, 4, 3}, then output should be 10

We strongly recommend that you click [here](#) and practice it, before moving on to the solution.

Solution

This problem is a variation of standard Longest Increasing Subsequence (LIS) problem. We need a slight change in the Dynamic Programming solution of LIS problem. All we need to change is to use sum as a criteria instead of length of increasing subsequence.

Following are C/C++ and Python implementations for Dynamic Programming solution of the problem.

C/C++

Java

Python

```
/* Dynamic Programming implementation of Maximum Sum Increasing
Subsequence (MSIS) problem */
#include<stdio.h>

/* maxSumIS() returns the maximum sum of increasing subsequence
in arr[] of size n */
int maxSumIS( int arr[], int n )
{
    int i, j, max = 0;
    int msis[n];

    /* Initialize msis values for all indexes */
    for ( i = 0; i < n; i++ )
        msis[i] = arr[i];

    /* Compute maximum sum values in bottom up manner */
    for ( i = 1; i < n; i++ )
        for ( j = 0; j < i; j++ )
            if ( arr[i] > arr[j] && msis[i] < msis[j] + arr[i] )
                msis[i] = msis[j] + arr[i];

    /* Pick maximum of all msis values */
    for ( i = 0; i < n; i++ )
        if ( max < msis[i] )
            max = msis[i];

    return max;
}

/* Driver program to test above function */
int main()
{
    int arr[] = {1, 101, 2, 3, 100, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Sum of maximum sum increasing subsequence is %d\n",
        maxSumIS( arr, n ) );
    return 0;
}
```

Run on IDE

Output:

Sum of maximum sum increasing subsequence is 106

Time Complexity: $O(n^2)$

Source: Maximum Sum Increasing Subsequence Problem

Asked in: Amazon Morgan Stanley

Find the smallest positive number missing from an unsorted array

You are given an unsorted array with both positive and negative elements. You have to find the smallest positive number missing from the array in $O(n)$ time using constant extra space. You can modify the original array.

Examples

Input: {2, 3, 7, 6, 8, -1, -10, 15}

Output: 1

Input: { 2, 3, -7, 6, 8, 1, -10, 15 }

Output: 4

Input: {1, 1, 0, -1, -2}

Output: 2

Source: To find the smallest positive no missing from an unsorted array

A naive method to solve this problem is to search all positive integers, starting from 1 in the given array. We may have to search at most $n+1$ numbers in the given array. So this solution takes $O(n^2)$ in worst case.

We can use sorting to solve it in lesser time complexity. We can sort the array in $O(n \log n)$ time. Once the array is sorted, then all we need to do is a linear scan of the array. So this approach takes $O(n \log n + n)$ time which is $O(n \log n)$.

We can also use hashing. We can build a hash table of all positive elements in the given array. Once the hash table is built. We can look in the hash table for all positive integers, starting from 1. As soon as we find a number which is not there in hash table, we return it. This approach may take $O(n)$ time on average, but it requires $O(n)$ extra space.

A $O(n)$ time and $O(1)$ extra space solution:

The idea is similar to this post. We use array elements as index. To mark presence of an element x , we change the value at the index x to negative. But this approach doesn't work if there are non-positive (-ve and 0) numbers. So we segregate positive from negative numbers as first step and then apply the approach.

Following is the two step algorithm.

- 1) Segregate positive numbers from others i.e., move all non-positive numbers to left side. In the following code, segregate() function does this part.
- 2) Now we can ignore non-positive elements and consider only the part of array which contains all positive elements. We traverse the array containing all positive numbers and to mark presence of an element x , we change the sign of value at index x to negative. We traverse the array again and print the first index which has positive value. In the following code, findMissingPositive() function does this part. Note that in findMissingPositive, we have subtracted 1 from the values as indexes start from 0 in C.

Find the two numbers with odd occurrences in an unsorted array

Given an unsorted array that contains even number of occurrences for all numbers except two numbers. Find the two numbers which have odd occurrences in $O(n)$ time complexity and $O(1)$ extra space.

Examples:

Input: {12, 23, 34, 12, 12, 23, 12, 45}
Output: 34 and 45

Input: {4, 4, 100, 5000, 4, 4, 4, 4, 100, 100}
Output: 100 and 5000

Input: {10, 20}
Output: 10 and 20

A naive method to solve this problem is to run two nested loops. The outer loop picks an element and the inner loop counts the number of occurrences of the picked element. If the count of occurrences is odd then print the number. The time complexity of this method is $O(n^2)$.

We can use sorting to get the odd occurring numbers in $O(n \log n)$ time. First sort the numbers using an $O(n \log n)$ sorting algorithm like Merge Sort, Heap Sort.. etc. Once the array is sorted, all we need to do is a linear scan of the array and print the odd occurring number.

We can also use hashing. Create an empty hash table which will have elements and their counts. Pick all elements of input array one by one. Look for the picked element in hash table. If the element is found in hash table, increment its count in table. If the element is not found, then enter it in hash table with count as 1. After all elements are entered in hash table, scan the hash table and print elements with odd count. This approach may take $O(n)$ time on average but it requires $O(n)$ extra space.

A $O(n)$ time and $O(1)$ extra space solution:

The idea is similar to method 2 of [this](#) post. Let the two odd occurring numbers be x and y . We use bitwise XOR to get x and y . The first step is to do XOR of all elements present in array. XOR of all elements gives us XOR of x and y because of the following properties of XOR operation.

- 1) XOR of any number n with itself gives us 0, i.e., $n \wedge n = 0$
- 2) XOR of any number n with 0 gives us n , i.e., $n \wedge 0 = n$
- 3) XOR is cumulative and associative.

XOR

So we have XOR of x and y after the first step. Let the value of XOR be $xor2$. Every set bit in $xor2$ indicates that the corresponding bits in x and y have values different from each other. For example, if $x = 6$ (0110) and y is 15 (1111), then $xor2$ will be (1001), the two set bits in $xor2$ indicate that the corresponding bits in x and y are different. In the second step, we pick a set bit of $xor2$ and divide array elements in two groups. Both x and y will go to different groups. In the following code, the rightmost set bit of $xor2$ is picked as it is easy to get rightmost set bit of a number. If we do XOR of all those elements of array which have the corresponding bit set (or 1), then we get the first odd number. And if we do XOR of all those elements which have the corresponding bit 0, then we get the other odd occurring number. This step works because of the same properties of XOR. All the occurrences of a number will go in same set. XOR of all occurrences of a number which occur even number of times will result in 0 in its set. And the xor of a set will be one of the odd occurring elements.

Dynamic Programming | Set 15 (Longest Bitonic Subsequence)

Given an array `arr[0 ... n-1]` containing `n` positive integers, a **subsequence** of `arr[]` is called Bitonic if it is first increasing, then decreasing. Write a function that takes an array as argument and returns the length of the longest bitonic subsequence.

A sequence, sorted in increasing order is considered Bitonic with the decreasing part as empty. Similarly, decreasing order sequence is considered Bitonic with the increasing part as empty.

