Recursion 2

In this module, we are going to understand how to solve different kinds of problems using recursion.

Recursion With Strings

Recursion in strings is not a very different logic, it is the same as we apply in arrays, in fact it becomes more easy to pass a complete new string using substring method.

Problem Statement - Replace pi

You are given a string of size n containing characters a-z. The task is to write a recursive function to replace all occurrences of pi with 3.14 in the given string and print the modified string.

Approach

1. **Step of the trivial case:** In this step, we will prove the desired statement for a base case like **size of string = 0** or **1**.

2. **Small calculation and recursive part interlinked:** In this step, you will first check character at 0th index and character at 1st index of the string.
    - If it comes out to be ‘p’ and ‘i’ then we make a recursive call passing the string from index 2. And thereafter “3.14” needs to be concatenated with the recursive answer and this updated string will be the answer.

- Else we will just make a recursive call passing the string from index 1. Thereafter the character at 0th index needs to be concatenated with the recursive answer and return the same.

Binary Search Using Recursion

In a nutshell, this search algorithm takes advantage of a collection of elements that are already sorted by ignoring half of the elements after just one comparison.

You are given a target element X and a sorted array. You need to check if X is present in the array. Consider the algorithm given below:

- Compare X with the middle element in the array.
- **If** X is the same as the middle element, we return the index of the middle element.
- **Else if** X is greater than the mid element, then X can only lie in the right (greater) half subarray after the mid element. Thus, we apply the algorithm, recursively, for the right half. `#Condition1`
- **Else if** X is smaller, the target X must lie in the left (lower) half. So we apply the algorithm, recursively, for the left half. `#Condition2`

```
// Returns the index of x in arr if present, else -1
public static int binary_search(arr, low, high, x){
    if (high >= low){ // Check base case
        mid = (high + low) / 2;
        if (arr[mid] == x) //If element is at the middle itself
            return mid;
        else if (arr[mid] > x)   //Condition 2
            return binary_search(arr, low, mid - 1, x);
        else //Condition 1
            return binary_search(arr, mid + 1, high, x);
    }
    else
        return -1;  // Element is not present in the array
```

```
}
```

Sorting Techniques Using Recursion - Merge Sort

Merge sort requires dividing a given list into equal halves until it can no longer be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

- **Step 1 –** If it is only one element in the list it is already sorted, return.
- **Step 2 –** Divide the list recursively into two halves until it can't be divided further.
- **Step 3 –** Merge the smaller lists into a new list in sorted order.

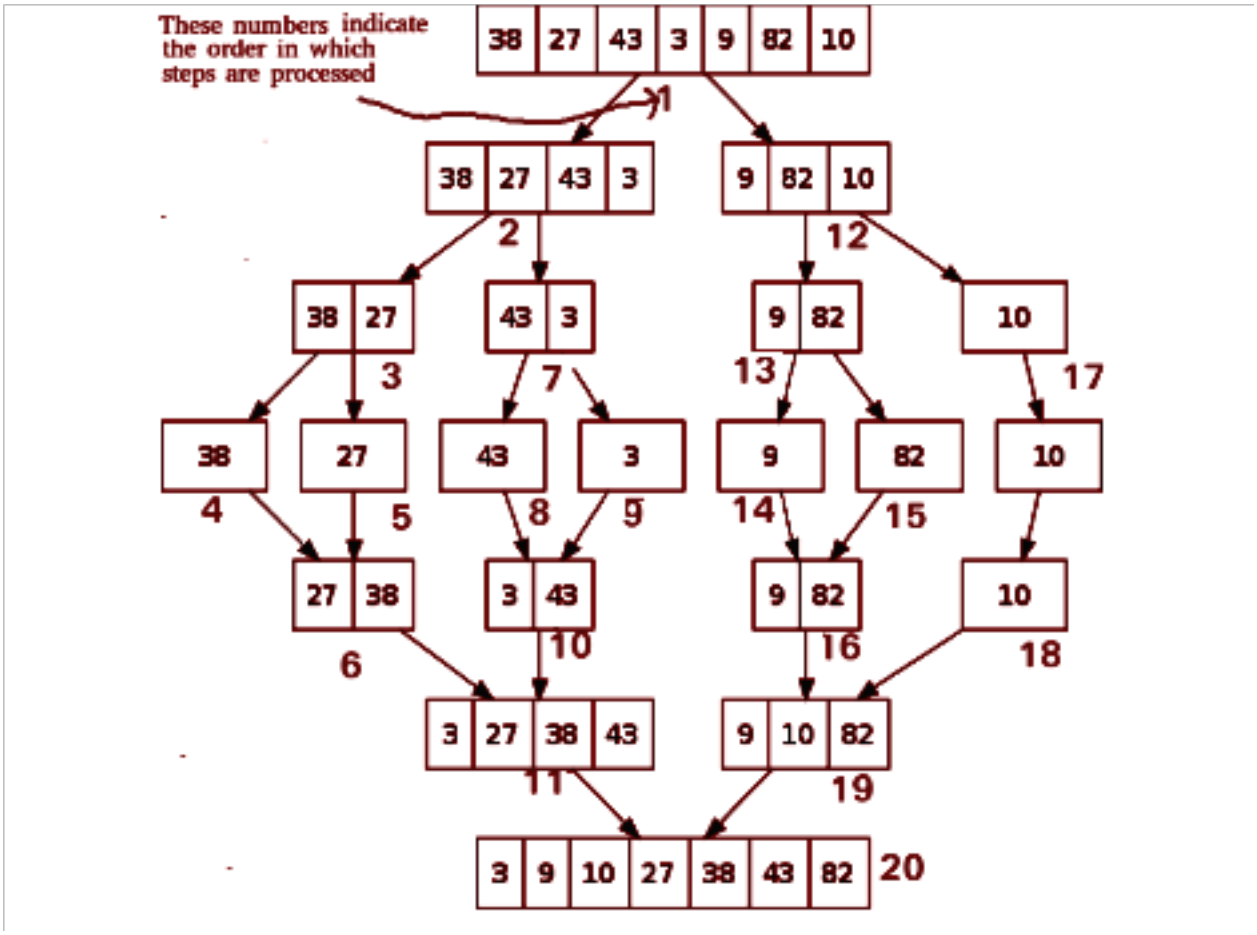
It has just one disadvantage and that is it's **not an in-place sorting** technique i.e. it creates a copy of the array and then works on that copy.

Pseudo-Code

```
public static void mergeSort(arr[], l,  r){
   if (r > l){
     1. Find the middle point to divide the array into two halves:
             middle m = (l+r)/2
     2. Call mergeSort for the first half:
             Call mergeSort(arr, l, m)
     3. Call mergeSort for the second half:
             Call mergeSort(arr, m+1, r)
     4. Merge the two halves sorted in step 2 and 3:
             Call merge(arr, l, m, r)
   }
}
```

The following diagram shows the complete merge sort process for an example array `[38,27,43,3,9,82,10]`. If we take a closer look at the diagram, we can see

that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



Quick Sort

Quick-sort is based on the **divide-and-conquer approach**. It works along the lines of choosing one element as a pivot element and partitioning the array around it such that:

- The left side of the pivot contains all the elements that are less than the pivot element