BRAC UNIVERSITY

Inspiring Excellence

# CSE221

# Algorithms: *Binary Search*

Summer 2020

# Binary Search

- A divide and conquer algorithm
- Does not search the entire array like linear search
- Given an array of **sorted elements** and the item, k, to search for, we first check if the middle element matches k.
- If we find then bingo! Else we check if k > middle element or k < middle element? If the first condition is true we look into the right half of the array. If the second condition is true we look into the left half of the array.
- Note that the red marked conditions are true if and only if the array is sorted.

# Binary Search

## Pseudo code (Iterative)

```
boolean binarySearch(A[], l , r, item){
        while (l <=r){
                mid = (l +r)/2;
                if (A[mid]==item){
                        return true;
                }else{
                        if (item<A[mid]){
                                r = mid-1;
                        }else{
                                l = mid+1;
                        }
                }
        }
        return false;
}
```

*l* must always be less or equal to r for the loop to run

Find the mid index

Check if item matches with the mid element, return true if does else next step.

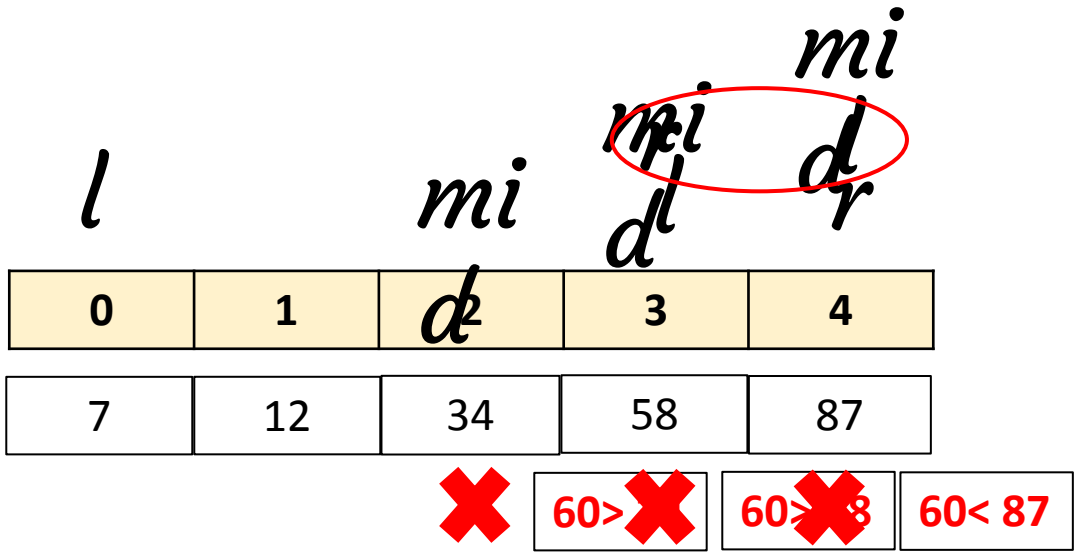
Check if the item is smaller, if yes shift the search range to left by moving r

If the above condition is dissatisfied, shift the search range to right by moving *l*

The program will reach this line after while loop is complete. False is returned.

# Binary Search

## Simulation




```
boolean binarySearch(A[], l , r, item){
      while (l <=r){
            mid = (l +r)/2;
            if (A[mid]==item){
                  return true;
            }else{
                  if (item<A[mid]){
                        r = mid-1;
                  }else{
                        l = mid+1;
                  }
            }
      }
      return false;
}
```

# Binary Search

## Time Complexity (Iterative)

In order to compute the time complexity of divide and conquer problems we will focus on the problem size and **work done** at every step. Work done refers to finding the mid, matching, changing left or right. These are constant operations. Refer to the previous slide. You will see that at each step the problem size (range) was reducing by factor of 2.

Initially the domain of search was the entire array of length ***n***. In the second step we only considered the right half. The domain got reduced to ***n/2***. In the following steps this sub problem was getting smaller by factor of 2; ***n/4, n/8, …*** until 1 or 0. We need to count how many steps it took for the problem of size ***n*** to become 0 or 1 then multiply it by work done at each step.

| Problem Size | Step No. |
|---|---|
| n | 0 |
| n/2 | 1 |
| n/4 | 2 |
| n/8 | 3 |
| …. | …. |
| 1 | k |

# Binary Search

## Time Complexity (Iterative) Contd.

| Problem Size | Step No. | Work done at each step |
|---|---|---|
| n | 0 | 1 |
| n/2 | 1 | 1 |
| n/4 | 2 | 1 |
| n/8 | 3 | 1 |
| …. | …. | 1 |
| 1 | k | 1 |

It took ***k*** steps for the searching to end and work done at each step was constant. We need to find ***k*** in terms of ***n.*** If you notice the divisors of the problem size (marked red) are all powers of 2 and we can use the step numbers as exponents. Therefore each problem size can be written as,

***n/2^(step no.)***

The last line can be written an, ***1 = n/2^k***.