Examples:

```
Input arr[] = {1, 11, 2, 10, 4, 5, 2, 1};
```

```
Output: 6 (A Longest Bitonic Subsequence of length 6 is 1, 2, 10, 4, 2, 1)
```

```
Input arr[] = {12, 11, 40, 5, 3, 1}
```

```
Output: 5 (A Longest Bitonic Subsequence of length 5 is 12, 11, 5, 3, 1)
```

```
Input arr[] = {80, 60, 30, 40, 20, 10}
```

```
Output: 5 (A Longest Bitonic Subsequence of length 5 is 80, 60, 30, 20, 10)
```

Source: Microsoft Interview Question

Solution

This problem is a variation of standard Longest Increasing Subsequence (LIS) problem. Let the input array be `arr[]` of length `n`. We need to construct two arrays `lis[]` and `lds[]` using Dynamic Programming solution of LIS problem. `lis[i]` stores the length of the Longest Increasing subsequence ending with `arr[i]`. `lds[i]` stores the length of the longest Decreasing subsequence starting from `arr[i]`. Finally, we need to return the max value of `lis[i] + lds[i] - 1` where `i` is from 0 to `n-1`.

```

/* Dynamic Programming implementation of longest bitonic subsequence problem */
#include<stdio.h>
#include<stdlib.h>

```

```

/* lbs() returns the length of the Longest Bitonic Subsequence in
arr[] of size n. The function mainly creates two temporary arrays
lis[] and lds[] and returns the maximum lis[i] + lds[i] - 1.

```

```

    lis[i] ==> Longest Increasing subsequence ending with arr[i]
    lds[i] ==> Longest decreasing subsequence starting with arr[i]

```

```

*/
int lbs( int arr[], int n )
{

```

```

    int i, j;

```

```

    /* Allocate memory for LIS[] and initialize LIS values as 1 for
    all indexes */

```

```

    int *lis = new int[n];
    for (i = 0; i < n; i++)
        lis[i] = 1;

```

```

    /* Compute LIS values from left to right */

```

```

    for (i = 1; i < n; i++)
        for (j = 0; j < i; j++)
            if (arr[i] > arr[j] && lis[i] < lis[j] + 1)
                lis[i] = lis[j] + 1;

```

```

    /* Allocate memory for lds and initialize LDS values for
    all indexes */

```

```

    int *lds = new int [n];
    for (i = 0; i < n; i++)
        lds[i] = 1;

```

```

    /* Compute LDS values from right to left */

```

```

    for (i = n-2; i >= 0; i--)
        for (j = n-1; j > i; j--)
            if (arr[i] > arr[j] && lds[i] < lds[j] + 1)
                lds[i] = lds[j] + 1;

```

```

    /* Return the maximum value of lis[i] + lds[i] - 1*/

```

```

    int max = lis[0] + lds[0] - 1;
    for (i = 1; i < n; i++)
        if (lis[i] + lds[i] - 1 > max)
            max = lis[i] + lds[i] - 1;

```

```

    return max;
}

```

```

/* Driver program to test above function */

```

```

int main()
{
    int arr[] = {0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5,
                13, 3, 11, 7, 15};
    int n = sizeof(arr)/sizeof(arr[0]);
    printf("Length of LBS is %d\n", lbs( arr, n ));
    return 0;
}

```

Find a sorted subsequence of size 3 in linear time

Given an array of n integers, find the 3 elements such that $a[i] < a[j] < a[k]$ and $i < j < k$ in $O(n)$ time. If there are multiple such triplets, then print any one of them.

Examples:

Input: `arr[] = {12, 11, 10, 5, 6, 2, 30}`

Output: 5, 6, 30

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

Output: 1, 2, 3 OR 1, 2, 4 OR 2, 3, 4

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

Output: No such triplet

Source: Amazon Interview Question

Hint: Use Auxiliary Space

Solution:

- 1) Create an auxiliary array `smaller[0..n-1]`. `smaller[i]` should store the index of a number which is smaller than `arr[i]` and is on left side of `arr[i]`. `smaller[i]` should contain -1 if there is no such element.
- 2) Create another auxiliary array `greater[0..n-1]`. `greater[i]` should store the index of a number which is greater than `arr[i]` and is on right side of `arr[i]`. `greater[i]` should contain -1 if there is no such element.
- 3) Finally traverse both `smaller[]` and `greater[]` and find the index i for which both `smaller[i]` and `greater[i]` are not -1.

Dynamic Programming | Set 18 (Partition problem)

Partition problem is to determine whether a given set can be partitioned into two subsets such that the sum of elements in both subsets is same.

Examples

```
arr[] = {1, 5, 11, 5}
```

Output: true

The array can be partitioned as {1, 5, 5} and {11}

```
arr[] = {1, 5, 3}
```

Output: false

The array cannot be partitioned into equal sum sets.

We strongly recommend that you click [here](#) and practice it, before moving on to the solution.

Following are the two main steps to solve this problem:

- 1) Calculate sum of the array. If sum is odd, there can not be two subsets with equal sum, so return false.
- 2) If sum of array elements is even, calculate $\text{sum}/2$ and find a subset of array with sum equal to $\text{sum}/2$.

The first step is simple. The second step is crucial, it can be solved either using recursion or Dynamic Programming.

Recursive Solution

Following is the recursive property of the second step mentioned above.

Let $\text{isSubsetSum}(\text{arr}, n, \text{sum}/2)$ be the function that returns true if there is a subset of $\text{arr}[0..n-1]$ with sum equal to $\text{sum}/2$

The isSubsetSum problem can be divided into two subproblems

- a) $\text{isSubsetSum}()$ without considering last element (reducing n to $n-1$)
- b) isSubsetSum considering the last element (reducing $\text{sum}/2$ by $\text{arr}[n-1]$ and n to $n-1$)

If any of the above the above subproblems return true, then return true.

```
isSubsetSum (arr, n, sum/2) = isSubsetSum (arr, n-1, sum/2) ||  
                             isSubsetSum (arr, n-1, sum/2 - arr[n-1])
```

Dynamic Programming Solution

The problem can be solved using dynamic programming when the sum of the elements is not too big. We can create a 2D array `part[i][j]` of size $(\text{sum}/2) \times (n+1)$. And we can construct the solution in bottom up manner such that every filled entry has following property

`part[i][j] = true` if a subset of `{arr[0], arr[1], ..arr[j-1]}` has sum equal to `i`, otherwise `false`

C/C++

Java

```
// A Dynamic Programming based C program to partition problem
#include <stdio.h>

// Returns true if arr[] can be partitioned in two subsets of
// equal sum, otherwise false
bool findPartiion (int arr[], int n)
{
    int sum = 0;
    int i, j;

    // Caculcate sun of all elements
    for (i = 0; i < n; i++)
        sum += arr[i];

    if (sum%2 != 0)
        return false;

    bool part[sum/2+1][n+1];

    // initialize top row as true
    for (i = 0; i <= n; i++)
        part[0][i] = true;

    // initialize leftmost column, except part[0][0], as 0
    for (i = 1; i <= sum/2; i++)
        part[i][0] = false;

    // Fill the partition table in bottom up manner
    for (i = 1; i <= sum/2; i++)
    {
        for (j = 1; j <= n; j++)
        {
            part[i][j] = part[i][j-1];
            if (i >= arr[j-1])
                part[i][j] = part[i][j] || part[i - arr[j-1]][j-1];
        }
    }

    /* // uncomment this part to print table
    for (i = 0; i <= sum/2; i++)
    {
        for (j = 0; j <= n; j++)
            printf ("%4d", part[i][j]);
        printf("\n");
    } */

    return part[sum/2][n];
}

// Driver program to test above funtion
int main()
{
    int arr[] = {3, 1, 1, 2, 2, 1};
    int n = sizeof(arr)/sizeof(arr[0]);
    if (findPartiion(arr, n) == true)
        printf("Can be divided into two subsets of equal sum");
    else
        printf("Can not be divided into two subsets of equal sum");
    getchar();
    return 0;
}
```

Run on IDE

Output:

Can be divided into two subsets of equal sum

Maximum Product Subarray

Given an array that contains both positive and negative integers, find the product of the maximum product subarray. Expected Time complexity is $O(n)$ and only $O(1)$ extra space can be used.

