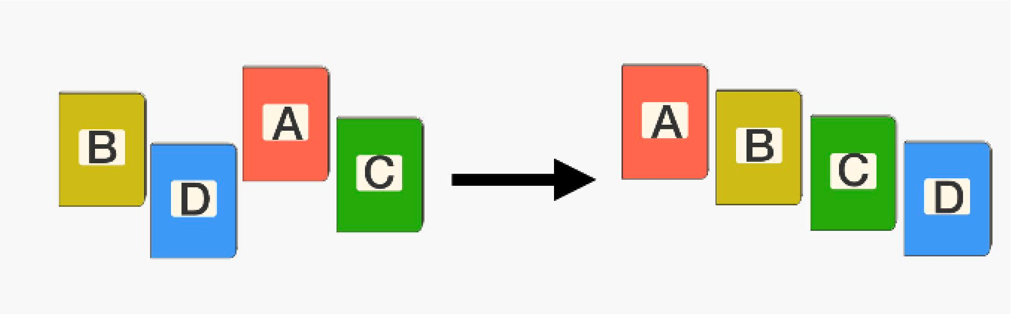
Data Structures and Algorithms

Sorting Algorithms



Prepared by:

Eng. Malek Al-Louzi

School of Computing and Informatics– Al Hussein Technical University

Spring 2021/2022

Outlines

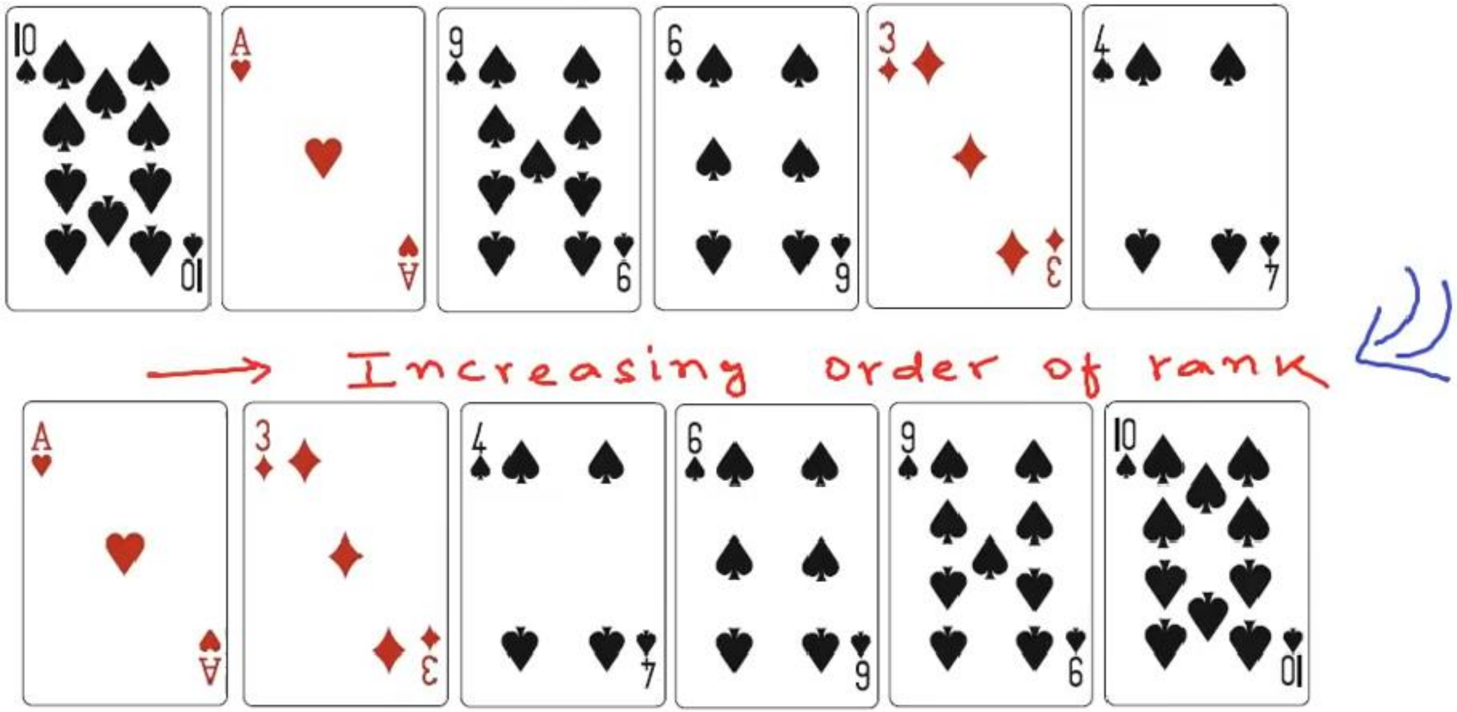
- Introduction
- Selection Sort
- Divide and Conquer Strategy
- Merge Sort

Introduction

- Sorting as a concept is deeply embedded in a lot of things that we do.
- It is quite often that we like to arrange things or data in a certain order.
- Sometimes to improve the readability of that data, at others to be able to search or extract some information quickly out of that data.

Introduction

- For example, when are playing a card game, even though the number of cards in our hand is small, we like to keep our hand of cards sorted by rank or suit.



Introduction

- When we go to a travel website to book a hotel, then the website gives us options of sorting the hotels:
  - By price from low to high, by star rating, or by guest rating.
  - Sorting is a helpful feature in this case.
- There are many places where we like to keep our data sorted.

2012 Summer Olympics medal table[15]

Rank	NOC	Gold	Silver	Bronze	Total
1	United States (USA)	46	29	29	104
2	China (CHN)	38	27	23	88
3	Great Britain (GBR)*	29	17	19	65
4	Russia (RUS)	24	26	32	82
5	South Korea (KOR)	13	8	7	28
6	Germany (GER)	11	19	14	44
7	France (FRA)	11	11	12	34
8	Italy (ITA)	8	9	11	28
9	Hungary (HUN)	8	4	6	18
10	Australia (AUS)	7	16	12	35
11	Japan (JPN)	7	14	17	38
12	Kazakhstan (KAZ)	7	1	5	13
13	Netherlands (NED)	6	6	8	20
14	Ukraine (UKR)	6	5	9	20

Introduction

- Sorting formal definition:

Sorting is arranging the elements in a list or collection in increasing or decreasing order of some property.

- The list should be **homogenous**, that is all the elements in the list should be of same type.

Introduction

- For example, if we have a list of integers:

2, 3, 9, 4, 6

- Sorting it in increasing order of value will mean rearranging the elements like this:

⇒ 2, 3, 4, 6, 9 (increasing order of value)

- Sorting it in decreasing order of value will mean rearranging the elements like this:

⇒ 9, 6, 4, 3, 2 (decreasing order of value)

Introduction

- From the definition, we can sort on any property.
- If we want to sort this list based on increasing number of factors.

2, 3, 9, 4, 6

$\Rightarrow$ 2, 3, 9, 4, 6 (increasing order of number of factors)

Introduction

- The previous list was list of integers, we may have a list of any datatype.
- We may want to sort a list of strings or words in lexicographical order, the order in which they will occur in dictionary.
- A list of strings like this:

"fork", "knife", "mouse", "screen", "key"

- In lexicographical order will be arranged in this order:

"fork", "key", "knife", "mouse", "screen"

Introduction

- We may have a list of complex datatypes as well.
- A Hotel object in a hotel list (like a list of available hotels on internet) is a complex type.
- Hotel may have many properties, like it’s price, distance from the city center, guest rating, star rating, and more.
- The list can be sorted on any of these properties.

Introduction

- Sorted data is good not just for presentation or manual retrieval of information.
- Even when we are using computational power of machines, sorted data is helpful.
- For example, if a list stored in computer as unsorted, then to search something in this list, we will have to run Linear Search.

Unsorted: Linear Search

Introduction

- In linear search, if there are **n** elements in the list, we will make **n** comparisons in the worst case.

$$size = n \rightarrow n \text{ comparisions}$$

- What if **n** is large? If 1 comparison takes 1 millisecond.

$$n = 2^{64} \rightarrow 2^{64} \text{ ms}$$

- This will take **years**!

Introduction

- If list is sorted, we can use Binary Search.

Sorted: Binary search

- In Binary search, if size of the list is equal to **n**:

size $= n \longrightarrow \log_2 n$ comparisions

- If we compare it with linear search:

$n = 2^{64} \longrightarrow 64$ ms

Introduction

- Sorting as a problem is well studied and researches have gone into devising efficient algorithms for sorting.
- Some of the sorting algorithms that are available:

Bubble sort

Selection sort

Insertion sort

Merge sort

Quick sort

Heap sort

Counting sort

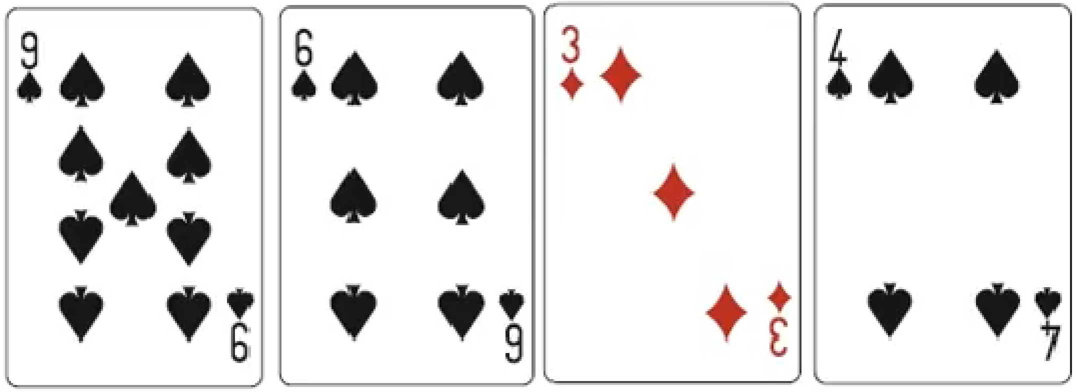
Radix sort

Introduction

- This is not all, there are more.
- You can imagine how important sorting as a problem is.
- We have so many algorithms for sorting that have been designed over a period of time.

Selection Sort

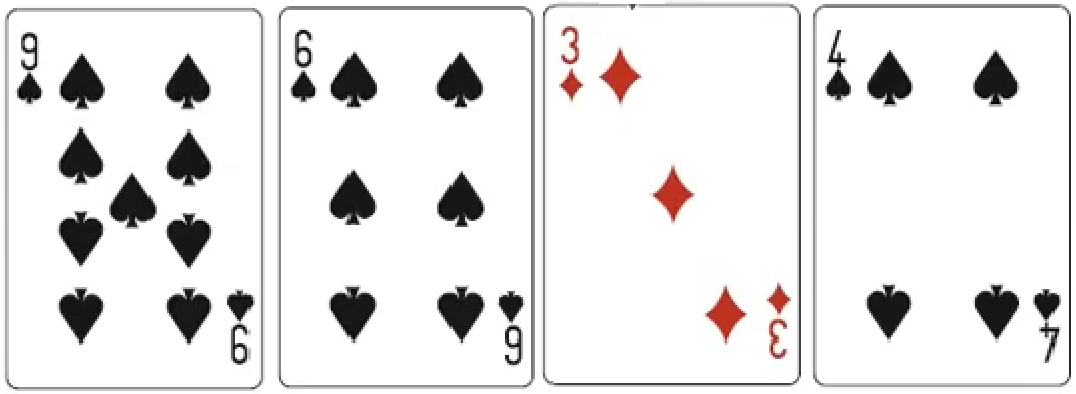
- Let's think of a simple sorting scenario.
- We have a set of cards, and we want to arrange these cards in increasing order of rank.



Selection Sort

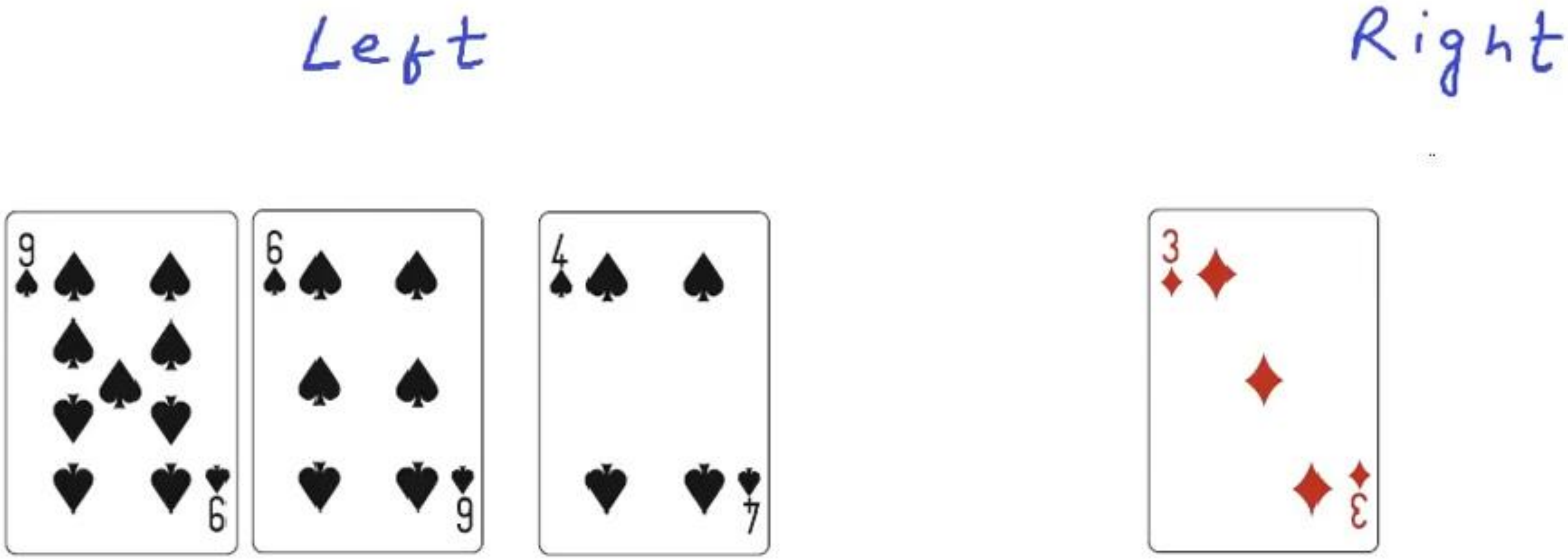
- One simple thing what we can do is initially we keep all the cards in our left hand:

Left



Selection Sort

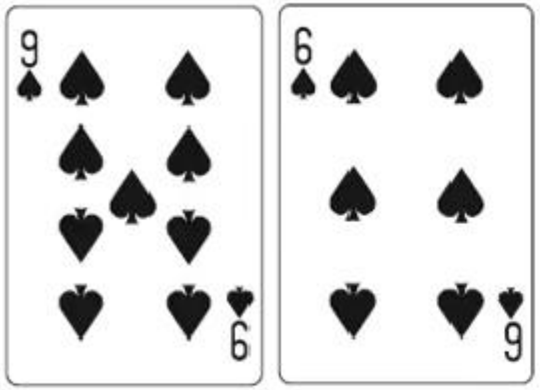
- And then, first we can select the minimum card out of these cards and move it to the right hand:



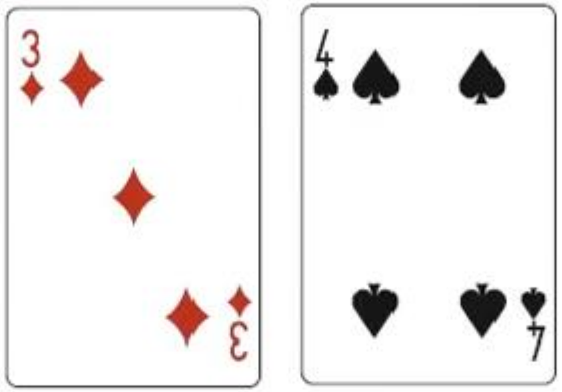
Selection Sort

- Now, once again, from whatever card is left in the left hand, we can select the minimum and move it next to the previous card in the right hand.

Left



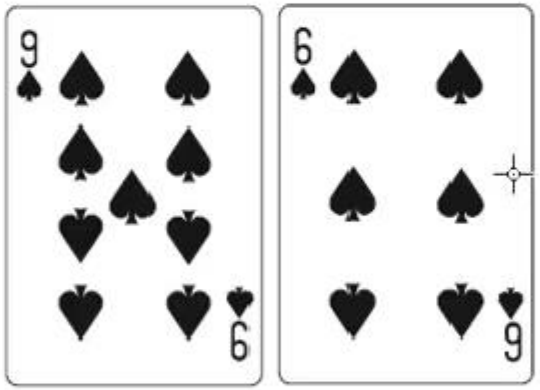
Right



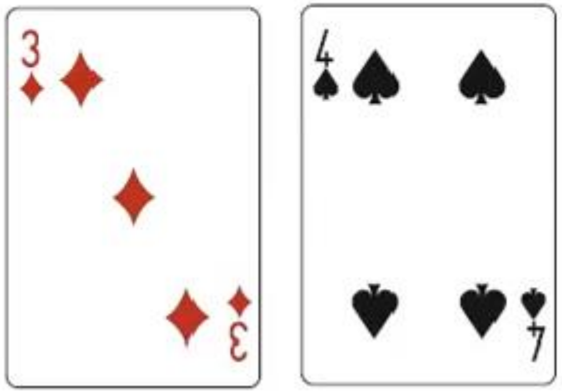
Selection Sort

- We can go on repeating this process.
- At any stage during this process, the left hand will be an unsorted set of cards, and the right hand will be sorted set of cards.

