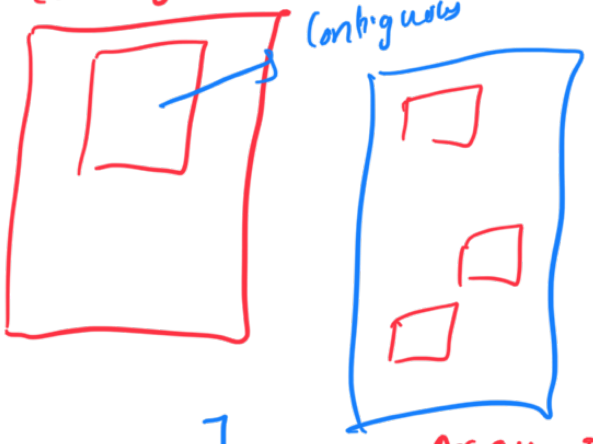
# Arrays + Dynamic Arrays

**What** is an array?

A collection of the same type of data stored in contiguous memory locations



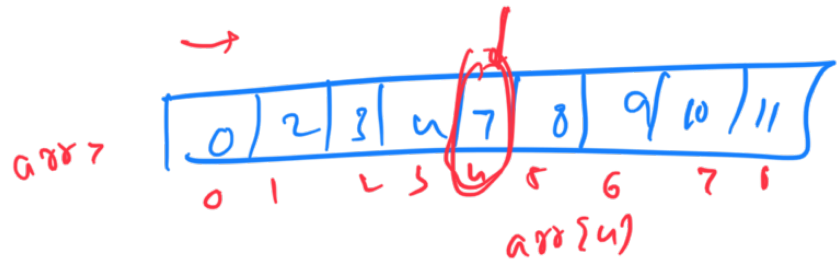

Ayy = [ . . . ] 15ft ⇒ Array of Integers

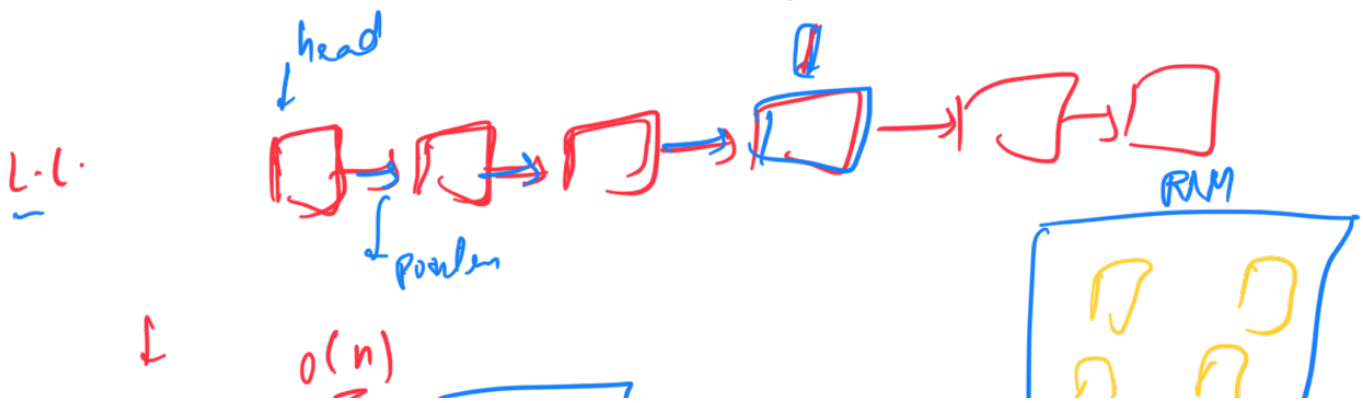
Heights / Weights = [ ] ⇒ Arrays of float/double

Standout Points

1) Random Access ⇒

2) O(1) look up time



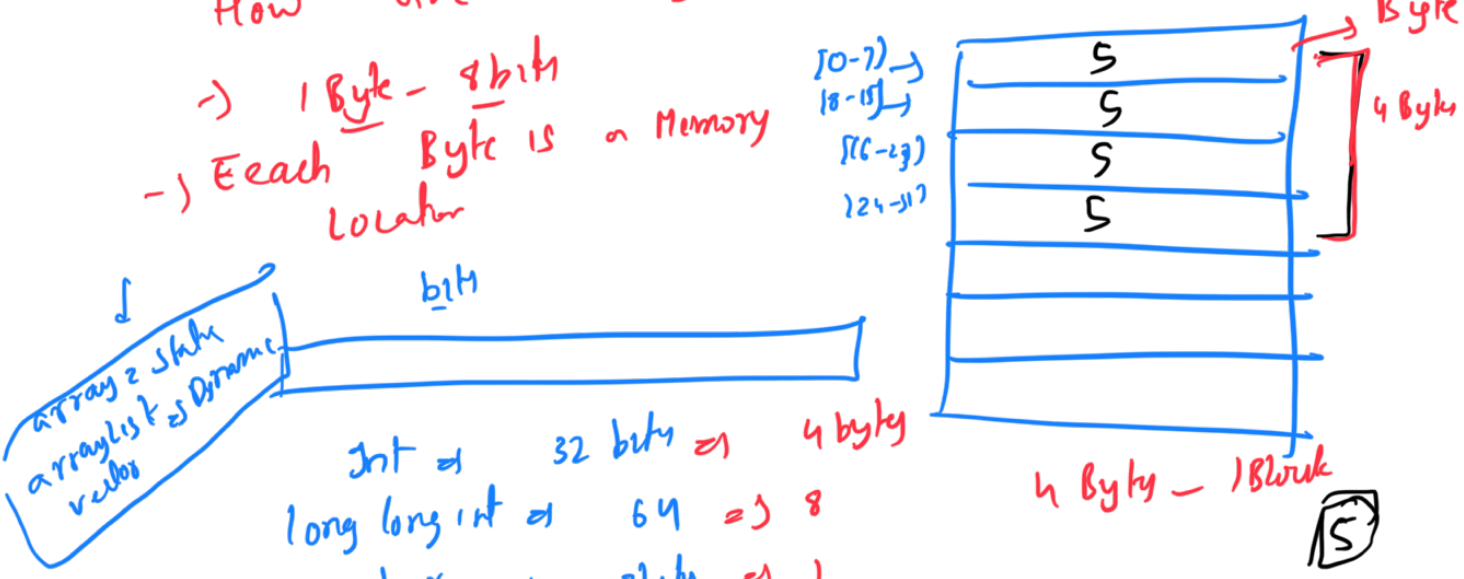

↓ O(n)

