Hashing 12

Thursday, 3 August 2023 6:32 AM

(1) Intersection of Unsorted Arrays

Naive

1) Initialize res = 0
2) Traverse through every element of a[]
   - ⟶ Check if it has not appeared already
   - ⟶ If a new element and also present in b[], do res++
3) Return res

Time: $O(m*(m+n))$

```
def intersection(arr1, m, arr2, n):
    res = 0
    for i in range(m):
        flag = False
        for j in range(i-1):            # Check if element has appeared
            if arr1[j] == arr1[i]:
                flag = True
                break
        if flag == True:                # If appeared ignore
            continue
        for j in range(n):              # if not appeared, look for it in another array
            if arr1[i] == arr2[j]:
                res += 1
                break
    return res
```

Efficient

1) Insert all elements of a[] in a set (s_a) — $\theta(m)$
2) Insert all elements of b[] in another set (s_b) — $\theta(n)$
3) Now traverse through s_a and increment count for elements that are present in s_b also — $O(m)$

Time: $O(m+n)$ Aux Space: $O(m+n)$

1) Insert all elements of a[] in set (s_a) — $\theta(m)$
2) Traverse through b[]. Search for every element b[i] in s_a. If present: — $\theta(n)$
   (a) increment res
   (b) remove b[i] from s_a

Time: $\theta(m+n)$ Aux Space: $O(m)$

```
def intersection(arr1, m, arr2, n):
    us = set()
    for i in range(m):
        us.add(arr1[i])
    res = 0
    for i in range(n):
        if arr2[i] in us:
            res += 1
            us.remove(arr2[i])
    return res
```

(2) Union of Two Unsorted Arrays

Naive

```
def unionSize(arr1, m, arr2, n):
    c = [0]*(m+n)
    for i in range(m):
        c[i] = arr1[i]
    for i in range(n):
        c[i+m] = arr2[i]
    res = 0
    for i in range(m+n):
        flag = False
        for j in range(i):
            if c[i] == c[j]:
                flag = True
                break
        if flag == False:
            res += 1
    return res
```

Time: $O((m+n)*(m+n))$ Aux: $O(m+n)$

Efficient

```
def unionSize(arr1, m, arr2, n):
    us = set()
    for i in arr1:
        us.add(i)
    for i in arr2:
        us.add(i)
    return len(us)
```

Time: $O(m+n)$ Aux: $O(m+n)$

(3) Pair with given sum in Unsorted Array

Naive

```
def pairWithSum(arr, x):
    for i in range(len(arr)):
        for j in range(i+1, len(arr)):
            if (arr[i] + arr[j] == x):
                return 1
    return 0
```

Time: $O(n^2)$ Aux: $O(1)$

Efficient

```
def pairWithSum(arr, x):
    us = set()
    for i in arr:
        if x-i in us:
            return 1
        us.add(i)
    return 0
```

Time: $O(n)$ Space: $O(n)$

(4) Subarray with 0 sum in Python

Naive

```
def isZero(l):
    n = len(l)
    for i in range(n):
        for j in range(i+1, n+1):
            if sum(l[i:j]) == 0:
                return True
    return False
```

Time: $O(n^2)$

Efficient

Idea: Using Prefix sum and Hashing

$a_0\ a_1\ a_2 \ldots a_{i-1}, a_i, a_{i+1}, \ldots a_j \ldots a_{n-1}$ (pre_sum1 = $a_0 \ldots a_{i-1}$; pre_sum2 = $a_0 \ldots a_j$)

⟶ if sum of a[i:j] is 0, then pre_sum1 must be same as pre_sum2

```
def isZeroSum(arr):
    pre_sum = 0
    h = set()
    for i in range(len(l)):
        pre_sum += l[i]
        if pre_sum == 0 or pre_sum in h:
            return True
        h.add(pre_sum)
    return False
```

Time: $O(n)$

(5) Subarray with Given Sum

(There are no negative elements in array)