If we solve it, we will find that $k = \log_2 n$

Therefore the time complexity is $\log_2 n$ x ***1***, which eventually is O($\log_2 n$)

# Binary Search

## Recursive

```
boolean binarySearch(A[], l , r, item){
    if (l <=r){
        mid = (l +r)/2;
        if (A[mid]==item){
            return true;
        }else{
            if (item<A[mid]){
                r = mid-1;
                return binarySearch(A, l , r, item)
            }else{
                l = mid+1;
                return binarySearch(A, l , r, item)
            }
        }
    }
    return false;
}
```

The core idea of the search is same; the while loop gets replaced by method calls.

The underlying mechanism of recursion (use of stack, return) is shown in the *linear search lecture*. Follow the same technique.

# Binary Search

## Time Complexity (Recursive)

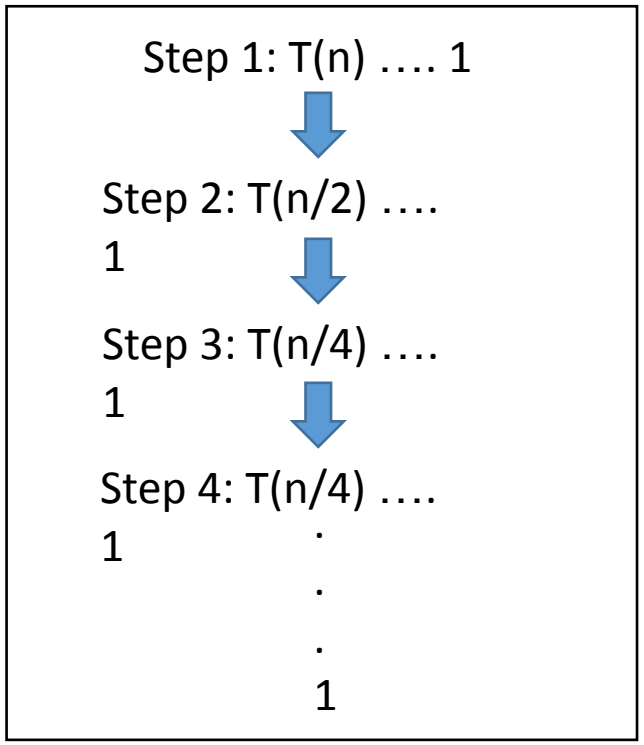
For recursive problems you need to have or find out the recurrence equation; then solve it to find the running time. Let us find the recurrence equation of binary search step by step.

| | |
|---|---|
| 1. | Recursion starts with a method call where we pass the array. Let the name of the method be T and the size of the array be n. As we pass n as a parameter, we can write T(n). Note that there were other parameters too but we know that the time complexity is dependant on input. |
| 2. | T is the name of the method and now we need to find the work of the method. Recall that binary search divides the problem (array) of size n in 2 halves. We can write **T(n) = n/2 + n/2**. We only consider only one of the portions so we rewrite **T(n) = n/2**. On the split we find the mid, check with the mid, change left or right. These are constant operations. Thus **T(n) = n/2 + 1**. Lastly that half portion is further broken down in 2 and this is done by calling the same binarySearch method. That is the method T is invoked on n/2. Thus we can write **T(n) = T(n/2)+1**. |

# Binary Search

## Time Complexity (Recursive)

T(n) = T(n/2) + 1 There are a number of techniques to solve recurrence equations but we will focus on the tree method. The tree below is exactly like the one we drew while computing the running time of the iterative Version of the problem. Therefore the method of finding the solution from the tree is exactly the same.




It took ***k*** steps for the searching to end and work done at each step was constant. We need to find ***k*** in terms of ***n.*** Each problem size can be written as,
***n/2^(step no.)***
The last line can be written an, ***1 = n/2^k***.
If we solve it, we will find that $k = \log_2 n$
Therefore the time complexity is $\log_2 n$ x ***1***, which eventually is O($\log_2 n$)