O(1) Lookup ⇒ Contiguous Memory

How are array elements stored in RAM

-) 1 Byte - 8 bits

-) Each Byte is a Memory Location




Int ⇒ 32 bits ⇒ 4 bytes

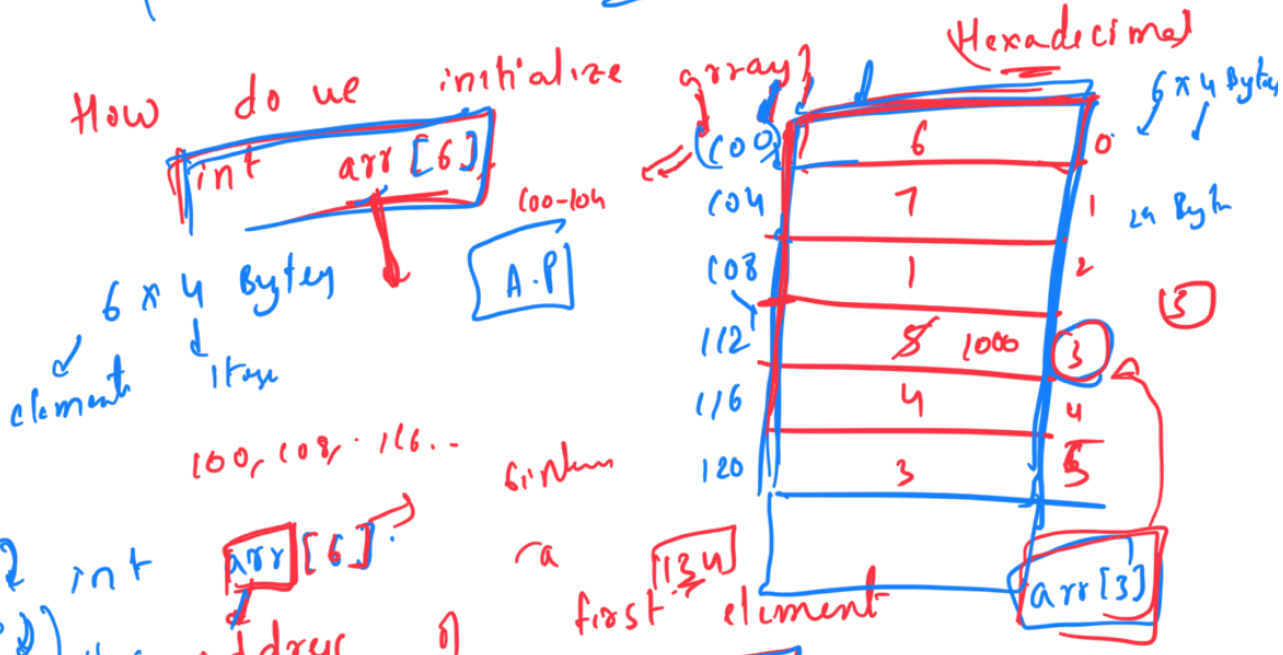
long long int ⇒ 64 ⇒ 8

char ⇒ 8 bits ⇒ 1

Computers 0's / 1's

(0 0 0 0 0 0 0 0 1 0 1) ⇒ 32 bit

How do we initialize array?




arr[6] ⇒ address of first element

100 + i * 4

base address

address = 112

(100 + 4 * 3) → O(1)

arr[i] =

address =

arr + i * (space per index) => ith element location

arr

Out of Bound Error)

T.C:

100 + i * (1 (char)) => O(1) => long long int (8)

T.C: O(1)

Random Access

T.C: ith Position

arr[i] = 1000 => O(1)

(arr + i * i) =>

arr[index]

address = arr + i * index
Ready the data
Return the val

(arr + i * i)

Array

Array

[0 .... n-1]

Question: Nearly Sorted Array

(N-1) numbers are sorted.

Sort the array

arr

1 2 3 6 8 13 20 24 27 14

0 1 2 3

Binary Search

log(n)

45% O(log n) => 14 [Related]

Naive Way:

Sort (O(N²) => O(N)

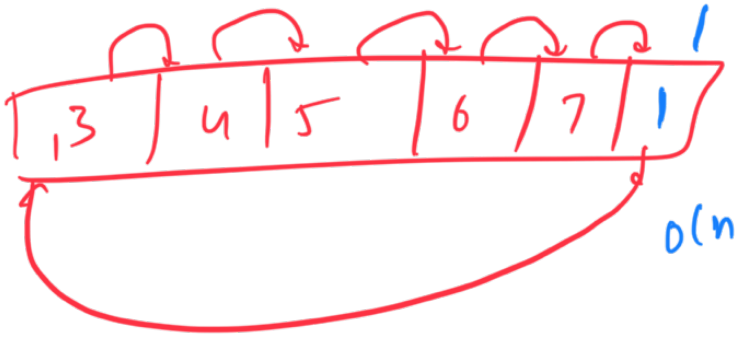
Insertion ... Merge Sort / Q sort = O(NlogN)

Why log(N)? Binary Search – Sorted Arr (Binary Lectu

2 steps

1) Find Correct Position ⇒ O(n) ⇒ O(logn)

2) Put it in the correct location.




O(n) operation

T.C: O(logn) + O(n) (nlogn)

T.C: O(n)

$N \to \frac{N}{2} \to \frac{N}{4} \to \frac{N}{8} \cdots$

$\frac{N}{2^0}$ $\frac{N}{2^1}$ $\frac{N}{2^2}$ $\frac{N}{2^3}$ ... $\frac{N}{2^k}$

K steps.

$\frac{N}{2^k} = 1$

$N = 2^k$

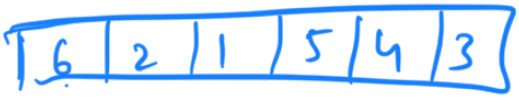
$\log N = \log(2^k)$

$\log N = k \log 2 \Rightarrow 1$

$k = \log N$

O(logn) + O(n) ⇒ O(n)

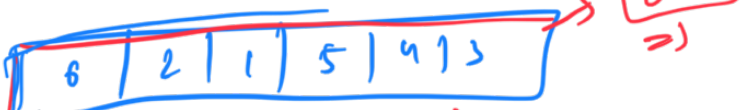
**Question:** Reverse the array


for(i=n-1 to 0)

**Naive Approach:**

→ Take a new Array B



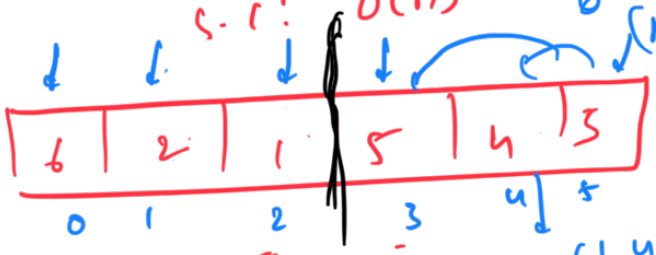

$B_i = n-1-i$

T.C: O(n)

S.C: O(n)

for(i=0; i<5; i++)

