SCALER

Searching - 1

TABLE OF CONTENTS

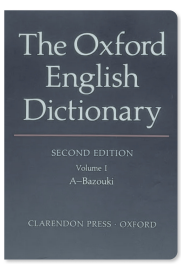
Searching

Finding something

Phone Lost
- Brand, IMEI, Model - - - - (what to search) Target
- Lost Location. (Where to Search) Search Space.

1. Searching a word in newspaper

Searching a word in dictionary



Target : Word

Search Space : Dictionary / Newspaper.

2. Search phone no in Contact List

Search phone no in diary

Target : Phone NO

Search Space : Contact List, Diary.

Obs : Searching becomes easy if the search space is SORTED

Search "Dog" in Physical Dictionary

We randomly open a page & discard one of the sides & move to the other side.

Binary Search

Dividing the array into 2 halves & based on some condition we discard one of the half.

When can we apply Binary Search?

After dividing the array into 2 halves, if we are able to discard one of the half using some conditions. we can apply BS.

Note: Data may or maynt be sorted.

< Question > : Given a sorted arr[ ]. Search if an element K is present or not.

0	1	2	3	4	5	6	7
3	6	9	12	14	19	20	23

K = 12 True.

K = 17 False.

BF Approach

Linear Search

T.C : O(N) S.C : O(1)

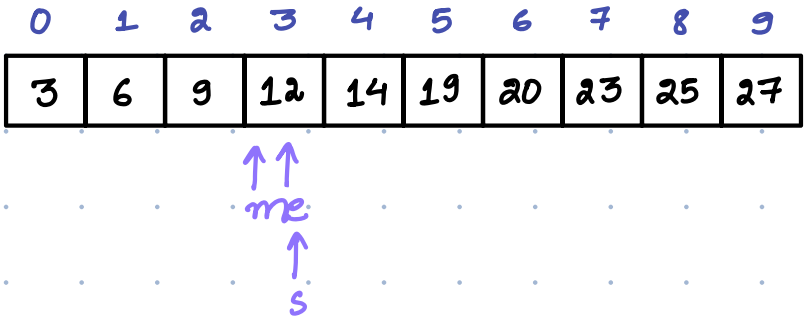
Idea -2 Binary Search

Target : K Search Space : Entire Array

A[mid] == K Return True.

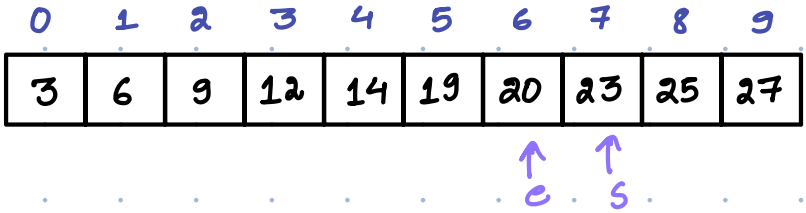
A[mid] < K Discard Left
Move towards Right.

A[mid] > K Discard Right side
Move towards Left.



K = 12

s	e	m = s + e / 2	A[m]	K A[m]	
0	9	4	14	12 < 14	Go to LHS. e = m - 1
0	3	1	6	12 > 6	Go to RHS s = m + 1
2	3	2	9	12 > 9	Go to RHS s = m + 1
3	3	3	12	12 == 12	Return True



K = 21 → False.

s	e	m = s + (e-s)/2	A[m]	K A[m]	
0	9	4	14	21 > 14	Go to RHS s = m+1
5	9	7	23	21 < 23	Go to LHS. e = m-1
5	6	5	19	21 > 19	Go to RHS s = m+1
6	6	6	20	21 > 20	Go to RHS s = m+1
7	6	Search Space Exhausted. (s > e)			Return False.

< / > Code

```
boolean search (int[] A, int K) {
    int n = A.size();
    int s = 0, e = n-1;
    while (s <= e) {
        m = (s+e)/2;
        if (A[m] == K) {
            return True;
        }
        else if (A[m] < K) {        // Move to RHS.
            s = m+1;
        }
        else {                      // A[m] > K
            e = m-1;                // Move to LHS.
        }
    }
    return False;
}
```

$N \longrightarrow \frac{N}{2} \longrightarrow \frac{N}{4} - - - - 1$

T.C : O(logN)

S.C : O(1)

< Question > : Given a sorted arr[N]. Find first occurrence of K.

