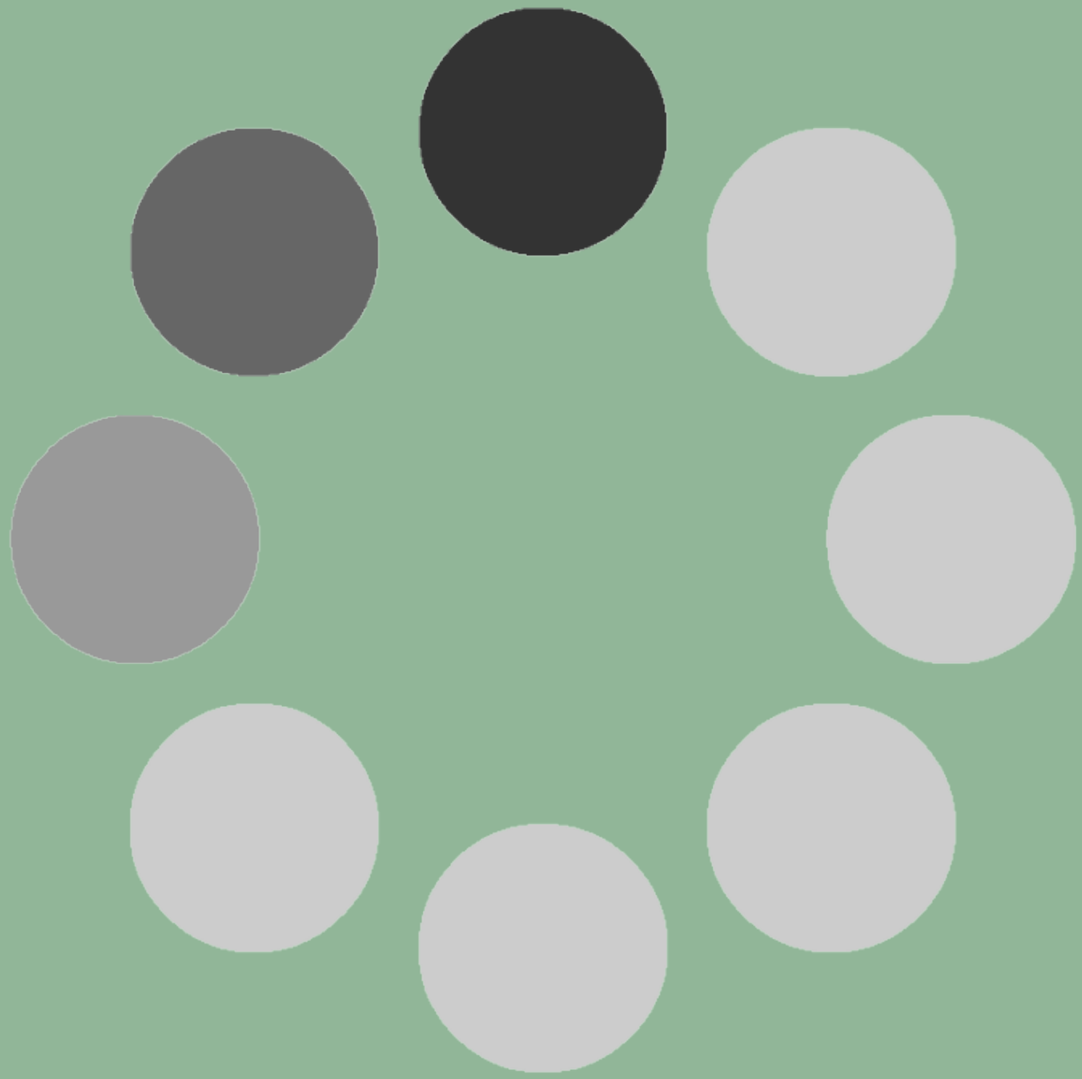


Algorithms &
Complexities

# MergeSort

WITH THE OLDIES

# Mergesort


- efficient, general-purpose
- comparison-based sorting algorithm
- sorts by breaking it into halves, sort, and then merging together
- works in recursion
- divide and conquer

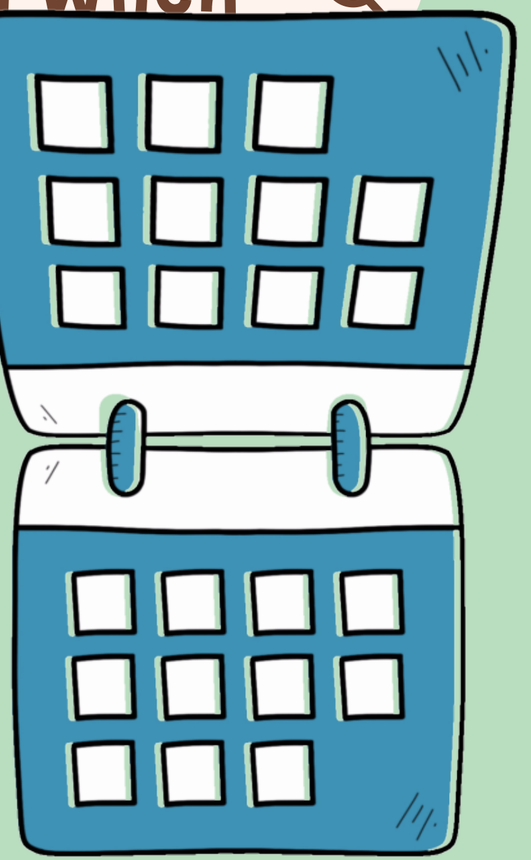
# Mergesort - Who and When


## John von Neumann

# Mergesort - Who and When


Created in the year

**1945**




# Mergesort - Who and When

- prepared programs for internal merge sorting (1945)
- to test adequacy of some instruction codes
- EDVAC computer

(The Art of Computer Programming by Knuth, 1968)

# Original Version of Mergesort

Based on

Mastering Algorithms in C

Kyle Loudon, 1999

# Original Version of Mergesort

```
static int merge(void *data, int esize, int i, int j, int k,
int (*compare)(const void *key1, const void *key2)) {
        char *a = data,*m;
        int ipos,jpos,mpos;
        ipos = i;
        jpos = j + 1;
        mpos = 0;

 if ((m = (char *)malloc(esize * ((k - i) + 1))) == NULL){
  return -1;
 }
```

Declare and initialize counters used for merging

Allocate storage for merged elements

# Original Version of Mergesort

```
while (ipos <= j || jpos <= k) {
  if (ipos > j) {
     while (jpos <= k) {
        memcpy(&m[mpos * esize], &a[jpos * esize], esize);
        jpos++;
        mpos++;
  }
  continue;
}
```

While either division has elements to merge

If the left division has no more elements to merge, copy the next ordered element in the right division

# Original Version of Mergesort

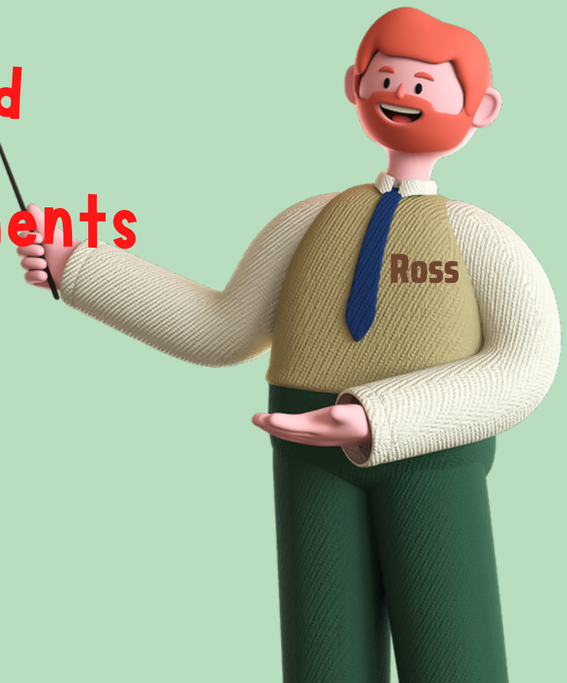
```
else if (jpos > k) {
    while (ipos <= j) {
    memcpy(&m[mpos * esize], &a[ipos * esize], esize);
     ipos++;
     mpos++;
    }
    continue;
  }
```

If the right division has no more elements to merge, copy the next ordered element in the left division

# Original Version of Mergesort

```
if (compare(&a[ipos * esize], &a[jpos * esize]) < 0) {
    memcpy(&m[mpos * esize], &a[ipos * esize], esize);
    ipos++;
  mpos++;
  }
  else {
    memcpy(&m[mpos * esize], &a[jpos * esize], esize);
    jpos++;
    mpos++;
  }
}
```

Append the next ordered element to the merged elements



# Original Version of Mergesort

```
        memcpy(&a[i * esize], m, esize * ((k - i) + 1));
```

Prepare to pass back merged data

```
        free(m);
        return 0;
}
```

Free storage allocated for merging

# Original Version of Mergesort

```
int mgsort(void *data, int size, int esize, int i, int k, int
(*compare)(const void *key1, const void *key2)) {
        int j;
        if (i < k) {

            j = (int)(((i + k - 1)) / 2);
```

Stop the recursion when no more divisions can be made

Determine where to divide the elements

# Original Version of Mergesort

```
if (mgsort(data, size, esize, i, j, compare) < 0){
    return -1;
}

if (mgsort(data, size, esize, j + 1, k, compare) < 0){
    return -1;
}
```


Recursively sort the two divisions

# Original Version of Mergesort

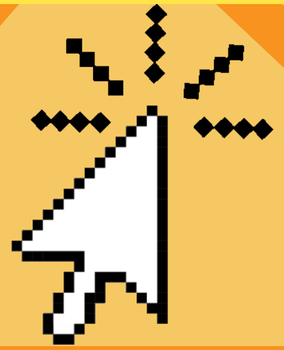
Recursively sort the two divisions

```
    if (mgsort(data, size, esize, i, j, compare) < 0){
        return -1;
    }
    if (mgsort(data, size, esize, j + 1, k, compare) < 0){
        return -1;
    }

    if (merge(data, esize, i, j, k, compare) < 0){
        return -1;
    }
}
return 0;
}
```

Merge the two sorted divisions into a single sorted set

# Mergesort Variations on the Net

from: Lance

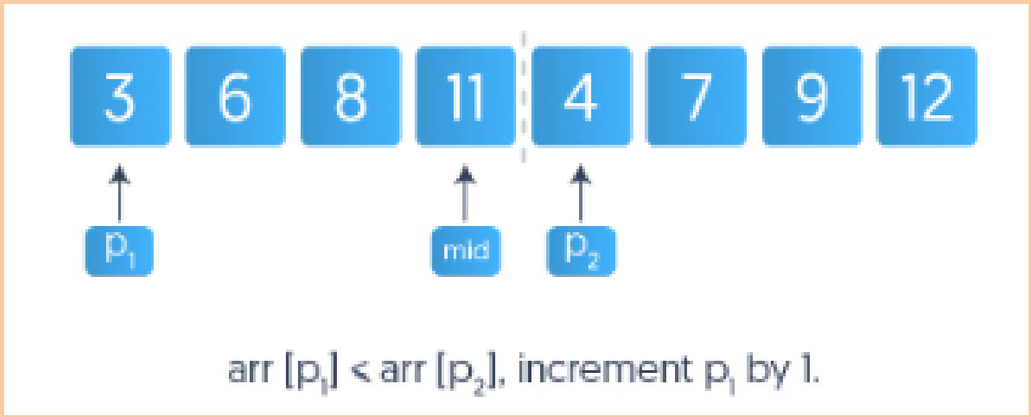


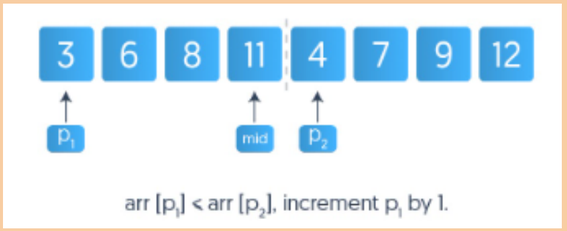

## Abstract In-place

1

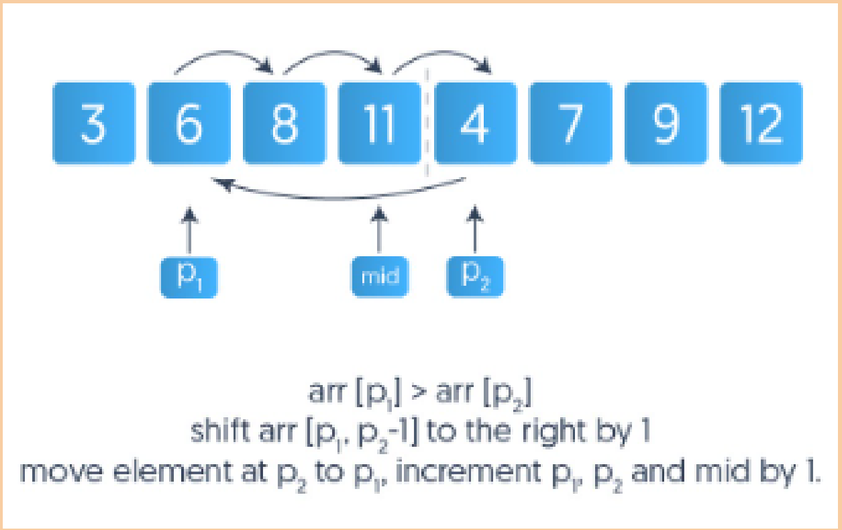
Sorting the first half of the array in place, then sorting the second half of the array in place, then do the merge of the two halves by moving the items around within the array.