N=6



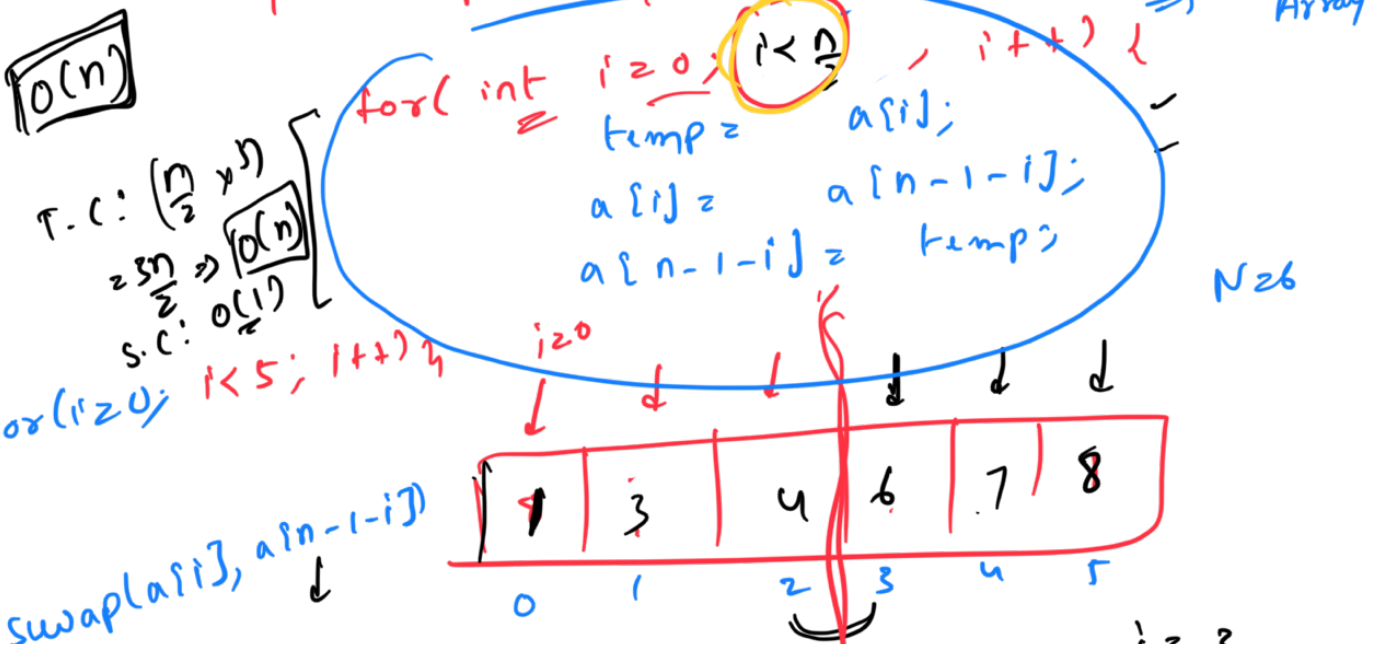

i = 3

n-1-i = 6-1-3 = 2

| i | n-1 | (n-1) | (n-1-1) | (n-1-2) |
|---|---|---|---|---|
| 0 | n-1 | (n-1) | (n-1-1) | (n-1-2) |
| 1 | n-1-1 | | | |
| 2 | n-1-2 | | | |
| i | n-1-i | | | |

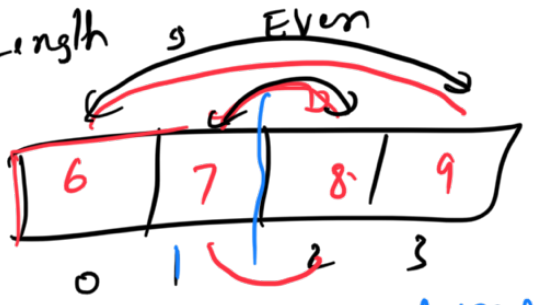
(1, 4)

n-1-1



$i < \frac{n}{2}$ , $i \le \frac{n}{2}$

Case 1: Length is Even




(i = 1 dr mano)

$i \le \frac{n}{2}$

N = 4

i <= 2

i = 2

n - 1 - i =

6 - 1 - 3 = 2

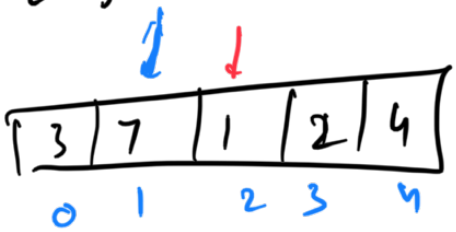
swap (3, 2)

N = 4 => 2

$i \le \frac{n}{2}$

i = 1

Case 2: Length is odd




N = 5

N = 7

i = 2

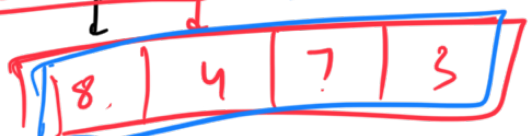
swap (i, n-1-i)

i = 2

n - 1 - i = 5 - 1 - 2 = 2

i = 0, 1, 2

orig = 3 4 7 8 => 8 7 4 3




$i \le \frac{n}{2}$

for (i = 0; i <= 2; i++)

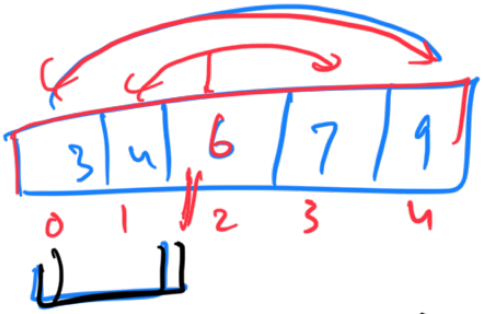
N = 4

i = 0

i = 1

i = 2

i = 3




N = 5

$i = \frac{n}{2}$

swap (i, n-1-i)

i = 2

$i = \frac{n}{2}$

$i = \frac{5}{2}$ = 2

$i = \frac{n}{2}$

T.C:

A = [1, 2, 3, 4, 5, 6]

n - 1 - i = 5 - 1 - 2 = 2

swap(2, 2)

i < n/2

## Subarray

A contiguous part of an array.

Array within array

[3, 4, 5] ✓

[2, 4, 5] ✗

[ ] ✓

[3, 6, 4, 1] ✗

## Subsequence

sequence that can be derived by deleting some elements.