Left - Unsorted

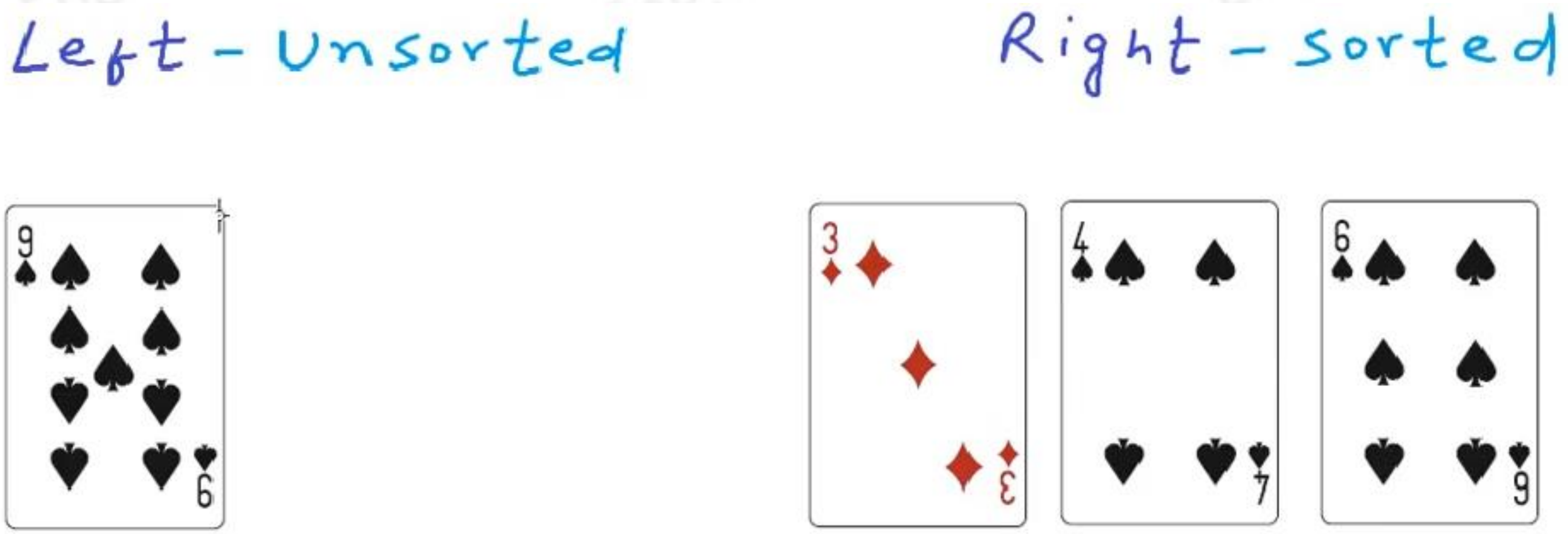


Right - sorted



Selection Sort

- After 3 and 4, 6 will go to the right hand.

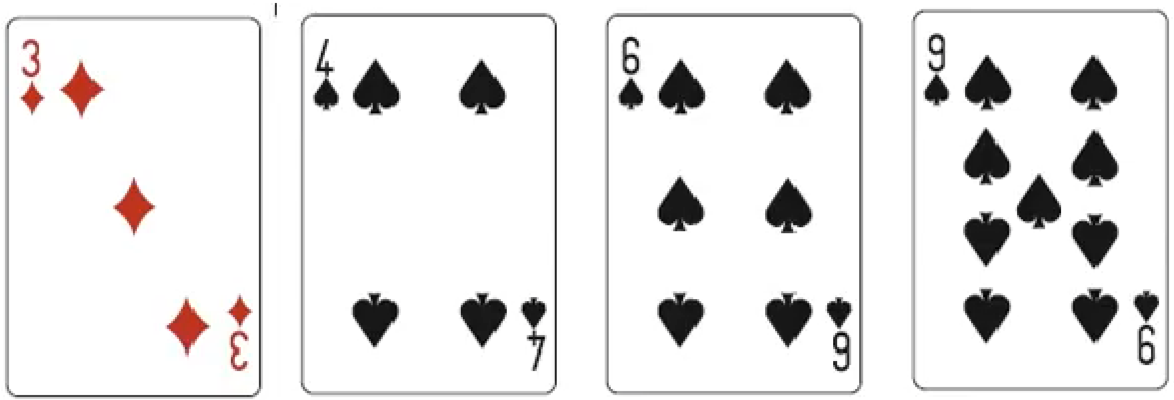


Selection Sort

- Finally, 9 will go to the right hand.

Left - Unsorted

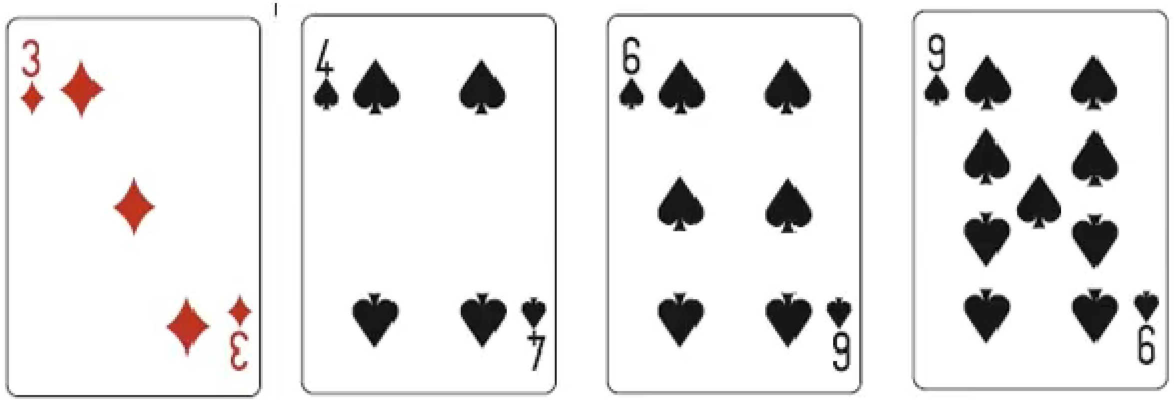
Right - sorted



Selection Sort

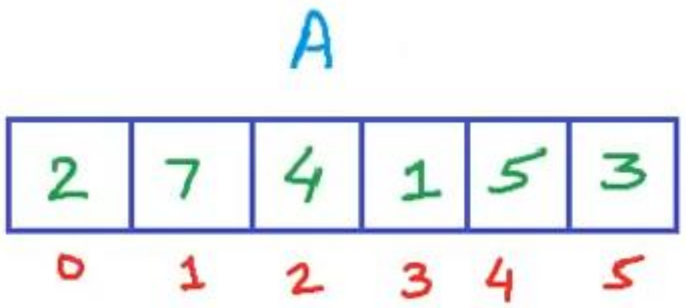
- In the end, right hand will be a sorted arrangement of cards.
- Cards will be arranged in increasing order of rank.

Right - sorted



Selection Sort

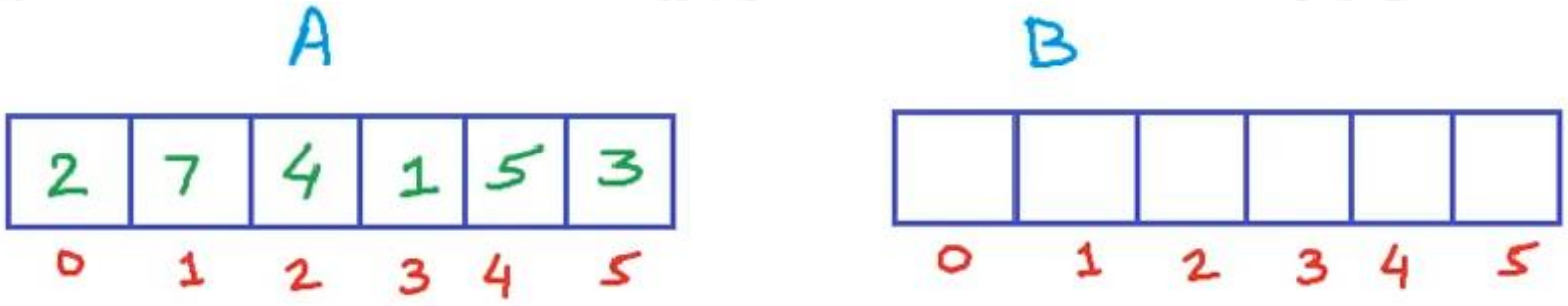
- Now, we want to write a program to sort a list of integers given to us in the form of an array.



- To sort this list, we can do something similar to what we were doing in our cards example.

Selection Sort

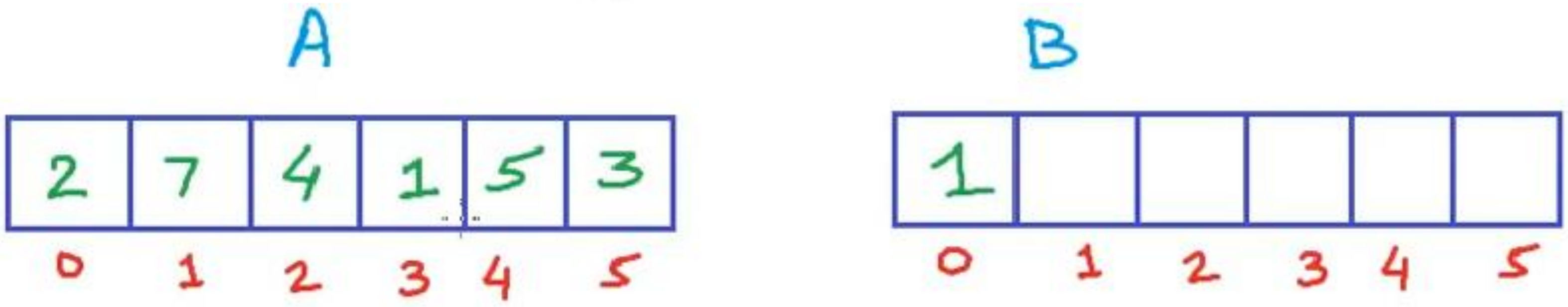
- We can create another array of same size as A.



- Now, we can start creating B as a sorted list by selecting the minimum from A at each step.
- There will be multiple passes on A, at the first pass

Selection Sort

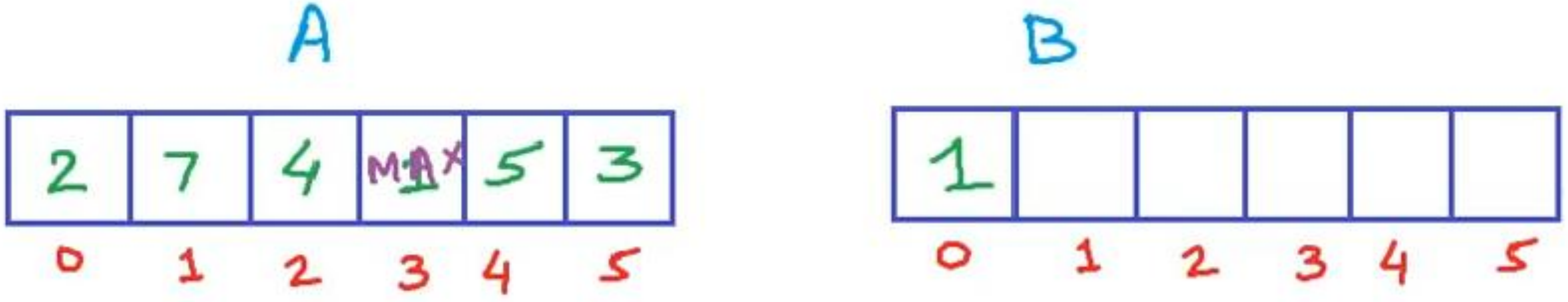
- There will be multiple passes on A, in the first pass 1 will be the minimum, so 1 will go at the 0^{th} position of B.



- There should be a way to mark that 1 has already been selected, so it is not considered in the second time.

Selection Sort

- One way to do this, we can replace the selected element by some large integer, which is guaranteed to the maximum in the array at any step.

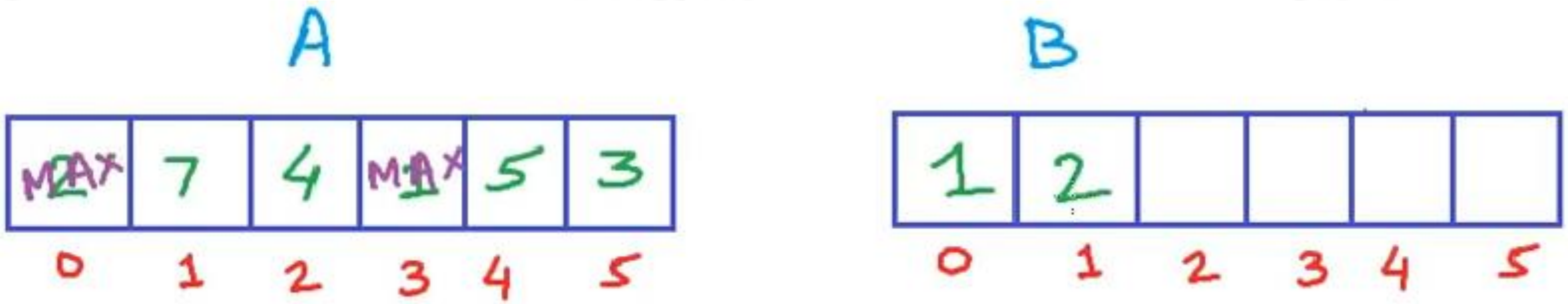


- We can choose this max to be the largest possible value in a 32-bit integer.

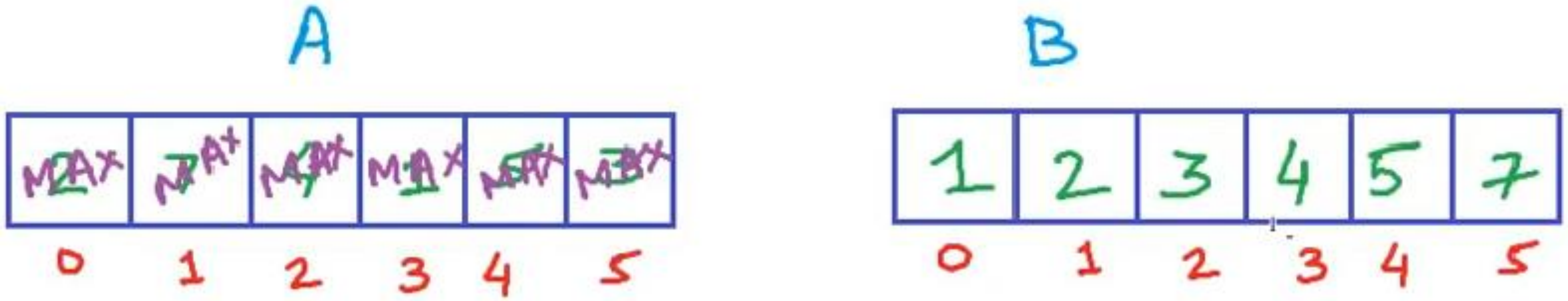
$$MAX = 2^{31} - 1$$

Selection Sort

- Now, we will scan A again for the second largest element that will go to index 1 in B.



- We will go on doing this until all the positions in B are filled.

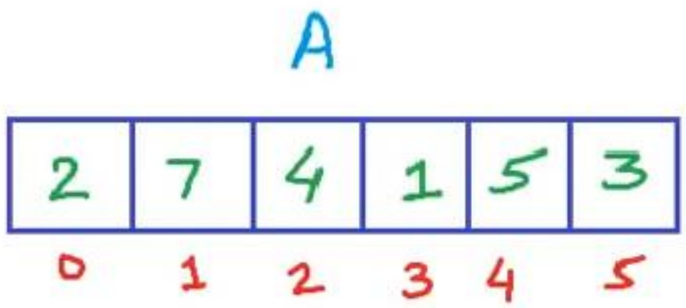


Selection Sort

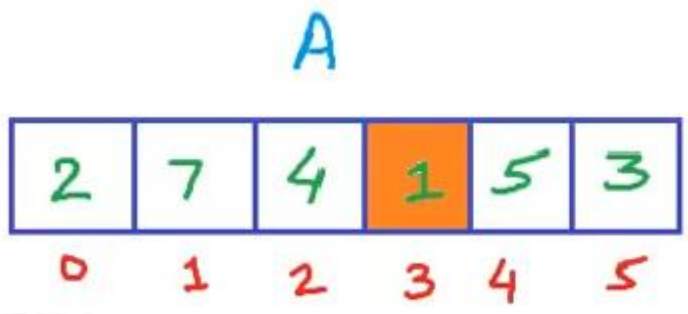
- In the end, we can copy the content of B back to A, so A itself will become a sorted arrangement of its elements.
- This logic will work fine, but we are using an extra array B. Larger the size of A means larger the size of B.
- We can do something similar where we will select the minimum element at each step, but we will not have to use an extra array.

Selection Sort

- We have the following unsorted list:

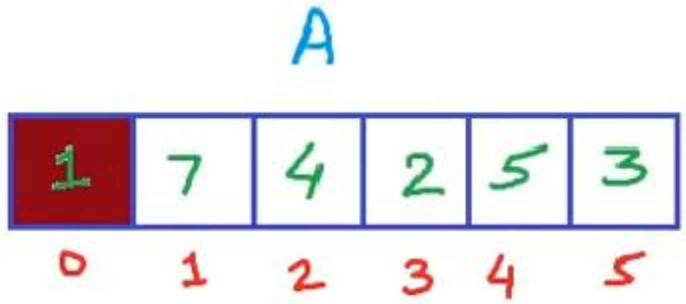


- Again, we will look for the minimum element in the array.



