Lecture #9

COLUMN MAJOR ORDER

→ The elements of the array are stored on a column by **column basis**.

	0	1	2
0	0,0	0,1	0,2
1	1,0	1,1	1,2
2	2,0	2,1	2,2

→ logical representation

D_1 (Row) = 3

D_2 (Column) = 3

then,

Size = Row × Column

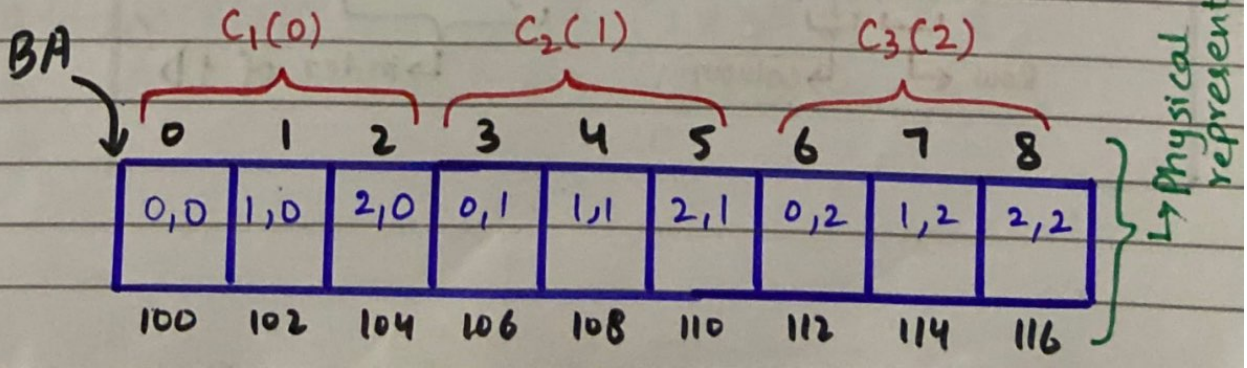
= 3 x 3

= 9

So,

Upper Bound = **8**

Lower Bound = **0**



NOTE:

* In Column major order, column becomes major and row becomes minor.

CALCULATING ADDRESS OF MEMORY FOR 2-D ARRAY

(COLUMN MAJOR ORDER MAPPING FORMULA)

* We need to map logical indices to physical indices to access a particular element of the array.

* Suppose a **2D** array has **u_1 rows** and **u_2 columns**, an element in **i^{th} row** and **j^{th} column** can be obtained by following formula:

GENERAL FORMULA

$$A[u_1][u_2]$$

$$Address(A[i][j]) = BA + (ju_1 + i) \times ES$$

Row ← i, j → column

(ju₁ + i) → index of 1D

* To rea
have

* For th
numbe
with
row

* This
eleme
1D

FOR

EXAM

Find

Add

Addr

* To reach to the **jth column** we have to skip all previous columns.

* For this, simply multiply the total number of rows u_1 (first dimension) with **j** and add the desired row number **i**.

* This gives us the index of required element exist in the physical **1D** array.

FOR PARTICULAR ELEMENT

EXAMPLE (1)

$i = 2$
$j = 1$
$u_1 = 3$
$u_2 = 3$

Find address of this **2D** in **1D**?

$$A[u_1][u_2]$$

Row ↔ Column

From here [index] column '1' starts

$$\text{Address}(2,1) = BA + (1 \times 3 + 2) \times ES$$

$$\text{Address}(A[2][1]) = 100 + (1 \times 3 + 2) \times 2$$
$$= 100 + (5) \times 2$$
$$= 100 + 10$$
$$= 110$$

EXAMPLE (2):

$i = 1$

$j = 2$

$A[4_1][4_2]$

Address(A[1][2]) = 100 + (2 × 4 × 1) × 2 = 118

EXAMPLE (3):

