Time and space Complexity:

- Searching → Data retrieval
  1. Linear Search.
  2. Binary Search.
- Sort → Data Arrangement
  1. Bubble Sort
  2. selection Sort
  3. Insertion Sort
  4. Merge Sort
  5. quick Sort
  6. Count sort
  7. Radix Sort
  8. Date Sort

Theory.

→ Complexity (Relative) (Reoccurence)
Analysis from math.

→ notations (overview)

① Linear Search:

Find → loop from i=0; to i < arr.length (n elements)

```
if( arr[i] == data){        // Input
    return true;
}
} return false;
```

Time complexity, Traversal 0 to (n-1) — Total tr. 'n'

complexity - O(n)

Space complexity - Input is not include in space complexity.

→ O(1)

no. of data store in database = 10^9

no. of retrieval in a single day = 10^9 times (searching data from Database)

Linear Search

no. of data = $10^9 \equiv 2^{32} = n$

no. of retrieval = 10^9

cost of checking for single data = $O(n) = 10^9$

cost of checking 10^9 data = $10^9 \times 10^9$

$= 10^{18}$

Total Cost = 10^{18}

Linear Search is very costly as compare from Binary Search.

$10^{10} \times 10^{10} = 10^{20}$ ($a^i \times a^i = a^2$)

$10^{10} + 10^{10} = 2 \times 10^{10}$ ($a + a = 2a$)

Binary Search.

Data Size = $10^9 = n \equiv 2^{32}$ (nearly)

Sort ⟶ cost = $n \log n$ } best sorting

$= 10^9 \times \log_2(10^9)$

$= 10^9 \times \log_2(2^{32})$ [$\because \log_a a^b \equiv b$]

$= 10^9 \times 32 \equiv n$

Sort for single time $\equiv 10^{10}$

No. of checking = 10^9 times,

cost for single checking = $\log(n)$

$= \log(10^9) \equiv \log_2 2^{32}$

$= 32$

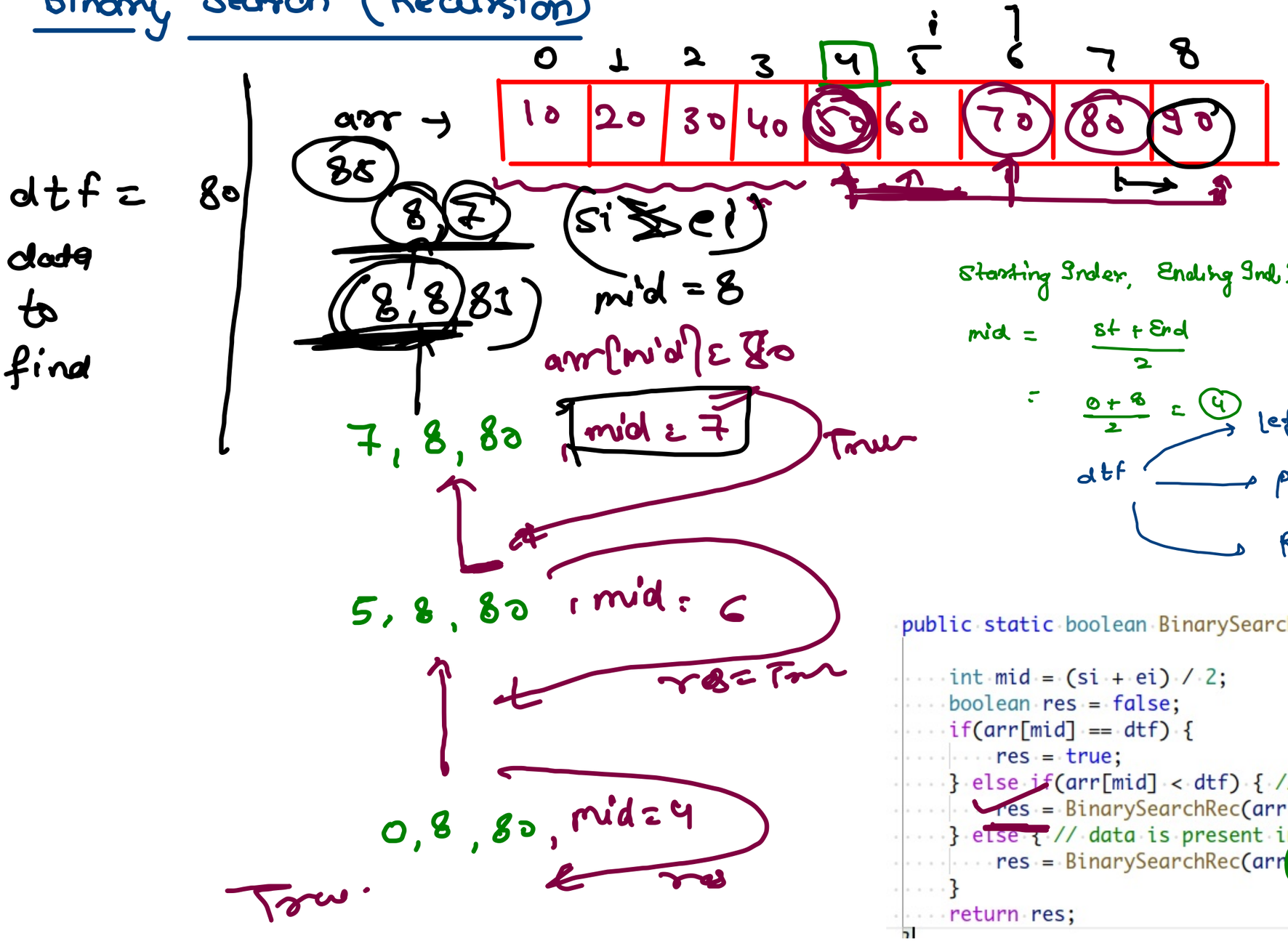
T. cost = 10^{10} (sorting cost) + $10^9 \times 32 \equiv 10$

