
# Python Programming
# Unit 5
# (KNC-302)

Prepared By

Abhishek Kesharwani

**Assistant Professor,UCER Naini,Allahabad**

# UNIT V

- Iterators & Recursion:
- Recursive Fibonacci
- Tower of Hanoi
- Search: Simple Search and Estimating Search Time
- Binary Search and Estimating Binary Search Time
- Sorting & Merging:
- Selection Sort,
- Merge List,
- Merge Sort
- Higher Order Sort

# Recursion

- Recursive Fibonacci
- Tower Of Hanoi

# Fibonacci series

In mathematical terms, the sequence Fn of Fibonacci numbers is defined by the recurrence relation

$$Fn = Fn\text{-}1 + Fn\text{-}2$$

with seed values $F0 = 0$ and $F1 = 1$.

- The Fibonacci numbers are the numbers in the following integer sequence.
- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ……..

# Fibonacci Number in Python

```
def Fibonacci(n):
    if n < 0:
        print("Incorrect input")
        # First Fibonacci number is 0
    elif n == 1:
        return 0
    # Second Fibonacci number is 1
    elif n == 2:
        return 1
    else:
        return Fibonacci(n - 1) + Fibonacci(n - 2)

print(Fibonacci(int(input("Enter the number"))))
```

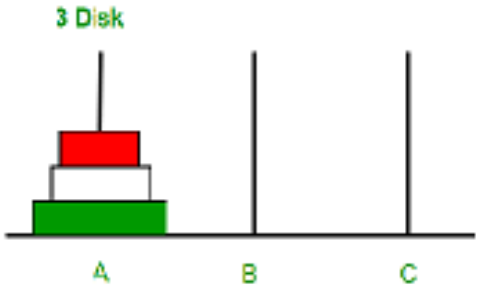
# Tower of Hanoi

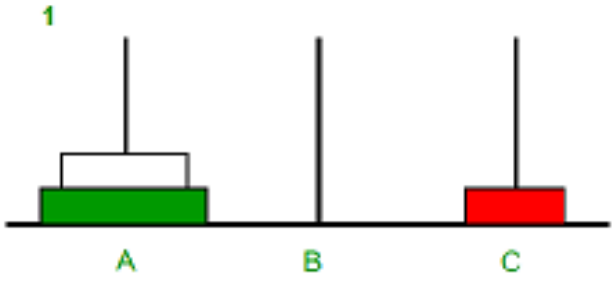
Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

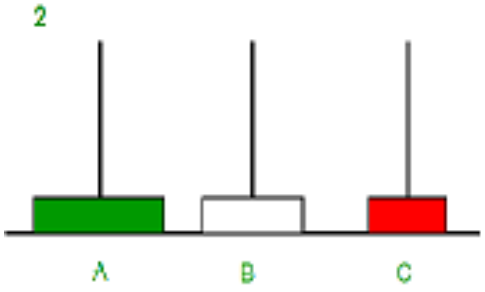
1) Only one disk can be moved at a time.

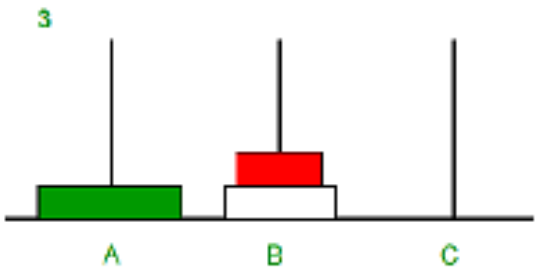
2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.

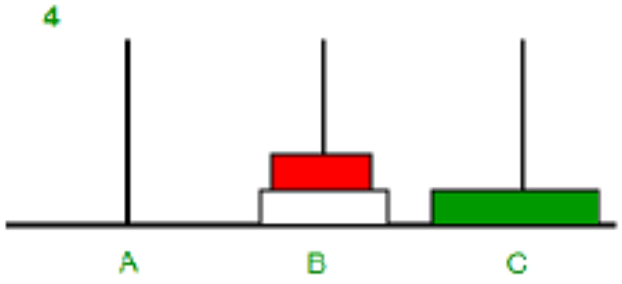
3) No disk may be placed on top of a smaller disk.