Examples:

Input: arr[] = {6, -3, -10, 0, 2}

Output: 180 // The subarray is {6, -3, -10}

Input: arr[] = {-1, -3, -10, 0, 60}

Output: 60 // The subarray is {60}

Input: arr[] = {-2, -3, 0, -2, -40}

Output: 80 // The subarray is {-2, -40}

We strongly recommend that you click [here](#) and practice it, before moving on to the solution.

The following solution assumes that the given input array always has a positive output. The solution works for all cases mentioned above. It doesn't work for arrays like {0, 0, -20, 0}, {0, 0, 0}, etc. The solution can be easily modified to handle this case.

It is similar to Largest Sum Contiguous Subarray problem. The only thing to note here is, maximum product can also be obtained by minimum (negative) product ending with the previous element multiplied by this element. For example, in array {12, 2, -3, -5, -6, -2}, when we are at element -2, the maximum product is multiplication of, minimum product ending with -6 and -2.

```
/* Returns the product of max product subarray.  
Assumes that the given array always has a subarray  
with product more than 1 */
```

```
int maxSubarrayProduct(int arr[], int n)
```

```
{  
    // max positive product ending at the current position  
    int max_ending_here = 1;  
  
    // min negative product ending at the current position  
    int min_ending_here = 1;  
  
    // Initialize overall max product  
    int max_so_far = 1;  
  
    /* Traverse through the array. Following values are  
    maintained after the i'th iteration:  
    max_ending_here is always 1 or some positive product  
        ending with arr[i]  
    min_ending_here is always 1 or some negative product  
        ending with arr[i] */  
    for (int i = 0; i < n; i++)  
    {  
        /* If this element is positive, update max_ending_here.  
        Update min_ending_here only if min_ending_here is  
        negative */  
        if (arr[i] > 0)  
        {  
            max_ending_here = max_ending_here*arr[i];  
            min_ending_here = min (min_ending_here * arr[i], 1);  
        }  
  
        /* If this element is 0, then the maximum product  
        cannot end here, make both max_ending_here and  
        min_ending_here 0  
        Assumption: Output is always greater than or equal  
            to 1. */  
        else if (arr[i] == 0)  
        {  
            max_ending_here = 1;  
            min_ending_here = 1;  
        }  
  
        /* If element is negative. This is tricky  
        max_ending_here can either be 1 or positive.  
        min_ending_here can either be 1 or negative.  
        next min_ending_here will always be prev.  
        max_ending_here * arr[i] next max_ending_here  
        will be 1 if prev min_ending_here is 1, otherwise  
        next max_ending_here will be prev min_ending_here *  
        arr[i] */  
        else  
        {  
            int temp = max_ending_here;  
            max_ending_here = max (min_ending_here * arr[i], 1);  
            min_ending_here = temp * arr[i];  
        }  
  
        // update max_so_far, if needed  
        if (max_so_far < max_ending_here)  
            max_so_far = max_ending_here;  
    }  
    return max_so_far;  
}
```

Find a pair with the given difference

Given an unsorted array and a number n , find if there exists a pair of elements in the array whose difference is n .

Examples:

Input: `arr[] = {5, 20, 3, 2, 50, 80}`, $n = 78$

Output: Pair Found: (2, 80)

Input: `arr[] = {90, 70, 20, 80, 50}`, $n = 45$


Output: No Such Pair

Source: find pair

The simplest method is to run two loops, the outer loop picks the first element (smaller element) and the inner loop looks for the element picked by outer loop plus n . Time complexity of this method is $O(n^2)$.

We can use sorting and Binary Search to improve time complexity to $O(n \log n)$. The first step is to sort the array in ascending order. Once the array is sorted, traverse the array from left to right, and for each element `arr[i]`, binary search for `arr[i] + n` in `arr[i+1..n-1]`. If the element is found, return the pair.

Both first and second steps take $O(n \log n)$. So overall complexity is $O(n \log n)$.



The second step of the above algorithm can be improved to $O(n)$. The first step remain same. The idea for second step is take two index variables i and j . initialize them as 0 and 1 respectively. Now run a linear loop. If `arr[j] - arr[i]` is smaller than n , we need to look for greater `arr[j]`, so increment j . If `arr[j] - arr[i]` is greater than n , we need to look for greater `arr[i]`, so increment i . Thanks to Aashish Barnwal for suggesting this approach.

Find four elements that sum to a given value | Set 2 ($O(n^2 \log n)$ Solution)

Important

Given an array of integers, find all combination of four elements in the array whose sum is equal to a given value X .

For example, if the given array is $\{10, 2, 3, 4, 5, 9, 7, 8\}$ and $X = 23$, then your function should print "3 5 7 8" ($3 + 5 + 7 + 8 = 23$).

Sources: Find Specific Sum and Amazon Interview Question

We have discussed a $O(n^3)$ algorithm in the previous post on this topic. The problem can be solved in $O(n^2 \log n)$ time with the help of auxiliary space.

Thanks to itsnimish for suggesting this method. Following is the detailed process.

Let the input array be $A[]$.

- 1) Create an auxiliary array $aux[]$ and store sum of all possible pairs in $aux[]$. The size of $aux[]$ will be $n*(n-1)/2$ where n is the size of $A[]$.
- 2) Sort the auxiliary array $aux[]$.
- 3) Now the problem reduces to find two elements in $aux[]$ with sum equal to X . We can use method 1 of this post to find the two elements efficiently. There is following important point to note though. An element of $aux[]$ represents a pair from $A[]$. While picking two elements from $aux[]$, we must check whether the two elements have an element of $A[]$ in common. For example, if first element sum of $A[1]$ and $A[2]$, and second element is sum of $A[2]$ and $A[4]$, then these two elements of $aux[]$ don't represent four distinct elements of input array $A[]$.

Imp 

Maximum circular subarray sum ✓

Given n numbers (both +ve and -ve), arranged in a circle, find the maximum sum of consecutive number.

Examples:

Input: $a[] = \{8, -8, 9, -9, 10, -11, 12\}$

Output: 22 ($12 + 8 - 8 + 9 - 9 + 10$)

Input: $a[] = \{10, -3, -4, 7, 6, 5, -4, -1\}$

Output: 23 ($7 + 6 + 5 - 4 - 1 + 10$)

Input: $a[] = \{-1, 40, -14, 7, 6, 5, -4, -1\}$

Output: 52 ($7 + 6 + 5 - 4 - 1 - 1 + 40$)

There can be two cases for the maximum sum:

Case 1: The elements that contribute to the maximum sum are arranged such that no wrapping is there.

Examples: $\{10, 2, -1, 5\}$, $\{-2, 4, -1, 4, -1\}$. In this case, Kadane's algorithm will produce the result.

Case 2: The elements which contribute to the maximum sum are arranged such that wrapping is there.

Examples: $\{10, -12, 11\}$, $\{12, -5, 4, -8, 11\}$. In this case, we change wrapping to non-wrapping. Let us see how.

