ARRAYS - SET 1 SOLUTIONS

Note: All solutions are written in C++.

Question 1:

- **Problem link**: https://leetcode.com/problems/maximum-subarray/
- **Difficulty level**: Easy

Explanation:

This is a direct implementation of Kadane's algorithm. Kadane's algorithm gives a simple way to find the maximum subarray in an array. It basically keeps track of each of the positive subarrays (max_ending_here is used for this). Also, it keeps track of the 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.

Solution:

```
class Solution {
public:
   int maxSubArray(vector<int>& nums) {
       int max_so_far = INT_MIN;
       int max_ending_here = 0;

       for(int i=0; i<nums.size(); i++)
       {
           max_ending_here += nums[i];

           if(max_so_far < max_ending_here)
           {
```

```
                max_so_far = max_ending_here;
            }

            if(max_ending_here < 0)
            {
                max_ending_here = 0;
            }
        }
        return max_so_far;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(1)

Question 2:

- **Problem link**: https://leetcode.com/problems/find-n-unique-integers-sum-up-to-zero/
- **Difficulty level**: Easy

Explanation:

Suppose the number of integers is 'n'. If n is even, then, we can choose the 1st (n/2) positive integers and the first (n/2) negative integers. These numbers will cancel each other out giving 0 as sum. If n is odd, then, we need to do the same process as before. Just add 0 as an extra element.

Solution:

```
class Solution {
public:
    vector<int> sumZero(int n) {

        vector<int> output;

        if(n % 2 == 1)
        {
            output.push_back(0);
        }

        for(int i=1; i<=n/2; i++)
        {
            output.push_back(i);
            output.push_back(i*(-1));
        }

        return output;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(n)
  Space complexity is O(1) if the output array does not count as extra space for space complexity analysis.

Question 3:

- **Problem link**: https://leetcode.com/problems/missing-number/
- **Difficulty level**: Easy

Explanation:

Simply subtracting the sum of all the elements in the array from the sum of first 'n' natural numbers will give us the missing number.

Solution:

```
class Solution {
public:
    int missingNumber(vector<int>& nums) {
        int n=nums.size();
        int ideal_sum=(n*(n+1))/2;
        int actual_sum=0;
        for(int i=0;i<n;i++) actual_sum+=nums[i];
        return ideal_sum-actual_sum;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(1)

Question 4:

- **Problem link**: https://leetcode.com/problems/slowest-key/
- **Difficulty level**: Easy

Explanation:

By traversing the 'releaseTimes' array once starting from index 1, we calculate the duration required for the i-th key pressed as 'releaseTimes[i]-releaseTimes[i-1]'. At every iteration we check whether the 'time' (duration of key press) is longer than before. If yes, we update both the longest duration as well as its corresponding key. If

the duration is the same as the current longest duration found, then the lexicographically largest key of the keypresses is stored.

Solution:

```
class Solution {
public:
    char slowestKey(vector<int>& releaseTimes, string keysPressed) {
        char ch=keysPressed[0];//initializing max-character
        int time=releaseTimes[0];//initializing longest duration

        for(int i=1;i<releaseTimes.size();i++)
        {
            if(releaseTimes[i]-releaseTimes[i-1]>=time)
            {
                if(releaseTimes[i]-releaseTimes[i-1]==time)
                    ch=max(ch,keysPressed[i]);
                else ch=keysPressed[i];
                time=releaseTimes[i]-releaseTimes[i-1];
            }
        }

        return ch;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(1)

Question 5:

- **Problem link**: https://leetcode.com/problems/product-of-array-except-self/
- **Difficulty level**: Medium

Explanation:

Suppose the length of an array is 'n'. Then the product of the array except the element at index 'i' can be calculated optimally by multiplying all elements to left of 'i' and then, multiplying this with the product of all elements to right of 'i'. So, traverse the array from left to right and for each position 'i' calculate the product of all elements to the left of 'i'. Then, traverse the array again from right to left and for each position 'i' multiply the existing product with the product of all elements to the right of 'i'.