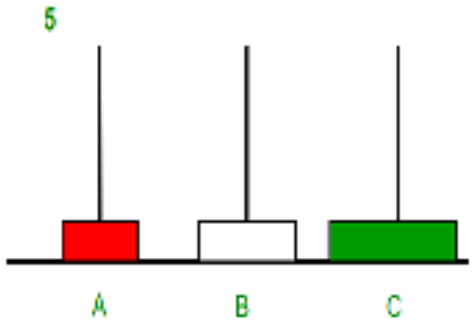

3 Disk
A
B
C

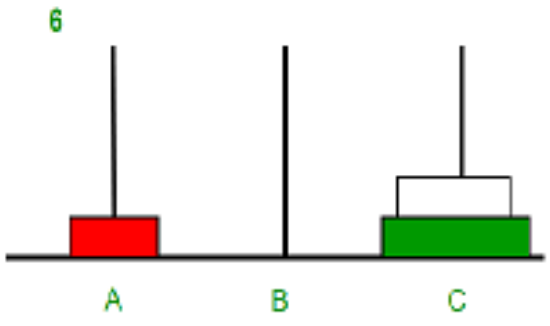

1
A
B
C

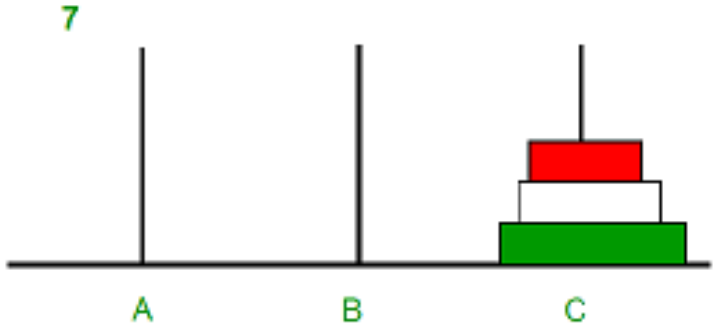

2
A
B
C


3
A
B
C


4
A
B
C


5
A
B
C


6
A
B
C


7
A
B
C

# Python Program of Tower of Hanoi

```
def TowerOfHanoi(n, source, destination, auxiliary):
    if n == 1:
        print("Move disk 1 from source", source, "to destination", destination)
        return
    TowerOfHanoi(n - 1, source, auxiliary, destination)
    print("Move disk", n, "from source", source, "to destination", destination)
    TowerOfHanoi(n - 1, auxiliary, destination, source)


# Driver code
n = 3
TowerOfHanoi(n, 'A', ‘C', ‘B')
# A, C, B are the name of rods
```

# Output

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

# Python Iterators

- An iterator is an object that contains a countable number of values.
- An iterator is an object that can be iterated upon, meaning that you can traverse through all the values.
- Technically, in Python, an iterator is an object which implements the iterator protocol, which consist of the methods __iter__() and __next__()

## Return an iterator from a tuple, and print each value:

```
mytuple = ("apple", "banana", "cherry")
myit = iter(mytuple)

print(next(myit))
print(next(myit))
print(next(myit))
```

**Strings are also iterable objects, containing a sequence of characters:**

```
mystr = "banana"
myit = iter(mystr)

print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
print(next(myit))
```

# Create an Iterator

To create an object/class as an iterator you have to implement the methods __iter__() and __next__() to your object.

- The __iter__() method acts similar, you can do operations (initializing etc.), but must always return the iterator object itself.

- The __next__() method also allows you to do operations, and must return the next item in the sequence.

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    x = self.a
    self.a += 1
    return x
```

```
myclass = MyNumbers()
myiter = iter(myclass)

print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
print(next(myiter))
```

