DIVIDE AND CONQUER

AOA : Module 2

CONTENTS

INTRODUCTION

- Original problem is divided into <u>similar kind of subproblems</u> that are smaller in size and easy to be find.
- The solution of these small independent subproblems are combined to obtain the solution of whole problem.
- Divide and Conquer paradigm solves a problem in three steps at each level of recursion:
  1. Divide
  2. Conquer
  3. Combine

INTRODUCTION

- Time complexity to solve "Divide & Conquer" problem is given by recurrence relations.
- Recurrence relation is derived from algorithm and solved to calculate complexity.
- The general recurrence relation for divide and conquer is given as follows:

$$\boldsymbol{T(n) = aT\left(\frac{n}{b}\right) + f(n)}$$

Where, T(n/b) : time required to solve each subproblem

f(n) : time required to combine the solutions of all subproblems

BINARY SEARCH

- There are two approaches:
  1. Iterative or Non-recursive
  2. Recursive
- There is a linear Array 'a' of size 'n'.
- Binary Search is one of the fastest searching algorithm.
- Binary Search can only be applied on "Sorted Arrays"- either ascending or descending order.
- We compare "key" with item in the middle position. If they are equal, search ends successfully.
- Otherwise,

  if key is less than element present in the middle position,

  then apply binary search on lower half,

  else apply BINARY SEARCH on upper half of the array.
- Same process is applied to remaining half until match is found or there are no more elements left

BINARY SEARCH

Iterative Approach:

```
Algorithm IBinaryS(arr[ ], start, end, key){
        int mid;
        while(start<=end){
                mid = (start + end)/2;
                if (arr[mid] == key)
                        return 1;
                if (arr[mid]<key)
                        start = mid+1;
                else
                        end = mid-1;
        }
        return 0;
}
```

