17/02/23

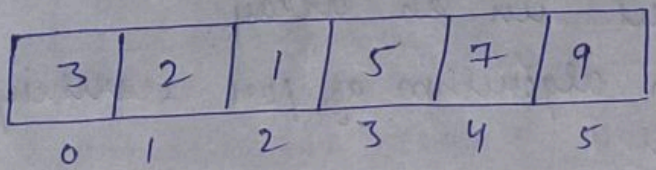
Searching:

Linear search: It is a easiest way of searching any element in any array.. It looks out with every element in an array & check if it equals to target element then print it.

Drawbacks of Linear search

If we have 1000 of elements in an array ~~the~~ compiler need to make 1000 of comparison to check whether required element is found or not.

ex:



target = 2

code:

```
for (int i=0; i<n; i++){
    if (arr[i] == target){
        cout<<found;
}   }
```

The time complexity of above code is O(n) bcz the for loop runs from 0 to n. no.. of times. ↳ also known as linear complexity.

```
for(i=0; i<n; i++){
    for(j=0; j<n; j++){
        if ( )
    }
}
```

for i=0 ⟶ j=0 to n
i=1 ⟶ j= 0 to n
i=2 ⟶ j= 0 to n
⋮
i=n ⟶ j= 0 to n

outer loop runs from 0 to n and inner loop also runs from 0 to n.

T.C of above code is $O(n*n)$
(outer loop * inner loop)

T.C is $O(n*n)$ i.e $O(n^2)$ which is bad complexity we need to optimized this code

* Searching Algo:

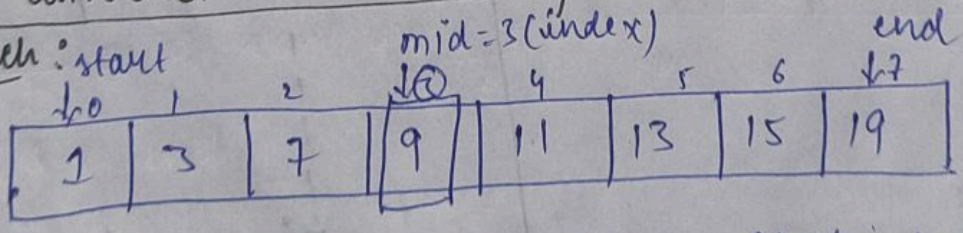
→ **Linear search**: T.C is $O(n)$ for 1000 elements need 1000 comparison which is avoided to use.

→ **Binary search:**
- 1000 elements of an array can be searched in 10 comparison
- It is an optimized way of searching an element in an array.
- It is an algorithm for searching an element
- Condition is there that elements must be in monotonic order. i.e. in sorted order. either (↑) or (↓)

* Many sorted order?

If we have array of an elements in sorted order either (↑) or (↓). Then we can find the mid of an array which will help in deciding to find the targeted element to the left of mid or to the right this reduces the time if (~~mid~~ target > mid) search in right else search in left if(target < mid)

* Steps involved in searching an element in Binary search:



① initialize start & end. size of array (last index)
start = 0 and end = n-1
↓ index

② find mid = $s + \frac{(e-s)}{2}$ or $\frac{(s+e)}{2}$ Explained later.

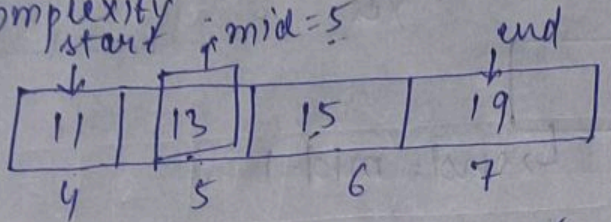
→ mid = $\frac{start + end}{2} = \frac{0+7}{2} = 3$

③ check whether the targeted element is target = 15

④ Now check (target > arr[mid]) or (target < arr[mid])

15 > 9 (Yes)

Now search right of the mid as we need not to find to the left of the mid which reduces time & complexity



