Searching and Sorting 36

Sunday, 8 October 2023 9:10 AM

① Find first and last position of an element in sorted array

Naive

Iterate on all elements and keep track of first and last index.

```
def find(arr, x):
    first, last = -1,-1
    for i in range(len(arr)):
        if arr[i] == x:
            continue
        if first == -1:
            first = i
        last = i
    return [first, last]
```

Time : $O(n)$

Space : $O(1)$

Efficient — Using Binary Search

For first occurence — use find index of number and then search again to left subarray

For last " " " " " " " " right " "

```
def find(arr, x):
    low, high, res = 0, n-1, -1
    while low <= high:
        mid = (low+high)//2
        if arr[mid] > x:
            high = mid-1
        elif arr[mid] < x:
            low = mid+1
        else:
            res = mid
            high = mid-1
    return res
```

```
def last(arr, x):
    low, high, res = 0, len(arr)-1, -1
    while low <= high:
        mid = (low+high)//2
        if arr[mid] <= x:
            low = mid+1
        elif arr[mid] > x:
            high = mid-1
        else:
            res = mid
            low = mid+1
    return res
```

$O(\log n)$

$O(1)$ space

Based on above: Count no. of occurences (or frequency) in a sorted array.

② Find a fixed point (value equal to index) in a given array

a) Linear Search $O(n^2)/O(1)$

b) Binary Search $O(\log n)$

— array is sorted

— Check whether middle element is fixed point or not.
If it is, return mid, Otherwise, If index of middle element is less than or equal to value at right index, then fixed point exists in the right side.

— Similarly, check if middle-1 element is greater than or equal to value of the index.

```
def binary_search(arr, low, high):
    if low <= high:
        mid = (low+high)//2
        if mid == arr[mid]:
            return mid
        res = -1
        if mid+1 <= arr[high]:
            res = binary_search(arr, mid+1, high)
        if res != -1:
            return res
        if mid-1 >= arr[0]:
            return binary_search(arr, low, mid-1)
```

Implicitly Allowed

left side ⇒ check = check ; end = min(arr[mid-1], mid-1)

right side ⇒ check = max(arr[mid+1], mid+1) ; end = end

③ Search in Sorted Rotated Array

a) Find pivot point, divide array into two-sub array and perform binary search.

b) Directly apply binary search without finding pivot $O(\log n)$

```
def search(arr,x,k):
    low, high = 0, n-1
    while low <= high :
        mid = (low+high)//2
        if arr[mid] == k:
            return mid
        if arr[low] <= arr[mid]:
            if arr[low]<= k < arr[mid]:
                high = mid-1
            else:
                low = mid+1
        else:
            if arr[mid] < k <= arr[high]:
                low = mid+1
            else:
                high = mid-1
    return -1
```

Implicitly allowed to array

eg: 5 1 2 3 3 3 3

```
if (arr[low]<= arr[mid]) and (arr[high]==arr[mid]):
    low += 1
    high -= 1
if arr[low] <= arr[mid]:
    if arr[low]<= k <= arr[mid]:
        high = mid-1
    else:
        low = mid+1
if arr[mid] <= k <= arr[high]:
    low = mid+1
else:
    high = mid-1
return False
```

Minimum element in a sorted and rotated array:

— Linear Search $O(n)$

— Binary Search $O(\log n)$

```
def find(arr,n):
    i,j = 0,n-1
    while i<=j:
        m = (i+j)//2
        if arr[m] < arr[m-1]:
            return arr[m]
        if arr[m] < arr[i] :
            j = m-1
        else:
            i = m+1
```

Find if there is a pair with a given sum in the sorted rotated array

```
def find(arr,sum):
    pivot = 0
    for i in range(n-1):
        if arr[i] > arr[i+1]:
            pivot = i+1
            break
    left = pivot
    right = pivot-1   # (pivot-1+n)%n
    while left != right:
        if arr[left] + arr[right] == x:
            return True
        elif arr[left] + arr[right] < x:
            left = (left+1)%n
        else:
            right = (right-1+n)%n
    return False
```

④ Copies list of an integer

— Binary Search — Find largest integer i whose square is less than or equal to given number. $O(\log x)$

```
def floorSqrt(x):
    low = 1    high = x    ans = -1
    if x==0 or x==1:
        return x
    while low <= high:
        mid = (low+high)//2
        if mid*mid == x:
            return mid
        if mid*mid < x:
            ans = mid
            low = mid+1
        else:
            high = mid-1
    return ans
```

— floor(x) — In floating

```
sqrt = x*0.5
return int(sqrt)
```

⑤ Move and Mile of an array using minimum number of comparisons

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

— If n is even then initialize min and max as min and max of first two elements

— For rest pick from two pairs and compare.

$O(n)$ time $O(1)$ space

⑥ Find Majority Element (Moore's Voting Algo)

— First chg you find element that may be majority element in the array

— There is a majority element, then this step will definitely return majority element otherwise, return a candidate element.

