Recursion-1

Introduction

The process in which a function calls itself is called **recursion** and the corresponding function is called a **recursive function**.

Since computer programming is a fundamental application of mathematics, so let us first try to understand the mathematical reasoning behind recursion.

In general, we all are aware of the concept of functions. In a nutshell, functions are mathematical equations that produce an output on providing input. **For example:** Suppose the function **F(x)** is a function defined by:

$$F(x) = x^2 + 4$$

We can write the **Python Code** for this function as:

```
def F(int x):
    return (x * x + 4)
```

Now, we can pass different values of x to this function and receive our output accordingly.

Before moving onto the recursion, let's try to understand another mathematical concept known as the **Principle of Mathematical Induction (PMI)**.

Principle of Mathematical Induction (PMI) is a technique for proving a statement, a formula, or a theorem that is asserted about a set of natural numbers. It has the following three steps:

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

2. **Step of assumption:** In this step, we will assume that the desired statement is valid for **n = k**.
3. **To prove step:** From the results of the assumption step, we will prove that, **n = k + 1** is also true for the desired equation whenever **n = k** is true.

For Example: Let's prove using the **Principle of Mathematical Induction** that:

```
S(N): 1 + 2 + 3 + ... + N = (N * (N + 1))/2
```

(The sum of first N natural numbers)

Proof:

Step 1: For N = 1, S(1) = 1 is true.

Step 2: Assume, the given statement is true for N = k, i.e.,

```
1 + 2 + 3 + .... + k = (k * (k + 1))/2
```

Step 3: Let's prove the statement for N = k + 1 using step 2.

To Prove: `1 + 2 + 3 + ... + (k+1) = ((k+1)*(k+2))/2`

Proof:

Adding (k+1) to both LHS and RHS in the result obtained on step 2:

```
1 + 2 + 3 + ... + (k+1) = (k*(k+1))/2 + (k+1)
```

Now, taking (k+1) common from RHS side:

```
1 + 2 + 3 + ... + (k+1) = (k+1)*((k + 2)/2)
```

According the statement that we are trying to prove:

```
1 + 2 + 3 + ... + (k+1) = ((k+1)*(k+2))/2
```

Hence proved.

One can think, why are we discussing these over here. To answer this question, we need to know that these three steps of PMI are related to the three steps of recursion, which are as follows:

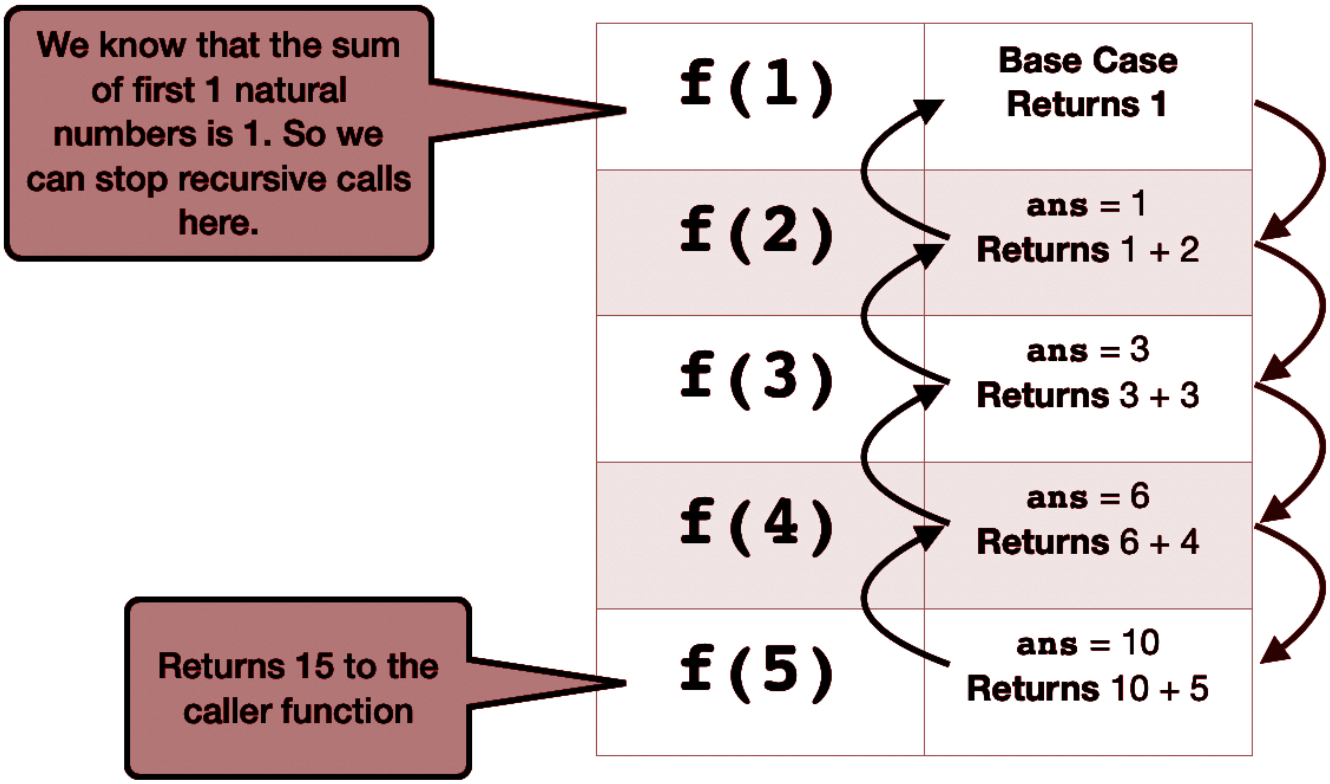
1. **Induction Step and Induction Hypothesis:** Here, the Induction Step is the main problem which we are trying to solve using recursion, whereas the Induction Hypothesis is the sub-problem, using which we'll solve the induction step. Let's define the Induction Step and Induction Hypothesis for our running example:
   Induction Step: Sum of first n natural numbers - F(n)
   Induction Hypothesis: This gives us the sum of the first n-1 natural numbers - F(n-1)
2. Express F(n) in terms of F(n-1) and write code:

```
F(N) = F(N-1)+ N
```

   Thus, we can write the Python code as:

```
def f(N):
    ans = f(N-1) #Induction Hypothesis step
    return ans + N  #Solving problem from result in previous step
```

3. The code is still not complete. The missing part is the base case. Now we will dry run to find the case where the recursion needs to stop.



4. After the dry run, we can conclude that for N equals 1, the answer is 1, which we already know. So we'll use this as our base case. Hence the final code becomes:

```
def f(N):
    if(N == 1):    #Base Case
        return 1
    ans = f(N-1)
    return ans + N
```

This is the main idea to solve recursive problems. To summarize, we will always focus on finding the solution to our starting problem and tell the function to compute the rest for us using the particular hypothesis. This idea will be studied in detail in further sections with more examples.

Now, we'll learn more about recursion by solving problems which contain smaller subproblems of the same kind. Recursion in computer science is a method where the solution to the question depends on solutions to smaller instances of the same problem. By the exact nature, it means that the approach that we use to solve the original problem can be used to solve smaller problems as well. So, in other words, in recursion, a function calls itself to solve smaller problems. **Recursion** is a popular approach for solving problems because recursive solutions are generally easier to think than their iterative counterparts, and the code is also shorter and easier to understand.

Working of recursion

We can define the steps of the recursive approach by summarizing the above three steps:

- **Base case:** A recursive function must have a terminating condition at which the process will stop calling itself. Such a case is known as the base case. In the absence of a base case, it will keep calling itself and get stuck in an infinite loop. Soon, the **recursion depth*** will be exceeded and it will throw an error.
- **Recursive call:** The recursive function will invoke itself on a smaller version of the main problem. We need to be careful while writing this step as it is crucial to correctly figure out what your smaller problem is.
- **Small calculation:** Generally, we perform a calculation step in each recursive call. We can achieve this calculation step before or after the recursive call depending upon the nature of the problem.

Note*: Recursion uses an in-built stack which stores recursive calls. Hence, the number of recursive calls must be as small as possible to avoid memory-overflow. If the number of recursion calls exceeded the maximum permissible amount, the **recursion depth*** will be exceeded.

Now, let us see how to solve a few common problems using Recursion.

Problem Statement - Find Factorial of a Number

We want to find out the factorial of a natural number.

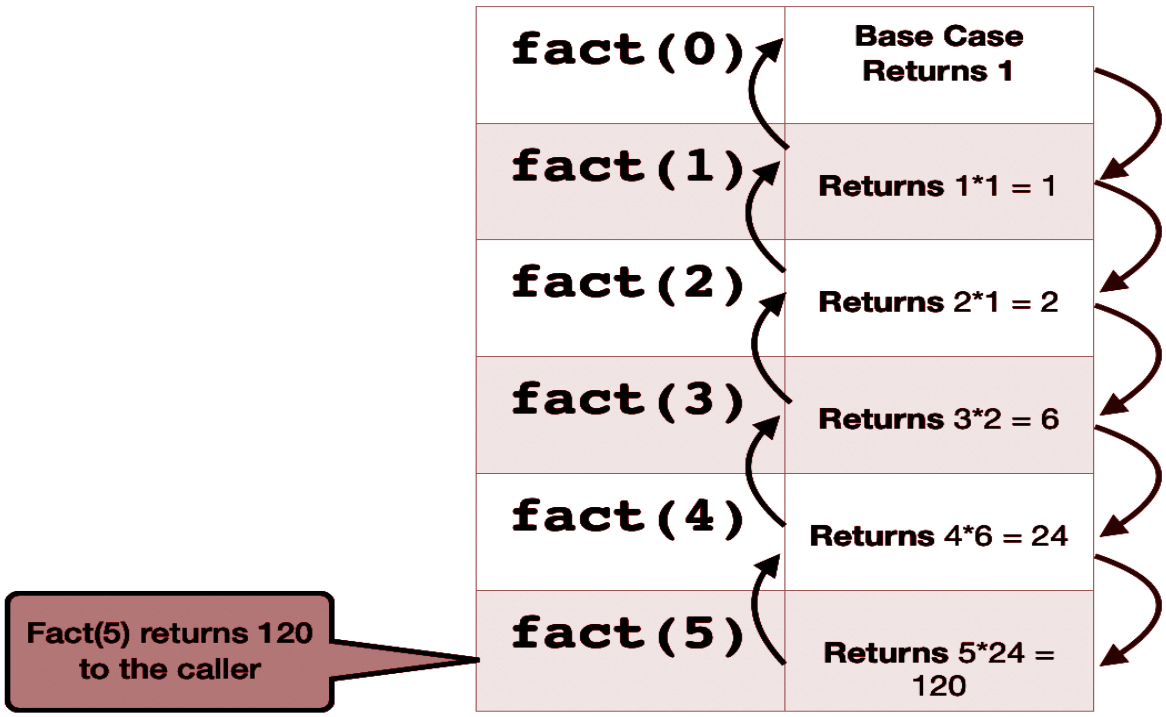
Approach: Figuring out the three steps of PMI and then relating the same using recursion.

1. **Induction Step:** Calculating the factorial of a number n - **F(n)**
   Induction Hypothesis: We have already obtained the factorial of n-1 - **F(n-1)**

2. Expressing F(n) in terms of F(n-1): `F(n)=n*F(n-1)`. Thus we get:

```
def fact(n):
     ans = fact(n-1) #Assumption step
     return ans * n; #Solving problem from assumption step
```

3. The code is still not complete. The missing part is the base case. Now we will dry run to find the case where the recursion needs to stop. Consider **n = 5**:



As we can see above, we already know the answer of n = 0, which is 1. So we will keep this as our base case. Hence, the code now becomes:

```
def factorial(n):
  if n == 0: #base case
      return 1
  else:
      return n*factorial(n-1) # recursive case
```

Problem Statement - Fibonacci Number

Function for Fibonacci series:

F(n) = F(n-1) + F(n-2), F(0) = 0 and F(1) = 1

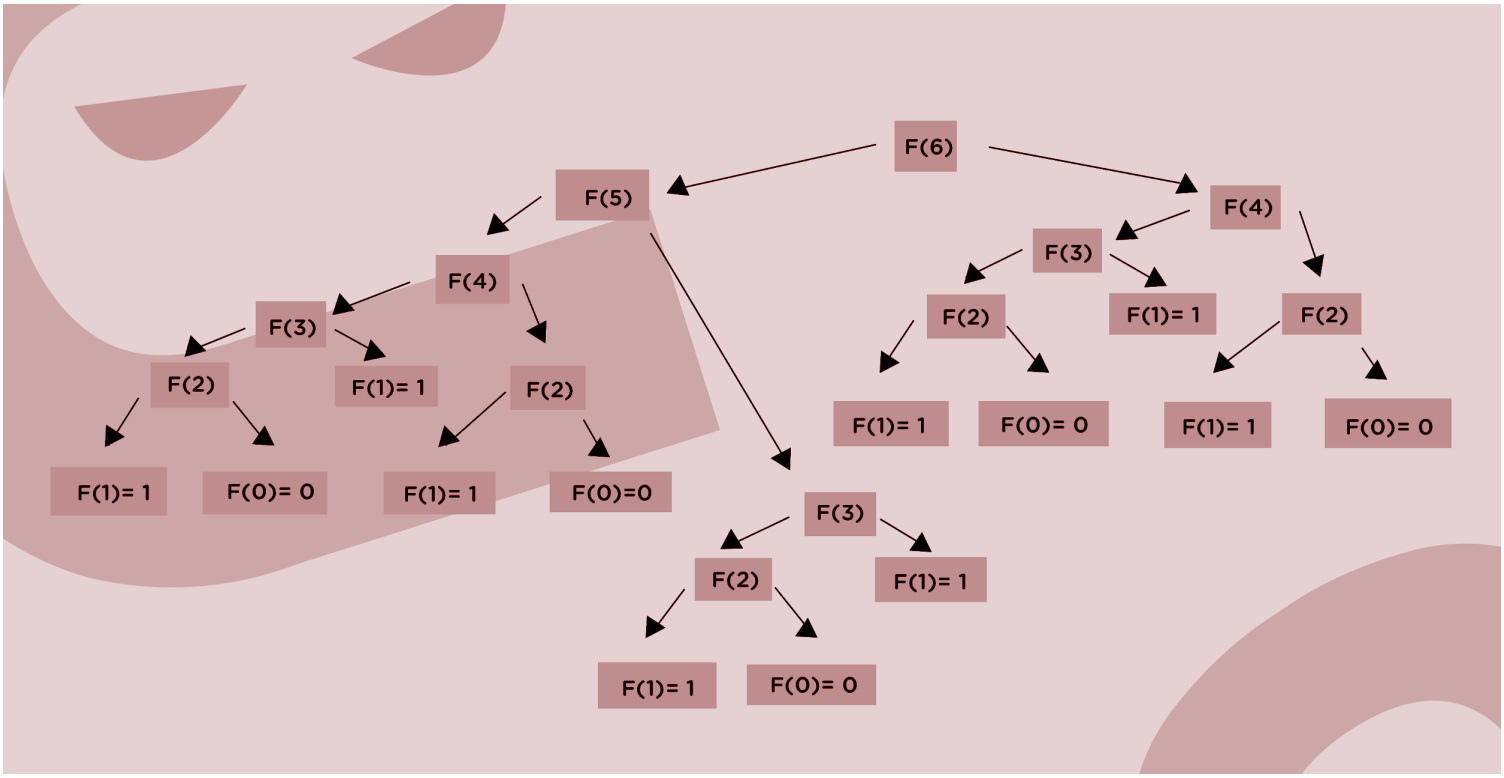
Approach: Figuring out the three steps of PMI and then relating the same using recursion.

1. **Induction Step:** Calculating the n^{th} Fibonacci number n.
   Induction Hypothesis: We have already obtained the $(n-1)^{th}$ and $(n-2)^{th}$ Fibonacci numbers.

2. Expressing F(n)in terms of F(n-1) and F(n-2): $\mathbf{F_n = F_{n-1} + F_{n-2}}$.

```
def f(n):
     ans = f(n-1) + f(n-2) #Assumption step
     return ans #Solving problem from assumption step
```

3. Let's dry run the code for achieving the base case: (Consider n= 6)



From here we can see that every recursive call either ends at 0 or 1 for which we already know the answer: **F(0) = 0 and F(1) = 1**. Hence using this as our base case in the code below:

```
def fib(n):
  if n <= 1:
      return (n)
  else:
      return (fib(n-1) + fib(n-2))
```

Recursion and array

Let us take an example to understand recursion on arrays.

Problem Statement - Check If Array Is Sorted.

We have to tell whether the given array is sorted or not using recursion.

For example:

- If the array is {2, 4, 8, 9, 9, 15}, then the output should be **YES.**
- If the array is {5, 8, 2, 9, 3}, then the output should be **NO.**

Approach: Figuring out the three steps of PMI and then relating the same using recursion.

1. **Assumption step:** We assume that we have already obtained the answer to the array starting from index 1. In other words, we assume that we know whether the array (starting from the first index) is sorted.
2. **Solving the problem from the results of the "Assumption step":** Before going to the assumption step, we must check the relation between the first two elements. Find if the first two elements are sorted or not. If the elements are not in sorted order, then we can directly return false. If the first two elements are in sorted order, then we will check for the remaining array through recursion.

```
def isSorted(a, size):
    if (a[0] > a[1]): #Small Calculation
        return false
    isSmallerSorted = isSorted(a + 1, size - 1) #Assumption step
    return isSmallerSorted
```

3. We can see that in the case when there is only a single element left or no element left in our array, the array is always sorted. Let's check the final code now:

```
def isSorted(a, size):
    if (size == 0 or size ==1) #Base case
        return true

    if (a[0] > a[1]) # Small calculation
        return false
```

```
    isSmallerSorted = isSorted(a + 1, size - 1) #Recursive call
    return isSmallerSorted

arr = [2, 3, 6, 10, 11]
if(isSorted(arr, 5)):
    print("Yes")
else:
    print("No")
```

Problem Statement - First Index of Number

Given an array of length **N** and an integer **x**, you need to find and return the first index of integer **x** present in the array. Return **-1** if it is not present in the array. The first index means that if **x** is present multiple times in the given array, you have to return the index at which **x** comes first in the array.

To get a better understanding of the problem statement, consider the given cases:

Case 1: `Array = {1,4,5,7,2}, Integer = 4`

Output: 1

Explanation: 4 is present at 1st position in the array.

Case 2: `Array = {1,3,5,7,2}, Integer = 4`

Output: -1

Explanation: 4 is not present in the array

Case 3: `Array = {1,3,4,4,4}, Integer = 4`

Output: 2

Explanation: 4 is present at 3 positions in the array; i.e., [2, 3, 4]. But as the question says, we have to find out the first occurrence of the target value, so the answer should be 2.

Approach:

Now, to solve the question, we have to figure out the following three elements of the solution:

1. **Base case**
2. **Recursive call**
3. **Small calculation**

Small calculation part:

Let the array be: [5, 5, 6, 2, 5] and x = 6. Now, if we want to find 6 in the array, then first we have to check with the first index.

```
if(arr[0] == x):
    return 0
```

Recursive Call step:

- Since, in the running example, the 0^{th} index element is not equal to 6, so we will have to make a recursive call for the remaining array: `[5,6,2,5], x=6`.
- The recursive call will look like this:

```
f(arr+1, size-1, x)
```

- In the recursive call, we are incrementing the pointer and decrementing the size of the array.
- We have to assume that the answer will come from the recursive call. The answer will come in the form of an integer.
- If the answer is -1, this denotes that the element is not present in the remaining array.
- If the answer is any other integer (other than -1), then this denotes that the element is present in the remaining array.
- If the element is present at the **i^{th}** index in the remaining array, then it will be present at **$(i+1)^{th}$** index in the main array. For instance, in the running example, 6 is present at index 1 in the remaining array and at index 2 in the array.

Base case step:

- The base case for this question can be identified by dry running the case when you are trying to find an element that is not present in the array.
- **For example:** Consider the array [5, 5, 6, 2, 5] and x = 10. On dry running, we can conclude that the base case will be the one when the size of the array becomes zero.
- When the size of the array becomes zero, then we will return -1. This is because if the base case is reached, then this means that the element is not present in the entire array.
- We can write the base case as:

```
if(size == 0): #Base Case
    return -1
```

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

Problem Statement - First Index of Number

Given an array of length **N** and an integer **x**, you need to find and return the first index of integer **x** present in the array. Return **-1** if it is not present in the array. The last index means that if **x** is present multiple times in the given array, you have to return the index at which **x** comes last in the array.

Case 1: `Array = {1,4,5,7,2}, Integer = 4`

Output: 1 (**Explanation:** 4 is present at 1st position in the array, which is the last and the only place where 4 is present in the given array.)

Case 2: `Array = {1,3,5,7,2}, Integer = 4`

Output: -1 (**Explanation:** 4 is not present in the array.)

Case 3: `Array = {1,3,4,4,4}, Integer = 4`

Output: 4 (**Explanation**: 4 is present at 3 positions in the array; i.e., [2, 3, 4], but as the question says, we have to find out the last occurrence of the target value, so the answer should be 4.)

Approach:

Now, to solve the question, we have to figure out the following three elements of the solution.

1. **Base case**
2. **Recursive call**
3. **Small calculation**

Let the array be: [5, 5, 6, 2, 5] and x = 6. Now, if we want to find 6 in the array, then first we have to check with the first index. This is the **small calculation part**.

Code:

```
if(arr[0] == x):
    return 0
```

Since, in the running example, the 0^{th} index element is not equal to 6, so we will have to make a recursive call for the remaining array: [5, 6, 2, 5] and x = 6. This is the **recursive call step**.

The recursive call will look like this:

```
f(arr+1, size-1, x)
```

- In the recursive call, we are incrementing the pointer and decrementing the size of the array.
- We have to assume that the answer will come for a recursive call. The answer will come in the form of an integer.
- If the answer is -1, this denotes that the element is not present in the remaining array; otherwise, we need to add 1 to our answer as for recursion, it might be the 0^{th} index, but for the previous recursive call, it was the first position.
- **For instance:** In the running example, 6 is present at index 1 in the remaining array and at index 2 in the array.

Base Case:

- The base case for this question can be identified by dry running the case when you are trying to find an element that is not present in the array.
- **For example:** [5, 5, 6, 2, 5] and x = 10. On dry running, we can conclude that the base case will be the one when the size of the array becomes zero.
- When the size of the array becomes zero, then we will return -1.
- This is because if the size of the array becomes zero, then this means that we have traversed the entire array and we were not able to find the target element.

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

All Indices of A Number

Here, given an array of length N and an integer x, you need to find all the indexes where x is present in the input array. Save all the indexes in an array (in increasing order) and return the size of the array.

Case 1: `Array = {1,4,5,7,2}, Integer = 4`

Output: [1], the size of the array will be 1 (as 4 is present at 1st position in the array, which is the only position where 4 is present in the given array).

Case 2: `Array = {1,3,5,7,2}, Integer = 4`

Output: [], the size of the array will be 0 (as 4 is not present in the array).

Case 3: `Array = {1,3,4,4,4}, Integer = 4`

Output: [2, 3, 4], the size of the array will be 3 (as 4 is present at three positions in the array; i.e., [2, 3, 4]).

Now, let's think about solving this problem...

Approach:

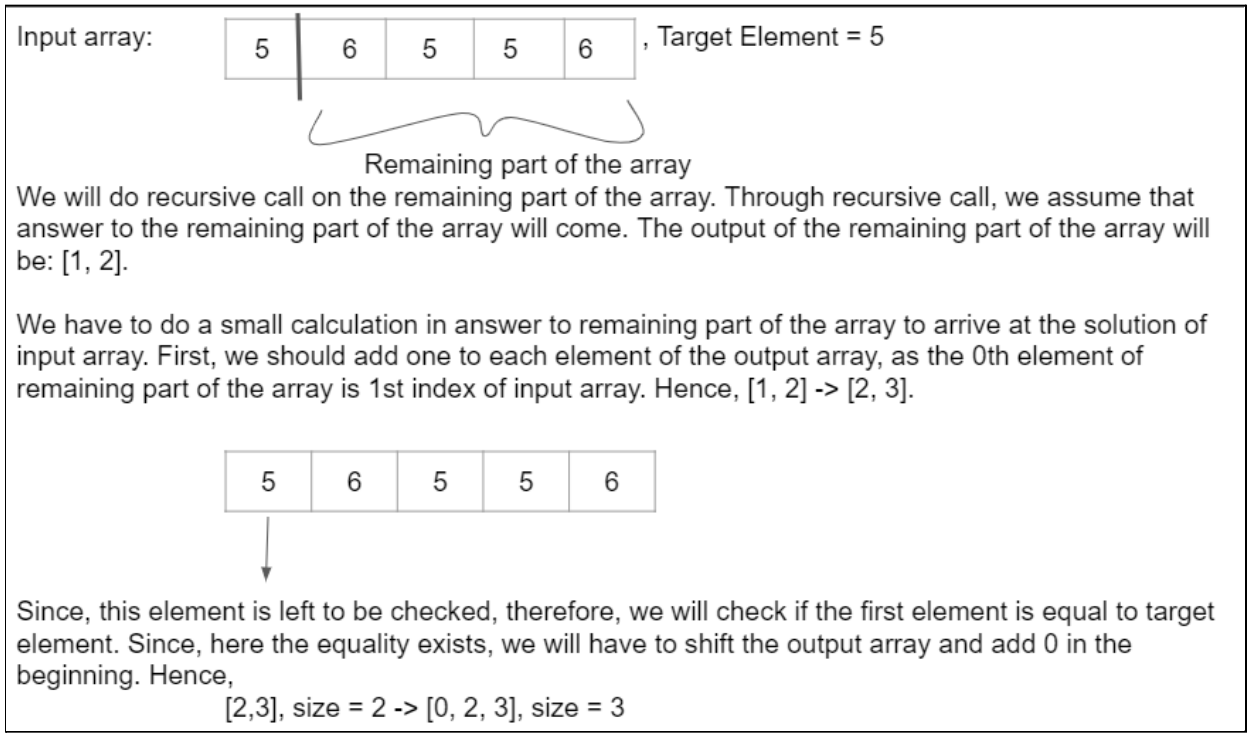
Now, to solve the question, we have to figure out the following three elements of the solution:

1. **Base case**
2. **Recursive call**
3. **Small calculation**

Let us assume the given array is: **[5, 6, 5, 5, 6]** and the target element is **5**, then the output array should be **[0, 2, 3]** and for the same array, let's suppose the target element is 6, then the output array should be **[1, 4]**.

To solve this question, the base case should be the case when the size of the input array becomes zero. In this case, we should simply return 0, since there are no elements.

The next two components of the solution are Recursive call and Small calculation. Let us try to figure them out using the following images:



So, the following are the recursive call and small calculation components of the solution:

Recursive Call

```
size = fun(arr + 1, size - 1, x, output)
```

Small Calculation:

1. Update the elements of the output array by adding one to them.

2. If the equality exists, then shift the elements and add 0 at the first index of the output array. Moreover, increment the size, as we have added one element to the output array.

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

Using the same concept, other problems can be solved using recursion, just remember to apply PMI and three steps of recursion intelligently.

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.