Selection Sort

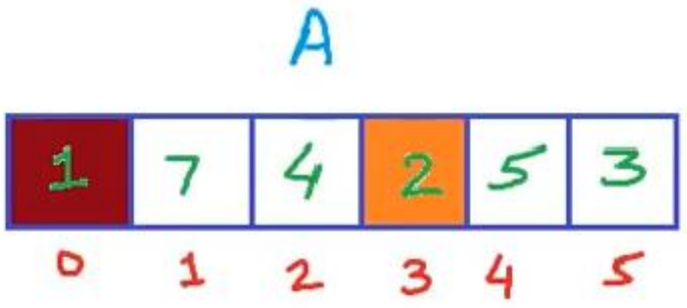
- Instead of filling up 1 at 0^{th} index in another array B, we can swap 1with element at 0^{th} index.



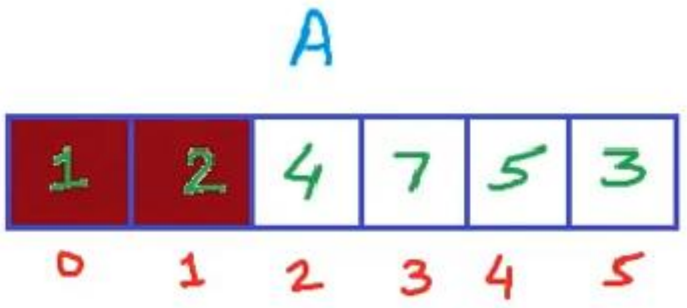
- Now, we need to look for the next minimum, and 1 need not be considered.

Selection Sort

- We can scan the elements from index 1 to index 5 to find the second minimum, which is number 2 at index 3.

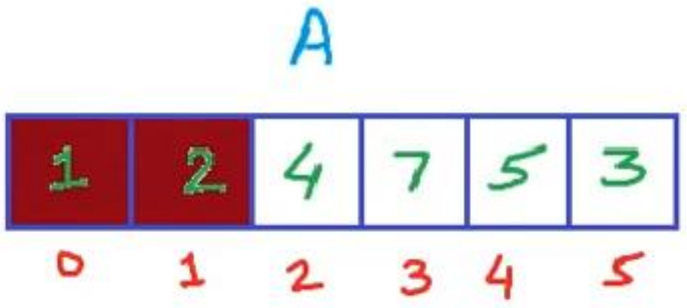


- Now 2 deserves to be at position 1, so we can swap 2 with the element at position 1.



Selection Sort

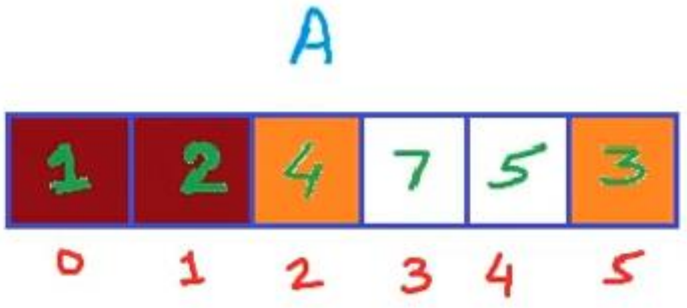
- As we can notice, in each pass, we are finding out the element that should go to a particular position.
- At any stage, the array is divided into two parts, some part of it is sorted (the cells in the brown are sorted).



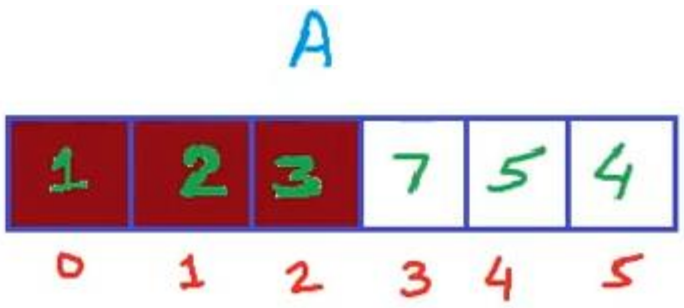
- With each pass we add one more cell to the sorted part, and eventually the whole array will be sorted.

Selection Sort

- Now again, the minimum in index 2 to index 5 is number 3, so number 3 needs to go to position 2 (needs to be swapped with number 4).

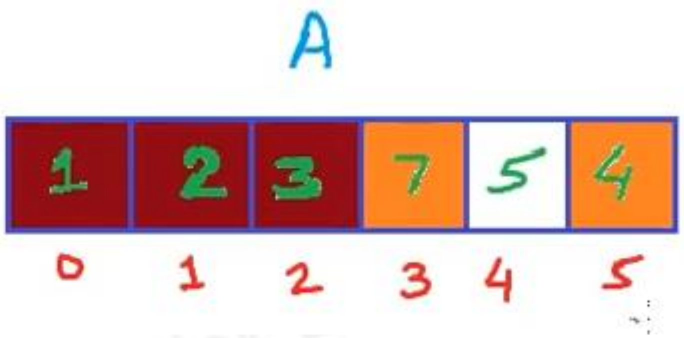


- After swapping:

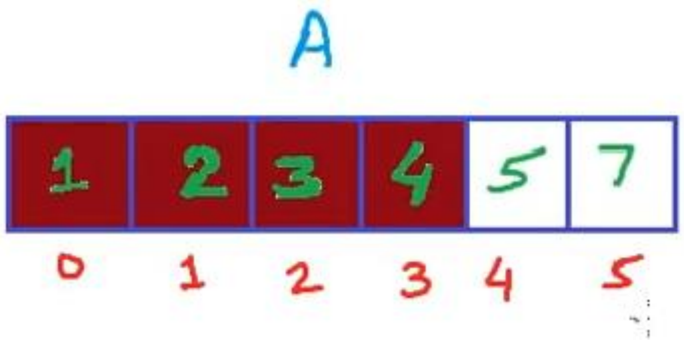


Selection Sort

- We will go on like this.

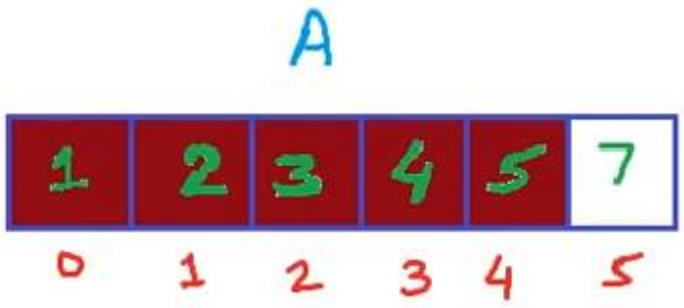


- After swapping:



Selection Sort

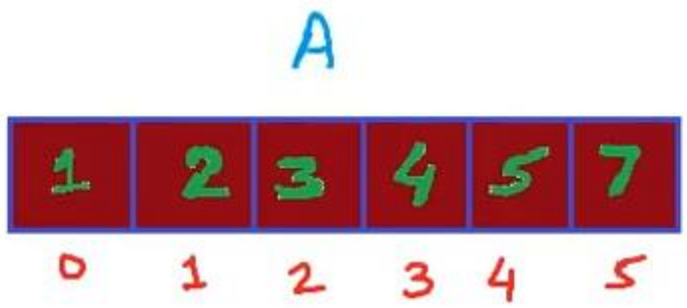
- 5 is at its appropriate position, it doesn't need to be swapped.



- If we have n elements, after n-1 passes, we will have only one more cell left, it will be at its appropriate position.

Selection Sort

- Finally, our list is sorted.



- This logic of selecting the minimum in each pass and putting it at its appropriate position is **Selection Sort Algorithm**.

Selection Sort

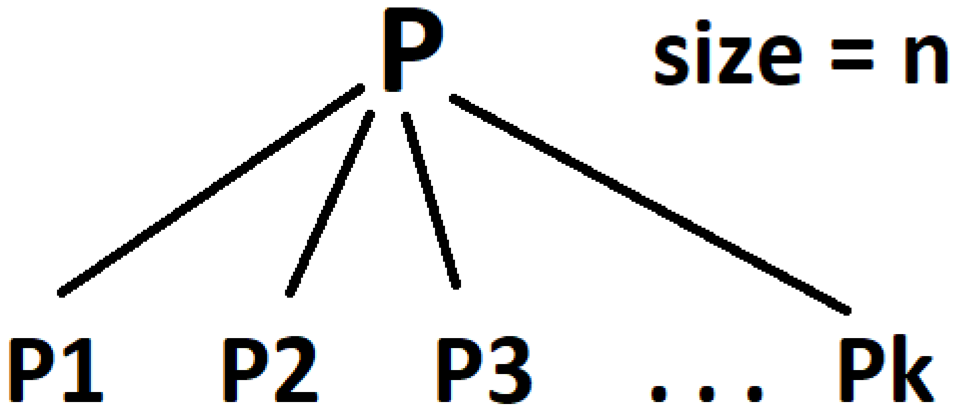
- Let's now write the pseudocode for this algorithm.
- We will write a function named **SelectionSort** that will take the array **A** and the number of elements **n** in the array **A** as arguments.
- The pseudocode in the next slide!

Selection Sort

```
SelectionSort (A, n)
{   for i ← 0 to n-2
    {
        iMin ← i
        for j ← i+1 to n-1
        {
            if (A[j] < A[iMin])
                iMin ← j
        }
        temp ← A[i]
        A[i] ← A[iMin]
    }   A[iMin] ← temp
}
```

Divide and Conquer Strategy

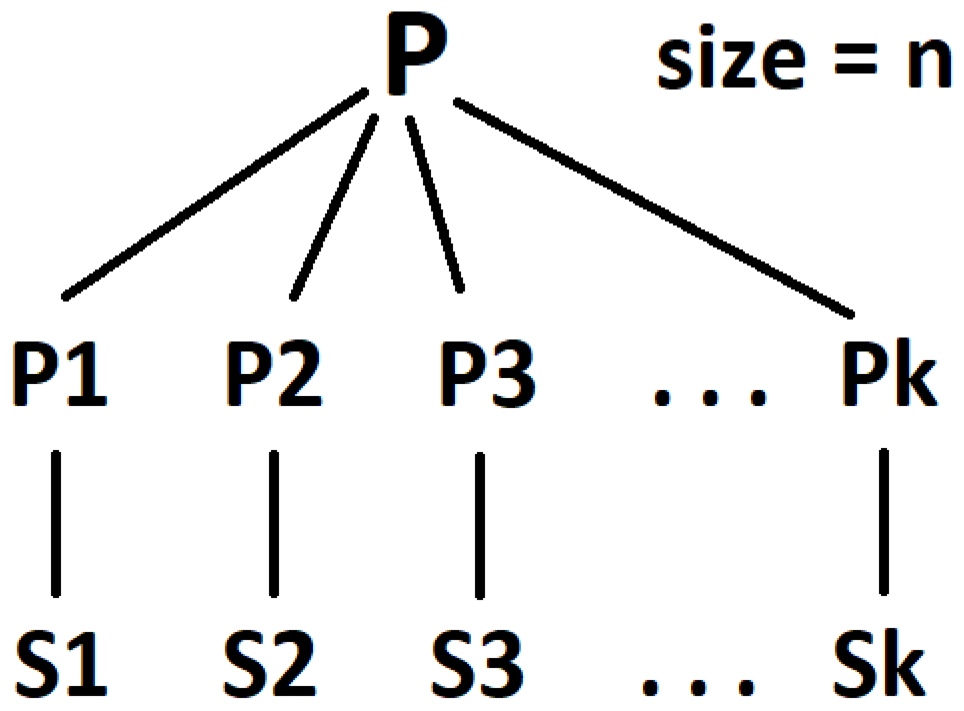
- It is a strategy for solving a problem.
- If a problem of size n is given, n is the size of the input for a problem, then we can break this problem into smaller subproblems.



- We can divide the problem as many subproblems as possible, that depends on you, suppose K subproblems.

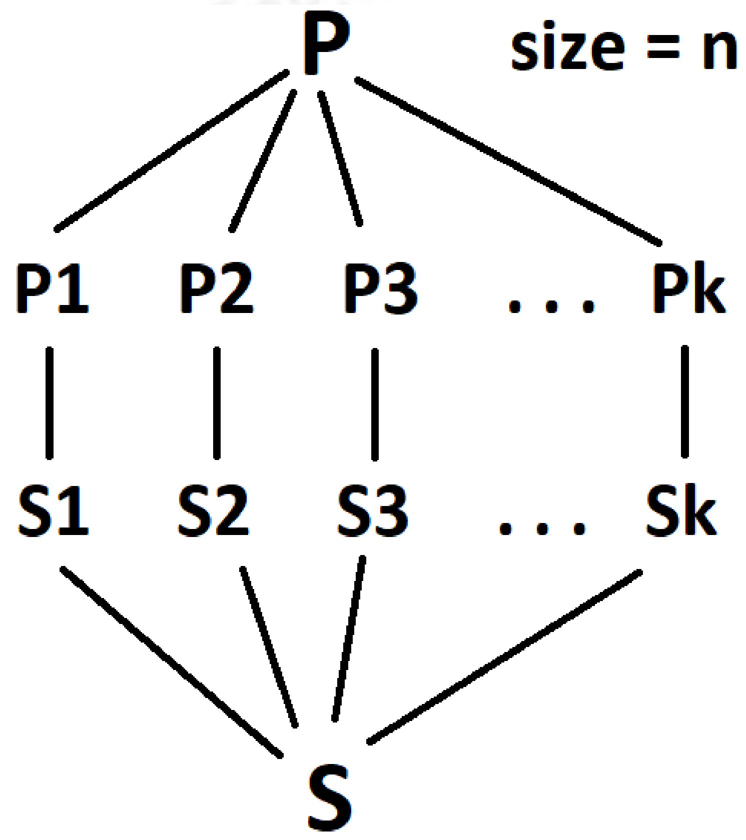
Divide and Conquer Strategy

- Now, these subproblems can be solved individually to obtain their solutions.



Divide and Conquer Strategy

- Once we have solutions for these subproblems, we cam combine these solutions to a solution for the main problem.

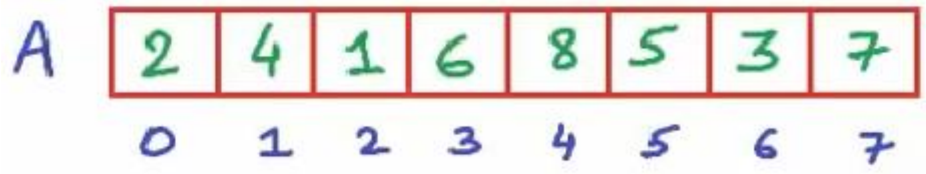


Divide and Conquer Strategy

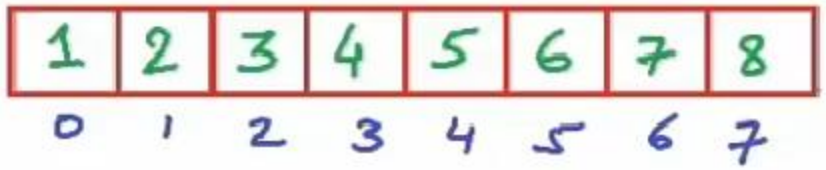
- If a subproblem is also large, then do the same thing.
- Whatever the problem is, the subproblems will be same as that problem.
- For example, if the problem is to sort , then the subproblems should also be sort (each subproblem should be sort only).
- This strategy called **Divide and Conquer**.
- Divide and Conquer is recursive in nature.
- We should have some method for combining their solutions to get a main solution, if you unable to combine then you cannot adopt this strategy.

Merge Sort

- Let's start with an example, given a list of integers in the form of an array.

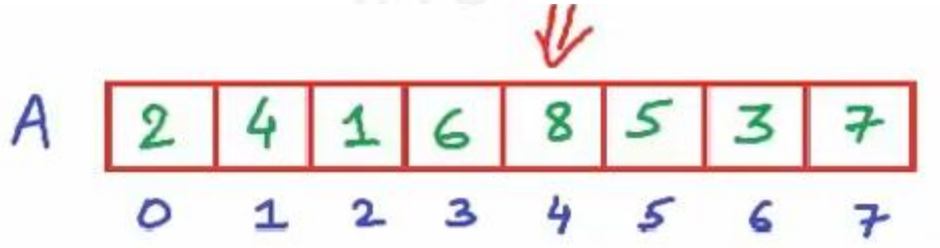


- We want to sort this list in increasing order of the value of integers.