Lance

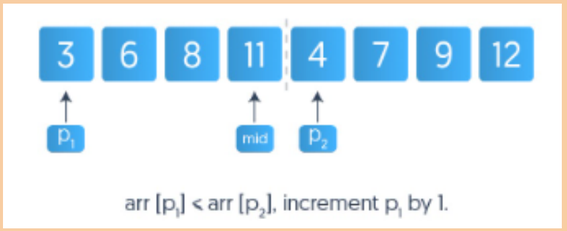

3
6
8
11
4
7
9
12
p1
mid
p2
arr [p1] < arr [p2], increment p1 by 1.

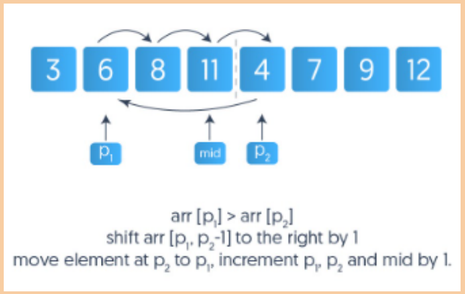


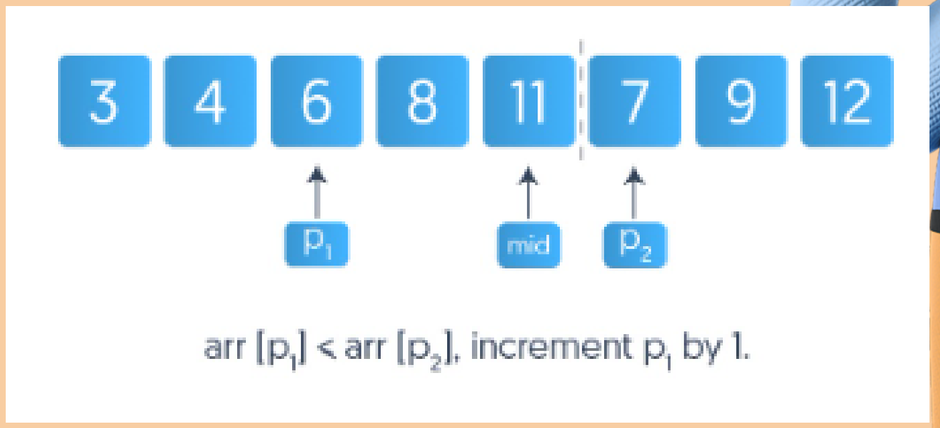

arr [$p_1$] < arr [$p_2$], increment $p_1$ by 1.



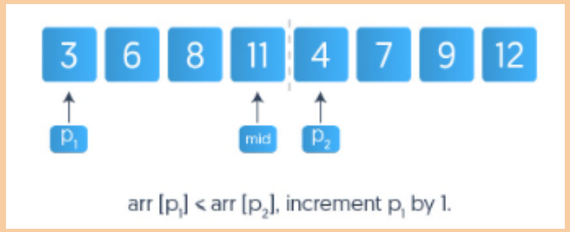

arr [$p_1$] > arr [$p_2$]
shift arr [$p_1$, $p_2$-1] to the right by 1
move element at $p_2$ to $p_1$, increment $p_1$, $p_2$ and mid by 1.

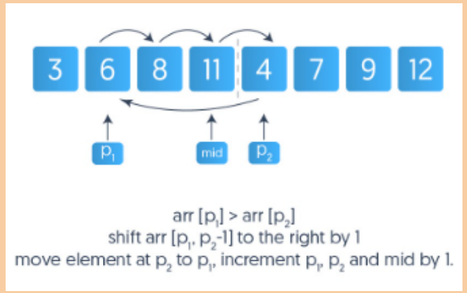

3 6 8 11 4 7 9 12
p1
mid
p2
arr [p1] < arr [p2], increment p1 by 1.

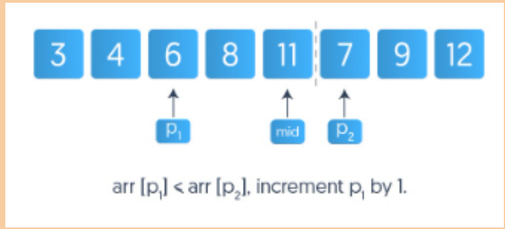

3 6 8 11 4 7 9 12
p1
mid
p2
arr [p1] > arr [p2]
shift arr [p1, p2-1] to the right by 1
move element at p2 to p1, increment p1, p2 and mid by 1.


3 4 6 8 11 7 9 12
p1
mid
p2
arr [p1] < arr [p2], increment p1 by 1.

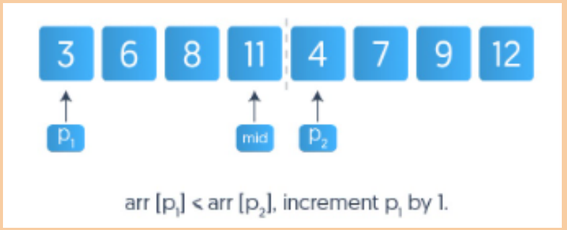

Lance

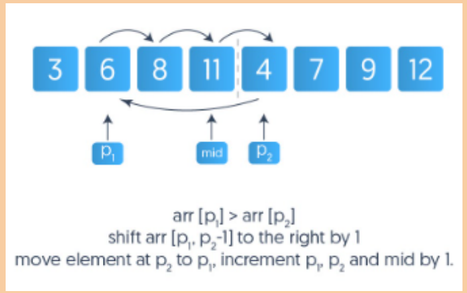


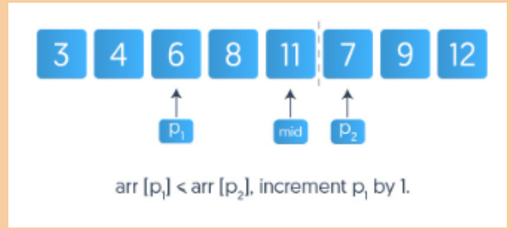



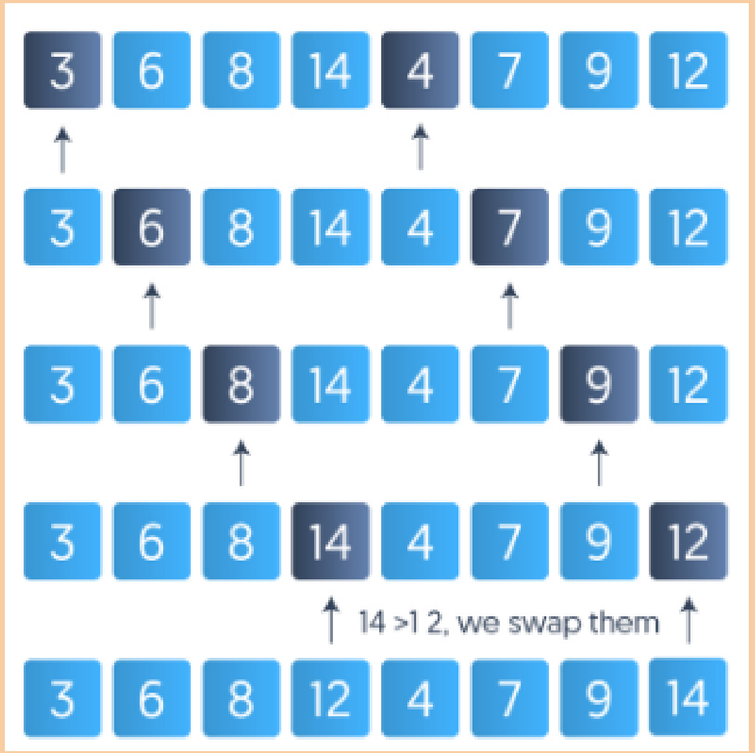

Time Complexity: O(n^2 )

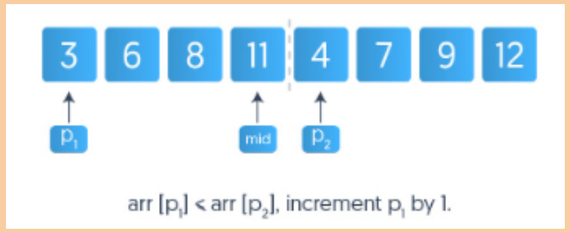

3 6 8 11 4 7 9 12
p1
mid
p2
arr [p1] < arr [p2], increment p1 by 1.

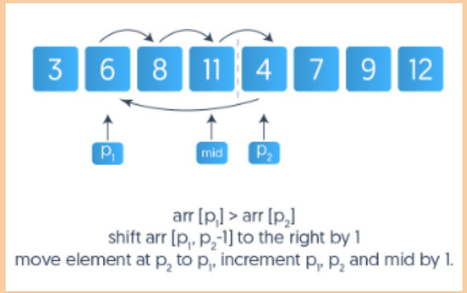

3 6 8 11 4 7 9 12
p1
mid
p2
arr [p1] > arr [p2]
shift arr [p1, p2-1] to the right by 1
move element at p2 to p1, increment p1, p2 and mid by 1.

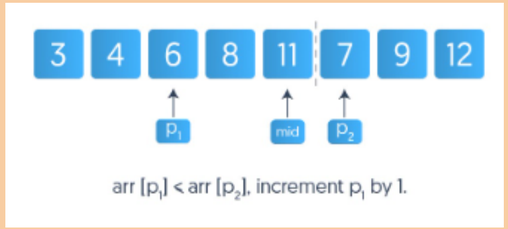

3 4 6 8 11 7 9 12
p1
mid
p2
arr [p1] < arr [p2], increment p1 by 1.


Lance

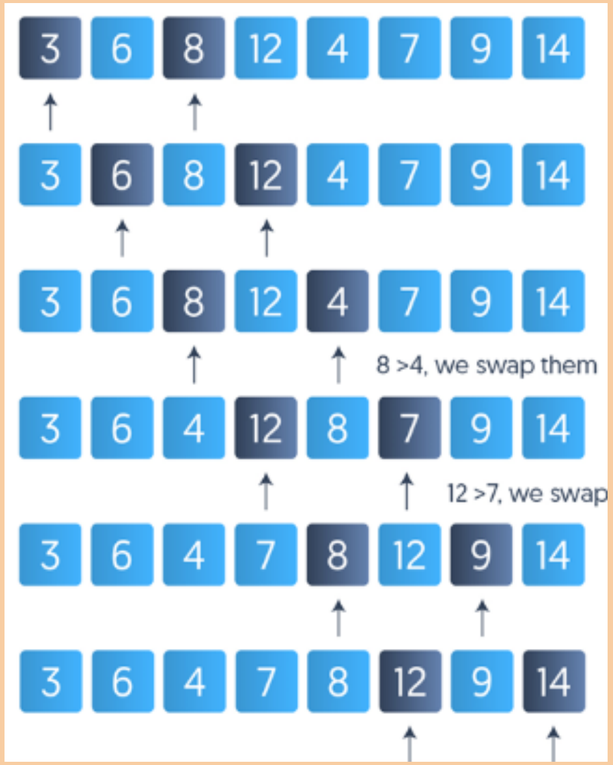

3 6 8 14 4 7 9 12
3 6 8 14 4 7 9 12
3 6 8 14 4 7 9 12
3 6 8 14 4 7 9 12
14 >1 2, we swap them
3 6 8 12 4 7 9 14


3 6 8 11 4 7 9 12
p1
mid
p2
arr [p1] < arr [p2], increment p1 by 1.

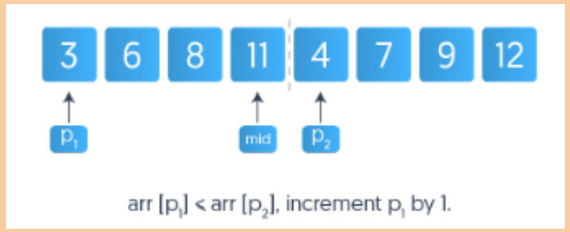

3 6 8 11 4 7 9 12
p1
mid
p2
arr [p1] > arr [p2]
shift arr [p1, p2-1] to the right by 1
move element at p2 to p1, increment p1, p2 and mid by 1.

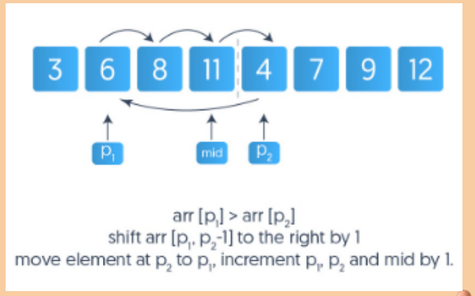

3 4 6 8 11 7 9 12
p1
mid
p2
arr [p1] < arr [p2], increment p1 by 1.

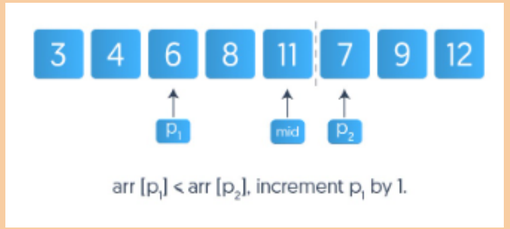

3 6 8 14 4 7 9 12
3 6 8 14 4 7 9 12
3 6 8 14 4 7 9 12
3 6 8 14 4 7 9 12
14 >1 2, we swap them
3 6 8 12 4 7 9 14


3 6 8 12 4 7 9 14
3 6 8 12 4 7 9 14
3 6 8 12 4 7 9 14
8 >4, we swap them
3 6 4 12 8 7 9 14
12 >7, we swap
3 6 4 7 8 12 9 14
3 6 4 7 8 12 9 14

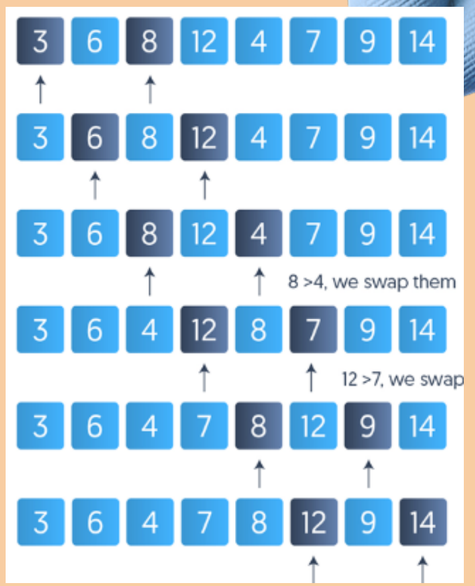

Lance


3 6 8 11 4 7 9 12
p1
mid
p2
arr [p1] < arr [p2], increment p1 by 1.


