Logarithmic Complexity. $\log_b(n) \Leftrightarrow y$

If & only if $\hookrightarrow b^y = n$.

in D·S·A if Time Complexity $\Rightarrow \log_2(1) = 0$

↳ by default it is 2.

as $2^0 = 1$

Base. while Considering Time Complexity.

$\log(4) \Rightarrow \log_2^4 \Rightarrow 2$

↳ by default

what impact does it have

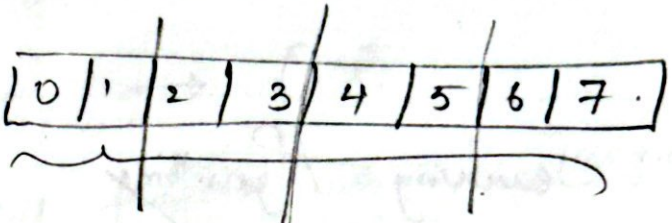
↳ Searching

Linear → $O(n)$

Binary → $O(\log n) \Rightarrow O(\log_2^n)$.

Why it is Always 2 Not 10?

Binay Search →



if we want to Search for element of array size 8. we need to Make 3 cuts

lets → 0 → half way → $\frac{\text{halfway}}{2}$ → 0 | 1 → find it from there.

So 8 → 3 cuts

→ $\log(_2^8)$ *

So it is Always 2 Not 10

if we take 16 length → 0 to 15

16 → 4 cuts

$\log(_2^{16}) \rightarrow 4$.

Numbers increasing

As value of N increases.

$\log_2(N)$ → is also increasing.

8 → 3 Cuts.

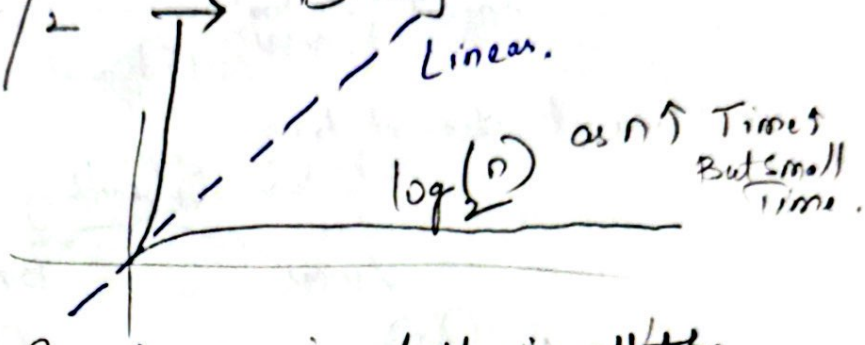
16 → 4

$\log_2^n$ is also increasing only by a tiny amount.

* in Linear O(n) as n values increases → Search space also increases
* but in binary n → increases search space ↑'s very Tiny

$n/2 \rightarrow n/2/2 \rightarrow n/2/2/2 \rightarrow$ Binary Search.

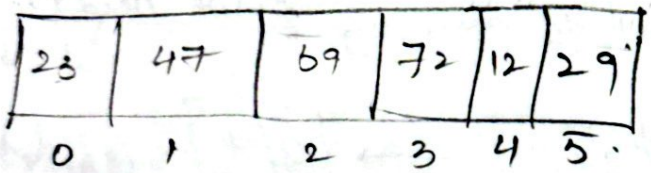
We can say it on decimal Graph



i.e why Logarithemic Time Complexity is better in all the Scenarios

---

Searching Algorithms



arr[3] = 72

let's we say if we find to 12 Target = 12 & we want to return index as 4!

if Target = 40 → Not available return → -1.

Target = 47	Target = 70
Output = 1	output = -1.

Naive Approach / Brute force ✱ ✱ (2)

iterate through Array if Element found → Target == Array[i] return i.

```
for i in range(n)
    if (Target == arr[i])
        return i
```

arr[i] == Target

O(n). as we need to Traverse entire list

else return -1 → if Not found. (just return -1)

(2) Approach (2) → in python array == List.