Wrapping of contributing elements implies non wrapping of non contributing elements, so find out the sum of non contributing elements and subtract this sum from the total sum. To find out the sum of non contributing, invert sign of each element and then run Kadane's algorithm.

Our array is like a ring and we have to eliminate the maximum continuous negative that implies maximum continuous positive in the inverted arrays.

Finally we compare the sum obtained by both cases, and return the maximum of the two sums.

```
// The function returns maximum circular contiguous sum
// in a[]
int maxCircularSum(int a[], int n)
{
    // Case 1: get the maximum sum using standard kadane's
    // algorithm
    int max_kadane = kadane(a, n);

    // Case 2: Now find the maximum sum that includes
    // corner elements.
    int max_wrap = 0, i;
    for (i=0; i<n; i++)
    {
        max_wrap += a[i]; // Calculate array-sum
        a[i] = -a[i]; // invert the array (change sign)
    }

    // max sum with corner elements will be:
    // array-sum - (-max subarray sum of inverted array)
    max_wrap = max_wrap + kadane(a, n);

    // The maximum circular sum will be maximum of two sums
    return (max_wrap > max_kadane)? max_wrap: max_kadane;
}
```

Count the number of possible triangles

Given an unsorted array of positive integers. Find the number of triangles that can be formed with three different array elements as three sides of triangles. For a triangle to be possible from 3 values, the sum of any two values (or sides) must be greater than the third value (or third side).

For example, if the input array is {4, 6, 3, 7}, the output should be 3. There are three triangles possible {3, 4, 6}, {4, 6, 7} and {3, 6, 7}. Note that {3, 4, 7} is not a possible triangle.

As another example, consider the array {10, 21, 22, 100, 101, 200, 300}. There can be 6 possible triangles: {10, 21, 22}, {21, 100, 101}, {22, 100, 101}, {10, 100, 101}, {100, 101, 200} and {101, 200, 300}

Method 1 (Brute force)

The brute force method is to run three loops and keep track of the number of triangles possible so far. The three loops select three different values from array, the innermost loop checks for the triangle property (the sum of any two sides must be greater than the value of third side).

Time Complexity: $O(N^3)$ where N is the size of input array.

Method 2 (Tricky and Efficient)

Let a, b and c be three sides. The below condition must hold for a triangle (Sum of two sides is greater than the third side)

- i) $a + b > c$
- ii) $b + c > a$
- iii) $a + c > b$

Following are steps to count triangle.

1. Sort the array in non-decreasing order.
2. Initialize two pointers 'i' and 'j' to first and second elements respectively and initialize count of triangles as 0.
3. Fix 'i' and 'j' and find the rightmost index 'k' (or largest 'arr[k]') such that 'arr[i] + arr[j] > arr[k]'. The number of triangles that can be formed with 'arr[i]' and 'arr[j]' as two sides is 'k - j'. Add 'k - j' to count of triangles.

Let us consider 'arr[i]' as 'a', 'arr[j]' as b and all elements between 'arr[j+1]' and 'arr[k]' as 'c'. The above mentioned conditions (ii) and (iii) are satisfied because 'arr[i] < arr[j] < arr[k]'. And we check for condition (i) when we pick 'k' 4. Increment 'j' to fix the second element again.

Note that in step 3, we can use the previous value of 'k'. The reason is simple, if we know that the value of 'arr[i] + arr[j-1]' is greater than 'arr[k]', then we can say 'arr[i] + arr[j]' will also be greater than 'arr[k]', because the array is sorted in increasing order.

5. If 'j' has reached end, then increment 'i'. Initialize 'j' as 'i + 1', 'k' as 'i+2' and repeat the steps 3 and 4.

Arrange given numbers to form the biggest number

Given an array of numbers, arrange them in a way that yields the largest value. For example, if the given numbers are {54, 546, 548, 60}, the arrangement 6054854654 gives the largest value. And if the given numbers are {1, 34, 3, 98, 9, 76, 45, 4}, then the arrangement 998764543431 gives the largest value.

We strongly recommend that you click here and practice it, before moving on to the solution.

A simple solution that comes to our mind is to sort all numbers in descending order, but simply sorting doesn't work. For example, 548 is greater than 60, but in output 60 comes before 548. As a second example, 98 is greater than 9, but 9 comes before 98 in output.

So how do we go about it? The idea is to use any comparison based sorting algorithm. In the used sorting algorithm, instead of using the default comparison, write a comparison function `myCompare()` and use it to sort numbers. Given two numbers X and Y, how should `myCompare()` decide which number to put first – we compare two numbers XY (Y appended at the end of X) and YX (X appended at the end of Y). If XY is larger, then X should come before Y in output, else Y should come before. For example, let X and Y be 542 and 60. To compare X and Y, we compare 54260 and 60542. Since 60542 is greater than 54260, we put Y first.

Divide and Conquer | Set 3 (Maximum Subarray Sum)

You are given a one dimensional array that may contain both positive and negative integers, find the sum of contiguous subarray of numbers which has the largest sum.

For example, if the given array is {-2, -5, 6, -2, -3, 1, 5, -6}, then the maximum subarray sum is 7 (see highlighted elements).

The **naive method** is to run two loops. The outer loop picks the beginning element, the inner loop finds the maximum possible sum with first element picked by outer loop and compares this maximum with the overall maximum. Finally return the overall maximum. The time complexity of the Naive method is $O(n^2)$.

Using **Divide and Conquer** approach, we can find the maximum subarray sum in $O(n \log n)$ time. Following is the Divide and Conquer algorithm.

- 1) Divide the given array in two halves
- 2) Return the maximum of following three
 -a) Maximum subarray sum in left half (Make a recursive call)
 -b) Maximum subarray sum in right half (Make a recursive call)
 -c) Maximum subarray sum such that the subarray crosses the midpoint

The lines 2.a and 2.b are simple recursive calls. **How to find maximum subarray sum such that the subarray crosses the midpoint?** We can easily find the crossing sum in linear time. The idea is simple, find the maximum sum starting from mid point and ending at some point on left of mid, then find the maximum sum starting from mid + 1 and ending with sum point on right of mid + 1. Finally, combine the two and return.



Find a peak element

Given an array of integers. Find a peak element in it. An array element is peak if it is NOT smaller than its neighbors. For corner elements, we need to consider only one neighbor. For example, for input array {5, 10, 20, 15}, 20 is the only peak element. For input array {10, 20, 15, 2, 23, 90, 67}, there are two peak elements: 20 and 90. Note that we need to return any one peak element.

Following corner cases give better idea about the problem.

- 1) If input array is sorted in strictly increasing order the last element is always a peak element. For example, 50 is peak element in {10, 20, 30, 40, 50}.
- 2) If input array is sorted in strictly decreasing order the first element is always a peak element. 100 is the peak element in {100, 80, 60, 50, 20}.
- 3) If all elements of input array are same, every element is a peak element.

It is clear from above examples that there is always a peak element in input array in any input array.

A simple solution is to do a linear scan of array and as soon as we find a peak element, we return it. The worst case time complexity of this method would be $O(n)$.

Can we find a peak element in worst time complexity better than $O(n)$?

We can use Divide and Conquer to find a peak in $O(\log n)$ time. The idea is Binary Search based, we compare middle element with its neighbors. If middle element is not smaller than any of its neighbors, then we return it. If the middle element is smaller than the its left neighbor, then there is always a peak in left half (Why? take few examples). If the middle element is smaller than the its right neighbor, then there is always a peak in right half (due to same reason as left half). Following are C and Java implementations of this approach.