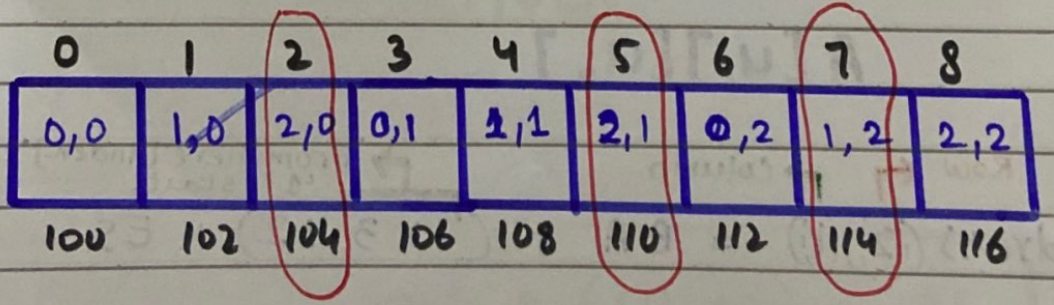
$i = 2$

$j = 0$

$A[4_1][4_2]$

Address (A[2][0]) = 100 + (0 × 4 + 2) × 2 = 104

```
int A[3][3];
```



Calculating Address of Memory for 3-D Array
(COLUMN MAJOR ORDER MAPPING FORMULA)

$A[u_1][u_2][u_3]$

* To access any arbitrary element i, j, k, following mapping formula can be used:

$$\text{Address}(A[i][j][K]) = BA + (K u_1 u_2 + j u_1 + i) \times E \cdot S$$

↳ If starting index is other than '0', then we will minus that index from every dimension.

Generalized Formula (COLUMN MAJOR ORDER MAPPING FORMULA)

$A[u_1][u_2][u_3] \dots [u_{n-1}][u_n]$

* To access any arbitrary element i, j, k, l, ... z following mapping formula can be used:

$$\text{Address}(A[i][j] \dots [y][z]) = BA + (z u_{n-1} u_{n-2} \dots u_1 + y u_{n-2} u_{n-3} \dots u_1 + \dots + j u_1 + i) \times ES$$

Column Major and Row Major Programming Languages:

* Both column major and row major are being implied by programming languages to handle multi-dimensional arrays.

* **C , C++ , Objective C** are few examples of languages used row major order.

* **MATLAB , Fortran** are few examples of languages used column major order.

* **JAVA** language can be used for both row major order and column major order.

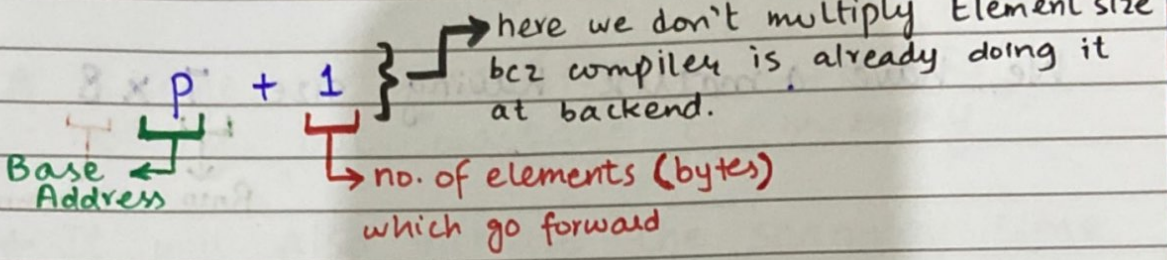
NOTE:

* The time complexity of row as well as column major order is **O(1)**.

* As time complexity is same for both row major order and column major order, so we are free to use / choose which one should be used.

IMPLEMENTATION (OF ROW MAJOR AND COLUMN MAJOR ORDER IN CODE)

```
int *p = ...;
```




```
p + 1 * sizeof(int);
```

* We don't need to do it in code as compiler is already doing it.
* It is abstraction bcz compiler is doing it already and it is hidden from us.

