Accessing at given index - O(1).

updating - O(1)

inserting an item at beginning - O(n).

Middle - O(n).

End
- Static - O(n)
- Dynamic - O(1)

Removing Beginning - O(n)

Middle - O(n)

End - O(1)

Copying an array - O(n).

Sorting array - O(n log n).

→ Static Array ⇒ → Fixed amount of memory to be used for storing the array's values.

→ Appending values to array therfore involves copying the entire array and allocating new memory to it, Accounting for the extra space needed for newly appended value.

↳ linear time operation.

Dynamic Array : Preemptively allocates double the amount of memory needed to store the array's values.

Appending values to array is const time operation until the allocated memory is filled up.

→ if memory is filled up, array is copied & double the memory space is once again allocated for it.

↳ Amortized Const time insertion at end.

① Linear Search → O(n,1)

arr → | 1 | 2 | 3 | 4 | 5 |

input key → 1 | key = 6

o/p → 0 | o/p = -1.

→ loop over the array

till 0 to n-1

if found return index

→ return -1

---

② Binary Search. O(log n, 1)

↳ applicable only for Sorted Array.

key = 7

| 1 | 3 | 5 | 6 | 7 |

↑ left 0 ↑ Mid ↑ right n-1

mid = (left + right)//2

if arr[mid] = key:
    return mid.

else if arr[mid] < key:
    left = mid + 1

else if arr[mid] > key:
    right = mid - 1

while not (left = right)

or

left = right

---

③ Longest Elem in array O(n,1)

max = -infinite

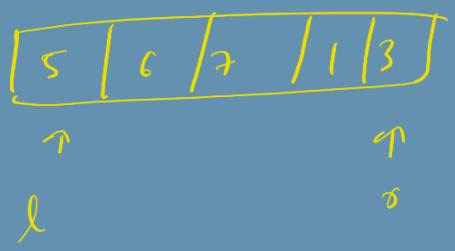
loop through an array → (0, n-1)

if Elem < max:

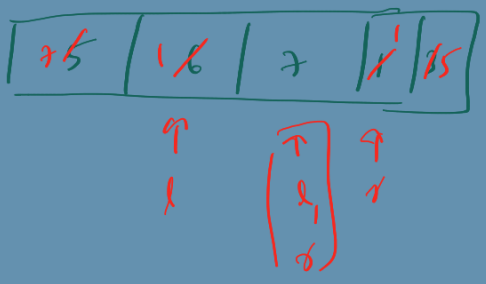
max = Elem.

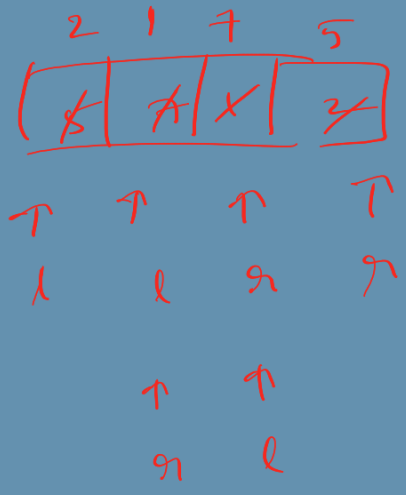
return max.

④ Reverse Array. O(n,1)



while l < r:
swap (nums, left, right)

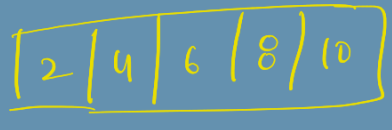




→ Pairs in an array

O(n², 1)

↳ Combination.



(2,4), (2,6), (2,8), (2,10)

(4,6), (4,8), (4,10)

(6,8), (6,10)

(8,10)

2 nested loops.
↳ O(n²).

i ⇒ 0 → n-2

j ⇒ ↳ i+1 → n-1

Total possible soln s ⇒ $^{n}C_{2}$

→ Subarrays.

↳ continuous part of array

→ $O(n^2, 1)$

[2, 4, 6, 8, 10]

(2), (2,4), (2,4,6), (2,4,6,8), (2,4,6,8,10) - 5

4, (4,6), (4,6,8), (4,6,8,10) - 4

6, (6,8), (6,8,10) - 3

8, (8,10) - 2

10 - 1

Total sub arrays = $\frac{n(n+1)}{2}$

• Maximum index Difference. (GFG)

$O(n^2, 1)$

[34, 8, 10, 3] output = 1

→ conditions

↳ i <= j

↳ arr[i] <= arr[j].