Unbounded Binary Search Example (Find the point where a monotonically increasing function becomes positive first time) ✓

Given a function 'int f(unsigned int x)' which takes a **non-negative integer** 'x' as input and returns an **integer** as output. The function is monotonically increasing with respect to value of x, i.e., the value of $f(x+1)$ is greater than $f(x)$ for every input x. Find the value 'n' where $f()$ becomes positive for the first time. Since $f()$ is monotonically increasing, values of $f(n+1)$, $f(n+2)$, ... must be positive and values of $f(n-2)$, $f(n-3)$, ... must be negative. Find n in $O(\log n)$ time, you may assume that $f(x)$ can be evaluated in $O(1)$ time for any input x.

A **simple solution** is to start from i equals to 0 and one by one calculate value of $f(i)$ for 1, 2, 3, 4 .. etc until we find a positive $f(i)$. This works, but takes **$O(n)$ time**.

Can we apply **Binary Search to find n in $O(\log n)$ time?** We **can't directly** apply Binary Search as **we don't have an upper limit or high index**. The **idea is to do repeated doubling until we find a positive value**, i.e., **check values of $f()$ for following values until $f(i)$ becomes positive.**

```
f(0)
f(1)
f(2)
f(4)
f(8)
f(16)
f(32)
....
....
f(high)
```

Let 'high' be the value of i when $f()$ becomes positive for first time.

Can we apply Binary Search to find n after finding 'high'? **We can apply Binary Search now, we can use 'high/2' as low and 'high' as high indexes in binary search**. The **result n must lie between 'high/2' and 'high'**.

Number of steps for finding 'high' is $O(\log n)$. So we can find 'high' in $O(\log n)$ time. What about time taken by Binary Search between high/2 and high? The value of 'high' must be less than 2^n . The number of elements between high/2 and high must be $O(n)$. Therefore, time complexity of Binary Search is $O(\log n)$ and overall time complexity is $2 * O(\log n)$ which is **$O(\log n)$** .

// O(n) solution for finding smallest subarray with sum
// greater than x
#include <iostream>
using namespace std;

```
// Returns length of smallest subarray with sum greater than x.  
// If there is no subarray with given sum, then returns n+1  
int smallestSubWithSum(int arr[], int n, int x)  
{  
    // Initialize current sum and minimum length  
    int curr_sum = 0, min_len = n+1;  
  
    // Initialize starting and ending indexes  
    int start = 0, end = 0;  
    while (end < n)  
    {  
        // Keep adding array elements while current sum  
        // is smaller than x  
        while (curr_sum <= x && end < n)  
            curr_sum += arr[end++];  
  
        // If current sum becomes greater than x.  
        while (curr_sum > x && start < n)  
        {  
            // Update minimum length if needed  
            if (end - start < min_len)  
                min_len = end - start;  
  
            // remove starting elements  
            curr_sum -= arr[start++];  
        }  
    }  
    return min_len;  
}
```

```
/* Driver program to test above function */  
int main()  
{  
    int arr1[] = {1, 4, 45, 6, 10, 19};  
    int x = 51;  
    int n1 = sizeof(arr1)/sizeof(arr1[0]);  
    cout << smallestSubWithSum(arr1, n1, x) << endl;  
  
    int arr2[] = {1, 10, 5, 2, 7};  
    int n2 = sizeof(arr2)/sizeof(arr2[0]);  
    x = 9;  
    cout << smallestSubWithSum(arr2, n2, x) << endl;  
  
    int arr3[] = {1, 11, 100, 1, 0, 200, 3, 2, 1, 250};  
    int n3 = sizeof(arr3)/sizeof(arr3[0]);  
    x = 280;  
    cout << smallestSubWithSum(arr3, n3, x);  
  
    return 0;  
}
```



Sort an array according to the order defined by another array

Given two arrays A1[] and A2[], sort A1 in such a way that the relative order among the elements will be same as those are in A2. For the elements not present in A2, append them at last in sorted order.

```
Input: A1[] = {2, 1, 2, 5, 7, 1, 9, 3, 6, 8, 8}
       A2[] = {2, 1, 8, 3}
Output: A1[] = {2, 2, 1, 1, 8, 8, 3, 5, 6, 7, 9}
```

The code should handle all cases like number of elements in A2[] may be more or less compared to A1[]. A2[] may have some elements which may not be there in A1[] and vice versa is also possible.

Source: Amazon Interview | Set 110 (On-Campus)

We strongly recommend that you click here and practice it, before moving on to the solution.

Method 1 (Using Sorting and Binary Search)

Let size of A1[] be m and size of A2[] be n.

- 1) Create a temporary array temp of size m and copy contents of A1[] to it.
- 2) Create another array visited[] and initialize all entries in it as false. visited[] is used to mark those elements in temp[] which are copied to A1[].
- 3) Sort temp[]
- 4) Initialize the output index ind as 0.
- 5) Do following for every element of A2[i] in A2[]
 -a) Binary search for all occurrences of A2[i] in temp[], if present then copy all occurrences to A1[ind] and increment ind. Also mark the copied elements visited[]
- 6) Copy all unvisited elements from temp[] to A1[].

Time complexity: The steps 1 and 2 require $O(m)$ time. Step 3 requires $O(m \log m)$ time. Step 5 requires $O(n \log m)$ time. Therefore overall time complexity is $O(m + n \log m)$.

Method 4 (By Writing a Customized Compare Method)

We can also customize compare method of a sorting algorithm to solve the above problem. For example `qsort()` in C allows us to pass our own customized compare method.

1. If `num1` and `num2` both are in `A2` then number with lower index in `A2` will be treated smaller than other.
2. If only one of `num1` or `num2` present in `A2`, then that number will be treated smaller than the other which doesn't present in `A2`.
3. If both are not in `A2`, then natural ordering will be taken.

Time complexity of this method is $O(mn \log m)$ if we use a $O(n \log n)$ time complexity sorting algorithm. We can improve time complexity to $O(m \log m)$ by using a Hashing instead of doing linear search.

Maximum Sum Path in Two Arrays

Given two sorted arrays such the arrays may have some common elements. Find the sum of the maximum sum path to reach from beginning of any array to end of any of the two arrays. We can switch from one array to another array only at common elements.

Expected time complexity is $O(m+n)$ where m is the number of elements in $ar1[]$ and n is the number of elements in $ar2[]$.

Examples:

Input: $ar1[] = \{2, 3, 7, 10, 12\}$, $ar2[] = \{1, 5, 7, 8\}$

Output: 35

35 is sum of $1 + 5 + 7 + 10 + 12$.

We start from first element of $ar2$ which is 1, then we move to 5, then 7. From 7, we switch to $ar1$ (7 is common) and traverse 10 and 12.

Input: $ar1[] = \{10, 12\}$, $ar2 = \{5, 7, 9\}$

Output: 22

22 is sum of 10 and 12.

Since there is no common element, we need to take all elements from the array with more sum.

Input: $ar1[] = \{2, 3, 7, 10, 12, 15, 30, 34\}$

$ar2[] = \{1, 5, 7, 8, 10, 15, 16, 19\}$

Output: 122

122 is sum of 1, 5, 7, 8, 10, 12, 15, 30, 34

We strongly recommend to minimize the browser and try this yourself first.