3 6 8 11 4 7 9 12
p1
mid
p2
arr [p1] > arr [p2]
shift arr [p1, p2-1] to the right by 1
move element at p2 to p1, increment p1, p2 and mid by 1.

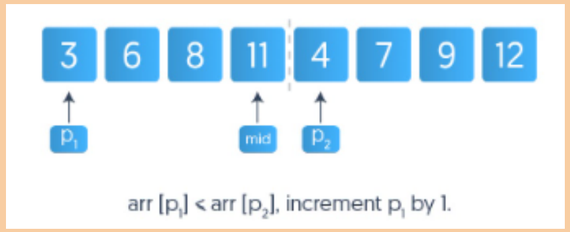

3 4 6 8 11 7 9 12
p1
mid
p2
arr [p1] < arr [p2], increment p1 by 1.

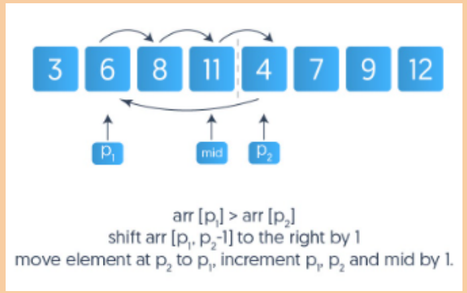

3 6 8 14 4 7 9 12
3 6 8 14 4 7 9 12
3 6 8 14 4 7 9 12
3 6 8 14 4 7 9 12
14 >1 2, we swap them
3 6 8 12 4 7 9 14

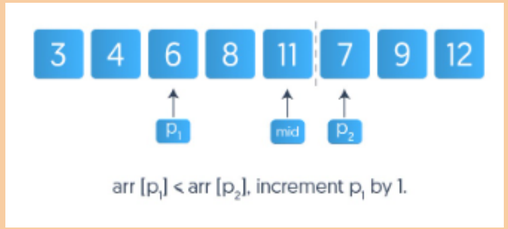

3 6 8 12 4 7 9 14
3 6 8 12 4 7 9 14
3 6 8 12 4 7 9 14
8 >4, we swap them
3 6 4 12 8 7 9 14
12 >7, we swap
3 6 4 7 8 12 9 14
3 6 4 7 8 12 9 14


Lance


3 6 4 7 8 12 9 14
3 6 4 7 8 12 9 14
6 >4, we swap them
3 4 6 7 8 12 9 14
3 4 6 7 8 12 9 14
3 4 6 7 8 12 9 14
3 4 6 7 8 12 9 14
12 >9, we swap them
3 4 6 7 8 9 12 14
h=1, we stop the algorithm.
3 4 6 7 8 9 12 14



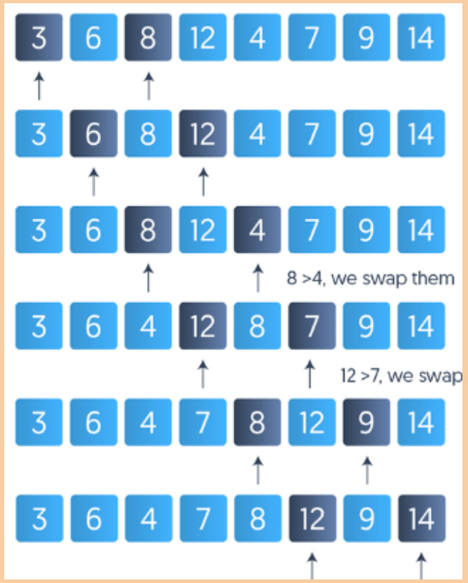





Time Complexity: O(n (log2n)2 )





Bottom-up +

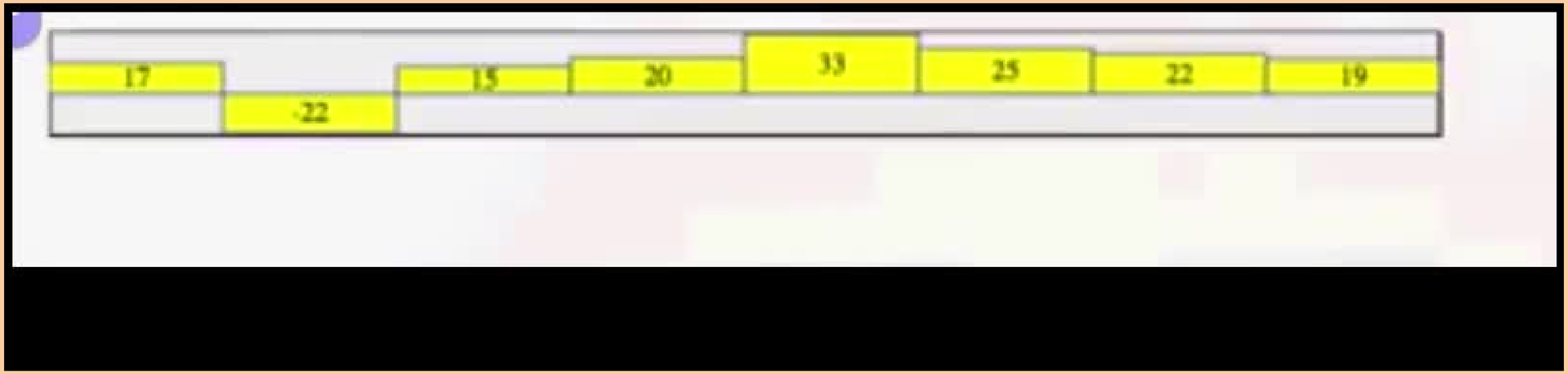
# 2

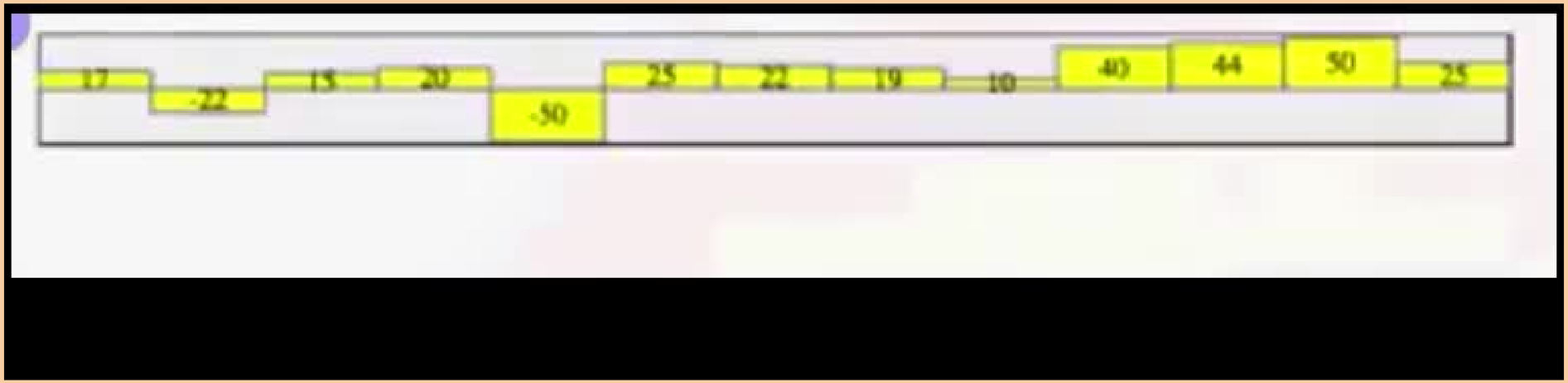
This variation of Merge sort uses iteration(loops) to sort the given list

Time Complexity : O(n * log n)

Space Complexity: O(n)


Lance


17
-22
15
20
33
25
22
19


17
-22
15
20
-50
25
22
19
10
40
44
50
25


Top-down +

# 3

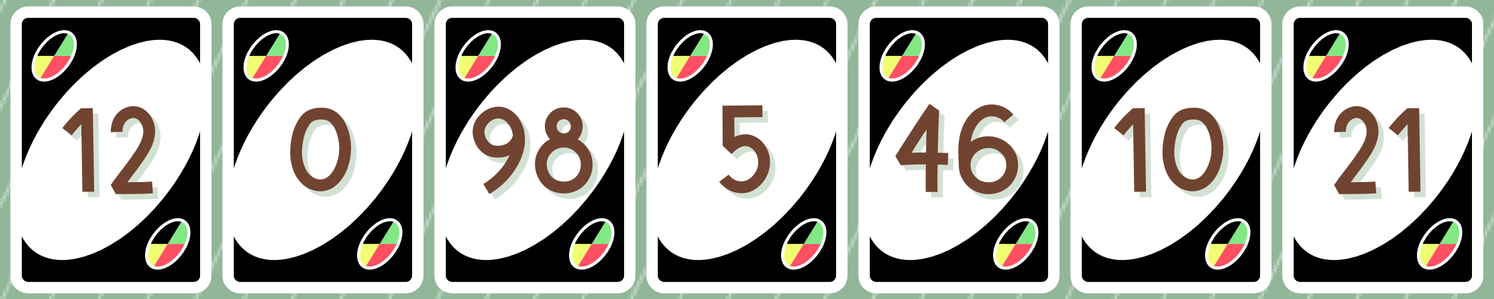
This variation is the standard variation for Merge sort. This variation uses recursion to sort the list.

Time Complexity : O(n * log n)

Space Complexity: O(n)


Mergesort
Simulation
fr: Eque

# MergeSort Simulation





Steps

Let's note the steps as we go along!

# MergeSort Simulation


Find midPoint of array; split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves

## Steps

Find the mid-point to divide array in half , as evenly as possible

# MergeSort Simulation


Find midPoint of array; split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves

**Steps**

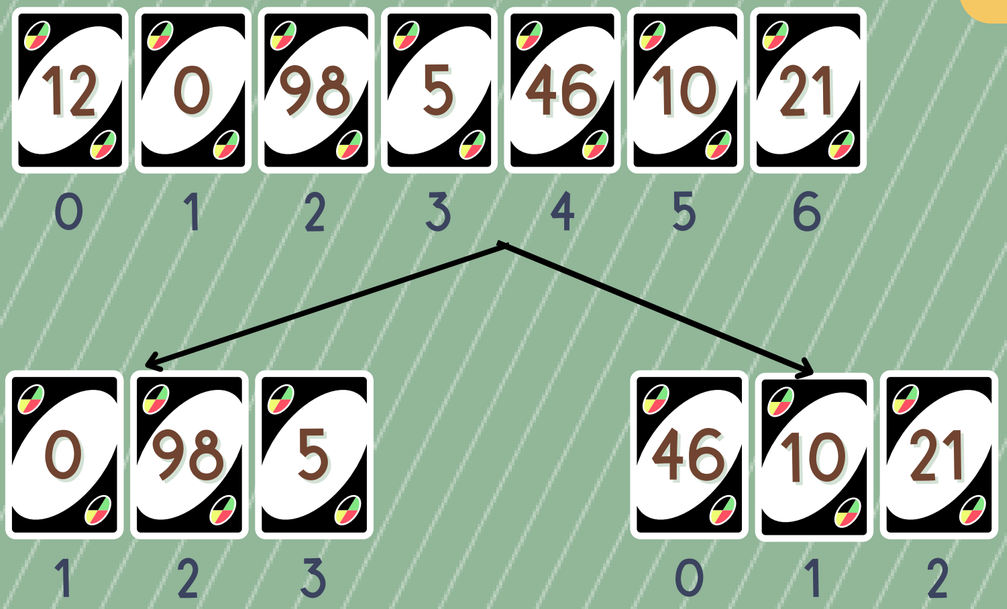
Find the mid-point to divide array in half , as evenly as possible

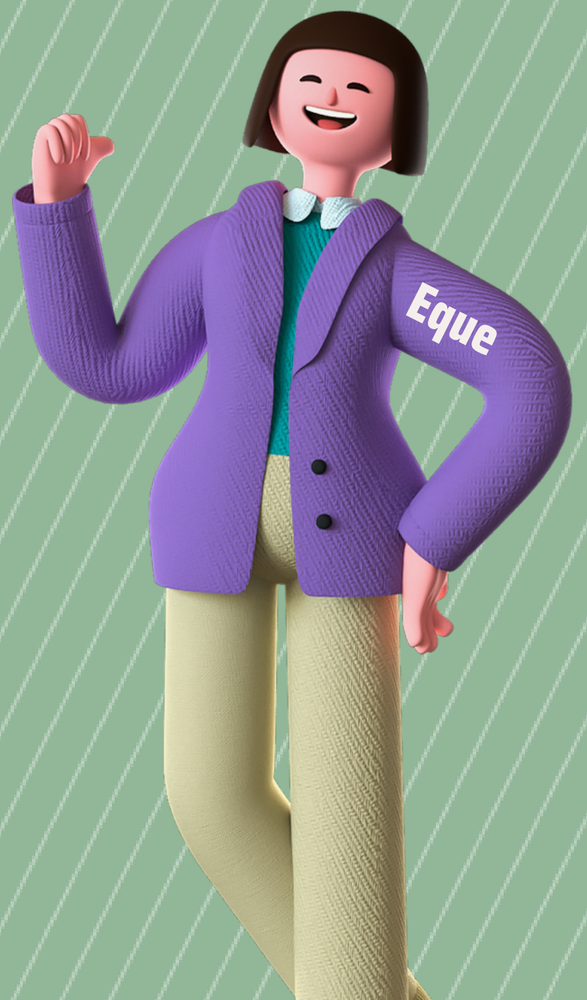
# MergeSort Simulation

Find midPoint of array;
split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves

Find the mid-point to divide array in half , as evenly as possible



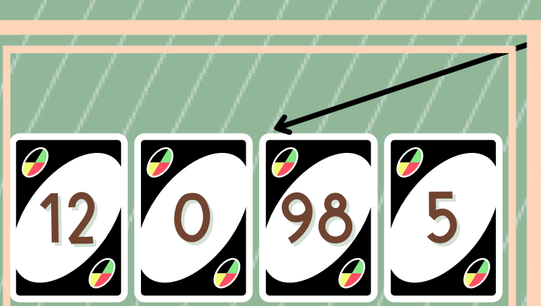



# MergeSort Simulation

Find midPoint of array;
split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves

Find the mid-point to divide array in half , as evenly as possible



## Steps

start with the left sub-array

# MergeSort Simulation

Find midPoint of array:
split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves

Find the mid-point to divide array in half , as evenly as possible

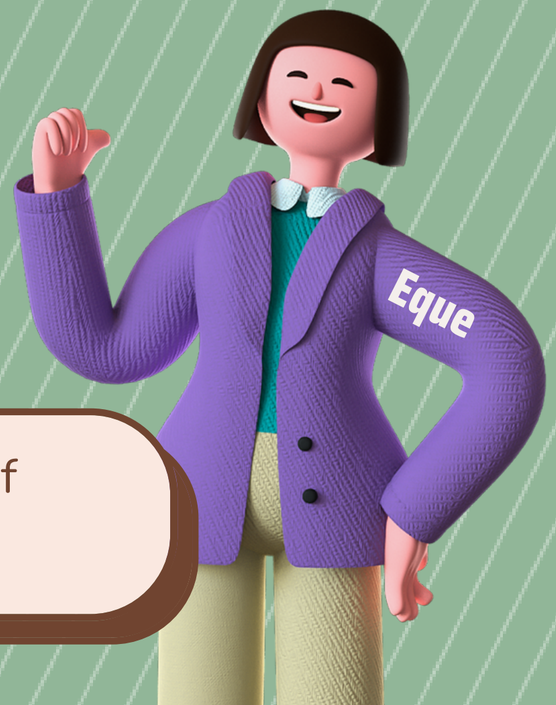

if arrayLength of Left SubArray is more than 1, repeat previous step



midpoint

## Steps

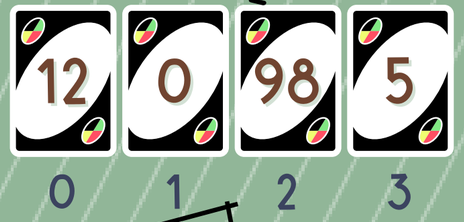
Keep splitting the left subarrays in half until there is ONE left in the array.



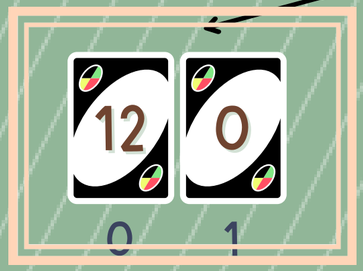
# MergeSort Simulation

Find midPoint of array;
split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves







Find the mid-point to divide array in half , as evenly as possible

if arrayLength of Left SubArray is more than 1, repeat previous step

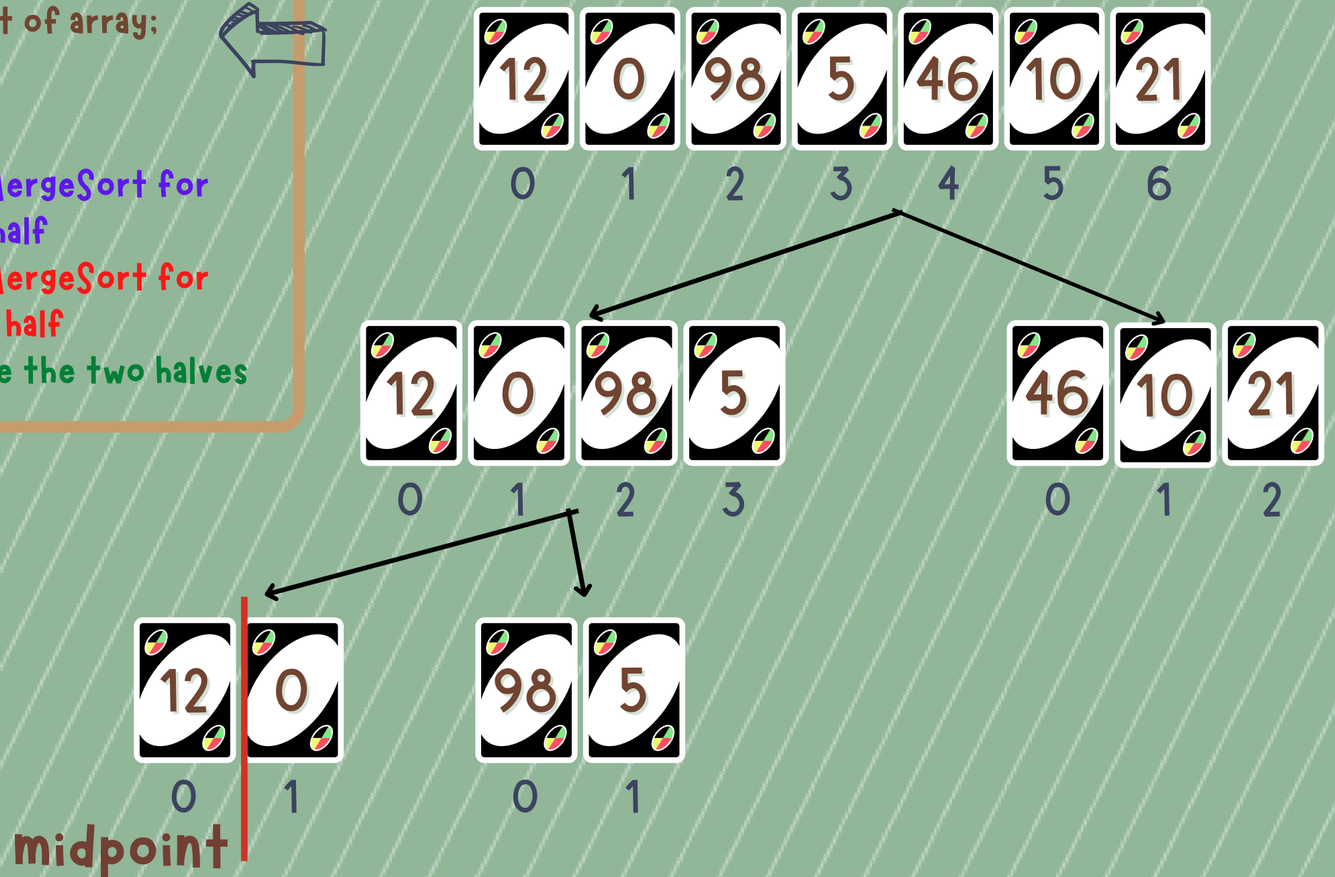

**Steps**

Keep splitting the left subarrays in half until there is ONE left in the array.

# MergeSort Simulation

Find midPoint of array; split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves




Find the mid-point to divide array in half , as evenly as possible

if arrayLength of Left SubArray is more than 1, repeat previous step

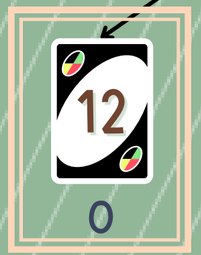
# MergeSort Simulation

Find midPoint of array;
split evenly

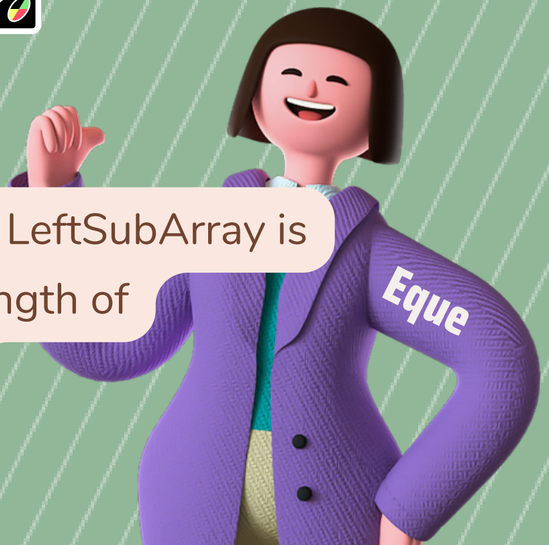
1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves






if arrayLength of LeftSubArray is 1, check arrayLength of RighSubArray



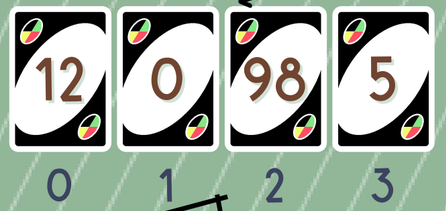

Find the mid-point to divide array in half , as evenly as possible

if arrayLength of Left SubArray is more than 1, repeat previous step

# MergeSort Simulation

Find midPoint of array;
split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves







If ArrayLength of RightSubArray is also 1, ready to merge. Else, repeat 2nd step - but for RightSubArray instead of Left


Find the mid-point to divide array in half , as evenly as possible

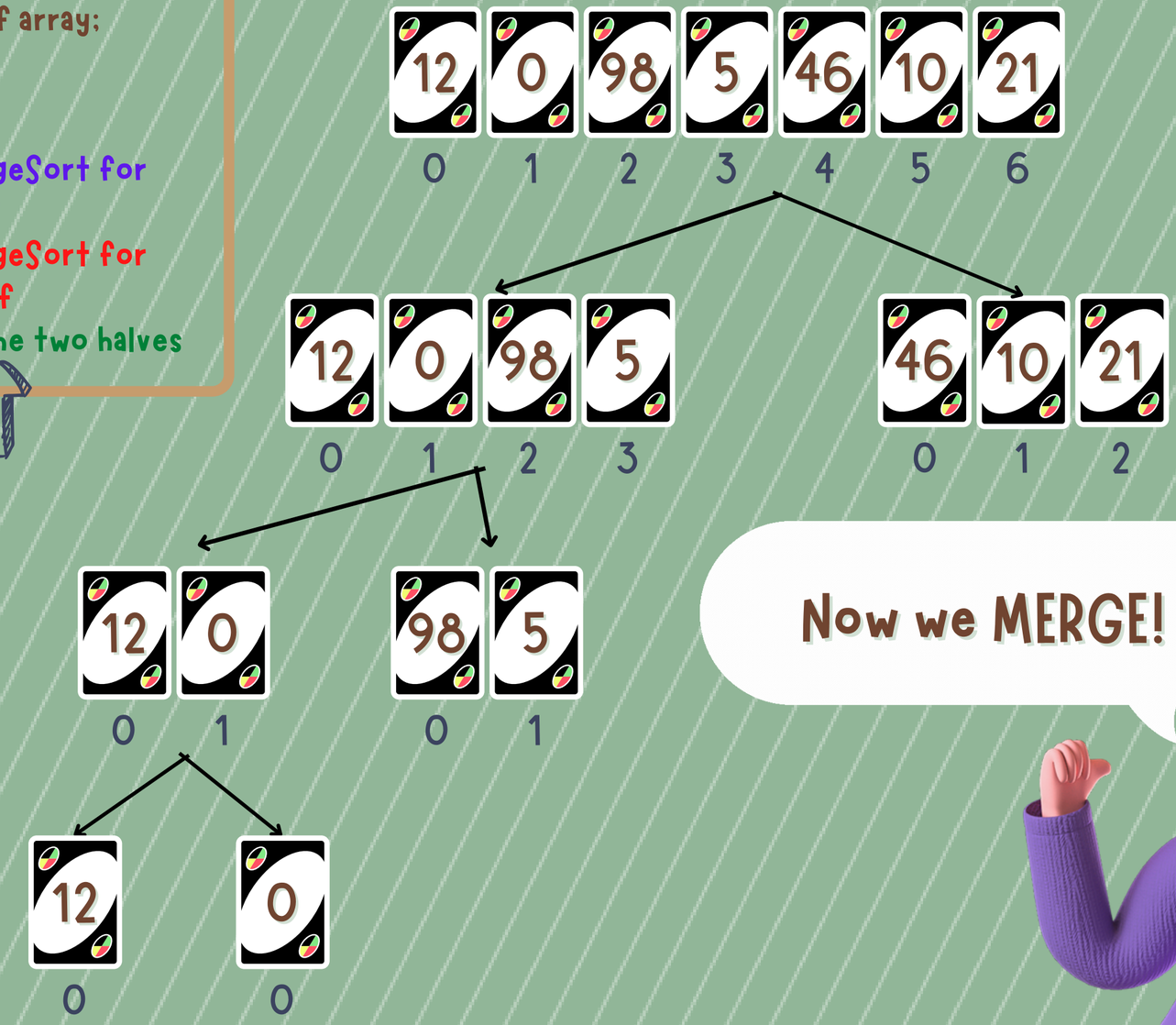
if arrayLength of Left SubArray is more than 1, repeat previous step

if arrayLength of LeftSubArray is 1, check arrayLength of RighSubArray

# MergeSort Simulation

Find midPoint of array;
split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves




Now we MERGE!