We can directly use → `List.index(element)`

O(n) → in worst case if element is at last then Traverse entire List which is O(n)

(2) List.index(element) → O(1)

↓ Pre-defined function which is internally using the Searching algo Only.

Unsorted Array → Linear Search → O(n)

If it is a unsorted Array Index() will be using the Linear Search Algorithm Only.

So O(n) in the 2nd Case Algo.

arr.sort() → also internally use the function but → O(n) ✓

Always use the Original Approach ⇒ don't use the direct Approach.

✱ Don't use any In-built method

if Sorting Or Searching is an intermediate part of Question. Then use inbuilt function

Space Complexity in Linear Search.

O(1) => Constant => as we didn't use any extra space.

#Code → O(1) > O(log n) > O(n log n) > O(n) > O(n²) > O(n³) ...

Constant Logarithmic Linear Quadratic

Binary Search → O(log₂n).

Practical Use Case → B.S → best.

1st → Linear Search Code:

```
# Driver Code:
arr = [12, 14, 11, 29, 31, 45, 62]
target = 31
result = LinearSearch(arr, target)
```

Logical Part.

```
# Function Definition.
def LinearSearch(arr, target):
    n = len(arr)
    for i in range(n):
        if arr[i] == target:
            return i
    else
        return -1
```

Time Complexity: O(n)

Space Complexity: O(1).

Output as 4

Binary Search: {important for Interview}.

↳ Will works only if we have Sorted data (Ascending Or Descending).

time Complexity → O(log n).

Why Only Sorted Array ? why not Unsorted Array?

2	4	8	12	27	45	49	52
0	1	2	3	4	5	6	7.

let's say Target = 45.

Steps:

→ Try to find middle index value

Start = 0, End = 7 ⇒ $\frac{0+7}{2}$ = 3.5 ⇒ 3.

→ Go to Middle element & Cross check it with Target-Element.

while (start <= end):

if arr[mid] == target

return mid.

mid = start + end // 2.

Decimal.

else if arr[mid] < target : → 12 < 45

deciding Left or Right → if & only if Sorted Array.

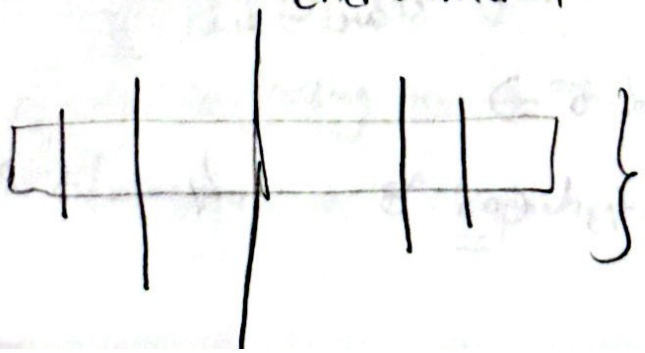
As 12 < 45 so right side.

start = mid + 1 {change start index.}

We reduced the Search Space.

else:

end = mid - 1 { if target < arr[mid] }.



} divide the Array to Equal halves Everytime.

while (start <= end) intution Behind this ?

arr = [2, 4, 6, 8, 10] , target = 6.

start = 0, end = 5 , Mid = 2.

(0 <= 5)

if (arr[mid] == target)

return mid.

6 == 6. (case 1)

target = 7

(0 <= 5)

if (arr[mid] == target).

return mid

if (arr[mid] < target).

start = mid + 1

*.

else

end = mid - 1.

1st iteration start = 0, end = 5

2nd => start = 3, end = 5 => 3 <= 5 => pass in.

as arr[mid] = arr[4] = 8 < target (7).

mid = 4, end = 5.

8 > 7

start = 4 + 1 = 5

next iter => start = 5, end = 5 =>

in Next iter -> start = 4, end = 3

4 <= 3 X.

2	4	6	8	10

Target = 7.

Start = 0, End = 4, mid = 2. ⟶ (1) 0 ≤ 4

after 1st → start = mid+1 = 3, end = 4.

after 2nd. start = 3, end = 4., mid = $\frac{3+4}{2}$ = 3.5 = 3.