Naive

```
def isSubSum(arr, sum):
    for i in range(len(arr)):
        curr = 0
        for j in range(i, len(arr)):
            curr += arr[j]
            if curr == sum:
                return True
    return False
```

Efficient

We use window sliding technique with a window of variable size.

```
def isSubSum(arr, sum):
    s = 0
    curr = 0
    for e in range(len(arr)):
        curr += arr[e]
        while (curr > sum):
            curr -= arr[s]
            s += 1
        if curr == sum:
            return True
    return False
```

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

(6) Check for Palindrome Permutation

Answer is going to be true if there is at most one character with odd frequency.

abba, acbccab, aaaabbbbddd

Else false

abcc, ab, aaabbcdd

⟶ Let $(e_1,e_1)(e_2,e_2)\ldots(e_k,e_k)$ be pair of even frequency characters, we can always form a palindrome $e_1 e_2 \ldots e_{k-1} e_k e_k e_{k-1} \ldots e_2 e_1$

If there is one odd frequency character, we can still form a palindrome $e_1 e_2 \ldots e_{k-1} e_k e_{k-1} \ldots e_2 e_1$

```
from collections import counter
def isPalPer(s):
    count = Counter(s)          # θ(n)
    odd = 0
    for freq in count.values():     # O(n)
        if freq % 2 != 0:
            odd = odd + 1
            if odd > 1:
                return False
    return True
```

Time: $\theta(n)$

(7) Longest Subarray with Given Sum

Naive

```
def longestSubarrayWithSum(arr, sum):
    n = len(arr)
    res = 0
    for i in range(n):
        curr_sum = 0
        for j in range(i, n):
            curr_sum += arr[j]
            if curr_sum == sum:
                res = max(res, j-i+1)
    return res
```

Time: $O(n^2)$ Space: $O(1)$

Efficient

```
def longestSubarrayWithSum(arr, sum):
    n = len(arr)
    mydict = dict()
    pre_sum = 0
    res = 0
    for i in range(n):
        pre_sum += arr[i]
        if pre_sum == sum:
            res = i+1
        if pre_sum not in mydict:
            mydict[pre_sum] = i
        if pre_sum - sum in mydict:
            res = max(res, i-mydict[pre_sum-sum])
    return res
```

Time: $O(n)$ Space: $O(n)$

(8) Longest Subarray with equal number of zero and one

* (Longest subarray with zero sum)

Naive

```
def largestZeroSubarray(arr):
    n = len(arr)
    res = 0
    for i in range(n):
        c_0 = 0
        c_1 = 0
        for j in range(i, n):
            if arr[j] == 0:
                c_0 += 1
            else:
                c_1 += 1
            if (c_0 == c_1):
                res = max(res, j-i+1)
    return res
```

Time: $O(n^2)$ Space: $O(1)$

Efficient

```
def largestZeroSubarray(arr):
    n = len(arr)
    for i in range(n):
        if arr[i] == 0:
            arr[i] = -1
    mydict = dict()
    sum = 0
    maxlen = 0
    for i in range(n):
        sum += arr[i]
        if sum == 0:
            maxlen = i+1
        if sum in mydict:
            maxlen = max(maxlen, i-mydict[sum])
        else:
            mydict[sum] = i
    return maxlen
```

Time: $O(n)$ Space: $O(n)$

(9) Longest Common Span with same sum in Binary Array

⟶ we are given two binary subarrays of same sizes

Naive

```
def longestCommonSpan(arr1, arr2):
    res = 0, n = len(arr1)
    for i in range(n):
        sum1 = 0
        sum2 = 0
        for j in range(i, n):
            sum1 += arr1[j]
            sum2 += arr2[j]
            if sum1 == sum2:
                res = max(res, j-i+1)
    return res
```

Time: $O(n^2)$ Space: $O(1)$

Efficient

— Problem is going to reduce into problem of longest subarray with 0 sum in array.