If K is not present in array then return -1

arr[ ] → [	-5	-5	-3	0	0	1	5	5	5	5	5	8	10	10	15]
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

m

K = 5 → 6

K = -5 → 0

K = 20 → -1

BF Approach

Linear Search

T.C : O(N)

S.C : O(1)

Idea -2

Target : 1st occ. of K.

Search Space : Entire Array.

A[mid] < K

Discard Left
Move to RHS.
s = m + 1.

A[mid] > K

Discard Right
Move to LHS.
e = m - 1

A[mid] = = K

Discard Right

Store Mid. as an Ans.
&
Move to LHS.

</ > Code

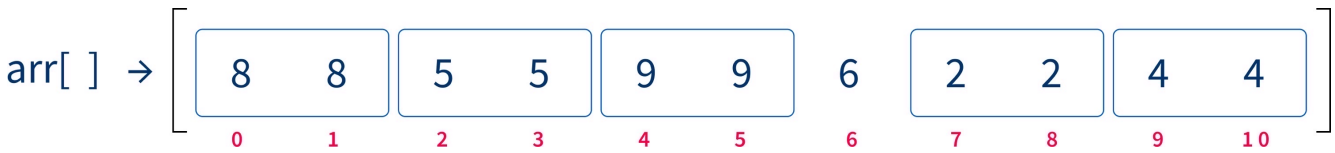
```
int firstOccurence(int[] A, int K){
    int s=0 , e = n-1 ;
    int ans = -1 ;

    while (s <= e){
        m = (s+e)/2 ;
        if (A[m] == K) {
            ans = m ;
            e = m-1 ;
        }
        else if (A[m] < K) {
            s = m+1 ;
        }
        else {
            e = m-1 ;
        }
    }

    return ans ;
}
```

T.C: O(log N)
S.C: O(1)

Unique Element

< *Question* > : Every element occurs twice except for 1. Find that unique element.

Note : Duplicates are adjacent to each other and array isn't necessarily sorted.



ans : 6.

Idea -1 XOR of all elements.

T.C : O(N) S.C : O(1)

Idea -2 Hashmap.

T.C : O(N) S.C : O(N)