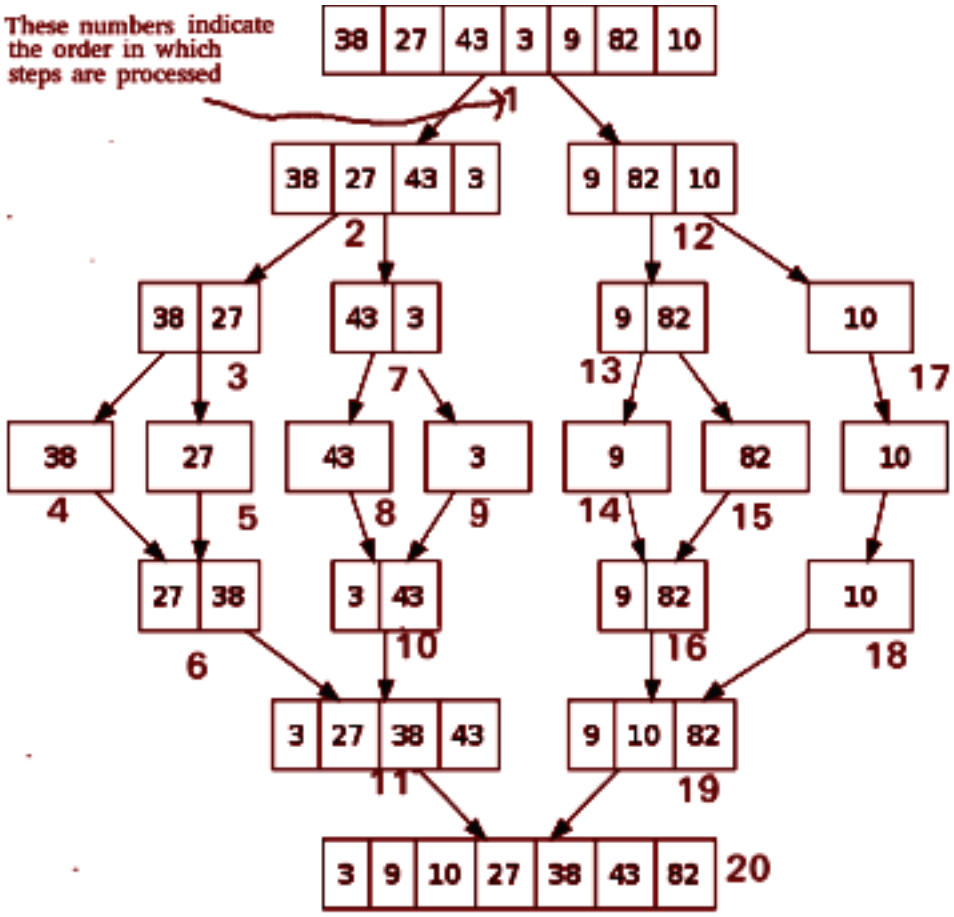
- **Step 1 −** If it is only one element in the list it is already sorted, return.
- **Step 2 −** Divide the list recursively into two halves until it can't be divided further.
- **Step 3 −** Merge the smaller lists into a new list in sorted order.

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.

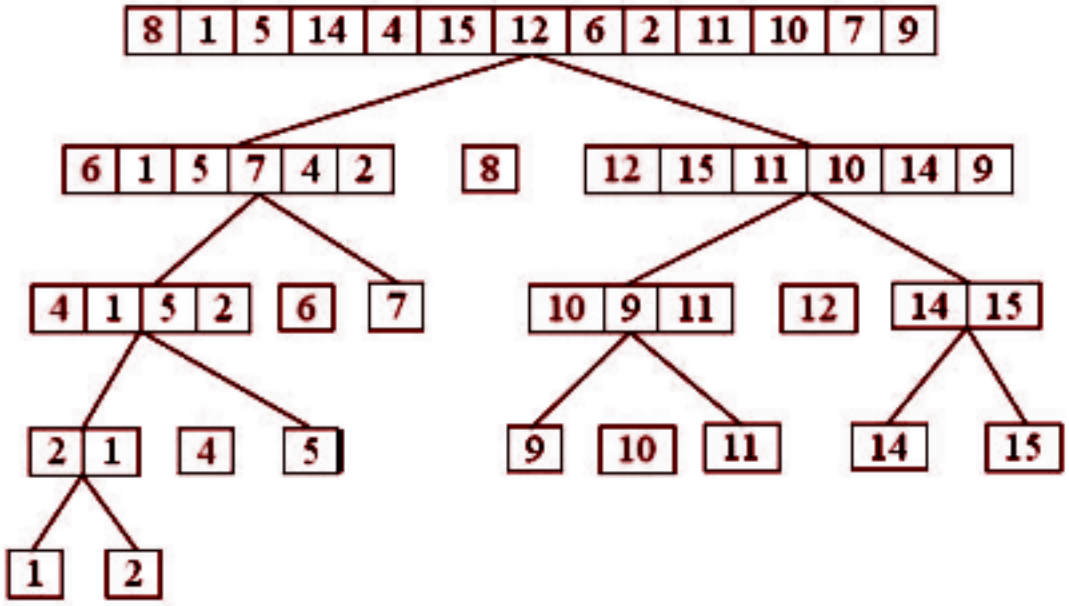
The advantage of quicksort over merge sort is that it does not require any extra space to sort the given array, that it is an in-place sorting technique.

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 :

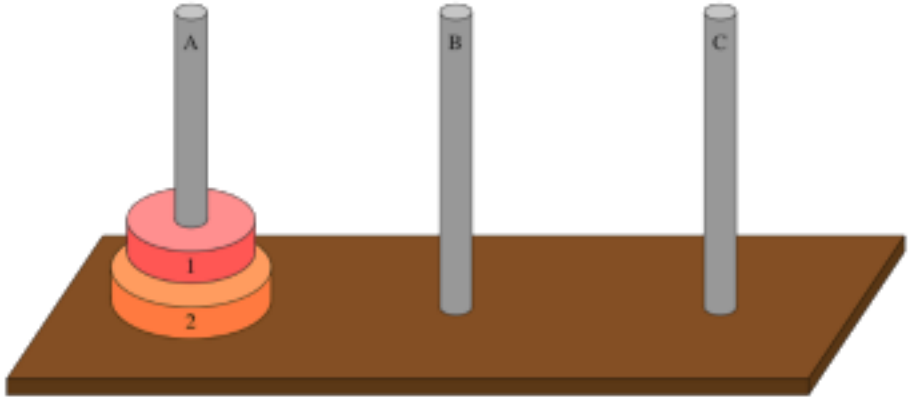
- Only one disk can be moved at a time.
- A disk can be moved only if it is on the top of a rod.
- No disk can be placed on the top of a smaller disk.

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'**.

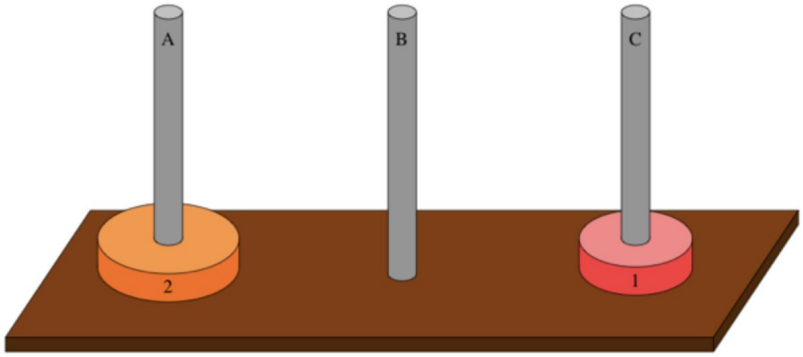
Let's see how to solve the problem recursively. We'll start with a really easy case **N=1**. We just need to move one disk from source to destination.

- 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.

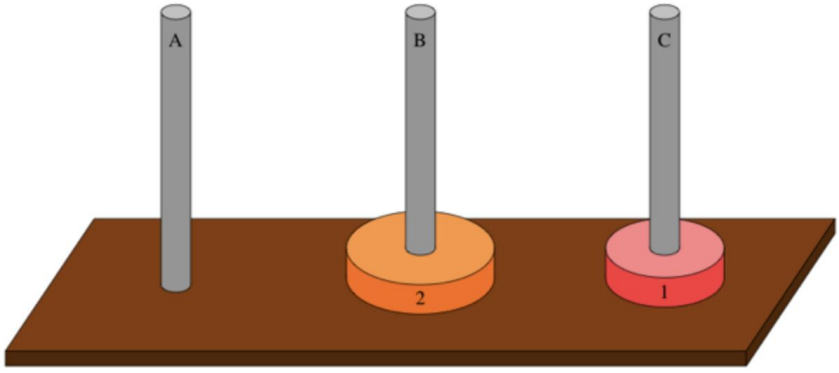
Now consider the case **N=2**. Here's what it looks like at the start:



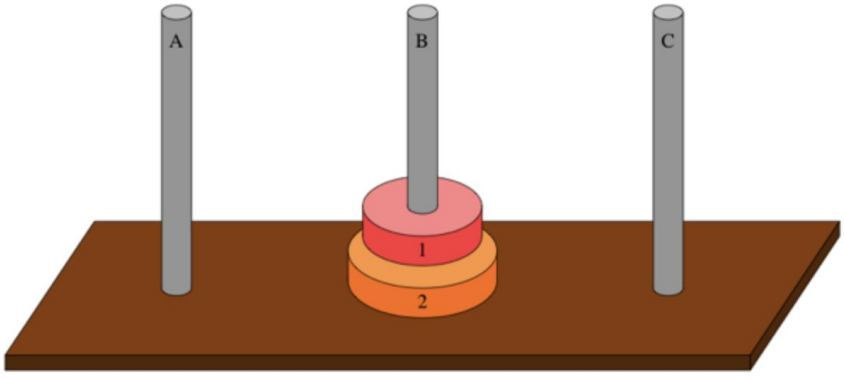
First, move **disk 1** from peg **A** to peg **C**:



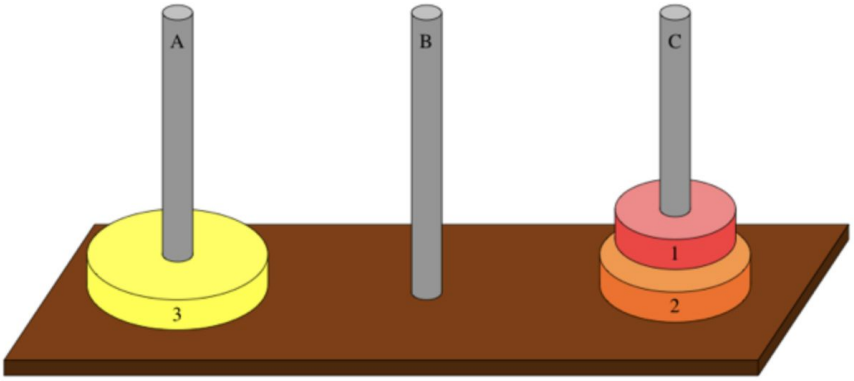
Notice that we're using peg **C** as a spare peg, a place to put **disk 1** so that we can get at **disk 2**. Now that **disk 2**—the bottommost disk—is exposed, move it to peg **B**:



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

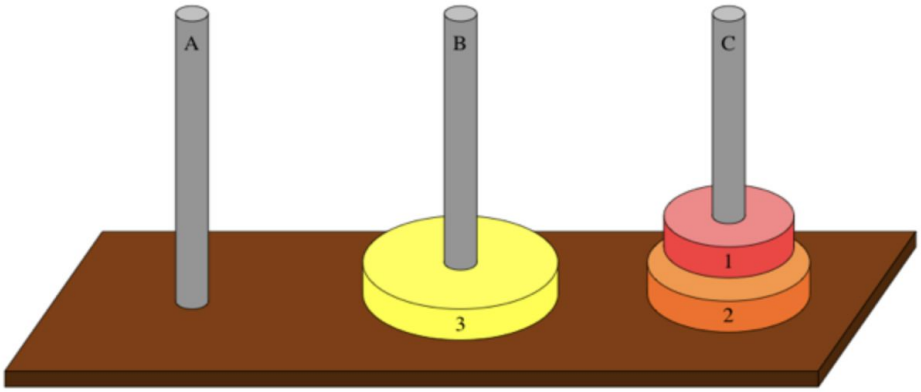


Now let us solve this problem for **3** disks. You need to expose the bottom disk (**disk 3**) so that you could move it from peg **A** to peg **B**. To expose **disk 3**, you needed to move disks 1 and 2 from peg **A** to the spare peg, which is peg **C**:

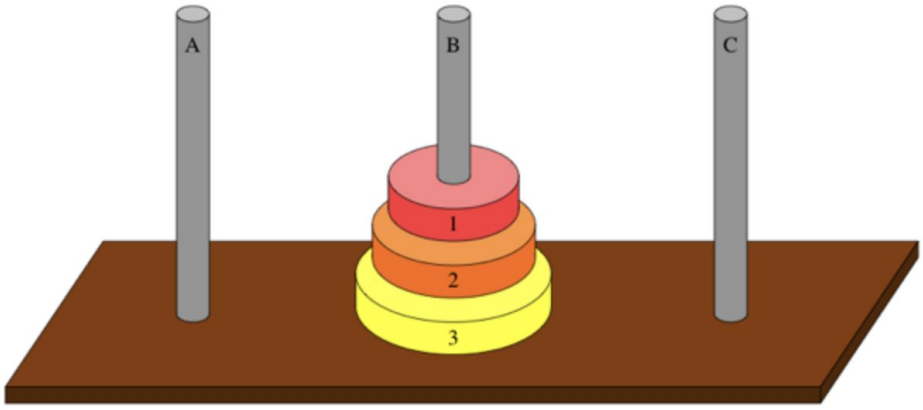


Wait a minute—it looks like two disks moved in one step, violating the first rule. But they did not move in one step. You agreed that you can move disks 1 and 2 from any peg to any peg, **using three steps.** The situation above represents what you have after three steps. (Move **disk 1** from peg **A**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

Object-Oriented Programming (OOPS-1)

Introduction to OOPS

Object-oriented programming System(OOPs) is a programming paradigm based on the concept of *"objects"* and *"classes"* that contain data and methods. The primary purpose of OOP is to increase the flexibility and maintainability of programs. It is used to structure a software program into simple, reusable pieces of code **blueprints** (called *classes*) which are used to create individual instances of *objects*.

Python supports a variety of programming approaches. One of the most useful and popular programming approaches is OOPS.

What is an Object?

The object is an entity that has a state and a behavior associated with it. It may be any real-world object like the mouse, keyboard, chair, table, pen, etc.
Integers, strings, floating-point numbers, even arrays and dictionaries, are all objects. More specifically, any single integer or any single string is an object. The number **12** is an object, the string **"Hello, world"** is an object, a list is an object that can hold other objects, and so on. You've been using objects all along and may not even realize it.

What is a Class?

A class is a **blueprint** that defines the variables and the methods (Characteristics) common to all objects of a certain kind.

Example: If **Car** is a class, then **Maruti 800** is an object of the **Car** class. All cars share similar features like 4 wheels, 1 steering wheel, windows, breaks etc. Maruti 800 (The **Car** object) has all these features.

Classes vs Objects (Or Instances)

Classes are used to create user-defined data structures. Classes define functions called **methods**, which identify the behaviors and actions that an object created from the class can perform with its data.

In this module, you'll create a **Car** class that stores some information about the characteristics and behaviors that an individual **Car** can have.

A class is a blueprint for how something should be defined. It doesn't contain any data. The **Car** class specifies that a name and a top-speed are necessary for defining a **Car**, but it doesn't contain the name or top-speed of any specific **Car**.

While the class is the blueprint, an instance is an object that is built from a class and contains real data. An instance of the **Car** class is not a blueprint anymore. It's an actual car with a **name**, like Creta, and with a **top speed** of 200 Km/Hr.

Put another way, a class is like a form or questionnaire. An **instance** is like a form that has been filled out with information. Just like many people can fill out the same form with their unique information, many instances can be created from a single class.

Defining a Class in Python

All class definitions start with the `class` keyword, which is followed by the name of the class and a colon(**:**). Any code that is indented below the class definition is considered part of the class's body.
Here is an example of a **Car** class:

```
class Car:
    pass
```

The body of the **Car** class consists of a single statement: the `pass` keyword. As we have discussed earlier, `pass` is often used as a placeholder indicating where code will eventually go. It allows you to run this code without Python throwing an error.

Note: Python class names are written in *CapitalizedWords* notation by convention. **For example**, a class for a specific model of Car like the Bugatti Veyron would be written as **BugattiVeyron**. The first letter is capitalized. This is just a good programming practice.

The **Car** class isn't very interesting right now, so let's spruce it up a bit by defining some properties that all Car objects should have. There are several properties that we can choose from, including **color**, **brand**, and **top-speed**. To keep things simple, we'll just use **color** and **top-speed**.

Constructor

- Constructors are generally used for instantiating an object.
- The task of a constructor is to initialize(assign values) to the data members of the class when an object of the class is created.
- In Python, the `.__init__()` method is called the **constructor** and is always called when an object is created.

- **Note:** Names that have leading and trailing double underscores are reserved for special use like the **__init__** method for object constructors. These methods are known as **dunder methods**.

Syntax of Constructor Declaration

```
def __init__(self):
    # body of the constructor
```

Types of constructors

- **Default Constructor:** The default constructor is a simple constructor that doesn't accept any arguments. Its definition has only one argument which is a reference to the instance being constructed known as `self`.
- **Parameterized Constructor:** A constructor with parameters is known as a parameterized constructor. The parameterized constructor takes its first argument as a reference to the instance being constructed known as `self` and the rest of the arguments are provided by the programmer.

The properties that all **Car** objects must have been defined in `.__init__()`. Every time a new **Car** object is created, `.__init__()` sets the initial state of the object by assigning the values of the object's properties. That is, `.__init__()` initializes each new instance of the class.
When a new class instance is created, the instance is automatically passed to the self parameter in `.__init__()` so that new attributes can be defined on the object.

The `self` Parameter

- The `self` parameter is a reference to the current instance of the class and is used to access variables that belong to the class.
- It does not have to be named `self`, you can call it whatever you like, but it has to be the first parameter of any function in the class.

- You can give `.__init__()` any number of parameters, but the first parameter will always be a variable called `self`.

Let's update the Car class with the `.__init__()` method that creates `name` and `topSpeed` attributes:

```
class Car:
    def __init__(self, name, topSpeed):
        self.name = name
        self.topSpeed= topSpeed
```

Note: The `.__init__()` method's signature is indented four spaces. The body of the method is indented by eight spaces. This indentation is vitally important. It tells Python that the `.__init__()` method belongs to the **Car** class.

In the body of `.__init__()`, two statements are using the self variable:

1. `self.name = name` creates an attribute called `name` and assigns to it the value of the `name` parameter.
2. `self.topSpeed= topSpeed` creates an attribute called `topSpeed` and assigns to it the value of the `topSpeed` parameter.

Instance Attributes

Attributes created in `.__init__()` are called **instance** attributes. An instance attribute's value is specific to a particular instance of the class. All **Car** objects have a `name` and a `topSpeed`, but the values for the `name` and `topSpeed` attributes will vary depending on the **Car** instance. Different objects of the **Car** class will have different names and top speeds.

Class Attributes

On the other hand, class attributes are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of `.__init__()`.

For example, the following **Car** class has a class attribute called `color` with the value `"Black"`:

```
class Car:
    # Class attribute
    color = "Black"

    def __init__(self, name, topSpeed):
        self.name = name
        self.topSpeed= topSpeed
```

- Class attributes are defined directly beneath the first line of the class name and are indented by four spaces.
- They must always be assigned an initial value.
- When an instance of the class is created, the class attributes are automatically created and assigned to their initial values.

You should use class attributes to define properties that should have the same value for every class instance and you must use instance attributes for properties that vary from one instance to another.

Now that we have a **Car** class, let's create some cars!

Instantiating an Object in Python

Creating a new object from a class is called instantiating an object. Consider the previous simpler version of our **Car** class:

```
>>> class Car()
...     pass
```

You can instantiate a new **Car** object by typing the name of the class, followed by opening and closing parentheses:

```
>>> Car()
<__main__.Car object at 0x106702d30>
```

You now have a new **Car** object at `0x106702d30`. This string of letters and numbers is a memory address that indicates where the **Car** object is stored in your computer's memory. Note that the address you see on your screen will be different.

Now, instantiate a second **Car** object:

```
>>> Car()
<__main__.Car object at 0x0004ccc90>
```

The new **Car** instance is located at a different memory address. That's because it's an entirely new instance and is completely different from the first **Car** object that you instantiated.

Consider the following code snippet:

```
>>> a = Car()
>>> b = Car()
>>> a == b
False
```

In this code, you create two new **Car** objects and assign them to the variables **a** and **b**. When you compare **a** and **b** using the **==** operator, the result is **False**. This is

because even though **a** and **b** are both instances of the **Car** class, they represent two distinct objects in memory.

Class and Instance Attributes

Now, consider the **Car** class we created with class attribute **color** and instance attributes `name` and `topSpeed`.

```
>>> class Car:
...     color = "Black"
...     def __init__(self, name, topSpeed):
...         self.name = name
...         self.topSpeed= topSpeed
```

To instantiate objects of this **Car** class, you need to provide values for the `name` and `topSpeed`. If you don't, then Python raises a `TypeError`.

```
>>> Car()
TypeError: __init__() missing 2 required positional arguments: 'name'
and 'topSpeed'
```

To pass arguments to the `name` and `topSpeed` parameters, put values into the parentheses after the class name.

```
>>> c1 = Car("Creta", 200)
>>> c1 = Car("Wagon R", 190)
```

This creates two new **Car** instances. Now, observe that the **Car** class's `.__init__()` method has three parameters, so why are only two arguments passed to it in the example?

The reason is that when you instantiate a **Car** object, Python creates a new instance and passes it to the first parameter of `.__init__()`. This essentially removes the `self` parameter, so you only need to worry about the `name` and `topSpeed` parameters.

After you create the **Car** instances, you can access their instance attributes using dot notation:

```
>>> c1.name
'Creta'
>>> c2.topSpeed
190
```

You can access class attributes the same way:

```
>>> c1.color
'Black'
```

One of the biggest advantages of using classes to organize data is that instances are guaranteed to have the attributes you expect. The values of these attributes ***can*** be changed dynamically:

```
>>> c1.topSpeed= 250
>>> c1.topSpeed
250
>>> c2.color = "Red"
>>> c2.color
'Red'
```

In this example, you change the `topSpeed` attribute of the `c1` object to **250**. Then you change the `color` attribute of the `c2` object to `"Red"`.

Note:

- The key takeaway here is such custom objects are **mutable** by default i.e. their states can be modified.
- Further, the value of the class attributes will be the same for all instances of that class **initially**. However, we can modify the values of these class attributes for individual objects.

- For instance, in the given example the value of **`color`** of **`c2`** object is now `"Red"`, however the value of **`color`** for **`c1`** is still `"Black"`.

Object-Oriented Programming (OOPS-2)

Inheritance

- Inheritance is a powerful feature in Object-Oriented Programming.
- Inheritance can be defined as the process where one class acquires the properties (methods and fields) of another. With the use of inheritance, the information is made manageable in a hierarchical order.
- The class which inherits the properties of the other is known as **subclass** *(derived class or child class)* and the class whose properties are inherited is known as **superclass** *(base class, parent class)*.

Let us take a real-life example to understand inheritance. Let's assume that **Human** is a class that has properties such as **height**, **weight**, **age**, etc and functionalities (or methods) such as **eating()**, **sleeping()**, **dreaming()**, **working()**, etc.

Now we want to create **Male** and **Female** classes. Both males and females are humans and they share some common properties (like **height**, **weight**, **age**, etc) and behaviors (or functionalities like **eating()**, **sleeping()**, etc), so they can inherit these properties and functionalities from the **Human** class. Both males and females also have some characteristics specific to them (like men have short hair and females have long hair). Such properties can be added to the **Male** and **Female** classes separately.

This approach makes us write less code as both the classes inherited several properties and functions from the superclass, thus we didn't have to re-write them. Also, this makes it easier to read the code.

Python Inheritance Syntax

```
class SuperClass:
    #Body of base class
class SubClass(BaseClass):
    #Body of derived class
```

The name of the superclass is passed as a parameter in the subclass while declaration.

Example of Inheritance in Python

To demonstrate the use of inheritance, let us take an example.

A polygon is a closed figure with 3 or more sides. Say, we have a class called Polygon defined as follows.

```
class Polygon:
    def __init__(self, no_of_sides): #Constructor
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    def inputSides(self): #Take user input for side lengths
       self.sides=[int(input("Enter side: "))for i in range(self.n)]

    def displaySides(self): #Print the sides of the polygon
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])
```

This class has **data attributes** to store the number of sides **n** and magnitude of each side as a list called **sides**.

The `inputSides()` method takes in the magnitude of each side and `dispSides()` displays these side lengths.

Now, a triangle is a polygon with 3 sides. So, we can create a class called **Triangle** which inherits from **Polygon**. In other words, we can say that every triangle is a polygon. This makes all the attributes of the **Polygon** class available to the **Triangle** class.

Constructor in Subclass

The constructor of the subclass must call the constructor of the superclass by accessing the **__init__** method of the superclass as:

```
<SuperClassName>.__init__(self,<Parameter1>,<Parameter2>,...)
```

Note: The parameters being passed in this call must be the same as the parameters being passed in the superclass' **__init__** function, otherwise it will throw an error.

The **Triangle** class can be defined as follows.

```
class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,3) #Calling constructor of superclass

    def findArea(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
```

However, the class **Triangle** has a new method `findArea()` to find and print the area of the triangle. This method is only specific to the **Triangle** class and not **Polygon** class. Here is a sample run:

```
>>> t = Triangle() #Instantiating a Triangle object
>>> t.inputSides()
Enter side 1 : 3
Enter side 2 : 5
Enter side 3 : 4
>>> t.dispSides()
Side 1 is 3
Side 2 is 5
Side 3 is 4
>>> t.findArea()
The area of the triangle is 6.00
```

We can see that even though we did not define methods like `inputSides()` or `displaySides()` for class Triangle separately, we were able to use them. If an attribute is not found in the subclass itself, the search continues to the superclass.

Access Modifiers

Various object-oriented languages like C++, Java, Python control access modifications which are used to restrict access to the variables and methods of the class. Most programming languages have three forms of access modifiers, which are **Public**, **Private,** and **Protected** in a class.

Public Modifier

The members of a class that are declared **public** are easily accessible from any part of the program. All data members and member functions of a class are **public** by default.

Consider the given example:

```
class Student:
    name = None # public member by default
    public age = None # public member

     # constructor
    def __init__(self, name, age):
        self.name = name
        self.age = age

obj = Student("Boy", 15)
print(obj.age)  #calling a public member of the class
print(obj.name) #calling a private member of the class
```

We will get the output as:

```
10
Boy
```

We will be able to access both **name** and **age** of the object from outside the class as they are **public**. However, this is not a good practice due to *security concerns.*

Private Modifier

The members of a class that are declared **private** are accessible within the class only. A private access modifier is the most secure access modifier. Data members of a class are declared private by adding a double underscore **'__'** symbol before the data member of that class. Consider the given example:

```
class Student:
    __name = None # private member
    age = None # public member

    def __init__(self, name, age):   # constructor
        self.__name = name
        self.age = age
obj = Student("Boy", 15)
print(obj.age)  #calling a public member of the class
print(obj.name) #calling a private member of the class
```

We will get the output as:

```
10
AttributeError: 'Student' object has no attribute 'name'
```

We will get an **AttributeError** when we try to access the **name** attribute. This is because **name** is a **private** attribute and hence it cannot be accessed from outside the class.

Note: *We can even have* ***public*** *and* ***private*** *methods.*

Private and Public modifiers with Inheritance

- The subclass will be able to access any **public** method or instance attribute of the superclass.
- The subclass will not be able to access any **private** method or instance attribute of the superclass.

Protected Modifier

The members of a class that are declared protected are only accessible to a class derived from it. Data members of a class are declared **protected** by adding a single underscore **'_'** symbol before the data member of that class.

The given example will help you get a better understanding:

```
# superclass
class Student:
     _name = None # protected data member
     # constructor
     def __init__(self, name):
          self._name = name
```

This is the parent class **Student** with a **protected** instance attribute **_name**. Now consider a subclass of this class:

```
class Display(Student):
       # constructor
       def __init__(self, name):
                Student.__init__(self, name)
       def displayDetails(self):
                # accessing protected data members of the superclass
                print("Name: ", self._name)

obj = Display("Boy") # creating objects of the derived class
obj.displayDetails() # calling public member functions of the class
obj.name  # trying to access protected attribute
```

This class **Display** inherits the **Student** class. The method `displayDetails()` accesses the **protected** attribute **_name**. Further, we try to access it again outside this class.