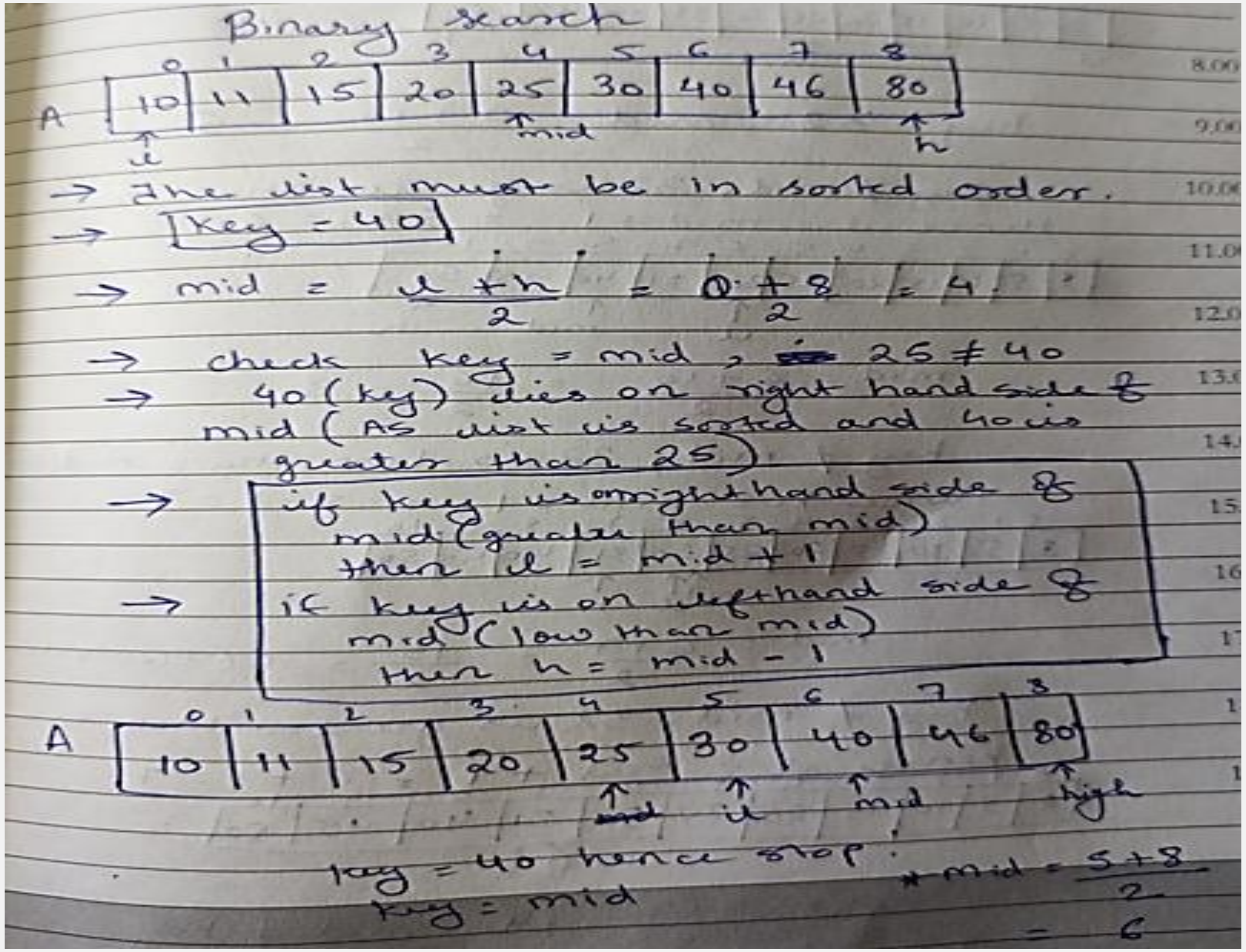
BINARY SEARCH

Recursive Approach:

```
Algorithm RBinaryS(arr[ ], start, end, key){……….T(n)
    int mid;
     if (start > end)(list is empty)
     {
             return 0;
     }
     Else
     {
          mid = (start + end)/2;…………O(1)
          if (key == arr[mid])…………O(1)
          return (mid);
          else
          if (key < arr[mid]){………O(1)
          RBinaryS(arr[],key, start, mid-1)……….T(n/2)
          else                                      or
          RBinaryS(arr[],key, mid+1, end)………………T(n/2)
     }
}     T(n)= T(n/2) + 1 = O(logn) (Solve using masters theorem)
```

