Recursion 2

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

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
def binary_search(arr, low, high, x):
    if high >= low:# Check base case
        mid = (high + low) // 2
        if arr[mid] == x:# If element is present at the middle itself
            return mid
        elif 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.

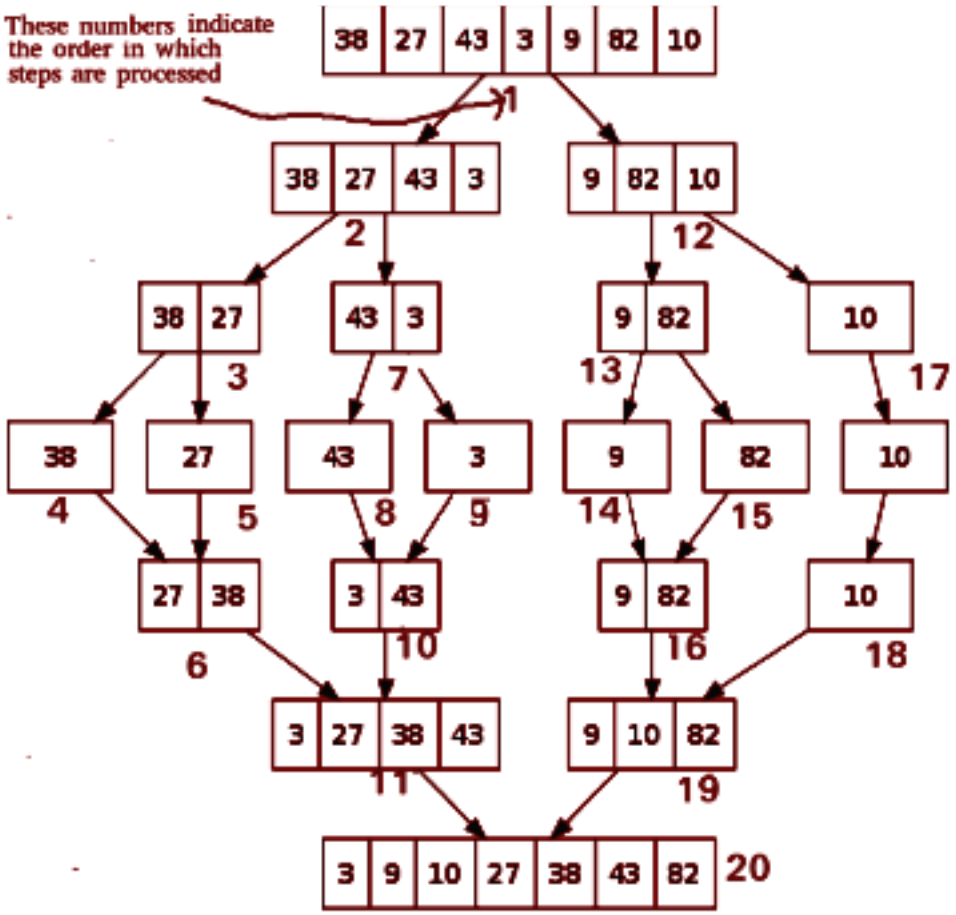
- **Step 1 –** If it is only one element in the list it is already sorted, return.
- **Step 2 –** Divide the list recursively into two halves until it can't be divided further.
- **Step 3 –** Merge the smaller lists into a new list in sorted order.