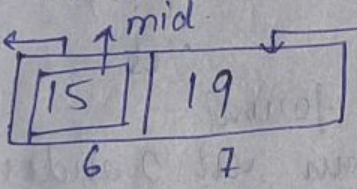
start = 4
end = 7

⑤ Now again find the mid = $\frac{(s+e)}{2} = \frac{4+7}{2} = 5$

⑥ Again check (target > arr[mid] or target < arr[mid])

as 15 > arr[5]

15 > 13 . Now search to the right need not to check on the right left.



start = 6
end = 7

⑥ Now again find the mid = $\frac{(s+e)}{2} = \frac{6+7}{2} = 6$

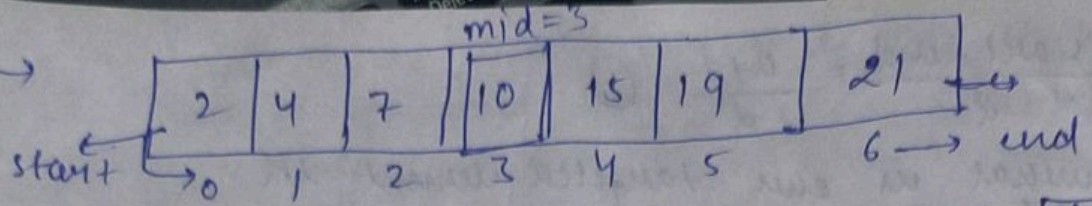
⑦ Now check one more condition i.e

(target == arr[mid]) (target > arr[mid]) (target < arr[mid])

⑧ as 15 = arr[6]

15 = 15 then element found at index 6

⑨ return ⑥.



start = 0, end = 7-1 = 6

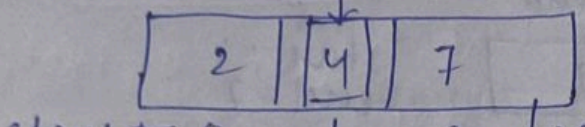
target = 2

$mid = \frac{(s+e)}{2} = \frac{0+6}{2} = 3$

if (target == arr[mid]) (target > arr[mid]) (target < arr[mid])

ϕ (2 ≠ 3) (2 > 3) (2 < 3)

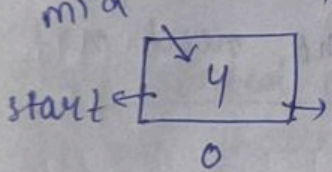
↓ yes



$mid = \frac{0+2}{2} = 1$

target < arr[mid]

2 < 4



start = 0

end = mid-1

= 1-1 = 0

mid = 0

[target == arr[mid]]

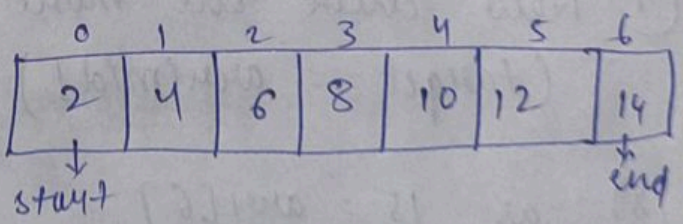
4 = 4

yes

found

return at 0 index found

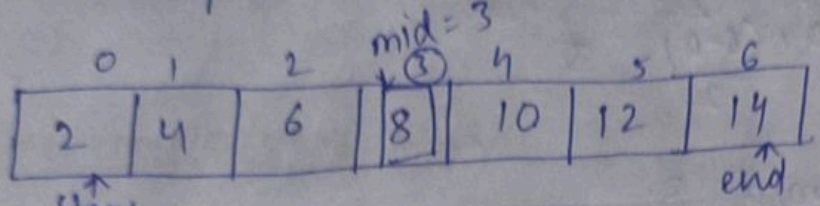
* Binary search code:




```
int s=0;
int e= n-1=6
        ↓size
mid= (s+e)/2; we wont use s+e/2
while (s<=end){
    if(arr[mid]==target){
        return mid; → index
    }
    else if (target > arr[mid]){
        start= mid+1; // right part
    }
}
```

```
else {
    // left me search
    end = mid-1;
}
mid = (s+e)/2;
}
return -1;
```

1) If no element not found.