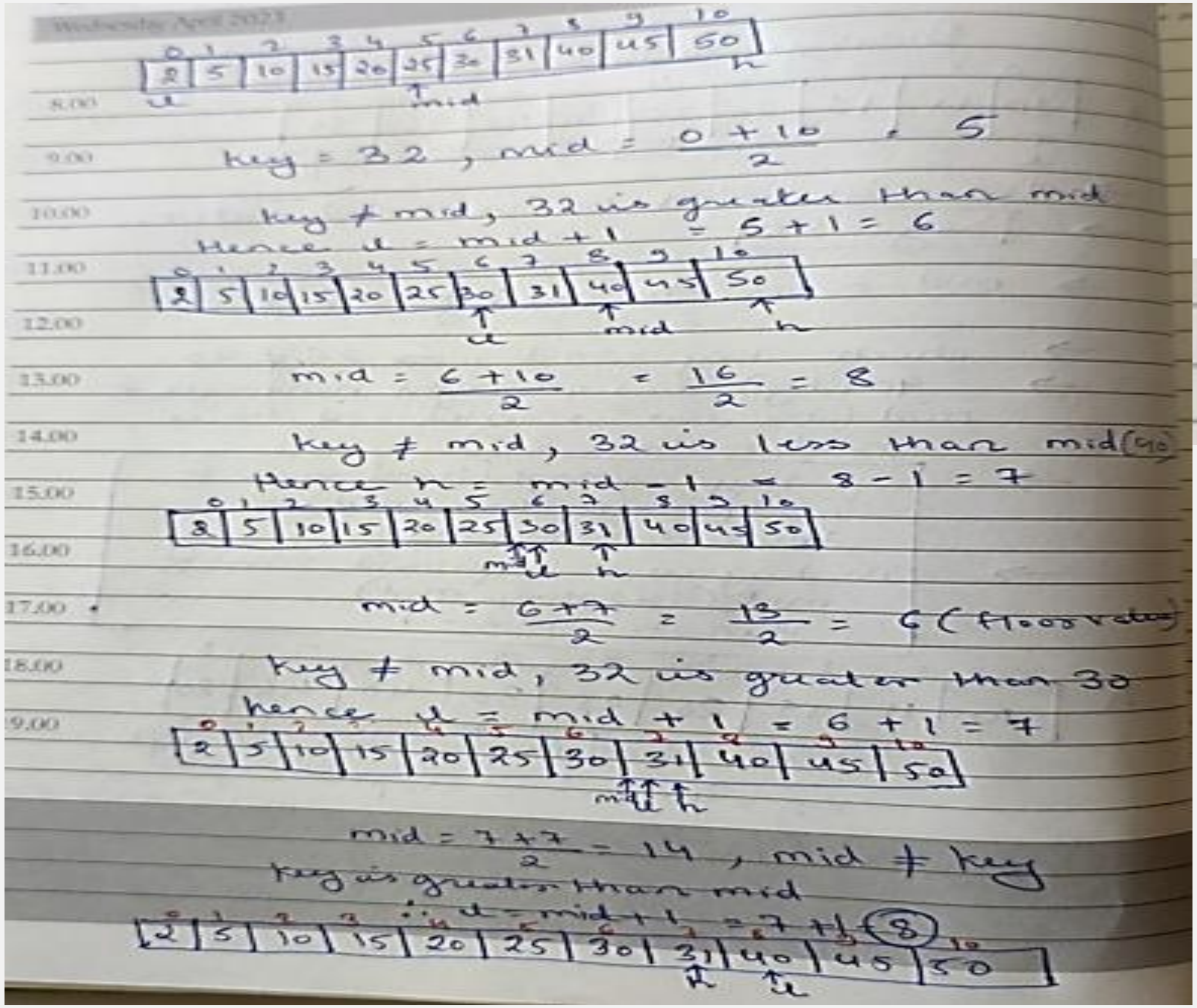
EXAMPLE OF BINARY SEARCH

- Example 1



EXAMPLE OF BINARY SEARCH

- Example 2:



0	1	2	3	4	5	6	7	8	9	10
2	5	10	15	20	25	30	31	40	45	50
l					mid					h

key = 32, mid = $\frac{0+10}{2}$ = 5

key ≠ mid, 32 is greater than mid

Hence l = mid + 1 = 5 + 1 = 6

0	1	2	3	4	5	6	7	8	9	10
2	5	10	15	20	25	30	31	40	45	50
						l		mid		h

mid = $\frac{6+10}{2}$ = $\frac{16}{2}$ = 8

key ≠ mid, 32 is less than mid(40)

Hence h = mid − 1 = 8 − 1 = 7

0	1	2	3	4	5	6	7	8	9	10
2	5	10	15	20	25	30	31	40	45	50
						mid l	h			

mid = $\frac{6+7}{2}$ = $\frac{13}{2}$ = 6 (floor value)

key ≠ mid, 32 is greater than 30

hence l = mid + 1 = 6 + 1 = 7

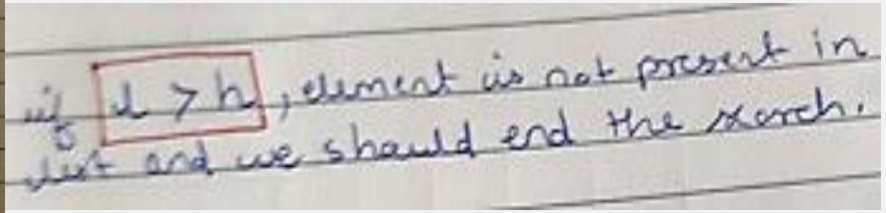
0	1	2	3	4	5	6	7	8	9	10
2	5	10	15	20	25	30	31	40	45	50
							mid l h			