Brute force : → $O(n^2)$

[34, 8, 10, 3]

i → 0 → n-1

j → i, n-1

if arr[i] <= arr[j] : ⇒ max diff = max(max diff, j - i)

i <= j

if arr[i] <= arr[j]

max diff = 0 1

i=0 { j=0 → 34,34 → 0
j=1 → 34,8
j=2 → 34,10
j=3 → 34,3

i=1 { j=1 8,8 → 0
j=2 8,10 → 1
j=3 8,3 → x

i=2 { 10,10 → 0
10,3 → x

i=3 x

→ By Sliding window Approach. O(n), O(n)

if condition satisfies → max window length / Expand it → j increase

else : decrease window size / Contract it by increasing i

→ conditions

① i <= j

② arr[i] <= arr[j]

{ 34, 8, 10, 3, 2, 80, 30, 33, 1 }

→ Consider,

[10 , 30
i=2, J=6 ⇒ diff = 6-2 = 4.

But how to check if arr[i] <= arr[j] ??

maintain an arr for it.

if we take min elem from Left to right our Ans will be Optimum.

We can use prefix & suffix arrays

↓ Left min

↓ Right max.

[8 , 30
i=1 , J=6 ⇒ diff = 6-1 = 5

Similarly if we take max elem from Right to left. it would be optimal. } maintain an array for it

[8, 33
1 7 ⇒ 7-1 = 6

→ 34, 8, 10, 3, 2, 80, 33, 1

Start filling from Left side choose the min elem.

Lmin = prefix arr = [34, 8, 8, 3, 2, 2, 2, 1]

Rmax = Suffix arr = [80, 80, 80, 80, 80, 80, 33, 1]

it goes like this.

Now we can use Sliding window approach.

Just loop through it.

```
ans = 0
while (i<n & j<n):
    if lmin[i] <= rmax[j]:
        ans = max(ans, j-i)
        j += 1
    else:
        i += 1
```

Dry Run For [34, 8, 10, 3]

lmin = [34, 8, 8, 3]

rmax = [34, 10, 10, 3]

} ⇒ 34,34 (0 0) ⇒ 34,10 (0 1) ⇒ 8,10 (1 1) ⇒ 8,10 (1 2)

ans = 1.

3,3 <= 10,3 <= 8,3 <= 8,10

3,3 2,3 1,3

Match Sticks Game:

→ 2 Players A & B → Match sticks.

→ N Match sticks.

→ The player can pick any number of match sticks from 1, 2, 3, 4 during their chance.

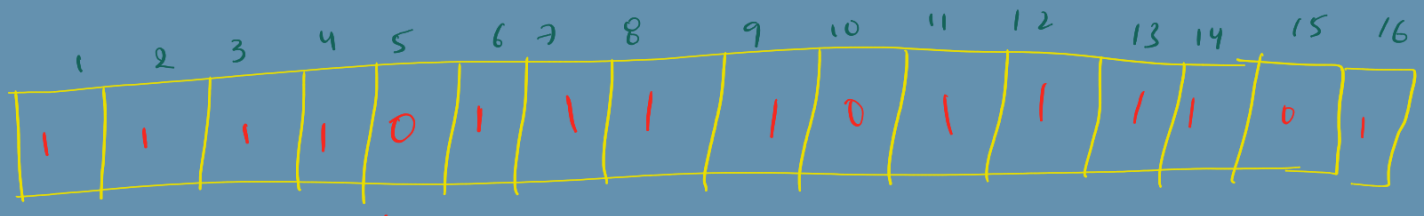
→ The person who takes last match stick wins the game ⇒

→ If A is starting first how many matchsticks he need to pick so that he can win the game.

Rough

① Winning state : State from which the current player can make
(1) a move such that he directly wins or makes a move such that the opponent gets a loosing state

② Losing state : State from where no matter whatever move
(0) you make the current player makes the opponent will always get a winning state.



As per losing state apply winning state part 2. (opponent gets loosing state)

=) second largest elem of array:

→ sort array then choose last 2nd elem if sorted in ascendg order.

Brute force → n log n.

→ Optimal Approach => 2 variable First, Second.

if no -ve nums in arr.

First, second = -1 (or int.min value)

```
for i in nums:
    if i > First:
        2nd = 1st
        1st = i
    elif i > 2nd:
        2nd = i
```

- Remove Duplicates from an array.

- if arr is sorted

[1, 1, 2, 3, 3]

check for before element.