SPARSE MATRIX

* It is matrix having majority of elements **zero / NULL.**

FOR EXAMPLE:

We have a matrix having size 7 x 8
(7 → Row, 8 → Column)

Total Elements = 56 - 9
(9 → useful elements (non zero))

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 2 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 6 & 0 & 0 \\ 0 & 0 & 0 & 0 & 7 & 0 & 0 & 0 \\ 0 & 0 & 8 & 0 & 0 & 0 & 9 & 0 \end{bmatrix}$$

Sparse Matrix; Storage And Scanning

* Matrices are stored using two dimensional arrays.

* Storage of zero / NULL elements is nothing but "wastage of memory."

* It will also increase the scanning time of non-zero elements, because we have to scan the whole **m x n** matrix each time.

* **For Example:**
  If we have an integer matrix of **10 x 10** in which only **10** values are **non-zero**.

* So, in total we allocate **10 x 10 x 2 = 200 bytes** of space to store this integer matrix.

* To access these **10 non-zero** elements we have to make **100** scan hence "computationally expensive".

TO MAKE IT SIMPLE:

↳ We only store non-zero values.

★ This will save the space as well as the computing time.

★ **Data Structure Definition:**

"Best Arrangement of data in computer's memory."

SPARSE REPRESENTATION TECHNIQUES:

★ For this, two sparse Matrix Representation techniques are used:

(1) Triplet Representation (Array / 3-Column representation)

(2) Linked List Representation

(1) TRI

★ In thi
is use
m+1
numbe
an ex
about
Sparse

★ Each
only
their
value

★ The
total
numb
of n

(1) TRIPLET REPRESENTATION:

* In this technique, a **2D** array is used having **3-columns** and **m+1 rows**, where **m** is total number of non-zero values and an extra row to store information about sparse matrix (Data about Sparse Matrix).

* Each row of **2D** array stores only non-zero values along with their row and column index values.

* The first row of **2D** array stores total number of rows, total number of columns and total number of non-zero values of sparse matrix.

SPARSE TRIPLET REPRESENTATION

★ consider a matrix of 5×5 containing 8 non zero values.

$$A = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 3 & 2 & 0 \\ 0 & 5 & 0 & 4 & 0 \\ 0 & 0 & 0 & 9 & 0 \\ 0 & 6 & 0 & 0 & 1 \end{bmatrix}_{5\times5} \end{array}$$

If traverse this,

$5 \times 5 = 25$ searches.

So,

→ First, we count no. of non-zero values.

Count of non-zero element = 8

So, we will make 2D Array having 3 columns.
And,

Array for Meta-Data

Rows = count of non-zero element + 1
= 8 + 1 = 9

⇒ A[9][3]

ROW	COLUMN	VALUE
5	5	8
0	1	1
1	2	3
1	3	2
2	1	5
2	3	5
3	3	9
4	1	6
4	4	1

0 1 2 3 4 5 6 7 8

3 → Meta data (Total rows, total column, total non zero values)

Representation in Memory

BENEFIT OF THIS:

For addition as well as substraction, we want size, which we will get from this theow.

Sparse Matrix → Sparse Representation → 1-D

FOR TRAVERSING IN SPARSE MATRIX REPRESENTATION:

Now,

* As Triplet Representation have **3** columns, so there is no need to use nested loops, for column (second dimension) will always be **0, 1** and **2**.

* We will only use one loop which will execute from **'0'** to **m** index.

* If we are searching for values (non-zero), then we will not need to go on **'0'** index. We will start traversing from index **'1'**, because there is no value on **'0'** index; there is only Meta-data.

* Now, by this Processing will be fast.

→ But if we want to copy data from sparse matrix to triplet representation, then we will require nested loops.

RIX

NOTE:

* We will not show user the sparse matrix.
* We will only show 3-column (Triplet) representation.
* As we have order (in Meta-data row '0') of sparse matrix, so we can regenerate the sparse matrix to show it to user whenever needed.

ADDITION OF TWO SPARSE MATRICES TRIPLET REPRESENTATION

A

Row	Col.	Value
5	5	8
0	1	1
1	2	3
1	3	2
2	1	5
2	3	4
3	3	9
4	1	6
4	4	1

B

Row	Col.	Value
5	5	6
0	0	9
0	1	4
1	2	5
2	0	8
3	1	2
3	3	7

$C = A + B$

CHECK POSIBILITY FOR ADDITION:

★ First, we will check if both sparse matrices are of same size by Meta data (data from row '0').
  ↳ number of rows must be equal to the number of columns of '0' index.