Start = 3, end = 4, mid = 3. ⟶ (2) 3 ≤ 3.

after 3rd start = 4, end = 4, mid = 4.

→ end = 4−1 = 3. (mid − 1)

4th ⇒ end = 3, start = 4, mid = 3 4 ≤ 3 ✗ false

iter.

Loop Breaks.

Start = Mid+1 & end = Mid−1

if condition is Met Loop breaks.

Calculating Length of array → what is Space Complexity & Time.

as we know the Memory knows start & last index - & Memory

Address, so it is Constant → for Time Complexity).

Lambda, Regex, Patterns, opical, Gemin, openai, Pandas < All Methods

file handelling in python, join Method., apply Method.

Api calls → Parsing How → Thread (Multi-Thread)

List Expression, Comprehension.

Binary Search.

Middle = start + end/2 , but optimistic approach is

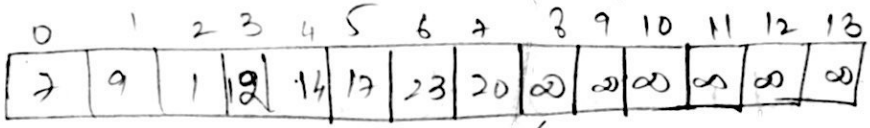
Middle = start + (start - end)//2.

By default python will consider the start, end numbers as integers, Every Number has Certain range by default. if it is Not able to do if we add directly.

* so we subtract Now divide Then add. `(start - end)//2`

To Avoid floating Error!! (As addition Value is Very Very Big)

Binary Search ⟶ Should have Sorted Data! Why. as to Get idea whether to Go Left or Right side of Mid



* let's Consider an Array of n=14 if a interviewer Asks the position of First infinite (initially some values are Numeric)

dp = 8 ⟶ How do we solve

Brute Force.

1) Go for a Linear Search

```
for i in range(n):
    if (arr[i] == ∞)
        return index
    else
        return -1
```

Complexity → O(n)

Can we reduce Time Complexity? Binary Search

1) it Must be Sorted.

So if we Go for Merge Sort => nlogn, next Binary Search => logn

nlogn + logn => nlogn >> n.

So in Binary Approach we find for a Target Right, here Our target is fixed which is infinite. So just find for the element (Mid) which is ∞

Sorted Array is only Helping us to whether we should go Left or Right:

Start = 0, End = 13 $\quad 0 + (13-0)//2 = 6.$

arr[6] == ∞ → No. (should we Go Towards the Left or right) → right side as we have stated that we have Numbers on left.

So start = Mid + 1 ⇒ 7

~~arr[7]~~ Mid = 7 + (13-7)//2 = 10.

arr[10] == ∞ { But we want the first infinite }

Not Random Infinite → We want first infinite.

```
if (arr(mid) != target)
    start = mid + 1
```

if (arr(mid) == target)

we need to checks the previous element.

arr[mid - 1] == target.

So if (arr[mid] == target) and arr[mid-1] == target)

decrease End.

End = Mid - 1. = 10 - 1 = 9.

End = Mid - 1

Start = 7.

⇒ $\frac{9+7}{2} = 8.$

Go to 8 → check for 8 & 7 if not return 8 (2)

So Binary Search helps to solve Logically instead of Going for sorted Array. (Time Complexity = O(log(n)))

Interview Questions: {Bit Manipulation}.

Bit → 0 or 1 (There are only Bits)

Byte → 8 bits { combination of 0 and 1 }.

Code speed will be > than Decimal in bits

AND

x	y	
0	0	0
0	1	0
1	0	0
1	1	1

OR

x	y	
0	0	0
0	1	1
1	0	1
1	1	1

NOT

x	x̄
0	1
1	0

XOR .. { Left & Right shift }.

x	y	
0	0	0
0	1	1
1	0	1
1	1	0

$a\bar{b} + b\bar{a}$

Left shift ✓ Right shift.

→ <<1. → >>1

Decimal Number → Binary →

2	5	
2	2	1
	1	0

same.