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

```
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
- The right side contains all elements greater than the pivot.

Algorithm for Quick Sort:

Based on the **Divide-and-Conquer** approach, the quicksort algorithm can be explained as:

- **Divide:** The array is divided into subparts taking pivot as the partitioning point. The elements smaller than the pivot are placed to the left of the pivot and the elements greater than the pivot are placed to the right side.
- **Conquer:** The left and right sub-parts are again partitioned using the by selecting pivot elements for them. This can be achieved by recursively passing the subparts into the algorithm.
- **Combine:** This part does not play a significant role in quicksort. The array is already sorted at the end of the conquer step.

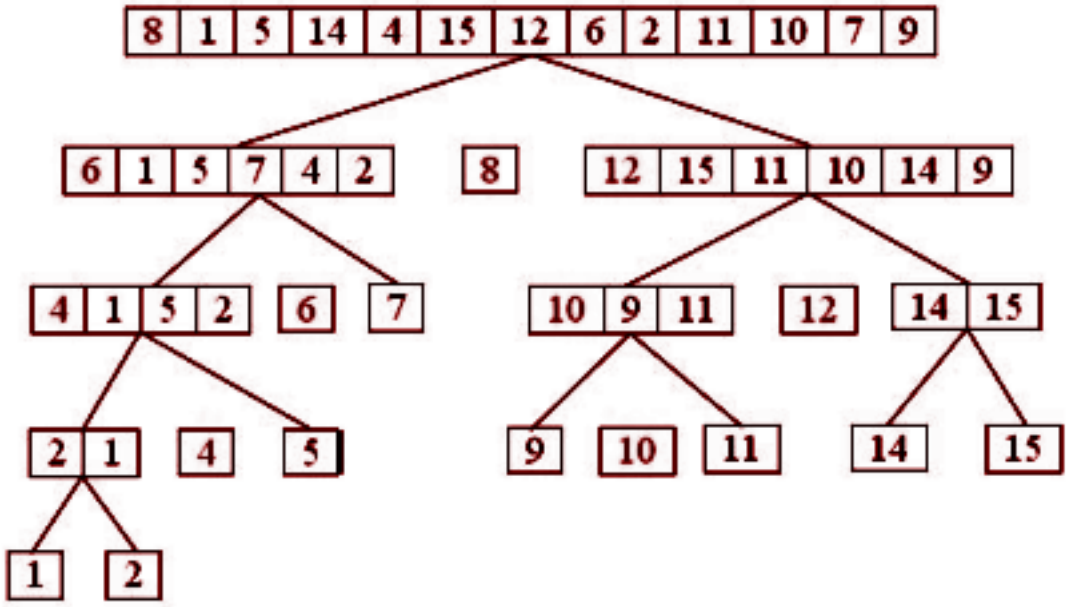
The advantage of quicksort over merge sort is that it does not require any extra space to sort the given array, that it is an in-place sorting technique.

There are many ways to pick a pivot element:

1. Always pick the first element as the pivot.
2. Always pick the last element as the pivot.
3. Pick a random element as the pivot.
4. Pick the middle element as the pivot.

Given below is a pictorial representation of how this algorithm sorts the given array:

```
[8,1,5,14,4,15,12,6,2,11,10,7,9]
```



- In **step 1**, 8 is taken as the pivot.
- In **step 2**, 6 and 12 are taken as pivots.
- In **step 3**, 4, and 10 are taken as pivots.
- We keep dividing the list about pivots till there are only 2 elements left in a sublist.

Problem Statement - Tower Of Hanoi

Tower of Hanoi is a **mathematical puzzle** where we have **3** rods and **N** disks. The objective of the puzzle is to move all disks from **source rod** to **destination rod** using a **third rod (say auxiliary)**. The rules are :

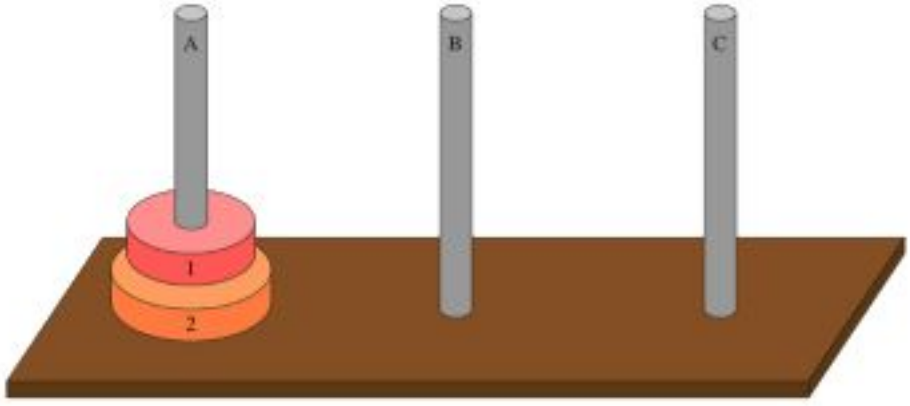
- Only one disk can be moved at a time.
- A disk can be moved only if it is on the top of a rod.
- No disk can be placed on the top of a smaller disk.

Print the steps required to move **N** disks from source rod to destination rod.
Source Rod is named as **'A'**, the destination rod as **'B'**, and the auxiliary rod as **'C'**.

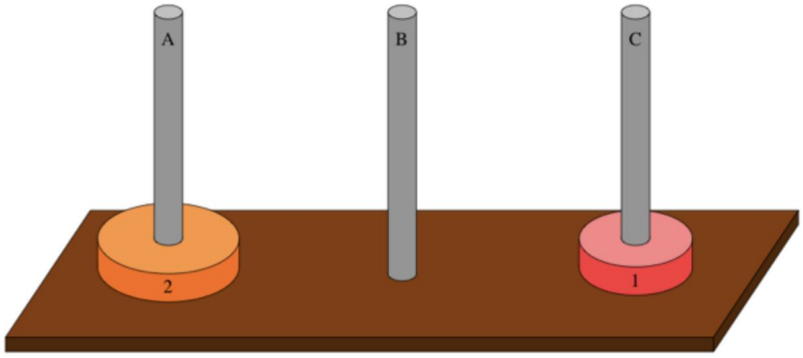
Let's see how to solve the problem recursively. We'll start with a really easy case **N=1**. We just need to move one disk from source to destination.

- You can always move **disk 1** from peg **A** to peg **B** because you know that any disks below it must be larger.
- There's nothing special about pegs **A** and **B**. You can move disk 1 from peg **B** to peg **C** if you like, or from peg **C** to peg **A**, or from any peg to any peg.
- Solving the Towers of Hanoi problem with one disk is trivial as it requires moving only the one disk one time.

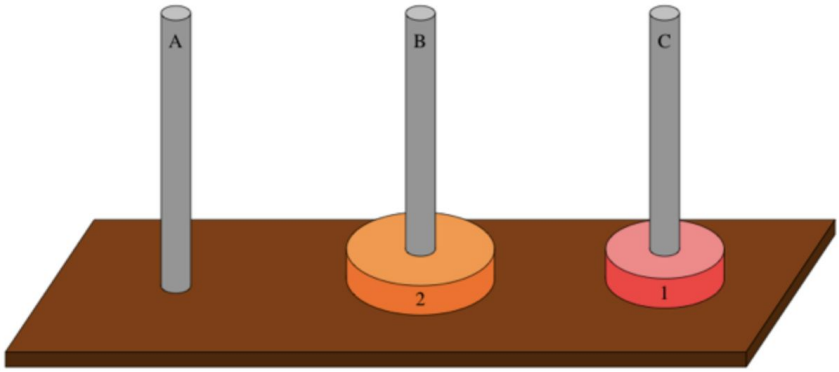
Now consider the case **N=2**. Here's what it looks like at the start:



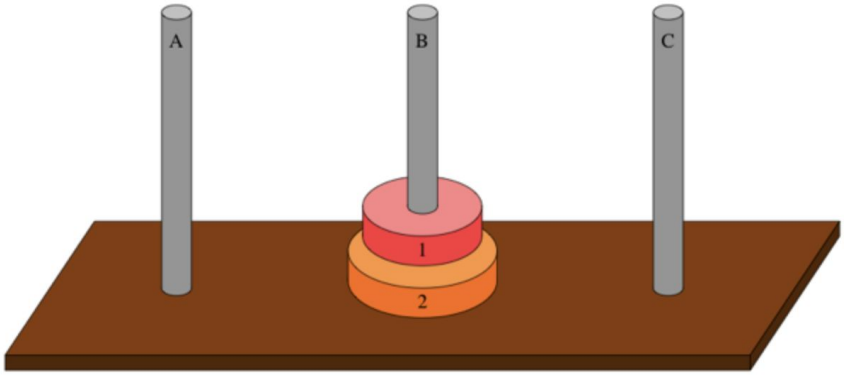
First, move **disk 1** from peg **A** to peg **C**:



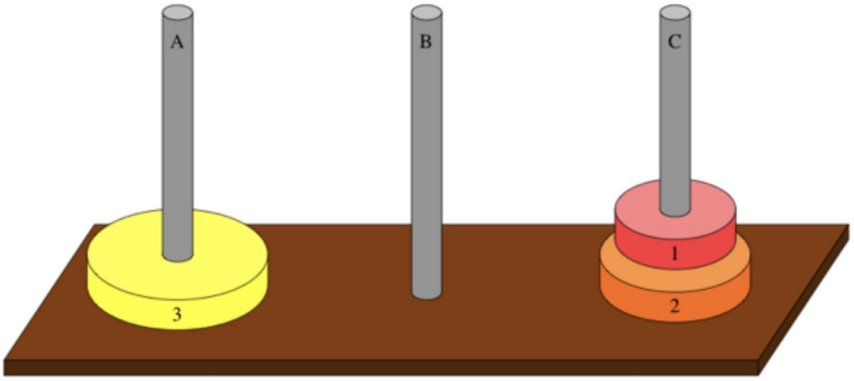
Notice that we're using peg **C** as a spare peg, a place to put **disk 1** so that we can get at **disk 2**. Now that **disk 2**—the bottommost disk—is exposed, move it to peg **B**:



Finally, move **disk 1** from peg **C** to peg **B**:

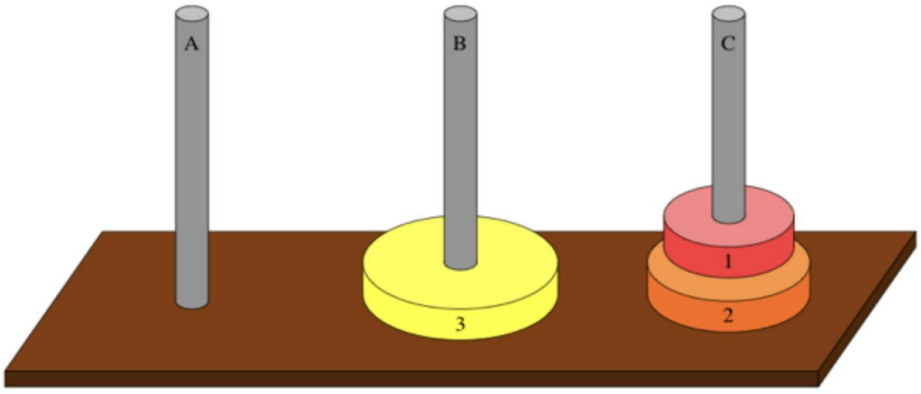


Now let us solve this problem for **3** disks. You need to expose the bottom disk (**disk 3**) so that you could move it from peg **A** to peg **B**. To expose **disk 3**, you needed to move disks 1 and 2 from peg **A** to the spare peg, which is peg **C**:

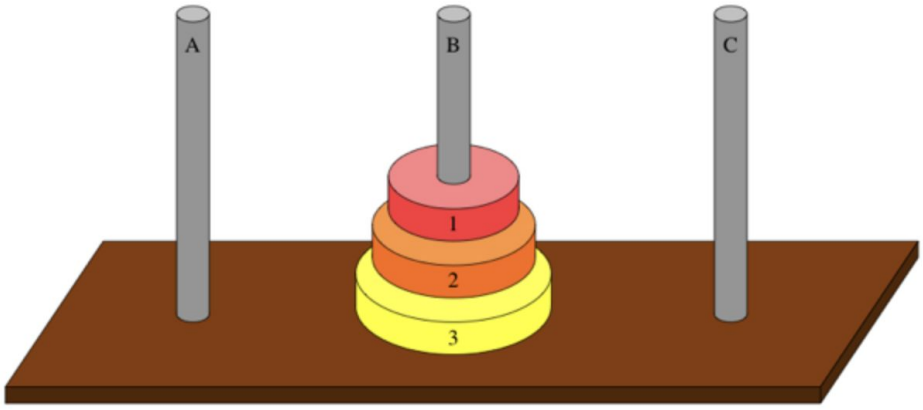