★ So, here the order of both matrices **A** and **B** is same, so the addition can be performed.

NOTE :

→ The size of resultant matrix is unknown therefore we can take maximum size i-e. the total count of values of matrix A and B which is 5 + 6 = 11 + 1 for first row.

So, if true:

A

	ROW	COL.	VALUE
~~i→~~	5	5	8
i→	0	1	1
i→	1	2	3
i→	1	3	2
i→	2	1	5
i→	2	3	4
i→	3	3	9
i→	4	1	6
i→	4	4	1

B

	ROW	COL.	VALUE
~~j→~~	5	5	6
j→	0	0	9
j→	0	1	4
j→	1	2	5
j→	2	0	8
j→	3	1	2
j→	3	3	7

Resultant matrix, 'C' will create:

C

	ROW	COL.	VALUE
	5	5	~~14~~ 11
K→	0	0	9
K→	0	1	5
K→	1	2	8
K→	1	3	2
K→	2	0	8
K→	2	1	5
K→	2	3	4
K→	3	1	2
K→	3	3	16
K→	4	1	6
K→	4	4	1
K→			
K→			
K→			

→ We will increment this count when element adds.

* We will keep adding elements in this matrix until the data of either A or B matrices finish.
* When data of one matrix finish, we copy data of other matrix.

→ here, memory is also wasting, but wastage is less.

→ To reduce wastage, we will copy the data in a new array whose size equal to non-zero element.

VALUE
6
9
4
5
8
2
7

NOTE:

* Now, by this memory will not waste.
* In sparse matrix (3-column / Triplet) representation, the time complexity is **O(1)**.

TRANSPOSE OF SPARSE MATRIX

→ Only replace columns by rows.

→ If after doing triplet represent, we want take transpose of sparse matrix, then first we will create a new matrix and then will equal the number of columns of sparse matrix to the number of rows of new matrix, and vice versa.

BENEFIT OF THIS TRIPLET REPRESENTATION:

→ Lower Triangle
→ Diagonal
→ Upper Triangle

→ For all these type of matrices, we use sparse matrix.

→ Storage Operation on these type of matrices become easy.

Lecture #10

BEST, WORST, AVERAGE CASE ALGORITHM ANALYSIS

* The running time of an algorithm increases as the input size **n** increases or remains constant in case of constant running time.
* In that case, we perform Best, Average and Worst case analysis of an algorithm.
* These measures are performed asymptotically.
* For every algorithm, there are **THREE TIMES:**

1) BEST CASE:

* How much **minimum** time an algorithm takes to execute
* **Minimum** number of steps taken by an algorithm on any instance of size **n.**

* The best case is usually not estimated because it does not give us better measure about the performance of an algorithm.

(2) WORST CASE:

* How much **maximum** time an algorithm takes to execute.

* **Maximum** number of steps taken by an algorithm on any instance of size **n.**

* This gives us the better measure about the performance of an algorithm.

* We mostly interested in finding the worst case of an algorithm.

* The worst case is mostly performed in algorithm analysis.

(3) AVERAGE CASE:

* Average case of an algorithm is the function defined by the **average number** of steps taken on any instance of size **n**.

* We sum up all the possible cases time and divide them with the number of cases.

$$\text{Average case} = \frac{\text{All possible cases Time}}{\text{Number of cases}}$$

* The average case is also usually performed in algorithm analysis.

* Most of the time the average case is roughly equal to the worst case. (Similar)

* Average case analysis may not be possible always.

* It is complex to determine average case, because we have to identify each case individually.

* It is difficult to find Average case of an algorithm.

→ BEST, WORST AND AVERAGE CASE ANALYSIS WITH THE HELP OF SOME SORTING ALGORITHM:

* Suppose an algorithm sorts a list of n elements in **ascending** order

* **BEST CASE:**
  When the input list is already sorted.