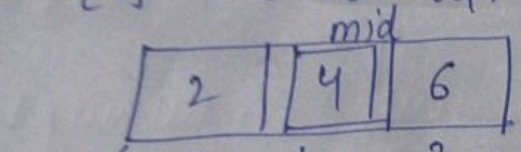


start = 0, end = n-1 = 7-1 = 6

mid = $\frac{0+6}{2} = 3$

target = 2

~~arr[3]~~ 2 < 8 left search

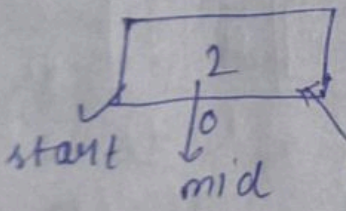


↳ end = mid-1 = 3-1 = 2

mid = $\frac{(s+e)}{2} = \frac{0+2}{2} = 1$

2 < arr[1]

2 < 4 (Yes)



end = mid-1
= 1-1 = 0

mid = $\frac{0+0}{2} = 0$

(target == arr[mid])
return mid;

Yes 2 is found at 0

* Why we need to take mid = $s + \frac{(e-s)}{2}$ not $\frac{s+e}{2}$

↔ Range of int = $-2^{31} \to 2^{31}-1$

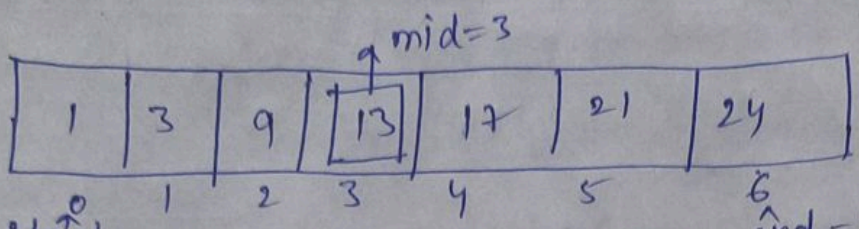
If we give start = $2^{31}-1$

∴ mid = $\frac{(s+e)}{2}$ then it will cross the ranges of integer i.e integer overflow so we will use.

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

$= s + \frac{e}{2} - \frac{s}{2} = \frac{s}{2} + \frac{e}{2} = \frac{s+e}{2}$

$s = s + \frac{(e-s)}{2}$ $(e-s)$ will reduce the value & then divide by 2 will reduce it to more



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

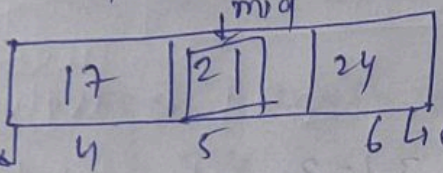
$= 0 + \frac{(6-0)}{2} = 3$

target = 22

start = 0, end = 6 mid = 3

22 > 13 i.e (target == arr[mid])

Right search



start = mid+1 = 4

$mid = s + \frac{(e-s)}{2} = \frac{6+4}{2} = \frac{10}{2} = 5$

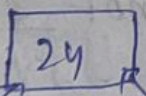
$= \frac{4 + (6-4)}{2}$

$= \frac{4+2}{2} = (5)$

target > arr[mid]

22 > 21 Yes

right shift



start = mid+1 = 6

end = 6

mid = 6

22 < 24 (NO

// left search

end = mid-1 = 6-1 = 5

start = 6

end = 5, start = 6 → come out of the loop.