```
1
2
3
4
5
```

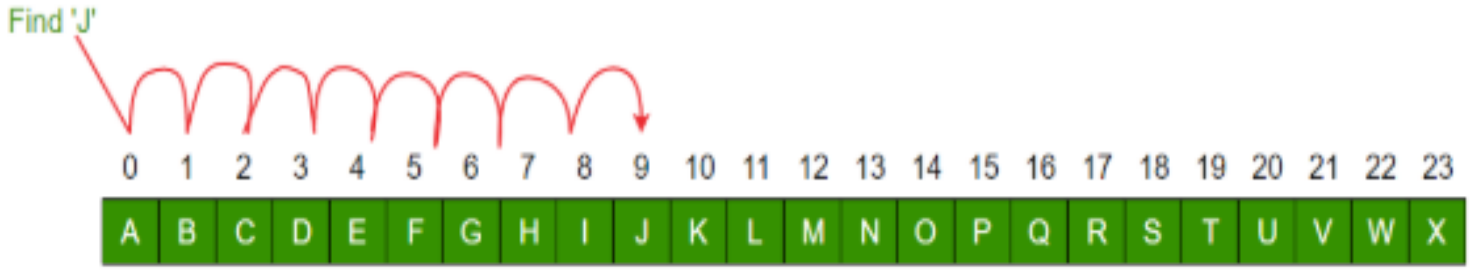
# Search

# Python Program for Linear Search

A simple approach is to do linear search

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[]
- If x matches with an element, return the index.
- If x doesn’t match with any of elements, return -1.

## Example:


Find 'J'
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
A B C D E F G H I J K L M N O P Q R S T U V W X

# Code For Linear Search

```
def search(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1
```

NOTE: The time complexity of  Linear Search is O(n)

# Python Program for Binary Search

We basically ignore half of the elements just after one comparison.

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half sub array after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.

```
def binarySearch(arr, l, r, x):
    while l <= r:
        mid = (l + r ) //2;
        # Check if x is present at mid
        if (arr[mid] == x):
            return mid
            # If x is greater, ignore left half
        elif (arr[mid] < x):
            l = mid + 1
        # If x is smaller, ignore right half
        else:
            r = mid - 1
    # If we reach here, then the element was not
present
    return -1
```

