18/07/22

DSA

Introduction to Arrays

✱ Importance of constraints

If $1 <= N <= 10^5$, then which of the following BigO will work?

Complexity	Iterations	Works?
$O(N^3)$	$(10^5)^3 \simeq 10^{15}$	No
$O(N^2) \log N$	$(10^5)^2 \log_2 10^5$	No
$O(N^2)$	$(10^5)^2 \simeq 10^{10}$	No
$O(N * \log N)$	$10^5 \log_2 10^5 \leq 10^8$	Yes
$O(N)$	10^5	Definitely work

If $1 <= N <= 100$ then N^3 $(100)^3 < 10^8$ ✓

If $1 <= N <= 20$ then 2^N $(2)^{20} < 10^8$ ✓

Note - If we are working on small inputs no matter how many loops you use it works, but when the inputs are higher you have to take care of loops & iterations you use while writing a code.

Q) How to approach a problem?

- Read the question & constraints carefully.
- Formulate an idea or logic.
- Verify the correctness of the logic.
- Mentally develop a pseudocode or rough idea of loops.

- Determine the Time complexity based on the Pseudocode.
- Assess if the time complexity is feasible & wont result in TLE error.
- Note: In worst case we can only have 10^7 or 10^8 iterations
- Re-evaluate the idea/logic if the time constraints are not met, otherwise proceed.
- Code the idea if it is deemed feasible.

✱ Space Complexity:

Space utilised at any point during running the algorithm.

	type	size
a, b, c	(int)	4 bytes

3 x 4 = 12 bytes → we drop the constant
∵ O(1)

if i/p is list[] then it takes (N) size, as we don't know the size of list.

∴ list[] takes O(N) space.

if

li = [1] * 5 ⇒ 5-sized list

li = [0] * k ⇒ k-sized list

Q) def func (N):

arr = [0] * 10	4bytes * 10
x, y, z = 1, 2, 3	4bytes * 3
arr dynamic = [0] * N	4bytes * N

Total space = 40 + 12 + 4N → remove constants
= 4N → remove constant
= O(N)

Calculate space complexity in Big O notation
1) Pick highest order term.
2) Remove constant

★ Rules:

- Dont consider the space acquired by the input size. Space complexity is the order of extra space used by the algorithm.
- arr[] is already given to us, we didn't create it, hence it'll not be counted in the space.
- int N will also not be counted in the space but since it is constant hence doesn't matter.
- Additional space is also called as computational or auxiliary space.

Q) Print all elements of array A.

```
for i in range (0, len(A)):
    print (arr[i])
```

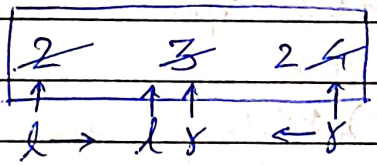
```
for element in arr(A):
    print (element)
```

TC = O(N) & SC = O(1)

Reverse in a range :

Given an array [N] and integers 'l' & 'r'. Reverse the array from 'l' to 'r'.

N = 5, arr = {1, 2, 3, 4, 5} (indices 0, 1, 2, 3, 4) l = 1, r = 3



1 4 3 2 5

Code:

```
def reverse (li[], l, r):
    s = l
    e = r
    while s < e:
        li[s], li[e] = li[e], li[s]
        s += 1
        e -= 1
    return li()
```

TC: O(N)
SC: O(1)

★ Reverse the array:

Given an array [N], reverse the entire array.

N=5, arr={1,2,3,4,5}

o/p: 5 4 3 2 1

5	4	3	2	1
~~1~~	~~2~~	~~3~~	~~4~~	~~5~~

Approach:

1	2	3	4	5

(s) (s) s e (e) (e)

s = 0 // 2 e = 4 // 2

∴ s < e → we need swapping

s >= e → no swap.

```
def reverse(li[], N):
    s = 0
    e = N-1
    while s < e:
        li[s], li[e] = li[e], li[s] #swapping
        s += 1
        e -= 1
```

TC: N/2 iterations → O(N) → we only traversed to the middle of list. ∴ N/2

SC: 4+4 → O(1)

s (4) e (4)

TC: O(N) SC: O(1)

★ Rotate k times:

Q) Given an array [N], rotate the array from right to left 'k' times? (i.e if k=1, last element will come at first position)

N = 5 arr = {1, 2, 3, 4, 5}
k = 3

k=1	5	1	2	3	4
k=2	4	5	1	2	3
k=3	3	4	5	1	2

• Brute force approach:

Rotating one element at a time

1> Take the last element & insest at arr(0) = arr(n-1)
2> Shift all elements to right side by one position.

Repeat step 1 & 2 for k times

```
def rotate (li, N, k):
    for i -> 0 to k-1 : #handling k no. of rotations : k
    temp = li[N-1]
    for j -> N-1 to 1 :
        li[j] = li[j-1]
    li[0] = temp
```

TC = O(k*N) SC: O(1)

Now, we will look for optimized approach

N=9 {10 20 30 40 50 60 70 80 90}
k=4

After 1st rotation: 90 10 20 30 40 50 60 70 80
2nd rotation: 80 90 10 20 30 40 50 60 70
3rd rotation: 70 80 90 10 20 30 40 50 60
4th rotation: 60 70 80 90 10 20 30 40 50

	Part 1	Part 2
Org:	(10 20 30 40 50)	(60 70 80 90)
Rotated:	(60 70 80 90)	(10 20 30 40 50)

Part 1 Part 2
Part 2 Part 1

k=4 Approach 2: {10 20 30 40 50 60 70 80 90}

Reverse whole: 90 80 70 60 50 40 30 20 10
reverse (li, 0, n-1) ↑ ↑ ↑ ↑
0 k-1 k n-1

0 1 2 3 4 5 6 7 8
Reverse Partwise: 60 70 80 90 40 20 30 40 50
reverse (li, 0, k-1)
reverse (li, k, n-1)

Steps: 1) Reverse whole list # O(N)
2) Reverse part 1 # O(N)
3) Reverse part2 # O(N)

TC: N+N+N = 3N ⇒ O(N)
SC: O(1)

Edge case: k>N
N=3 k=6
(0,2) ✓, (0,5) Not possible, it exceeds index of array

N= 1 2 3 (k rotations)

k=1	3	1	2	k=4	3	1	2
k=2	2	3	1	k=5	2	3	1
k=3	1	2	3	k=6	1	2	3

k = k % N

k=3, n=3 k = 3%3 = 0 (zero rotation reqd)
k=4, n=3 k = 4%3 = 1
k=5, n=3 k = 5%3 = 2
k=2, n=3 k = 2%3 = 2

```
def reverse(li[], l, r):
    s = l
    e = r
    while (s < e):
        li[s], li[e] = li[e], li[s]
        s += 1
        e -= 1
```

```
def rotate (li,k):
    n = len (li)
    k = k % N

    # Reverse whole
    reverse (li, 0, n-1)
    # Reverse part 1
    reverse (li, 0, k-1)
    #Reverse part 2
    reverse ( li, k, n-1)
```

TC : O(N)
SC : O(1)

★ Dynamic Array:

What is the drawback of normal array?
→ Size has to be declared

Dynamic Arrays - Automatic resizing

Benefits: Fast lookup
Resizing