Return (-1) → invalid index

Element Not found

Let $e = 2^{31}-1$

$s = 2^{31}-1$

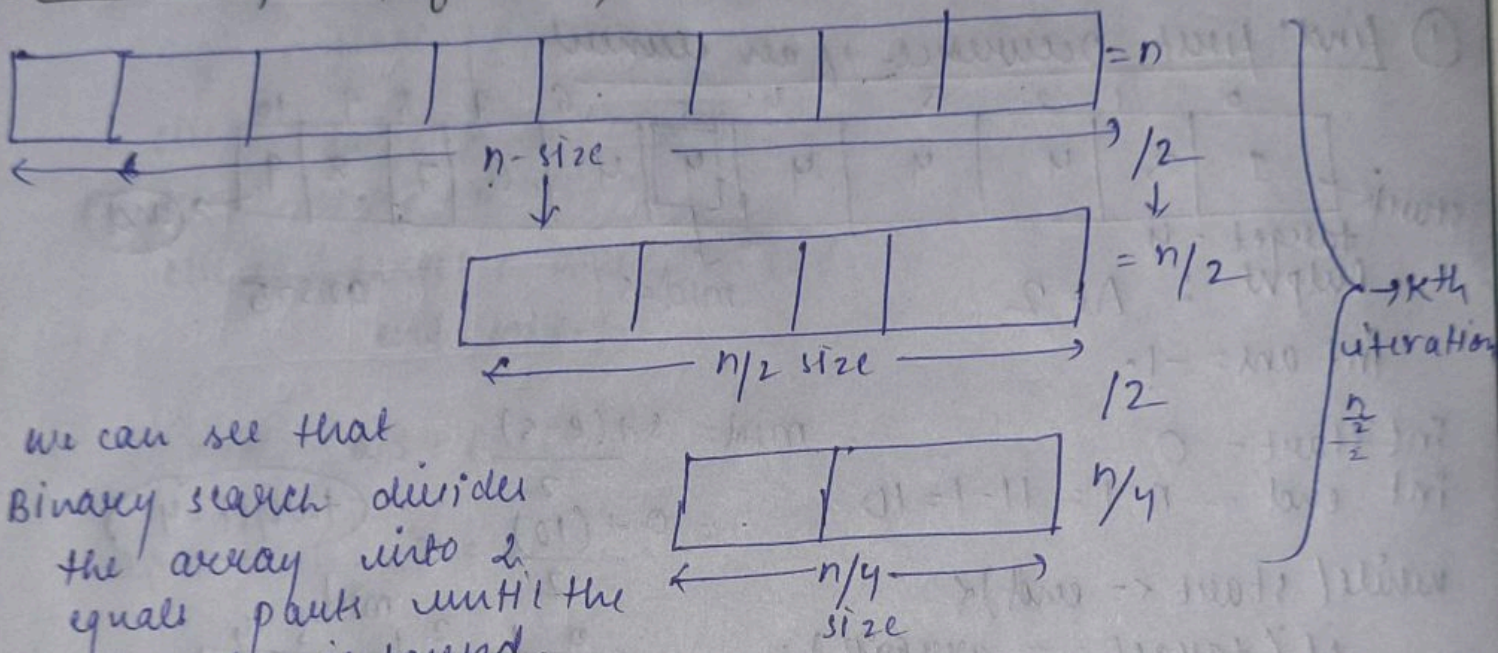
As $mid = \frac{(e+s)}{2} = \frac{2^{31}-1+2^{31}-1}{2}$ > range of the integer.

so we will use.

$mid = s + \frac{(e-s)}{2} = (2^{31}-1) + \frac{(2^{31}-1-2^{31}+1)}{2}$

$\boxed{mid = 2^{31}-1}$ max range of the integer

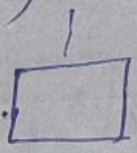
* Time complexity of Binary search:



we can see that Binary search divides the array into 2 equal parts until the element is found.

$O(K)$ → no. of times being divided (K times)

↳ let K be the Time complexity of B.S



$\therefore \frac{n}{2^K} = 1$

$n = 2^K$

Now replace the K with $\log n$

$\log n = K \log 2$

$K = \log n$ → $O(\log n)$

∴ T.C of Binary search is $O(\log n)$

$1024 \xrightarrow{/2} 512 \xrightarrow{/2} 256 \xrightarrow{/2} 128 \xrightarrow{/2} 64 \xrightarrow{/2} 32 \xrightarrow{/2} 16 \xrightarrow{/2} 8 \xrightarrow{/2} 4 \xrightarrow{/2} 2 \xrightarrow{/2} 1$

10 comparison me ho gya.