Output:

```
Name: Boy
AttributeError: 'Display' object has no attribute 'name'
```

You can observe that we were able to access the **protected** attribute **_name** from inside the `displayDetails()` method in the subclass. However, we were not able to access it outside the subclass and we got an **AttributeError**. This justifies the definition of the **protected** modifier.

Polymorphism

The literal meaning of polymorphism is the condition of occurrence in different forms. Polymorphism is a very important concept in programming. It refers to the use of a single type entity (method, operator, or object) to represent different types in different scenarios. Let's take a few examples:

Example 1: Polymorphism in addition(+) operator

We know that the **+** operator is used extensively in Python programs. But, it does not have a single usage. For integer data types, the **+** operator is used to perform arithmetic addition operation.

```
num1 = 1
num2 = 2
print(num1+num2)
```

Hence, the above program outputs **3**.

Similarly, for string data types, the **+** operator is used to perform concatenation.

```
str1 = "Python"
str2 = "Programming"
print(str1+" "+str2)
```

As a result, the above program outputs `"Python Programming"`.

Here, we can see that a single operator + has been used to carry out different operations for distinct data types. This is one of the most simple occurrences of **polymorphism** in Python.

Example 2: Functional Polymorphism in Python

There are some functions in Python which are compatible to run with multiple data types.

One such function is the `len()` function. It can run with many data types in Python. Let's look at some example use cases of the function.

```
print(len("abcdefgeh"))
print(len(["a", "b", "c"]))
print(len({"a": 1, "b": 2}))
```

Output

```
9
3
2
```

Here, we can see that many data types such as string, list, tuple, set, and dictionary can work with the `len()` function. However, we can see that it returns specific information(the length) about the specific data types.

Class Polymorphism in Python

Polymorphism is a very important concept in Object-Oriented Programming. We can use the concept of polymorphism while creating class methods as Python allows different classes to have methods with the same name.

We can then later generalize calling these methods by disregarding the object we are working with.

Let's look at an example:

Example 3: Polymorphism in Class Methods

```
class Male:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def info(self):
        print("Hi, I am Male")

class Female:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def info(self):
        print("Hi, I am Female")


M = Male("Sid", 20)
F = Female("Zee",21)

for human in (M, F): #Run loop over the set of objects
    human.info() #call the info function common to both
```

Output

```
Hi, I am Male
Hi, I am Female
```

Here, we have created two classes **Male** and **Female**. They share a similar structure and have the same method `info()`. However, notice that we have not created a common superclass or linked the classes together in any way. Even then, we can pack these two different objects into a tuple and iterate through them using a common **human** variable. It is possible due to **polymorphism**. We can call both the

`info()` methods by just using `human.info()` call, where **human** is first **M** (Instance of **Male**) and then **F** (Instance of **Female**).

Polymorphism and Inheritance

Like in other programming languages, the child classes in Python also inherit methods and attributes from the parent class. We can redefine certain methods and attributes specifically to fit the child class, which is known as **Method Overriding**.

Polymorphism allows us to access these overridden methods and attributes that have the same name as the parent class. Let's look at an example:

```
class Human:
    def __init__(self, name):
        self.name = name

    def info(self):
        print("Hi, I am Human")

class Male(Human):
    def __init__(self, name):
        super().__init__(name)

    def info(self):
        print("Hi, I am Male")

class Female(Human):
    def __init__(self, name):
        super().__init__(name)

    def info(self):
        print("Hi, I am Female")

M = Male("Sid")
F = Female("Zee")

for human in (M, F): #Run loop over the set of objects
```

```
    human.info() #call the info function common to both
```

Output

```
Hi, I am Male
Hi, I am Female
```

Due to polymorphism, the Python interpreter automatically recognizes that the `info()` method for object M (**Male** class) is **overridden**. So, it uses the one defined in the subclass **Male**. Same with the object F (**Female** Class).

Note: ***Method Overloading*****,** a way to create multiple methods with the same name but different arguments, is not possible in Python.

Multiple Inheritance

- A class can be inherited from more than one superclass in Python, similar to C++. This is called **multiple inheritance**.
- In multiple inheritance, the features of all the superclasses are inherited into the subclass. The syntax for multiple inheritance is similar to single inheritance.

Example:

```
class SuperClass1:
    pass
class SuperClass2:
    pass
class SubClass(SuperClass1, SuperClass2):
    pass
```

Here, the `SubClass` class is derived from `SuperClass1` and `SuperClass2` classes and it has access to all the instance attributes and methods from both these superclasses.

Multilevel Inheritance

We can also inherit from a derived class. This is called **multilevel inheritance**. It can be of any depth in Python.

An example is given below.

```
class SuperClass:
    pass
class SubClass(SuperClass):
    pass
class SubSubClass(SubClass):
    pass
```

Here, `SubClass` is derived from `SuperClass`, and `SubSubClass` is derived from `SubClass.`

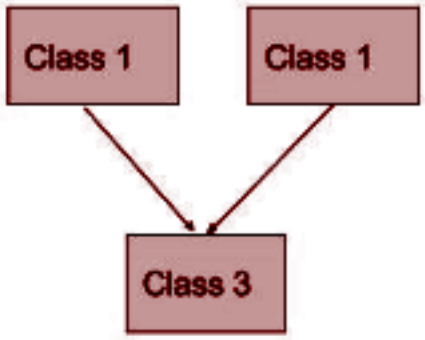


Fig: Mutiple Inheritance

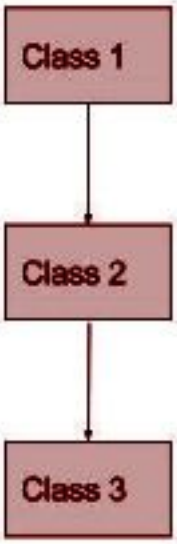


Fig: Multilevel Inheritance

Object Class

Every class in Python is directly or indirectly derived from a built-in class called the **Object** class. If a class does not inherit any other class then it is a direct child class of **Object** and if inherits some other class then it is indirectly derived. Therefore the **Object** class methods are available to all Python classes. There are three such methods provided by the **Object** class:

- **__new__() :** Used in instantiating a new object.
- **__init__():** Used for initialising a new object. We usually override this method while defining the constructors for any object. If we don't override it, the default constructor from the **Object** class gets called.
- **__str__() :** Returns a string representation of the object.

We can override all of these methods, but we usually override only **__init__()** and **__str__()**. Consider the given examples to understand it better:

```
class Circle(object):
    def __init__(self,radius):#Overriding the constructor
        self.radius = radius
c = Circle(4)
print(c) #Invokes the default __str__ method
```

Output:

```
<__main__.Circle object at 0x10794c850> #Some memory location
```

This is the output of the default **__str__** method. However, we can override it as follows:

```
class Circle(object):
    def __init__(self,radius):
        self.radius = radius
    def __str__(self): #Overriding the default __str__ method
        return "This is My circle"
c = Circle(4)
print(c)
```

Output:

```
This is My circle
```

Note: In Python 3.x, `class Circle(object)` and `class Circle` are the same.

Method Resolution Order (MRO)

Method Resolution Order(MRO) denotes the way a programming language resolves a method or an attribute. It defines the order in which the subclasses are searched when a method is called. First, the method or the attribute is searched within the subclass and then it is searched in its parent class and so on.

```
class A:
```

```
    def rk(self):
        print("In class A")
class B(A):
    def rk(self):
        print("In class B")
r = B()
r.rk()
```

Output:

```
In class B
```

In the above example, the method that is invoked is from class **B** but not from class **A**, and this is due to the Method Resolution Order(MRO). The order of priority that is followed in the above code is- **class B - > class A.**

In **multiple inheritances**, the methods are executed based on the **order specified while inheriting the classes** *(Order inside parenthesis)*. Let's look over another example to deeply understand the method resolution order:

```
class B:
    pass

class A:
    pass

class C(A, B):
    pass
```

The order in which the methods will be resolved will be **C, A, B**. This is because while inheriting the order is **A, B**.

Methods for Method Resolution Order(MRO) of a class:

To get the method resolution order of a class we can use the `mro()` method. By using this method we can display the order in which methods are resolved.

```
class B:
    pass

class A(B):
    pass

class C(A, B):
    pass

class D(C,A):
    pass

print(D.mro())
```

Output:

```
[<class '__main__.D'>, <class '__main__.C'>, <class '__main__.A'>, <class
'__main__.B'>, <class 'object'>]
```

Object-Oriented Programming (OOPS-3)

Abstract Classes

An abstract class can be considered as a blueprint for other classes. Abstract classes are classes that contain one or more abstract methods. An abstract method is a method that has a declaration but does not have an implementation. This set of methods must be created within any child classes which inherit from the abstract class. *A class that contains one or more abstract methods is called an* ***abstract class****.*

Creating Abstract Classes in Python

- By default, Python does not provide abstract classes.
- Python comes with a module that provides the base for defining Abstract Base classes(ABC) and that module name is **abc.**
- **abc** works by decorating methods of the base class as abstract and then registering concrete classes as implementations of the abstract base.
- A method becomes abstract when decorated with the keyword `@abstractmethod`.
- You are required to import **ABC** superclass and `abtractmethod` from the **abc** module before declaring your abstract class.
- An abstract class cannot be directly instantiated i.e. we cannot create an object of the abstract class.
- However, the subclasses of an abstract class that have definitions for all the abstract methods declared in the abstract class, can be instantiated.
- While declaring abstract methods in the class, it is not mandatory to use the `@abstractmethod` decorator (i.e it would not throw an exception). However, it is considered a good practice to use it as it notifies the compiler that the user has defined an abstract method.

The given Python code uses the **ABC** class and defines an abstract base class:

```
from abc import ABC, abstractmethod #Importing the ABC Module

class AbstractClass(ABC):

    def __init__(self, value): #Class Constructor
        self.value = value
        super().__init__()

    @abstractmethod
    def do_something(self): #Our abstract method declaration
        pass
```

Note:

- You are required to define (implement) all the abstract methods declared in an Abstract class, in all its subclasses to be able to instantiate the subclass.

For example, We will now define a subclass using the previously defined abstract class. You will notice that since we haven't implemented the `do_something` method, in this subclass, we will get an exception.

```
class TestClass(AbstractClass):
    pass #No definition for do_something method
x = TestClass(4)
```

We will get the output as:

```
TypeError: Can’t instantiate abstract class TestClass with abstract
methods do_something
```

We will do it the correct way in the following example, in which we define two classes inheriting from our abstract class:

```
class add(AbstractClass):
    def do_something(self):
        return self.value + 42

class mul(AbstractClass):
    def do_something(self):
        return self.value * 42

x = add(10)
y = mul(10)

print(x.do_something())
print(y.do_something())
```

We get the output as:

```
52
420
```

Thus, we can observe that a class that is derived from an abstract class cannot be instantiated unless all of its abstract methods are overridden.

Note: Concrete classes contain only concrete (normal) methods whereas abstract classes may contain both concrete methods and abstract methods.

- An abstract method can have an implementation in the abstract class.
- However, even if they are implemented, this implementation shall be overridden in the subclasses.
- If you wish to invoke the method definition from the abstract superclass, the abstract method can be invoked with `super()` call mechanism. (*Similar to cases of "normal" inheritance*).

- Similarly, we can even have concrete methods in the abstract class that can be invoked using `super()` call. Since these methods are not abstract it is not necessary to provide their implementation in the subclasses.
- Consider the given example:

```
from abc import ABC, abstractmethod

class AbstractClass(ABC):
    @abstractmethod
    def do_something(self): #Abstract Method
        print("Abstract Class AbstractMethod")

    def do_something2(self): #Concrete Method
        print("Abstract Class ConcreteMethod")

class AnotherSubclass(AbstractClass):
    def do_something(self):
        #Invoking the Abstract method from super class
        super().do_something()

    #No concrete method implementation in subclass
x = AnotherSubclass()
x.do_something() #Calling abstract method
x.do_something2() #Calling concrete method
```

We will get the output as:

```
Abstract Class AbstractMethod
Abstract Class ConcreteMethod
```

Another Example

The given code shows another implementation of an abstract class.

```
# Python program showing how an abstract class works
from abc import ABC, abstractmethod
class Animal(ABC): #Abstract Class
    @abstractmethod
    def move(self):
        pass

class Human(Animal): #Subclass 1
    def move(self):
        print("I can walk and run")

class Snake(Animal): #Subclass 2
    def move(self):
        print("I can crawl")

class Dog(Animal): #Subclass 3
    def move(self):
        print("I can bark")

# Driver code
R = Human()
R.move()
K = Snake()
K.move()
R = Dog()
R.move()
```

We will get the output as:

```
I can walk and run
I can crawl
I can bark
```

Exception Handling

Error in Python can be of two types i.e. Syntax errors and Exceptions.

- Errors are the problems in a program due to which the program will stop the execution.
- On the other hand, exceptions are raised when some internal events occur which changes the normal flow of the program.

Difference between Syntax Errors and Exceptions

Syntax Error: As the name suggests this error is caused by the wrong syntax in the code. It leads to the termination of the program.

Example:

Consider the given code snippet:

```
amount = 10000
if(amount>2999)
    print("Something")
```

We will get the output as:

```
SyntaxError: invalid syntax
```

The syntax error is because there should be a **":"** at the end of an **if** statement. Since it is not present, it gives a syntax error.

Exceptions: Exceptions are raised when the program is syntactically correct but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

Example:

```
marks = 10000
a = marks / 0
print(a)
```

Output:

```
ZeroDivisionError: division by zero
```

The above example raised the **ZeroDivisionError** exception, as we are trying to divide a number by 0 which is not defined.

Exceptions in Python

- Python has many built-in exceptions that are raised when your program encounters an error (something in the program goes wrong).
- When these exceptions occur, the Python interpreter stops the current process and passes it to the calling process until it is handled.
- If not handled, the program will crash.
- For example, let us consider a program where we have a function A that calls function B, which in turn calls function C. If an exception occurs in function C but is not handled in C, the exception passes to B and then to A.
- If never handled, an error message is displayed and the program comes to a sudden unexpected halt.

Some Common Exceptions

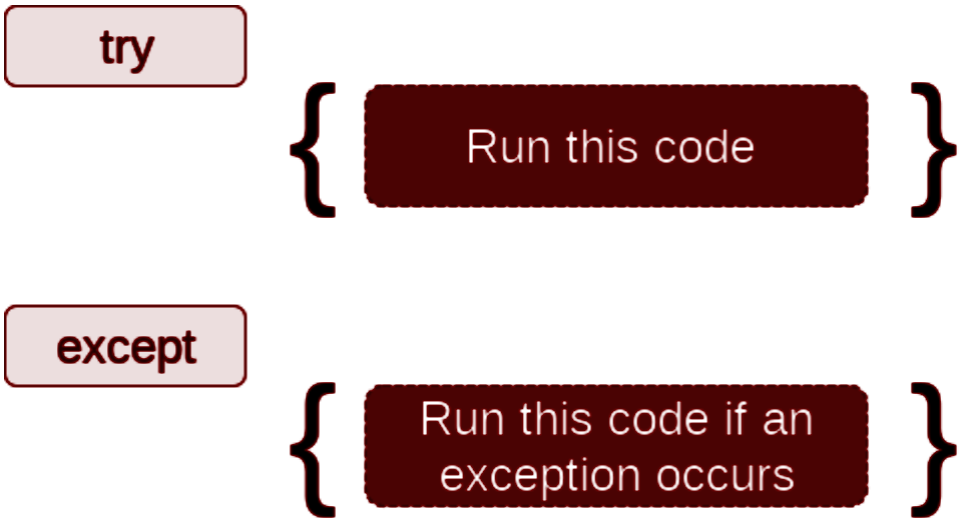
A list of common exceptions that can be thrown from a standard Python program is given below.

- **ZeroDivisionError:** This occurs when a number is divided by zero.
- **NameError:** It occurs when a *name* is not found. It may be local or global.
- **IndentationError:** It occurs when incorrect indentation is given.
- **IOError:** It occurs when an Input-Output operation fails.
- **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

Catching Exceptions

In Python, exceptions can be handled using `try-except` blocks.

- If the Python program contains suspicious code that may throw the exception, we must place that code in the `try` block.
- The `try` block must be followed by the `except` statement, which contains a block of code that will be executed in case there is some exception in the `try` block.
- We can thus choose what operations to perform once we have caught the exception.



- Here is a simple example:

```
# import module sys to get the type of exception
import sys
l = ['a', 0, 2]
for ele in l:
    try:#This block might raise an exception while executing
        print("The entry is", ele)
        r = 1/int(ele)
        break
    except:#This block executes in case of an exception in "try"
        print("Oops!", sys.exc_info()[0], "occurred.")
```

```
        print()

print("The reciprocal of", ele, "is", r)
```

We get the output to this code as:

```
The entry is a
Oops! <class 'ValueError'> occurred.

The entry is 0
Oops! <class 'ZeroDivisionError'> occured.

The entry is 2
The reciprocal of 2 is 0.5
```

- In this program, we loop through the values of a list `l`.
- As previously mentioned, the portion that can cause an exception is placed inside the try block.
- If no exception occurs, the except block is skipped and normal flow continues(for last value).
- But if any exception occurs, it is caught by the except block (first and second values).
- Here, we print the name of the exception using the `exc_info()` function inside `sys` module.
- We can see that element “a” causes ValueError and 0 causes ZeroDivisionError.

Every exception in Python inherits from the base **Exception** class. Thus we can write the above code as:

```
l = ['a', 0, 2]
for ele in l:
    try:
        print("The entry is", ele)
```

```
        r = 1/int(ele)
    except Exception as e: #Using Exception class
        print("Oops!", e.__class__, "occurred.")
        print("Next entry.")
        print()
print("The reciprocal of", ele, "is", r)
```

This program has the same output as the above program.

Catching Specific Exceptions in Python

- In the above example, we did not mention any specific exception in the `except` clause.
- This is not a good programming practice as it will catch all exceptions and handle every case in the same way.
- We can specify which exceptions an `except` clause should catch.
- A try clause can have any number of `except` clauses to handle different exceptions, however, only one will be executed in case an exception occurs.
- You can use multiple `except` blocks for different types of exceptions.
- We can even use a tuple of values to specify multiple exceptions in an `except` clause. Here is an example to understand this better:

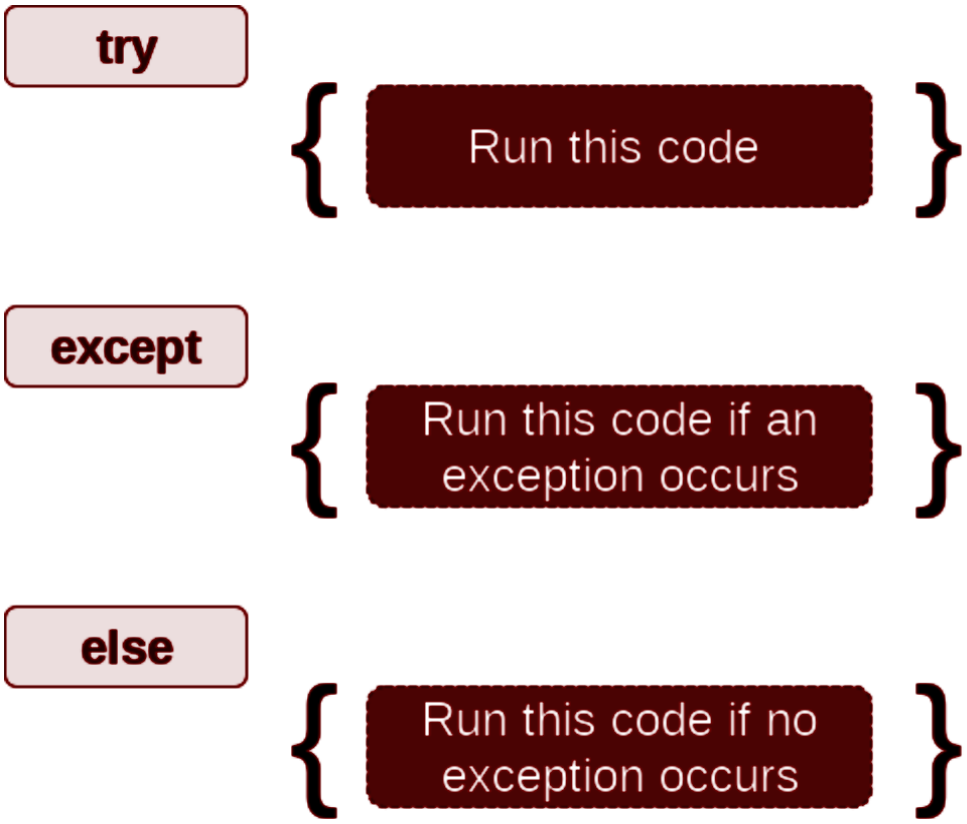
```
try:
    a=10/0;
except(ArithmeticError, IOError):
    print("Arithmetic Exception")
else:
    print("Successfully Done")
```

Output:

```
Arithmetic Exception
```

`try-except-else` Statements

We can also use the else statement with the try-except statement in which, we can place the code which will be executed in the scenario if no exception occurs in the `else` block. The syntax is given below:



Consider the example code to understand this better:

```
try:
    c = 2/1
```

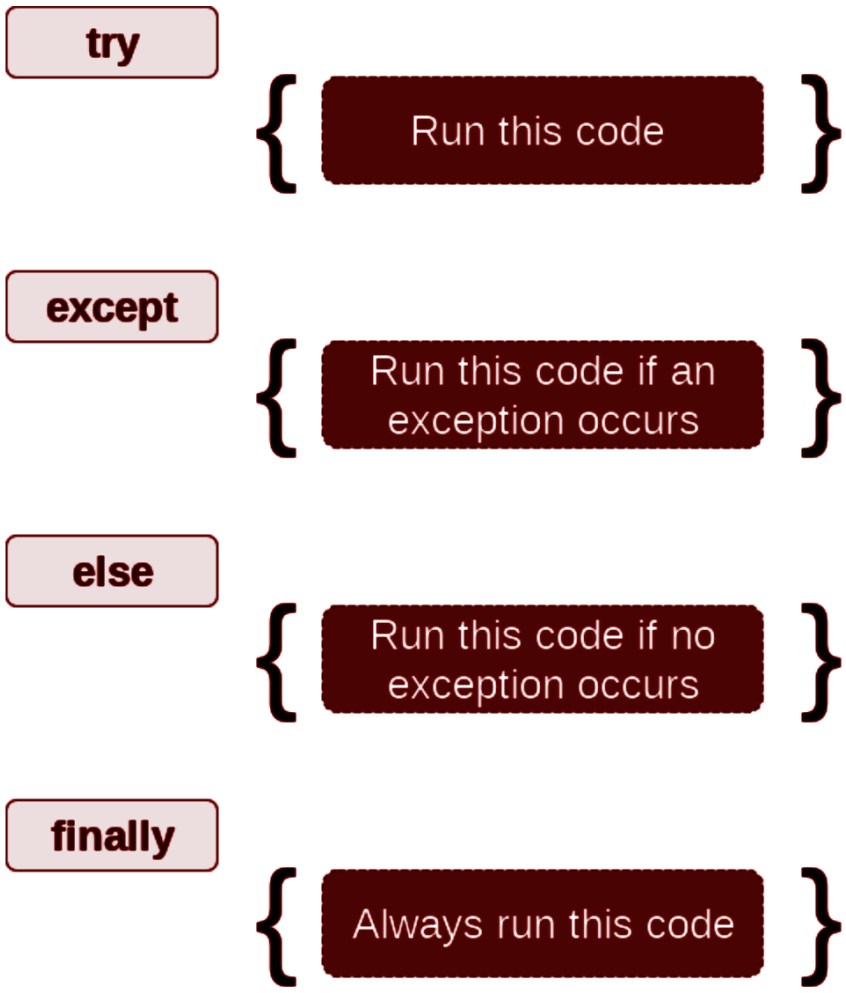
```
except Exception as e:
    print("can't divide by zero")
    print(e)
else:
    print("Hi I am else block")
```

Output:

```
Hi I am else block
```

We get this output because there is no exception in the try block and hence the else block is executed. If there was an exception in the `try` block, the `else` block will be skipped and `except` block will be executed.

`finally` Statement



The `try` statement in Python can have an optional `finally` clause. This clause is executed no matter what and is generally used to release external resources. Here is an example of file operations to illustrate this:

```
try:
   f = open("test.txt",encoding = 'utf-8')
   # perform file operations
finally:
   f.close()
```

This type of construct makes sure that the file is closed even if an exception occurs during the program execution.

Raising Exceptions in Python

In Python programming, exceptions are raised when errors occur at runtime. We can also manually raise exceptions using the `raise` keyword. We can optionally pass values to the exception to clarify why that exception was raised. Given below are some examples to help you understand this better

```
>>> raise KeyboardInterrupt
Traceback (most recent call last):
...
KeyboardInterrupt
```

```
>>> raise MemoryError("This is an argument")
Traceback (most recent call last):
...
MemoryError: This is an argument
```

Now, consider the given code snippet:

```
try:
```

```
    a = -2
    if a <= 0:
        raise ValueError("That is not a positive number!")
except ValueError as ve:
    print(ve)
```

The output of the above code will be:

```
That is not a positive number!
```

Time Complexity Analysis

Introduction

An important question while programming is: How efficient is an algorithm or a piece of code?

Efficiency covers a lot of resources, including:

- CPU (time) usage
- Memory usage
- Disk usage
- Network usage

All are important but we are mostly concerned about the **CPU time**. Be careful to differentiate between:

1. Performance: How much time/memory/disk/etc. is used when a program is run. This depends on the machine, compiler, etc. as well as the code we write.

2. Complexity: How do the resource requirements of a program or algorithm scale, i.e. what happens as the size of the problem being solved by the code gets larger.

Note: *Complexity affects performance but not vice-versa.*

Algorithm Analysis

Algorithm analysis is an important part of computational complexity theory, which provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem. Analysis of algorithms is the determination of the amount of time and space resources required to execute it.

Why Analysis of Algorithms?

- To predict the behavior of an algorithm without implementing it on a specific computer.
- It is much more convenient to have simple measures for the efficiency of an algorithm than to implement the algorithm and test the efficiency every time a certain parameter in the underlying computer system. changes.
- It is impossible to predict the exact behavior of an algorithm. There are too many influencing factors.
- The analysis is thus only an approximation; it is not perfect.
- More importantly, by analyzing different algorithms, we can compare them to determine the best one for our purpose.

Types of Analysis

To analyze a given algorithm, we need to know, with which inputs the algorithm takes less time (i.e. the algorithm performs well) and with which inputs the algorithm takes a long time.
Three types of analysis are generally performed:
• **Worst-Case Analysis:** The worst-case consists of the input for which the algorithm takes the longest time to complete its execution.
• **Best Case Analysis:** The best case consists of the input for which the algorithm takes the least time to complete its execution.
• **Average case:** The average case gives an idea about the average running time of the given algorithm.

There are two main complexity measures of the efficiency of an algorithm:

- **Time complexity** is a function describing the amount of time an algorithm takes in terms of the amount of input to the algorithm.
- **Space complexity** is a function describing the amount of memory (space) an algorithm takes in terms of the amount of input to the algorithm.

Big-O notation

We can express algorithmic complexity using the **big-O** notation. For a problem of size N:

- A constant-time function/method is *"order 1"*: **O(1)**
- A linear-time function/method is *"order N"*: **O(N)**
- A quadratic-time function/method is *"order N squared"*: **O(N^2)**

Definition: Let g and f be functions from the set of natural numbers to itself. The function f is said to be **O(g)** (read big-oh of g), if there is a constant **c** and a natural **n_0** such that **f (n) ≤ cg(n)** for all **n > n_0.**

Note: O(g) is a set!

Abuse of notation: f = O(g) does not mean f ∈ O(g).

Examples:

- **$5n^2 + 15 = O(n^2)$**, since **$5n^2 + 15 ≤ 6n^2$**, for all **n > 4**.
- **5n2 + 15 = O(n^3)**, since **$5n^2 + 15 ≤ n^3$**, for all **n > 6**.
- **O(1)** denotes a constant.

Although we can include constants within the big-O notation, there is no reason to do that. Thus, we can write **O(5n + 4) = O(n)**.