```
arr = [2, 3, 4, 10, 40]
x = 10
length=len(arr) - 1
result = binarySearch(arr, 0, length, x)
```

# Sorting & Merging

# Selection Sort Algorithm

- Selection sort is conceptually the most simplest sorting algorithm. This algorithm will first find the **smallest** element in the array and swap it with the element in the **first** position, then it will find the **second smallest** element and swap it with the element in the **second** position, and it will keep on doing this until the entire array is sorted.
- It is called selection sort because it repeatedly **selects** the next-smallest element and swaps it into the right place.

# Algorithm

Step 1 – Set MIN to location 0

Step 2 – Search the minimum element in the list

Step 3 – Swap with value at location MIN

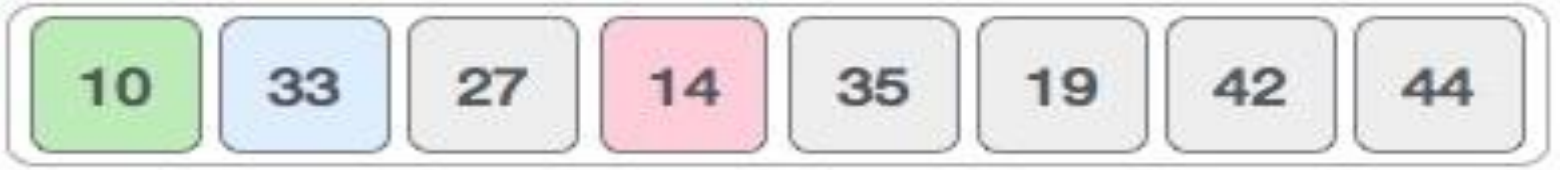
Step 4 – Increment MIN to point to next element

Step 5 – Repeat until list is sorted


For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

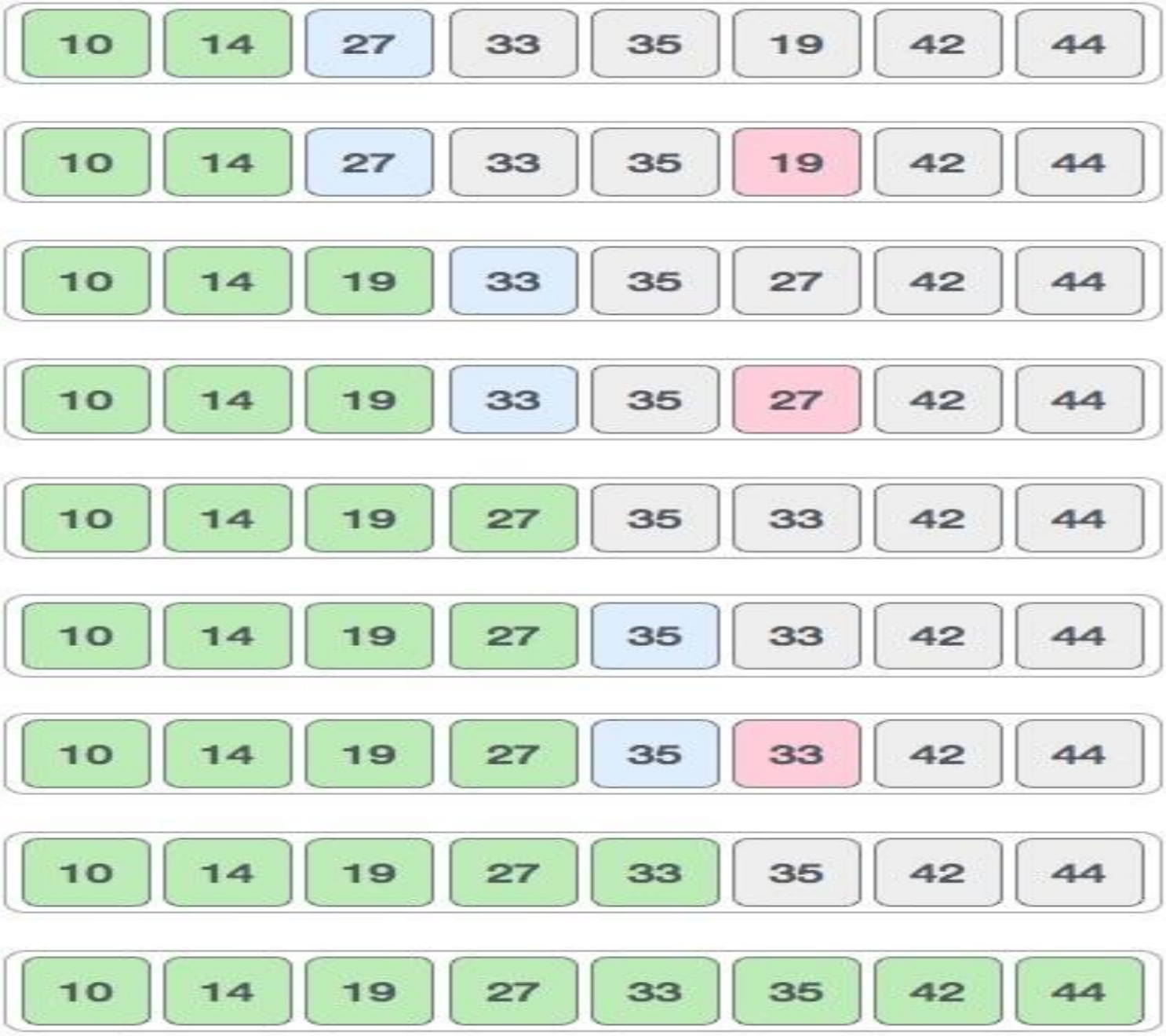

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

For the second position, where 33 is residing, we start scanning the rest of the list in a l




We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.


After two iterations, two least values are positioned at the beginning in a sorted manner.


10 14 27 33 35 19 42 44
10 14 27 33 35 19 42 44
10 14 19 33 35 27 42 44
10 14 19 33 35 27 42 44
10 14 19 27 35 33 42 44
10 14 19 27 35 33 42 44
10 14 19 27 35 33 42 44
10 14 19 27 33 35 42 44
10 14 19 27 33 35 42 44

# Complexity Analysis of Selection Sort

- Selection Sort requires two nested for loops to complete itself.
- Hence for a given input size of n, following will be the time and space complexity for selection sort algorithm:
  - Worst Case Time Complexity [ Big-O ]: **$O(n^2)$**
  - Best Case Time Complexity [Big-omega]: $\Omega$ **$(n^2)$**
  - Average Time Complexity [Big-theta]: **$\theta(n^2)$**
  - Space Complexity: **$O(1)$**