[3, 4, 5] ✓

[2, 4, 5] ✓

[ ] ✓

[3, 6, 4, 1] ✗

## Subset

Any combination of the array elements

→ order does not matter

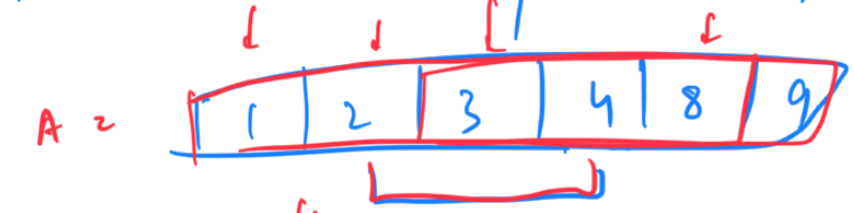
→ No Duplicates

[3, 4, 5] ✓

[2, 4, 5] ✓

[ ] ✓

[3, 6, 4, 1] ✓




[2, 3, 4]

[1, 3, 4]

[4, 3, 2] ✗


{2, 4, 9} ≠ {4, 2, 9}

(order Matters)

A = [1 2 4 3 6 5 7 6]

[1 6 7 6] ✗ Subarray ✗ subseq ✓ subset

{2, 4, 3, 5}

(A)

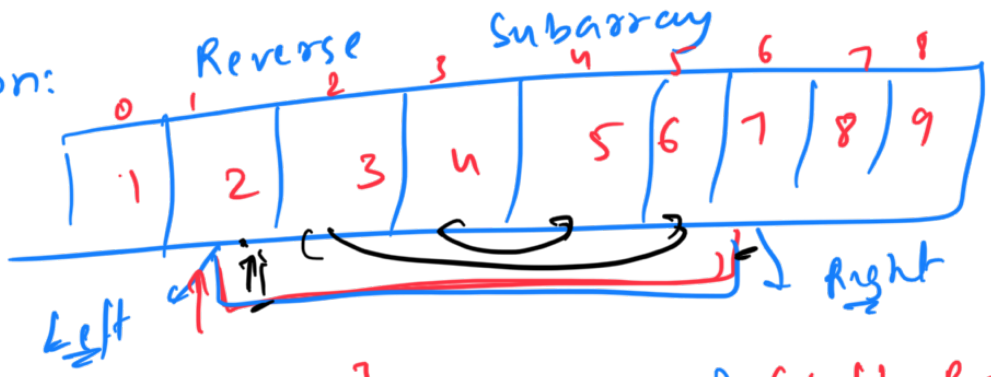
Every subsequence → Subset? NO

Every subarray → Subset? NO

→ Subsequence? YES

n

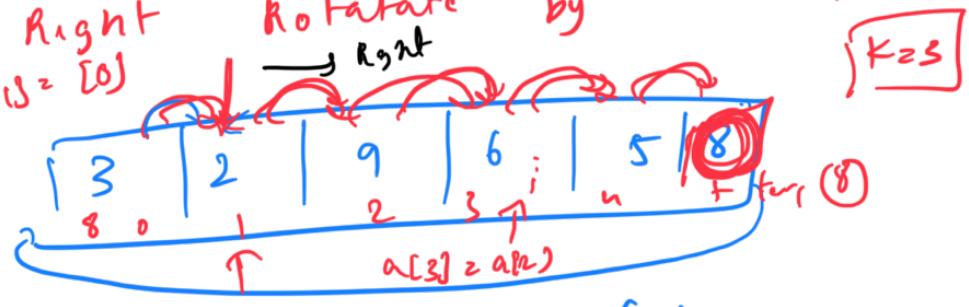
Question: Reverse Subarray




a[ 2 : 7 ]

1) (left, Right)
1) (left, length)

Question: Right Rotatate by k steps




1st steps : 8 3 2 9 6 5

2nd : 5 8 3 2 9 6

3rd 6 5 8 3 2 9

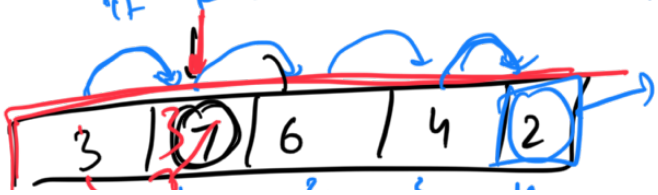
```
int shift (arr) {
    temp = arr[n-1];
    for(int i = i; i<=n-1; i++) {
        a[i] = a[i-1];
    }
    a[0] = temp;
}
```

2) Right Rotate Exactly one

2) O(n)

T.C: O(n x k) = O(n x k)

2) O(n²)



temp = a[n-1];

for (i = n-1; i ≥ 1 ; i--) {

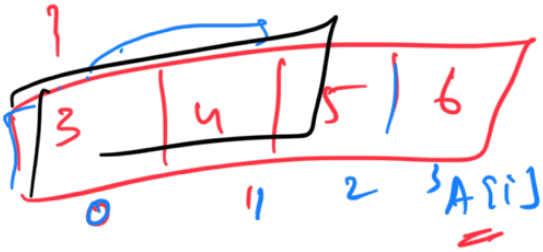
a[i] = a[i-1];

}

a[0] = temp;

i = 0

A =

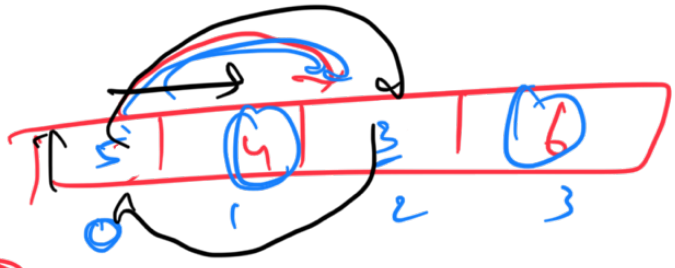



A[i] → A[(i+k)%N]

N = 4

K = 2

temp = 5




temp = 3

(2+2)%N

(4)%4 = 0

(i+k)%N

i = 0

(0+2)%N = 2

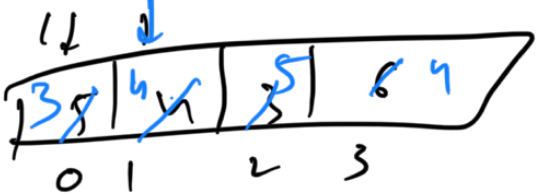
gcd(4,2) = 2

for (i = 0; i < gcd(n,k); i++) {

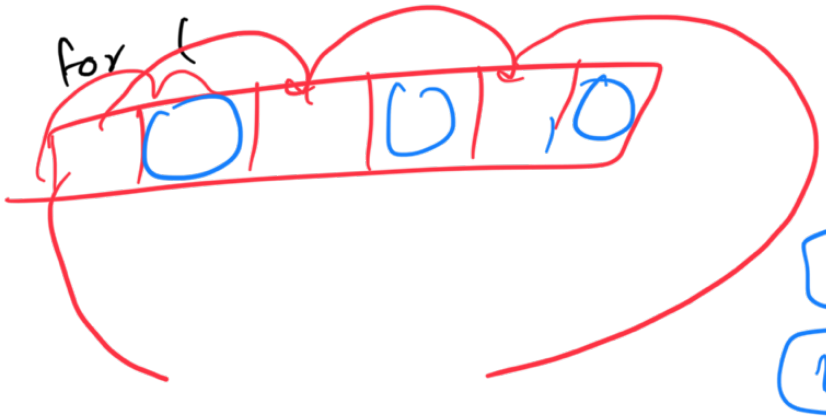
i = 0

i = 1

for (int i = 0; i < k; i++)