Note: The **big-O** expressions do not have constants or low-order terms. This is because, when N gets large enough, constants and low-order terms don't matter (a constant-time function/method will be faster than a linear-time function/method, which will be faster than a quadratic-time function/method).

Determining Time Complexities Theoretically

In general, how can you determine the running time of a piece of code? The answer is that it depends on what kinds of statements are used.

1. Sequence of statements

```
statement 1;
statement 2;
...
statement k;
```

The total time is found by adding the times for all statements:

```
totalTime = time(statement1) + time(statement2) +..+ time(statementk)
```

2. `if-else` statements

```
if (condition):
    #sequence of statements 1
else:
    #sequence of statements 2
```

Here, either **sequence 1** will execute, or **sequence 2** will execute. Therefore, the worst-case time is the slowest of the two possibilities:

```
max(time(sequence 1), time(sequence 2))
```

For example, if sequence 1 is O(N) and sequence 2 is O(1) the worst-case time for the whole if-then-else statement would be **O(N)**.

3. `for` loops

```
for i in range N:
    #sequence of statements
```

Here, the loop executes **N** times, so the sequence of statements also executes **N** times. Now, assume that all the statements are of the order of **O(1)**, then the total time for the **for** loop is **N * O(1)**, which is **O(N)** overall.

4. Nested loops

```
for i in range N:
    for i in range M:
        #statements
```

The outer loop executes **N** times. Every time the outer loop executes, the inner loop executes **M** times. As a result, the statements in the inner loop execute a total of **N * M** times. Assuming the complexity of the statement inside the inner loop to be **O(1)**, the overall complexity will be **O(N * M).**

Sample Problem:

What will be the Time Complexity of following while loop in terms of 'N' ?

```
while N>0:
    N = N//8
```

We can write the iterations as:

Iteration Number	Value of N
1	N
2	N//8
3	N//64
...	...
k	$N//8^k$

We know, that in the last i.e. the k^{th} iteration, the value of **N** would become **1**, thus, we can write:

```
N//8^k = 1
=> N = 8^k
=> log(N) = log(8^k)
=> k*log(8) = log(N)
=> k = log(N)/log(8)
=> k = log_8(N)
```

Now, clearly the number of iterations in this example is coming out to be of the order of $\log_8(N)$. Thus, the time complexity of the above while loop will be $O(\log_8(N))$.

Qualitatively, we can say that after every iteration, we divide the given number by 8, and we go on dividing like that, till the number remains greater than 0. This gives the number of iterations as $O(\log_8(N))$.

Time Complexity Analysis of Some Common Algorithms

Linear Search

Linear Search time complexity analysis is done below-
Best case- In the best possible case:

- The element being searched will be found in the first position.
- In this case, the search terminates in success with just one comparison.
- Thus in the best case, the linear search algorithm takes **O(1)** operations.

Worst Case- In the worst possible case:

- The element being searched may be present in the last position or may not present in the array at all.
- In the former case, the search terminates in success with **N** comparisons.
- In the latter case, the search terminates in failure with **N** comparisons.
- Thus in the worst case, the linear search algorithm takes **O(N)** operations.

Binary Search

Binary Search time complexity analysis is done below-

- In each iteration or each recursive call, the search gets reduced to half of the array.
- So for **N** elements in the array, there are **$\log_2 N$** iterations or recursive calls.

Thus, we have-

- Time Complexity of the Binary Search Algorithm is **$O(\log_2 N)$**.
- Here, **N** is the number of elements in the sorted linear array.

This time complexity of binary search remains unchanged irrespective of the element position even if it is not present in the array.

Big-O Notation Practice Examples

Example-1 Find upper bound for **$f(n) = 3n + 8$**

Solution: $3n + 8 \leq 4n$, for all $n \geq 8$

$\therefore 3n + 8 = O(n)$ with $c = 4$ and $n_0 = 8$

Example-2 Find upper bound for **$f(n) = n^2 + 1$**

Solution: $n^2 + 1 \leq 2n^2$, for all $n \geq 1$

$\therefore n^2 + 1 = O(n^2)$ with $c = 2$ and $n_0 = 1$

Example-3 Find upper bound for **$f(n) = n^4 + 100n^2 + 50$**

Solution: $n^4 + 100n^2 + 50 \leq 2n^4$, for all $n \geq 11$

$\therefore n^4 + 100n^2 + 50 = O(n^4)$ with $c = 2$ and $n_0 = 11$

Space Complexity Analysis

Introduction

- The space complexity of an algorithm represents the amount of extra memory space needed by the algorithm in its life cycle.
- Space needed by an algorithm is equal to the sum of the following two components:
    - A fixed part is a space required to store certain data and variables (i.e. simple variables and constants, program size, etc.), that are not dependent on the size of the problem.
    - A variable part is a space required by variables, whose size is dependent on the size of the problem. For example, recursion stack space, dynamic memory allocation, etc.
- Space complexity **S(p)** of any algorithm **p** is **S(p) = A + Sp(I)** Where **A** is treated as the fixed part and **S(I)** is treated as the variable part of the algorithm which depends on instance characteristic **I**.

Note: It's necessary to mention that space complexity depends on a variety of things such as the programming language, the compiler, or even the machine running the algorithm.
To get warmed up, let's consider a simple operation that sums two integers (numbers without a fractional part):

```
def difference(a, b):
    return a + b
```

In this particular method, three variables are used and allocated in memory:

The first integer argument, a; the second integer argument, b; and the returned sum which is also an integer.
In Python, these three variables point to three different memory locations. We can see that the space complexity is constant, so it can be expressed in big-O notation as **O(1)**.
Next, let's determine the space complexity of a program that sums all integer elements in an array:

```
def sumArray(array):
    size = 0
    sum = 0
    for iterator in range(size):
        sum += array[iterator]
    return sum
```

Again, let's list all variables present in the above code:

- **array**
- **size**
- **sum**
- **iterator**

The space complexity of this code snippet is **O(n),** which comes from the reference to the array that was passed to the function as an argument.
Let us now analyze the space complexity for a few common sorting algorithms. This will give you deeper insight into complexity analysis.

Quick-Sort Space Complexity Analysis

Let us consider the various scenarios possible :
Best case scenario: The best-case scenario occurs when the partitions are as evenly balanced as possible, i.e their sizes on either side of the pivot element are either equal or have a size difference of 1 of each other.

- **Case 1:** The case when the sizes of the sublist on either side of the pivot become equal occurs when the subarray has an odd number of elements and the pivot is right in the middle after partitioning. Each partition will have **(n-1)/2** elements.
- **Case 2:** The size difference of 1 between the two sublists on either side of pivot happens if the subarray has an even number, **n**, of elements. One partition will have **n/2** elements with the other having **(n/2)-1**.
- In either of these cases, each partition will have at most **n/2** elements, and the tree representation of the subproblem sizes will be as below:

Worst case scenario:

This happens when we encounter the most unbalanced partitions possible, then the original call takes place **n** times, the recursive call on **n-1** elements will take place **(n-1)** times, the recursive call on **(n-2)** elements will take place **(n-2)** times, and so on.

Based on the above-mentioned cases we can conclude that:

- The space complexity is calculated based on the space used in the recursion stack. The worst-case space used will be **O(n)**.
- The average case space used will be of the order **O(log n).**
- The worst-case space complexity becomes **O(n)** when the algorithm encounters its worst-case when we need to make **n** recursive call for getting a sorted list.

Practice Problems

Problem 1: What is the time & space complexity of the following code:

```
a = 0
b = 0
for i in range(n):
    a = a + i

for j in range(m):
    b = b + j
```

Problem 2: What is the time & space complexity of the following code:

```
a = 0
b = 0
for i in range(n):
    for j in range(n):
        a = a + j

for k in range(n):
    b = b + k
```

Problem 3: What is the time and space complexity of the following code:

```
a = 0
b = 0
for i in range(n):
    j = n
    while j>i:
        a = a + i + j
        j=j-1
```

Linked-List 1

Data Structures

Data structures are just a way to store and organize our data so that it can be used and retrieved as per our requirements. Using these can affect the efficiency of our algorithms to a greater extent. There are many data structures that we will be going through throughout the course, linked-lists are a part of them.

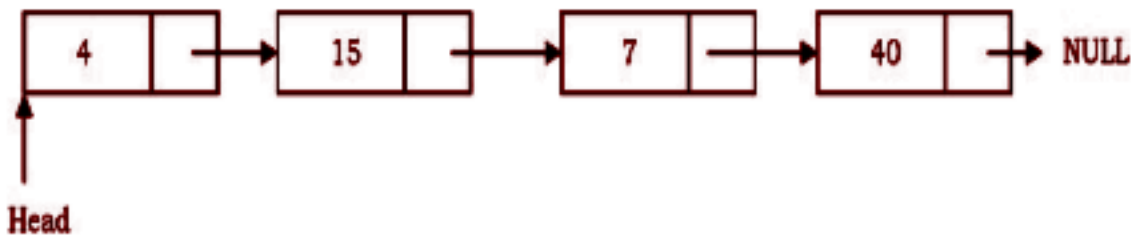
Introduction

A **Linked List** is a data structure used for storing collections of data. A linked list has the following properties:

- Successive elements are connected by pointers.
- Can grow or shrink in size during the execution of a program.
- Can be made just as long as required (until systems memory exhausts).
- Does not waste memory space (but takes some extra memory for pointers). It allocates memory as the list grows.

Basic Properties:

- Each element or node of a list is comprising of two items:
    - Data
    - Pointer(reference) to the next node.
- In a Linked List, the elements are not stored at contiguous memory locations.
- The first node of a linked list is known as **Head**.
- The last node of a linked list is known as **Tail.**
- The last node has a reference to **None**.



None

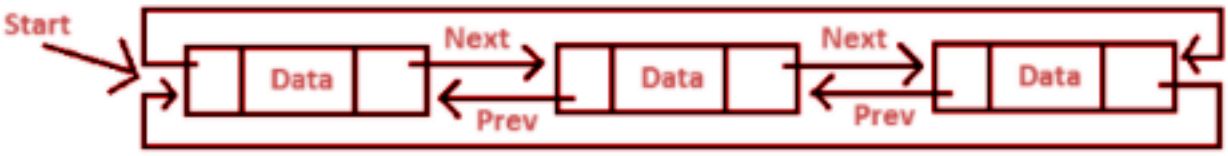
Types of A Linked List

- **Singly-Linked List:** Generally “linked list” means a singly linked list. Each node contains only one link which points to the subsequent node in the list.

- **Doubly-Linked List:** It’s a two-way linked list as each node points not only to the next pointer but also to the previous pointer.

- **Circular-Linked List:** There is no tail node i.e., the next field is never **None** and the next field for the last node points to the head node.

- **Circular Doubly-Linked List:** Combination of both Doubly linked list and circular linked list.



Node Class (Singly Linked List)

```
# Node class
class Node:
    # Function to initialize the node object
    def __init__(self, data):
        self.data = data  # Data that the node contains
        self.next = None  # Next node that this node points to
```

Note: The first node in the linked list is known as **Head** pointer and the last node is referenced as **Tail** pointer. We must never lose the address of the head pointer as it references the starting address of the linked list and if lost, would lead to loss of the list.

Traversing the Linked List

Let us assume that the head points to the first node of the list. To traverse the list we do the following:

- Follow the pointers.
- Display the contents of the nodes (or count) as they are traversed.
- Stop when the next pointer points to **None**.

Printing the Linked List

To print the linked list, we will start traversing the list from the beginning of the list(head) until we reach the **None** pointer which will always be the tail pointer. Let us add a new function `printList()` to our **LinkedList** class.

```
# This function prints contents of linked list starting from head
def printList(headNode):
    temp = headNode #Start from the head of the list
    while (temp): #Till we reach the last node when temp points to None
        print (temp.data)
        temp = temp.next #Update temp to point to the next Node
```

Insertion of A Node in a Singly Linked List

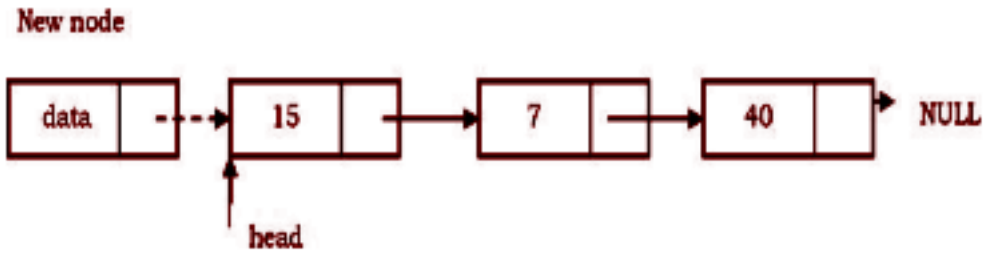
There are 3 possible cases:

- Inserting a new node before the head (at the beginning).
- Inserting a new node after the tail (at the end of the list).
- Inserting a new node in the middle of the list (random location).

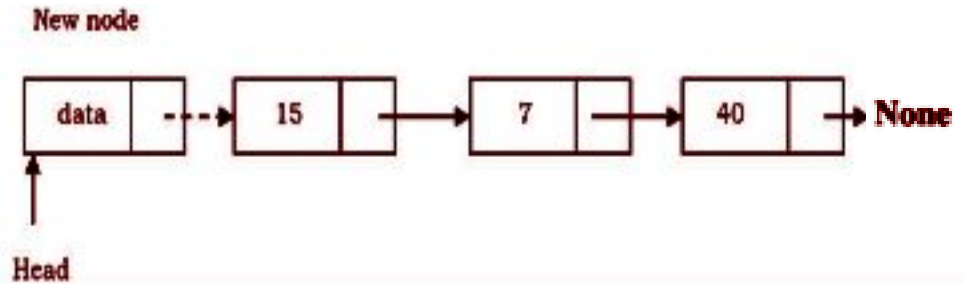
Case 1: Insert node at the beginning:

In this case, a new node is inserted before the current head node. Only one next pointer needs to be modified (new node's next pointer) and it can be done in two steps:

• Create a new node. Update the **next** pointer of the new node, to point to the current head.



• Update **head** pointer to point to the new node.



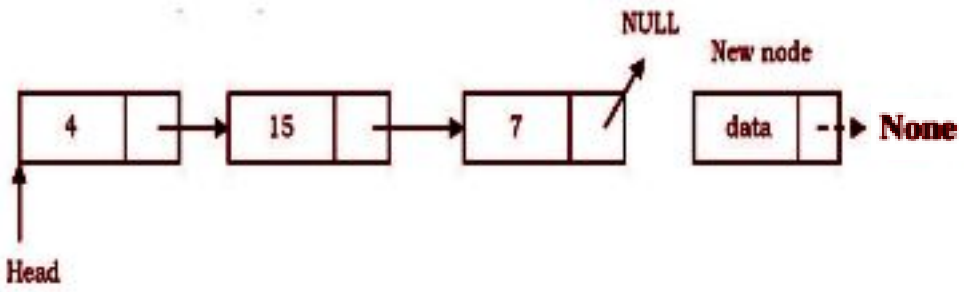
Python Code:

```
def insertAtStart(head, data):
    newNode = Node(data) #Create a new node
    newNode.next = head #Set next node of new node to current head
    head= newNode #Update the head pointer to the new node
```

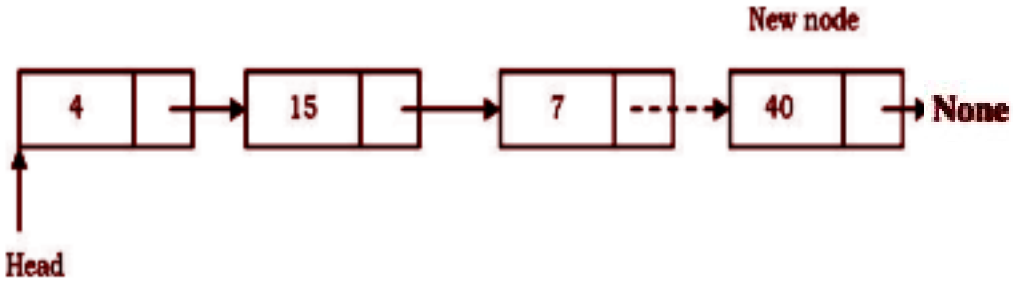
Case 2: Insert node at the ending:

In this case, we need to modify two next pointers (last nodes next pointer and new nodes next pointer).

• New node's **next** pointer points to **None**.



• Last node's **next** pointer points to the new node.



Python Code:

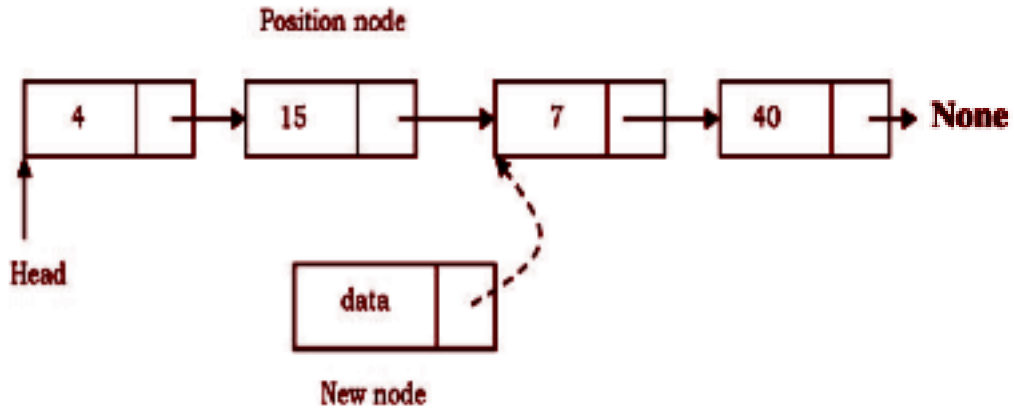
```
def insertAtEnd(head, data):
        newNode= Node(data)#Create a new node
        if head is None: #Incase of empty LL
            head = newNode
            return
        n = head
        while n.next is not None:#If not empty traverse till last node
            n= n.next
        n.next = newNode #Set next = new node
```

Case 3: Insert node anywhere in the middle: (At any specified Index)

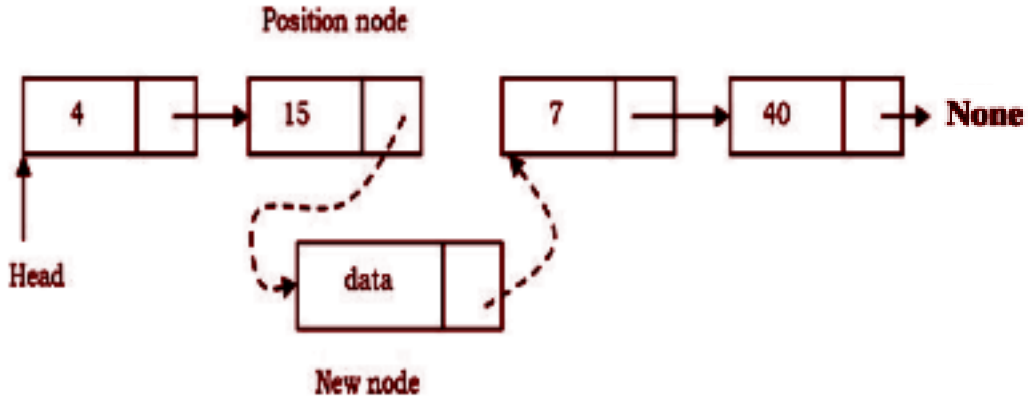
Let us assume that we are given a position where we want to insert the new node. In this case, also, we need to modify two next pointers.

• If we want to add an element at position 3 then we stop at position 2. That means we traverse 2 nodes and insert the new node.

• For simplicity let us assume that the second node is called the **position node**. The new node points to the next node of the position where we want to add this node.



• Position node's **next** pointer now points to the new node.



Python Code:

```
def insertAtIndex (head, index, data):
        if index == 1: #Insert at beginning
            insertAtStart(head, data)
        i = 1
        n = head
        while i < index-1 and n is not None:
            n = n.next
            i = i+1
        if n is None:
            print("Index out of bound")
        else:
            newNode = Node(data)
            newNode.next = n.next
            n.next = newNode
```

Deletion of A Node in a Singly Linked List

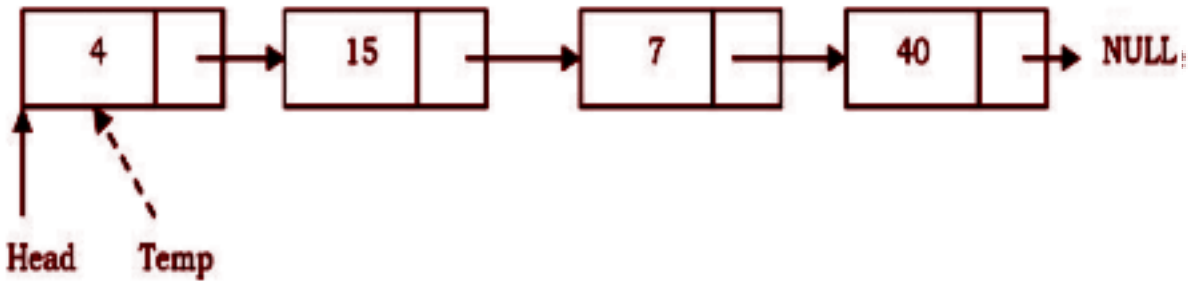
Similar to insertion, here we also have three cases.

- Deleting the first node
- Deleting the last node
- Deleting an intermediate node.

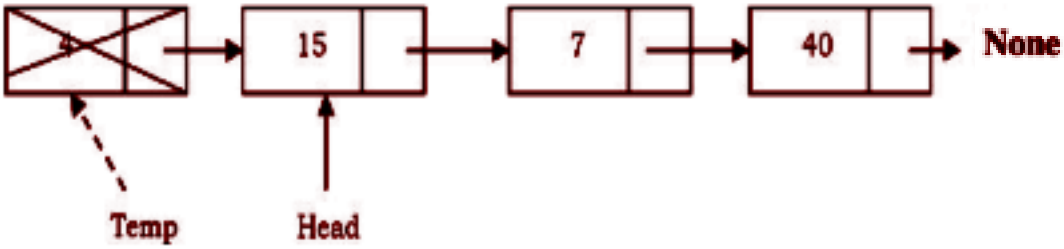
Deleting the First Node in Singly Linked List

It can be done in two steps:

• Create a temporary node which will point to the same node as that of **head**.



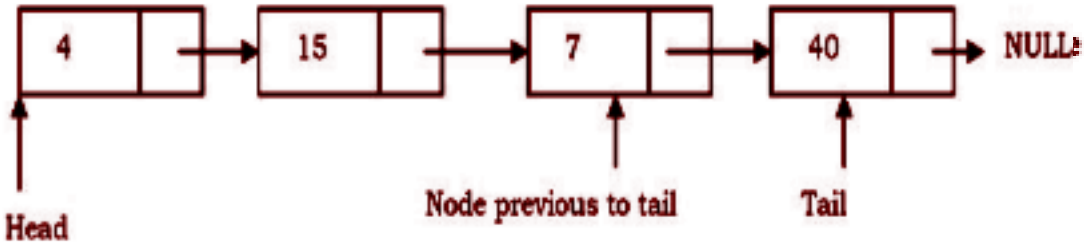
• Now, move the head nodes pointer to the next node and dispose of the temporary node.



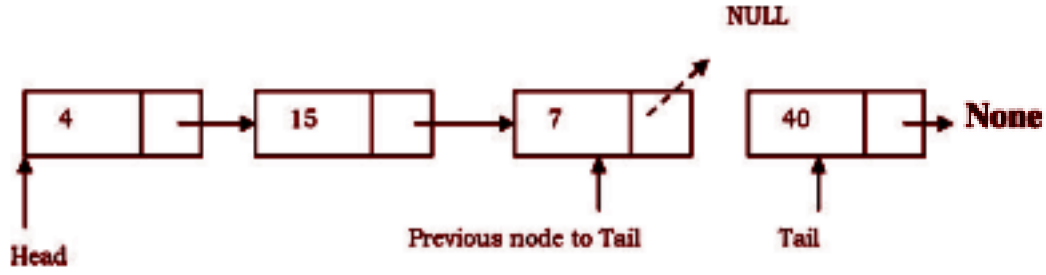
Deleting the Last Node in Singly Linked List

In this case, the last node is removed from the list. This operation is a bit trickier than removing the first node because the algorithm should find a node, which is previous to the tail. It can be done in three steps:

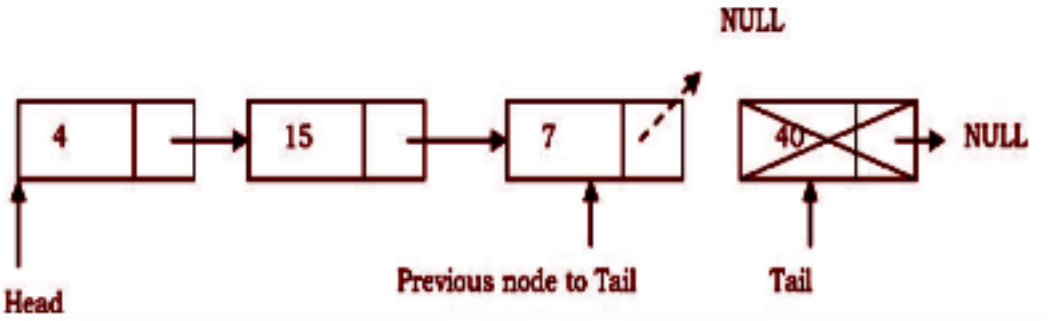
• Traverse the list and while traversing maintain the previous node address also. By the time we reach the end of the list, we will have two pointers, one pointing to the tail node and the other pointing to the node before the tail node.



• Update the previous node's next pointer with **None**.



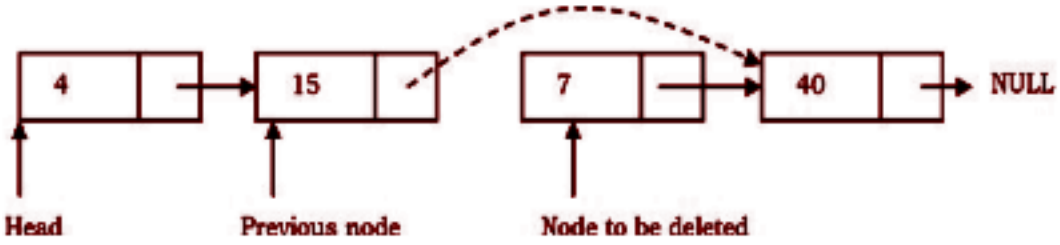
• Dispose of the tail node.



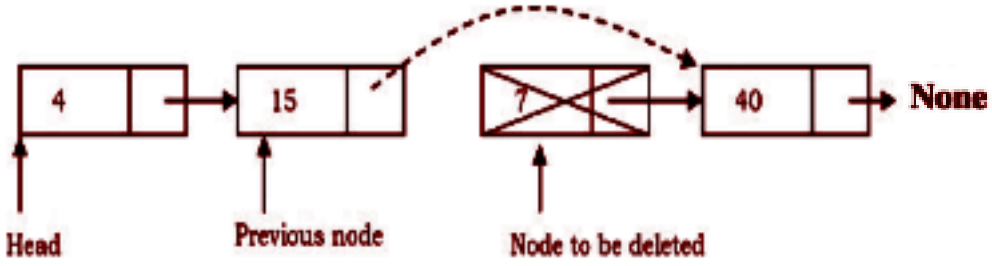
Deleting an Intermediate Node in Singly Linked List

In this case, the node to be removed is always located between two nodes. Head and tail links are not updated in this case. Such removal can be done in two steps:

• Similar to the previous case, maintain the previous node while traversing the list. Once we find the node to be deleted, change the previous node's next pointer to the next pointer of the node to be deleted.



• Dispose of the current node to be deleted.



Insert node recursively

Follow the steps below and try to implement it yourselves:

- If **Head** is **None** and **position** is not 0. Then exit it.
- If **Head** is **None** and **position** is 0. Then insert a new Node to the **Head** and exit it.
- If **Head** is not **None** and **position** is 0. Then the **Head** reference set to the new Node. Finally, a new Node set to the Head and exit it.
- If not, iterate until finding the Nth position or **end**.