mid = $\frac{7+7}{2}$ = 14, mid ≠ key

key is greater than mid

∴ l = mid + 1 = 7 + 1 = 8

0	1	2	3	4	5	6	7	8	9	10
2	5	10	15	20	25	30	31	40	45	50
							h	l		

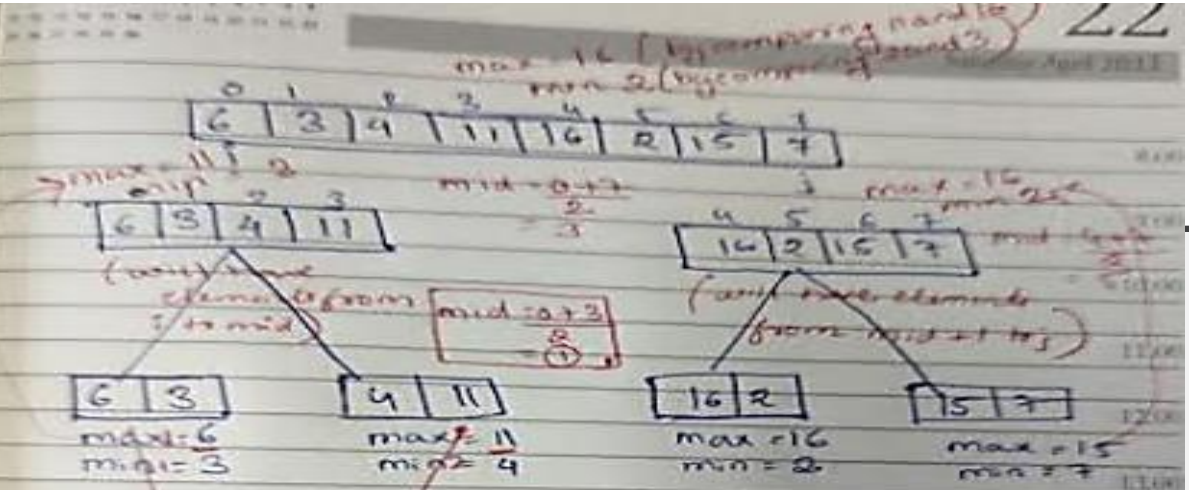


if $\boxed{l > h}$, element is not present in list and we should end the search.

FINDING MINIMUM AND MAXIMUM

Iterative Approach:

```
Algorithm MinMax(a[ ], n, max, min){
        max=min=a[1];
        for(i=2 to n)do
{
        if(a[i]> max) then max=a[i];
        if (a[i]< max) then min=a[i];
}
}
```



→ We divide the array until there are 2 elements in every subarray.

→ From the array of 2 elements, we find out max and min elements by comparing 2 elements of array.

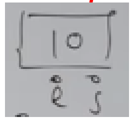
Compare max1 with max2 whichever is maximum will be the max of parent array. Similarly compare min1 with min2 and choose the minimum. Therefore every pass requires 2 comparison if there are more than 2 elements.

FINDING MINIMUM AND MAXIMUM

Recursive Approach:

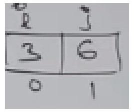
Algorithm MinMax(a[ ],l,h,max,min)..*T(n)* {

if(i==j) then *(only 1 element present in array hence 0 comparison)*



max=min=a[i];

else if(i==j-1), then



{

if(a[i]>=a[j]), then *(2 elements present in array hence 1 comparison)……O(1)(we ignore this as smaller problem)*

max=a[i];

min=a[j];

else{

max=a[j];

min=a[i];

}

else{

Mid = (i+j)/2;

MinMax(a[],i, mid,max1,min1);………*T(n/2)*

MinMax(a[],mid+1,j,max2,min2);...*T(n/2)*

if(a[max1]< a[max2]) then *(more than 2 elements present in array hence 2 comparisons)……O(1)*

max=max2;

Else

max=max1;

if (a[min1]<a[min2])then,.......*O(1)*

min = min1;

Else

min = min2;

}

}

T(n)= 2T(n/2)+2

FINDING MINIMUM AND MAXIMUM TIME COMPLEXITY

Time complexity :- We need to form recurrence relation.

i) when $n=1$, ie single element in array. Hence No comparison required.

$$\therefore T(n) = 0 \quad \text{for } n=1$$

when $n=2$; ie. 2 elements are in array, There ~~are~~ is ~~2~~ 1 comparisons required.

$$\therefore T(n) = 1, \quad \text{for } n=2$$

For all other cases, we are dividing array into 2 halves. and calling the function recursively twice. ~~After~~ Within that there are 2 comparisons.

$$\therefore T(n) = T(n/2) + T(n/2) + 2 \quad \text{for } n>2$$

$$T(n) = 2T(n/2) + 2$$

FINDING MINIMUM AND MAXIMUM

Time Complexity:(solve it using Masters theorem or substitution method)

Recurrence relation :

$$T(n) = 2T\left(\frac{n}{2}\right) + 2 \qquad n > 2$$
$$T(n) = 1 \qquad n = 2$$
$$T(n) = 0 \qquad n = 1$$

MERGE SORT

- Simple and efficient algorithm for sorting a list of numbers
- Based on divide and Conquer paradigm
- Performed in three steps:

1. <u>Divide:</u>
   i. List of n elements is divided into 2 sub-lists of n/2 elements
   ii. Computes middle of the array, so it takes constant time O(1).
2. <u>Conquer:</u>
   1. Each half is sorted independently.
   2. Merge sort is recursively used to sort elements of smaller sub-lists.
   3. This step contributes T(n/2) + T(n/2) to running time.