* **WORST CASE:**
  When the input list is reverse sorted.

* **AVERAGE CASE:**
  When the input list is partially sorted (have to consider all the cases).

Asymptotic Notations for Best, Worst and Average Cases:

Note:

* Any asymptotic notation **Big-OH (O)**, **Big-Omega (Ω)** or **Big Theeta (θ)** can be used to represent best, worst and average cases of an algorithm.

* (ii) **For example,** if an algorithm performs **n+1** steps in best case, we can represent it as **O(n)**, **Ω(n)** or **θ(n)**.

* (iii) **For example,** if an algorithm performs **$3n^2+5n+1$** steps in worst case, we can represent it as **$O(n^2)$**, **$\Omega(n^2)$** or **$\theta(n^2)$**

* In the same way, average case can be represented with any of the asymptotic notations.

Searc

(1) Line

* Linear algorith

* It se of the or th search

* A va check arra

  • if

  • if

SEARCHING ALGORITHMS

(1) LINEAR SEARCH:

★ Linear search is very simple search algorithm.

★ It sequentially checks each element of the list until a match found or the whole list has been searched.

★ A value (key) is given, we will check/ compare to find it in the array.

- if value found → return 'index'
- if value not found → return '-1'

PROGRAM (OF LINEAR SEARCH)

```
int  LinearSearch(int a[], int n, int key)
```

(return index; list; size; search)

```
{   we use for loop because no of iterations are known.
    for(int i=0; i<n; i++){          —   1        n+1
        if(a[i]==key)                —   1        n
            return i;                —   1        1
    }                                    T(n)=3   T(n)=2n+2
    return -1; // if element does not exist.
}
```

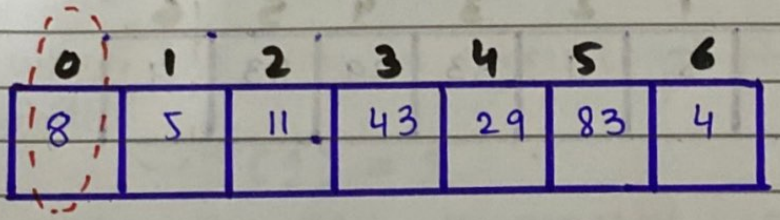
i.e: i = 0 ... n-1

→ Now, we will find the Best, Worst and Average case of Linear Search:

BEST CASE:

* Best case of linear search exist when the desired **Key** to be searched is present at the first index of array.

* If value exists at index = **0**



* We measure best case mostly in **big-O**. (but it is our choice; we can also use Big-Omega and Big-Theeta)

$T(n) = 3$ → It is efficient algorithm as it takes constant time.

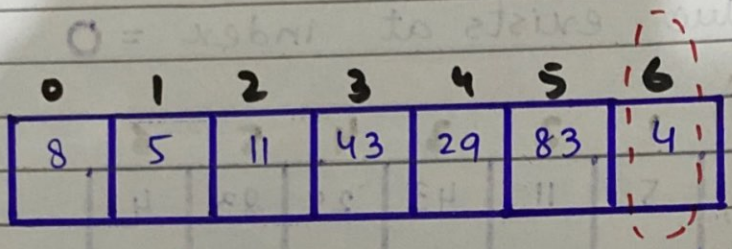
$O(1)$ ↔ bcz we only compare at '1' time.

* We can also use $O(1)$, $\Omega(1)$ or $\Theta(1)$ for Best case.

Worst Case:

* Worst case of linear search exist when the desired **key** to be searched is present at the last index of array (if value exist at end index)

* Worst case of linear search exist when the value does not exist.



$T(n) = 2n + 2$

$O(n)$ ⟷ Linear growth; it is not a good growth.

* It means linear search has $O(n)$ worst case time complexity.

no. of comparisons

AVERAGE CASE:

* Average case does not mean picking up middle value and taking its time complexity.

* We individually check time complexity of every case one by one in average case.

