Rearrange an array such that ‘arr[j]’ becomes ‘i’ if ‘arr[i]’ is ‘j’

Given an array of size n where all elements are in range from 0 to n-1, change contents of arr[] so that arr[i] = j is changed to arr[j] = i.

Examples:

```
Example 1:
Input: arr[]  = {1, 3, 0, 2};
Output: arr[] = {2, 0, 3, 1};
Explanation for the above output.
Since arr[0] is 1, arr[1] is changed to 0
Since arr[1] is 3, arr[3] is changed to 1
Since arr[2] is 0, arr[0] is changed to 2
Since arr[3] is 2, arr[2] is changed to 3

Example 2:
Input: arr[]  = {2, 0, 1, 4, 5, 3};
Output: arr[] = {1, 2, 0, 5, 3, 4};

Example 3:
Input: arr[]  = {0, 1, 2, 3};
Output: arr[] = {0, 1, 2, 3};

Example 4:
Input: arr[]  = {3, 2, 1, 0};
Output: arr[] = {3, 2, 1, 0};
```

A **Simple Solution** is to create a temporary array and one by one copy ‘i’ to ‘temp[arr[i]]’ where i varies from 0 to n-1.

Below is C implementation of the above idea.

```
// A simple C program to rearrange contents of arr[]
// such that arr[j] becomes j if arr[i] is j
#include<stdio.h>

// A simple method to rearrange 'arr[0..n-1]' so that 'arr[j]'
```

```
// becomes 'i' if 'arr[i]' is 'j'
void rearrangeNaive(int arr[], int n)
{
    // Create an auxiliary array of same size
    int temp[n], i;

    // Store result in temp[]
    for (i=0; i<n; i++)
      temp[arr[i]] = i;

    // Copy temp back to arr[]
    for (i=0; i<n; i++)
      arr[i] = temp[i];
}

// A utility function to print contents of arr[0..n-1]
void printArray(int arr[], int n)
{
    int i;
    for (i=0; i<n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Drive program
int main()
{
    int arr[] = {1, 3, 0, 2};
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, n);

    rearrangeNaive(arr,  n);

    printf("Modified array is \n");
    printArray(arr, n);
    return 0;
}
```

Output:

```
Given array is
1 3 0 2
Modified array is
2 0 3 1
```

Time complexity of the above solution is O(n) and auxiliary space needed is O(n).

Can we solve this in O(n) time and O(1) auxiliary space?

The idea is based on the fact that the modified array is basically a permutation of input array. We can find the target permutation by storing the next item before updating it.

Let us consider array '{1, 3, 0, 2}' for example. We start with i = 0, arr[i] is 1. So we go to arr[1] and change it to 0 (because i is 0). Before we make the change, we store old value of arr[1] as the old value is going to be our new index i. In next iteration, we have i = 3, arr[3] is 2, so we change arr[2] to 3. Before making the change we store next i as old value of arr[2].

The below code gives idea about this approach.

```
// This function works only when output is a permutation
// with one cycle.
void rearrangeUtil(int arr[], int n)
{
    // 'val' is the value to be stored at 'arr[i]'
    int val = 0;   // The next value is determined
                   // using current index
    int i = arr[0];  // The next index is determined
                     // using current value

    // While all elements in cycle are not processed
    while (i != 0)
    {
        // Store value at index as it is going to be
        // used as next index
        int new_i = arr[i];

        // Update arr[]
        arr[i] = val;

        // Update value and index for next iteration
        val = i;
        i = new_i;
    }

    arr[0] = val;  // Update the value at arr[0]
}
```

The above function doesn't work for inputs like {2, 0, 1, 4, 5, 3}; as there are two cycles. One cycle is (2, 0, 1) and other cycle is (4, 5, 3).

How to handle multiple cycles with the O(1) space constraint?

The idea is to process all cycles one by one. To check whether an element is processed or not, we change the value of processed items arr[i] as -arr[i]. Since 0 can not be made negative, we first change all arr[i] to arr[i] + 1. In the end, we make all values positive and subtract 1 to get old values back.

```
// A space efficient C program to rearrange contents of
//  arr[] such that arr[j] becomes j if arr[i] is j
#include<stdio.h>
```

```
// A utility function to rearrange elements in the cycle
// starting at arr[i]. This function assumes values in
// arr[] be from 1 to n.  It changes arr[j-1] to i+1
// if arr[i-1] is j+1
void rearrangeUtil(int arr[], int n, int i)
{
    // 'val' is the value to be stored at 'arr[i]'
    int val = -(i+1);  // The next value is determined
                       // using current index
    i = arr[i] - 1;  // The next index is determined
                     // using current value

    // While all elements in cycle are not processed
    while (arr[i] > 0)
    {
        // Store value at index as it is going to be
        // used as next index
        int new_i = arr[i] - 1;

        // Update arr[]
        arr[i] = val;

        // Update value and index for next iteration
        val = -(i + 1);
        i = new_i;
    }
}

// A space efficient method to rearrange 'arr[0..n-1]'
// so that 'arr[j]' becomes 'i' if 'arr[i]' is 'j'
void rearrange(int arr[], int n)
{
    // Increment all values by 1, so that all elements
    // can be made negative to mark them as visited
    int i;
    for (i=0; i<n; i++)
        arr[i]++;

    // Process all cycles
    for (i=0; i<n; i++)
    {
       // Process cycle starting at arr[i] if this cycle is
       // not already processed
       if (arr[i] > 0)
          rearrangeUtil(arr, n, i);
    }

    // Change sign and values of arr[] to get the original
    //  values back, i.e., values in range from 0 to n-1
```

```
    for (i=0; i<n; i++)
          arr[i] = (-arr[i]) - 1;
}

// A utility function to print contents of arr[0..n-1]
void printArray(int arr[], int n)
{
    int i;
    for (i=0; i<n; i++)
        printf("%d ", arr[i]);
    printf("\n");
}

// Drive program
int main()
{
    int arr[] =  {2, 0, 1, 4, 5, 3};
    int n = sizeof(arr)/sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, n);

    rearrange(arr,  n);

    printf("Modified array is \n");
    printArray(arr, n);
    return 0;
}
```

Output:

```
Given array is
2 0 1 4 5 3
Modified array is
1 2 0 5 3 4
```

The time complexity of this method seems to be more than O(n) at first look. If we take a closer look, we can notice that no element is processed more than constant number of times.

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

44 Comments Category: Arrays

Related Posts:

- Longest Span with same Sum in two Binary arrays
- Count Inversions of size three in a give array
- Find the subarray with least average
- Count triplets with sum smaller than a given value
- Find zeroes to be flipped so that number of consecutive 1's is maximized
- Reorder an array according to given indexes
- Find maximum value of Sum(i*arr[i]) with only rotations on given array allowed
- Find maximum average subarray of k length

(Login to Rate and Mark)

3.5 Average Difficulty : **3.5/5.0**
Based on **4** vote(s)

Add to TODO List

Mark as DONE

Writing code in comment? Please use code.geeksforgeeks.org, generate link and share the link here.