For the code, refer to the Solution section of the problem.

Delete node recursively

Follow the steps below and try to implement it yourselves:

- If the node to be deleted is the **root**, simply delete it.
- To delete a middle node, we must have a pointer to the node previous to the node to be deleted. So if the position is not zero, we run a loop position-1 times and get a pointer to the previous node.
- Now, simply point the previous node's next to the current node's next and delete the current node.

For the code, refer to the Solution section of the problem.

Linked-List 2

Now moving further with the topic, let's try to solve some problems now...

The Midpoint of A Linked List

The Trivial Approach- Two Passes

- This approach requires us to traverse through the linked list twice i.e. 2 passes.
- In the first pass, we will calculate the **length** of the linked list. After every iteration, update the **length** variable.
- In the second pass, we will find the element of the linked list at the **(length-1)/2th** position. This element shall be the middle element in the linked list.
- However, we wish to traverse the linked list only once, therefore let us see another approach.

The Optimal Approach- One Pass

- The midpoint of a linked list can be found out very easily by taking two pointers, one named **slow** and the other named **fast**.
- As their names suggest, they will move in the same way respectively.
- The **fast** pointer will move ahead **two pointers at a time**, while the **slow** pointer one will move at a speed of **a pointer at a time**.
- In this way, when the fast pointer will reach the end, by that time the slow pointer will be at the middle position of the array.
- These pointers will be updated like this:
    - `slow = slow.next`
    - `fast = fast.next.next`

Python Code

```
    def printMiddle(headNode):
        if headNode is None or headNode.next is None:
            return head
        slow = headNode #Slow pointer
        fast = headNode.next #Fast Pointer
        while (fast is not None and fast.next is not None):
            fast = fast.next.next
            slow = slow.next
        return slow #Slow pointer shall point to our middle element
```

Note:

- For odd length there will be only one middle element, but for the even length there will be two middle elements.
- In case of an even length LL, both these approaches, will return the first middle element and the other one will be the direct **next** of the first middle element.

Merge Two sorted linked lists

- We will be merging the linked list, similar to the way we performed merge over two sorted arrays.
- We will be using the two **head** pointers, compare their data and the one found smaller will be directed to the new linked list, and increase the **head** pointer of the corresponding linked list.
- Just remember to maintain the **head** pointer separately for the new sorted list.
- And also if one of the linked list's length ends and the other one's not, then the remaining linked list will directly be appended to the final list.
- Try to implement this approach on your own.

Mergesort over a linked list

- Like the merge sort algorithm is applied over the arrays, the same way we will be applying it over the linked list.
- Just the difference is that in the case of arrays, the middle element could be easily figured out, but here you have to find the middle element, each time you send the linked list to split into two halves using the above approach.
- The merging part of the divided lists can also be done using the **merge sorted linked lists code** as discussed above.
- The functionalities of this code have already been implemented by you, just use them directly in your functions at the specified places.
- Try to implement this approach on your own.

Reverse the linked list

Recursive approach:

- In this approach, we will store the last element of the list in the small answer, and then update that by adding the next last node and so on.
- Finally, when we will be reaching the first element, we will assign the **next** to NONE.
- Follow the Python code below, for better understanding.

```
def reverseLinkedList(head) :
    if head is None or head.next is None
        return head

    smallHead = reverseLinkedList(head.next)
    head.next.next head
    head.next = None

    return smallHead
```

After calculation, you can see that this code has a time complexity of **O(n^2)**. Now let's think about how to improve it.

Recursive approach (Optimal):

- There is another recursive approach to the order of **O(n)**.
- What we will be doing is that head but also the tail pointer, which can save our time in searching over the list to figure out the tail pointer for appending or removing.
- Check out the given code for your reference:

```
def reverse2(head):
      if head is None or head.next is None:
            return head, head

      smallHead,smallTail = reverse2(head.next)
      smallTail.next= head
      head.next = None
      return smallHead,head
```

Now let us try to improve this code further.

A simple observation is that the **tail** is always **head.next**. By making the recursive call we can directly use this as our **tail** pointer and reverse the linked list by **tail.next = head**. Refer to the code below, for better understanding.

```
def reverse3(head):
      if head is None or head.next is None:
            return head

      smallHead = reverse3(head.next)
      tail = head.next
      tail.next = head
      head.next = None
      return smallHead
```

Iterative approach:

- We will be using three-pointers in this approach: **previous, current,** and **next.**
- Initially, the **previous** pointer would be **NONE** as in the reversed linked list, we want the original head to be the last element pointing to **NONE**.
- The **current** pointer will point to the current node whose **next** will be pointing to the previous element but before pointing it to the previous element, we need to store the next element's address somewhere otherwise we will lose that element.
- Similarly, iteratively, we will keep updating the pointers as **current** to the **next**, **previous** to the **current,** and **next** to **current**'s **next**.

Refer to the given Python code for better understanding:

```
def reverse(headNode):
    if headNode==None:
        return None
    elif headNode.next==None:
        return headNode

    prev=headNode
    curr=prev.next
    next=curr.next

    prev.next None
    while next:
        curr.next = prev
        prev = curr
        curr = next
        next = next.next
    curr.next = prev
    return curr
```

Practice problems

Try over the following link to practice some good questions related to linked lists:

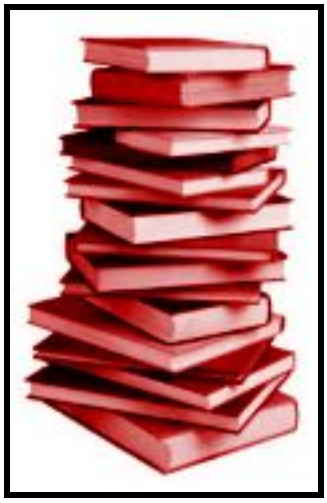
https://www.hackerrank.com/domains/data-structures?filters%5Bsubdomains%5D%5B%

5D=linked-lists

Stacks

Introduction

- Stacks are simple data structures that allow us to store and retrieve data sequentially.
- A stack is a linear data structure like arrays and linked lists.
- It is an abstract data type**(ADT)**.
- In a stack, the order in which the data arrives is essential. It follows the LIFO order of data insertion/abstraction. LIFO stands for **Last In First Out.**
- Consider the example of a pile of books:

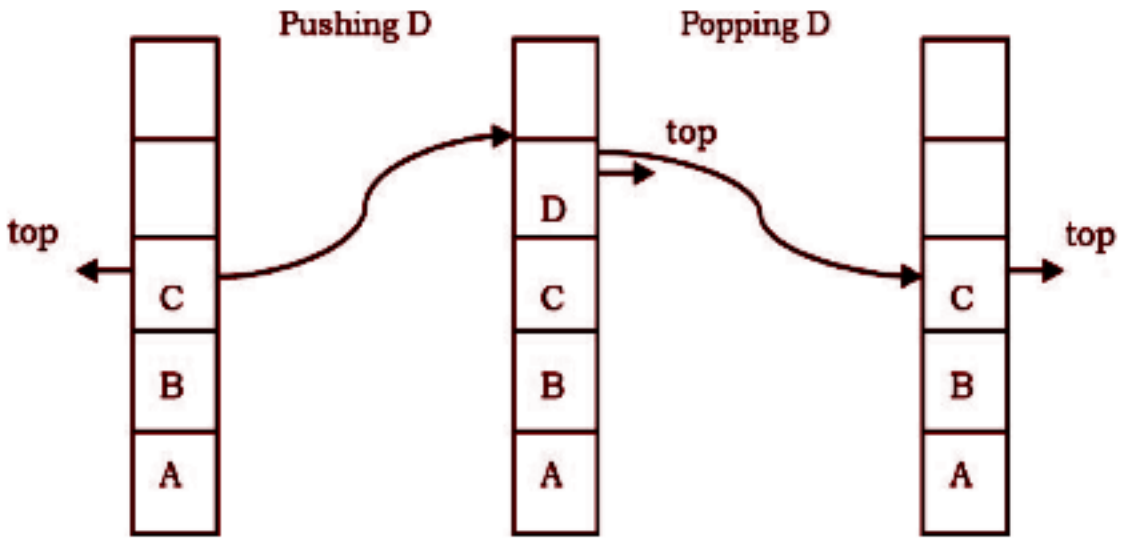


Here, unless the book at the topmost position is removed from the pile, we can't have access to the second book from the top and similarly, for the books below the second one. When we apply the same technique over the data in our program then, this pile-type structure is said to be a stack.

Like deletion, we can only insert the book at the top of the pile rather than at any other position. This means that the object/data that made its entry at the last would be one to come out first, hence known as **LIFO.**

Operations on the stack:

- In a stack, insertion and deletion are done at one end, called **top**.
- **Insertion**: This is known as a **push** operation.
- **Deletion**: This is known as a **pop** operation.



Main stack operations

• `Push (int data)`: Insert data onto the stack.

• `int Pop()`: Removes and returns the last inserted element from the stack.

Auxiliary stack operations

• `int Top()`: Returns the last inserted element without removing it.

• `int Size()`: Returns the number of elements stored in the stack.

• `int IsEmptyStack()`: Indicates whether any elements are stored in the stack or not.

• `int IsFullStack()`: Indicates whether the stack is full or not.

Performance

Let n be the number of elements in the stack. The complexities of stack operations with this representation can be given as:

Space Complexity (for n push operations)	O(*n*)
Time Complexity of Push()	O(1)
Time Complexity of Pop()	O(1)
Time Complexity of Size()	O(1)
Time Complexity of IsEmptyStack()	O(1)
Time Complexity of IsFullStackf)	O(1)
Time Complexity of DeleteStackQ	O(1)

Exceptions

- Attempting the execution of an operation may sometimes cause an error condition, called an exception.
- Exceptions are said to be “thrown” by an operation that cannot be executed.
- Attempting the execution of pop() on an empty stack throws an exception called **Stack Underflow**.
- Trying to push an element in a full-stack throws an exception called **Stack Overflow**.

Implementing stack- Simple Array Implementation

This implementation of stack ADT uses an array. In the array, we add elements from left to right and use a variable to keep track of the index of the **top** element.

Consider the given implementation in Python for more understanding:

```
class Stack:
    #Constructor
    def __init__(self):
        self.stack = list()
        self.maxSize = 8 #Maximum size of the list
        self.top = 0 #Top element

    #Adds element to the Stack
    def push(self,data):
        if self.top>=self.maxSize:#Stack Overflow
            return ("Stack Full!")
        self.stack[top]= data #Assign the new element at 'top'
        self.top += 1 #Increment top
        return True

    #Removes element from the stack
    def pop(self):
        if self.top<=0:#Stack Underflow
            return ("Stack Empty!")
        item = self.stack[top-1]#Topmost element
        self.top -= 1 #Simply decrementing top
        return item #Returning the removed element

    #Size of the stack
    def size(self):
        return self.top
```

Limitations of Simple Array Implementation

In other programming languages like C++, Java, etc, the maximum size of the an array must first be defined i.e. it is fixed and it cannot be changed. However, in Python, arrays are resizable by nature. This means that even though Python internally handles the resizing of arrays, it is still very expensive.

Stack using Linked Lists

Till now we have learned how to implement a stack using arrays, but as discussed earlier, we can also create a stack with the help of linked lists. All the five functions that stacks can perform could be made using linked lists:

```
class Node:#Node of a Linked List
    def __init__(self, data):
       self.data = data
       self.next = None

class Stack:#Stack Implementation using LL
    def __init__(self):
        self.head = None

    def push(self, data):
        if self.head is None:
            self.head = Node(data)
        else:
            new_node = Node(data)
            new_node.next = self.head
            self.head = new_node

    def pop(self):
        if self.head is None:
            return None
        else:
            popped = self.head.data
            self.head = self.head.next
            return popped
```

Inbuilt Stack in Python

- The **queue** module also has a **LIFO Queue**, which is basically a **Stack**.

There are various functions available in this module:

- **maxsize** – Returns the maximum number of items allowed in the stack.
- **empty()** – Returns **True** if the stack is empty, otherwise it returns **False**.
- **get()** – Remove and return an item from the stack.
- **put(item)** – Put an item into the stack.
- **qsize()** – Return the number of items currently present in the stack.

```
from queue import Queue

stack = Queue(maxsize = 3) # Initializing a stack

print(q.maxsize())# Maximum size of the stack

stack.put('14') # Adding elements to the stack
stack.put('28')
stack.put('36')

print("\nisFull: ", stack.full()) # Check if the stack is full

print("\nElement dequeued from the stack: ")
print(stack.get()) # Removing an element from stack
```

We get the following output:

```
3
isFull:  True
Element dequeued from the stack:
36 #Stack follows LIFO
```

Problem Statement- Balanced Parenthesis

For a given string expression containing only round brackets or parentheses, check if they are balanced or not. Brackets are said to be balanced if the bracket which opens last, closes first. You need to return a boolean value indicating whether the expression is balanced or not.

Approach:

- We will use stacks.
- Each time, when an open parenthesis is encountered push it in the stack, and when closed parenthesis is encountered, match it with the top of the stack and pop it.
- If the stack is empty at the end, return Balanced otherwise, Unbalanced.

Python Code:

```
open = ["[","{","("]
close = ["]","}",")"]

# Function to check parentheses
def checkBalanced(inputStr):
    s = [] #The stack
    for i in inputStr:
        if i in open:
            s.append(i)
        elif i in close:
            position = close.index(i)
            if ((len(s)>0) and (open[position]==s[len(s)-1])):
                s.pop()
            else:
                return "Unbalanced"
    if len(s) == 0:
        return "Balanced"
    else:
        return "Unbalanced"
```

Practice Problems:

- https://www.hackerrank.com/challenges/equal-stacks/problem
- https://www.hackerrank.com/challenges/simple-text-editor/problem
- https://www.techiedelight.com/design-a-stack-which-returns-minimum-element-without-using-auxiliary-stack/
- https://www.hackerearth.com/practice/data-structures/stacks/basics-of-stacks/practice-problems/algorithm/monk-and-order-of-phoenix/

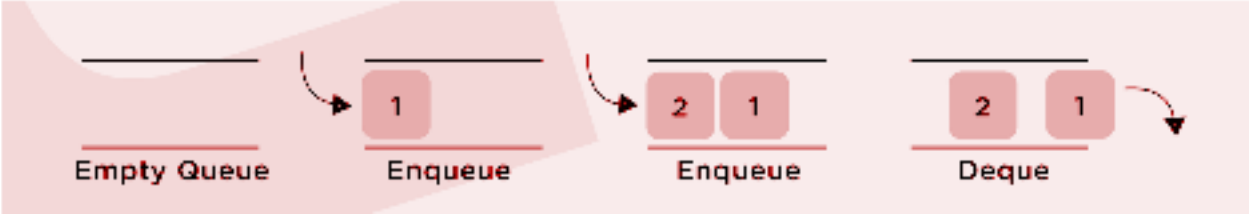
Queues

Introduction

- Like stack, the queue is also an abstract data type.
- As the name suggests, in queue elements are inserted at one end while deletion takes place at the other end.
- Queues are open at both ends, unlike stacks that are open at only one end(the top).

Let us consider a queue at a movie ticket counter:

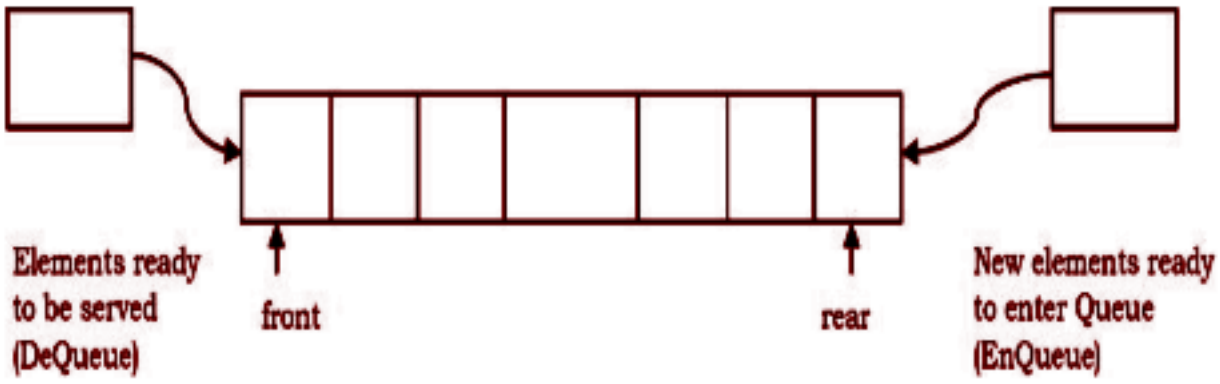
- Here, the person who comes first in the queue is served first with the ticket while the new seekers of tickets are added back in the line.
- This order is known as **First In First Out (FIFO)**.
- In programming terminology, the operation to add an item to the queue is called "enqueue", whereas removing an item from the queue is known as "dequeue".



Working of A Queue

Queue operations work as follows:

1. Two pointers called **FRONT** and **REAR** are used to keep track of the first and last elements in the queue.
2. When initializing the queue, we set the value of FRONT and REAR to -1.
3. On **enqueuing** an element, we increase the value of the REAR index and place the new element in the position pointed to by REAR.
4. On **dequeuing** an element, we return the value pointed to by FRONT and increase the FRONT index.
5. Before enqueuing, we check if the queue is already full.
6. Before dequeuing, we check if the queue is already empty.
7. When enqueuing the first element, we set the value of FRONT to 0.
8. When dequeuing the last element, we reset the values of FRONT and REAR to -1.



Implementation of A Queue Using Array

A Queue contains majorly these five functions that we will be implementing:

Main Queue Operations

• `enqueue(int data)`: Inserts an element at the end of the queue

• `int dequeue()`: Removes and returns the element at the front of the queue

Auxiliary Queue Operations

• `int front()`: Returns the element at the front without removing it

• `int size()`: Returns the number of elements stored in the queue

• `int IsEmpty()`: Indicates whether no elements are stored in the queue or not

Now, let's implement these functions in Python. Follow up the code along with the comments below:

```
class Queue():
    def __init__(self):
        self._queue = []
        self.size = 0 #Current size of the queue
        self.maxSize = 10 #Maximum size of the queue

    def enqueue(self, item): #Add an element to the queue
        if self.size < self.maxSize:
             self._queue.append(item) #Add element to last

    def dequeue(self): #Remove an element from the queue
        first = self._queue[0] #Remove the FIRST element
        del self._queue[0]
        return first
```

Queues using LL

Given below is an implementation of Queue using Linked List. This is similar to the way we wrote the LL Implementation for a Stack:

```
class Node: #Nodes of the Linked List
    def __init__(self, data):
        self.data = data
        self.next = None

class Queue: #Queue implementation using Linked List
    def __init__(self):
        self.front = self.rear = None

    def isEmpty(self):
        return self.front == None

    def enqueue(self, item):
        temp = Node(item) #Adding a new node
        if self.rear == None:
            self.front = self.rear = temp
            return
        self.rear.next = temp
        self.rear = temp

    def dequeue(self):
        if self.isEmpty():#If there is no element to dequeue
            return
        temp = self.front
        self.front = temp.next

        if(self.front == None):
            self.rear = None
```

Queue using Python List

`List` is Python's built-in data structure that can be used as a queue. Instead of `enqueue()` and `dequeue()`, `append()` and `pop()` function is used.

```
#Inbuilt implementation of Queue using List
queue = []

# Adding elements to the queue
queue.append('1')#Using the .append() function
queue.append('2')
queue.append('3')

print("Initial Queue:")
print(queue)#Queue after appending the elements

# Removing elements from the queue
print("\nElements dequeued from queue:")
print(queue.pop(0))#Removing first element from queue
print(queue.pop(0))
print(queue.pop(0))

print("\nQueue after removing elements:")
print(queue)
```

Output:

```
Initial queue:
['1', '2', '3']
Elements dequeued from queue:
1
2
3
Queue after removing elements:
[]
```

In-built Queue in Python

- **Queue** is built-in module of Python which is used to implement a queue.
- `queue.Queue(maxsize)` initializes a variable to a maximum size of maxsize.
- This Queue follows the **FIFO** rule.

There are various functions available in this module:

- **maxsize** – Returns the maximum number of items allowed in the queue.
- **empty()** – Returns **True** if the queue is empty, otherwise it returns **False**.
- **get()** – Remove and return an item from the queue.
- **put(item)** – Put an item into the queue.
- **qsize()** – Return the number of items currently present in the queue.

Note: A max size of zero '0' means an infinite queue.

```
from queue import Queue

q = Queue(maxsize = 3) # Initializing a queue

print(q.maxsize())# Maximum size of the Queue

q.put('14') # Adding elements to the queue
q.put('28')
q.put('36')

print("\nisFull: ", q.full()) # Check if the queue is full

print("\nElement dequeued from the queue: ")
print(q.get()) # Removing an element from queue

print("\nisEmpty: ", q.empty()) # Check if the queue is empty
```

We get the following output:

```
3

isFull:  True

Element dequeued from the queue:
14 # A queue follows FIFO

isEmpty:  False
```

Practice problems:

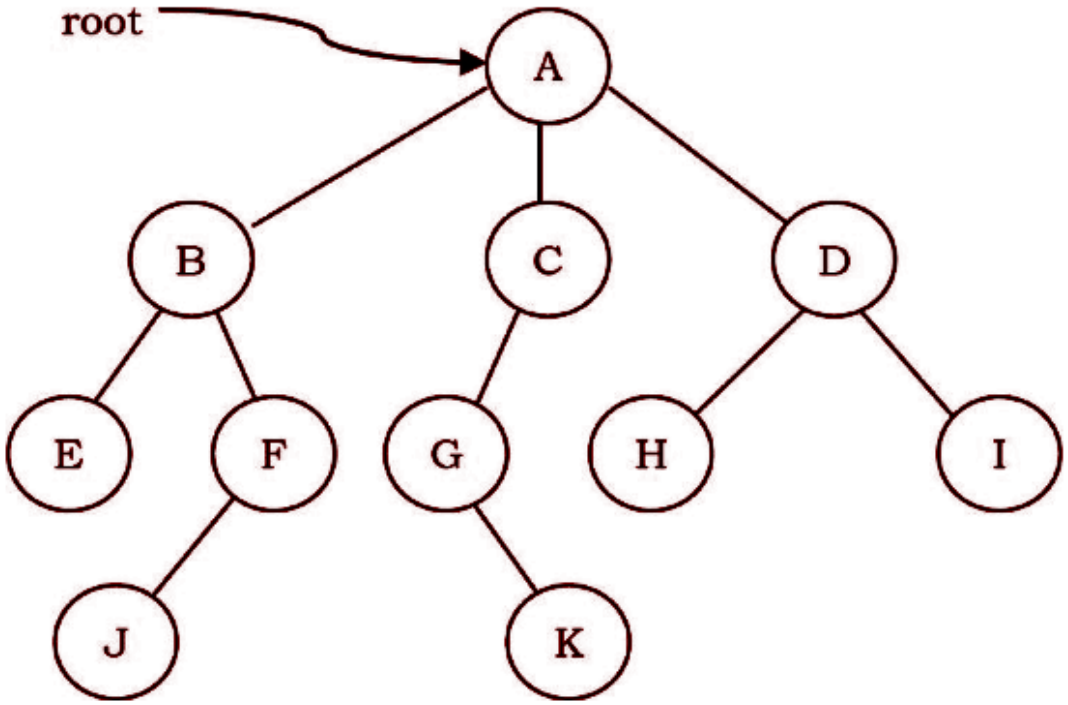
- https://www.spoj.com/problems/ADAQUEUE/
- https://www.hackerearth.com/practice/data-structures/queues/basics-of-queues/practice-problems/algorithm/number-recovery-0b988eb2/
- https://www.codechef.com/problems/SAVJEW
- https://www.hackerrank.com/challenges/down-to-zero-ii/problem

Binary Trees- 1

What is A Tree?

- A tree is a data structure similar to a linked list but instead of each node pointing simply to the next node in a linear fashion, each node points to several nodes.
- A tree is an example of a non- linear data structure.
- A tree structure is a way of representing the hierarchical nature of a structure in a graphical form.

Terminology Of Trees

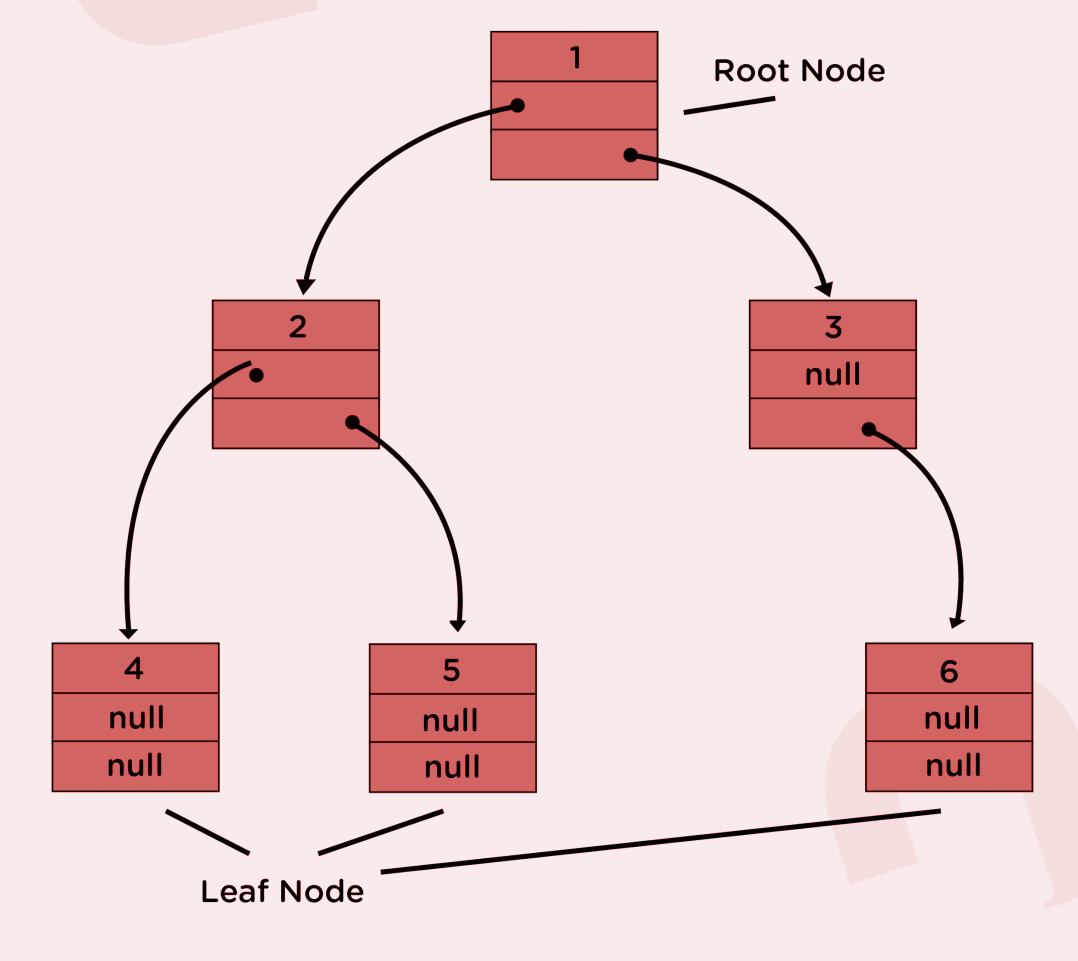


- The root of a tree is the node with no parents. There can be at most one root node in a tree **(node A in the above example)**.

- An **edge** refers to the link from a parent to a child **(all links in the figure)**.
- A node with no children is called a **leaf node** **(E, J, K, H, and I)**.
- The children nodes of the same parent are called **siblings** **(B, C, D are siblings of parent A and E, F are siblings of parent B)**.
- The set of all nodes at a given depth is called the **level** of the tree **(B, C, and D are the same level)**. The root node is at level zero.
- The **depth** of a node is the length of the path from the root to the node **(depth of G is 2, *A -> C –> G*)**.
- The **height** of a node is the length of the path from that node to the deepest node.
- The **height** of a tree is the length of the path from the root to the deepest node in the tree.
- A (rooted) tree with only one node (the root) has a height of zero.