$= 10^{10} + 10^9 \times 10$

$\equiv 2 \cdot 10^{10}$ (Total cost for retrieval)

$\equiv 10^{10}$ at max

Binary Search (Recursion)



Starting Index, Ending Index

array is sorted.

$mid = \frac{st + End}{2}$

$= \frac{0+8}{2} = 4$

dtf → present at mid

dtf ≶ mid → Left from mid → Right discarded

dtf ≷ mid → Right from mid → Left discard

arr[mid] > dtf

arr[mid] == dtf

arr[mid] < dtf

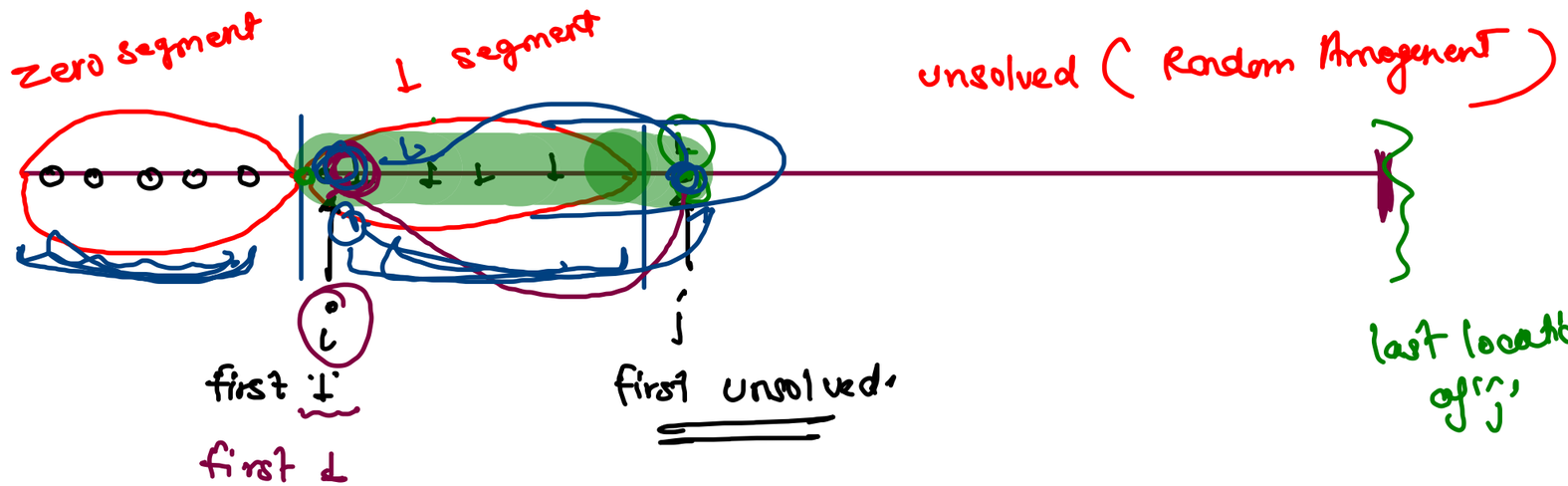
```java
public static boolean BinarySearchRec(int[] arr, int si, int ei, int dtf) {

    int mid = (si + ei) / 2;
    boolean res = false;
    if(arr[mid] == dtf) {
        res = true;
    } else if(arr[mid] < dtf) { // data is present in the right section
        res = BinarySearchRec(arr, mid + 1, ei, dtf);
    } else { // data is present in left
        res = BinarySearchRec(arr, si, mid - 1, dtf);
    }
    return res;
}
```