Find the mid-point to divide array in half , as evenly as possible

if arrayLength of Left SubArray is more than 1, repeat previous step

if arrayLength of LeftSubArray is 1, check arrayLength of RighSubArray

If ArrayLength of RightSubArray is also 1, ready to merge. Else, repeat 2nd step - but for RightSubArray instead of Left

# MergeSort Simulation

Find midPoint of array; split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves






Select both subArrays, and merge them together in Sorted order

Find the mid-point to divide array in half , as evenly as possible

if arrayLength of Left SubArray is more than 1, repeat previous step

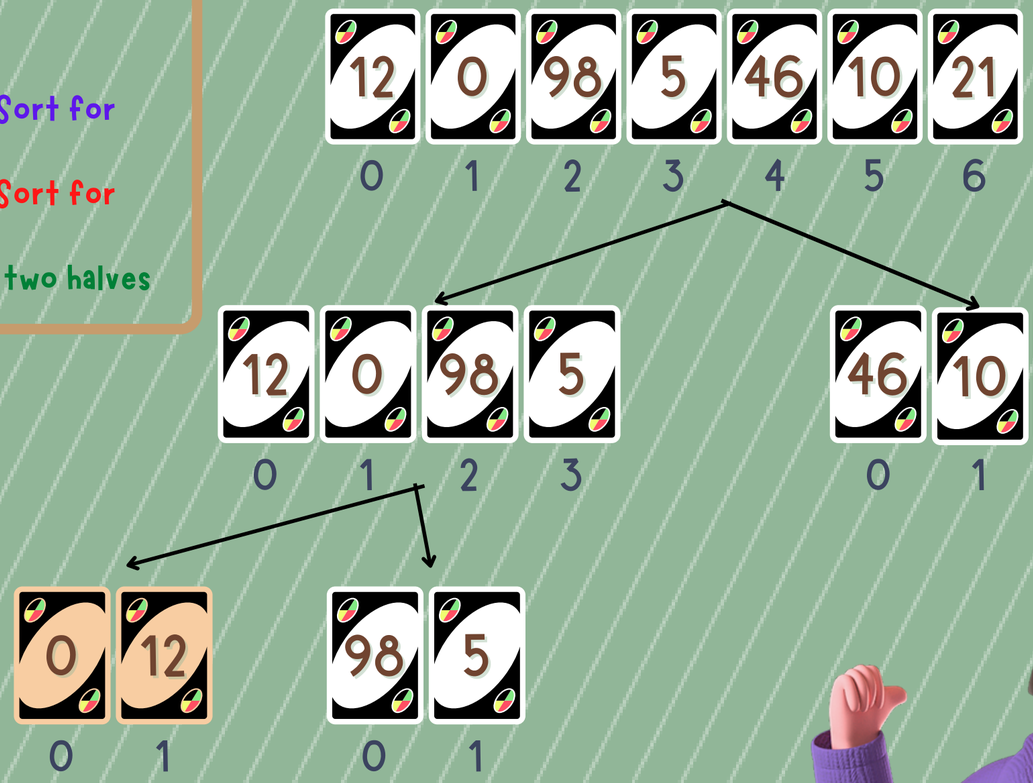
if arrayLength of LeftSubArray is 1, check arrayLength of RighSubArray

If ArrayLength of RightSubArray is also 1, ready to merge. Else, repeat 2nd step - but for RightSubArray instead of Left

# MergeSort Simulation

Find midPoint of array;
split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves





Find the mid-point to divide array in half , as evenly as possible

if arrayLength of Left SubArray is more than 1, repeat previous step

if arrayLength of LeftSubArray is 1, check arrayLength of RighSubArray

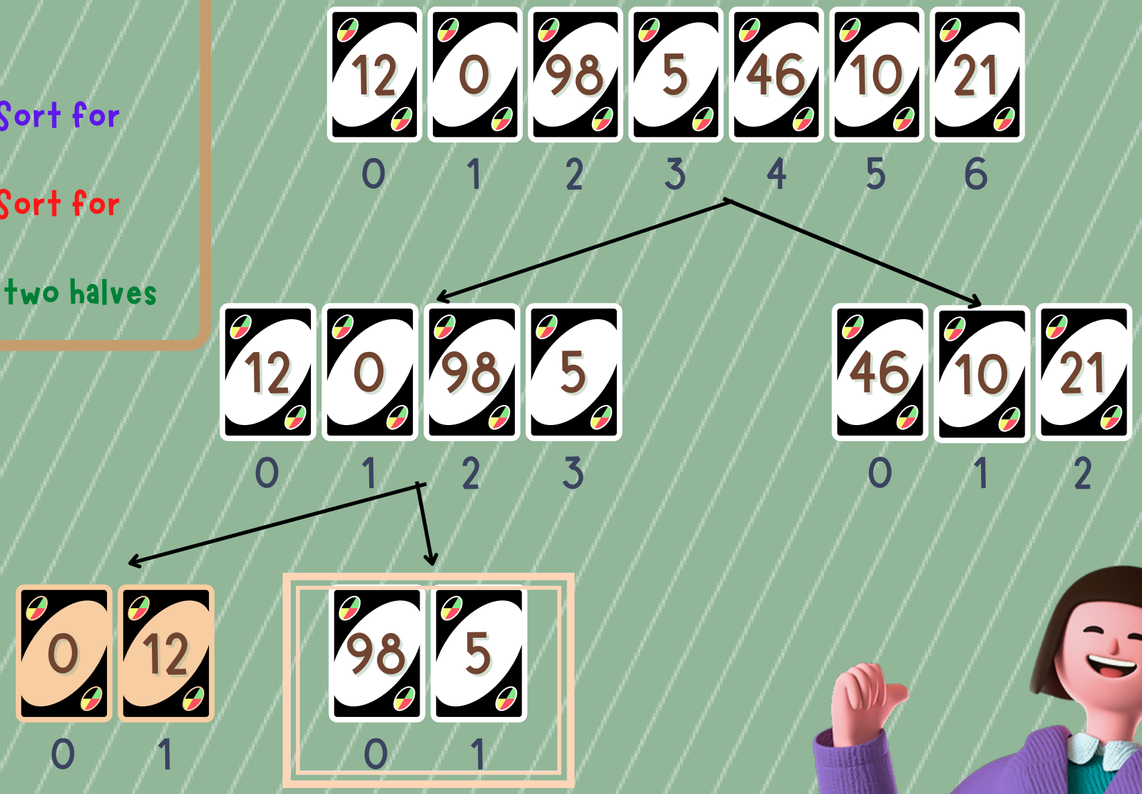
If ArrayLength of RightSubArray is also 1, ready to merge. Else, repeat 2nd step - but for RightSubArray instead of Left

Select both subArrays, and merge them together in Sorted order

# MergeSort Simulation

Find midPoint of array;
split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves





Find the mid-point to divide array in half , as evenly as possible

if arrayLength of Left SubArray is more than 1, repeat previous step

if arrayLength of LeftSubArray is 1, check arrayLength of RighSubArray

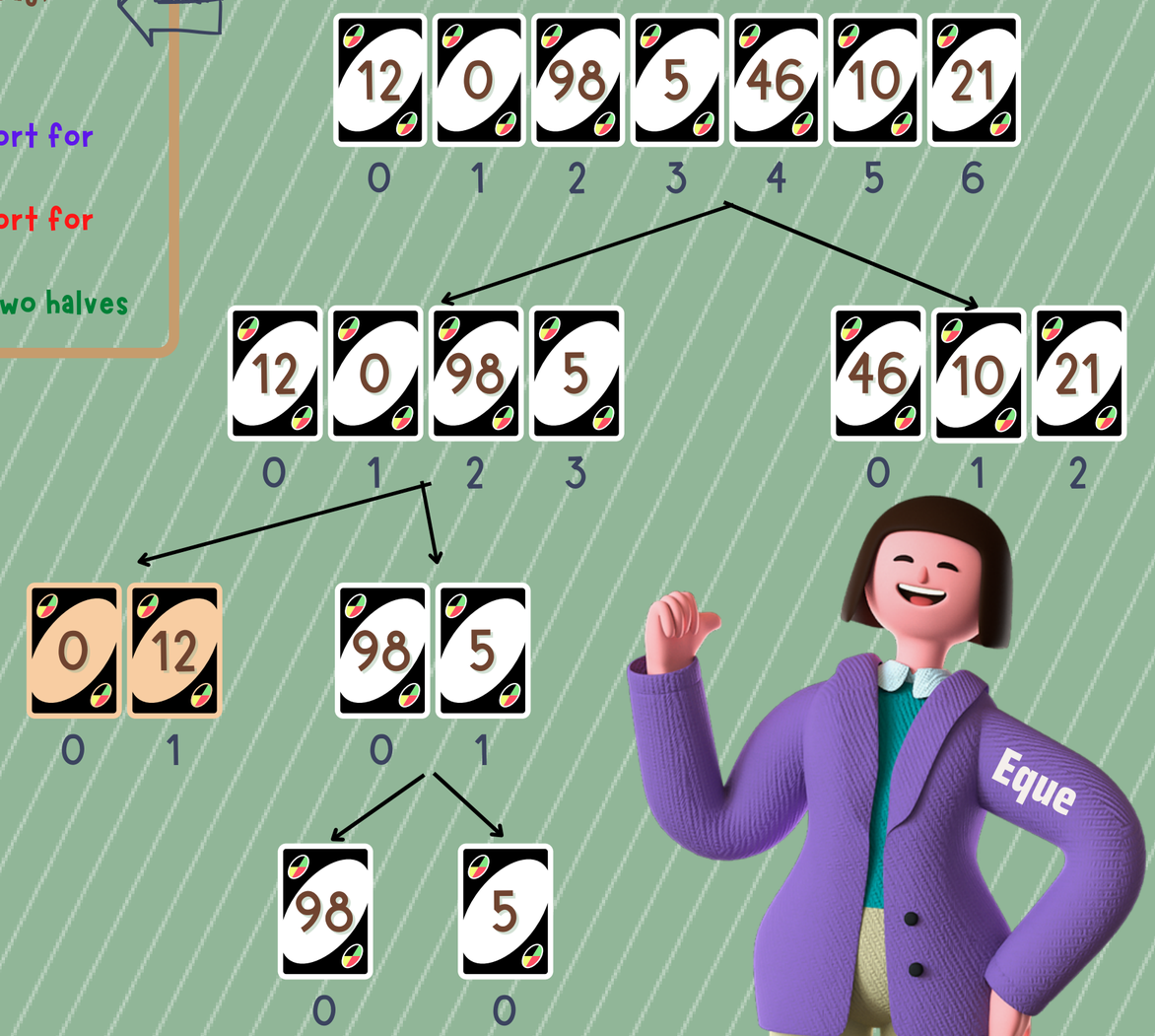
If ArrayLength of RightSubArray is also 1, ready to merge. Else, repeat 2nd step - but for RightSubArray instead of Left

Select both subArrays, and merge them together in Sorted order

# MergeSort Simulation

Find midPoint of array; split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves







Find the mid-point to divide array in half , as evenly as possible

if arrayLength of RIGHT SubArray is more than 1, repeat previous step

if arrayLength of LeftSubArray is 1, check arrayLength of RighSubArray

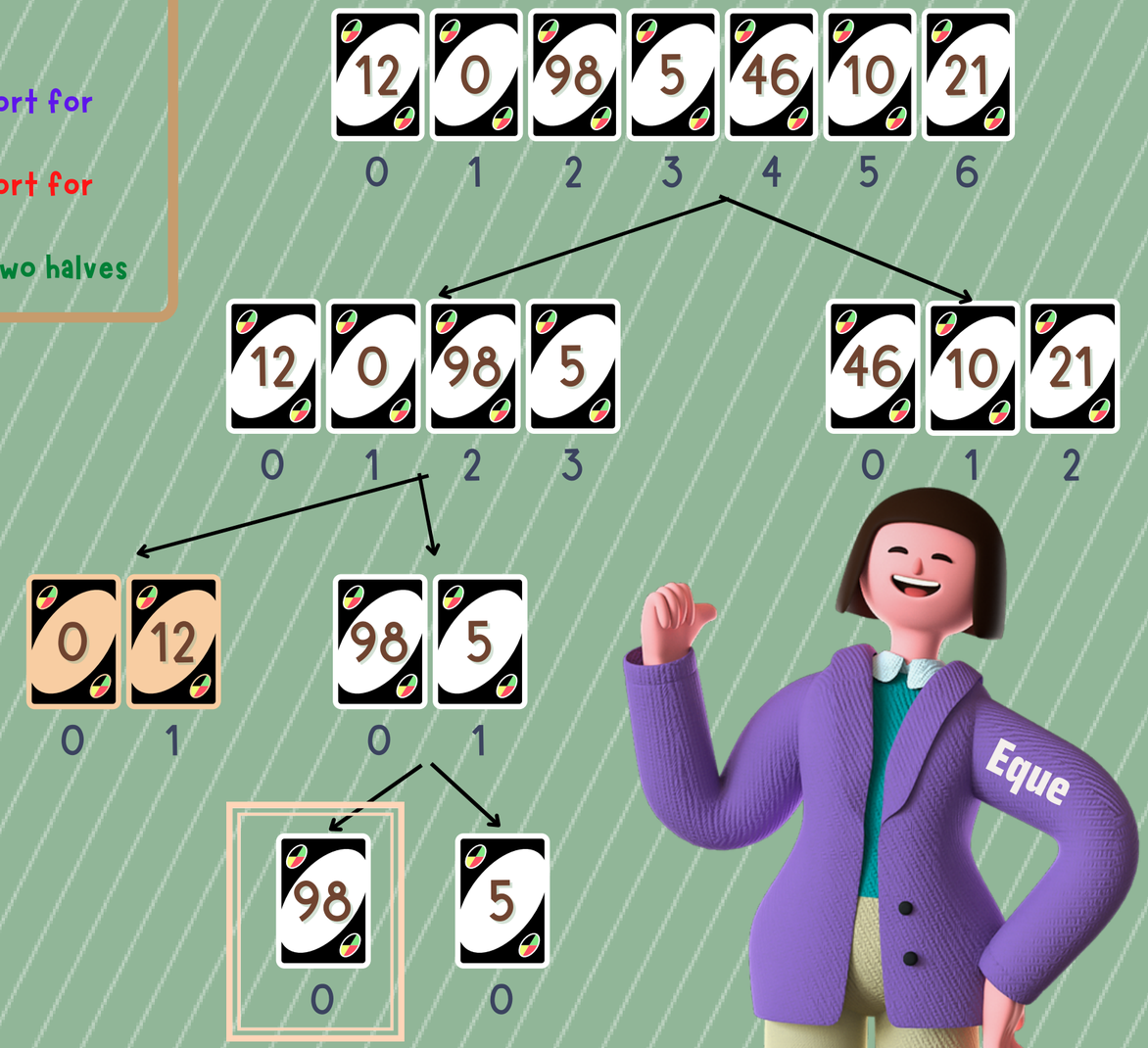
If ArrayLength of RightSubArray is also 1, ready to merge. Else, repeat 2nd step - but for RightSubArray instead of Left