MERGE SORT

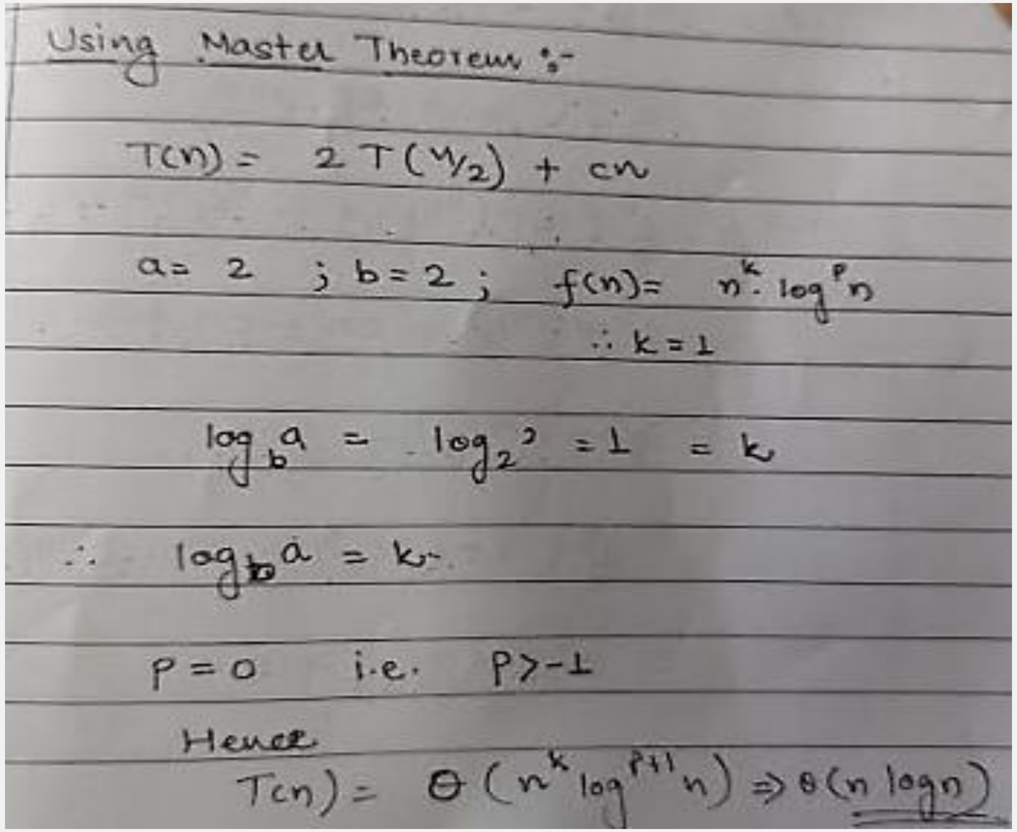
3. Combine:
   1. Two sorted halves are merged to obtain a sorted sequence
   2. This requires merging of n elements into 1 list.
   3. It contributes O(n) to running time.

NOTE: The Key operation of merge sort is Merging

MERGE SORT ALGORITHM

```
mergeSort(arr[ ],low, high)…….T(n)
//arr is array, low is left sub-list, high is right sub-list
{
    if(low<high)
    {
        mid = (low+high)/2;………O(1)
        mergeSort(arr, low, mid);……T(n/2)
        mergeSort(arr,mid+1,high);…….T(n/2)
        merge(arr, low, mid, high);….O(n)
    }
}
```

$T(n) = 2T(n/2)+n = O(n\log n)$



Using Master Theorem :-

$T(n) = 2T(n/2) + cn$

$a = 2$; $b = 2$; $f(n) = n^k \cdot \log^p n$

$\therefore k = 1$

$\log_b a = \log_2 2 = 1 = k$

$\therefore \log_b a = k$.

