Searching

- Searching is the process of finding some element in the array.

- If the element is present in the array, then the process is called successful and the process returns the location of that element, otherwise the search is called unsuccessful.

- There are two popular search methods:
  1. Linear Search
  2. Binary Search

Linear Search

Algorithm of Linear Search:

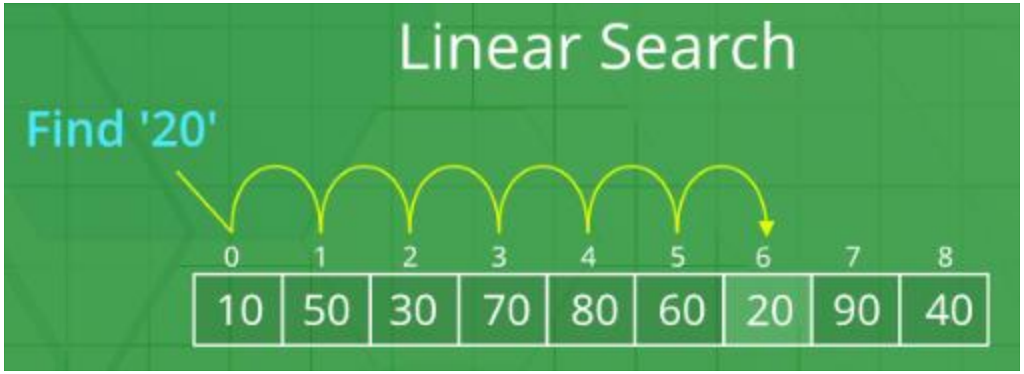
```
Linear_search(arr[],n,x)
{
    for i=0 to n-1
        if(arr[i] == x)
            return i
    return -1
}
```

Time Complexity:

Best Case: O(1)

Average Case: O(n)

Worst Case: O(n)



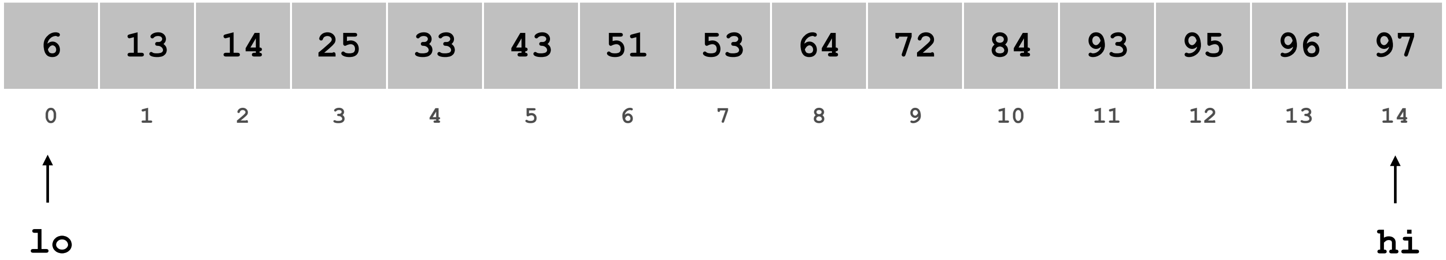
0	1	2	3	4	5	6	7	8
10	50	30	70	80	60	20	90	40

Binary Search

Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.

Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.

Ex. Binary search for 33.



6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Algorithm: Binary Search

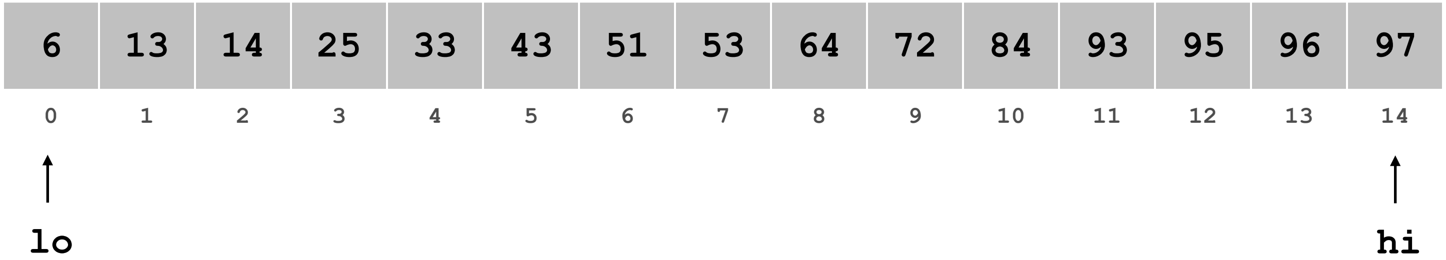
```
// Precondition: Elements of a are in sorted order

int binarySearch(a[], target, min, max)
{
   if (min > max)
        return -1;      // target not found
   else
   {
      int mid = (min + max) / 2;

      if (a[mid] < target) // too small; go right

           return binarySearch(a, target, mid + 1, max);

      else if (a[mid] > target) // too large; go left

           return binarySearch(a, target, min, mid - 1);

      else

           return mid;   // target found; a[mid] == target
   }
}
```

Binary Search

Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i] = value`, or report that no such index exists.

Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.

Ex. Binary search for 33.



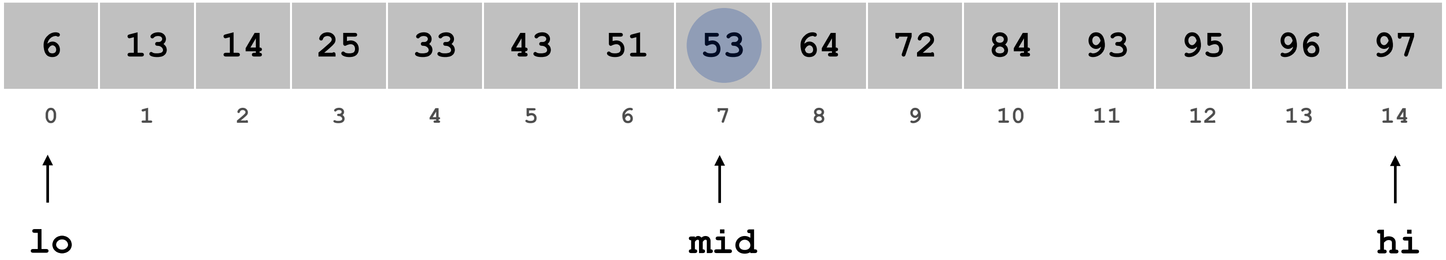
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Binary Search

Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i] = value`, or report that no such index exists.

Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.

Ex. Binary search for 33.



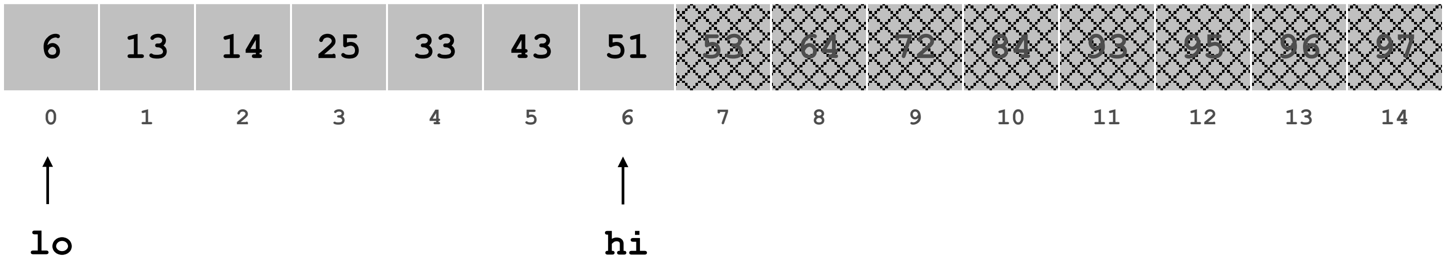
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Binary Search

Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i] = value`, or report that no such index exists.

Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.

Ex. Binary search for 33.



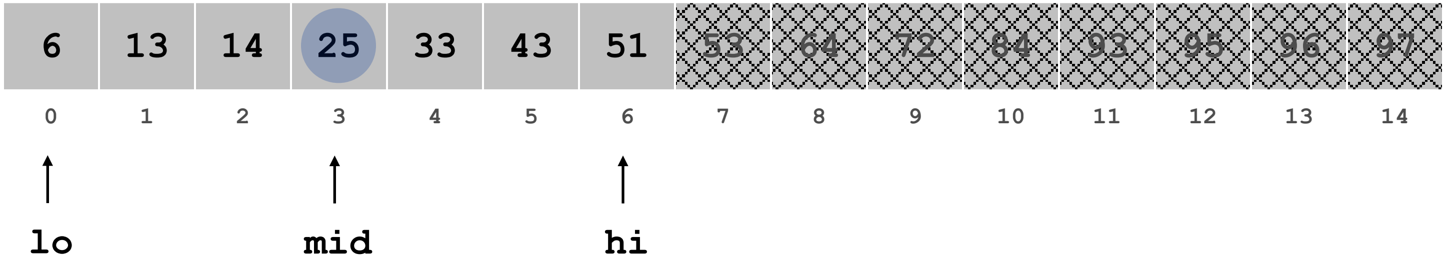
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Binary Search

Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i] = value`, or report that no such index exists.