① Sort 0 1 2 → arr.length

arr → {1,0,0,1,0,1,0,1,1,0,0,1,0,0}

nlogn → Sort } brute force. O(n)

O(n) → complexity, single iteration.



arr[j] == 0

swap(arr, i, j)

// after swapping, arr[j] = 1

i++; (arr[i] = 0 segment of zero increased)

j++; (segment of 1 increased)

size of one's segment is same.

arr[j] == 1

j++; [segment of one is increased]

```
arr = {0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1};
```

arr[j] == 1

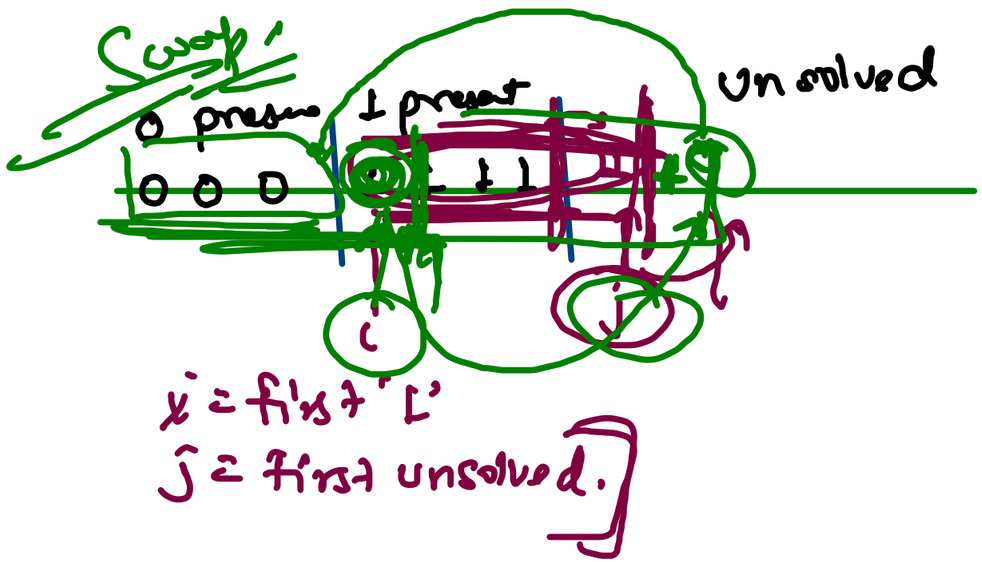
j ++;

arr[j] == 0

swap(arr, i, j);

j ++;

i ++;



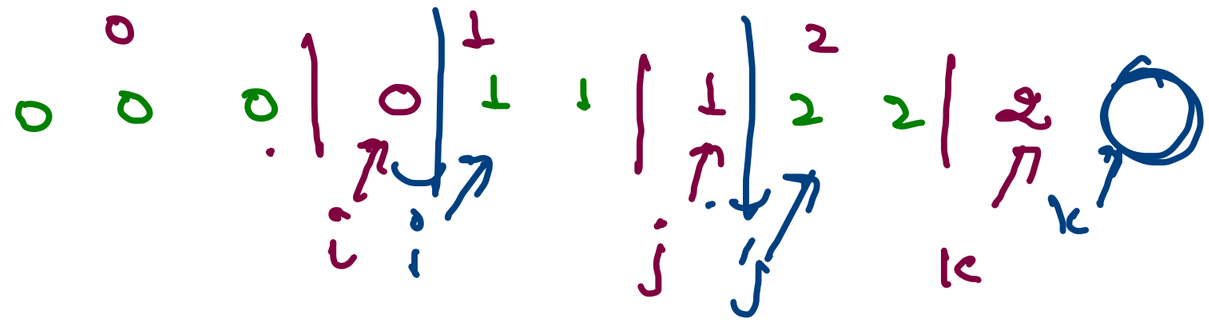
i = first '1'