$p = 0$ i.e. $p > -1$

Hence

$T(n) = \Theta(n^k \log^{p+1} n) \Rightarrow \Theta(n \log n)$

Using Substitution Method :-

$T(n) = 2T(n/2) + cn$ —①

$T(n/2) = 2T(n/4) + c \cdot n/2$ —②

$T(n/4) = 2T(n/8) + c \cdot n/4$ —③

$T(n/8) = 2T(n/16) + c \cdot \frac{n}{8}$ —④

$T(n) = 2\left[2T(n/4) + c \cdot \frac{n}{2}\right] + cn$

$= 2^2 T(n/4) + cn + cn$

$T(n) = 2^2\left(2T(n/8) + c \cdot n/4\right) + cn + cn$

$= 2^3 T(n/8) + cn + cn + cn$

$= 2^3 T(n/2^3) + 3cn$

$T(n) = 2^3\left(2T(n/16) + c\,n/8\right) + 3cn$

$= 2^4 T(n/16) + 4cn$

$T(n) = 2^k \cdot T(n/2^k) + kcn$

let $T(1) = 0$

$\frac{n}{2^k} = 1$

$n = 2^k \Rightarrow k = \log_2 n$

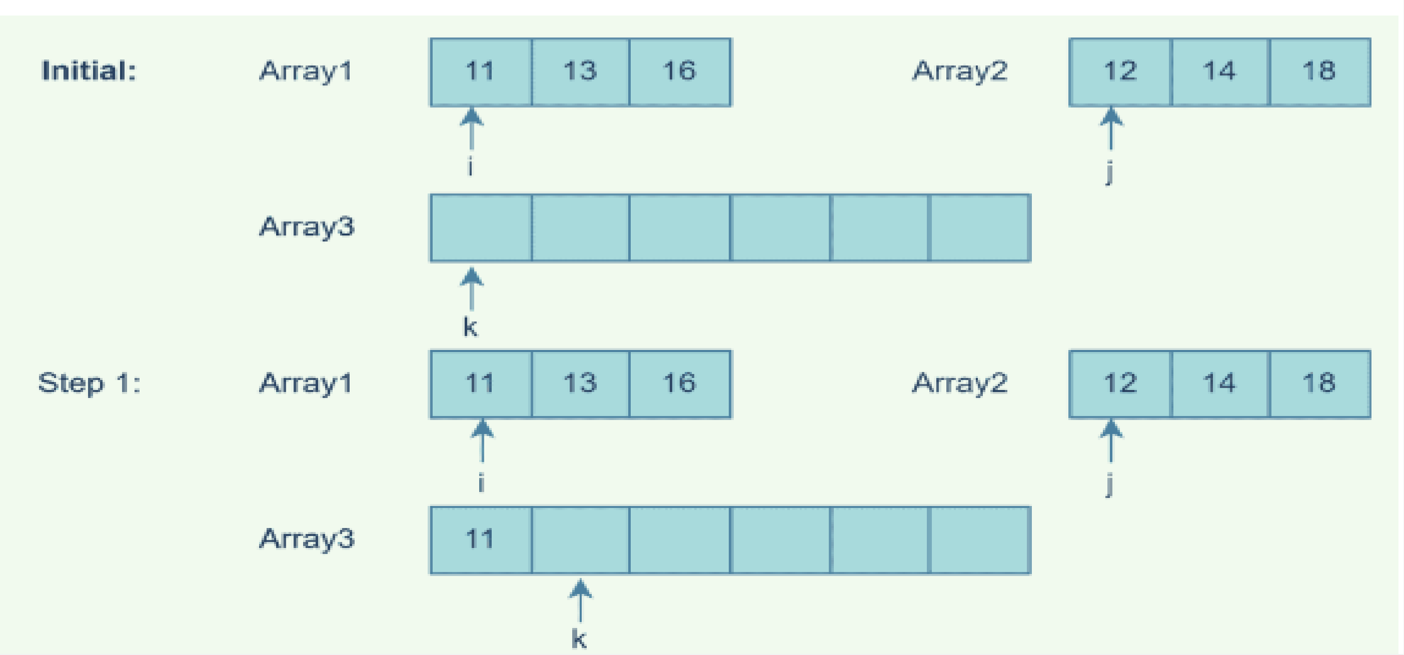
$T(n) = 2^k \cdot 0 + c \cdot n \cdot \log_2 n$