Binary Trees

- A generic tree with at most two child nodes for each parent node is known as a binary tree.
- A binary tree is made of nodes that constitute a **left** pointer, a **right** pointer, and a data element. The **root** pointer is the topmost node in the tree.
- The left and right pointers recursively point to smaller **subtrees** on either side.
- An empty tree is also a valid binary tree.
- *A formal definition is:* A **binary tree** is either empty (represented by a None pointer), or is made of a single node, where the left and right pointers (recursive definition ahead) each point to a **binary tree**.

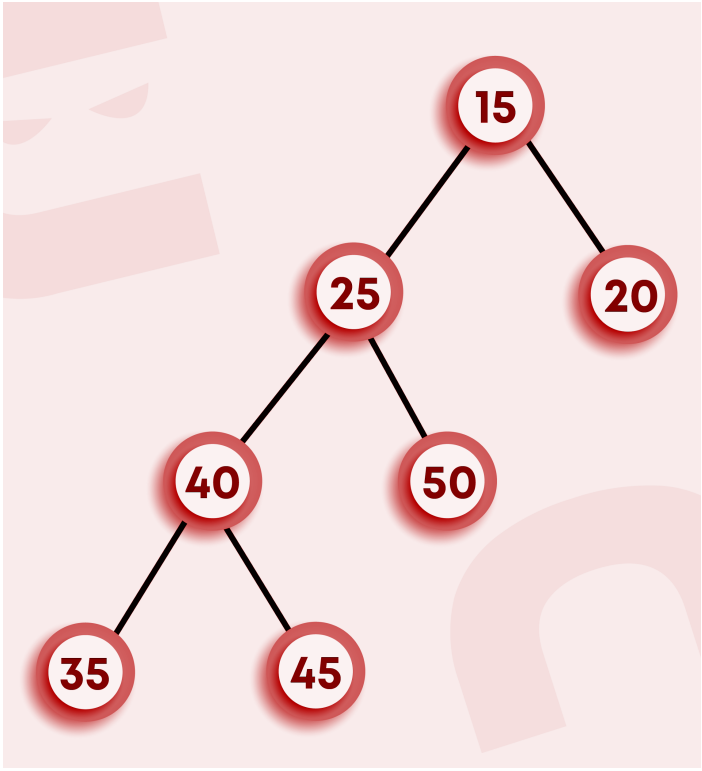


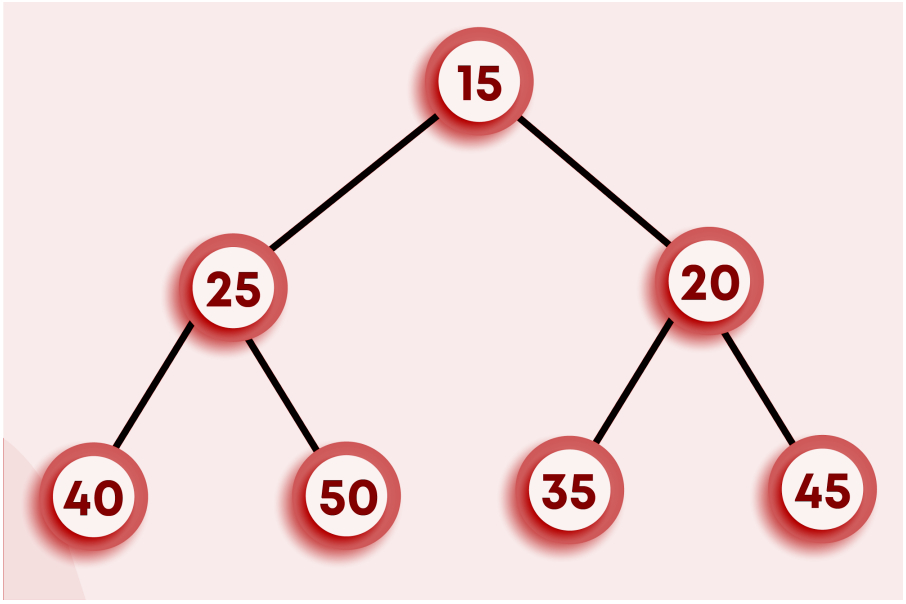
Types of binary trees:

Full binary trees: A binary tree in which every node has 0 or 2 children is termed as a full binary tree.

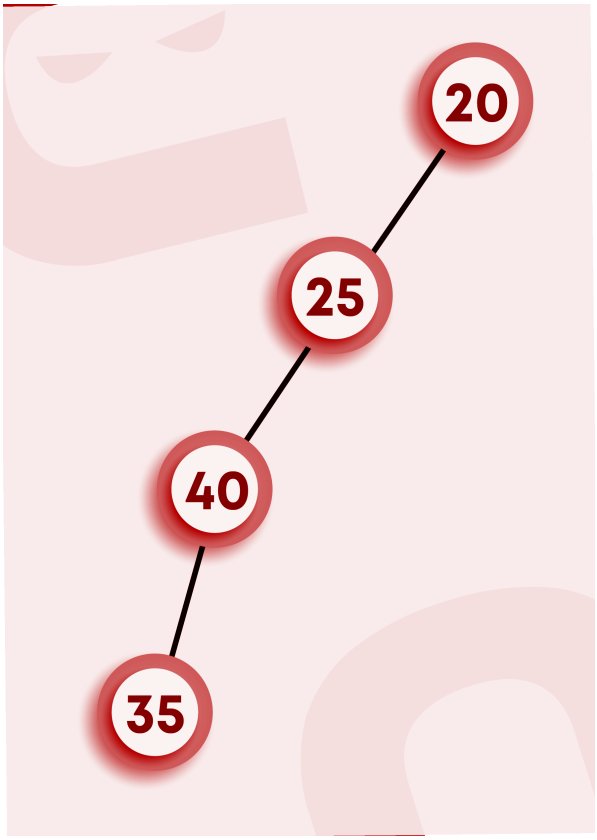
Complete binary tree: A complete binary tree has all the levels filled except for the last level, which has all its nodes as much as to the left.

Perfect binary tree: A binary tree is termed perfect when all its internal nodes have two children along with the leaf nodes that are at the same level.



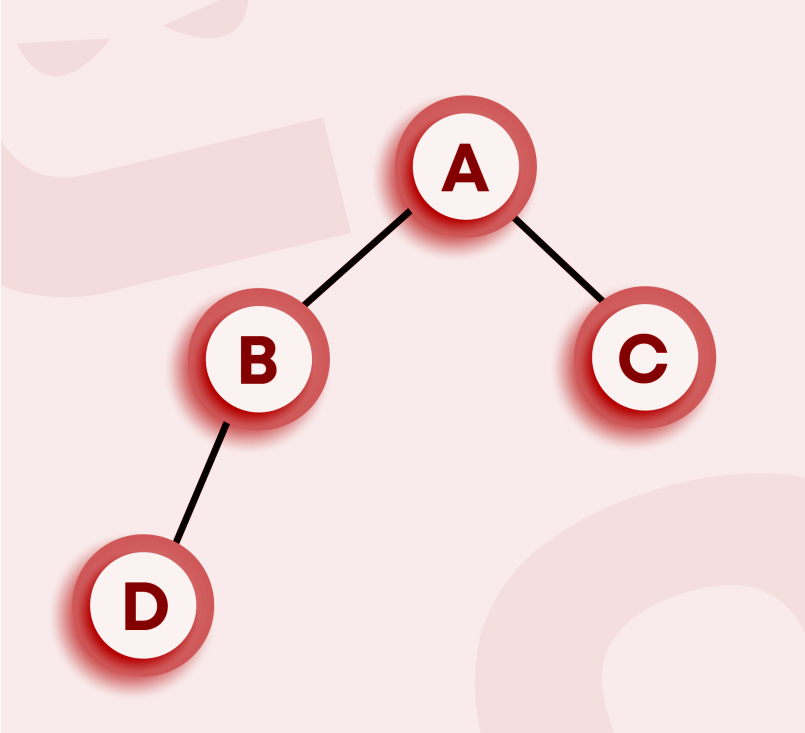


A degenerate tree: In a degenerate tree, each internal node has only one child.



The tree shown above is degenerate. These trees are very similar to linked-lists.

Balanced binary tree: A binary tree in which the difference between the depth of the two subtrees of every node is at most one is called a balanced binary tree.



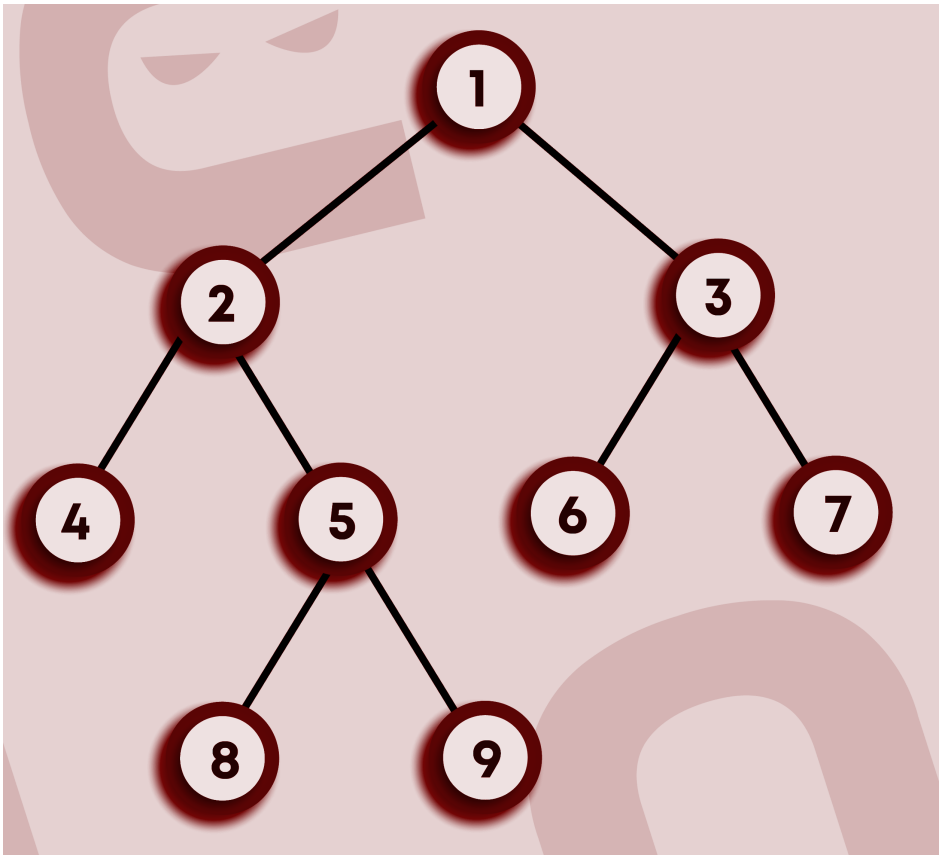
Binary tree representation:

Binary trees can be represented in two ways:

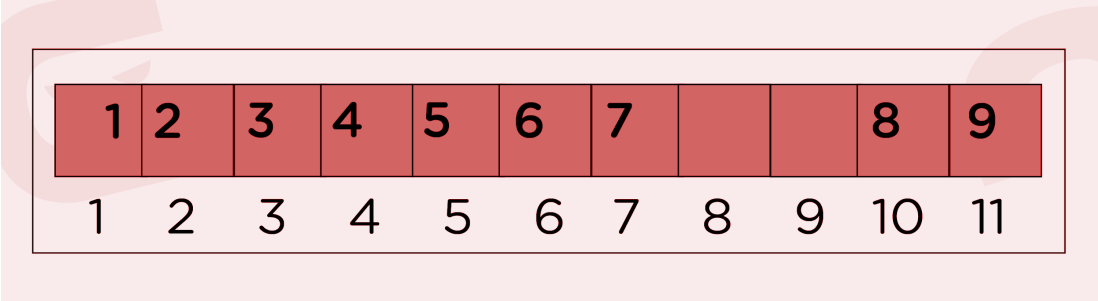
Sequential representation

- This is the most straightforward technique to store a tree data structure. An array is used to store the tree nodes.
- The number of nodes in a tree defines the size of the array.
- The root node of the tree is held at the first index in the array.
- In general, if a node is stored at the **i^{th}** location, then its **left** and **right** child are kept at **$(2i)^{th}$** and **$(2i+1)^{th}$** locations in the array, respectively.

Consider the following binary tree:



The array representation of the above binary tree is as follows:



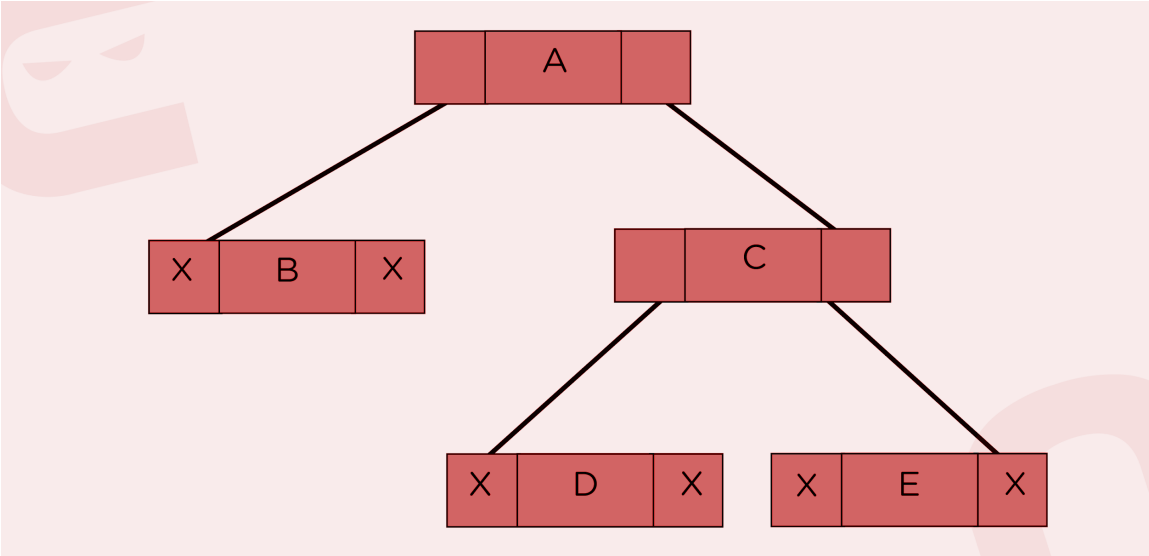
As discussed above, we see that the left and right child of each node is stored at locations **2*(nodePosition)** and **2*(nodePosition)+1**, respectively.

For Example, The location of node 3 in the array is 3. So its left child will be placed at **2*3 = 6**. Its right child will be at the location **2*3 +1 = 7**. As we can see in the array, children of 3, which are 6 and 7, are placed at locations 6 and 7 in the array.

Note: The sequential representation of the tree is not preferred due to the massive amount of memory consumption by the array.

Linked list representation:

In this type of model, a linked list is used to store the tree nodes. The nodes are connected using the parent-child relationship like a tree. The following diagram shows a linked list representation for a tree.

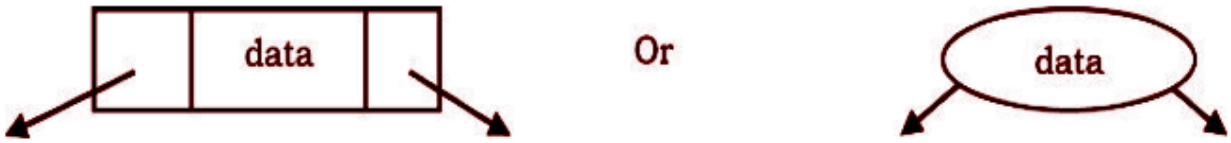


As shown in the above representation, each linked list node has three components:

- Pointer to the left child
- Data
- Pointer to the right child

Note: If there are no children for a given node (leaf node), then the left and right pointers for that node are set to **None**.

Let's now check the implementation of the **Binary tree class**.




```
class BinaryTreeNode:
    def __init__(self, data):
        self.left = None #To store data
        self.right = None #For storing the reference to left pointer
        self.data = data #For storing the reference to right pointer
```

Operations on Binary Trees

Basic Operations

- Inserting an element into a tree
- Deleting an element from a tree
- Searching for an element
- Traversing the tree

Auxiliary Operations

- Finding the size of the tree
- Finding the height of the tree
- Finding the level which has the maximum sum and many more...

Print Tree Recursively

Let's first write a program to print a binary tree recursively. Follow the comments in the code below:

```
def printTree(root):
      root == None: #Empty tree
            return
      print(root.data, end ":") #Print root data
      if root.left != None:
            print("L", root.left.data, end=",") #Print left child
      if root.right != None:
            print("R", root.right.data, end="") #Print right child
      Print #New line
```

```
    printTree(root.left) #Recursive call to print left subtree
    printTree(root.right) #Recursive call to print right subtree
```

Input Binary Tree

We will be following the level-wise order for taking input and -1 denotes the **None** pointer.

```
def treeInput():
    rootData = int(input())
    if rootData == -1: #Leaf Node is denoted by -1
        return None

    root = BinaryTreeNode(rootData) #Create a tree node
    leftTree =treeInput() #Take input for left subtree
    rightTree = treeInput() #Take input for right subtree
    root.left = leftTree #Assign the left subtree to the left child
    root.right = rightTree #Right subtree to the right child
    return root
```

Count nodes

- Unlike the Generic trees, where we need to traverse the children vector of each node, in binary trees, we just have at most left and right children for each node.
- Here, we just need to recursively call on the right and left subtrees independently with the condition that the node pointer is not None.
- Follow the comments in the upcoming code for better understanding:

```
def count_nodes(node):
    if node is None: #Check if root node is None
        return 0
```

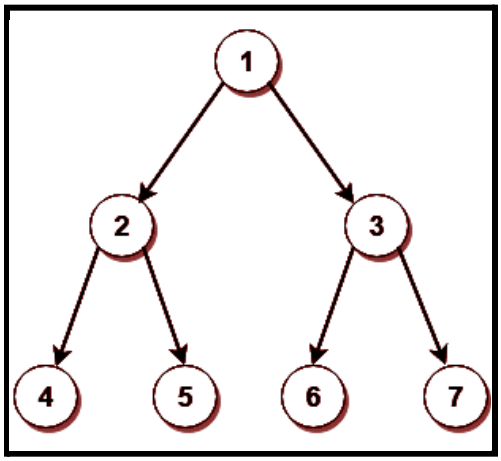
```
    return 1 + count_nodes(node.left) + count_nodes(node.right)
#Recursively count number of nodes in left and right subtree and add
```

Binary tree traversal

Following are the ways to traverse a binary tree and their orders of traversal:

- **Preorder traversal** : ROOT -> LEFT -> RIGHT
- **Postorder traversal** : LEFT -> RIGHT-> ROOT
- **Inorder traversal** : LEFT -> ROOT -> RIGHT

Some examples of the above-stated traversal methods:



- **Preorder traversal:** 1, 2, 4, 5, 3, 6, 7
- **Postorder traversal:** 4, 5, 2, 6, 7, 3, 1
- **Inorder traversal:** 4, 2, 5, 1, 6, 3, 7

Let's look at the code for inorder traversal, below:

```
# A function to do inorder tree traversal
def printInorder(root):
    if root:#If tree is not empty
        printInorder(root.left) # First recur on left child
```

```
        print(root.val)# Then print the data of the node
        printInorder(root.right) # Now recur on right child
```

Now, from this inorder traversal code, try to code preorder and postorder traversal yourselves. If you get stuck, refer to the solution tab for the same.

Node with the Largest Data

In a Binary Tree, we must visit every node to figure out the maximum. So the idea is to traverse the given tree and for every node return the maximum of 3 values:

- Node's data.
- Maximum in node's left subtree.
- Maximum in node's right subtree.

Below is the implementation of the above approach.

```
def findMaximum(root):
    # Base case
    if (root == None):
        return float('-inf') #**

    # Return maximum of 3 values:
    # 1) Root's data 2) Max in Left Subtree
    # 3) Max in right subtree
    max = root.data
    lmax = findMaximum(root.left) #Maximum of left subtree
    rmax = findMaximum(root.right) #Maximum of right subtree
    if (lmax > max):
        max = lmax
    if (rmax > max):
        max = rmax
    return max
```

Note**: In python, float values can be used to represent an infinite integer. One can use **float('-inf')** as an integer to represent it as "Negative" infinity or the smallest possible integer.

Binary Trees- 2

Construct a Binary Tree from Preorder and Inorder Traversal

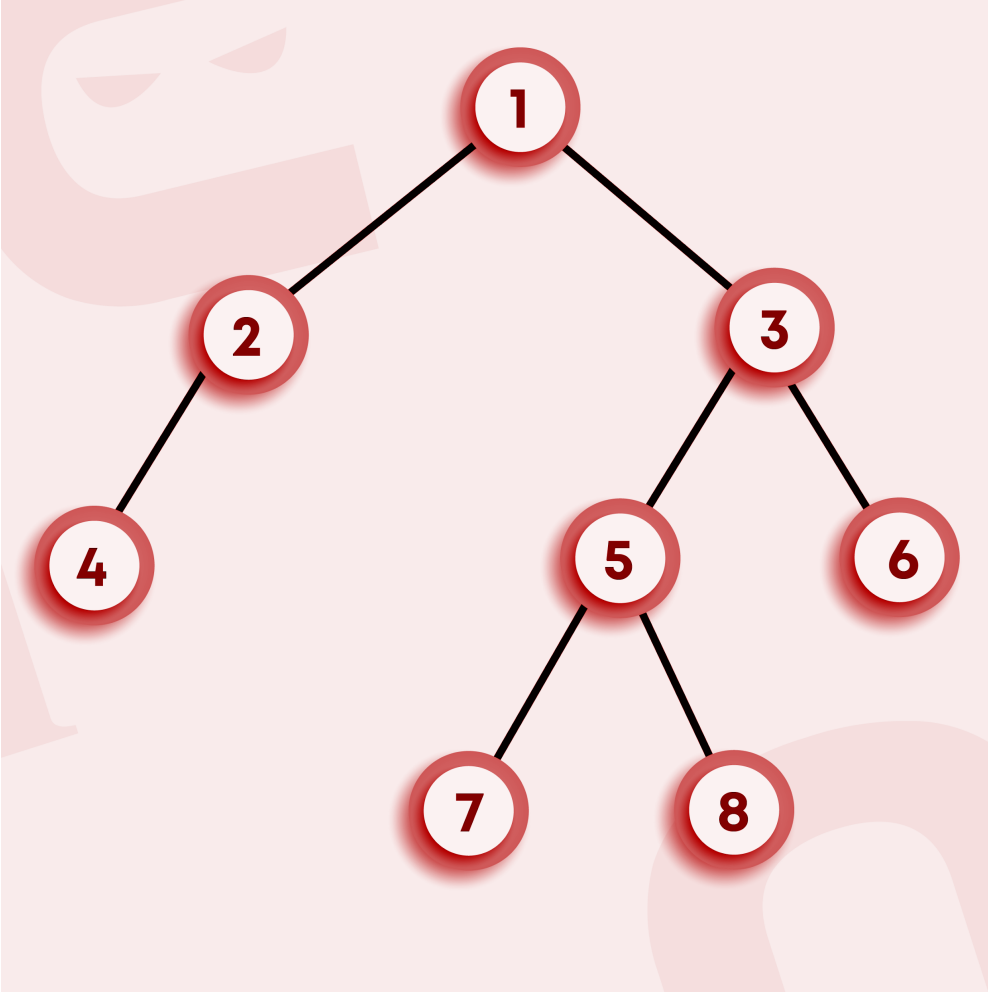
Consider the following example to understand this better.

Input:

Inorder traversal : {4, 2, 1, 7, 5, 8, 3, 6}

Preorder traversal : {1, 2, 4, 3, 5, 7, 8, 6}

Output: Below binary tree...



- The idea is to start with the root node, which would be the first item in the preorder sequence, and find the boundary of its left and right subtree in the inorder array.
- Now, all keys before the root node in the inorder array become part of the left subtree, and all the indices after the root node become part of the right subtree.
- We repeat this recursively for all nodes in the tree and construct the tree in the process.

To illustrate, consider below inorder and preorder sequence-
Inorder: {4, 2, 1, 7, 5, 8, 3, 6}
Preorder: {1, 2, 4, 3, 5, 7, 8, 6}

- The root will be the first element in the preorder sequence, i.e. 1.
- Next, we locate the index of the root node in the inorder sequence.
- Since 1 is the root node, all nodes before 1 must be included in the left subtree, i.e., {4, 2}, and all the nodes after one must be included in the right subtree, i.e. {7, 5, 8, 3, 6}.
- Now the problem is reduced to building the left and right subtrees and linking them to the root node.

Thus we get:

Left subtree:	**Right subtree:**
Inorder : {4, 2}	**Inorder :** {7, 5, 8, 3, 6}
Preorder : {2, 4}	**Preorder :** {3, 5, 7, 8, 6}

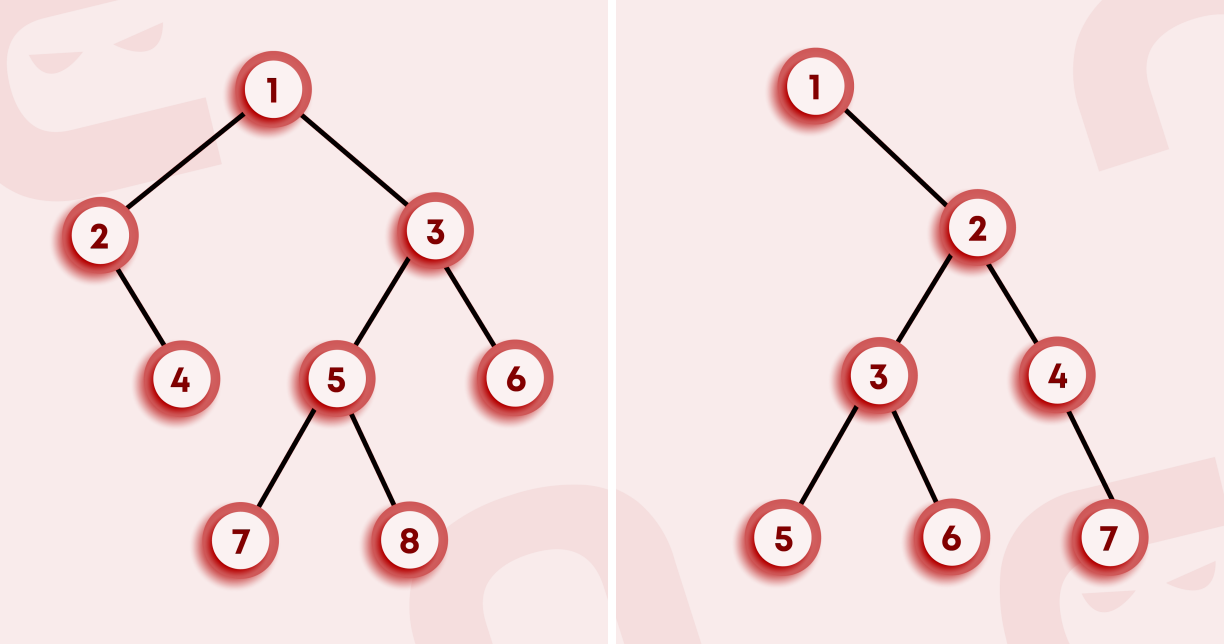
Follow the above approach until the complete tree is constructed. Now let us look at the code for this problem.

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

The Diameter of a Binary tree

- The **diameter** of a tree (sometimes called the width) is the number of nodes on the longest path between two child nodes.
- The diameter of the binary tree may pass through the root (not necessary).

For example, the figure below shows two binary trees having diameters 6 and 5, respectively. The diameter of the binary tree shown on the left side passes through the root node while on the right side, it doesn't.