The idea is to do something similar to merge process of merge sort. We need to calculate sums of elements between all common points for both arrays. Whenever we see a common point, we compare the two sums and add the maximum of two to the result. Following are detailed steps.

1) Initialize result as 0. Also initialize two variables $sum1$ and $sum2$ as 0. Here $sum1$ and $sum2$ are used to store sum of element in $ar1[]$ and $ar2[]$ respectively. These sums are between two common points.

2) Now run a loop to traverse elements of both arrays. While traversing compare current elements of $ar1[]$ and $ar2[]$.

2.a) If current element of $ar1[]$ is smaller than current element of $ar2[]$, then update $sum1$, else if current element of $ar2[]$ is smaller, then update $sum2$.

2.b) If current element of $ar1[]$ and $ar2[]$ are same, then take the maximum of $sum1$ and $sum2$ and add it to the result. Also add the common element to the result.

Find the largest pair sum in an unsorted array ✓

Given an unsorted of distinct integers, find the largest pair sum in it. For example, the largest pair sum in {12, 34, 10, 6, 40} is 74.

Difficulty Level: Rookie

Expected Time Complexity: $O(n)$ [Only one traversal of array is allowed]

We strongly recommend to minimize your browser and try this yourself first.

This problem mainly boils down to finding the largest and second largest element in array. We can find the largest and second largest in $O(n)$ time by traversing array once.

- 1) Initialize both first and second largest
first = max(arr[0], arr[1])
second = min(arr[0], arr[1])
- 2) Loop through remaining elements (from 3rd to end)
 - a) If the current element is greater than first, then update first and second.
 - b) Else if the current element is greater than second then update second
- 3) Return (first + second)

To note



Online algorithm for checking palindrome in a stream

Given a stream of characters (characters are received one by one), write a function that prints 'Yes' if a character makes the complete string palindrome, else prints 'No'.

Examples:

```
Input: str[] = "abcba"
Output: a Yes // "a" is palindrome
        b No  // "ab" is not palindrome
        c No  // "abc" is not palindrome
        b No  // "abcb" is not palindrome
        a Yes // "abcba" is palindrome

Input: str[] = "aabaacaabaa"
Output: a Yes // "a" is palindrome
        a Yes // "aa" is palindrome
        b No  // "aab" is not palindrome
        a No  // "aaba" is not palindrome
        a Yes // "aaba" is not palindrome
        c No  // "aabaac" is not palindrome
        a No  // "aabaaca" is not palindrome
        a No  // "aabaacaa" is not palindrome
        b No  // "aabaacaab" is not palindrome
        a No  // "aabaacaaba" is not palindrome
        a Yes // "aabaacaabaa" is palindrome
```

Let input string be str[0..n-1]. A **Simple Solution** is to do following for every character str[i] in input string. Check if substring str[0...i] is palindrome, then print yes, else print no.

A **Better Solution** is to use the idea of Rolling Hash used in **Rabin Karp algorithm**. The idea is to keep track of reverse of first half and second half (we also use first half and reverse of second half) for every index. Below is complete algorithm.

- 1) The first character is always a palindrome, so print yes for first character.
- 2) Initialize reverse of first half as "a" and second half as "b". Let the hash value of first half reverse be 'first' and that of second half be 'second'.
- 3) Iterate through string starting from second character, do following for every character str[i], i.e., i varies from 1 to n-1.
 - a) If 'first' and 'second' are same, then character by character check the substring ending with current character and print "Yes" if palindrome.
Note that if hash values match, then strings need not be same. For example, hash values of "ab" and "ba" are same, but strings are different. That is why we check complete string after hash.
 - b) Update 'first' and 'second' for next iteration.
 - If 'i' is even, then add next character to the beginning of 'first' and end of second half and update hash values.
 - If 'i' is odd, then keep 'first' as it is, remove leading character from second and append a next character at end.



Maximum profit by buying and selling a share at most twice

In a daily share trading, a buyer buys shares in the morning and sells it on same day. If the trader is allowed to make at most 2 transactions in a day, where as second transaction can only start after first one is complete (Sell->buy->sell->buy). Given stock prices throughout day, find out maximum profit that a share trader could have made.

Examples:

```
Input:  price[] = {10, 22, 5, 75, 65, 80}
Output: 87
Trader earns 87 as sum of 12 and 75
Buy at price 10, sell at 22, buy at 5 and sell at 80
```

```
Input:  price[] = {2, 30, 15, 10, 8, 25, 80}
Output: 100
Trader earns 100 as sum of 28 and 72
Buy at price 2, sell at 30, buy at 8 and sell at 80
```

```
Input:  price[] = {100, 30, 15, 10, 8, 25, 80};
Output: 72
Buy at price 8 and sell at 80.
```

```
Input:  price[] = {90, 80, 70, 60, 50}
Output: 0
Not possible to earn.
```

We strongly recommend to minimize your browser and try this yourself first.

A **Simple Solution** is to to consider every index 'i' and do following

```
Max profit with at most two transactions =
    MAX {max profit with one transaction and subarray price[0..i] +
         max profit with one transaction and subarray price[i+1..n-1] }
i varies from 0 to n-1.
```

Maximum possible using one transaction can be calculated using following O(n) algorithm

Maximum difference between two elements such that larger element appears after the smaller number

We can do this $O(n)$ using following **Efficient Solution**. The idea is to store maximum possible profit of every subarray and solve the problem in following two phases.

- 1) Create a table `profit[0..n-1]` and initialize all values in it 0.
- 2) Traverse `price[]` from right to left and update `profit[i]` such that `profit[i]` stores maximum profit achievable from one transaction in subarray `price[i..n-1]`
- 3) Traverse `price[]` from left to right and update `profit[i]` such that `profit[i]` stores maximum profit such that `profit[i]` contains maximum achievable profit from two transactions in subarray `price[0..i]`.
- 4) Return `profit[n-1]`

To do step 1, we need to keep track of maximum price from right to left side and to do step 2, we need to keep track of minimum price from left to right. Why we traverse in reverse directions? The idea is to save space, in second step, we use same array for both purposes, maximum with 1 transaction and maximum with 2 transactions. After an iteration i , the array `profit[0..i]` contains maximum profit with 2 transactions and `profit[i+1..n-1]` contains profit with two transactions.


```

// Returns maximum profit with two transactions on a given
// list of stock prices, price[0..n-1]
int maxProfit(int price[], int n)
{
    // Create profit array and initialize it as 0
    int *profit = new int[n];
    for (int i=0; i<n; i++)
        profit[i] = 0;

    /* Get the maximum profit with only one transaction
       allowed. After this loop, profit[i] contains maximum
       profit from price[i..n-1] using at most one trans. */
    int max_price = price[n-1];
    for (int i=n-2; i>=0; i--)
    {
        // max_price has maximum of price[i..n-1]
        if (price[i] > max_price)
            max_price = price[i];

        // we can get profit[i] by taking maximum of:
        // a) previous maximum, i.e., profit[i+1]
        // b) profit by buying at price[i] and selling at
        //     max_price
        profit[i] = max(profit[i+1], max_price-price[i]);
    }

    /* Get the maximum profit with two transactions allowed
       After this loop, profit[n-1] contains the result */
    int min_price = price[0];
    for (int i=1; i<n; i++)
    {
        // min_price is minimum price in price[0..i]
        if (price[i] < min_price)
            min_price = price[i];

        // Maximum profit is maximum of:
        // a) previous maximum, i.e., profit[i-1]
        // b) (Buy, Sell) at (min_price, price[i]) and add
        //     profit of other trans. stored in profit[i]
        profit[i] = max(profit[i-1], profit[i] +
                        (price[i]-min_price) );
    }
    int result = profit[n-1];

    delete [] profit; // To avoid memory leak

    return result;
}

```



Minimum number of swaps required for arranging pairs adjacent to each other

There are n -pairs and therefore $2n$ people. everyone has one unique number ranging from 1 to $2n$. All these $2n$ persons are arranged in random fashion in an Array of size $2n$. We are also given who is partner of whom. Find the minimum number of swaps required to arrange these pairs such that all pairs become adjacent to each other.