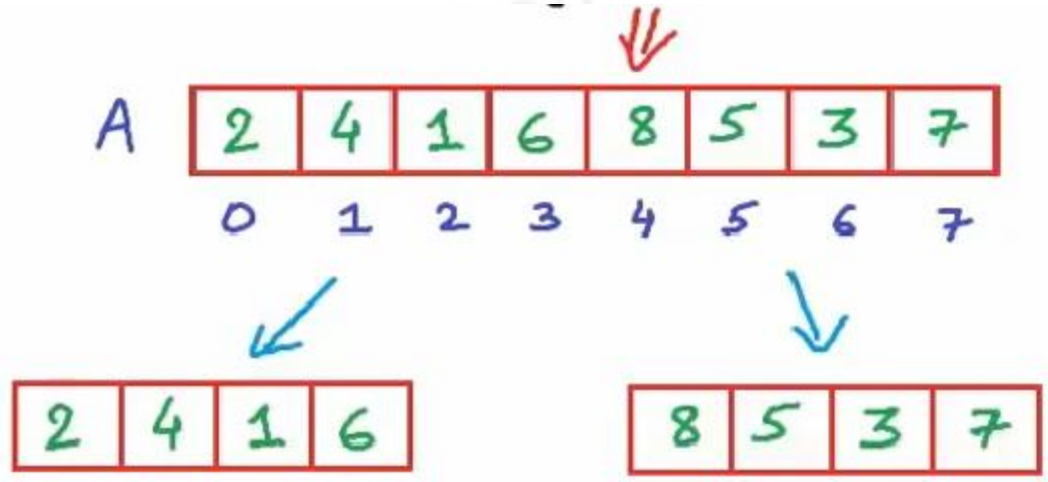
Merge Sort

- We will break this problem into subproblems, we will divide this array into two possibly equal halves.
- So, we will find some middle position and we can say that all the elements before this position belong to the first half, and all the elements on or after this position belong to the second half.



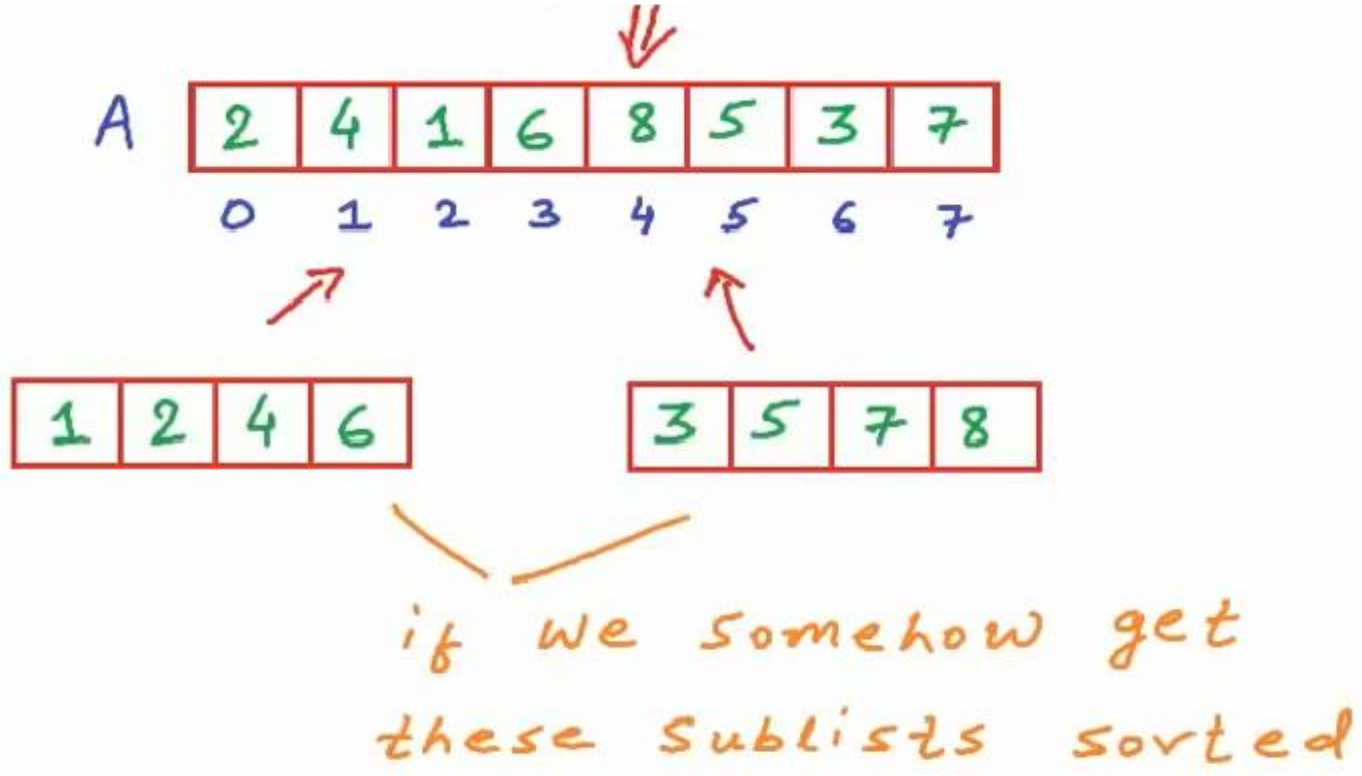
Merge Sort

- If an array would have odd number of elements, on of the halves will have more elements than other half.
- We have 8 elements in the original array in our example, so we have two equal halves.



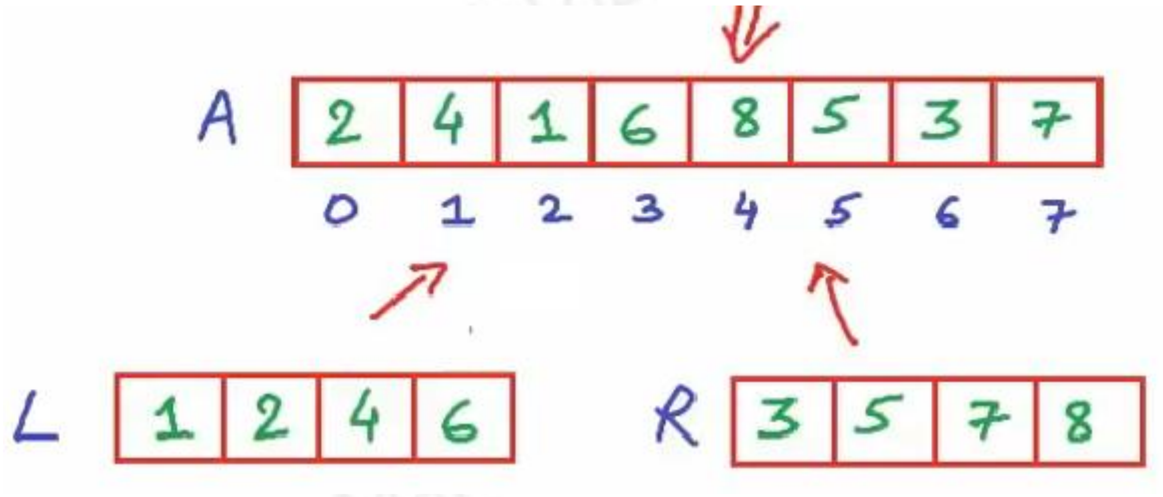
Merge Sort

- What if we are somehow able to sort these two halves, and let's say these two halves are entirely different arrays.
- Then we can merge these two lists together in original list in sorted order.



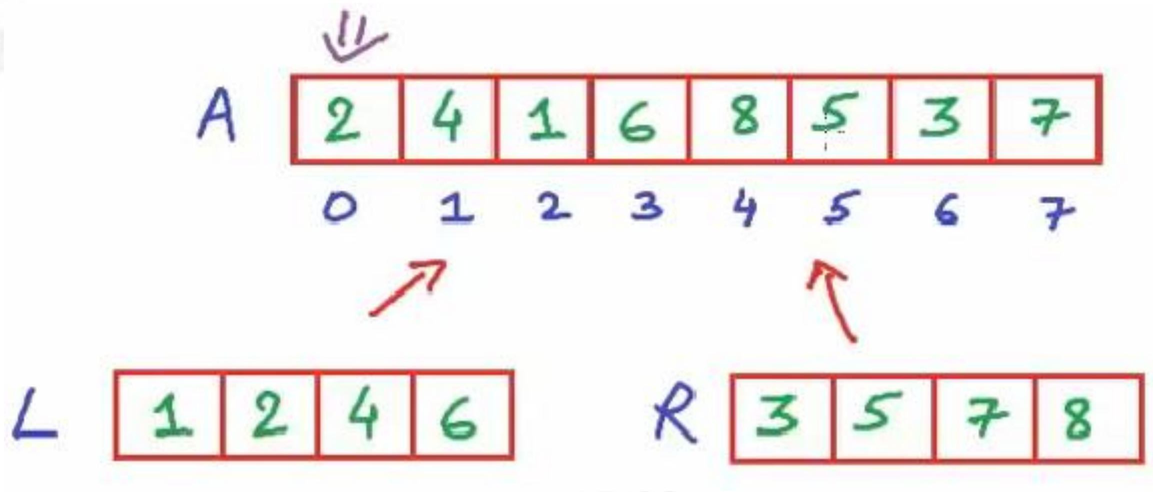
Merge Sort

- Of course, there has to be some algorithm to merge two sorted arrays into a third array in sorted order.
- The algorithm will be straightforward, let's say the first subarray is named L and the second subarray is named R.



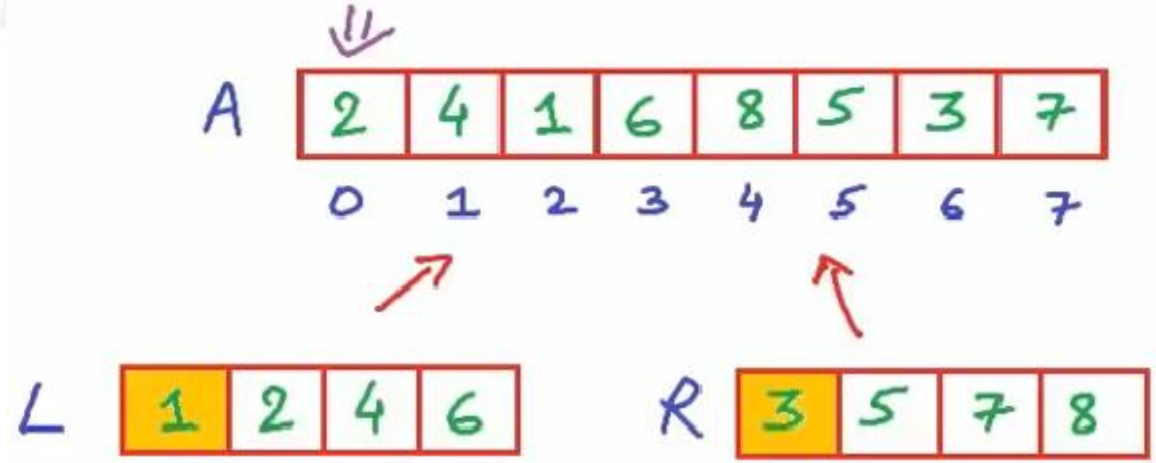
Merge Sort

- Because all the elements in A are present either in L or R, we can start overwriting A from left to right.
- We can start at 0^{th} position in A.



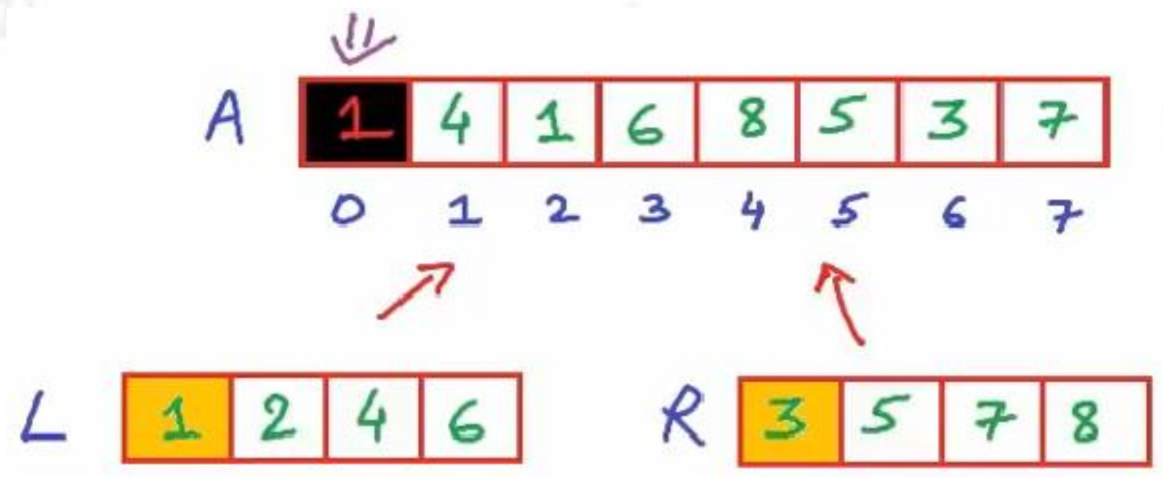
Merge Sort

- At any point, the smallest element will be either the smallest unpicked in L, or smallest unpicked in R.
- We will color the smallest unpicked in L and R by yellow.



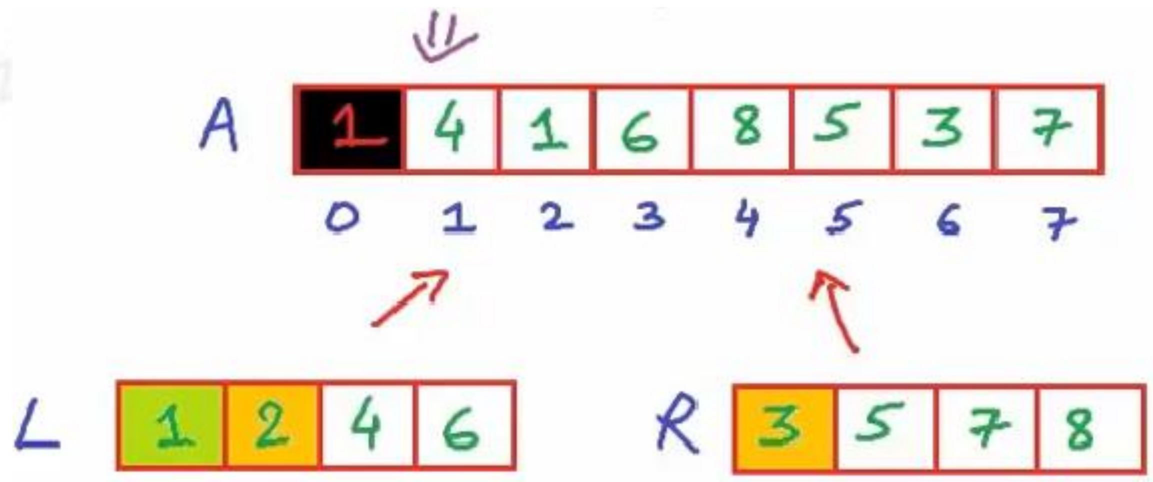
Merge Sort

- We can pick the smaller of the two smallest unpicked in L and R.
- We have two candidates here, 1 and 3, 1 is smaller, so we can write 1 at 0^{th} index.



Merge Sort

- Now, we can look for the number to fill at index 1 in A.
- The cells of the picked elements will be colored in green.



Merge Sort

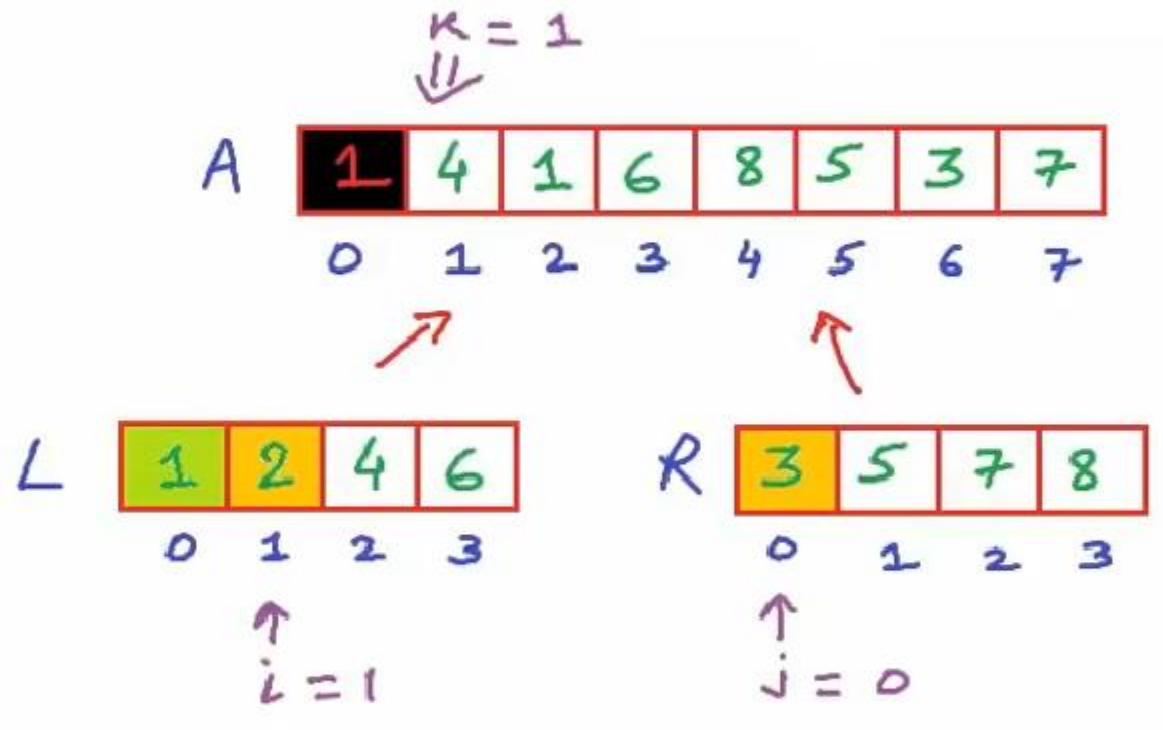
- We will write a pseudocode to merge the elements of two sorted arrays into a third array.
- We want to write a function named **Merge** that will take three arrays as arguments **L**, **R**, and **A** in which it should be merging the two sorted arrays **L** and **R**.
- In the pseudocode:
  - i will mark the smallest unpicked in L.
  - j will mark the smallest unpicked in R.
  - k will mark the index of the position that needs to be filled in A.