Wait a minute—it looks like two disks moved in one step, violating the first rule. But they did not move in one step. You agreed that you can move disks 1 and 2 from any peg to any peg, **using three steps.** The situation above represents what you have after three steps. (Move **disk 1** from peg **A** to peg **B**; move **disk 2** from peg **A** to peg **C**; move **disk 1** from peg **B** to peg **C**.)

More to the point, by moving disks 1 and 2 from peg **A** to peg **C**, you have recursively solved a subproblem: move disk **1 through n-1** (remember that n = 3) from peg **A** to peg **C**. Once you've solved this subproblem, you can move **disk 3** from peg **A** to peg **B**:



Now, to finish up, you need to recursively solve the subproblem of moving disks **1 through n-1**, from peg **C** to peg **B**. Again, you agreed that you can do so in three steps. (Move **disk 1** from peg **C** to peg **A**; move **disk 2** from peg **C** to peg **B**; move **disk 1** from peg **A** to peg **B**.) And you're done:



At this point, you might have picked up the pattern. The **algorithm** can be summarised as:

If **n == 1**, just move **disk 1**. Otherwise, when n>=2, solve the problem in three steps:

- Recursively solve the subproblem of moving disks **1 through n-1** from whichever peg they start on, to the spare peg.
- Move disk **N** from the peg it starts on, to the peg it's supposed to end up on.
- Recursively solve the subproblem of moving disks **1 through n-1**, from the spare peg to the peg they're supposed to end up on.

Python Code

```
# Recursive Python function to solve the Tower of Hanoi
def TowerOfHanoi(n,src, dest, aux):
    if n==1: #Base Case
        print("Move disk 1 from source",src,"to destination",dest)
        return
    TowerOfHanoi(n-1, src, aux, dest) #Recursive Call 1
    print ("Move disk",n,"from source",src,"to destination",dest)
    TowerOfHanoi(n-1, aux, dest, src) #Recursive Call 2

n = 4
TowerOfHanoi(n,'A','B','C')
# A, B, C are the name of rods
```

The output of this code will be:

```
Move disk 1 from source A to destination C
Move disk 2 from source A to destination B
Move disk 1 from source C to destination B
Move disk 3 from source A to destination C
Move disk 1 from source B to destination A
Move disk 2 from source B to destination C
Move disk 1 from source A to destination C
Move disk 4 from source A to destination B
Move disk 1 from source C to destination B
Move disk 2 from source C to destination A
Move disk 1 from source B to destination A
Move disk 3 from source C to destination B
Move disk 1 from source A to destination C
Move disk 2 from source A to destination B
Move disk 1 from source C to destination B
```