Solution:

```
class Solution {
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        int n = nums.size();
        vector<int> output (n, 1);

        for(int i=1; i<n; i++)
        {
            output[i] = output[i-1]*nums[i-1];
        }
        int rightProduct = 1;
        for(int i=n-1; i>=0; i--)
        {
            output[i] = output[i]*rightProduct;
            rightProduct = rightProduct*nums[i];
        }
        return output;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(n)
  Space complexity is O(1) if the output array does not count as extra space for space complexity analysis.

Question 6:

- **Problem link**: https://leetcode.com/problems/beautiful-arrangement-ii/
- **Difficulty level**: Medium

Explanation:

Method 1 (Brute Force): We check all the permutations of the array by calculating the number of unique differences of adjacent elements in each permutation. This is definitely not feasible as time complexity will be O(n!) and will give us TLE (Time Limit Exceeded).

Method 2 : This approach is known as constructive approach and questions of this type are known to be solved by forming 'constructive algorithms', where there are multiple solutions and we have to find a pattern that can be generated programmatically, and implement it. In our problem, let's consider these cases,

Case 1 (When k=1) : Then in order to get k=1, we can simply say that our 'answer' is the array [1,2,3,....,n].

Case 2 (When k=n-1) : Intuitively we can see that considering the array [1, n, 2, n-1, 3, n-2...] we can get the difference array as [n-1, n-2, n-3,..., 1].

Case 3 (Any other value of k) : By using Case 2 here, we can use the last 'k+1' elements out of the 'n' elements (these 'n' elements are basically the first 'n' positive integers) to compute 'k' unique differences. All the remaining elements, i.e. the first (n-(k+1)) elements, we can simply write them in sorted order.

Example (Case 3). When n = 6 and k = 3 we will construct the array as [1, 2, 3, 6, 4, 5]. This consists of two parts: a construction of [1, 2] and a construction of [1, 4, 2, 3] where every element has 2 added to it (i.e. [3, 6, 4, 5]).

Solution:

```
class Solution {
public:
    vector<int> constructArray(int n, int k) {
        vector<int> answer(n,-1);
        if(k==1)
        {
            for(int i=0;i<n;i++) answer[i]=i+1;
        }
        else if(k==n-1)
        {
            int x=1;
            for(int i=0;i<n;i+=2) answer[i]=x++;
            if(n%2)
            {
                for(int i=n-2;i>=0;i-=2) answer[i]=x++;
            }
            else for(int i=n-1;i>=0;i-=2) answer[i]=x++;
        }
        else
        {
            for(int i=0;i<n-k-1;i++) answer[i]=i+1;
            vector<int> temp=constructArray(k+1,k);
            for(int i=n-k-1;i<n;i++) answer[i]=temp[i-(n-k-1)]+(n-k-1);
        }
        return answer;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(n)
  Space complexity is O(1) if the output array does not count as extra space for space complexity analysis.

Question 7:

- **Problem link**: https://leetcode.com/problems/product-of-the-last-k-numbers/
- **Difficulty level**: Medium

Explanation:

Maintain an array 'productList'. For an index 'i', productList[i] will represent the product of the 1st 'i' added numbers. Also maintain a variable which holds the index of the last 0 added to the list of numbers. If the newly added number is a 0, update the index of last 0 and push 1 as the product. For any other number, push the product of all numbers including the currently added number into the 'productList' array. Whenever the product of last 'k' numbers is to be returned, just check if the last 'k' numbers contain a 0. If so, return 0 as the product. If 'k' is the same as the size of 'productList', then, the total product is to be returned which is stored as the last element of 'productList' array. For any other value of 'k', divide the total product (stored as the last element of 'productList' array) by the product of the first 'size - k' numbers and return this value.