Part of the Pseudocode

```
Merge(L, R, A)
{
  nL ← length(L)
  nR ← length(R)
  i ← j ← k ← 0
  while(i < nL && j < nR)
  {
     if( L[i] <= R[j])
     {
        A[k] ← L[i]
        i ← i+1
     }
     else
     {
        A[k] ← R[j]
        j ← j+1
     }
     k ← k+1
  }
```

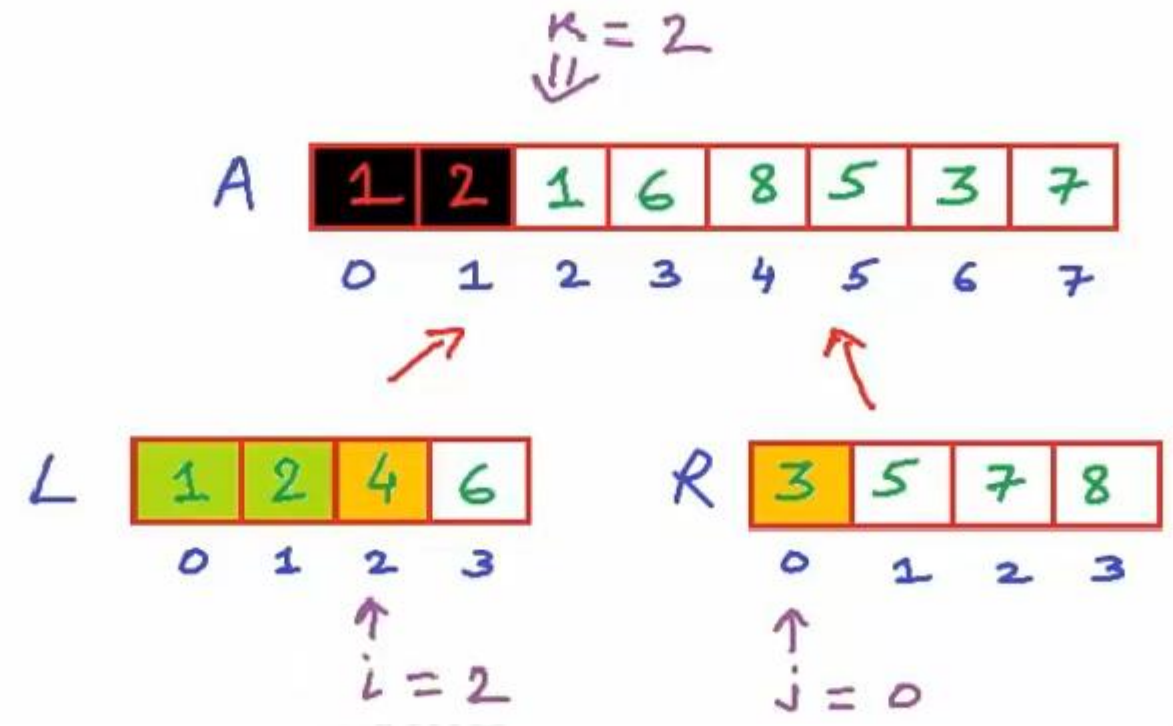
Merge Sort

- At this stage in our example:



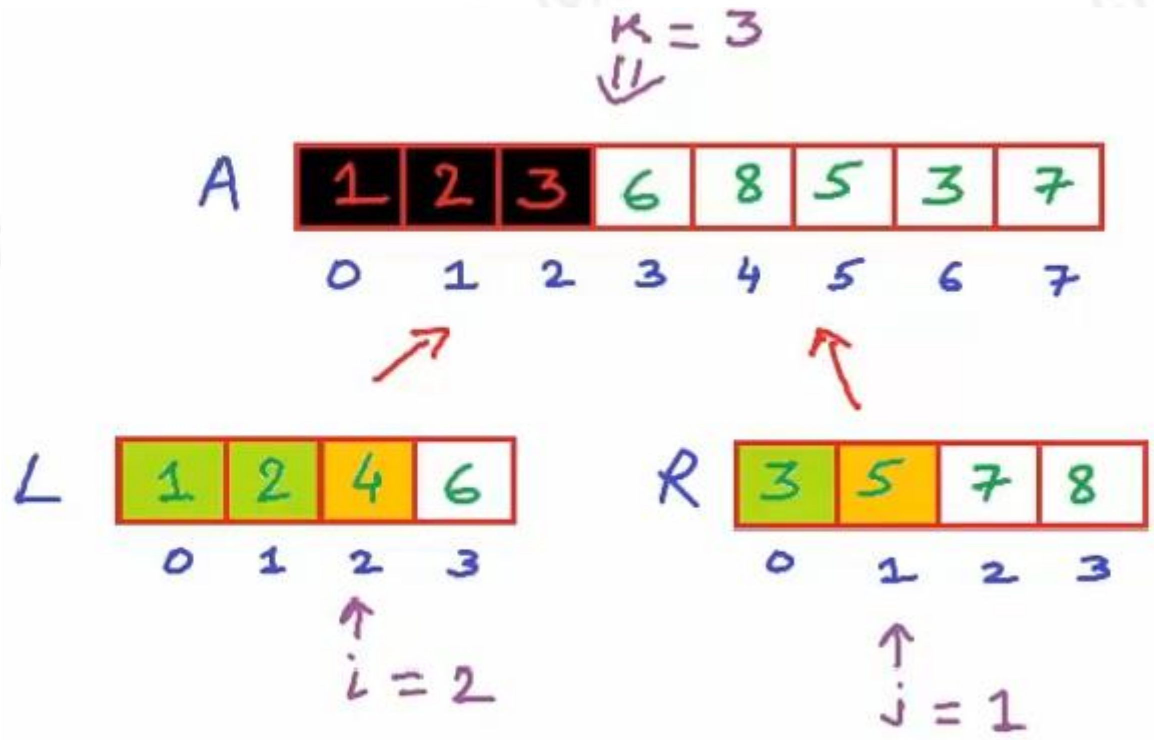
Merge Sort

- We will pick 2 for index 1, and increment both i and k.



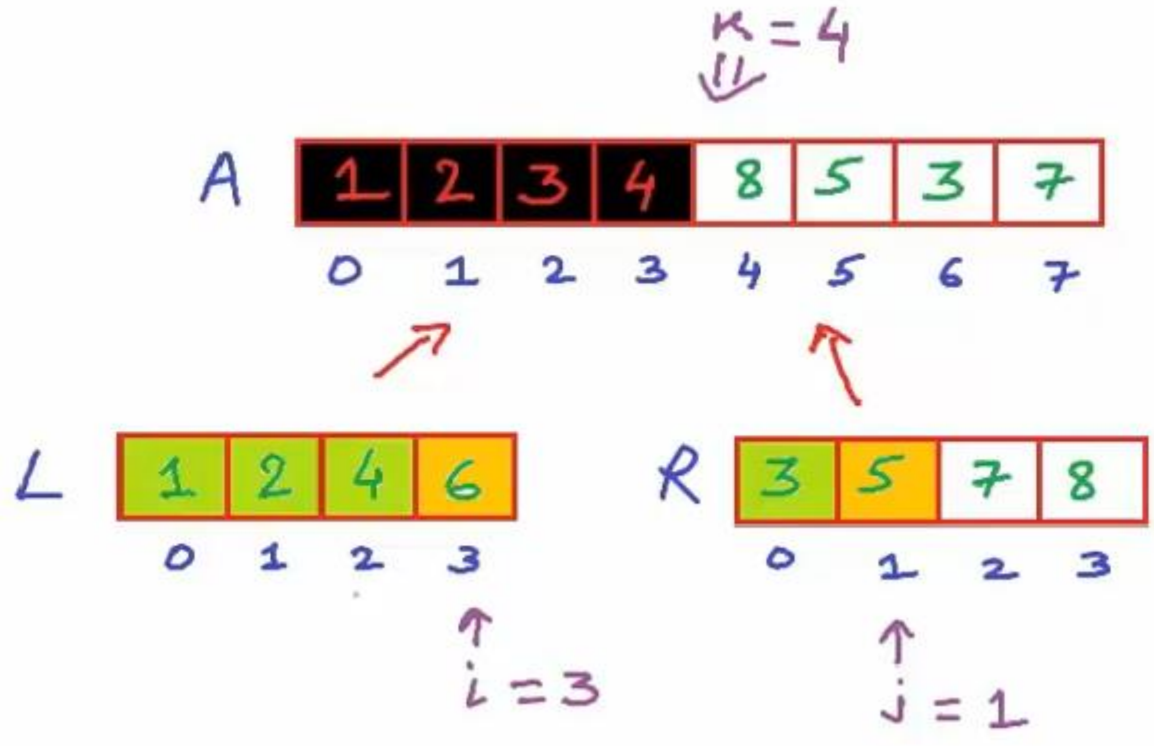
Merge Sort

- Next position between 4 and 3, 3 will go, and increment j and k.



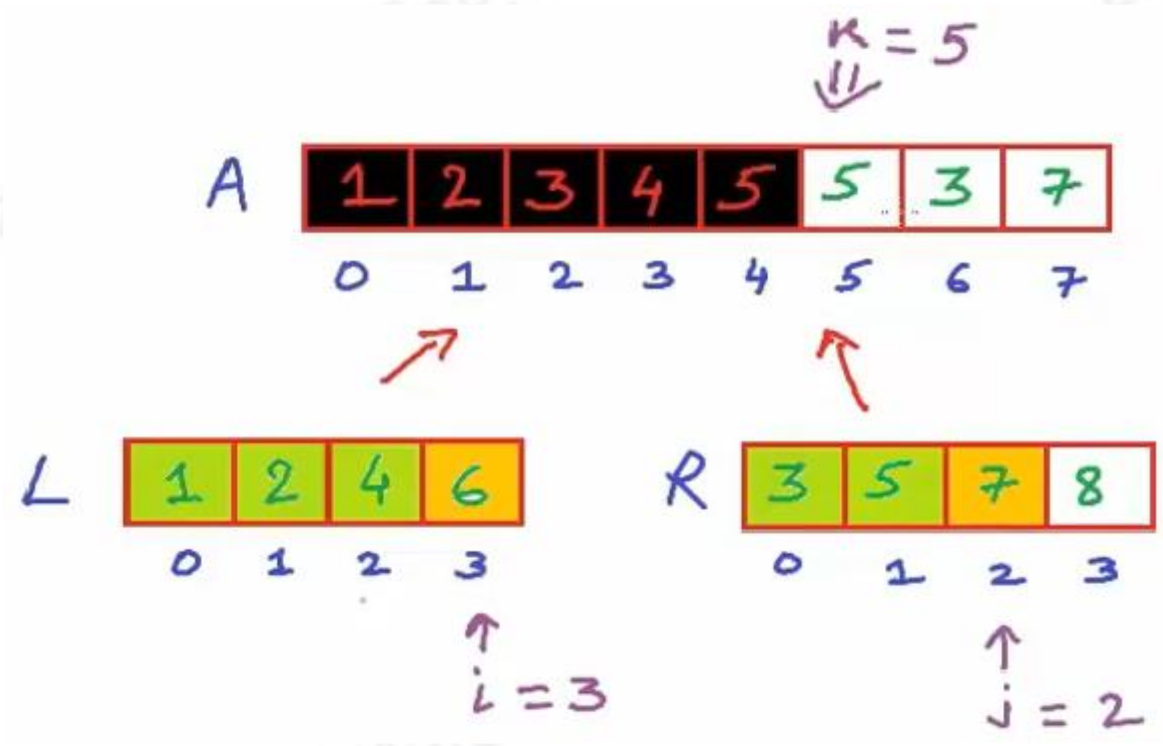
Merge Sort

- Next position between 4 and 5.



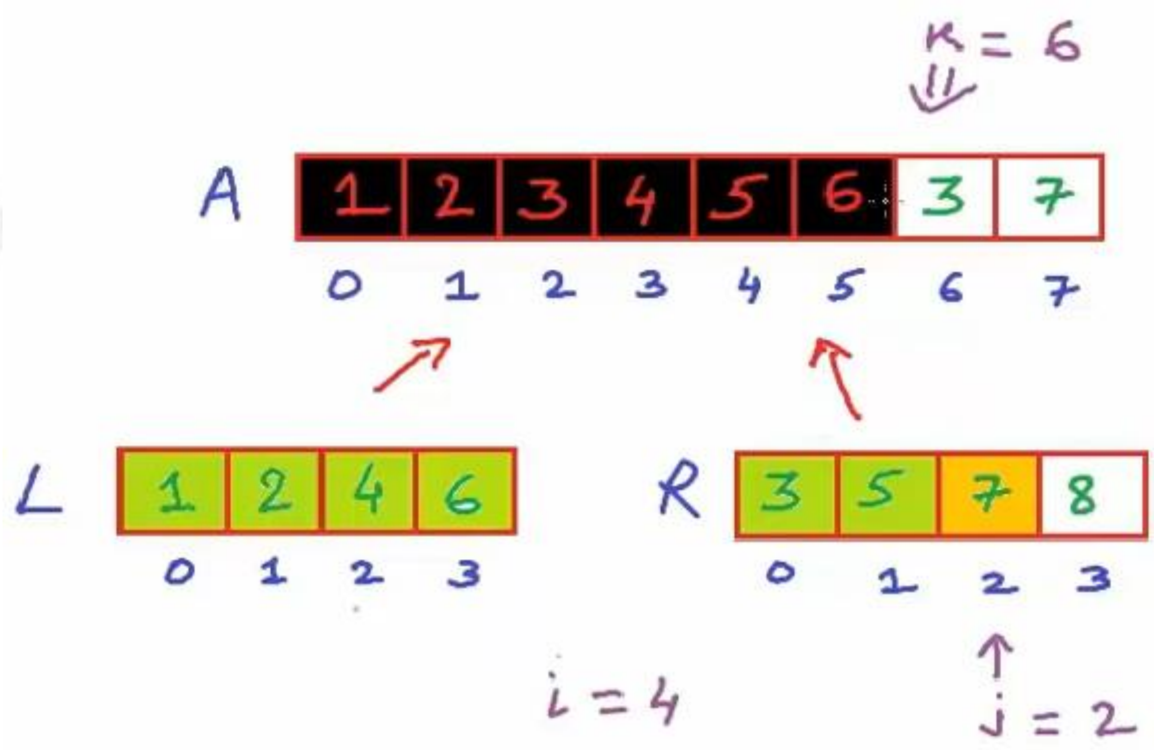
Merge Sort

- Next position between 5 and 6.



Merge Sort

- Next position between 6 and 7.



Merge Sort

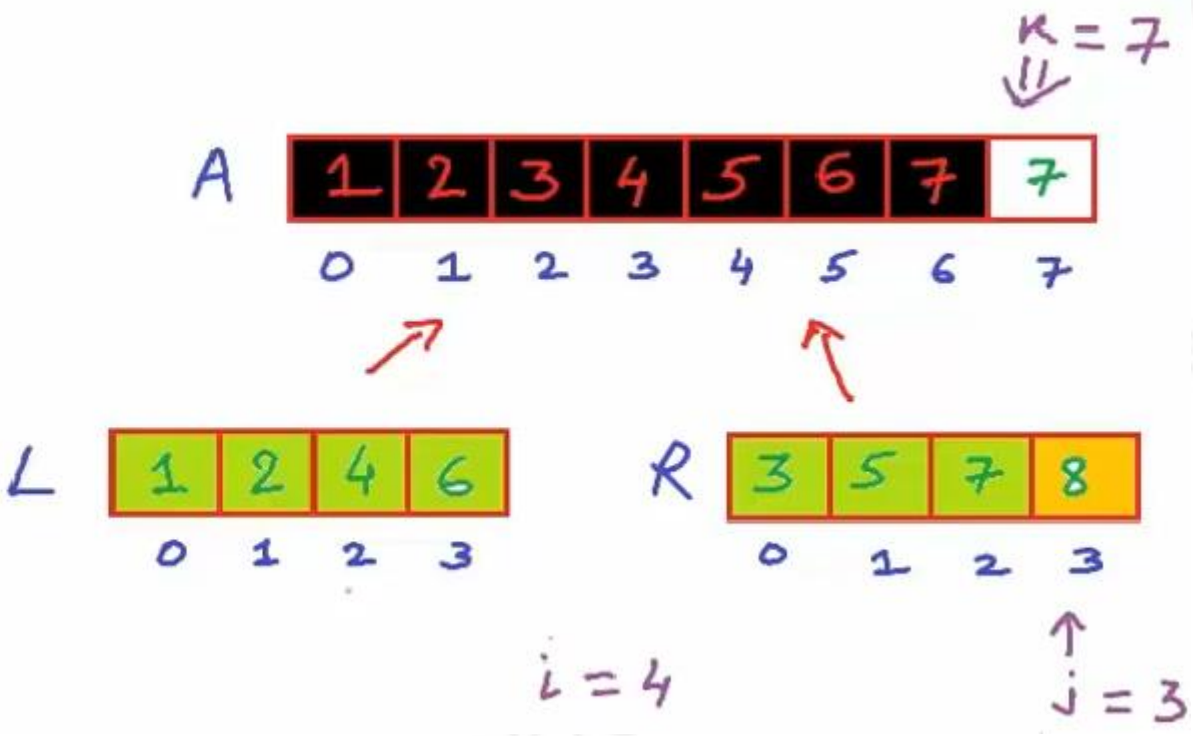
- After 6 has picked, we are done with all the elements in L, i is equal to 4 now, which is not a valid index.
- This is expected, one of the arrays L or R will exhaust first.
- In this case, we need to pick all the elements from the other array and fill the rest of positions in A.
- After the while loop, we can write the following (in the next slide).