1) Compute a difference array
   ```
   int temp[n]
   for i in range(n):
       temp[i] = arr1[i] - arr2[i]
   ```
2) Return length of the longest subarray with 0 sum in temp.

values in temp[]:
- We get 0 when values are same in both
- We get 1 when arr1[i]=1 and arr2[i]=0
- We get -1 when arr1[i]=0 and arr2[i]=1

```
def longestCommonSpan(arr1, arr2):
    n = len(arr1)
    temp = [0]*n
    for i in range(n):
        temp[i] = arr1[i] - arr2[i]
    mydict = dict()                 # longest subarray with zero sum
    sum = 0
    max_len = 0
    for i in range(n):
        sum += temp[i]
        if sum == 0:
            max_len = i+1
        if sum in mydict:
            max_len = max(max_len, i-mydict[sum])
        else:
            mydict[sum] = i
    return max_len
```

Time: $O(n)$ Space: $O(n)$

(10)* Longest Consecutive Subsequence

— We need to find the longest subsequence in the form of $x, x+1, x+2, \ldots, x+i$ with these elements appearing in any order.

Naive (use sorting)

```
def findLongest(arr):
    n = len(arr)
    arr.sort()
    res = 1
    curr = 1
    for i in range(1, n):
        if arr[i] == arr[i-1]+1:
            curr += 1
        else:
            res = max(res, curr)
            curr = 1
    res = max(res, curr)
    return res
```

Time: $O(n \log n)$ Space: $O(1)$

Efficient

- We first insert all elements in a hash table
- Then with 2n lookups, we find result

```
def findLongest(arr):
    s = set()
    res = 0
    for i in arr:
        s.add(i)
    for i in arr:
        if (i-1) not in s:
            curr = 1
            while (curr+i) in s:
                curr += 1
            res = max(res, curr)
    return res
```

Time: $O(n)$ Space: $O(n)$

(11) Count distinct elements in every window

Naive

1) There will be (n-k+1) windows
2) Traverse through every window and count distinct elements in it

```
def printDistinct(arr, k):
    for i in range(n-k+1):
        print(len(set(arr[i:i+k])))
```

Time: $O((n-k+1)*k)$ Space: $O(k)$

Efficient

— Idea is to use count of previous window to get the current count.

```
from collections import counter
def countDist(arr, k):
    mp = Counter(arr[0:k])          # first window
    print(len(mp))
    for i in range(k, len(arr)):
        x = arr[i-k]                # Remove first item of prev window
        mp[x] -= 1
        mp[arr[i]] += 1             # Add last item of current window
        if (mp[x] == 0):
            del mp[x]
        print(len(mp))
```

Time: $O(n)$ Space: $O(k)$

(12) More than n/k occurences (O(nk) solution)

Naive

1) Sort the given array/list
2) Traverse through the sorted array

```
def printNbyk(arr, k):
    arr.sort()
    i = 1
    count = 1
    while (i < len(arr)):
        while (i < len(arr) and arr[i] == arr[i-1]):
            count += 1
            i += 1
        if count > n//k:
            print(arr[i-1], end=" ")
        count = 1
        i = i+1
```

Time: $O(n \log n)$ Space: $O(1)$

Efficient (Hashing)

```
from collections import Counter
def printNbyk(arr, k):
    C = Counter(arr)
    for x in C:
        if C[x] > n//k:
            print(x, end=" ")
```

Time: $O(n)$ Space: $O(n)$

O(nk) Solution

1. Create an empty map m
2. for i in range(n):
   (a) if m contains arr[i]
       m[arr[i]] ++
   (b) else if m.size is less than (k-1)
       m.put(arr[i], 1)
   (c) else
       decrease all value in m by one.
       If value becomes 0, remove element.
3. For all elements in m, print the elements that actually appear more than n/k times.

{30, 10, 20, 20, 20, 10, 40, 30, 30}

How does this approach work?

Final map: {(10,1), (20,2), (30,2)}

(10, 20, 30, 40) Rejected — 40 rejected itself and three others

(10, 20, 20, 30, 30) Selected

In the rejected set, an element rejects (k-1) distinct others