Example:

```
Input:
n = 3
pairs[] = {1->3, 2->6, 4->5} // 1 is partner of 3 and so on
arr[] = {3, 5, 6, 4, 1, 2}
```

```
Output: 2
We can get {3, 1, 5, 4, 6, 2} by swapping 5 & 6, and 6 & 1
```

Source: Google Interview Question

We strongly recommend you to minimize your browser and try this yourself first.

The idea is to start from first and second elements and recur for remaining elements. Below are detailed steps/

- 1) If first and second elements are pair, then simply recur for remaining $n-1$ pairs and return the value returned by recursive call.
- 2) If first and second are NOT pair, then there are two ways to arrange. So try both of them return the minimum of two.
 - a) Swap second with pair of first and recur for $n-1$ elements. Let the value returned by recursive call be 'a'.
 - b) Revert the changes made by previous step.
 - c) Swap first with pair of second and recur for $n-1$ elements. Let the value returned by recursive call be 'b'.
 - d) Revert the changes made by previous step before returning control to parent call.
 - e) Return $1 + \min(a, b)$

Convert array into Zig-Zag fashion

Given an array of distinct elements, rearrange the elements of array in zig-zag fashion in $O(n)$ time. The converted array should be in form $a < b > c < d > e < f$.

Example:

Input: `arr[] = {4, 3, 7, 8, 6, 2, 1}`

Output: `arr[] = {3, 7, 4, 8, 2, 6, 1}`

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

Output: `arr[] = {1, 4, 2, 3}`

We strongly recommend you to minimize your browser and try this yourself first.

A **Simple Solution** is to first sort the array. After sorting, exclude the first element, swap the remaining elements in pairs. (i.e. keep `arr[0]` as it is, swap `arr[1]` and `arr[2]`, swap `arr[3]` and `arr[4]`, and so on). Time complexity is $O(n \log n)$ since we need to sort the array first.

We can convert in $O(n)$ time using an **Efficient Approach**. The idea is to use modified one pass of bubble sort. Maintain a flag for representing which order (i.e. $<$ or $>$) currently we need. If the current two elements are not in that order then swap those elements otherwise not.

Let us see the main logic using three consecutive elements A, B, C. Suppose we are processing B and C currently and the current relation is ' $<$ '. But we have $B > C$. Since current relation is ' $<$ ' previous relation must be ' $>$ ' i.e., A must be greater than B. So, the relation is $A > B$ and $B > C$. We can deduce $A > C$. So if we swap B and C then the relation is $A > C$ and $C < B$. Finally we get the desired order **A C B**.

Find maximum value of $\text{Sum}(i * \text{arr}[i])$ with only rotations on given array allowed

Given an array, only rotation operation is allowed on array. We can rotate the array as many times as we want. Return the maximum possible of summation of $i * \text{arr}[i]$.

Example:

Input: $\text{arr}[] = \{1, 20, 2, 10\}$

Output: 72

We can 72 by rotating array twice.

$\{2, 10, 1, 20\}$

$$20 * 3 + 1 * 2 + 10 * 1 + 2 * 0 = 72$$

Input: $\text{arr}[] = \{10, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$;

Output: 330

We can 330 by rotating array 9 times.

$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$;

$$0 * 1 + 1 * 2 + 2 * 3 \dots 9 * 10 = 330$$

We strongly recommend you to minimize your browser and try this yourself first.

A **Simple Solution** is to find all rotations one by one, check sum of every rotation and return the maximum sum. Time complexity of this solution is $O(n^2)$.

We can solve this problem in $O(n)$ time using an **Efficient Solution**.

Let R_j be value of $i * \text{arr}[i]$ with j rotations. The idea is to calculate next rotation value from previous rotation, i.e., calculate R_j from R_{j-1} . We can calculate initial value of result as R_0 , then keep calculating next rotation values.

How to efficiently calculate R_j from R_{j-1} ?

This can be done in $O(1)$ time. Below are details.

Let us calculate initial value of $i * \text{arr}[i]$ with no rotation

$$R_0 = 0 * \text{arr}[0] + 1 * \text{arr}[1] + \dots + (n-1) * \text{arr}[n-1]$$

After 1 rotation $\text{arr}[n-1]$, becomes first element of array, $\text{arr}[0]$ becomes second element, $\text{arr}[1]$ becomes third element and so on.

$$R_1 = 0 * \text{arr}[n-1] + 1 * \text{arr}[0] + \dots + (n-1) * \text{arr}[n-2]$$

$$R_1 - R_0 = \text{arr}[0] + \text{arr}[1] + \dots + \text{arr}[n-2] - (n-1) * \text{arr}[n-1]$$

After 2 rotations $\text{arr}[n-2]$, becomes first element of array, $\text{arr}[n-1]$ becomes second element, $\text{arr}[0]$ becomes third element and so on.

$$R_2 = 0 * \text{arr}[n-2] + 1 * \text{arr}[n-1] + \dots + (n-1) * \text{arr}[n-3]$$

$$R_2 - R_1 = \text{arr}[0] + \text{arr}[1] + \dots + \text{arr}[n-3] - (n-1) * \text{arr}[n-2] + \text{arr}[n-1]$$

If we take a closer look at above values, we can observe below pattern

$$R_j - R_{j-1} = \text{arrSum} - n * \text{arr}[n-j]$$

Where arrSum is sum of all array elements, i.e.,

$$\text{arrSum} = \sum_{i=0}^{n-1} \text{arr}[i]$$



left & right

approach

Count Inversions of size three in a give array

Given an array `arr[]` of size `n`. Three elements `arr[i]`, `arr[j]` and `arr[k]` form an inversion of size 3 if $a[i] > a[j] > a[k]$ and $i < j < k$. Find total number of inversions of size 3.

Example:

Input: {8, 4, 2, 1}

Output: 4

The four inversions are (8,4,2), (8,4,1), (4,2,1) and (8,2,1).

Input: {9, 6, 4, 5, 8}

Output: 2

The two inversions are {9, 6, 4} and {9, 6, 5}

Form minimum number from given sequence

Given a pattern containing only I's and D's. I for increasing and D for decreasing. Devise an algorithm to print the minimum number following that pattern. Digits from 1-9 and digits can't repeat.

Examples:

Input: D	Output: 21
Input: I	Output: 12
Input: DD	Output: 321
Input: II	Output: 123
Input: DIDI	Output: 21435
Input: IIDDD	Output: 126543
Input: DDIDDIID	Output: 321654798

Source: Amazon Interview Question



Rearrange an array in maximum minimum form

Given a sorted array of positive integers, rearrange the array alternately i.e first element should be maximum value, second minimum value, third second max, fourth second min and so on.

Examples:

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

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

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

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

Expected time complexity is $O(n)$.

We strongly recommend that you click here and practice it, before moving on to the solution.