* Inbuilt function for Binary search:

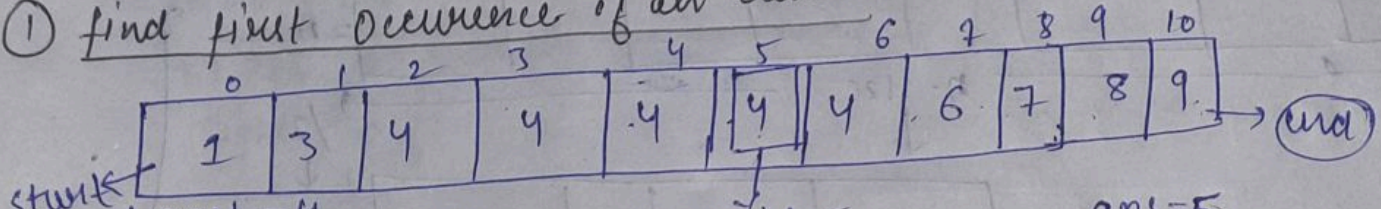
```
#include <algorithm>
binary_search ( arr, arr+n, target)
                 ↑          ↳ last index.
              base address
```

arr = 104
arr+1 = 108
↓
104 × 1 × 4 = 108
int = 4 bytes.

for vectors

```
#include <algorithms>
binary_search (v.begin(), v.end(), target);
vector<int> v;
```
(vector)

① find first occurrence of an element



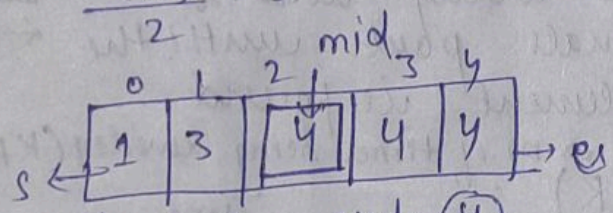
target = 4
Output : At 2
mid = 5
ans = 5

```
int ans = -1;
int start = 0
int end = n-1 = 11-1 = 10
while( start <= end){
    if( target == arr[mid] ){
        ans = mid;
        end = mid-1;
    }
elseif( target < arr[mid]){
        end = mid-1)
    }
else {
        start = mid+1;
    }
mid = s+(e-s)/2;
}
return ans;
}
```

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

$= 0 + \frac{(10)}{2} = 5$

target = 4

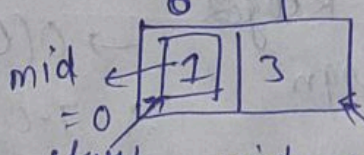


mid = $s + (e-s)/2$

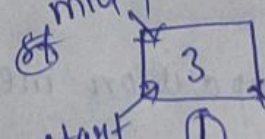
$= 0 + \frac{(4-0)}{2}$

ans = 2

end = 2-1



mid = $\frac{0+1}{2} = 0$



start = 1+1 = 2
end = 1

s = e+1 = 2
end

return ans;

ans = 2.

first occurence will be 2

(2) find last occurence of an element:

everything will be same.

just

while (start <= end) {

if (target == arr[mid]) {

ans = mid;

start = mid+1;

}

else if (target > arr[mid]) {

start = mid+1;

else if (target < arr[mid]) {

end = mid-1;

}

mid = s + (e-s)/2;

}

return ans;

}

(3) Total no. of Occurence:

0	1	2	3	4	5	6	7	8	9
2	4	4	4	4	4	4	6	8	10

first occurence is at 1

last occurence is at 6

Total occurence = first Occurence ~~of~~ last occurence +1;

merge both the first & last occurence & print total ~~occ~~ occurence.