Merge Sort

```
Merge(L, R, A)
{
  nL ← length(L)
  nR ← length(R)
  i ← j ← k ← 0
  while(i < nL && j < nR)
  {
    if( L[i] <= R[j])
    {
      A[k] ← L[i]; i ← i+1
    }
    else
    {
      A[k] ← R[j]; j ← j+1
    }
    k ← k+1
  }
  while (i < nL)
  { A[k] ← L[i]; i ← i+1; k ← k+1;
  }
  while(j < nR)
  { A[k] ← R[j]; j ← j+1; k ← k+1;
  }
}
```

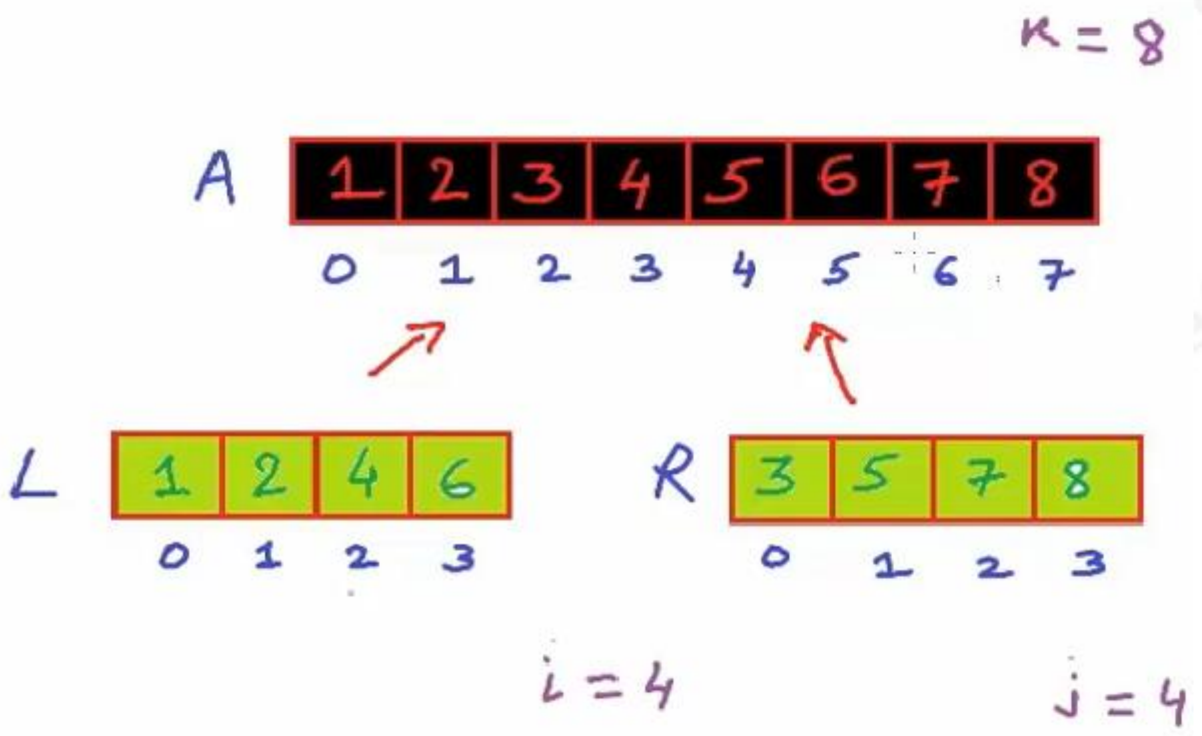
Merge Sort

- Now, we can fill up the remaining positions.



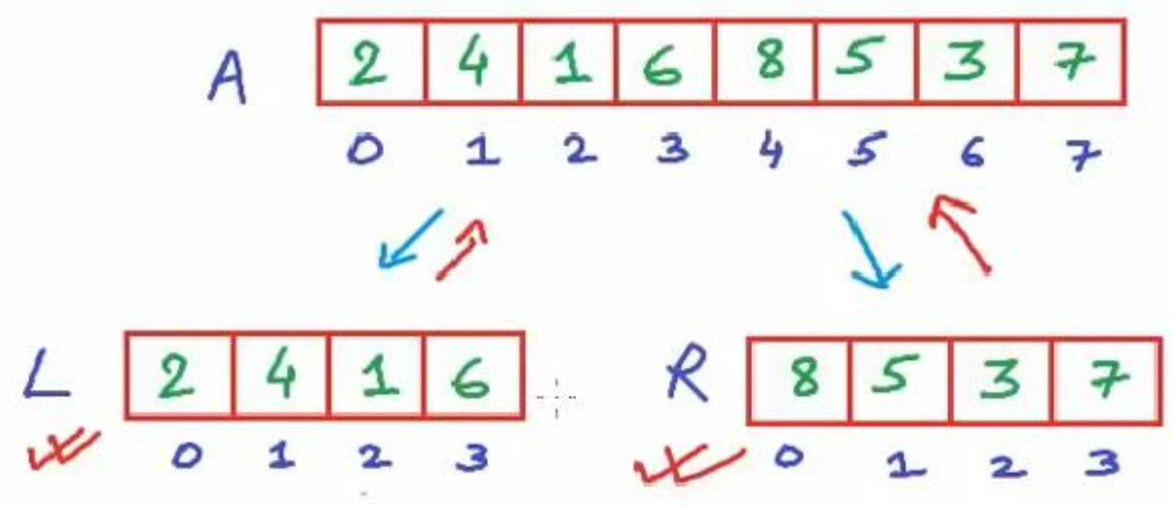
Merge Sort

- Finally, we will have a sorted arrangement in A.



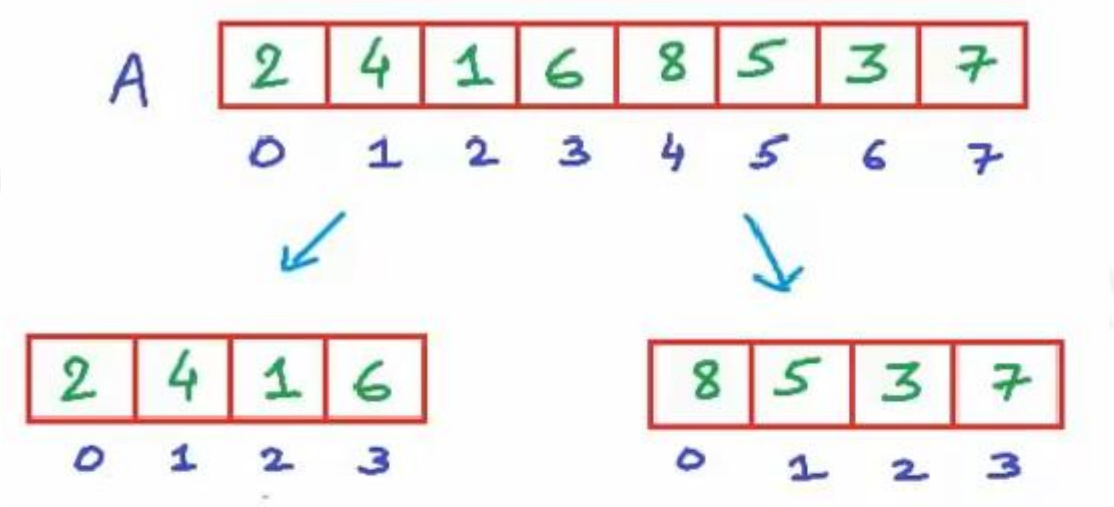
Merge Sort

- Going back where we had started, in the beginning we had supposed that if the two subarrays are sorted, we can merge them back into the original array.



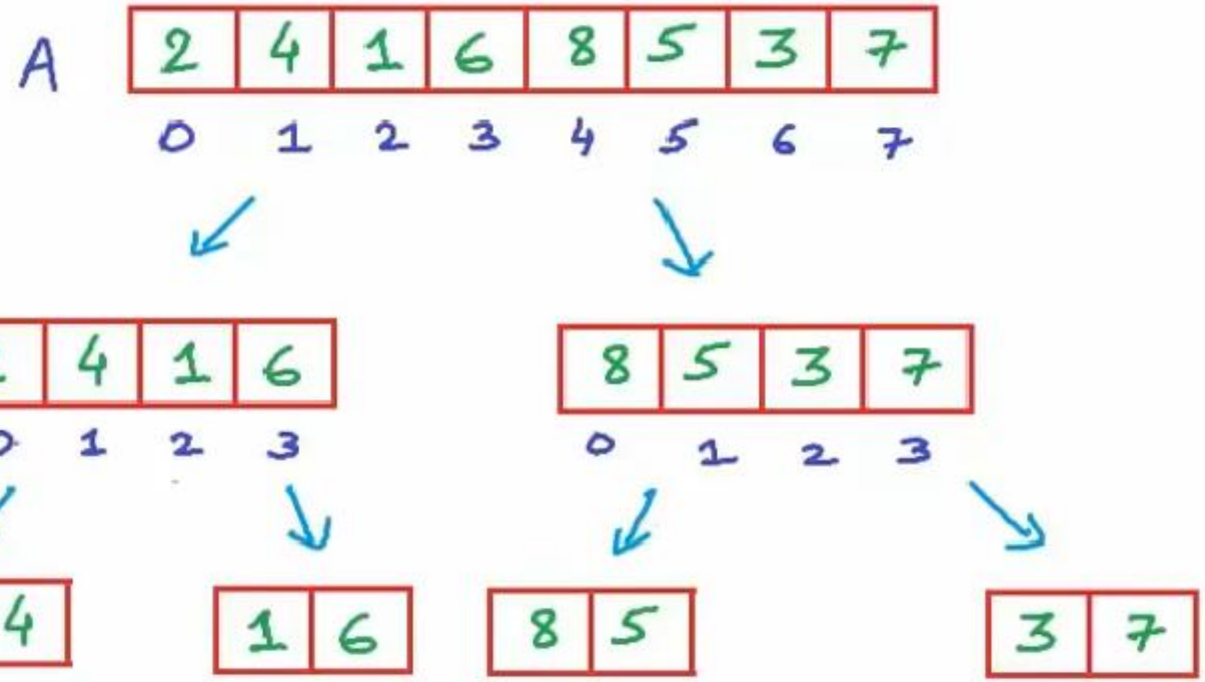
Merge Sort

- We need to have a logic to sort the two subarrays.



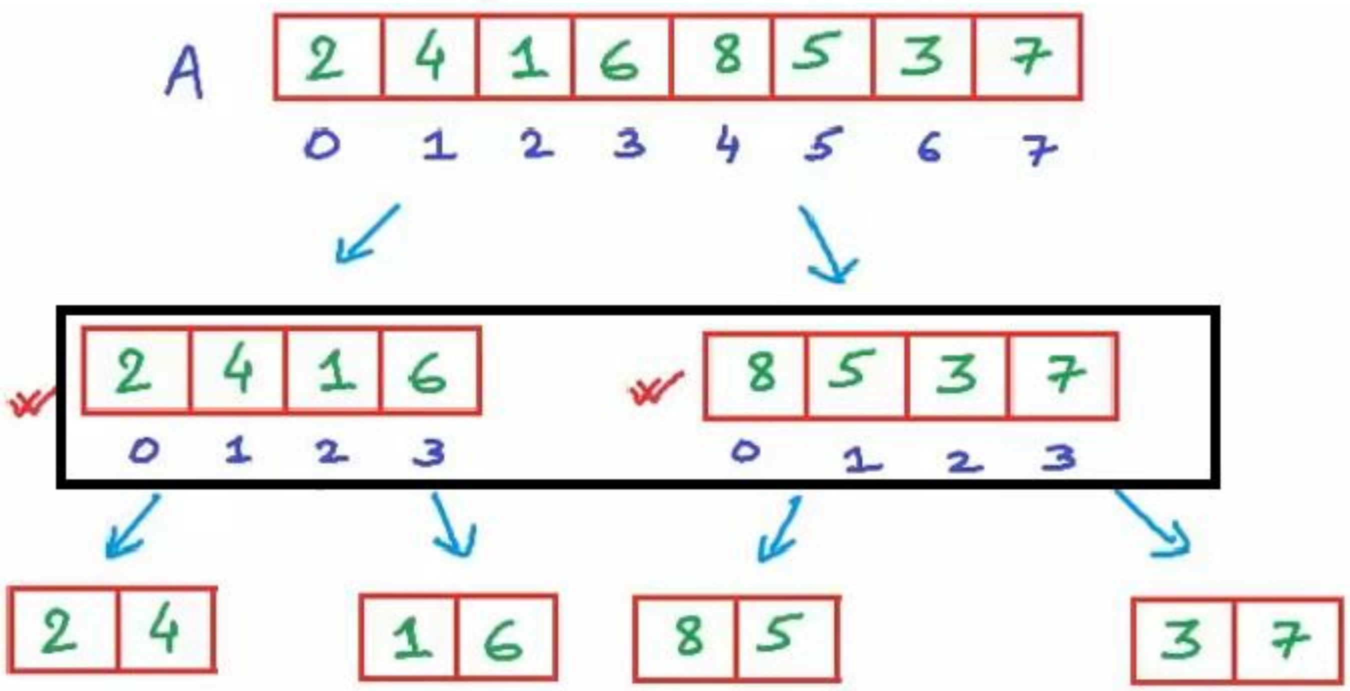
Merge Sort

- The logic is that we can break these subarrays even further.



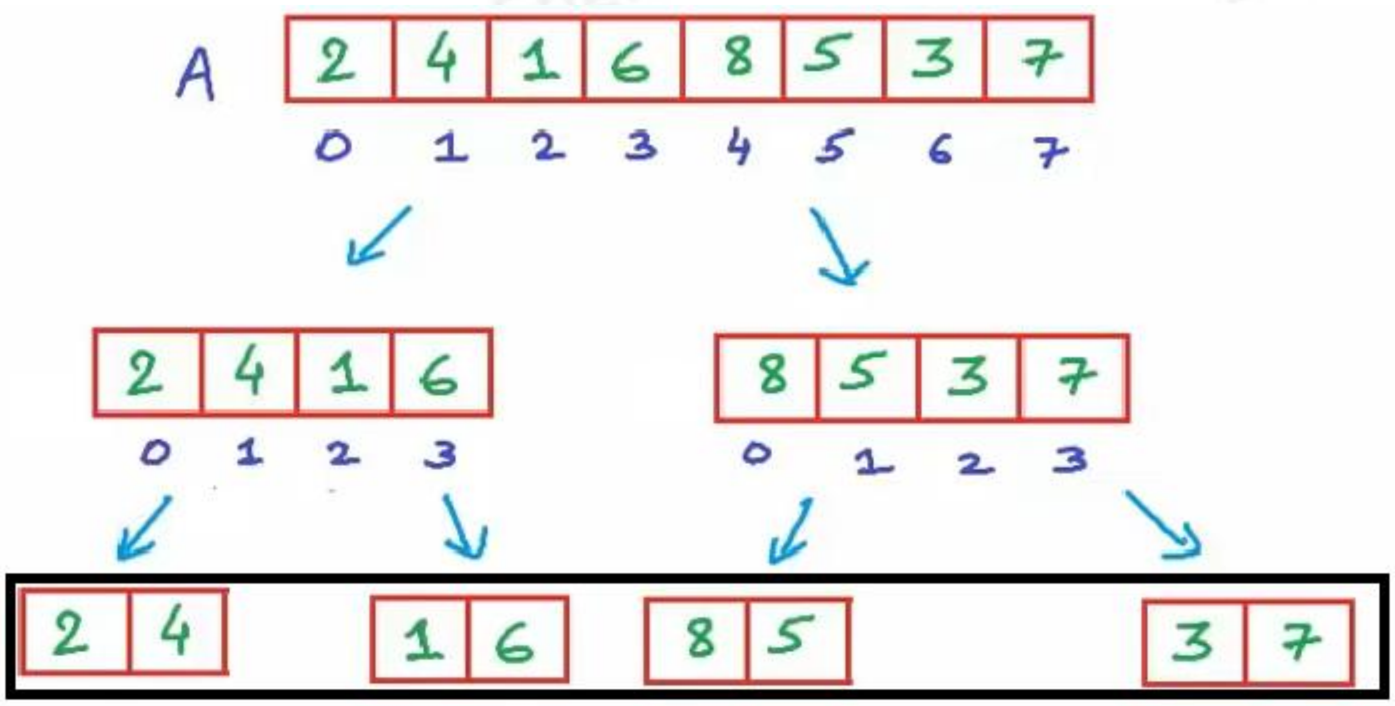
Merge Sort

- The solution of the first two subarrays can be constructed after we sort the new subarrays by merging them back.