This problem becomes very easy if extra space is allowed. We can copy the array to another array and then place elements at alternate positions.

How to do in-place? The idea is to use the fact that numbers are positive. One by one place numbers at their correct positions and mark them negative.

How to find correct or output position for an element arr[i]? we can observe that the output follows below pattern for an input array.

```
// Output index j for an element arr[i]
```

```
If (i < n/2)
```

```
    j = 2*i + 1
```

```
Else
```

```
    j = 2*(n-1-i);
```

Method 2 (Efficient: $O(n \log n)$)

Find minimum difference pair in the array

The idea is to use sorting. Below are steps.

- 1) Sort array in ascending order. This step takes $O(n \log n)$ time.
- 2) Initialize difference as infinite. This step takes $O(1)$ time.
- 3) Compare all adjacent pairs in sorted array and keep track of minimum difference. This step takes $O(n)$ time.

Sum

A better solution is possible in $O(n)$ time.

Below is the Algorithm.

1. Create a map to store frequency of each number in the array. (Single traversal is required)
2. In the next traversal, for every element check if it can be combined with any other element (other than itself!) to give the desired sum. Increment the counter accordingly.
3. After completion of second traversal, we'd have twice the required value stored in counter because every pair is counted two times. Hence divide count by 2 and return.



Count minimum steps to get the given desired array

Consider an array with n elements and value of all the elements is zero. We can perform following operations on the array.

1. **Incremental operations:** Choose 1 element from the array and increment its value by 1.
2. **Doubling operation:** Double the values of all the elements of array.

We are given desired array `target[]` containing n elements. Compute and return the smallest possible number of the operations needed to change the array from all zeros to desired array.

Sample test cases:

```
Input: target[] = {2, 3}
Output: 4
To get the target array from {0, 0}, we
first increment both elements by 1 (2
operations), then double the array (1
operation). Finally increment second
element (1 more operation)

Input: target[] = {2, 1}
Output: 3
One of the optimal solution is to apply the
incremental operation 2 times to first and
once on second element.

Input: target[] = {16, 16, 16}
Output: 7
The output solution looks as follows. First
apply an incremental operation to each element.
Then apply the doubling operation four times.
Total number of operations is 3+4 = 7
```

Source: <http://qa.geeksforgeeks.org/7023/create-desired-array-from-zero-array>

We strongly recommend you to minimize your browser and try this yourself first.

One important thing to note is that the task is to count the number of steps to get the given target array (not to convert zero array to target array).

The idea is to follow reverse steps i.e. to convert target to array of zeros. Below are steps.

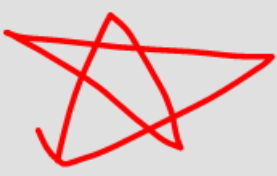
```
Take the target array first.

Initialize result as 0.

If all are even, divide all elements by 2
and increment result by 1.

Find all odd elements, make them even by
reducing them by 1, and for every reduction,
increment result by 1.

Finally we get all zeros in target array.
```





Find minimum number of merge operations to make an array palindrome

Given an array of positive integers. We need to make the given array a 'Palindrome'. Only allowed operation on array is merge. Merging two adjacent elements means replacing them with their sum. The task is to find minimum number of merge operations required to make given array a 'Palindrome'.

To make an array a palindromic we can simply apply merging operations $n-1$ times where n is the size of array (Note a single element array is always palindrome similar to single character string). In that case, size of array will be reduced to 1. But in this problem we are asked to do it in minimum number of operations.

Example:

Input : `arr[] = {15, 4, 15}`

Output : 0

Array is already a palindrome. So we do not need any merge operation.

Input : `arr[] = {1, 4, 5, 1}`

Output : 1

We can make given array palindrome with minimum one merging (merging 4 and 5 to make 9)

Input : `arr[] = {11, 14, 15, 99}`

Output : 3

We need to merge all elements to make a palindrome.

Expected time complexity is $O(n)$.

We strongly recommend that you click here and practice it, before moving on to the solution.

Let $f(i, j)$ be minimum merging operations to make subarray `arr[i..j]` a palindrome. If $i == j$ answer is 0. We start i from 0 and j from $n-1$.

- If `arr[i] == arr[j]`, then there is no need to do any merging operations at index i or index j . Our answer in this case will be $f(i+1, j-1)$.
- Else, we need to do merging operations. Following cases arise.
 - If `arr[i] > arr[j]`, then we should do merging operation at index j . We merge index $j-1$ and j , and update `arr[j-1] = arr[j-1] + arr[j]`. Our answer in this case will be $1 + f(i, j-1)$.
 - For the case when `arr[i] < arr[j]`, update `arr[i+1] = arr[i+1] + arr[i]`. Our answer in this case will be $1 + f(i+1, j)$.
- Our answer will be $f(0, n-1)$, where n is size of array `arr[]`.

Minimize the maximum difference between the heights

Given heights of n towers and a value k . We need to either increase or decrease height of every tower by k (only once) where $k > 0$. The task is to minimize the difference between the heights of the longest and the shortest tower after modifications, and output this difference.

Examples:

Input : arr[] = {1, 15, 10}, $k = 6$
Output : arr[] = {7, 9, 4}
Maximum difference is 5.
Explanation : We change 1 to 6, 15 to 9 and 10 to 4. Maximum difference is 5 (between 4 and 9). We can't get a lower difference.

Input : arr[] = {1, 5, 15, 10}
 $k = 3$
Output : arr[] = {4, 8, 12, 7}
Maximum difference is 8

Input : arr[] = {4, 6}
 $k = 10$
Output : arr[] = {14, 16} OR {-6, -4}
Maximum difference is 2

Input : arr[] = {6, 10}
 $k = 3$
Output : arr[] = {9, 7}
Maximum difference is 2

Input : arr[] = {1, 10, 14, 14, 14, 15}
 $k = 6$
Output: arr[] = {7, 4, 8, 8, 8, 9}
Maximum difference is 5

Input : arr[] = {1, 2, 3}
 $k = 2$
Output: arr[] = {3, 4, 5}
Maximum difference is 2

Source : Adobe Interview Experience | Set 24 (On-Campus for MTS)

The idea is to sort all elements increasing order. Below are steps.

- Sort array in increasing order
- Initialize maximum and minimum elements.
 $max_e = arr[n-1]$, $min_e = arr[0]$
- If k is more than difference between maximum and minimum, add/subtract k to all elements as k cannot decrease the difference. Example {8, 4}, $k = 10$.
- In sorted array, update first and last elements.
 $arr[0] += k$; // $arr[0]$ is minimum and k is +ve
 $arr[n-1] -= k$; // $arr[n-1]$ is maximum and k is -ve
- Initialize max and min of modified array (only two elements have been finalized)
 $new_max = \max(arr[0], arr[n-1])$, $new_min = \min(arr[0], arr[n-1])$
- Finalize middle $n-2$ elements. Do following for every element $arr[j]$ where j lies from 1 to $n-2$.
 - If current element is less than min of modified array, add k .
 - Else If current element is more than max of modified array, subtract k .
 - $arr[j]$ is between new_min and new_max .
 - If $arr[j]$ is closer to new_max , subtract k .
 - Else add k to $arr[j]$.
- Update new_max and new_min if required
 $new_max = \max(arr[j], new_max)$, $new_min = \min(arr[j], new_min)$
- Returns difference between new_max and new_min
 $\text{return } (new_max - new_min);$