```
A = [64, 25, 12, 22, 11]
for i in range(len(A)):
    min_idx = i
    for j in range(i+1, len(A)):
        if A[min_idx] > A[j]:
            min_idx = j
    A[i], A[min_idx] = A[min_idx], A[i]
print ("Sorted array")
for i in range(len(A)):
    print(A[i])
```

# Merge Sort Algorithm

- Merge Sort follows the rule of **Divide and Conquer** to sort a given set of numbers/elements, recursively, hence consuming less time.
- Merge sort runs in O(n*log n) time in all the cases.
- Before jumping on to, how merge sort works and it's implementation, first lets understand what is the rule of **Divide and Conquer**?

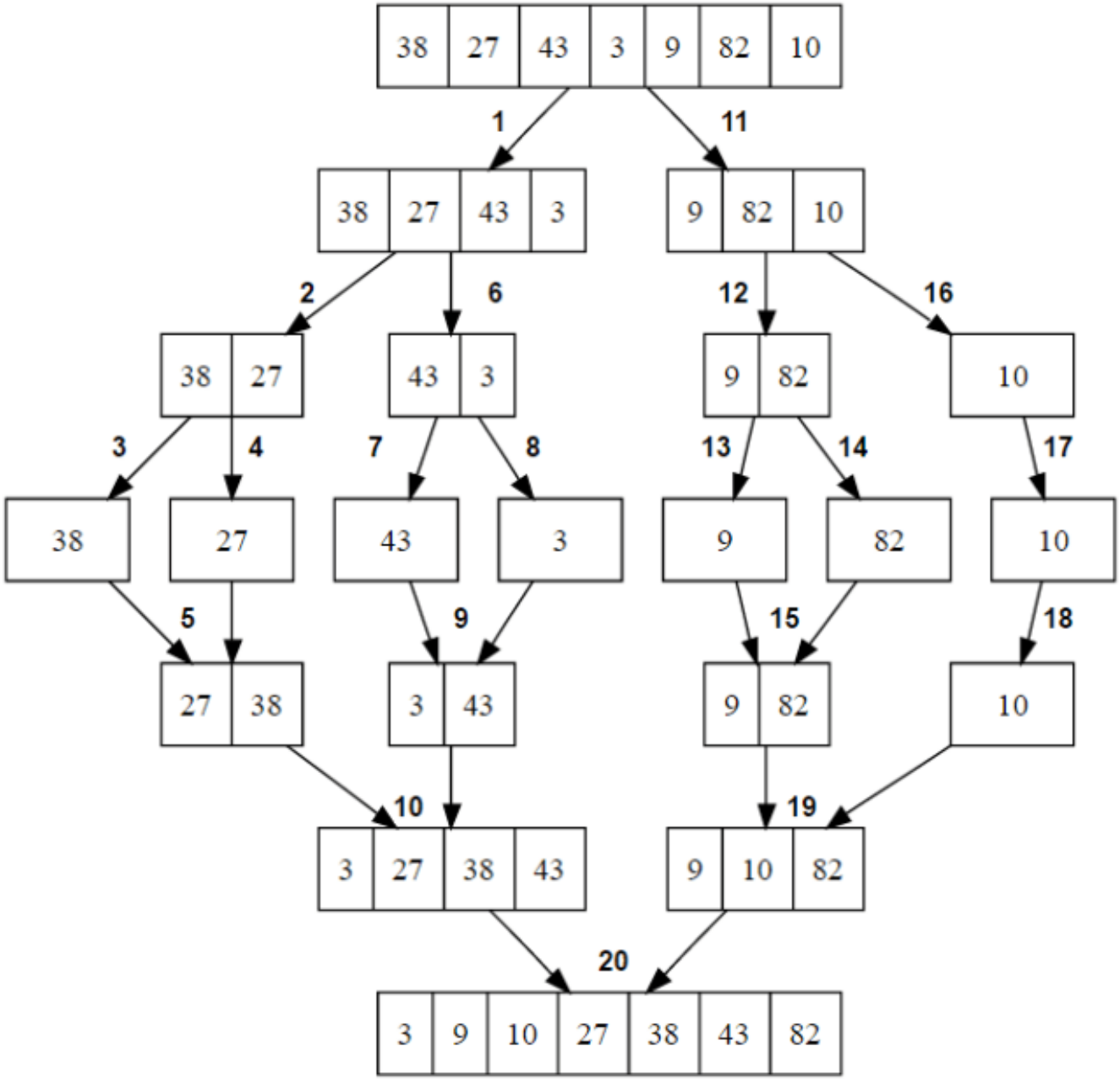
# Divide and Conquer

- If we can break a single big problem into smaller sub-problems, solve the smaller sub-problems and combine their solutions to find the solution for the original big problem, it becomes easier to solve the whole problem.
- In **Merge Sort**, the given unsorted array with n elements, is divided into n sub-arrays, each having **one** element, because a single element is always sorted in itself. Then, it repeatedly merges these sub-arrays, to produce new sorted sub-arrays, and in the end, one complete sorted array is produced.

The concept of Divide and Conquer involves three steps:

- **Divide** the problem into multiple small problems.
- **Conquer** the subproblems by solving them. The idea is to break down the problem into atomic subproblems, where they are actually solved.
- **Combine** the solutions of the subproblems to find the solution of the actual problem.

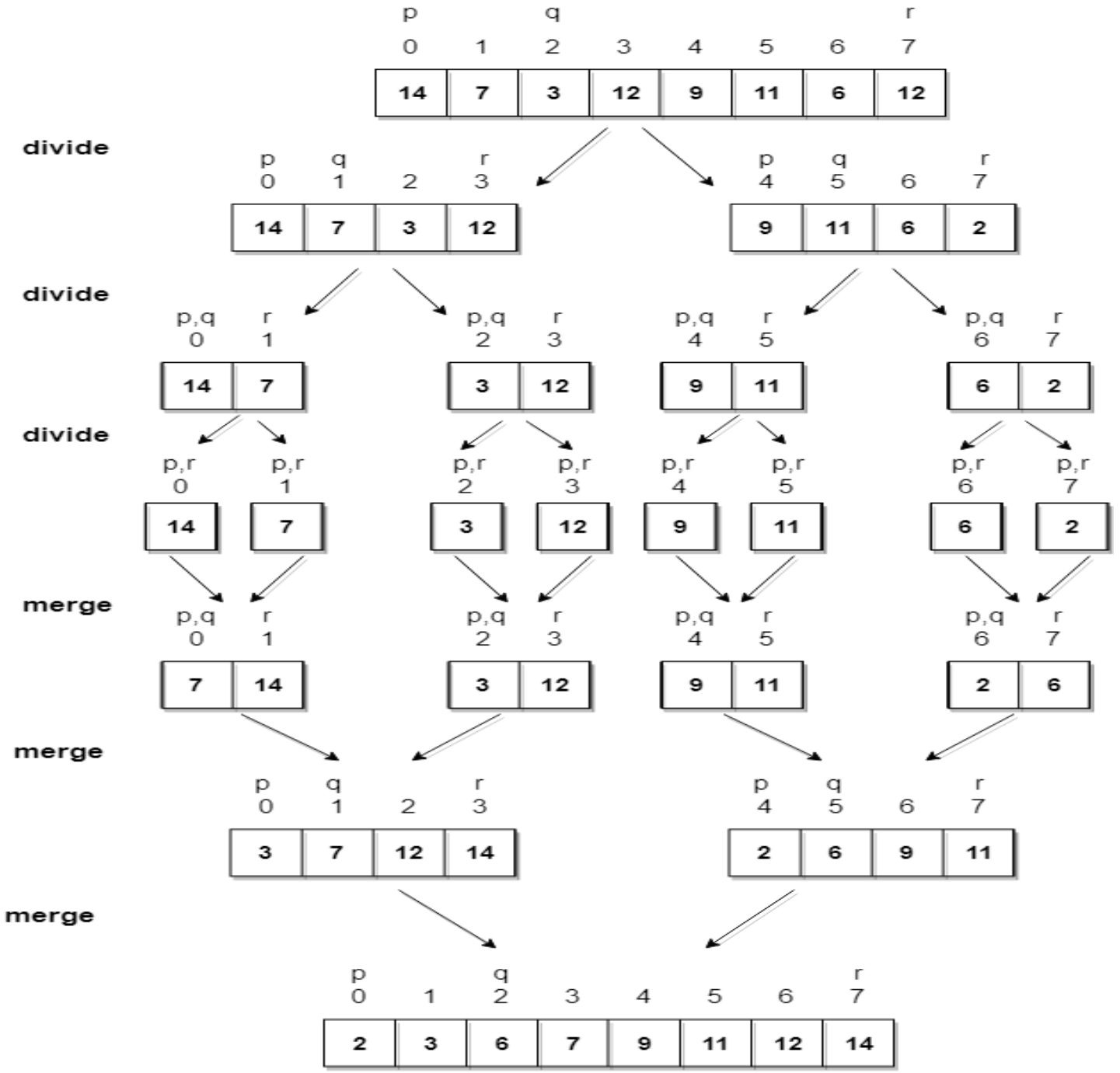
https://www.hackerearth.com/practice/algorithms/sorting/merge-sort/visualize/


38 27 43 3 9 82 10
1
11
38 27 43 3
9 82 10
2
6
12
16
38 27
43 3
9 82
10
3
4
7
8
13
14
17
38
27
43
3
9
82
10
5
9
15
18
27 38
3 43
9 82
10
10
19
3 27 38 43
9 10 82
20
3 9 10 27 38 43 82

# Pseudo Code