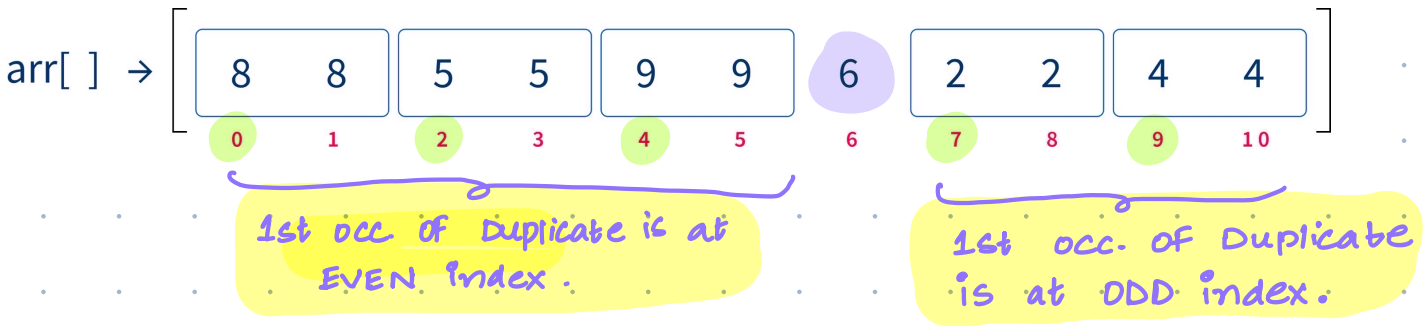
Idea -3 Linear Search

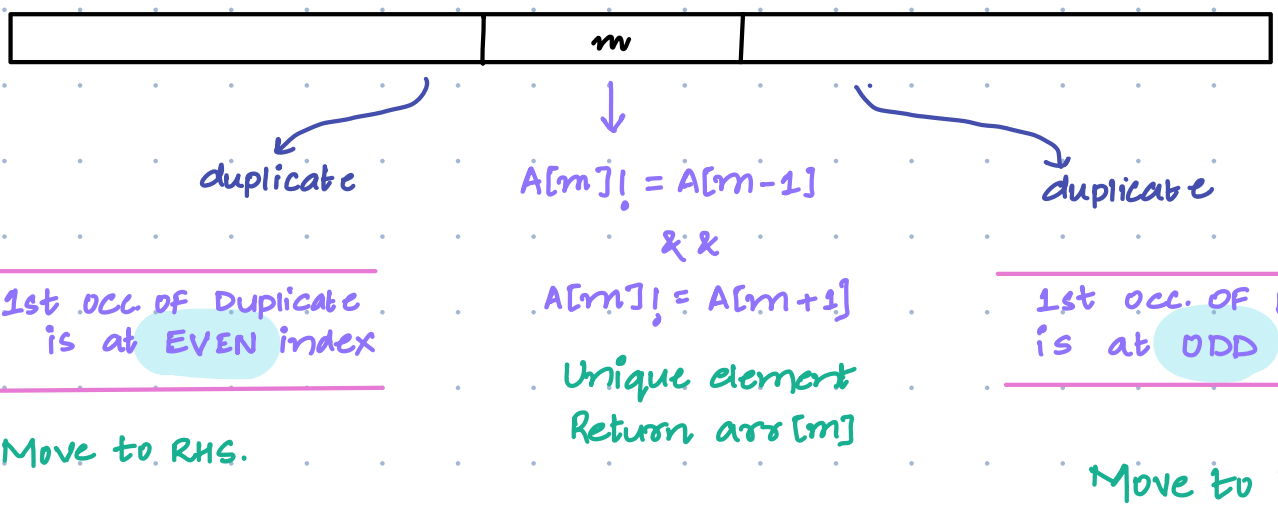
T.C : O(N) S.C : O(1)

Idea - 4 Binary search.

Target : Unique element

Search Space : Entire Array.



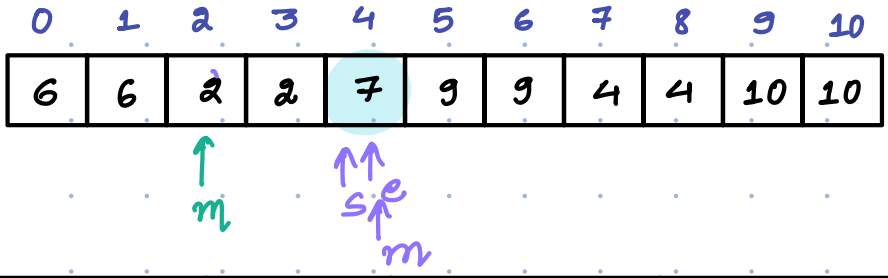


Algo :

Find mid & check if it is Unique or Not.

↓ Ans.

check for 1st occ of duplicate & decide which side to move.



s	e	m	arr[m] Unique ?	Is "m" at 2nd occurence?	m%2 == 0
0	10	5	✗	✗	✗ Go to LHS e = m-1
0	4	2	✗	✗	✓ Go to RHS s = m+2
4	4	4	✓ Return arr.		

< / > Code

T.C : O(log N)
S.C : O(1)

```
int unique(int arr[]){
    n = arr.size();
    if (n == 1) return arr[0];
    if (arr[0] != arr[1]) return arr[0];
    if (arr[n-1] != arr[n-2]) return arr[n-1];

    int s = 1, e = n-2;
    while (s <= e){
        m = (s+e)/2;
        // Case 1 : Element at m is unique.
        if (arr[m] != arr[m-1] && arr[m] != arr[m+1])
            return arr[m];

        // Case 2 : Element at m is Duplicate.
        if (arr[m] == arr[m-1])   // If m is at 2nd occ. bring it to 1st occ.
            m = m-1;

        if (m%2 == 0)
            s = m+2;
        else
            e = m-1;
    }
}
```

Local Minima

< *Question* > : Given an unsorted array of distinct elements. Find any one local minima.

