11. BINARY SEARCH

A binary search is efficient algorithm to find the position of a target (key) value within a sorted array.

- The binary search algorithm begins by comparing the target value to the value of the middle element of the sorted array. If the target value is equal to the middle element's value, then the position is returned and the search is finished.
- If the target value is less than the middle element's value, then the search continues on the lower half of the array.
- if the target value is greater than the middle element's value, then the search continues on the upper half of the array.
- This process continues, eliminating half of the elements, and comparing the target value to the value of the middle element of the remaining elements - until the target value is either found (position is returned).

Binary search is a remarkably efficient algorithm for searching in a sorted array (Say A). It works by comparing a search key K with the array's middle element A[m]. If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if K <A[m], and for the second half if K>A[m]:

Though binary search is clearly based on a recursive idea, it can be easily implemented as a non-recursive algorithm, too. Here is pseudocode of this non recursive version.

ALGORITHM *Binary Search (A*[0..*n* − 1]*, K)*

//Implements non recursive binary search

//Input: An array *A*[0..*n* − 1] sorted in ascending order and a search key *K*

//Output: An index of the array's element that is equal to *K*/ or −1 if there is no such element

l ← 0; *r* ← *n* − 1

while *l* ≤ *r* **do**

m ← ⌊(l + r)/2⌋

if *K* = *A*[*m*] **return** *m*

else if*K* <*A*[*m*]

r ← *m* − 1

else *l* ← *m* + 1

return −1

The standard way to analyze the efficiency of binary search is to count the number of times the search key is compared with an element of the array (three-way comparisons). One comparison of K with A[m], the algorithm can determine whether K is smaller, equal to, or larger than A[m].

As an example, let us apply binary search to searching for *K* = 70 in the array. The iterations of the algorithm are given in the following table:

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	3	14	27	31	39	42	55	70	74	81	85	93	98
iteration1	*l*						*m*						*r*
iteration2													*r*
								l		*m*			

iteration 3 *l,mr*

The worst-case inputs include all arrays that do not contain a given search key, as well as some successful searches. Since after one comparison the algorithm faces the same situation but for an array half the size,

The number of key comparisons in the worst case $C_{worst}(n)$ by recurrence relation.

orct(n) = orct(n[]) + 1 fo n > 1, orct(1) =1.

orct*(n)*= log2n]+1=]log2(n+1)] orct *(2k)* = (*k* + 1) = log2 k + 1 forn=2*k*

- First, The worst-case time efficiency of binary search is in Θ(logn).
- Second, the algorithm simply reduces the size of the remaining array by half on each iteration, the number of such iterations needed to reduce the initial size *n* to the final size 1 has to be about $\log_2 n$.
- Third, the logarithmic function grows so slowly that its values remain small even for very large values of*n*.

The average case slightly smaller than that in the worst case

$$C_{avg}(n) \approx \log_2 n$$

The average number of comparisons in a successful is

$$\boldsymbol{C_{avg}(n) \approx \log_2 n - 1}$$

The average number of comparisons in an unsuccessful is

$$\boldsymbol{C_{avg}(n) \approx \log_2 (n + 1).}$$