```
MERGE-SORT(A, p, r)
1  if p < r
2      q = ⌊(p + r)/2⌋
3      MERGE-SORT(A, p, q)
4      MERGE-SORT(A, q + 1, r)
5      MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
 1  n1 ← q − p + 1
 2  n2 ← r − q
 3  create arrays L[1 . . n1 + 1] and R[1 . . n2 + 1]
 4  for i ← 1 to n1
 5      do L[i] ← A[p + i − 1]
 6  for j ← 1 to n2
 7      do R[j] ← A[q + j]
 8  L[n1 + 1] ← ∞
 9  R[n2 + 1] ← ∞
10  i ← 1
11  j ← 1
12  for k ← p to r
13      do if L[i] ≤ R[j]
14            then A[k] ← L[i]
15                 i ← i + 1
16            else A[k] ← R[j]
17                 j ← j + 1
```


p q r
0 1 2 3 4 5 6 7
14 7 3 12 9 11 6 12
divide
p q r
0 1 2 3
14 7 3 12
p q r
4 5 6 7
9 11 6 2
divide
p,q r
0 1
14 7
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
6 2
divide
p,r
0
p,r
1
p,r
2
p,r
3
p,r
4
p,r
5
p,r
6
p,r
7
14
7
3
12
9
11
6
2
merge
p,q r
0 1
7 14
p,q r
2 3
3 12
p,q r
4 5
9 11
p,q r
6 7
2 6
merge
p q r
0 1 2 3
3 7 12 14
p q r
4 5 6 7
2 6 9 11
merge
p q r
0 1 2 3 4 5 6 7
2 3 6 7 9 11 12 14

# Merge List