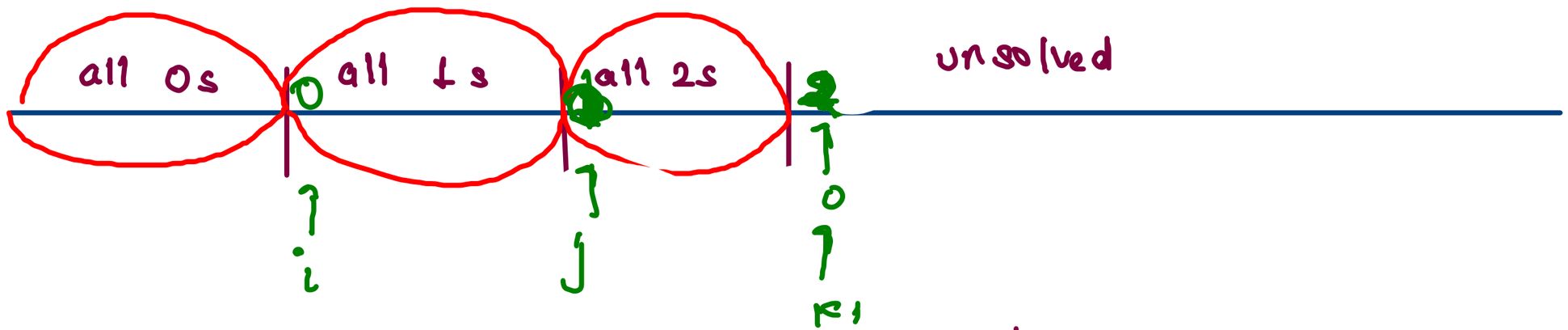
j = first unsolved.

arr [] ⇆ 0 0 1 2 1 1 0 1 2 2 0 2 2 2 1 2 0

Sort 0 1 2 →

O(n) → single iteration



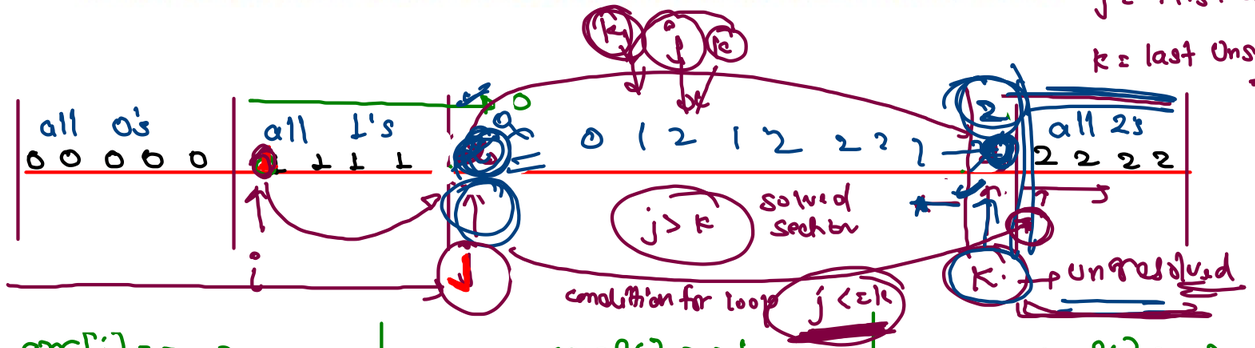


arr[k] == 0	arr[k] == 1	arr[k] == 2

swap(arr, k, j);
swap(arr, i, j)
i++;
j++;
k++;
	swap(arr, k, j);
	j++;
	k++;
		k++;