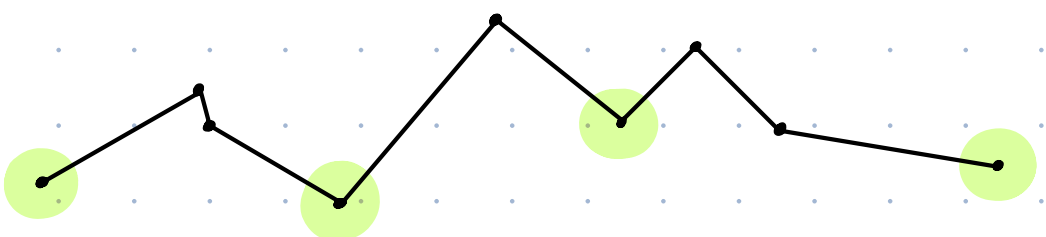
Local Minima : An element which is less than its adjacent neighbours.

0	1	2	3	4	5	6	7	8
9	8	7	3	6	4	1	5	2

Local Minima → $arr[i-1] > arr[i] < arr[i+1]$

$arr[0] < arr[1]$

$arr[n-2] > arr[n-1]$



BF Approach

Iterate & compare each element with its adjacent neighbors.

T.C : O(N)

S.C : O(1)

Idea

Target : local Minima

Search Space : Entire array.

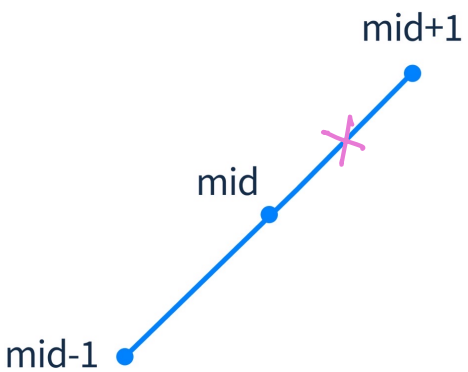
case : 1

Return arr[mid]

case : 2

Move to RHS.