Select both subArrays, and merge them together in Sorted order

# MergeSort Simulation

Find midPoint of array;
split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves







Find the mid-point to divide array in half , as evenly as possible

if arrayLength of RIGHT SubArray is more than 1, repeat previous step

if arrayLength of LeftSubArray is 1, check arrayLength of RighSubArray

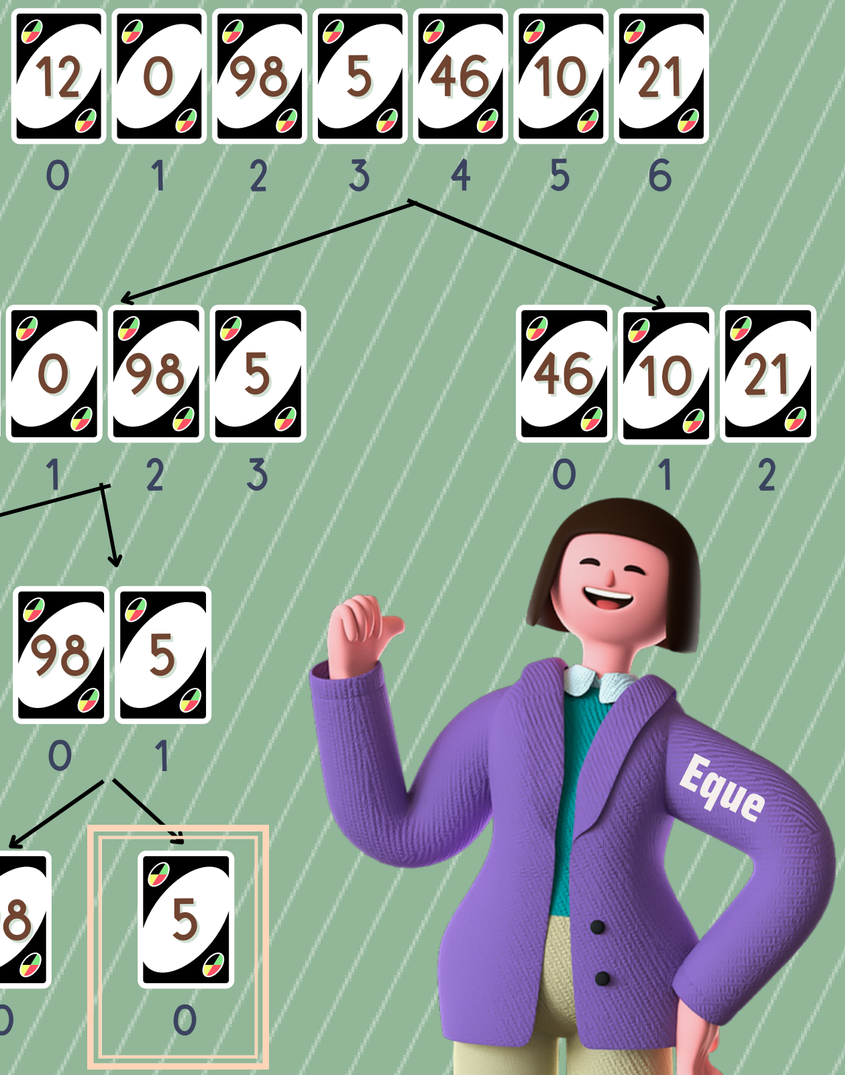
If ArrayLength of RightSubArray is also 1, ready to merge. Else, repeat 2nd step - but for RightSubArray instead of Left

Select both subArrays, and merge them together in Sorted order

# MergeSort Simulation

Find midPoint of array;
split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves







Find the mid-point to divide array in half , as evenly as possible

if arrayLength of RIGHT SubArray is more than 1, repeat previous step

if arrayLength of LeftSubArray is 1, check arrayLength of RighSubArray

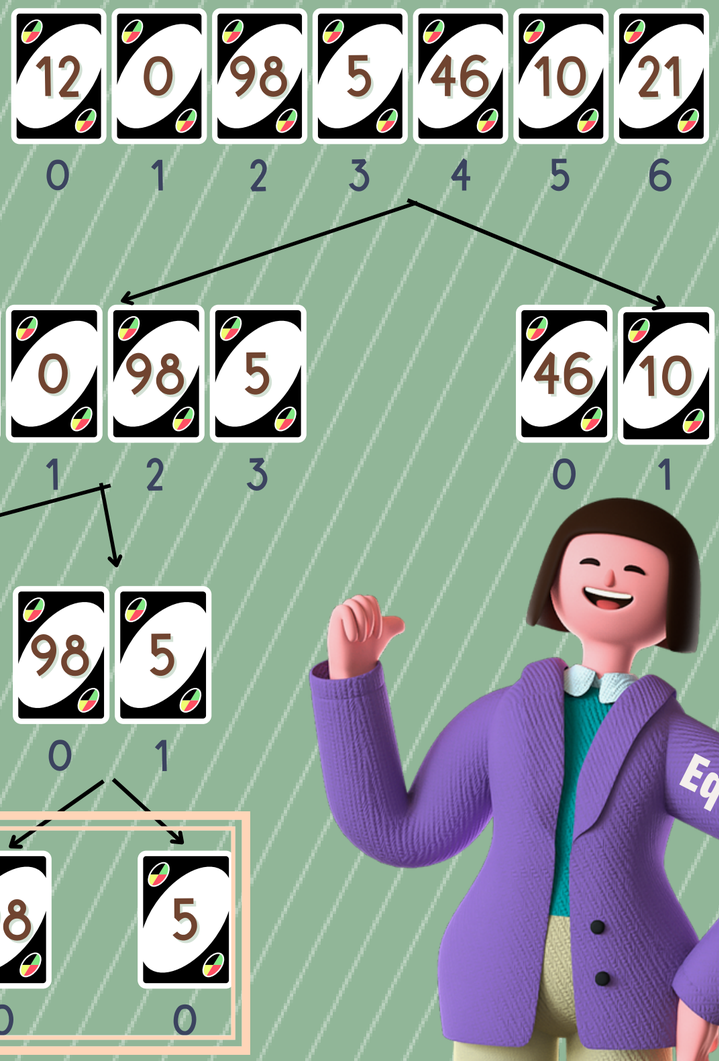
If ArrayLength of RightSubArray is also 1, ready to merge. Else, repeat 2nd step - but for RightSubArray instead of Left

Select both subArrays, and merge them together in Sorted order

# MergeSort Simulation

Find midPoint of array; split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves







Find the mid-point to divide array in half , as evenly as possible

if arrayLength of RIGHT SubArray is more than 1, repeat previous step

if arrayLength of LeftSubArray is 1, check arrayLength of RighSubArray

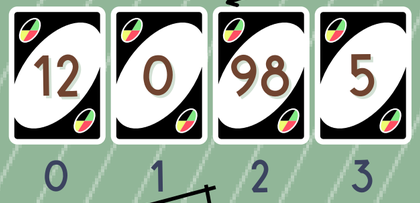
If ArrayLength of RightSubArray is also 1, ready to merge. Else, repeat 2nd step - but for RightSubArray instead of Left

Select both subArrays, and merge them together in Sorted order

# MergeSort Simulation

Find midPoint of array;
split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves






Find the mid-point to divide array in half , as evenly as possible

if arrayLength of RIGHT SubArray is more than 1, repeat previous step

if arrayLength of LeftSubArray is 1, check arrayLength of RighSubArray

If ArrayLength of RightSubArray is also 1, ready to merge. Else, repeat 2nd step - but for RightSubArray instead of Left

Select both subArrays, and merge them together in Sorted order

# MergeSort Simulation

Find midPoint of array;
split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves





Find the mid-point to divide array in half , as evenly as possible

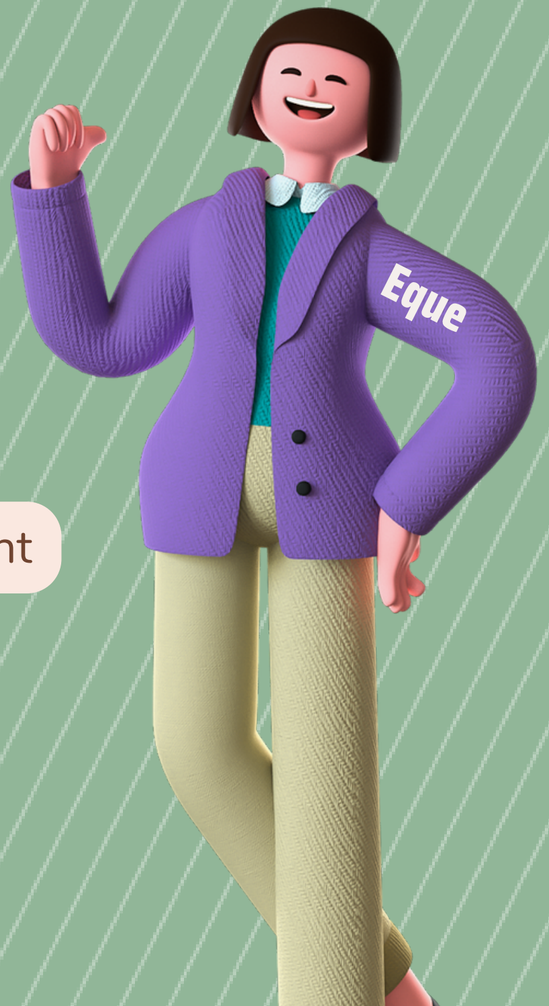
if arrayLength of RIGHT SubArray is more than 1, repeat previous step

if arrayLength of LeftSubArray is 1, check arrayLength of RighSubArray

If ArrayLength of RightSubArray is also 1, ready to merge. Else, repeat 2nd step - but for RightSubArray instead of Left

Select both subArrays, and merge them together in Sorted order

1. compare and take the smallest value from the front of each array



# MergeSort Simulation

Find midPoint of array;
split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves

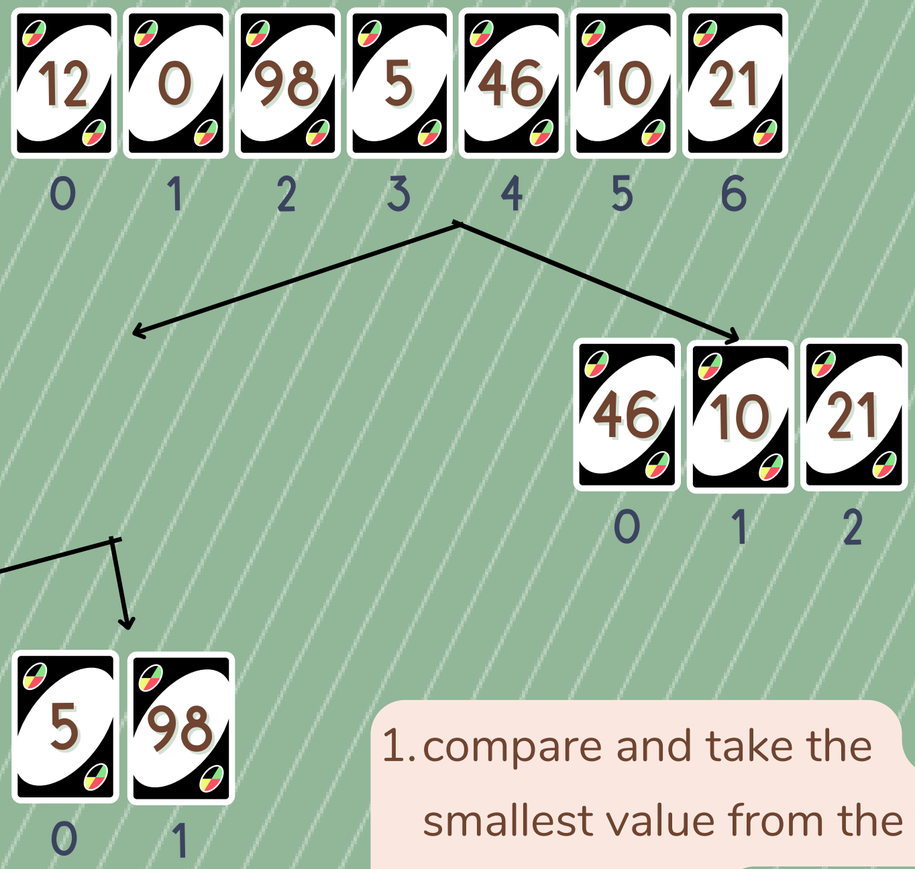
Find the mid-point to divide array in half , as evenly as possible

if arrayLength of RIGHT SubArray is more than 1, repeat previous step

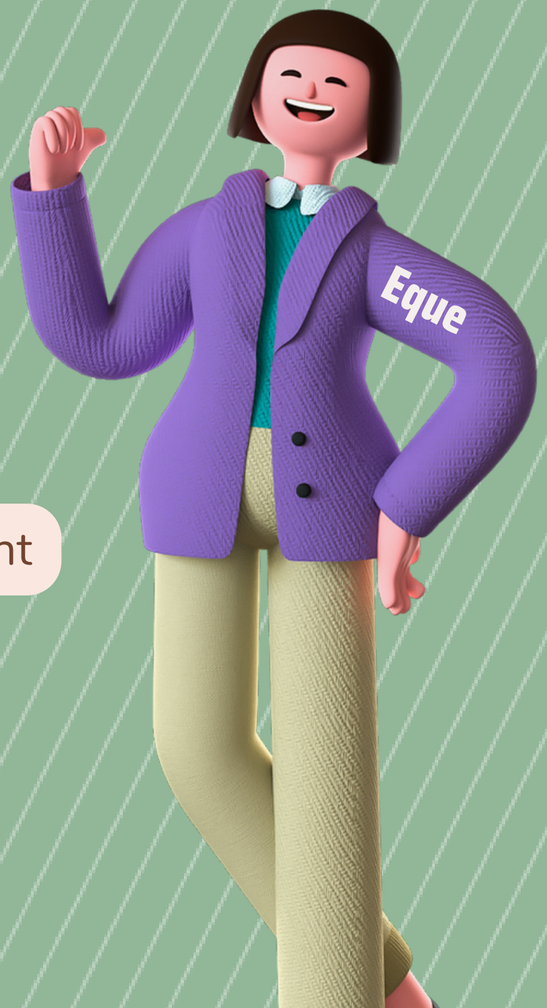
if arrayLength of LeftSubArray is 1, check arrayLength of RighSubArray