0 0 0 2 2 1 1 0 1 2 2 0 1 1 2 2 1 0 0 1 0

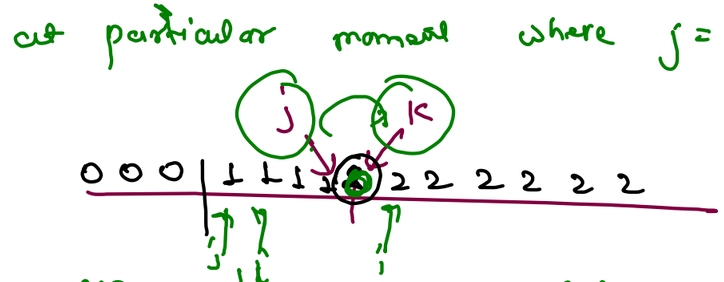
i = first 1
j = first Unsolved
k = last Unsolved

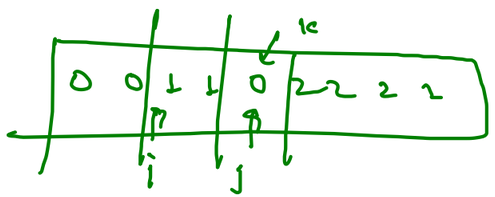


arr[j] == 0	arr[j] == 1	arr[j] == 2

swap(arr, j, i)
i++;
j++;
	Increase segment of 1.
	j++;
		swap(arr, j, k);
		k--; // Increase segment of 2
		j → No Requirement to change in j because 'j' is unsolved

at particular moment where j = k





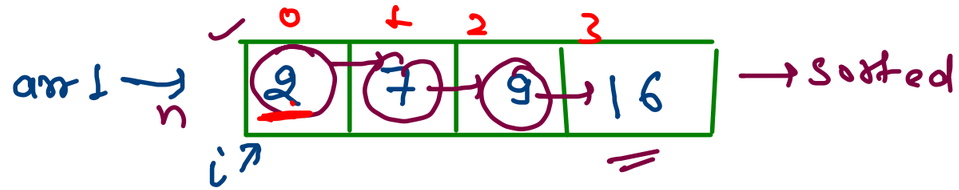
arr[j] = 0
swap(i, j);
i++;
j++;
Break
j > k

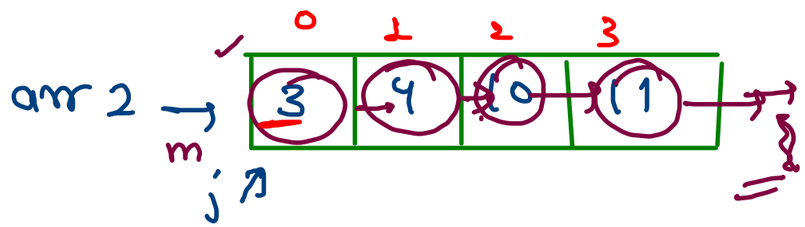
arr[j] = 1
j++;
Break
j <= k
j > k

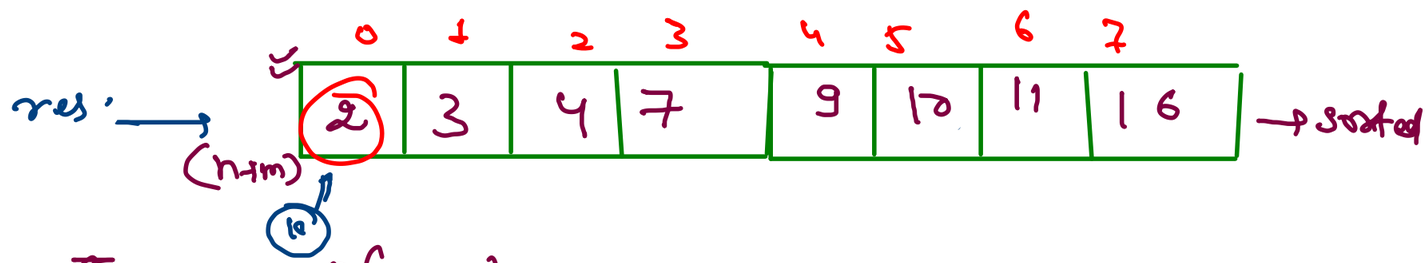
arr[j] == 2
swap(j, k);
k--;
Break
j > k

Merge two sorted array:

```
if(arr1[i] < arr2[j]) {
res[k] = arr1[i]
i++;
} else {
res[k] = arr2[j]
j++;
}
k++;
```







complexity → Time → O(n+m)