2	5	1
2	2	0
2	1	1
	0	

→ (101)₂

$2^0 \times 1 + 2^1 \times 0 + 2^2 \times 1$

$1 + 0 + 4 = 5$.

2	10	
2	5	0
2	2	1
	1	0

1010 → $(1010)_2$.

To Convert Back

$101 \longrightarrow 1\times2^0 + 0\times2^1 + 1\times2^2 = 5$

Left shift ⇒ ?

5 → 101 → left shift.

5 << 1 → 10 1

101 → shift each bit by 1.

So 1010 = (10).

Empty.

shifting 1010 →

1010

10100 = 20

10<<1

0s

5<<2

{ Complexity wise 5<<1 it's faster than Normal Multiplic" as involving bits).

5 left shift by 1 = 10 ×2 *2

5 left shift by 2 = 20 ×2

5 left shift by 3 = 40

Right shift. → 5 >>1. (Right shift by 1)

101 >>1 ⇒

101

10

Empty space so 0 → 010 = (2).

↓ >>1

Right shifting by → divide by 2 001 = (1)

So in Any part of our code requires division rather than traditional division we go for right shift which is Lesser Time Complexity.

Manual Division	Right shift
Manual Multiplication	left shift

Interview Question { Repeat & Missing Number Array }

[3, 1, 2, 5, 3] , output = [3, 4]

repeated & Missing Numbers from Array are. 3, 4

Can be Solved by Bit Manipulation → Application of XOR

Missing Element, Missing Element

Most optimised approach is Bit Manipulation

Elements are in sequence [3, 1, 2, 5, 3]

n = len(arr) = 5.

Take XOR between.

arr[i] | i = 1 to n

$3 \wedge 1 \wedge 2 \wedge 5 \wedge 3$ | $1 \wedge 2 \wedge 3 \wedge 4 \wedge 5$.

$3 \wedge 4$ → We need to print 3, 4. { where one is repeating & other is Missing }

Has to know which is repeating & Missing.

3 → 011

4 → 100

```
  011
^ 100
-------
  111   → 7
```

{ We need to point List [3,4] }

↳ This is the extreme right Set Bit!!

How to know the Extreme right set Bit.

010 → { Middle one is extreme right set bit }.

111 → { first is right set Bit } Finding set Bit

So → 7 - 1 => 6 (110) → NOT → 001

↓ This is the Right set Bit.

Bit Number = 1 { extreme Right }.

[3, 1, 2, 5, 3]

↑ XOR → ① { extreme right bit }

```
3   011
1 & 001
-------
    001
```

Array.

Yes

No.

set Bit.

XOR

```
  001
  001
-----
  000
```

```
  010
  001
-----
  011
```

XOR

arr[i] ✓

Now i = 1 to 5

repeat again

x

y

Again XOR.

3 remains

4 remains

which are our answer

Repeated & Missing Elements from ~~An Array~~. (1)

Missing = M, Repeated = R.

Step 1: XOR of all Elements of array & all Numbers from 1 to n.

XOR1 = arr[0] ^ arr[1] ^ arr[2] ... arr[n]

XOR2 = 1 ^ 2 ^ 3 ... n

Then. XOR1 ^ XOR2 = M ^ R

XOR_all = XOR1 ^ XOR2

↳ in this all numbers will cancel Out & left with M & R.

Step 2: find the right Most Bit (Bit where M and R differ?)

Logic for finding right most bit {001}.

right_most_set_bit = XOR_all & (~(XOR_all-1))

↳ The right most value where it is 1.

This bit helps us partition Numbers into 2 Groups, one Group:

1) with that bit set
2) with that bit Not set.

Step 3 Partition And XOR Again. → {Array numbers, 1 to n}

Separate **both the Array & Numbers 1 to n** into 2 Groups based on right Most set bit. ↓ Two partitions

(1) right Most set Bit.

Using ↙

RMSB

&

All Array ele & No's.

input = [3 1 2 5 3]

Output : [3, 4] → Missing.