If ArrayLength of RightSubArray is also 1, ready to merge. Else, repeat 2nd step - but for RightSubArray instead of Left

Select both subArrays, and merge them together in Sorted order



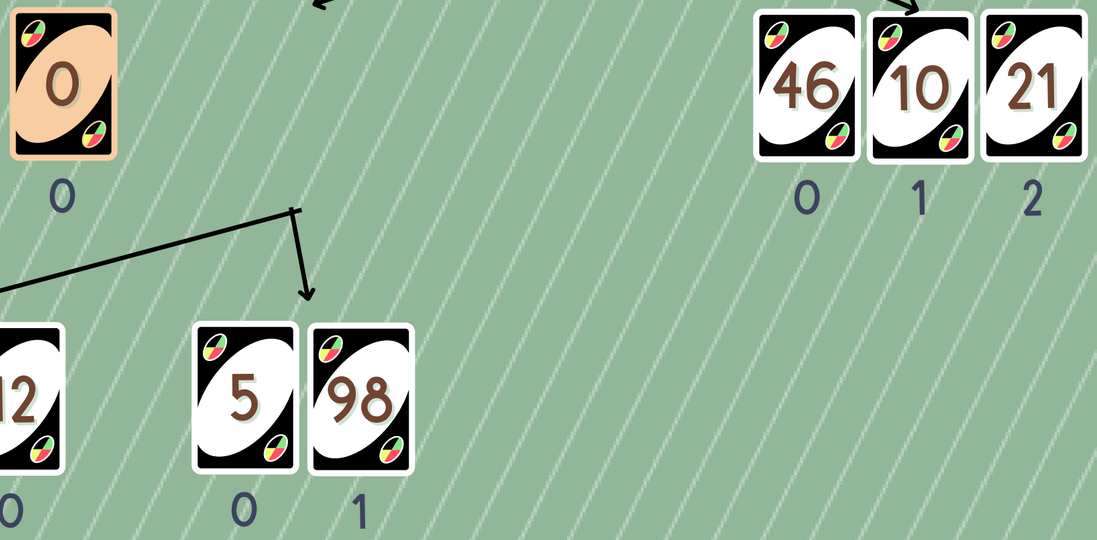

1. compare and take the smallest value from the front of each array



# MergeSort Simulation

Find midPoint of array;
split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves


compare and take the smallest value from the front of each array

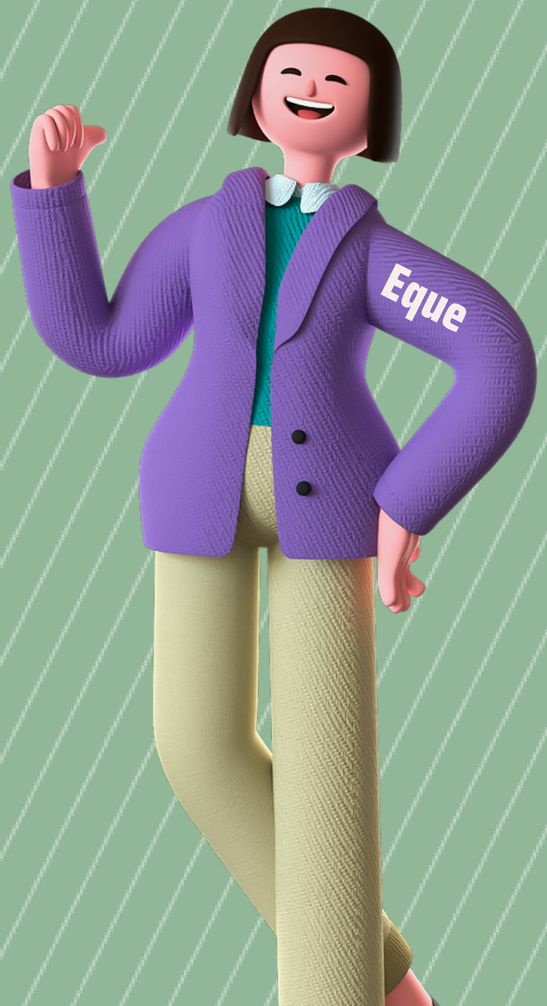
Find the mid-point to divide array in half , as evenly as possible

if arrayLength of RIGHT SubArray is more than 1, repeat previous step

if arrayLength of LeftSubArray is 1, check arrayLength of RighSubArray

If ArrayLength of RightSubArray is also 1, ready to merge. Else, repeat 2nd step - but for RightSubArray instead of Left

Select both subArrays, and merge them together in Sorted order



# MergeSort Simulation

Find midPoint of array;
split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves





Find the mid-point to divide array in half , as evenly as possible

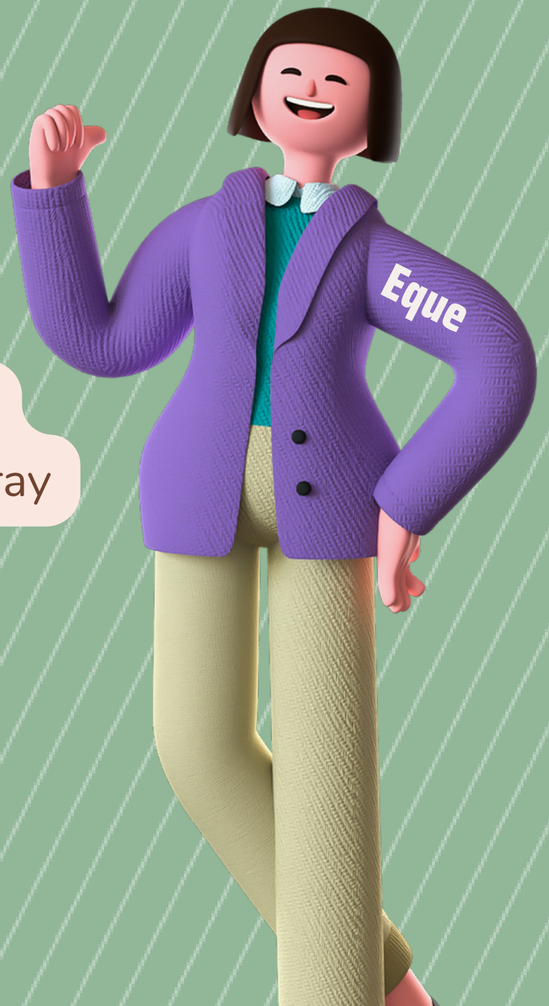
if arrayLength of RIGHT SubArray is more than 1, repeat previous step

if arrayLength of LeftSubArray is 1, check arrayLength of RighSubArray

If ArrayLength of RightSubArray is also 1, ready to merge. Else, repeat 2nd step - but for RightSubArray instead of Left

Select both subArrays, and merge them together in Sorted order

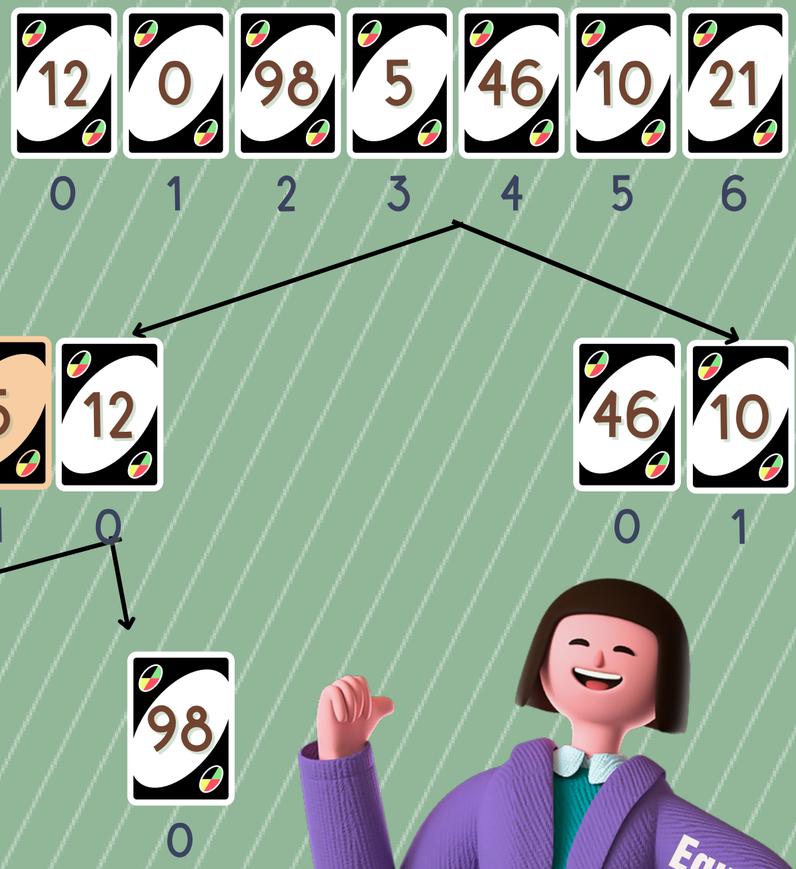
compare and take the smallest value from the front of each array



# MergeSort Simulation

Find midPoint of array;
split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves




Find the mid-point to divide array in half , as evenly as possible

if arrayLength of RIGHT SubArray is more than 1, repeat previous step

if arrayLength of LeftSubArray is 1, check arrayLength of RighSubArray

If ArrayLength of RightSubArray is also 1, ready to merge. Else, repeat 2nd step - but for RightSubArray instead of Left

Select both subArrays, and merge them together in Sorted order

since one of the arrays is already empty, simply copy the remaining values of the remaining array into the sorted array

# MergeSort Simulation

Find midPoint of array; split evenly

1. call MergeSort for left half
2. call MergeSort for right half
3. merge the two halves

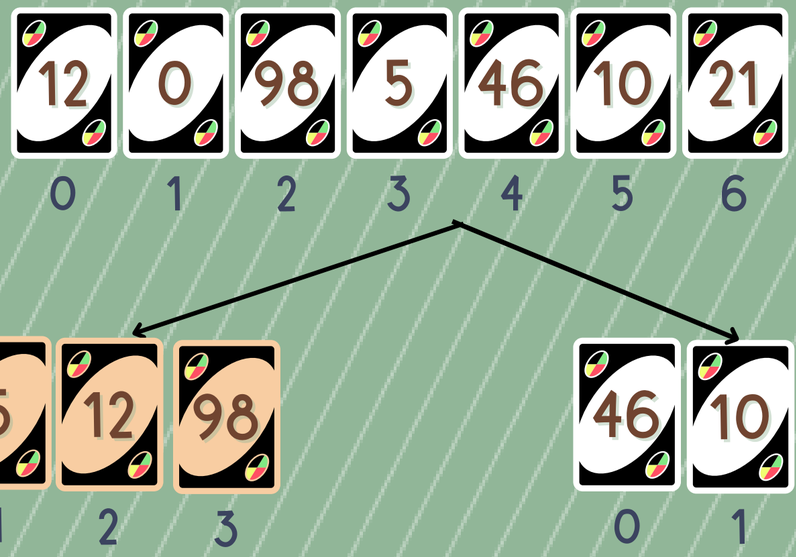
Find the mid-point to divide array in half , as evenly as possible

if arrayLength of RIGHT SubArray is more than 1, repeat previous step

if arrayLength of LeftSubArray is 1, check arrayLength of RighSubArray

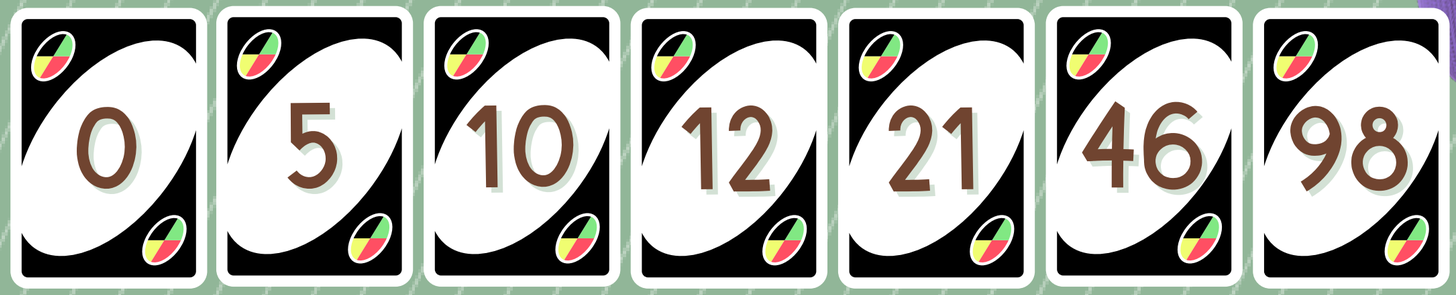
If ArrayLength of RightSubArray is also 1, ready to merge. Else, repeat 2nd step - but for RightSubArray instead of Left