```
test_list1 = [3, 4, 7, 8, 101]
test_list2 = [1, 5, 65, 95, 111,123,143,197]
size_1 = len(test_list1)
size_2 = len(test_list2)
res = []
i, j = 0, 0
while i < size_1 and j < size_2:
    if test_list1[i] < test_list2[j]:
        res.append(test_list1[i])
        i += 1
    else:
        res.append(test_list2[j])
        j += 1
res = res + test_list1[i:] + test_list2[j:]
print ("The combined sorted list is : " ,res)
```

# Merge Sort

```
def merge(arr, l, m, r):
    n1 = m - l + 1
    n2 = r - m
    L = [0] * (n1)
    R = [0] * (n2)
    for i in range(0, n1):
        L[i] = arr[l + i]
    for j in range(0, n2):
        R[j] = arr[m + 1 + j]
    i = 0  # Initial index of first subarray
    j = 0  # Initial index of second subarray
    k = l  # Initial index of merged subarray
```

```
while i < n1 and j < n2:
    if L[i] <= R[j]:
        arr[k] = L[i]
        i += 1
    else:
        arr[k] = R[j]
        j += 1
    k += 1
while i < n1:
    arr[k] = L[i]
    i += 1
    k += 1
while j < n2:
    arr[k] = R[j]
    j += 1
    k += 1
```

```
def mergeSort(arr, l, r):
    if l < r:
        m = (l + (r - 1)) // 2
        mergeSort(arr, l, m)
        mergeSort(arr, m + 1, r)
        merge(arr, l, m, r)
arr = [12, 11, 13, 5, 6, 7]
n = len(arr)
print("Given array is")
for i in range(n):
    print("%d" % arr[i]),
mergeSort(arr, 0, n - 1)
print("\n\nSorted array is")
for i in range(n):
    print("%d" % arr[i])
```