— Check if the element obtained from the above step is majority element. Necessary step as there might be no majority element.

```
def majorityElement(A,n):
    res = A[0]   count = 1
    for i in range(n):
        if A[i] == res:   count += 1
        else:   count -= 1
        if count == 0:   res = A[i]   count = 1
    count = 0
    for i in range(n):   if A[i] == res:   count += 1
    if count > n//2:   return res
    return -1
```

$O(n)$ time

$O(1)$ space

Find candidate element

Check

⑦ Optimal location of point to minimize total distance.

⑧ Find the maximum and minimum number

Approach 1

— Create temp = [0]*n size

— Traverse input array

— If temp[arr[i]-1] == 0: temp[arr[i]-1] = 1

— If temp[arr[i]-1] == 1: repeating no. = arr[i]

— Traverse temp[:] and output "i+1" where temp[i] == 0

missing = i+1

$O(n)$ and space $O(n)$

Use 2nd another approach

Approach 2

Traverse array, while traversing use absolute value of every element as index and make value at this index negative to mark it visited. For something is already marked negative, then the repeating element. Traverse array and look for positive value, missing element.

$O(n)$ time $O(1)$ space

```
def findMissingAndRepeating(arr,n):
    repeat, missed = 0,0
    for i in range(n):
        if arr[abs(arr[i])-1] < 0:
            repeat = abs(arr[i])
        else:
            arr[abs(arr[i])-1] = -arr[abs(arr[i])-1]
    for i in range(n):
        if arr[i] > 0:
            missed = i+1
            break
    return [repeat, missed]
```

⑨ Search in an array where adjacent differ by at most k

Simple: Traverse the array and compare with the x.

Optimized — using this fact that difference between all adjacent elements is atleast k.

— Idea is to start comparing from leftmost element and find the difference between current array element and x. (diff)

— We have that we must be "diff/k" away, so instead of considering one by one we jump "diff/k".

$O(n)$

```
def search(arr,n,x,k):
    i = 0
    while i < n:
        if arr[i] == x:
            return i
        else:
            i += max(1, abs(arr[i]-x)//k)
    return -1   # if not found
```

where k is 1

$i += abs(arr[i]-x)$

⑩ Find a pair with the given difference

— Sorting and Binary Search $O(n)$

```
def findPair(arr, diff):
    i = 0, j = 1
    while i < len(arr) and j < len(arr):
        if i != j and arr[j]-arr[i] == diff:
            return True
        elif arr[j] - arr[i] < diff:
            j += 1
        else:
            i += 1
    return False
```

⑪ Find low using Two Pointer Approach

$O(\sqrt{n} \cdot \log n)$ time

```
def findLow(arr,k):
    if len(arr)<1:
        return
    ans = []
    n = len(arr)
    ans.append(0)
    for i in range(n-1):
        if i>0 and arr[i]==arr[i-1]:
            continue
        for j in range(i+1,n-2):
            if j>i+1 and arr[j]==arr[j-1]:
                continue
            l = j+1
            r = n-1
            while l<r:
                sum = arr[i]+arr[j]+arr[l]+arr[r]
                if sum == k:
                    if (arr[i],arr[j],arr[l],arr[r]) not in ans:
                        ans.append((arr[i],arr[j],arr[l],arr[r]))
                    l = l+1
                    r = r-1
                    while l<r and arr[l]==arr[l-1]:
                        l = l+1
                    while r>l and arr[r]==arr[r+1]:
                        r = r-1
                elif sum > k:
                    r = r-1
                else:
                    l = l+1
    return ans
```

⑫ Maximum sum with at most two elements are adjacent (Kadane Half)

Naive — Match all subsequence of array and find the subsequence with max sum such that no two elements are adjacent.

— two options a) use the value. b) not use the value.

if a) can not use previous value, $(i-1)$ but can proceed to use before.

if b) potential left from previous value $(i-1)$ and all before.

— Formula : $ans[i] = \max(ans[i-1] + current\ value,\ ans[i-2])$

```
def rec(nums,index):
    if index >= len(nums):
        return 0
    return max(nums[index] + rec(nums, index+2), nums[index+1])
def findMaxSum(arr, n):
    return rec(arr,0)
```

Time : 2^n at each index two options hold, leads to $2 \times 2 \times \cdots = (2^n)$

Space : $O(n)$ — auxiliary space.

Dynamic Programming Time $O(N)$ Space $O(N)$

```
def find_Max_Sum(arr,N):
    dp = [0 for i in range(2)] for j in range(N)]
    if N==1: return arr[0]
    dp[0][0], dp[0][1] = 0, arr[0]
    for i in range(1,N):
        dp[i][0] = dp[i-1][0] + arr[i]
        dp[i][1] = max(dp[i-1][1], dp[i-2][0])
    return max(dp[N-1][0], dp[N-1][1])
```

The value of current state (for 2th element) depends on only two states of previous element. We can use two variable to store two states of previous element.