* Identify all the case times

- When the key is present at first '1' index.

$$T(1) = 1$$

- When the key is present at second '2' index.

$$T(2) = 2$$

- When the key is present at third '3' index.

$$T(3) = 3$$

- When the key is present at fourth '4' index.

$$T(4) = 4$$

⋮

- When the key is present at last 'n' index.

$$T(n) = n$$

no. of comparisons

FORMULA:

$$\text{Average case} = \frac{\text{Sum of all cases}}{\text{no. of cases}}$$

$$= \frac{1+2+3+4+\dots+n}{n}$$

$$= \frac{n(n+1)/2}{n}$$

$$= \frac{(n+1)}{2} \rightarrow \text{Linear growth}$$

★ So, average case time complexity of linear search is **O(n)**.

★ In this way, we learnt that mostly average and worst case will be same (similar).

CODE IMPLEMENTATION OF LINEAR SEARCH:

→ In Object Oriented Programming **(OOP)**, we don't pass **array** (ar[]) and its **size** (n) in object, whenever we are writing algorithm for linear search. It is because, array and its size already exist in object/class.

NOT

★ Lin
whi
as

(2)

★ B
fo
li

★

ir
t
t

→

a

n
n/2
n/4
n/8
⋮
n/r

NOTE:

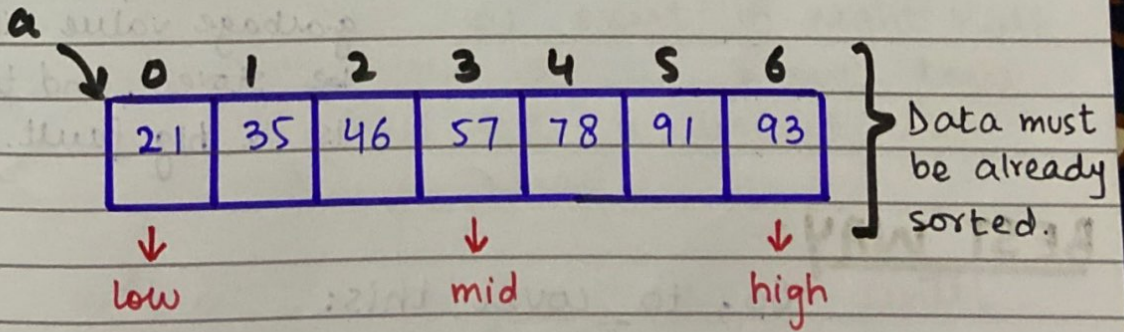
★ Linear Search is such algorithm which works efficiently for sorted as well as unsorted data.

(2) BINARY SEARCH:

★ Binary Search is an efficient algorithm for finding an item from a **sorted** list of items.

★ It works by repeatedly dividing list into **two halves** that could contain the item, until it narrowed down the possible locations to just one.

➡ BINARY SEARCH REQUIREMENT:



n
n/2
n/4
n/8
⋮
n/n = 1

- The **LOW** holds the first index of array.
- The **HIGH** holds the last index of array.

- For calculating

$$mid = \frac{low + high}{2}$$

↳ • This formula is right for paper work but for programming (coding), its not suitable. Bcz overflow of data type might occur in programming.

- if int takes '4' bytes and range of int is '32000'.

$$mid = \frac{17000+16000}{2}$$

The result of 'int' generated from here will be '33000', which exceeds the range of int (i.e 32000). So, '33000' cannot store in **int** and because of this, garbage value will be stored, and this is a big fault.

BEST WAY:

So, to cover this:

$$mid = low + \frac{(high - low)}{2}$$

↳ Now, overflow of data type does not occur.

$$mid = 0 + \frac{(6-0)}{2}$$

$$\boxed{mid = 3}$$

★ Compare key with **mid** element. If key found, search successful.

★ Otherwise two possibilities exist

- The key is smaller then from the **mid** element.
  - ↳ Then key exist in the left half of array. So, we move **high** to **mid - 1** to forget/ discard right half of array.
- The key is greater then from the **mid** element.
  - ↳ Then key exist in right half of array. So, we move **low** to **mid + 1** to discard left half of array.
- This process is repeated until **lower bound** crosses **upper bound**.