↓ repeating

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

Step1: $3 \wedge 1 \wedge 2 \wedge 5 \wedge 3$ $(\wedge)$ $1 \wedge 2 \wedge 3 \wedge 4 \wedge 5$

individual XOR's and Combined XOR { if same element then XOR is zero, if Not x }.

So $3 \wedge 3 = 0$ $1 \wedge 1 = 0$ $2 \wedge 2 = 0$ $5 \wedge 5 = 0$

So Left with. $3 \wedge 4$ { 3 = 011, 4 = 100 }.

it Gives two Numbers But Not sure which is repeating & Missing.

So for this we Go to step2

Step2: We Get 7 as result; but our Target the Uparate Numbers.

$3 \rightarrow 011$
$4 \rightarrow 100$
$3 \wedge 4 = 111$

$3 \wedge 4 = 11(1)$

Extreme Right Set Bit = 1. of from right side where we Getting 1

To Get This. ⟶ The Answer is (001) as the right most Bit is 1

1st : subtract by 1 $\Rightarrow (7-1) = 6. \rightarrow \sim \rightarrow \sim 6 \rightarrow$ 110.

$(XOR\_All) \;\&\; (\sim((XOR\_all) - 1))$

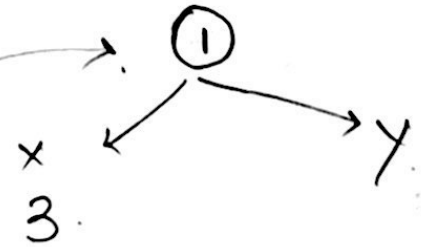
111 110

111
110
001 → Right Most Set Bit Or (right most 1)

→ This is the value where 1 is set at right side.

[3,1,2,5,3]
&
[1,2,3,4,5]

divide with the XOR



& 011
001
001 → Right side 1

So it belongs to

X

(2)

```
  011
  100
  ---
  111
```

→ XOR

To Get Set Bit → (XOR) & (~XOR)

~(XOR-1)

↳ 6 → 110

001

Gives ⇒ 001

where will be the Right most Set Bit for 7 which is 001.

(XOR-1) & (~(XOR-1 -1))

↓ 7-1=6 → 110 ⟶ 001

(111) & (001) → 001

Now Check which belongs to set Bit & which doesn't belongs to set bit.

(3) And Operation between set Bit & remaining All Elements.

(4) XOR blw individuals & the in between, & check them back with the Array to find MKR.

X is in arr[i]