Invariant. Algorithm maintains `a[lo]` $\leq$ `value` $\leq$ `a[hi]`.

Ex. Binary search for 33.

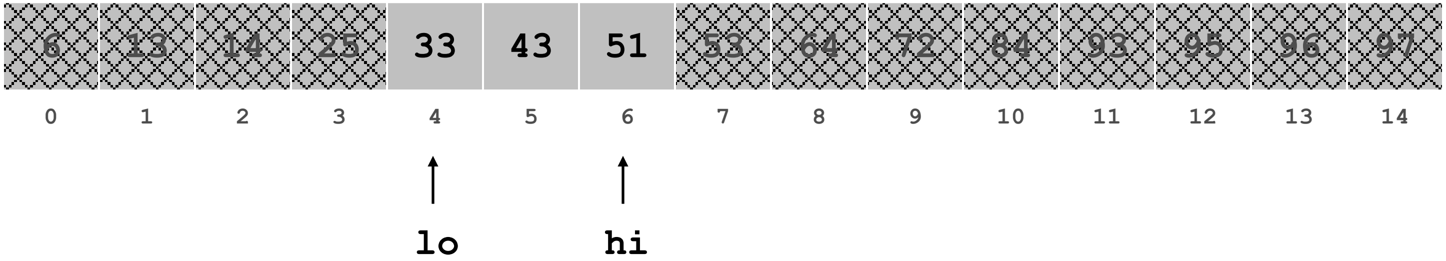


Binary Search

Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i] = value`, or report that no such index exists.

Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.

Ex. Binary search for 33.



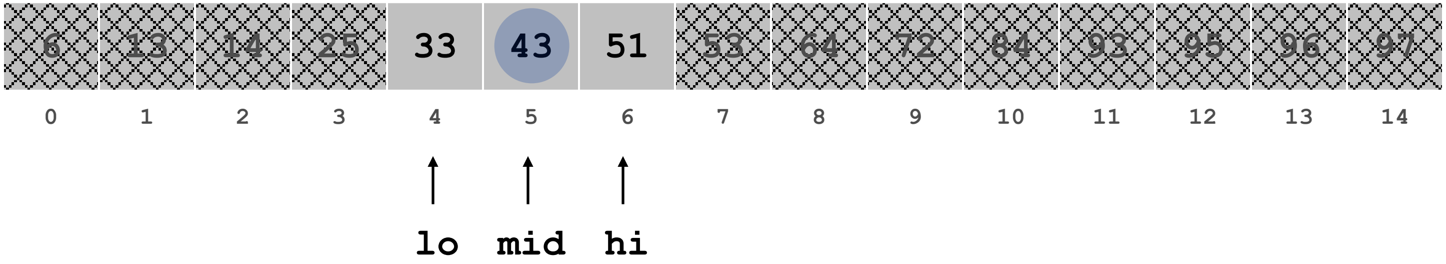
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Binary Search

Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i] = value`, or report that no such index exists.

Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.

Ex. Binary search for 33.

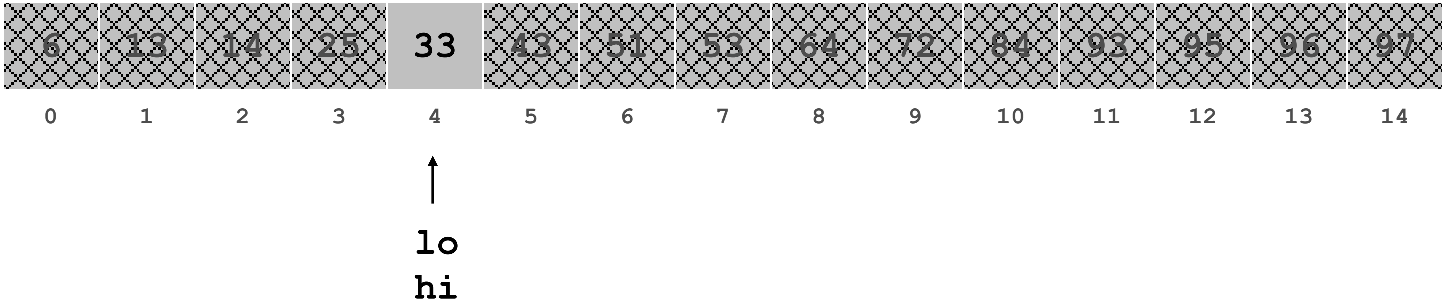


Binary Search

Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i] = value`, or report that no such index exists.

Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.

Ex. Binary search for 33.



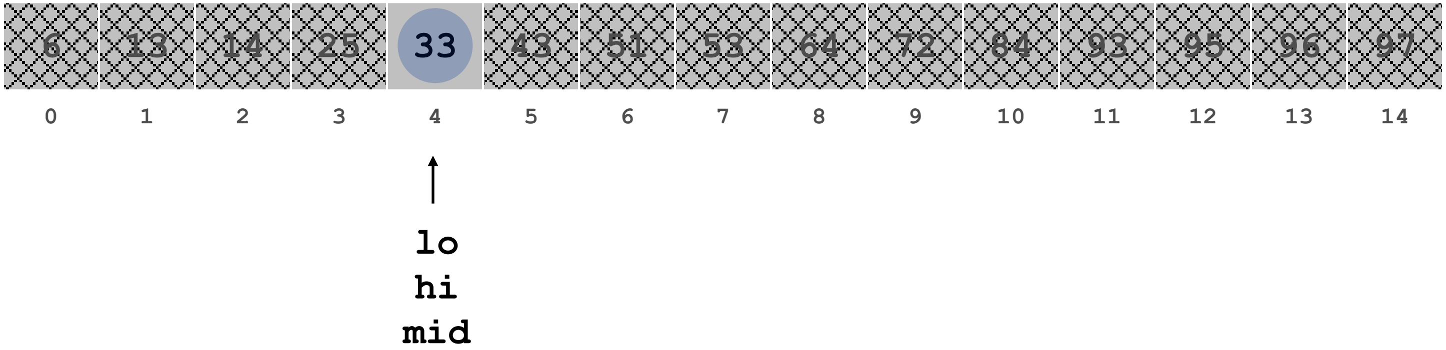
6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Binary Search

Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i]` = `value`, or report that no such index exists.

Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.

Ex. Binary search for 33.



6	13	14	25	33	43	51	53	64	72	84	93	95	96	97
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

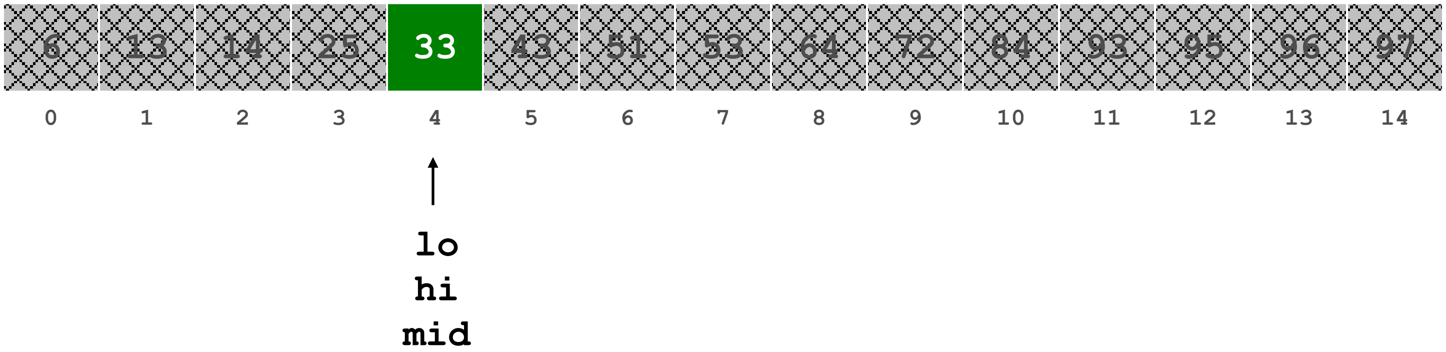
lo
hi
mid

Binary Search

Binary search. Given `value` and sorted array `a[]`, find index `i` such that `a[i] = value`, or report that no such index exists.

Invariant. Algorithm maintains `a[lo]` ≤ `value` ≤ `a[hi]`.

Ex. Binary search for 33.



Time Complexity: Binary Search

Recurrence relation:

$$T(n) = \begin{cases} c & \text{if } i >= j. \\ T(n/2) + c & \text{otherwise.} \end{cases}$$

Where c is constant.

So, T(n) = T(n/2) +1

Time Complexity:

Best Case: O(1)

Average Case: O(logn)

Worst Case: T(n)=T(n/2)+1, By using master Theorem O(logn)

Q: How many elements will it need to examine?

A: O(log N)

Note: Elements are in sorted order