There are three possible paths of the diameter:

1. The diameter could be the sum of the left height and the right height.
2. It could be the left subtree's diameter.
3. It could be the right subtree's diameter.

We will pick the one with the maximum value.

Now let's check the code for this...

```
def height(node):

    # Base Case : Tree is empty
    if node is None:
        return 0

    # If tree is not empty then height = 1 + max of left
    # height and right heights
    return 1 + max(height(node.left), height(node.right))

def diameter(root):

    # Base Case when tree is empty
    if root is None:
        return 0

    # Get the height of left and right subtrees
    leftH= height(root.left)
    rightH = height(root.right)

    # Get the diameter of left and right subtrees
    leftD = diameter(root.left)
    rightD = diameter(root.right)

    # Return max of the three
    return max(leftH + rightH + 1, max(leftD, rightD))
```

The time complexity for the above approach:

- Height function traverses each node once; hence time complexity will be **O(n).**
- Option2 and Option3 also traverse on each node, but for each node, we are calculating the height of the tree considering that node as the root node, which makes time complexity equal to **O(n*h)**. (worst case with skewed trees, i.e., a type of binary tree in which all the nodes have only either one child or no child.) Here, **h** is the height of the tree, which could be **O(n^2)**.

This could be reduced if the height and diameter are obtained simultaneously, which could prevent extra **n** traversals for each node.

The Diameter of a Binary tree: Better Approach

In the previous approach, for each node, we were finding the height and diameter independently, which was increasing the time complexity. In this approach, we will find the height and diameter for each node at the same time, i.e., we will store the height and diameter using a pair class where the **first** pointer will be storing the height of that node and the **second** pointer will be storing the diameter. Here, also we will be using recursion.

Let's focus on the **base case**: For a NULL tree, height and diameter both are equal to 0. Hence, the pair class will store both of its values as zero.

Now, moving to **Hypothesis**: We will get the height and diameter for both left and right subtrees, which could be directly used.

Finally, the **induction step**: Using the result of the Hypothesis, we will find the height and diameter of the current node:

Height = `max(leftHeight, rightHeight)`

Diameter = `max(leftHeight + rightHeight, leftDiameter, rightDiameter)`

To create a pair class, follow the syntax below:

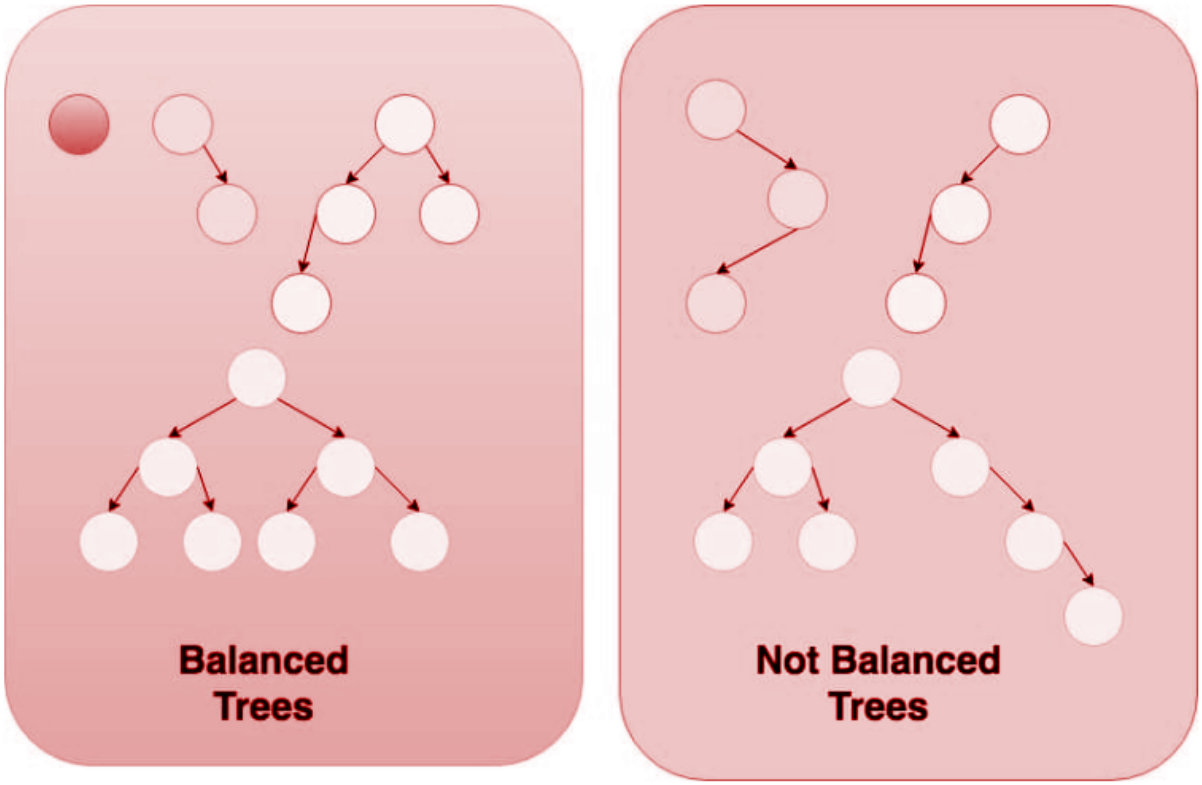
```
class Pair:
    def __init__(self, data): # Constructor to create a new Pair
        self.data = data
        self.left = self.right = None
```

It can be observed that we are just traversing each node once while making recursive calls and the rest of all other operations are performed in constant time, hence the time complexity of this program is **O(n)**, where **n** is the number of nodes.

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

Balanced Binary Tree

A binary tree is balanced if, for each node in the tree, the difference between the height of the right subtree and the left subtree is at most one.



Check if a Binary Tree is Balanced

From the definition of a balanced tree, we can conclude that a binary tree is balanced if at every node in the tree:

- The right subtree is balanced.
- The left subtree is balanced.
- The difference between the height of the left subtree and the right subtree is at most 1

Let's define a recursive function **is_balanced()** that takes a root node as an argument and returns a boolean value that represents whether the tree is balanced or not.

Let's also define a helper function **get_height()** that returns the height of a tree. Notice that **get_height()** is also implemented recursively

```
def get_height(root):#Returns height of the tree
    if root is None:
        return 0
    return 1 + max(get_height(root.left , get_height(root.right))

def is_balanced(root):
    # a None tree is balanced
    if root is None:
        return True
    return is_balanced(root.right) and is_balanced(root.left) and
abs(get_height(root.left) - get_height(root.right)) <= 1
```

The **is_balanced()** function returns true if the right subtree and the left subtree are balanced, and if the difference between their height does not exceed 1.

Check if a Binary Tree is Balanced- Improved Solution

Here, we are using two recursive functions: one that checks if a tree is balanced, and another one that returns the height of a tree. Can we achieve the same goal by using only one recursive function?

Let us define our recursive function **is_balanced_helper()** to be a function that takes one argument, the tree root and returns an integer such that:

- If the tree is balanced, return the height of the tree
- If the tree is not balanced, return -1

Notice that this new **is_balanced_helper()** can be easily implemented recursively as well by following these rules:

- Apply is_balanced_helper on both the right and left subtrees
- If either the right or left subtrees returns -1, then we should return -1 (because our tree is not balanced if either subtree is not balanced)
- If both subtrees return an integer value (indicating the heights of the subtrees), then we check the difference between these heights.
- If the difference doesn't exceed 1, then we return the height of this tree. Otherwise, we return -1

Let us now look at its implementation:

```
def is_balanced_helper(root):
    if root is None: # a None tree is balanced
        return 0

 # if the left subtree is not balanced, then tree is not balanced
    left_height = is_balanced_helper(root.left)
    if left_height == -1:
        return -1

    # if the right subtree is not balanced,then tree is not balanced
    right_height = is_balanced_helper(root.right)
    if right_height == -1:
        return -1

    # If the difference in heights is greater than 1
    if abs(left_height - right_height) > 1:
        return -1

    # this tree is balanced, return its height
    return max(left_height, right_height) + 1
```

Finally we know that, if **is_balanced_helper()** returns a number that is greater than -1, the tree is balanced, otherwise, it is not.

```
def is_balanced(root):
    return is_balanced_helper(root) > -1
```

Practice problems:

- https://www.hackerrank.com/challenges/tree-top-view/problem
- https://www.codechef.com/problems/BTREEKK
- https://www.spoj.com/problems/TREEVERSE/
- https://www.hackerearth.com/practice/data-structures/trees/binary-and-nary-trees/practice-problems/approximate/largest-cycle-in-a-tree-9113b3ab/

Binary Search Trees- 1

Introduction

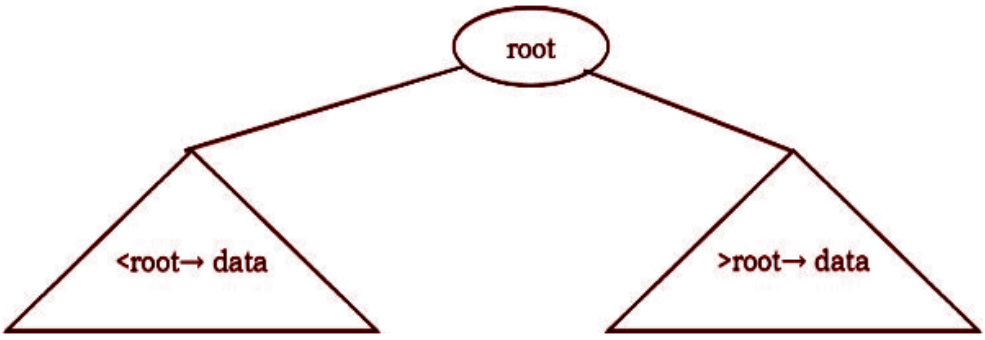
- These are the specific types of binary trees.
- These are inspired by the binary search algorithm.
- Time complexity on insertion, deletion, and searching reduces significantly as it works on the principle of binary search rather than linear search, as in the case of normal binary trees (will discuss it further).

Binary Search Tree Property

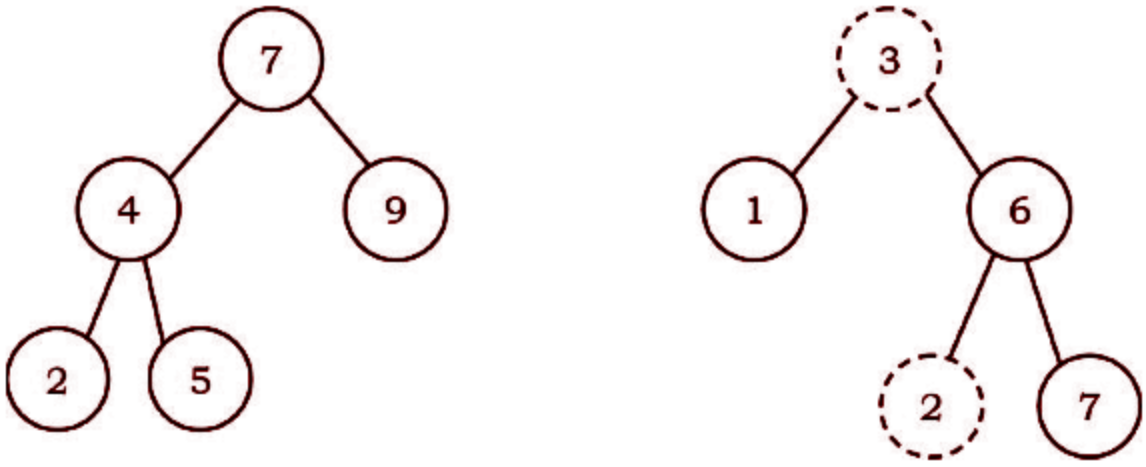
In binary search trees, all the left subtree elements should be less than root data and all the right subtree elements should be greater than root data. This is called **Binary Search Tree property**.

- The left subtree of a node ONLY contains nodes with keys less than the node's key.
- The right subtree of a node ONLY contains nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

Note: The **BST** property should be satisfied at every node in the tree.



Example: The left tree is a binary search tree and the right tree is not a binary search tree *(This because the BST property is not satisfied at node 6. Its child with key 2, is less than its parent with key 3, which is a violation, as all the nodes on the right subtree of root node 3, must have keys greater than or equal to 3)*.



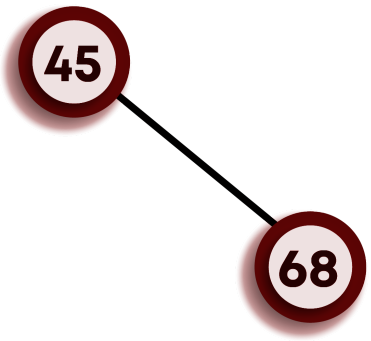
Store Data in BST

Example: Insert {45, 68, 35, 42, 15, 64, 78} in a BST in the order they are given.

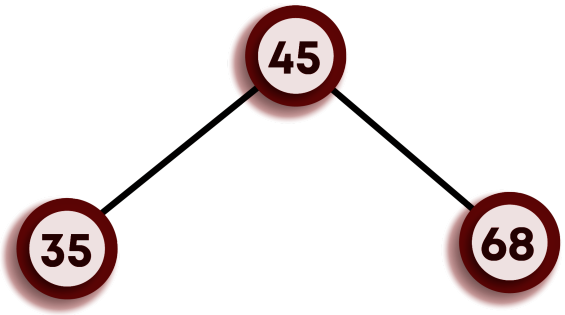
Solution: Follow the steps below:

1. Since the tree is empty, so the first node will automatically be the root node.

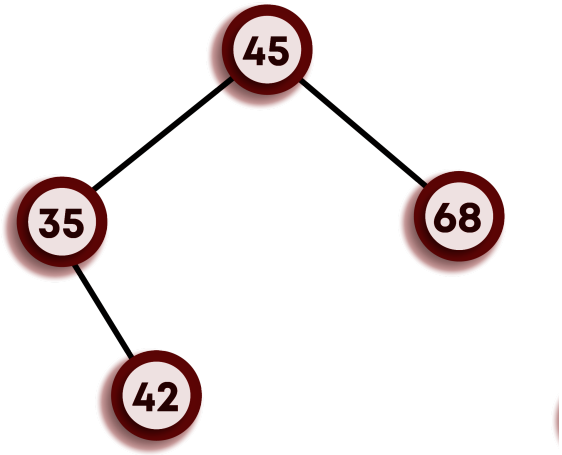
2. Now, we have to insert 68, which is greater than 45, so it will go on the right side of the root node.



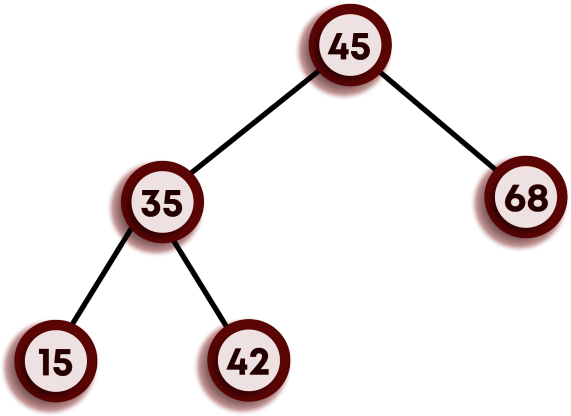
3. To insert 35, we will start from the root node. Since 35 is smaller than 45, it will be inserted on the left side.



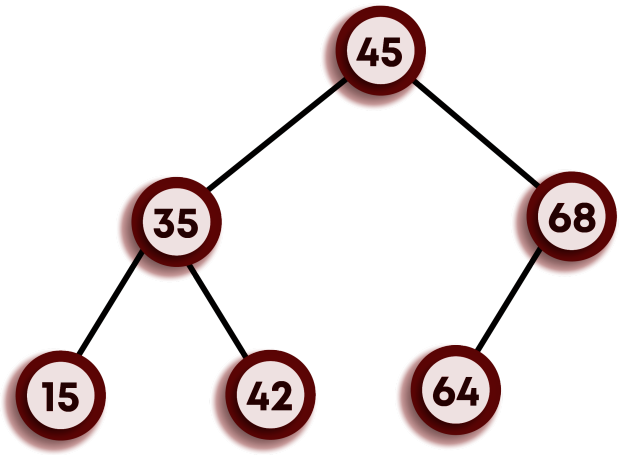
4. Moving on to inserting 42. We can see that 42 < 45, so it will be placed on the left side of the root node. Now, we will check the same on the left subtree. We can see that 42 > 35 means 42 will be the right child of 35, which is still a part of the left subtree of the root node.



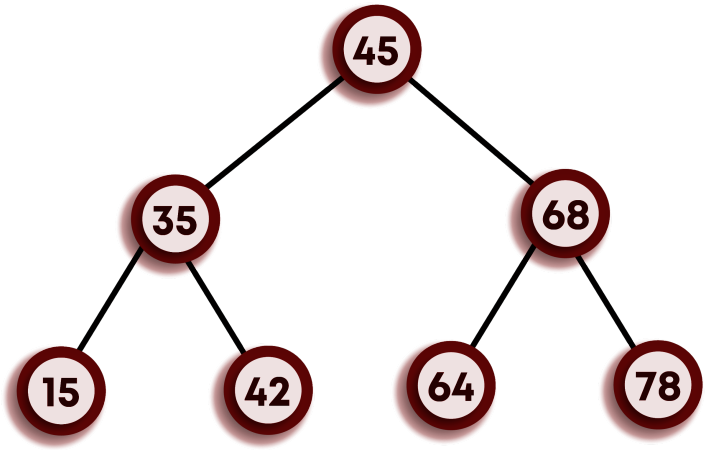
5. Now, to insert 15, we will follow the same approach starting from the root node. Here, 15 < 45, which means 15 will be a part of the left subtree. As 15 < 35, we will continue to move towards the left. As the left subtree of 35 is empty, so 15 will be the left child of 35.



6. Continuing further, to insert 64, we know that 64 > root node's data, but less than 68, hence 64 will be the left child of 68.



7. Finally, we have to insert 78. We can see that 78 > 45 and 78 > 68, so 78 will be the right child of 68.



In this way, the data is stored in a BST.

Note:

- If we follow the **inorder traversal** of the final BST, we will get the sorted array.
- As seen above, to insert an element in the BST, we will be traversing till either the left subtree's leaf node or right subtree's leaf node, in the worst-case scenario.

- Hence, the **time complexity of insertion** for each node is **O(log(H))** (where **H** is the height of the tree).
- For inserting **N** nodes, complexity will be **O(N*log(H))**.

Problem Statement: Search in BST

Given a BST and a target value(x), we have to return the binary tree node with data x if it is present in the BST; otherwise, return NULL.

Approach: As the given tree is BST, we can use the **binary search algorithm**. Using recursion will make it easier.

Base Case:

- If the tree is empty, it means that the root node is NULL, then we will simply return NULL as the node is not present.
- Suppose if root's data is equal to **x**, we don't need to traverse forward in this tree as the target value has been found out, so we will simply return the **root** from here.

Small Calculation:

- In the case of BST, we'll only check for the condition of binary search, i.e., if x is greater than the root's data, then we will make a recursive call over the right subtree; otherwise, the recursive call will be made on the left subtree.
- This way, we are entirely discarding half the tree to be searched as done in case of a binary search. Therefore, the time complexity of searching is **O(log(H))** (where H is the height of BST).

Recursive call: After figuring out which way to move, we can make recursive calls on either left or right subtree. This way, we will be able to search the given element in a BST.

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

Problem Statement: Print elements in a range

Given a BST and a range (L, R), we need to figure out all the elements of BST that are present in the given range inclusive of L and R.

Approach: We will be using recursion and binary searching for the same.

Base case: If the root is NULL, it means we don't have any tree to check upon, and we can simply return.

Small Calculation: There are three conditions to be checked upon:

- If the root's data lies in the given range, then we can print it.
- We will compare the root's data with the given range's maximum. If root's data is smaller than R, then we will have to traverse only the right subtree.
- Now, we will compare the root's data with the given range's minimum. If the root's data is greater than L, then we will traverse only the left subtree.

Recursive call: Recursive call will be made as per the small calculation part onto the left and right subtrees. In this way, we will be able to figure out all the elements in the range.

Note: Try to code this yourself, and refer to the solution tab in case of any doubts.

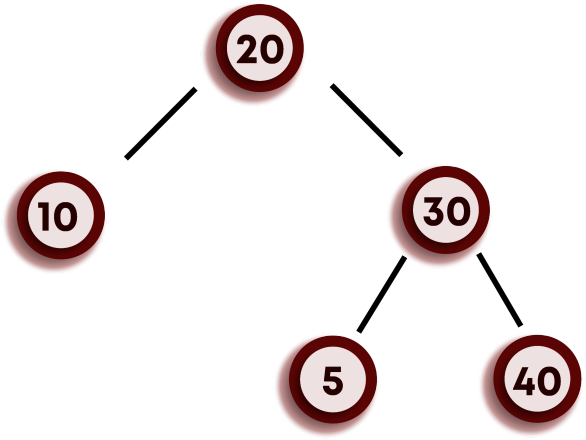
Problem Statement: Check BST

Given a binary tree, we have to check if it is a BST or not.

Approach: We will simply traverse the binary tree and check if the nodes satisfy the BST Property. Thus we will check the following cases:

- If the node's value is greater than the value of the node on it's left.
- If the node's value is smaller than the value of the node on its right.

Important Case: Don't just compare the direct left and right children of the node; instead, we need to compare every node in the left and right subtree with the node's value. Consider the following case:



- Here, it can be seen that for root, the left subtree is a BST, and the right subtree is also a BST (individually).
- But the complete tree is not a BST. This is because a node with value 5 lies on the right side of the root node with value 20, whereas it should be on the left side of the root node.
- Hence, even though the individual subtrees are BSTs, it is also possible that the complete binary tree is not a BST. Hence, this third condition must also be checked.

To check over this condition, we will keep track of the minimum and maximum values of the right and left subtrees correspondingly, and at last, we will simply compare them with root.

- The left subtree's maximum value should be less than the root's data.
- The right subtree's minimum value should be greater than the root's data.

Now, let's look at the Python code for this approach using this approach.

```
def minTree(root):#Returns minimum value in a subtree
      if (root == None):#If root is None, it implies empty tree
            return 100000 #Return a large number in this case
      leftMin = minTree(root.left) #Find min in left subtree
      rightMin = minTree(root.right) #Find min in right subtree
      return min(leftMin, rightMin, root.data) #Return overall min

def maxTree(root):#Returns minimum value in a subtree
      if (root == None): #If root is None, it implies empty tree
            return -100000 #Return a very small number in this case
      leftMax = maxTree(root.left) #Find max in left subtree
      rightMax = maxTree(root.right) #Find max in right subtree
      return max(leftMax, rightMax, root.data) #Return overall max

def isBST(root)
      if (root== None):#If root is None, it implies empty tree
            return True #Empty tree is a BST

      leftMax = maxTree(root.left) #Max in left subtree
      rightMin = minTree(root.right) #Min in right subtree
      if root.data > rightMin or root.data <= leftMax:
            return False #Checking BST Property

      isLeftBST = isBST(root.left)#Recursive call on left subtree
      isRightBST = isBST(root.right)#Recursive call on right subtree
      return isLeftBST and isRightBST #Both must be BST
```

Time Complexity: In the `isBST()` function, we are traversing each node, and for each node, we are then calculating the minimum and maximum value by again traversing that complete subtree's height. Hence, if there are **N** nodes in total and the height of the tree is **H**, then the time complexity will be **O(N*H)**.

Improved Solution for Check BST

- To improve our solution, observe that for each node, the minimum and maximum values are being calculated separately.

- We now wish to calculate these values, while checking the **isBST** condition itself, to get rid of another cycle of iterations.
- We will follow a similar approach as that of the diameter calculation of binary trees.
- At each stage, we will return the maximum value, minimum value, and the BST status (True/False) for each node of the tree, in the form of a tuple.

Let's look at its implementation now:

```
def isBST2(root):
      if root == None:
            return 100000, -100000, True

      leftMin, leftMax, isLeftBST= isBST2(root.left)
      rightMin, rightMax, isRightBST = isBST2(root.right)

      minimum= min(leftMin rightMin, root.data) #Minimum value
      maximum = max(leftMax, rightMax, root.data) #Maximum value
      isTreeBST=True

      #Checking the BST Property
      if root.data <= leftMax or root.data > rightMin:
            isTreeBST = False
      if not(isLeftBST) or not(isRightBST):
            isTreeBST = False

      return minimum, maximum, isTreeBST
```

Time Complexity: Here, we are going to each node and doing a constant amount of work. Hence, the time complexity for **N** nodes will be of **O(N)**.

Another Improved Solution for Check BST

The time complexity for this problem can't be improved further, but there is a better approach to this problem, which makes our code look more robust. Let's discuss that approach now.

Approach: We will be checking on the left subtree, right subtree, and combined tree without returning a tuple of maximum and minimum values. We will be using the concept of default arguments over here. Check the code below:

```
def isBST3(root, min_range, max_range):
      if root==None:#Empty tree is a BST
            return True

      if root.data < min_range or root.data > max_range:
            return False #Check the BST Property

      isLeftWithinConstraints = isBST3(root.left, min_range, root.data - 1)
      isRightwithinConstraints = isBST3(root.right, root.data, max_range)

      return isLeftWithinConstraints and isRightWithinConstraints
```

Time Complexity: Here also, we are just traversing each node and doing constant work on each of them; hence time complexity remains the same, i.e. **O(n)**.

Problem Statement: Construct BST from sorted array

Given a sorted array, we have to construct a BST out of it.

Approach:

- Suppose we take the first element as the root node, then the tree will be skewed as the array is sorted.
- To get a balanced tree (so that searching and other operations can be performed in **O(log(n))** time), we will be using the binary search technique.

- Figure out the middle element and mark it as the root.
- This is done so that the tree can be divided into almost 2 equal parts, as half the elements which will be greater than the root, will form the right subtree (These elements are present to its right).
- The elements in the other half, which are less than the root, will form the left subtree (These elements are present to its left).
- Just put root's left child to be the recursive call made on the left portion of the array and root's right child to be the recursive call made on the right portion of the array.
- Try it yourself and refer to the solution tab for code.

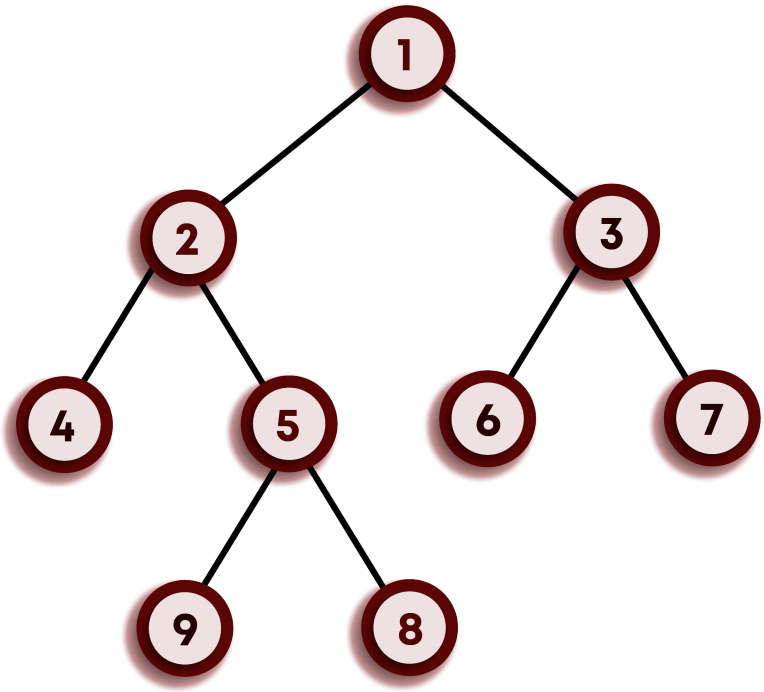
Binary Search Trees- 2

Root to Node Path in a Binary Tree

Problem statement:

Given a Binary tree, we have to return the path of the root node to the given node.