— exclude stores value of max subsequence sum till $i-1$ when $arr[i-1]$ is excluded

— include stores value of max subsequence sum till $i-1$ when $arr[i-1]$ is included

— exclude of current state will be max(exclude, include)

include of current state will be exclude + $arr[i]$

```
def find_Sum(arr,n):
    exclude, include = 0, arr[0]
    new_include, new_exclude = -1,-1
    for i in range(n):
        new_include = exclude + arr[i]
        new_exclude = max(include, exclude)
        exclude = include = new_exclude
        include = new_include
    return max(exclude, new_include)
```

$O(N)$ time

constant space

⑬ Maximum sum with exactly k elements from array.

```
def max_sum(arr,k):
    if k==0: return 0
    value1 = arr[0]
    if k==1 : return value1
    value2 = max(value1, arr[1])
    if k==2: return max(value1, value2)
    max_value = None
    for i in range(2,k):
        max_value = max(arr[i] + value1, value2)
        value1 = value2
        value2 = max_value
    return max_value
```

⑭ Count triplets with sum smaller than a given value. $O(n^2)$

Form three loops and check — $O(n^3)$

Efficient : Sort array first and then use Two Pointer technique. $O(n^2)$

```
count = 0
arr.sort()
for i in range(n-2):
    start = i+1
    end = n-1
    while start < end:
        if arr[i] + arr[start] + arr[end] < sum:
            count += (end - start)
            start += 1
        else:
            end -= 1
return count
```

⑮ Count triplets with sum equal to a given value. $O(n^2)$

Form three loops and check — $O(n^3)$

Efficient : Sort array first and then use Two Pointer technique. $O(n^2)$

```
count = 0
arr.sort()
for i in range(n-2):
    start = i+1
    end = n-1
    while start < end:
        if arr[i] + arr[start] + arr[end] == sum:
            count += 1
            start += 1
            end -= 1
        elif arr[i] + arr[start] + arr[end] < sum:
            start += 1
        else:
            end -= 1
return count
```

⑯ Two Sum

— two loops $O(n^2)$

— Sort array and two pointer approach $O(n \log n)$

— using Hashing $O(n)$ and $O(n)$ space

```
def find(nums, sum):
    s = set()
    for i in range(n):
        temp = sum - arr[i]
        if temp in s:
            return True
        s.add(arr[i])
    return False
```

Return true if exists, find all pairs where sum is n.

Count pairs with given sum

Count all distinct pairs with difference equal to k.

⑰ Merge two sorted arrays

⑱ Product Array Puzzle.

Space $O(n)$ — two extra array instead of two

— Store prefix and suffix product in output array

```
def product_Except_Self(nums,n):
    prod = [1]*n
    temp = 1
    for i in range(n):
        prod[i] = temp
        temp = temp*arr[i]
    temp = 1
    for i in range(n-1,-1,-1):
        prod[i] *= temp
        temp = temp*arr[i]
    return prod
```

Space $O(1)$

```
def ____(nums):
    prod = 1
    for i in arr:
        prod *= i
    for i in range(n):
        arr[i] = int(prod / arr[i])
    return arr
# assumption : array does not contain 0
```

⑲ Move zeroes to end of an array

— Make a new array (temp), initialize it with first of input array.

— Make a map that stores elements and its corresponding index.

— In all cases if starting from 1 to n-1,

— If i not in current map according to current array then

— set, fill the pos with current element from array.

— After swapping, use update mapping accordingly, so temp[i] to the pos and arr[i] to where temp[i] was earlier.

```
def moveZeroes(nums):
    arr = [[nums[i],i] for i in range(len(nums))]
    arr.sort()
    n = len(nums)
    count = 0
    while i<n:
        if i == arr[i][1]:
            i += 1
        else:
            temp_index = arr[i][1]
            arr[i],arr[temp_index] = arr[temp_index],arr[i]
            count += 1
    return count
```

Minimum number of swaps to sort array only adjacent swapping allowed

— Using the fact that :

$\boxed{\text{no. of cycle needed} == \text{no. of inversions}}$

— basically count inversions in array

⑳ Rotate Array Puzzle

Naive — Rotate one by one

Efficient — Using Reversal Algorithm

```
def rotate(arr,d,n):
    d = d%n
    reverse(arr,0,d-1)
    reverse(arr,d,n-1)
    reverse(arr,0,n-1)
```

```
def reverse(arr,i,j):
    while i<j:
        temp = arr[i]
        arr[i] = arr[j]
        arr[j] = temp
        i += 1
        j -= 1
```

㉑ Find max possible value at index from array.

```
def max_at(arr,x):
    if x==0: return 0
    value1 = arr[0]
    if x==1 : return value1
    value2 = max(value1, arr[1])
    if x==2: return max(value1, value2)
    max_value = None
    for i in range(2,x):
        max_value = max(arr[i] + value1, value2)
        value1 = value2
        value2 = max_value
    return max_value
```