for (



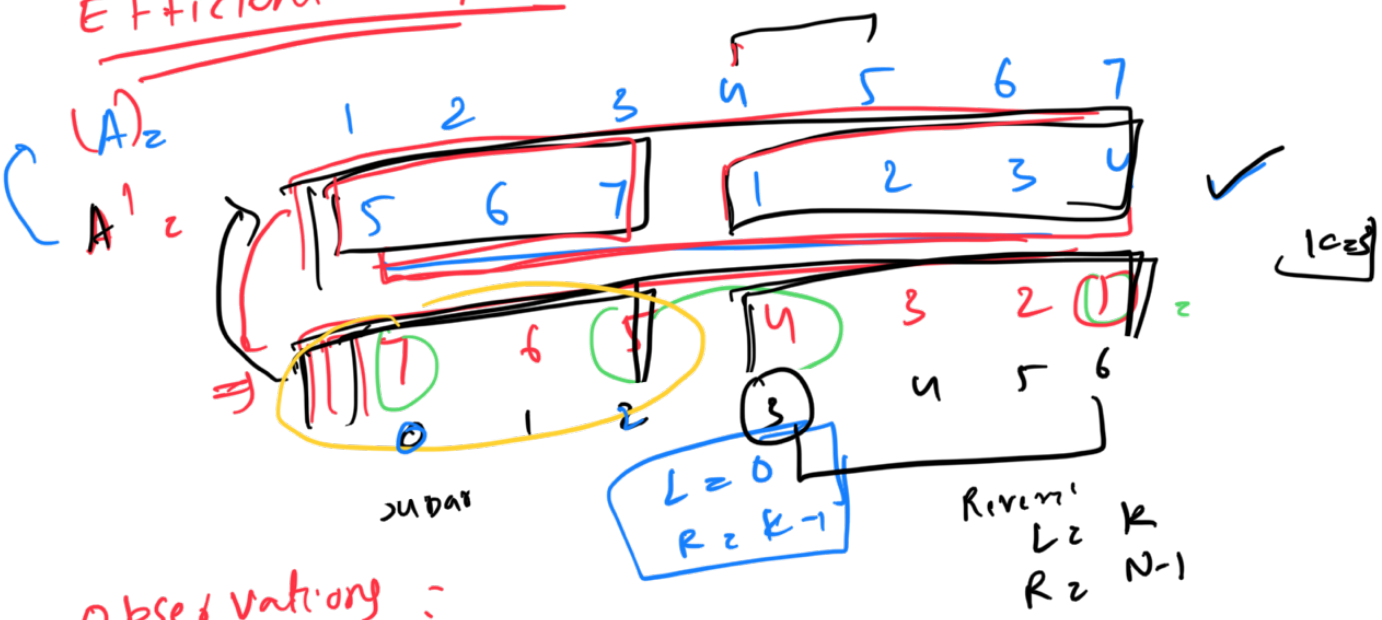
k = n

k%n

k = n

k%n

# Efficient Approach




## Observations :

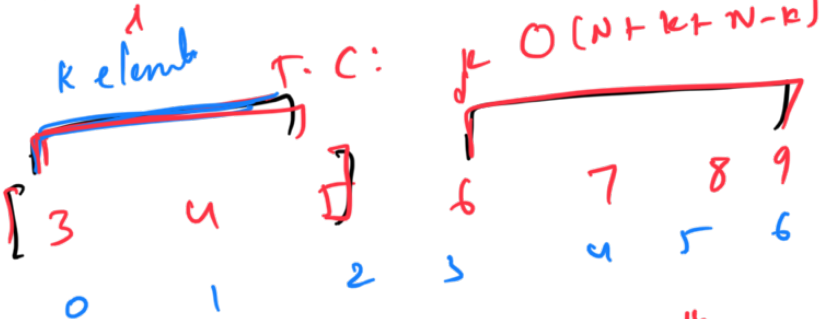
1) (4, 5) were beside each other, Now they are pole apart

2) (1, 7) were poles apart, now they are beside.

Step 1.

1) Reverse (A) — O(N)
2) Reverse (0, K-1) ✓ O(K)
3) Reverse (K, N-1) O(N-K)

O(N + K + N-K) ⇒ O(2N) ⇒ O(N)

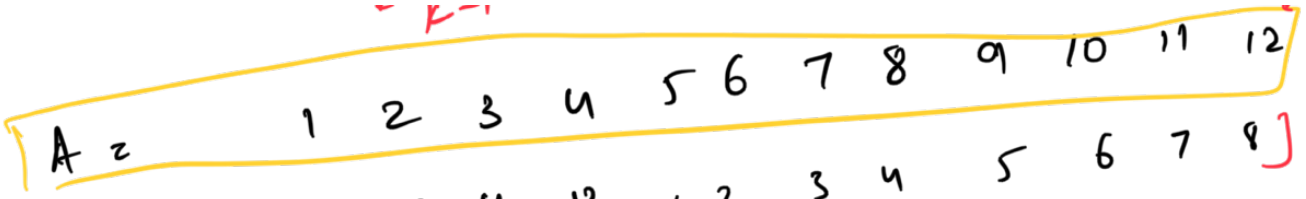



Index of $K^{th}$ element = K-1

$\begin{bmatrix} L = 0 \\ R = 2 \end{bmatrix}$

sub

K=4




Rev(A):

Rw(0, K-1)

(K = N)

K ≥ N

K = K % N

## Limitations of Array :

1) Size of the array is not flexible (static) we need to fix the size when we define it.

arr[ ]

arr[365]:


When arrays?

1) Size of dataset is known

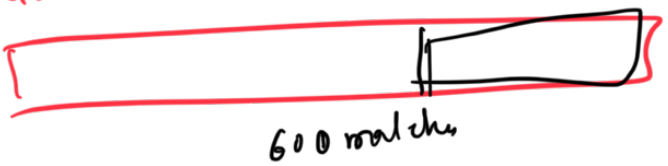
2) If we know a realistic estimate of the upper limit.