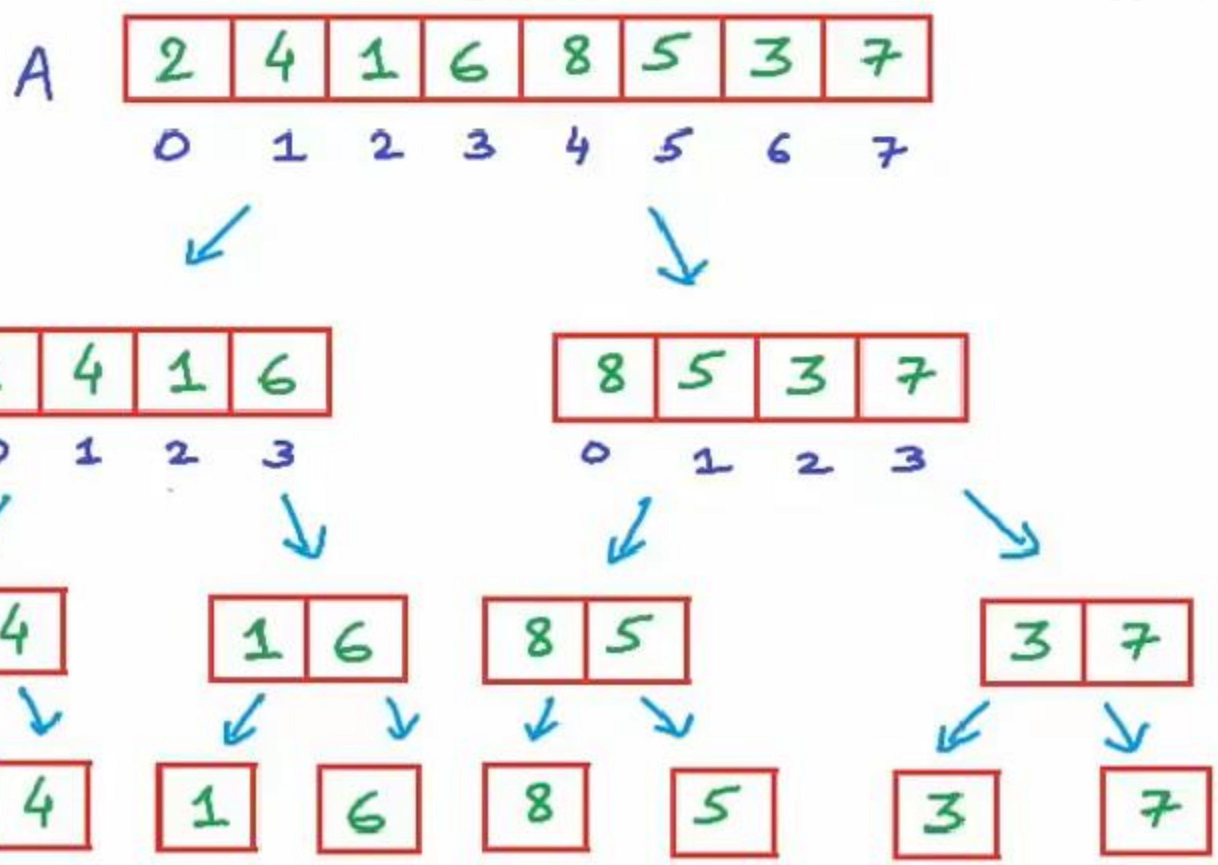
Merge Sort

- Once again, we have these four subarrays of two elements each.



Merge Sort

- These four subarrays can be also divided.



Merge Sort

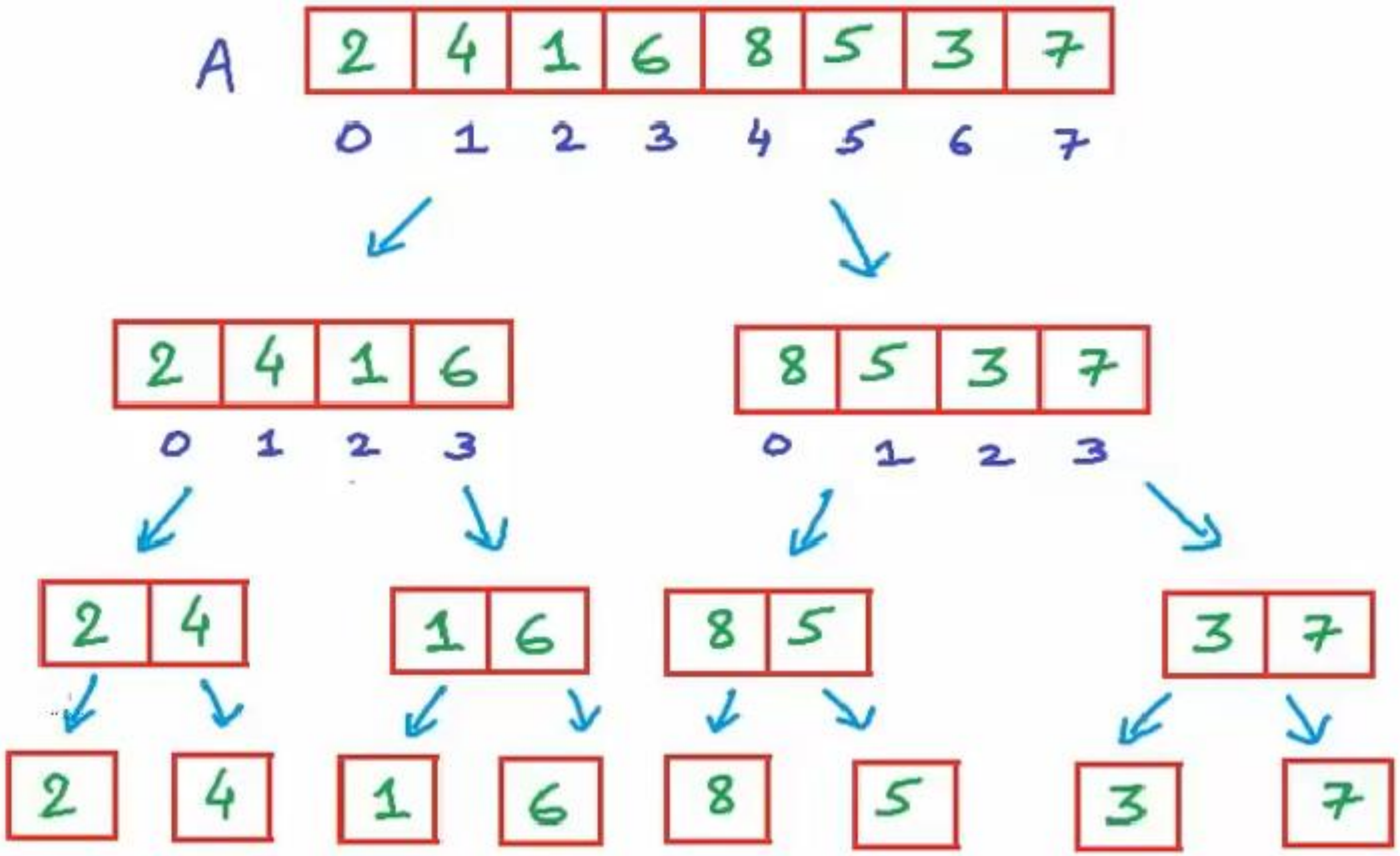
- What we are basically doing here is that we are reducing a problem into subproblems in a recursive or self similar manner.
- At any step, once we get a solution for the subproblems, we can construct the solution for the actual problem.
- For example, if we have two sorted sub lists, we can construct the parent list also.

Merge Sort

- We can go on reducing a subarray only if we have more than one element in the array.
- Once we reach a stage where we have only one element in an array, then we cannot reduce that subarray any further.
- An array with only one element is always sorted, we don't need to do anything to sort it.

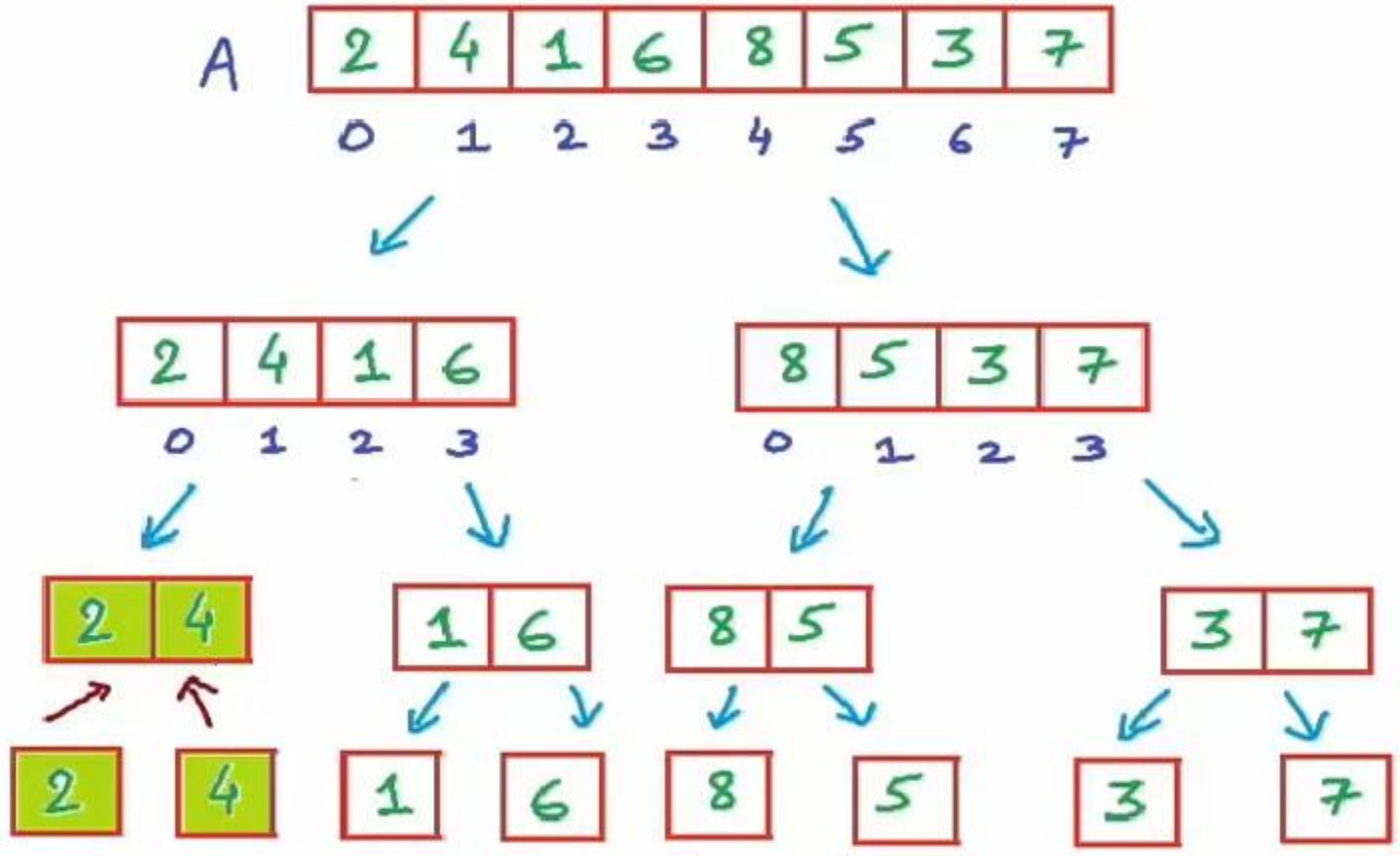
Merge Sort

- Now, at this stage we can start combining back or merging the subarrays.



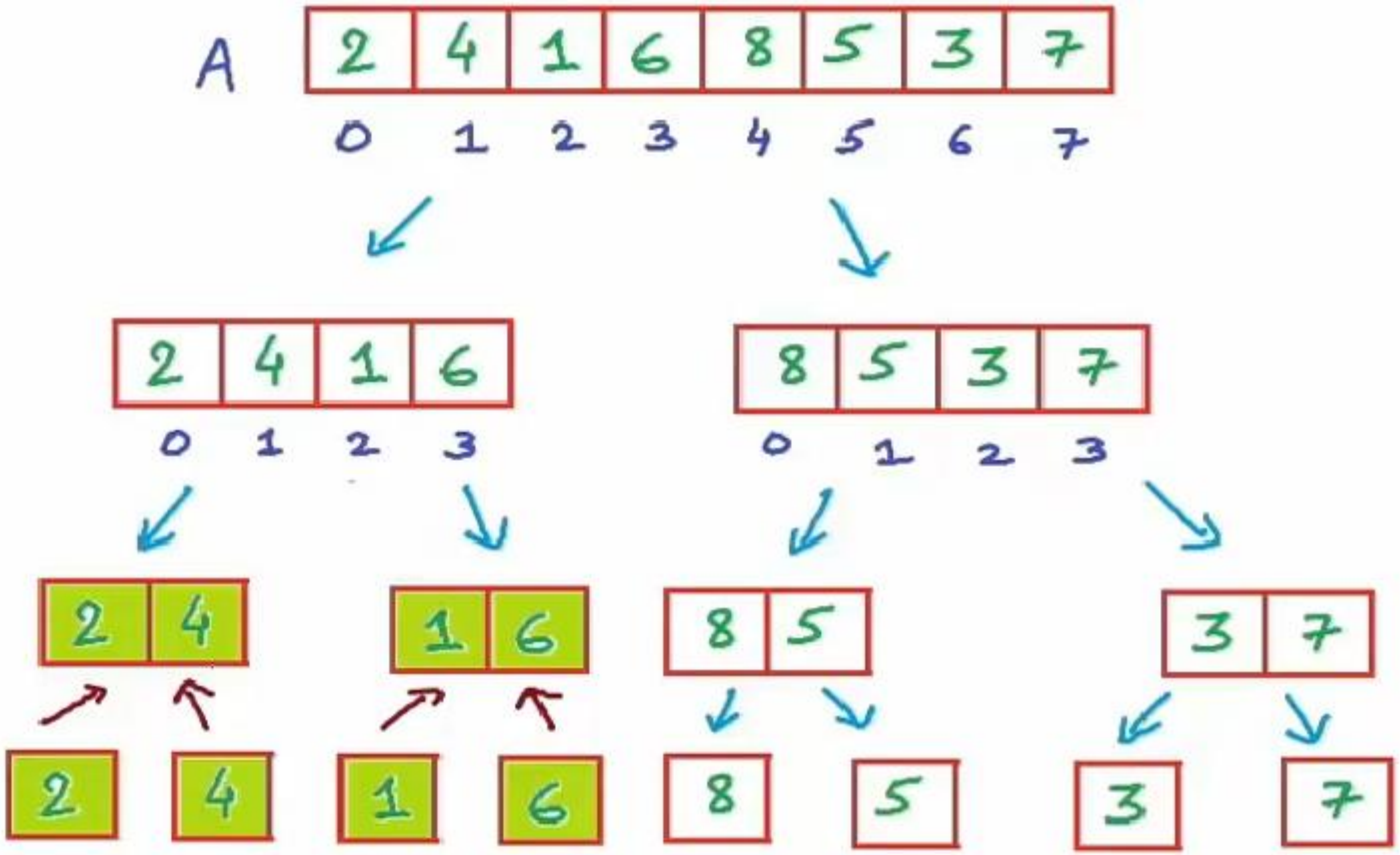
Merge Sort

- We will depict the cells in sorted subarrays in Green, we have already learned the merge logic.



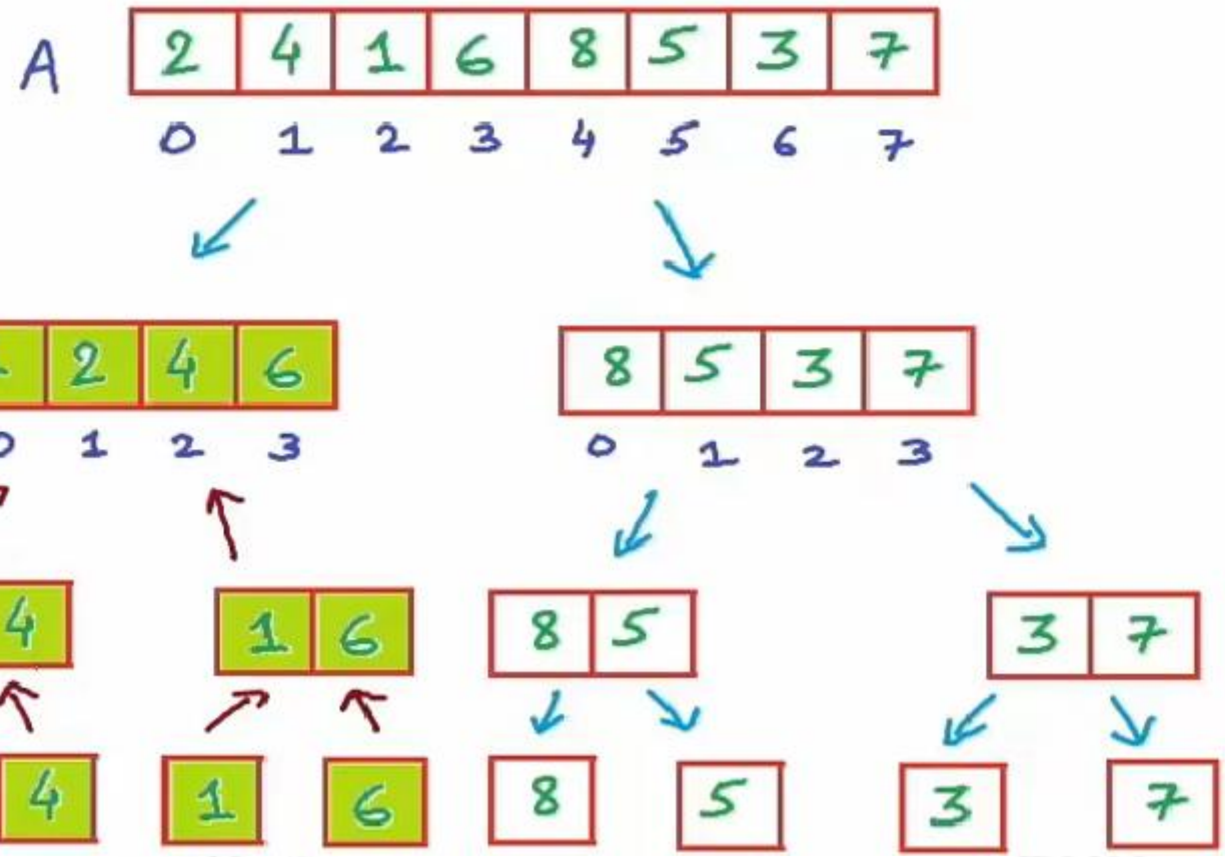
Merge Sort

- We can merge the next subarrays.