$= cn \log_2 n$

$= \boxed{O(n \log_2 n)}$

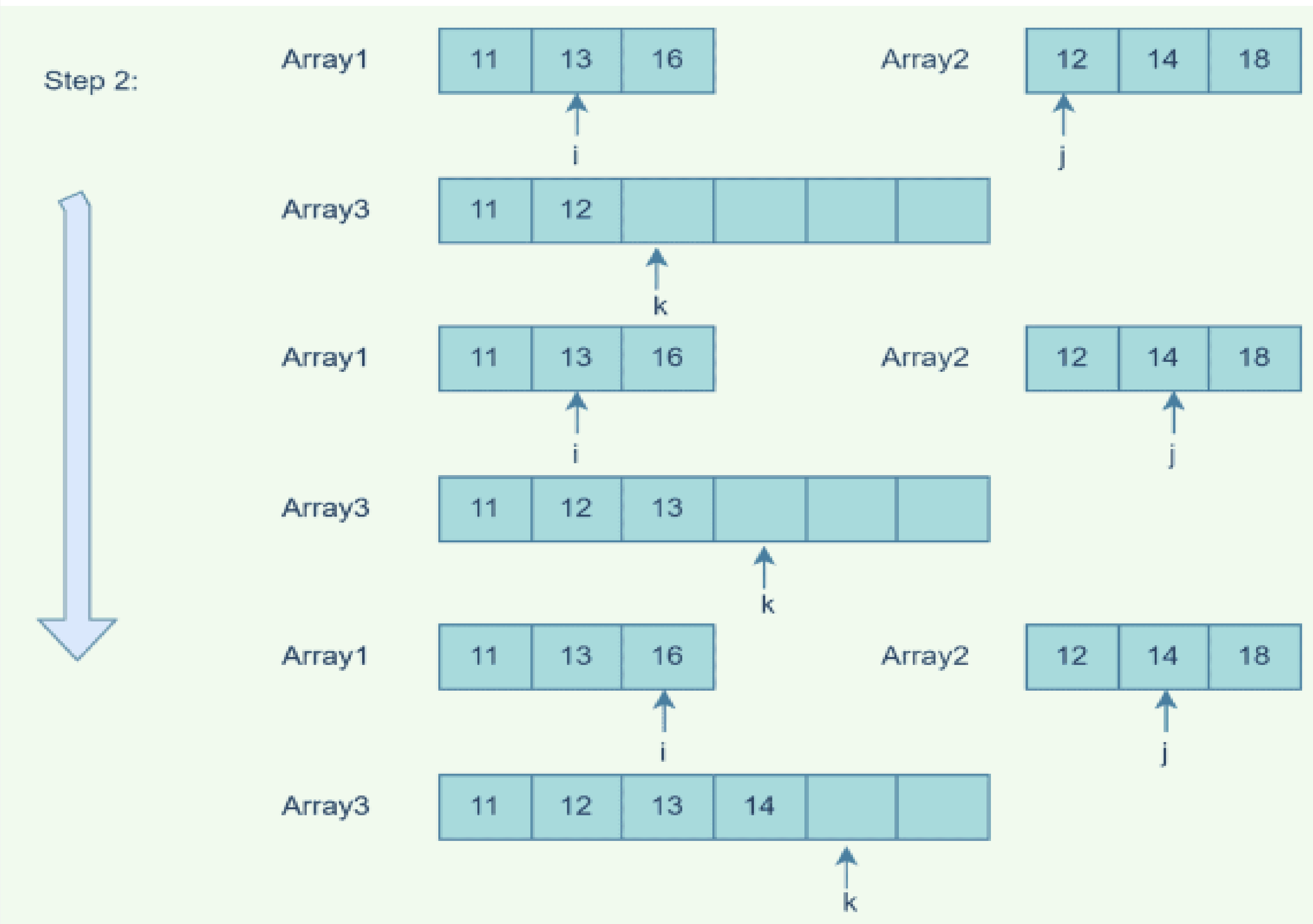
TWO-WAY MERGE

- A two-way merging, also known as binary merging, is generally an algorithm that takes two sorted lists and merges them into one list in sorted order.
- If we are merging two arrays size m and n respectively, then the merged array size will be the sum of m+n . Merging requires maximum (m + n) comparisons to get the merged array.
- We compare the first items in two arrays pointed by i and j and append the smaller element to the output array (Array3) pointed by k. As i<j , the element pointed by i from Array1 is copied to output array and i is incremented as well as k.