Virat Runs : [ 10, 47, 0, . . . . . ]

Sachin = 460 matches

2 * 460

arr[1000]



600 matches

arr[600]

arr[1000]

(400) * 4 Bytes

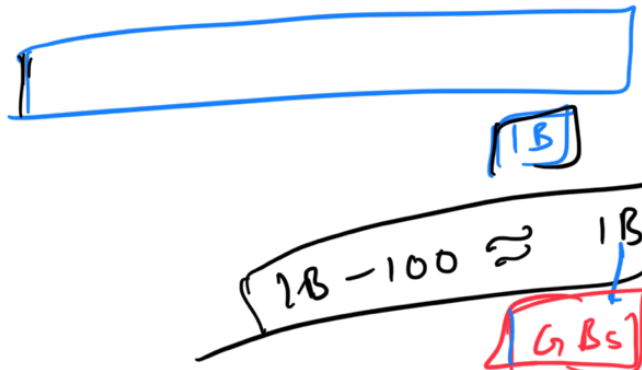
1600 Bytes

Scaler Youtube channel videos viewers

→ store names of my viewers

→ In last 3-5 videos, 10k max view

(Viral videos)


16 GB RAM

Photon Unity

Dynamic Arrays

→ Flexibility on Sizes

→ On Demand expansion

Python/R

C++ : Vector. ⇒ push_back

Java : ArrayList ⇒ add

List : Python ⇒ append

⇒ Insertion in Middle / ⇒ O(1) Linked

Deletion

---

Question : Design a Data Structure

→ O(1) random access

→ Flexibility in size

You don't have vectors / arraylist.

(Arrays)

arr[size]

ith (arr + i) ⇒

2) Contiguous Memory (Arrays)

2) Flexibility (X Array)

Virat kohli (500 matches)

arr[500];

501th Match

→ Create an array of size 501

→ (502nd Match))

Create arr[502]; // copy

size' = size + 1

T.C of Inserting N elements.

Slow

[ ]

a.push(1) → 1

a.push(2) → 2

a.push(3) → 3

a.push(N) → N


Cost for N elements

(1 + 2 + 3 + 4 + .. N) :

$\frac{n(n+1)}{2} \Rightarrow O(\frac{n^2}{2})$

→ Random Access — O(1)

→ Flexibility in Size

Insertion: O(n) → 1 element, N element

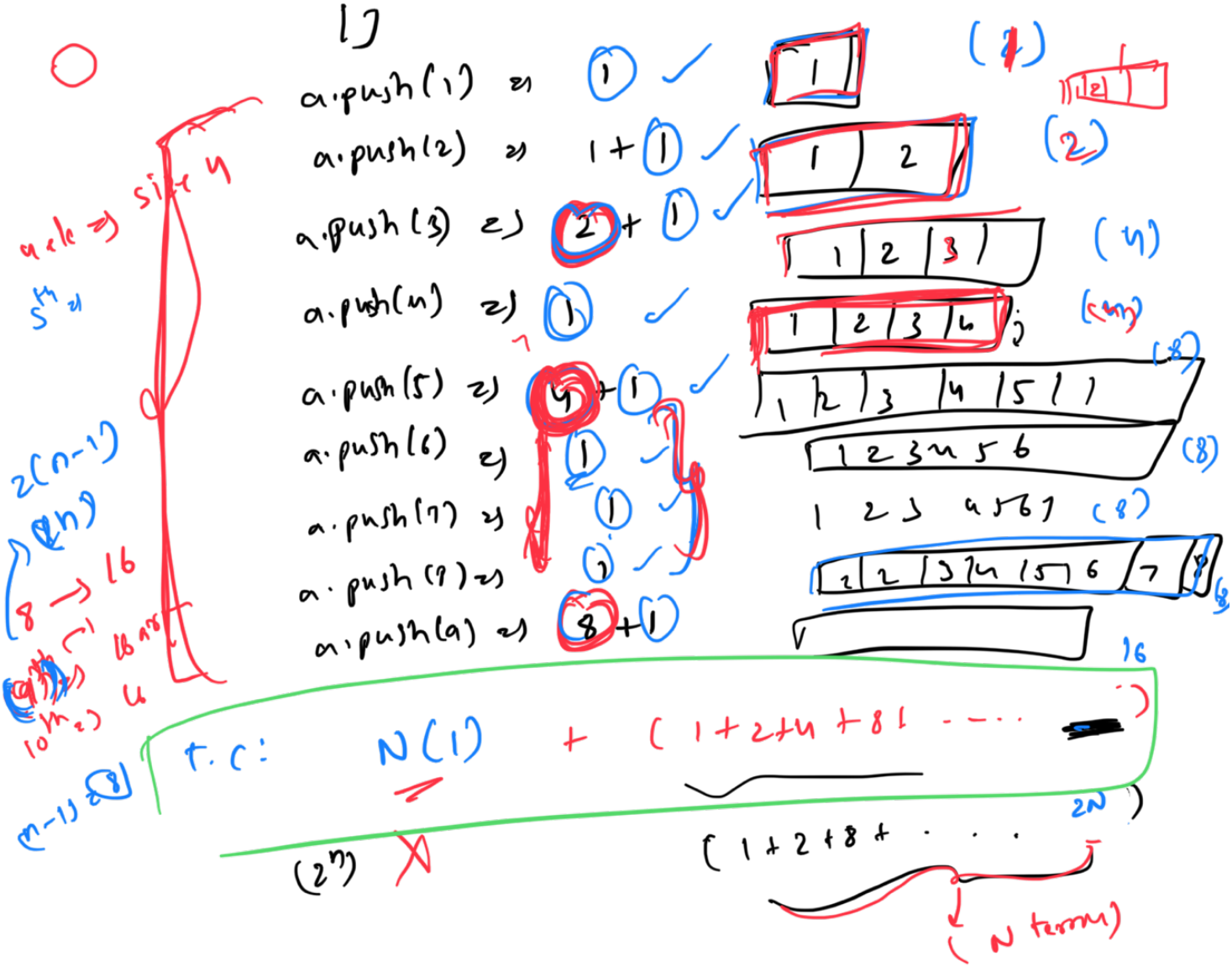
Strategy

new_size = [size+1]

2*size)

T.C of N insertions

[ ]