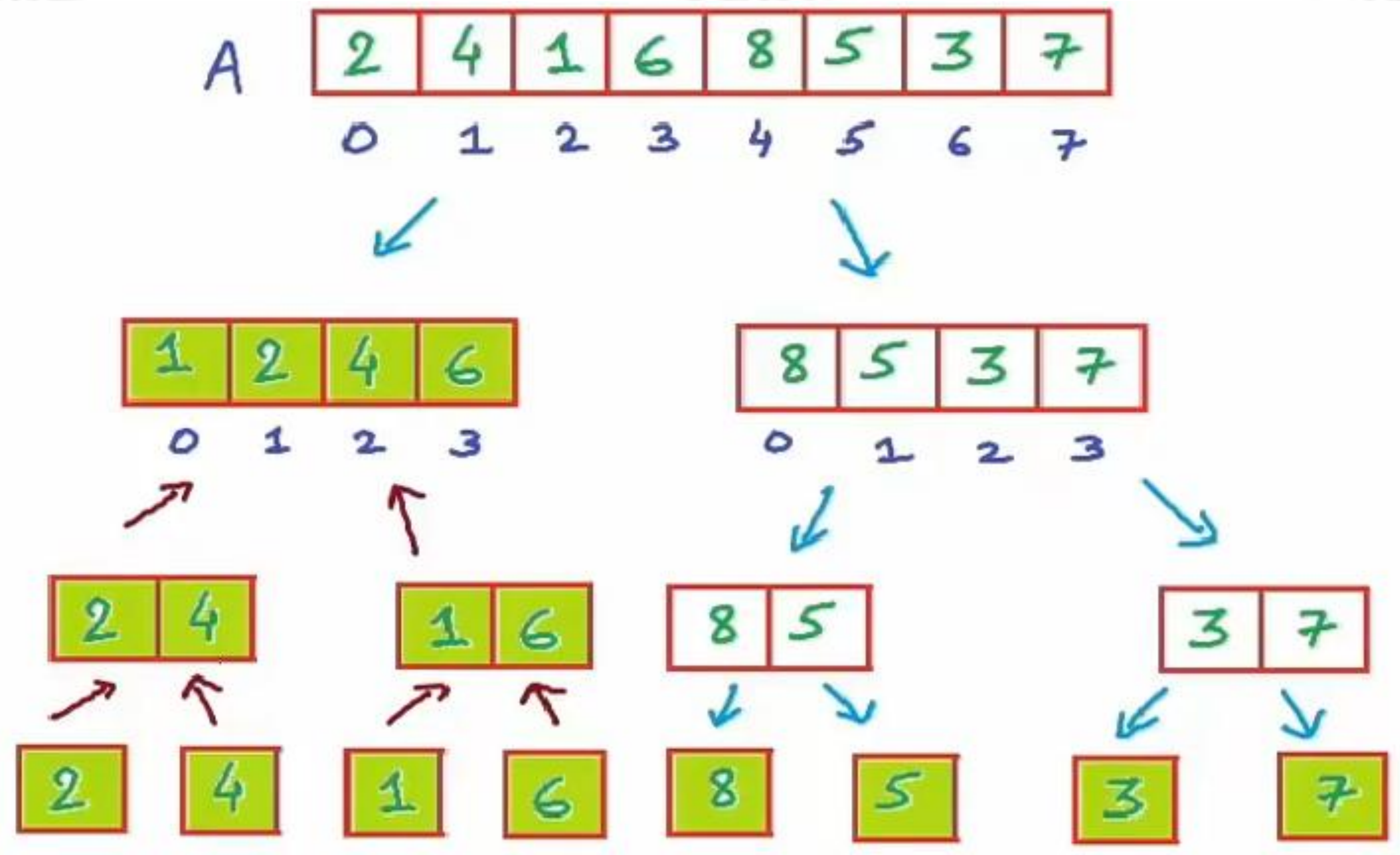
Merge Sort

- We can merge the next subarrays.



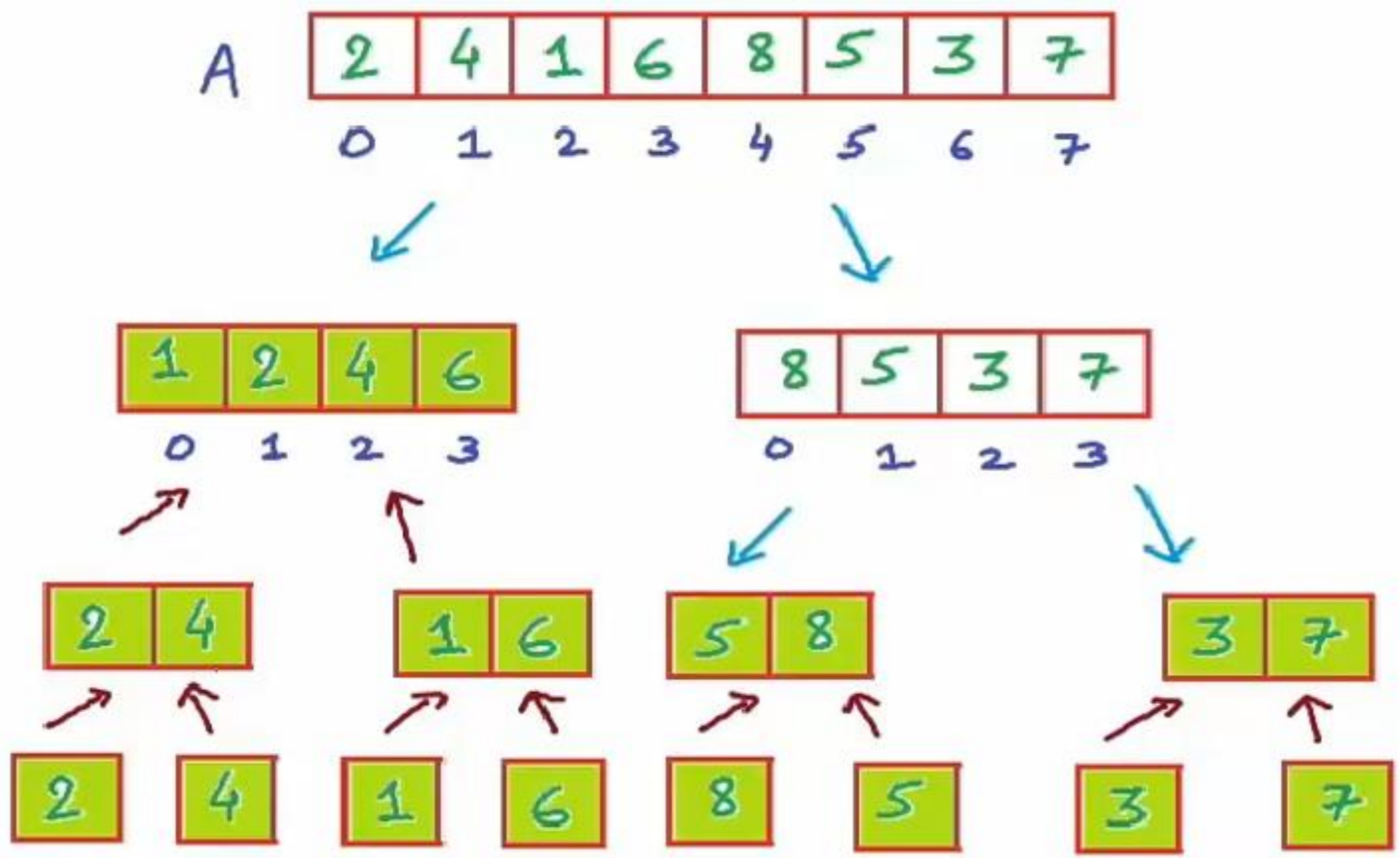
Merge Sort

- All the subarrays with only one elements is already sorted.



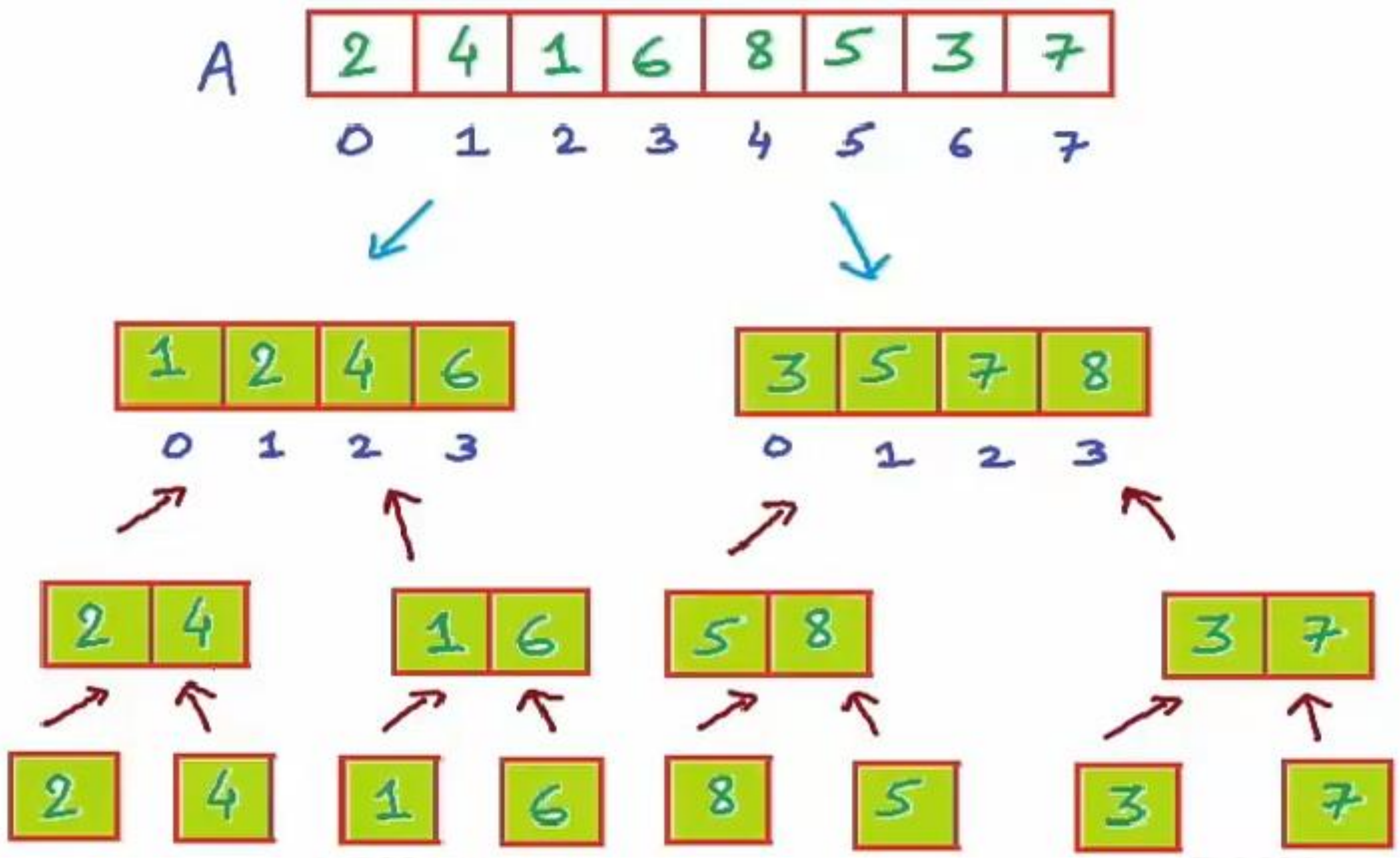
Merge Sort

- We can start merging them back.



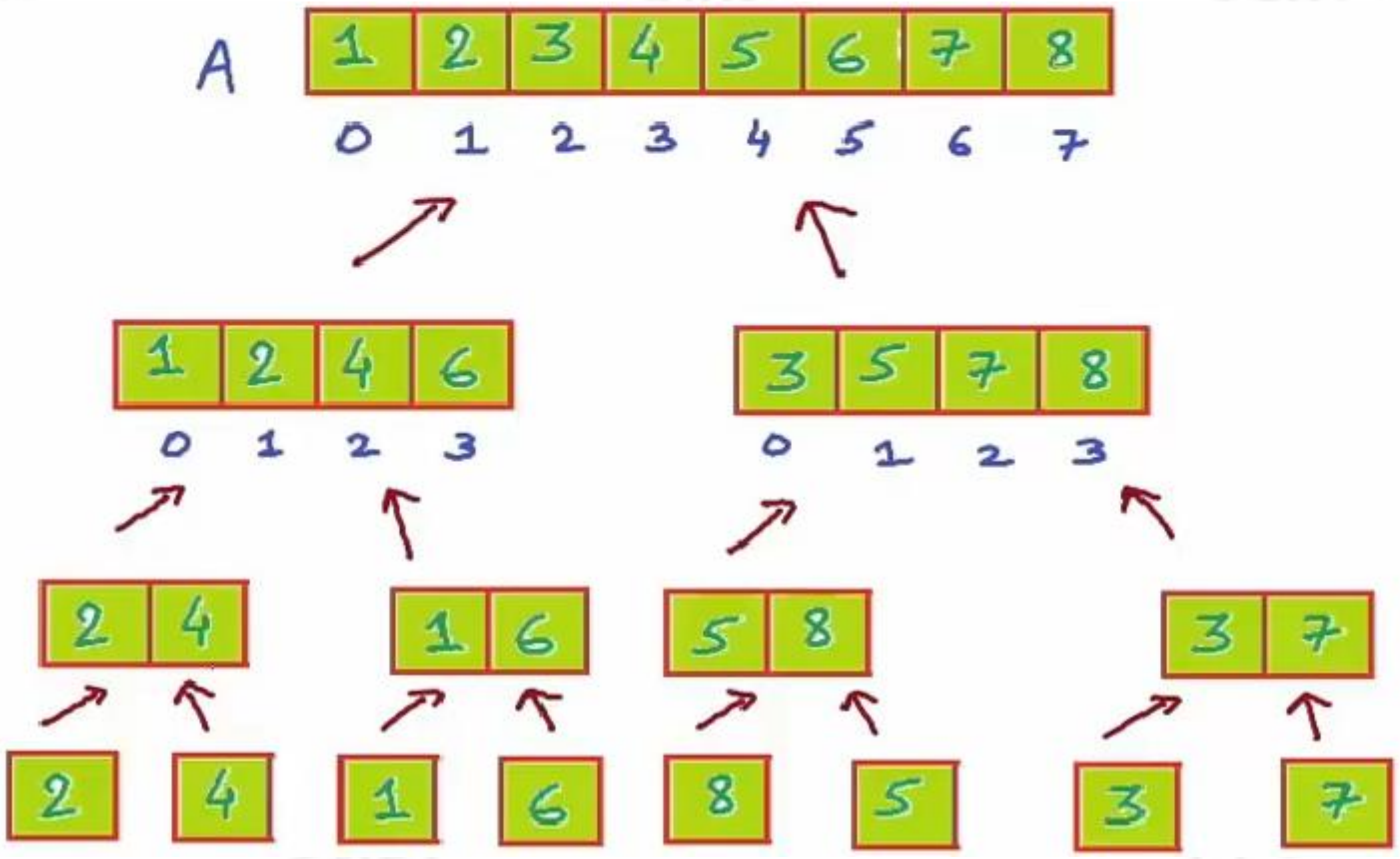
Merge Sort

- We can merge the next subarrays.



Merge Sort

- Finally, the last two sorted subarrays can be merged back to the original array A.



Merge Sort

- Now, we will write the pseudocode for this algorithm.
- We will write a function named **MergeSort** that will take an array **A** as an argument.

Merge Sort

```
Mergesort (A)
{ n ← length(A)
  if (n < 2) return
  mid ← n/2
  left ← array of size(mid)
  right ← array of size(n-mid)
  for i ← 0 to mid-1
      left[i] ← A[i]
  for i ← mid to n-1
      right[i-mid] ← A[i]
  Mergesort(left)
  Mergesort(right)
} Merge(left, right, A)
```

Merge Sort

- It is important to visualize how this recursion will execute, we will take an example in the class.

```
               Mergesort (A)
               { n ← length(A)
base      ←      if (n < 2) return
Condition        mid ← n/2
                 left ← array of size(mid)
                 right ← array of size(n-mid)
                 for i ← 0 to mid-1
recursive            left[i] ← A[i]
call             for i ← mid to n-1
                     right[i-mid] ← A[i]
  ⇒              Mergesort (left)
  ⇒              Mergesort (right)
               } Merge(left, right, A)
```

Any Questions???…