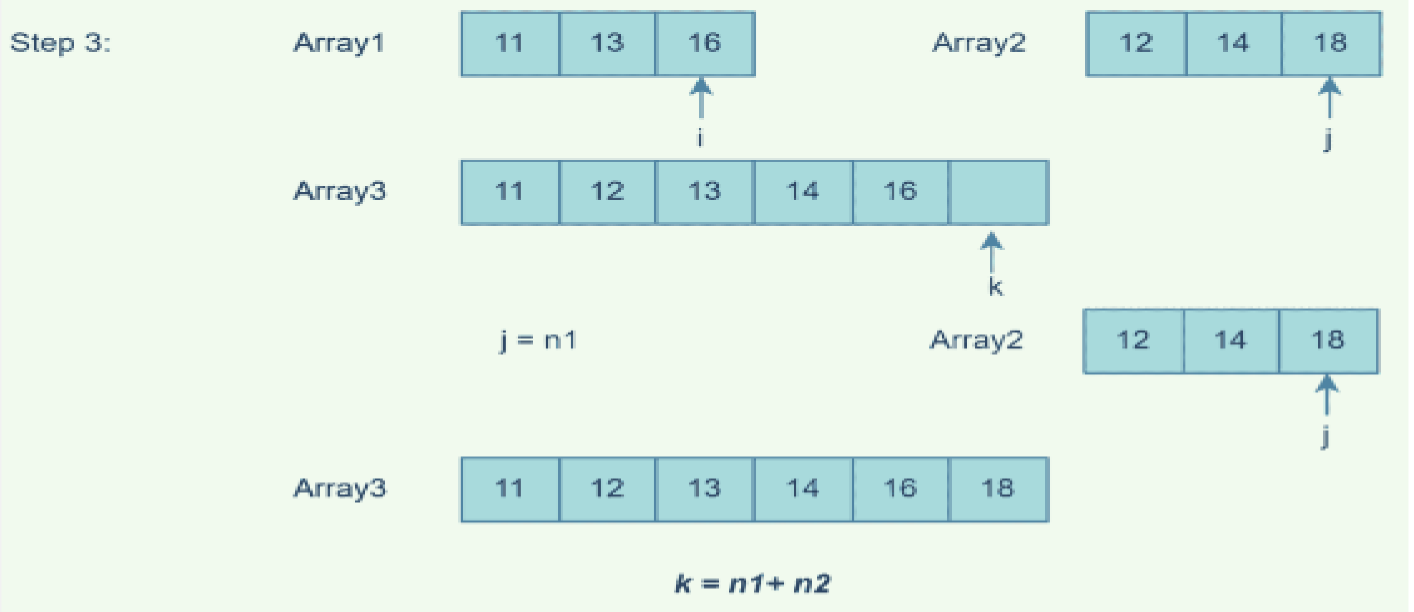
TWO-WAY MERGE

- After appending the smaller item to our output array, we continue comparing and appending them to array3 until the input arrays are empty.



TWO-WAY MERGE

- At this point, array1 is empty. The other input array must still be empty, so we take the smaller item in array2 and append it to array3. We repeat this process until array2 is empty.



- The time complexity of merging is $\theta(m+n)$. Where m are elements of Array1 and n are elements of Array2

MERGE ALGORITHM

```
void merge(int arr[ ], int low,
int mid, int high) {
  int i = low;
  int j = mid + 1;
  int k = low;
```

1

```
while (i <= mid && j <= high) {
    if (arr[i] <= arr[j]) {
      temp[k] = arr[i];
      i++;
      k++;
    }
else {
      temp[k] = arr[j];
      j++;
      k++;
    }
  }
```

2

MERGE ALGORITHM

```
/* Copy the remaining elements of first half, if there are any */
  while (i <= mid) {
    temp[k] = arr[i];
    i++;
    k++;
  }
```

3

```
/* Copy the remaining elements of 2nd half, if there are any */
  while (j <= high) {
    temp[k] = arr[j];
    j++;
    k++;
  }
```

4

```
/* Copy the temp array to original array */
  for (int k = low; k <= high; k++) {
    arr[k] = temp[k];
  }
}
```

5

MERGE SORT EXAMPLE