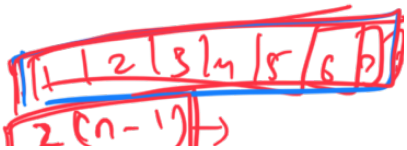


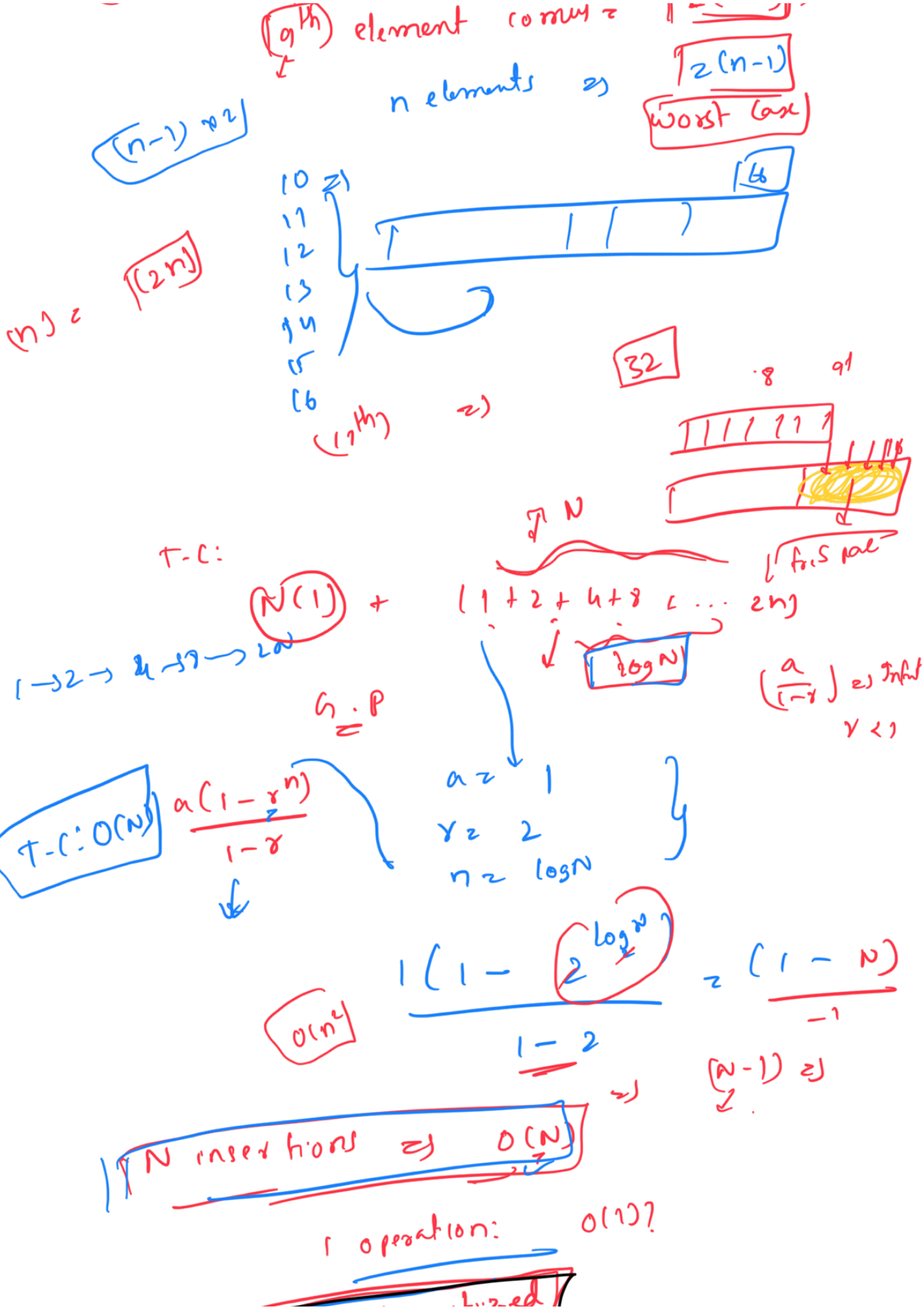

(For N elements, what is the max size of array at any point.)

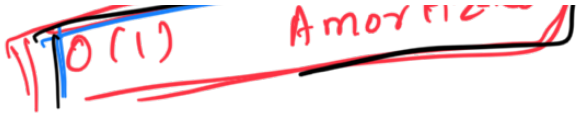
$n^{th}$ element => Size was full

$s^{th}$ element comes => (size = 4)

$2^n$ ((2*4)) => 8




9th element
n elements
2(n-1)
Worst case
(n-1) x 2
(2n)
32
T.C:
N(1)
1+2+4+8 ... 2n
logN
N
first pair
G.P
a = 1
r = 2
n = logN
T.C: O(N)
a(1-r^n)/(1-r)
O(n^2)
1(1-2^logN)/(1-2) = (1-N)/-1
(N-1)
N insertions = O(N)
1 operation: O(1)?

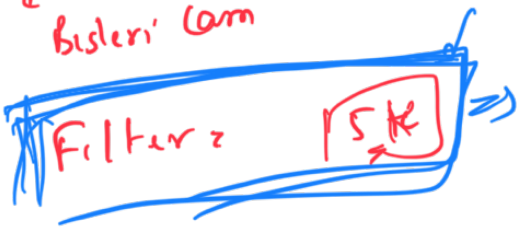



**Amortized:** We gradually write off the cost of an asset over a period of time.

Bigger Investments Now ⇒ Save costs in the Future



Bisleri Cam

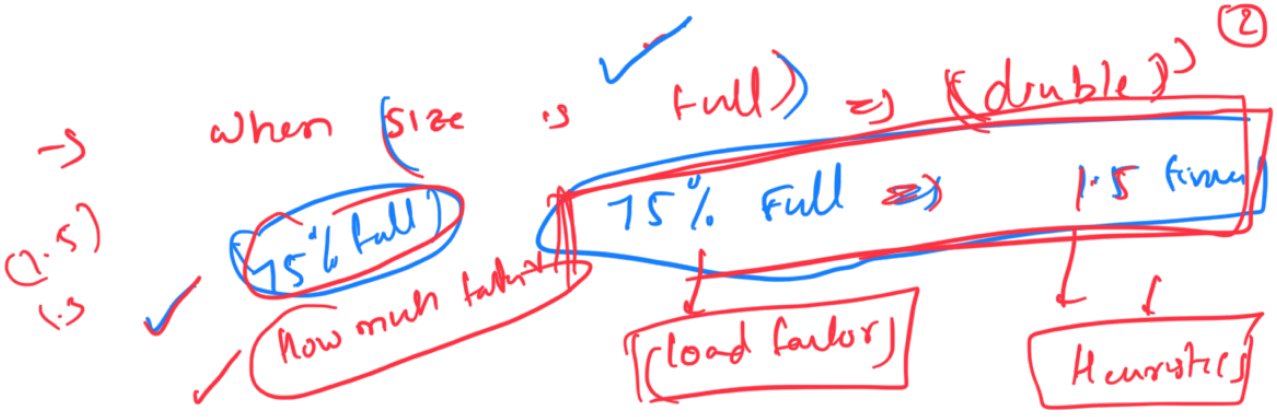



After 2 years

(Rented House) / EMI

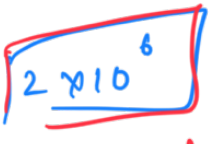
(30K) (30K)) for 10 year

Investment ⇒ Double the Size !!