PROGRAM (OF BINARY SEARCH)

```
int binarysearch(int a[], int n, int key)
{
  int low = 0, high = n-1, mid;
  while (low <= high)
  {
    mid = low + (high-low)/2;
    if (a[mid] == key)
      return mid;
    else if (a[mid] > key)
      high = mid - 1;
    else if (a[mid] < key)
      low = mid + 1;
  }
  return -1;
}
```

Loop will execute repeatedly until lower and upper bound crosses each other.

here, no. of iterations are unknown. so, we use 'while' loop.

if the element (key) is found at 'mid'

if the element (key) is smaller than 'mid'. So, we skip **right half** of array.

if the element (key) is greater than 'mid'. So, we skip **left half** of array.

BINA

Con

★ On ski un the

n

n

★ S s

W

★ I wl pr i

★ I u

BINARY SEARCH TIME COMPLEXITY ANALYSIS:

★ On each iteration, binary search skips the half array elements until only one element left in the array.

n	$n/2$	$n/4$	$n/8$	...	$n/n = 1$
$n/2^0$	$n/2^1$	$n/2^2$	$n/2^3$	...	$n/2^k = 1$

$(\because 1 = n/2^k$

$n = 2^k$

$\log_2 n = k)$

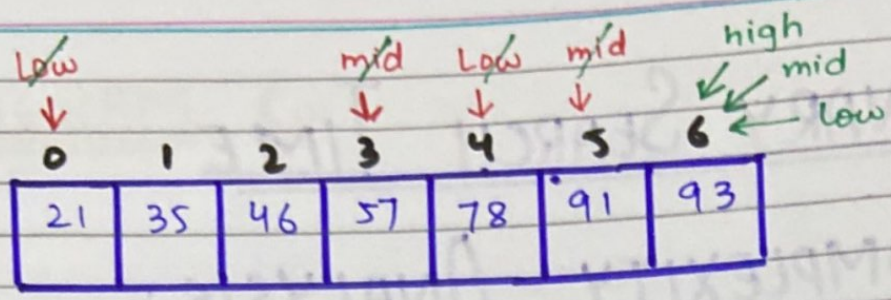
★ So, the time complexity of binary search is **O(lgn)**.

WORST CASE:

★ In binary search, worst case exists when finding element (key) is either present at start index or end index.

OR

★ In binary search, worst case exists when there is no element.



For finding '93':

$$mid = 0 + \frac{(6-0)}{2}$$

$$= 3$$

$57 < 93$

$$mid = 4 + \frac{(6-4)}{2}$$

$$= 5$$

$91 < 93$

$$mid = 6 + \frac{(6-6)}{2}$$

$$= 6$$

$n/2^0 \Rightarrow n$

$n/2^1 \Rightarrow n/2^1$

$n/4 \Rightarrow n/2^2$

$n/8 \Rightarrow n/2^3$

$\vdots \qquad \vdots$

$n/n = 1 \Rightarrow n/2^k$

1 =

2^k

⇒ log

★ So, searc

BES

★ In wh at

★ Tir bin

Av

★ W t

★ I f

$$1 = n/2^k$$

$$2^k = n$$

$\Rightarrow \log_2 n$] → Best / efficient growth rate

* So, the worst case of binary search takes **O(lgn).**

BEST CASE:

* In binary search, best case exists when finding element (key) is present at 'mid' at our first iteration

* Time complexity for best case of binary search is **O(1).**

AVERAGE CASE:

* We have to take mean of all the time cases.

→ For one element, two elements, three elements ... for n elements.

* We don't need to find average case for binary search separately as it becomes complex.

* So, that's why we will assume it similar to worst case.

* So, time complexity of average case of binary search is $O(\lg n)$.

NOTE:

* We apply binary search only when data is in sorted form.

* Binary Search cannot be applied for unsorted data.