case : 3

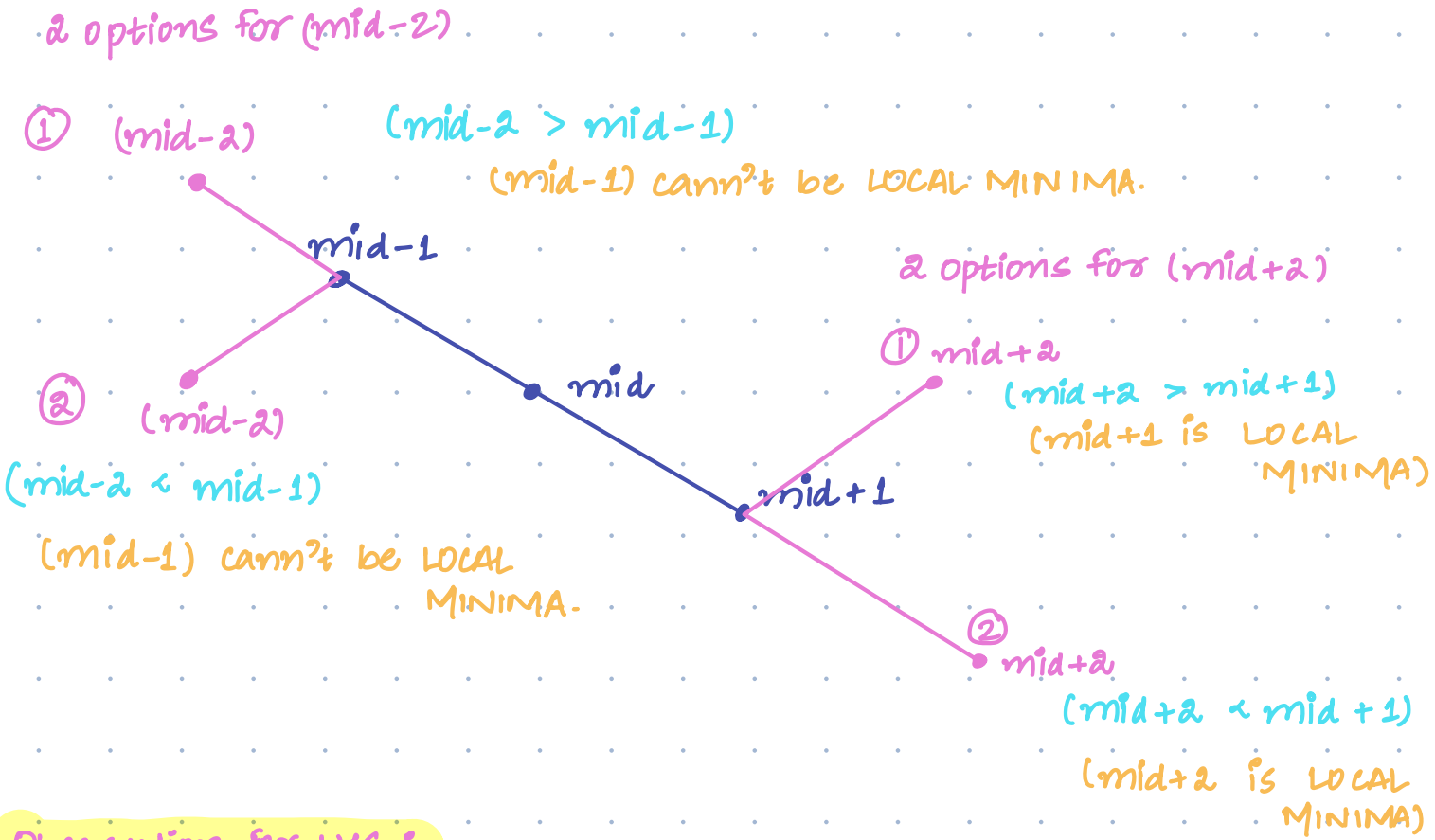


Move to LHS.

case : 4

Move to either side.

Analysing case 2:



Observation for LHS :

LHS may or mayn't have Local Minima.

Observations for RHS :

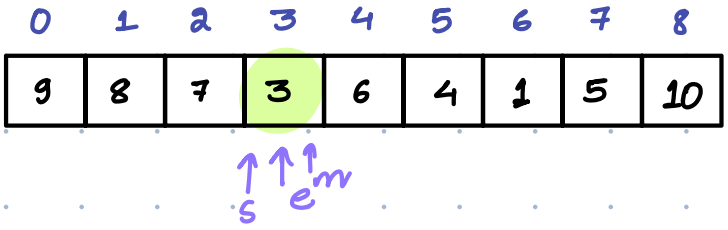
Definitely 1 Local Minima on RHS.

Conclusion : LHS may or mayn't have a LOCAL MINIMA but we are sure definitely there exists a MINIMA on RHS so **move to RHS.**

</> Code

```
int LocalMinima(int[] A){

    if( A.size() == 1) return A[0];
    if (A[0] < A[1]) return A[0];
    if (A[n-1] < A[n-2]) return A[n-1];

    s = 1 , e = n-2;

    while (s <= e){
        m = (s+e)/2;
        // case 1:
        if(A[m] < A[m-1] && A[m] < A[m+1])
                return A[m];
        // case 2:
        else if (A[m] < A[m-1] && A[m] > A[m+1])
                s = m+1;
        // case 3:
        else if( A[m] > A[m-1] && A[m] < A[m+1])
                e = m-1;
        // case 4:
        else
                e = m-1;
    }
}
```

T.C : O(log N)
S.C : O(1)



s	e	m		
0	8	4	Case 4 :	e = m-1
0	3	1	Case 2 :	s = m+1
2	3	2	Case 2 :	s = m+1
3	3	3	→ Ans.	Case 1.