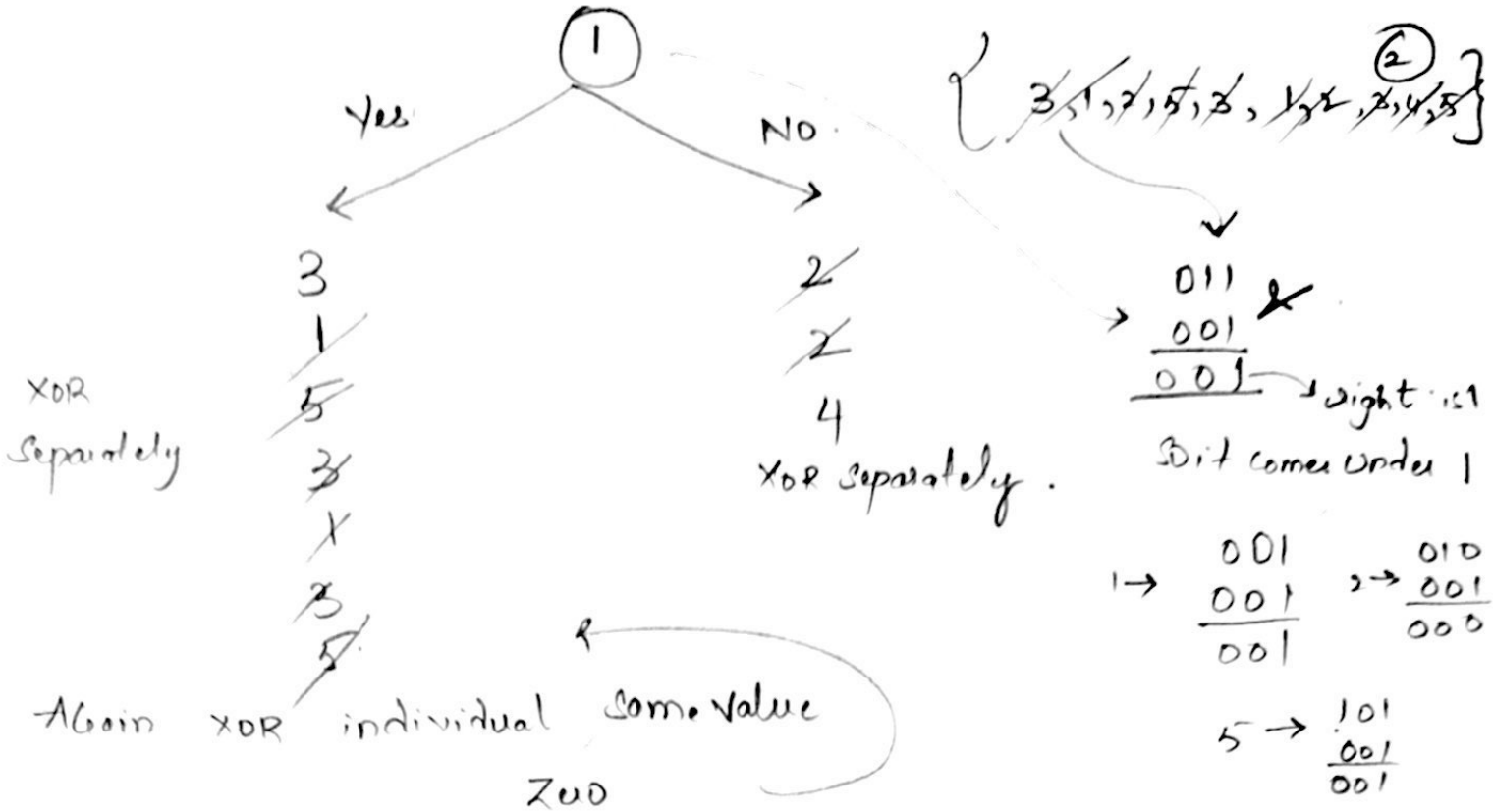
repeat = x.

Missing = y.

Otherwise

Missing = x

repeat = y.



So we Left with 3 and 4.

{3,4} → still Not sure which is R & M.

if it is available in original Array, it is repeating else Missing Repeat = 3 Missing = 4.

PSEUDO CODE.

① XOR → arr[i] ∧ i=1 to n.

⇓

Gives Numbers which is Combination of Missing ∧ repeated Or repeated ∧ Missing → 3∧4.

② which is repeated & Missing.

The right side will be different if both different Numbers

eg 7,3 → 111 / 011

6,4 → 110 / 100 { There will be atleast one different comb^n of 1 and 0 }.

Code Implementation $x=0, y=0$.

$x \oplus 1 = A[0]$ & $x=0, \; Y=0$

$[7, 8, 6, 5, 4, 8, 2, 1]$ → R

Output = $[8, 3]$ → M.

$7 \wedge 8 \wedge 6 \wedge 5 \wedge 4 \wedge 8 \wedge 2 \wedge 1 \quad \wedge 1 \wedge 2 \wedge 3 \wedge 4 \wedge 5 \wedge 6 \wedge 7 \wedge 8$

$8 \wedge 3 \rightarrow$

```
 1000
 0011
 1011
```

Right Most set Bit.

$(11) \& (\sim(11-1)) = (11) \& (\sim(10))$

$\downarrow 10 \rightarrow \sim 10 \quad \sim(1010) = 0101$

```
 1011
 0101
 0001
```

→ Right Most set Bit.

→ ①

Array to two Groups.

$7 \rightarrow 0111 \; \& \; 0001 = 0001$

$8 \rightarrow 1000 \; \& \; 0001 = 0000$

x	y
7	8
5	6
1	4
1	8
3	2
8	2
7	4
{XOR}	6
	8
	{XOR}

{3, 8} → check in x.