Select both subArrays, and merge them together in Sorted order




Now, repeat the same process with the right half

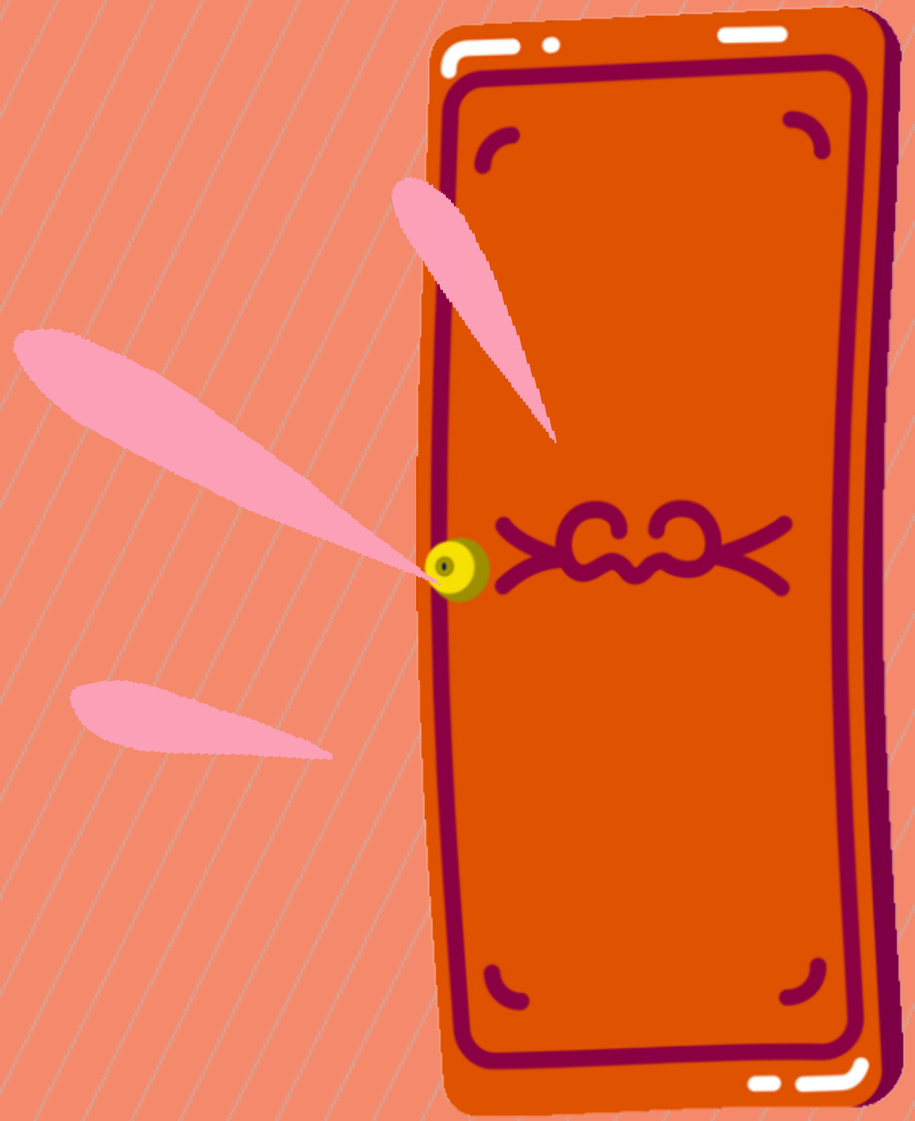
# MergeSort Simulation – Final Answer




0 1 2 3 4 5 6

Internet vs Group's code

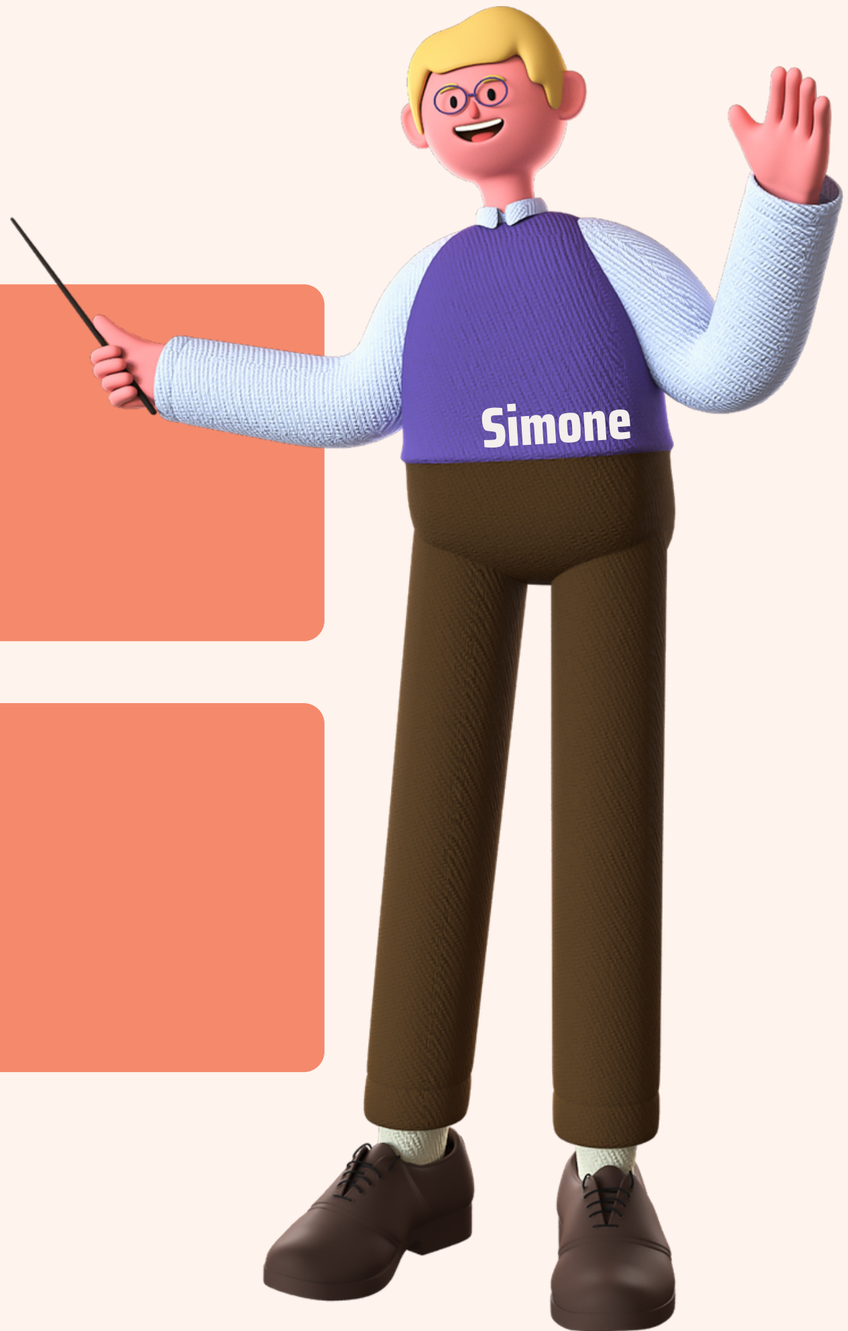
it's Simone



# Functions

- mergeSort( int arr[], int l, int r)
  - Divides the input array into halves.

- merge( int arr[], int l,int m, int r)
  - Merges the two sorted array.



# mergeSort Function

```
void mergeSort( int arr[], int l, int r ) {
        if (l < r) {

            int m = ( l + r ) / 2;

            mergeSort(arr, l, m);
            mergeSort(arr, m + 1, r);

            merge( arr, l, m, r );
        }
}
```

Finding mid element

Recursively sorting both the halves

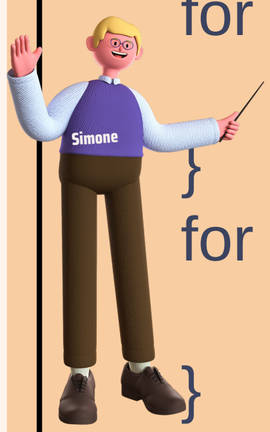
Merge the array

## Internet merge Function

```
void merge(int arr[], int l, int m, int r) {
        int i, j, k;
        int n1 = m - l + 1;
        int n2 = r - m;

        int L[n1], R[n2];

        for (i = 0; i < n1; i++){
            L[i] = arr[l + i];
        }
        for (j = 0; j < n2; j++){
            R[j] = arr[m + 1+ j];
        }
```

Create temp arrays

Copy data to temp array




## Groups merge Function

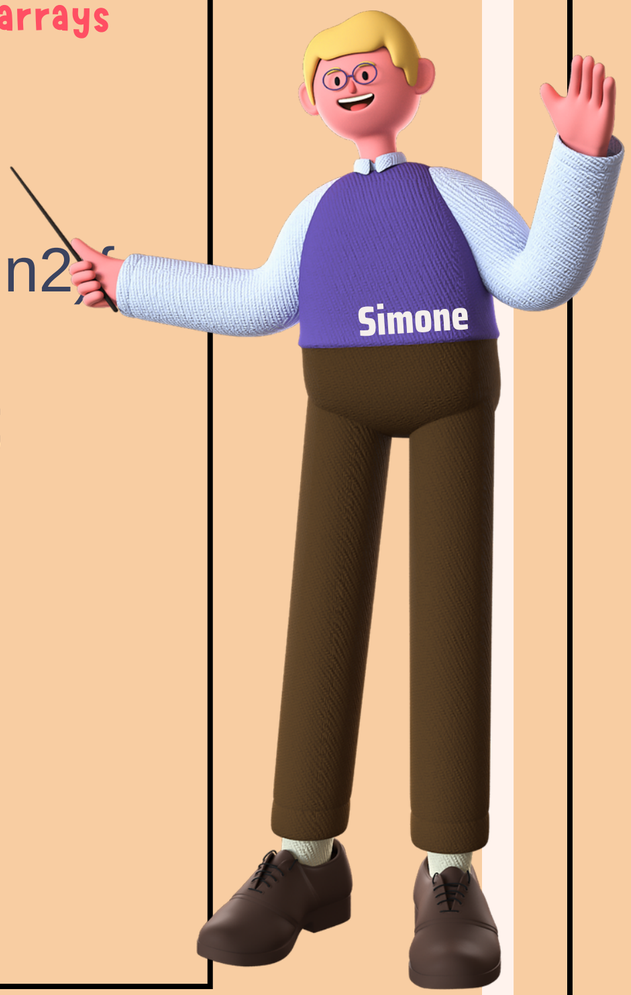
```
void merge(int arr[], int l, int m, int r) {
        int i, j, k;
        int size1 = m - l + 1;
        int size = r - l + 1;

        int temp[size];

        for(i=0; i<size; i++) {
            temp[i] = arr[l+i];
        }
```

Create 1 temp array

1 loop to copy data to temp

# Internet merge Function

```
Merge the temp arrays
i = 0;
j = 0;
k = l;
while (i < n1 && j < n2) {
        if (L[i] <= R[j]){
              arr[k] = L[i];
              i++;
        } else {
              arr[k] = R[j];
              j++;
        }
        k++;
}
```




# Group's merge Function

```
    i = 0;
    j = size1;
    k = l;

    while(i<size1 && j<size) {
          if(temp[i] < temp[j]) {
                arr[k] = temp[i];
                i++;
          } else {
                arr[k] = temp[j];
                j++;
          }
          k++;
    }
```

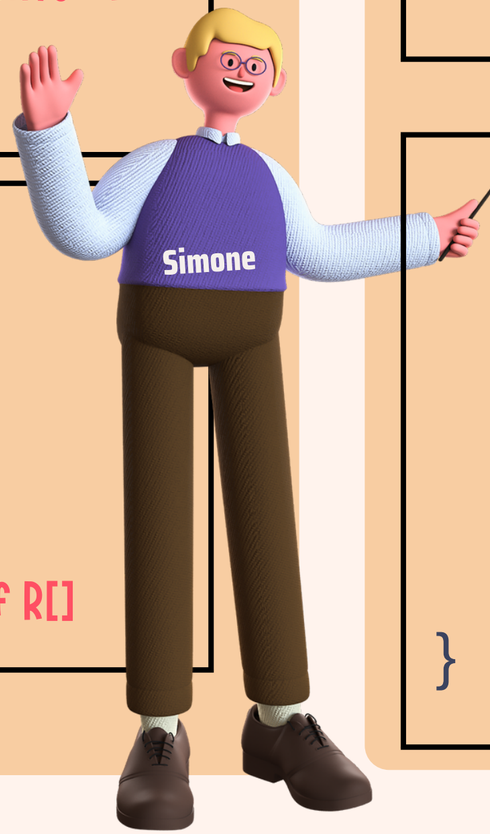
# Internet merge Function

```
while (i < n1){
        arr[k] = L[i];
        i++;
        k++;
}
```

Copy the remaining elements of L[]

```
while (j < n2){
        arr[k] = R[j];
        j++;
        k++;
}
}
```

Copy the remaining elements of R[]




# Group's merge Function

```
if(j<size) {
    i = j;
    size1 = size;
}
```

Check where to start copying the remaining elements

```
while (i<size1) {
        arr[k] = temp[i];
        k++;
        i++;
}
}
```

Copy the remaining elements of temp


# Thank You

SEE YOU NEXT TIME