⇒

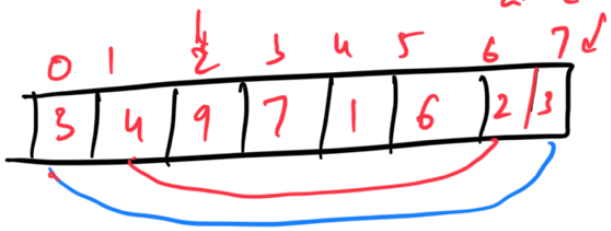



=

(Huge wastage)

(1.5)
(1.4)

O(1))




(n elements)

(n-1) -1

(n-1)

0 ⟶ n-1

1 ⟶ (n-1) -1

2 ⟶ (n-1) -2

i ⟶ (n-1) -1

k steps

(No. of steps = k)

n

$\frac{n}{2}$ $\frac{n}{4}\left(\frac{n}{2^2}\right)$ $\frac{n}{8}\left(\frac{n}{2^3}\right)$ $\left(\frac{n}{2^k}\right)^1$

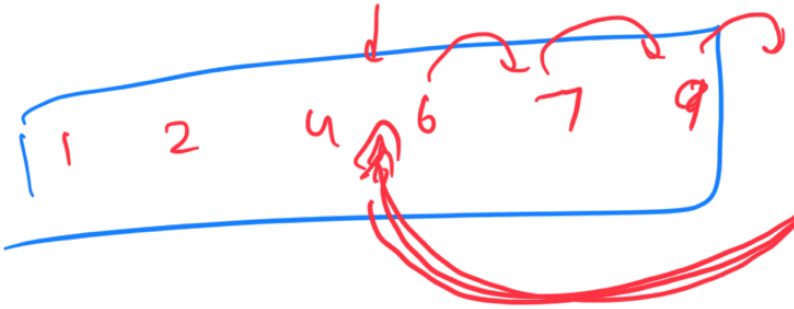
kth

1st 2nd 3rd . . . . . kth step

$\frac{n}{2^k} = 1$

$n = 2^k$

$\log n = k$




O(n)

push back

O(n) space

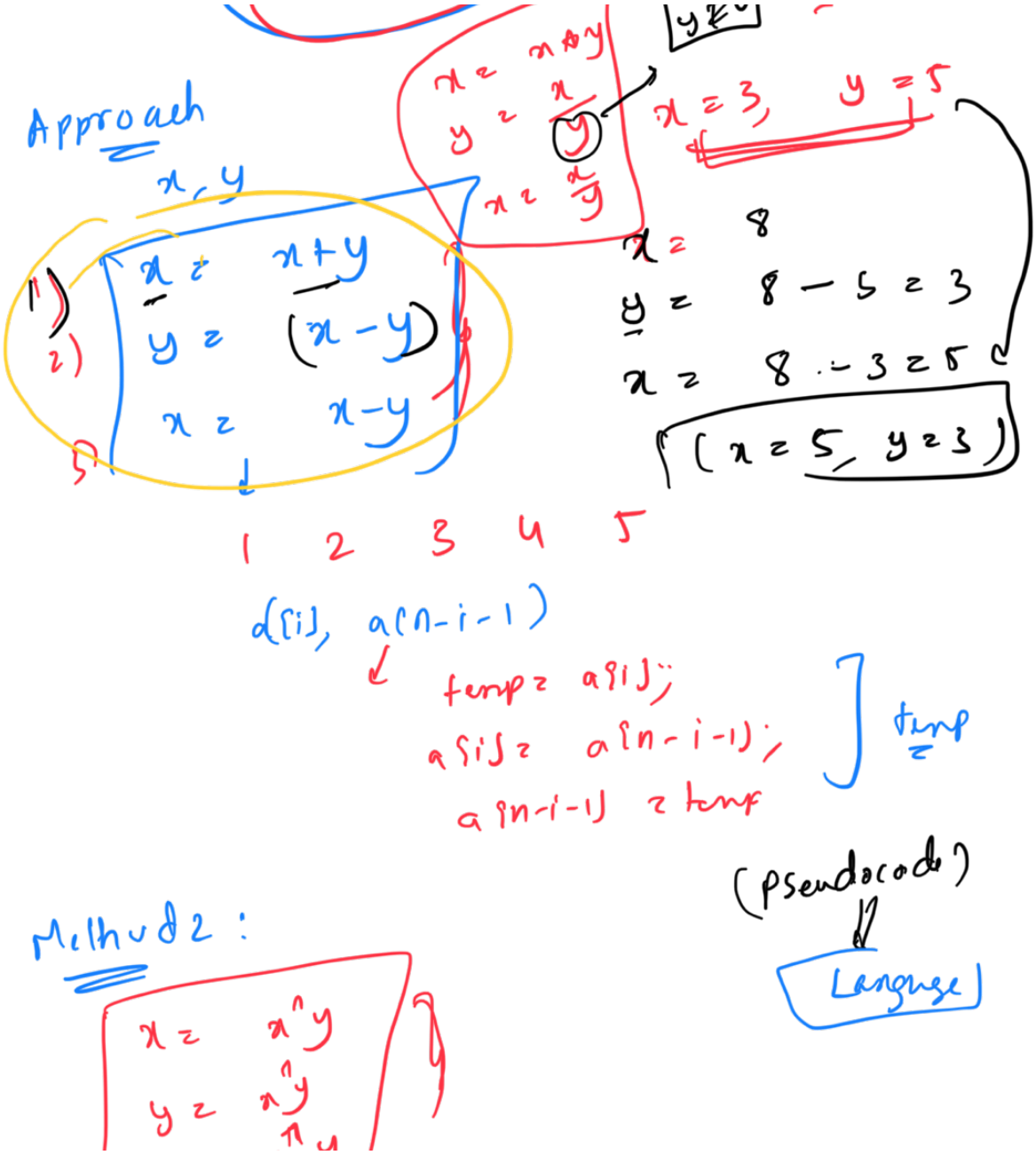
n element =)

(gcd) _ LCM

gcd(2,6) =

gcd, LCM

(x = 5, y = 5)


Approach
x, y
x = x+y
y = (x-y)
x = x-y
1)
2)
3)
x = x+y
y = x
x = x
x = 3, y = 5
x = 8
y = 8 - 5 = 3
x = 8 - 3 = 5
(x = 5, y = 3)
1 2 3 4 5
a[i], a[n-i-1]
temp = a[i];
a[i] = a[n-i-1];
a[n-i-1] = temp
temp
(pseudocode)
Language
Method 2:
x = x^y
y = x^y

**Sort a Nearly Sorted Array:**

```
int getPosition(int low, high, int last_ele){
	int ind;
	while(low <= high){
		int mid = (high + low)/2;
		if(a[mid] < last_ele){
			low = mid + 1;
		}
		else{
			ind = mid;
			high = mid - 1;
		}
	}
	return ind;
}

sortArray(int arr[], int n){
	last_ele = arr[n-1];
	correct_pos = getPosition(0, n-2, last_ele);
	for(int i = n-1; i > correct_pos; i--){
		a[i] = a[i-1];
	}
	a[correct_pos] = last_ele;
	return;
}
```

**Reverse an array**

```
for(int i = 0; i < n/2; i ++){
	temp = a[i];
	a[i] = a[n-1-i];
	a[n-1-i] = temp;
}
```

**Reverse a subarray**

```
for(int i = L; i < (L+R)/2; i++){
	temp = a[i];
	a[i] = a[R - (i - L)];
	a[R - (i - L)] = temp;
}
```