⇒ idx = 1

↳ start looping from first index

```
for i in range (1,n):
    if arr[i-1] == arr[i]:
        pass
    else:
        arr[idx] = arr[i]
        idx += 1
```

O(n,1)

- if arr is not sorted.

↳ Set ⇒ O(n,n)

↳ default dict → order is preserved
↳ O(n,n).

↳ Sort ⇒ O(nlogn, 1)

Order is not preserved.

- Rotate Array - Left By 1 Step.

I/P = [10, 2, 5, 4, 2]

O/P = [2, 5, 4, 2, 10]

0 ← 1
1 ← 2
2 ← 3
⋮ ⋮
n-1 ← n

Approach ⇒

① Copy the first Element

temp = arr[0]

② loop trough arr till n-1.

not inclusive

```
for i in range(0, n-1):
    arr[i] = arr[i+1]
```

③ Set the last element with temp.

arr[n-1] = temp

- Rotate array - left By K steps.

O(n+K, d).

arr => [1,2,3,4,5,6,7]

K => 3

o/p => 4,5,6,7,

① Brute force.

↳ Store first 3 elements in temp array.

↳ shift the elements ie, from K to n-1
[K, n)

↳ Put back those temp
from [n-K to n)

i = n-K ↑ bau index => if we Subtract them from => i-(n-K)
0,1,2.

or
we can use the j=0
& increment it as we go

optimized Code:

n=7,
d=3.

[1,2,3,4,5,6,7]
d | n-d

① reverse → 4,5,6,7,1,2,3.

② reverse → 7654

3,2,1

3 2 1 7 6 5 4

③ reverse it again => 4 5 6 7 1 2 3. ✓

→ similar Approach for right rotate By 1 & K.

- Rotate Array in Groups.

arr = [1, 2, 3, 4, 5] => 5

K = 3

O/P = [3, 2, 1, 5, 4]

for i = 0 ; i < n ; i = i + K

start = i

end = min (i + K - 1 , n - 1)

For getting the last index.

reverse (arr, start, end)

while start < end :

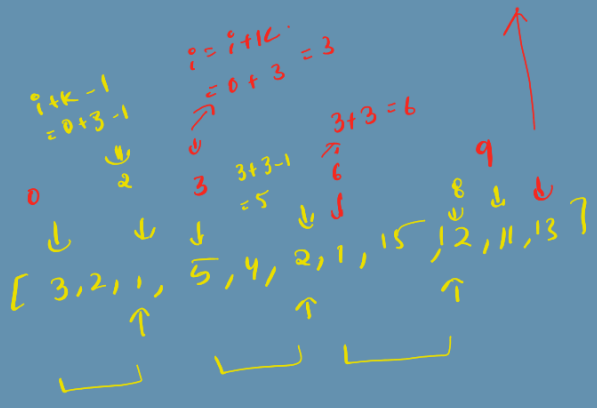
swap (arr, start, end)

start += 1

end += 1

10 = min(11, 10)

min(9+3-1, 10)



end = i + K - 1

↳ This works fine if the len is divisible by K.

i.e. multiple

- check if array is Sorted & Rotated.

Pivot point = 0

arr ⇒ [3, 4, 5, 1, 2]

→ For increasing order ⇒ arr[n-1] > arr[0]

Case 1 ⇒ Sorted array ⇒ [1, 2, 3, 4, 5] ⇒ arr[i] > arr[i+1] ⇒ Pivot += 1

Case 2 ⇒ Sorted & Rotated ⇒ [3, 4, 5, 1, 2]

i i+1

Case 3 ⇒ Not sorted / Rotated ⇒ [3, 5, 7, 1, 6]. check at last ⇒

Case 4 ⇒ if all elements are equal ⇒ [1, 1, 1, 1] ⇒ pivot = 0 ✓ accept

→ For decreasing order (descending order)

① ⇒ [5, 4, 3, 2, 1]

② ⇒ [3, 2, 1, 4, 5]

③ ⇒

if arr[i] < arr[i+1] :

Pivot += 1.

- Move zeros to End.

10, 15, 0, 20, 0, 25

↓

10, 15, 20, 25, 0, 0

→ Brute force

- use the temp array add elements to it if non zero.

after adding add those 0 at the end.

→ loop till n from size of temp array ⇒ -1.

→ optimal

20 0

[10, 15, 0, 20, 0, 25]

↑ i ↑ j

ii) ↑ ↑ i

if arr[i] != 0 :

i += 1.

j += 1.

if arr[j] == 0 :

j += 1

elif arr[j] != 0 :

Swap (arr[i], arr[j])

i += 1

j += 1