Solution:

```
class ProductOfNumbers {

private:
    vector<int> productList;
    int lastZeroIdx;
```

```
public:
    vector<int> numList;

    ProductOfNumbers() {
        lastZeroIdx = -1;
    }

    void add(int num) {
        int size = productList.size();

        if(num == 0)
        {
            lastZeroIdx = size;
            productList.push_back(1);
        }
        else if(num == 1)
        {
            if(size == 0)
            {
                productList.push_back(1);
            }
            else
            {
                productList.push_back(productList[size-1]);
            }
        }
        else
        {
            if(size == 0)
            {
                productList.push_back(num);
            }
            else
            {
                productList.push_back(num * productList[size-1]);
```

```
            }
        }
    }

    int getProduct(int k) {
        int size = productList.size();

        if(lastZeroIdx >= size-k)
        {
            return 0;
        }
        else
        {
            if(size == k)
                return productList[size-1];

            return productList[size-1]/productList[size-k-1];
        }
    }
};
```

Complexity:

- Each of the operations 'add' and 'product' take O(1) time. So, for 'n' operations, total time complexity will be O(n).
- Space: O(n)

Question 8:

- **Problem link**: https://leetcode.com/problems/first-missing-positive/
- **Difficulty level**: Hard

Explanation:

Suppose the given array is of length ‘n’. Maintain a ‘count’ array of same length ‘n’ and initialise each of the elements of the array to 0. Traverse through the given array and whenever the current element of the given array is in the range [1,n], increment the corresponding counter in the ‘count’ array. As a result, for an index ‘i’, count[i] will represent the number of times the number ‘i+1’ occurs in the given array. Finally, traverse the ‘count’ array once and if count[i] is 0 for an index ‘i’, then, ‘i+1’ is the first missing positive number. If all elements in the ‘count’ array are non-zero, then, ‘n+1’ is the first missing positive number.

Solution:

```
class Solution {
public:
    int firstMissingPositive(vector<int>& nums) {

        int n = nums.size();
        vector<int> count (n, 0);

        for(int i=0; i<n; i++)
        {
            if(nums[i]>=1 && nums[i]<=n)
            {
                count[nums[i]-1]++;
            }
        }

        for(int i=0; i<n; i++)
        {
            if(count[i] == 0)
            {
                return i+1;
            }
        }
```

```
        return n+1;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(n)

Question 9:

- **Problem link**: https://leetcode.com/problems/median-of-two-sorted-arrays/
- **Difficulty level**: Hard

Explanation:

Merge the two given sorted arrays such that the resultant array is also a sorted array. Then, return the median of this resultant merged array. If the merged array is of odd length, the median will be the element in the middle of the array. If the merged array is of even length, the median will be the mean of the two elements in the middle of the array.

Solution:

```
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>&
nums2) {

        int n1 = nums1.size();
        int n2 = nums2.size();
        int n = n1+n2;
```

```
        if(n == 0)
            return 0;

        vector<int> mergedArray(n1+n2);

        int i=0, j=0, k=0;

        while(i<n1 && j<n2)
        {
            if(nums1[i] < nums2[j])
            {
                mergedArray[k] = nums1[i];
                k++;
                i++;
            }
            else
            {
                mergedArray[k] = nums2[j];
                k++;
                j++;
            }
        }

        while(i<n1)
        {
            mergedArray[k] = nums1[i];
            k++;
            i++;
        }

        while (j<n2)
        {
            mergedArray[k] = nums2[j];
            k++;
            j++;
```

```
        }

        if(n == 1)
            return mergedArray[0];

        double median;

        if(n%2 == 1)
        {
            return mergedArray[n/2];
        }
        else
        {
            return (double)(mergedArray[n/2]+mergedArray[n/2-1])/2;
        }
    }
};
```

Complexity:

- Time: O(n+m)
- Space: O(n+m)
  ('n' and 'm' are lengths of the two given sorted arrays)