For Example: Refer to the image below...



Path from root to 8: [1 2 5 8]

Approach:

1. Start from the root node and compare it with the target value. If matched, then simply return, otherwise, recursively call over left and right subtrees.
2. Continue this process until the target node is found and return if found.

3. While returning, you need to store all the nodes that you have traversed on the path from the root to the target node in a list.
4. Now, in the end, you will be having your solution list.

Python Code:

```
def nodeToRootPath(root, s):
      if root == None: #Empty tree
            return None
      if root.data == s: #If data found at root node
            l = list() #Make a list and append -> Store the path
            l.append(root.data)
            return l

      leftOutput = nodeToRootPath(root.left, s)#Recursive call
      if leftOutput != None:
            leftOutput.append(root.data)
            return leftOutput

      rightOutput = nodeToRootPath(root.right, s)#Recursive call
      if rightOutput != None:
            rightOutput.append(root.data)
            return rightOutput
      else:
            return None
```

Now try to code the same problem with a BST instead of a binary tree.

BST class

Here, we will be creating our own BST class to perform operations like insertion, deletion, etc. Follow the given template for constructing your own BST Class. We will be creating a class with the given set of methods:

```
class BST:
      def __init__(self):#Constructor for the class
            self.root = None
            self.numNodes = 0

      def isPresent(self, data): #Check if an element is present
            return False

      def insert(self, data): #Insert a Node in BST
            return

      def deleteData(self, data): #Delete a Node
            return False

      def count(self):
            return self.numNodes
```

Search a Node in BST - `isPresent()`

The solution to this problem would be a recursive function that searches the node in the given BST, similar to what we have seen earlier. Go through the given code for better understanding:

```
def isPresentHelper(self, root, data):
      root == None:  # If the tree is empty, the node is not present
            return False

      root.data  == data: #If data is found at the root node
            return True

      if root.data>data: #call on left
            return isPresentHelper(root.left, data)
```

```
        else #call on right
                return isPresentHelper(root.right, data)

def isPresent(self, data): #Search a Node in the BST
        return isPresentHelper(self.root, data)
```

Insertion in BST:

We are given the root of the tree and the data to be inserted. Follow the same approach to insert the data as discussed above using Binary search algorithm. Check the code below for insertion in a BST:

```
def insertHelper(self, root, data):
        if root == None: #Empty tree
                node = BinaryTreeNode(data)
                return node

        if root.data >data: #Left recursion
                root.left = self.insertHelper(root.left, data)
                return root

        else: #Right recursion
                root.right = self.insertHelper(root.right, data)
                return root

def insert(self, data):
        self.numNodes +=1 # Update the number of nodes
        self.root = self.insertHelper(self.root, data) #Call helper
```

Deletion in BST:

Recursively, find the node to be deleted.

- **Case 1:** If the node to be deleted is the leaf node, then simply delete that node with no further changes and return **None**.
- **Case 2:** If the node to be deleted has only one child, then delete that node and return the child node.

- **Case 3:** If the node to be deleted has both the child nodes, then we have to delete the node such that the properties of BST remain unchanged. For this, we will replace the node's data with either the left child's largest node or the right child's smallest node and then simply delete the replaced node.

Now, let's look at the code below:

```
def min(self, root):
      if root == None:
            return 10000
      if root.left == None:
            return root.data
      return self.min(root.left)

def deleteData(self, data):
      deleted, newRoot = self.deleteDataHelper(self.root, data)
      if deleted:
            self.numNodes -= 1
      self.root = newRoot
      return deleted

def helper(self, root, data):
      if root = = None:
            return False, None
       #BST PROPERTY
      if root.data < data:
            deleted, newRightNode = self.helper(root.right, data)
            root.right = newRightNode
            return deleted, root

     if root.data > data:
            deleted, newleftNode = self.helper(root.left, data)
            root.left = newLeftNode
            return deleted, root

     # root is a leaf node
     if root.left == None and root.right == None:
            return True, None
```

```
        #root has one child
        if root.left == None:
             return True, root.right
        if root.right == None:
             return True, root.left

        # root has two children
        replacement = self.min(root.right)
        root.data = replacement
        deleted, newRightNode = self.helper(root.right, replacement)
        root.right = newRightNode
        return True, root
```

Types of Balanced BSTs

- For a balanced BST:

```
|Height_of_left_subtree - Height_of_right_subtree| <= 1
```

- This equation must be valid for every node present in the BST.
- By mathematical calculations, it was found that the height of a Balanced BST is **log(n)**, where **n** is the number of nodes in the tree.
- This can be summarised as the time complexity of operations like searching, insertion, and deletion can be performed in **O(log(n))**.
- Many BST types maintain balance. We will not be discussing them over here. These are as follows:
    - AVL Trees (also known as self-balancing BST, uses rotation to balance)
    - Red-Black Trees
    - 2 - 4 Tree

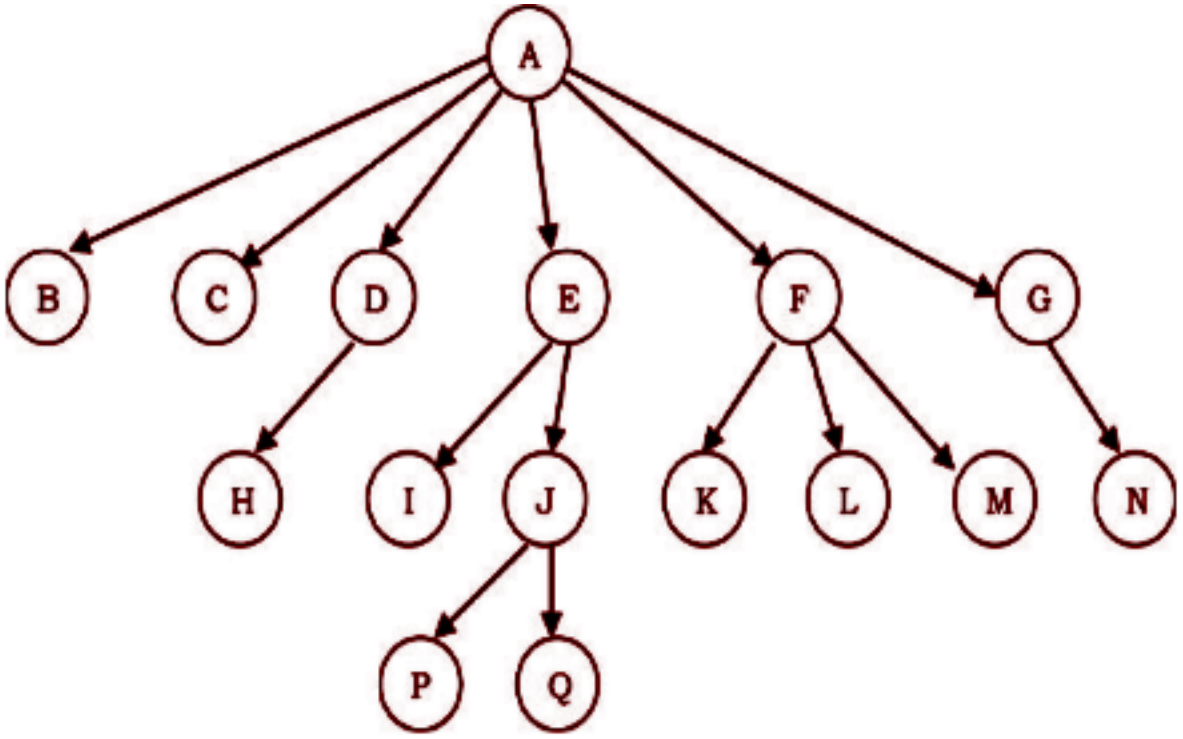
Practice Problems:

- https://www.hackerearth.com/practice/data-structures/trees/binary-search-tree/practice-problems/algorithm/dummy3-4/
- https://www.codechef.com/problems/KJCP01
- https://www.codechef.com/problems/BEARSEG

Generic Trees

Introduction

- In the previous modules, we discussed binary trees where each node can have a maximum of two children and these can be represented easily with two pointers i.e right child and left child.
- But suppose, we have a tree with many children for each node.
- If we do not know how many children a node can have, how do we represent such a tree?
- **For example**, consider the tree shown below.



Generic Tree Node

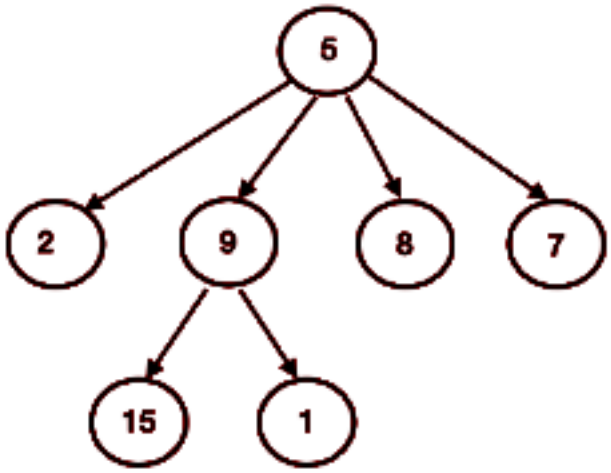
- Implementation of a generic tree node in Python is quite simple.
- The node class will contain two attributes:
    - The node **data**
    - Since there can be a variable number of children nodes, thus the second attribute will be a list of its children nodes. Each child is an instance of the same node class. This is a general **n-nary tree**.
- Consider the given implementation of the **Generic Tree Node** class:

```
class GenericTreeNode:
    def __init__ (self, data):
        self.data = data #Node data
        self.children = list() #List of children nodes
```

Adding Nodes to a Generic Tree

We can add nodes to a generic tree by simply using the **list.append()** function, to add children nodes to parent nodes. This is shown using the given example:

Suppose we have to construct the given Generic Tree:



Consider the given **Python code:**

```
#Create nodes
n1= TreeNode(5)
n2 =TreeNode(2)
n3 =TreeNode(9)
n4 =TreeNode(8)
n5 =TreeNode(7)
n6 =TreeNode(15)
n7 =TreeNode(1)

#Add children for node 5 (n1)
n1.children.append(n2)
n1.children.append(n3)
n1.children.append(n4)
n1.children.append(n5)

#Add children for node 9 (n3)
n3.children.append(n6)
n3.children.append(n7)
```

Print a Generic Tree (Recursively)

In order to print a given generic tree, we will recursively traverse the entire tree.

The steps shall be as follows:

- If the root is **None**, i.e. the tree is an empty tree, then return **None**.
- For every child node at a given node, we call the function recursively.

Go through the given Python code for better understanding:

```
def printTree(root):
      #Not a base case but an edge case
      if root == None:
            return

      print(root.data) #Print current node's data
      for child in root.children:
            printTree(child) #Recursively call the function for children
```

Take Generic Tree Input (Recursively)

Go through the given Python code for better understanding:

```
def takeTreeInput():
      print("Enter root Data")
      rootData  = int(input())#TAKE USER INPUT
      if rootData == -1:#Stop taking inputs
            return None

      root = TreeNode(rootData)

      print("Enter number of children for  ",  rootData)
      childrenCount = int(input()) #Get input for no. of child nodes
      for in range(childrenCount):
            child = takeTreeInput() #Input for all childs
            root.children.append(child) #Add child
      return root
```

Dictionaries/Maps

Hash-Maps in Python

In computer science, a Hash table or a Hashmap is a type of data structure that maps keys to its value pairs (implement abstract array data types). Suppose we are given a string or a character array and asked to find the maximum occurring character. It could be quickly done using arrays.

- We can simply create an array of size 256.
- Initialize this array to zero.
- Traverse the array to increase the count of each character against its ASCII value in the frequency array.
- In this way, we will be able to figure out the maximum occurring character in the given string.

The above method will work fine for all the 256 characters whose ASCII values are known. But what if we want to store the maximum frequency string out of a given array of strings? It can't be done using a simple frequency array, as the strings do not possess any specific identification value like the ASCII values. For this, we will be using a different data structure called **hashmaps**.

In **hashmaps**, the data is stored in the form of keys against which some value is assigned. **Keys** and **values** don't need to be of the same data type.

If we consider the above example in which we were trying to find the maximum occurring string, using hashmaps, the individual strings will be regarded as keys, and the value stored against them will be considered as their respective frequency.

For example, The given string array is:

```
str[] = {"abc", "def", "ab", "abc", "def", "abc"}
```

Hashmap will look like follows:

Key (datatype : string)	Value (datatype : int)
“abc”	3
“def”	2
“ab”	1

From here, we can directly check for the frequency of each string and hence figure out the most frequent string among them all.

Note: *One more limitation of arrays is that the indices could only be whole numbers but this limitation does not hold for hashmaps.*

When it comes to Python, Hash tables are used via **dictionary** ie, the built-in data type.

Dictionaries

- A Dictionary is a Python’s implementation of a data structure that is more generally known as an associative array.
- A dictionary is an unordered collection of key-value pairs.
- Indexing in a dictionary is done using these **“keys”.**
- Each pair maps the key to its value.
- Literals of the type **dictionary** are enclosed within curly brackets.
- Within these brackets, each entry is written as a key followed by a colon ” : ”, which is further followed by a value. This is the value that corresponds to the given key.
- These keys must be unique and cannot be any immutable data type. **Eg-** string, integer, tuples, etc. They are always mutable.
- The values need not be unique. They can be repetitive and can be of any data type (Both mutable and immutable)
- Dictionaries are mutable, which means that the key-value pairs can be changed.

What does a dictionary look like?

This is the general representation of a dictionary. Multiple key-value pairs are enclosed within curly brackets, separated by commas.

```
myDictionary = {
    <key>: <value>,
    <key>: <value>,
      .
      .
    <key>: <value>
}
```

Creating A Dictionary

Way 1- The Basic Approach

Simply put pairs of key-value within curly brackets. Keys are separated from their corresponding values by a colon. Different key-value pairs are separated from each other by commas. Assign this to a variable.

```
myDict= {"red":"boy", 6: 4, "name":"boy"}
months= {1:"January", 2:"February", 3: "March"}
```

Way 2- Type Casting

This way involves type casting a list into a dictionary. To typecast, there are a few mandates to be followed by the given list.

- The list must contain only tuples.
- These tuples must be of length **2**.
- The first element in these tuples must be the key and the second element is the corresponding value.

This type casting is done using the keyword **`dict`.**

```
>>> myList= [("a",1),("b",2),("c",2)]
>>> myDictionary= dict(myList)
>>> print(myDictionary)
{"a":1,"b":2,"c":2}
```

Way 3- Using inbuilt method `.fromkeys()`

This method is particularly useful if we want to create a dictionary with variable keys and all the keys must have the same value. The values corresponding to all the keys are exactly the same. This is done as follows:

```
>>> d1= dict.fromkeys(["abc",1,"two"])
>>> print(d1)
{"abc":None ,1: None, "two": None}
# All the values are initialized to None if we provide only one argument
i.e. a list of all keys.
```

```
# We can initialise all the values to a custom value too. This is done by
providing the second argument as the desired value.
>>> d2= dict.fromkeys(["abc",1,"two"],6)
>>> print(d2)
["abc":6 ,1:6, "two":6]
```

How to access elements in a dictionary?

We already know that the indexing in a dictionary is done with the keys from the various key-value pairs present within. Thus to access any value we need to use its index i.e. it's **key**.

Similar to the list and tuples, this can be done by the square bracket operator **[ ]**.

```
foo= {"a":1,"b":2,"c":3}
print(foo["a"])
--> 1
print(foo["c"])
--> 3
```

If we want the value corresponding to any key, we can even use an inbuilt dictionary method called **`get`**.

```
>>> foo= {"a":1,"b":2,"c":3}
```

```
>>> print(foo.get("a"))
1
>>> print(foo.get("c"))
3
```

A very unique feature about this method is that , in case the desired **key** is not present in the dictionary , it won't throw an error or an exception. It would simply return `None`.

We can make use of this feature in another way. Say we want the method to do the following action: If the key is present in the dictionary then return the value corresponding to the key. In case the key is not present, return a custom desired value (say 0).

This can be done as follows:

```
>>> foo= {"a":1,"b":2,"c":3}
>>> print(foo.get("a",0))
1
>>> print(foo.get("d",0))
0
```

Accessing all the available keys in the dictionary:

This can be done using the method `.keys()`. This method returns all the different keys present in the dictionary in the form of a list.

```
>>> foo= {"a":1,"b":2,"c":3}
>>> foo.keys()
dict_keys(["a","b","c"])
```

Accessing all the available values in the dictionary:

This can be done using the method `.values()`. This method returns all the different values present in the dictionary in the form of a list.

```
>>> foo= {"a":1,"b":2,"c":3}
>>> foo.values()
dict_values([1,2,3])
```

Accessing all the available items in the dictionary:

This can be done using the method `.items()`. This method returns all the different items (key-value pairs) present in the dictionary in the form of a list of tuples, with the first

element of the tuple as the key and the second element as the value corresponding to this key.

```
>>> foo= {"a":1,"b":2,"c":3}
>>> foo.items()
dict_items([("a",1),("b",2),("c",3)])
```

Checking if the dictionary contains a given key:

The keyword used is `in`. We can easily get a boolean output using this keyword. It returns True if the dictionary contains the given **key**, else the output is False. This checks the presence of the keys and not the presence of the values.

```
>>> foo= {"a":1,"b":2,"c":3}
>>> "a" in foo
True
>>> 1 in foo
False
```

Iterating over a Dictionary:

To traverse through the dictionary, we can use a simple for loop. The loop will go through the keys of the dictionary one by one and do the required action.

```
bar= {2:1,3:2,4:3}
for t in bar:
      print(t)
Out[]:
2
3
4
# Here t is the key in the dictionary and hence when we print t in all
iterations then all the keys are printed.
```

```
for t in bar:
      print(t, bar[t])
Out[]:
2 1
3 2
```

```
4 3
# Here along with the keys, the values which are bar[t] are printed. In
this loop, the values are accessed using the keys.
```

```
for key,item in dict.items():
      print(key, item)
Out[]:
2 1
3 2
4 3
# Here along with the keys, the values are printed.
```

Adding Elements In a Dictionary

Since a dictionary is mutable, we can add or delete entries from the dictionary. This is particularly useful if we have to maintain a dynamic data structure. To assign a value corresponding to a given key *(This includes over-writing the value present in the key or adding a new key-value pair)*, we can use the square bracket operators to simply assign the value to a key.

If we want to update the value of an already existing key in the dictionary then we can simply assign the new value to the given key. This is done as follows:

```
>>> bar= {2:1,3:2,4:3}
>>> bar[3]=4
# This operation updates the value of the key 3 to a new value i.e. 4.
>>> print(bar)
{2:1,3:4,4:3}
```

Now if we want to add a new key-value pair to our dictionary, then we can make a similar assignment. If we have to add a key-value pair as `"man":"boy"`, then we can make the assignment as:

```
>>> bar["man"]="boy"
>>> print(bar)
{2:1,3:2,4:3,"man":"boy"}
```

Adding or concatenation of two dictionaries:

If we have 2 dictionaries and we want to merge the contents of both the dictionaries and form a single dictionary out of it . It is done as follows:

```
a= {1:2,2:3,3:4}
b= {7:2,10:3,6:4}
a.update(b)
print(a)
--> {1:2,2:3,3:4,7:2,10:3,6:4}
```

In this process, the second dictionary is unchanged and the contents of the second dictionary are copied into the first dictionary. The uncommon keys from the second dictionary are added to the first with their corresponding values. However, if these dictionaries have any common key, then the value of the common key present in the first dictionary is updated to the new value from the second.

Deleting an entry:

To delete an entry corresponding to any key in a dictionary, we can simply pop the key from the dictionary. The method used here is `.pop()`. This method removes the key-value pair corresponding to any particular key and then returns the value of the removed key-value pair.

```
>>> c={1:2,2:(3,23,3),3:4}
>>> c.pop(2)
(3,23,3)
```

Deleting all the entries from the dictionary:

If we want to clear all the key-value pairs from the given dictionary and thus convert it into an empty dictionary we can use the `.clear()` method.

```
>>> c={1:2,2:(3,23,3),3:4}
>>> c.clear()
>>> print(c)
{}
```

Deleting the entire dictionary:

We can even delete the entire dictionary from the memory by using the `del` keyword. This would remove the presence of the dictionary. This is similar to tuples and lists.

Problem statement: Print all words with frequency k.

Approach to be followed:

First, we convert the given string of words into a list containing all the words individually. Some of these words are repetitive and to find all the words with a specific frequency, we convert this list into a dictionary with all the unique words as keys and their frequencies or the number of times they occur as their values.

To convert the string to a list, we use the `.split()` function. This gives us a list of words. Now, we run a loop through this list and keep making changes to the frequency in the dictionary. If the word in the current iteration already exists in the dictionary as a key, then we simply increase the value(or frequency) by 1. If the key does not exist, we create a new key-value pair with value as 1.

Now we have a dictionary with unique keys and their respective frequencies. Now we run another loop to print the keys with the frequency '**k**'.

Given below is a function that serves this purpose.

```
def printKFreqWords(string, k):
      # Converting the input string to a list
      myList= string.split()
      # Initialise an empty dictionary
      dict= {}
      # Iterate through the list in order to find frequency
      for i in myList:
            dict[i]=dict[i]+1
      else:
```

```
            dict[i]=1
      # Loop for printing the keys with frequency as k
      for t in dict:
            if dict[t]==k:
                  print(t)
```

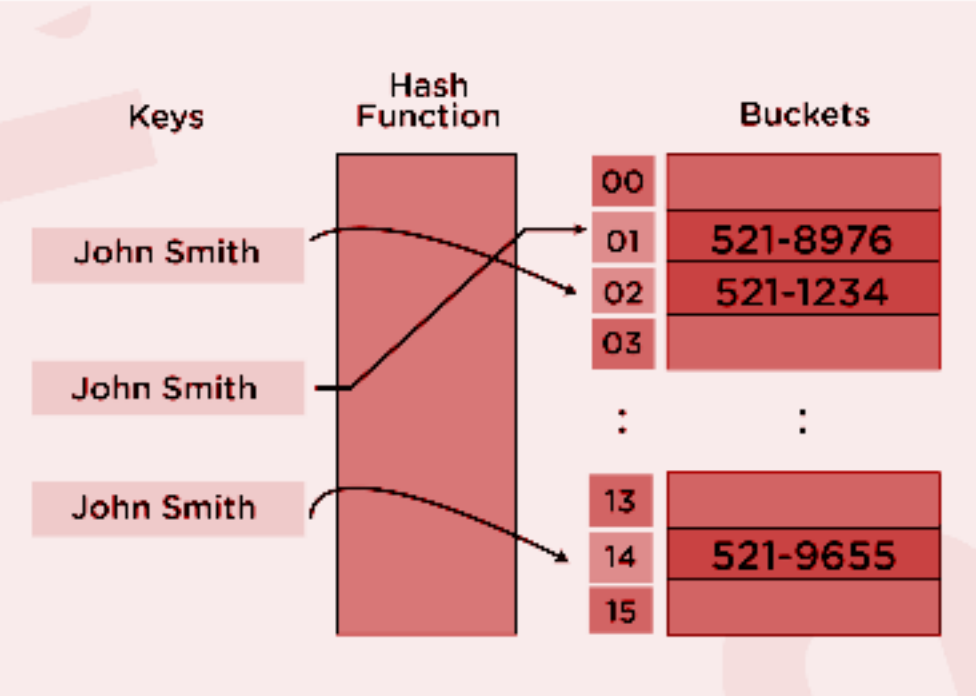
Implementing Our Own Hashmap

Bucket array and Hash function

Now, let's see how to perform insertion, deletion, and search operations using hash tables. Till now, we have seen that arrays are the fastest way to extract data as compared to other data structures as the time complexity of accessing the data in the array is O(1). So we will try to use them in implementing the hashmaps.

Now, we want to store the key-value pairs in an array, named **bucket array**. We need an integer corresponding to the key so that we can keep it in the bucket array. To do so, we use a **hash function**. A hash function converts the key into an integer, which acts as the index for storing the key in the array.

For example: Suppose, we want to store some names from the contact list in the hash table, check out the following image:



Suppose we want to store a string in a hash table, and after passing the string through the hash function, the integer we obtain is equal to 10593, but the bucket array's size is only 20. So, we can't store that string in the array as 10593, as this index does not exist in the array of size 20.

To overcome this problem, we will divide the hashmap into two parts:

- Hash code
- Compression function

The first step to store a value into the bucket array is to convert the key into an integer (this could be any integer irrespective of the size of the bucket array). This part is achieved by using a **hashcode**. For different types of keys, we will be having different kinds of hash codes. Now we will pass this value through the compression function, which will convert that value within the range of our bucket array's size. Now, we can directly store that key against the index obtained after passing through the compression function.

The compression function can be used as (`%bucket_size`).

One example of a hash code could be: (Example input: `"abcd"`)

```
"abcd" = ('a' * p^3) + ('b' * p^2) + ('c' * p^1) + ('d' * p^0)
#Where p is generally taken as a prime number so that they are well distributed.
```

But, there is still a possibility that after passing the key through from hash code, when we give the same through the compression function, we can get the same values of indices. For example, let s1 = “ab” and s2 = “cd”. Now using the above hash function for p = 2, h1 = 292 and h2 = 298. Let the bucket size be equal to 2. Now, if we pass the hash codes through the compression function, we will get:

```
Compression_function1 = 292 % 2 = 0
Compression_function2 = 298 % 2 = 0
```

This means they both lead to the same index 0.

This is known as a **collision**.

Collision Handling

We can handle collisions in two ways:

- Closed hashing (or closed addressing)
- Open addressing

In closed hashing, each entry of the array will be a linked list. This means it should be able to store every value that corresponds to this index. The array position holds the address to the head of the linked list, and we can traverse the linked list by using the head pointer for the same and add the new element at the end of that linked list. This is also known as **separate chaining**.
On the other hand, in open addressing, we will check for the index in the bucket array if it is empty or not. If it is empty, then we will directly insert the key-value pair

over that index. If not, then will we find an alternate position for the same. To find the alternate position, we can use the following:

$h_i(a) = hf(a) + f(i)$

Where **hf(a)** is the original hash function, and **f(i)** is the **i^{th}** try over the hash function to obtain the final position **$h_i(a)$**.

To figure out this **f(i)**, the following are some of the techniques:

1. **Linear probing:** In this method, we will linearly probe to the next slot until we find the empty index. Here, `f(i) = i`.
2. **Quadratic probing:** As the name suggests, we will look for alternate **i^2** positions ahead of the filled ones, i.e., **$f(i) = i^2$**.
3. **Double hashing:** According to this method, `f(i) = i * H(a)`, where H(a) is some other hash function.

In practice, we generally prefer to use **separate chaining** over **open addressing**, as it is easier to implement and is also more efficient.

Let's now implement the hashmap of our own.

Hashmap Implementation - Insert

As discussed earlier, we will be implementing separate chaining. We will be using value as a template and key as a string as we are required to find the hash code for the key. Taking the key as a template will make it difficult to convert it using hash code.

Let's look at the code for the same.

```
class MapNode:
      def __init__  (self,key,value):
            self.key = key #to store key
            self.value = value #to store value
```

```
            self.next = None #to the next pointer

class Map:

      def  __init__ (self):
            self.bucketSize = 10#Buckets for compression function
            # Store the head pointers
            self.buckets = [None for i in range(self.bucketSize)]
            self.count = 0 #To store the size

      def size(self):#Return the size of the map
            return self.count

      def getBucketIndex(self,hc):#Index using hash function
            return (abs(hc)%(self.bucketSize))

     def insert(self,key,value):
            hc = hash(key)
            index = self.getBucketIndex(hc)
            head = self.buckets[index]
            while head is not None:
                  if head.key == key:
                        head.value = value
                        return
                  head = head.next

            newNode = MapNode(key,value)
            newNode.next  = head
            self.buckets[index] = newNode
            self.count+=1
```

NOTE:

- The **hash()** method returns the hash value of an object if it has one. Hash values are just integers that are used to compare dictionary keys during a dictionary lookup quickly.

- Internally, **hash()** method calls **__hash__()** method of an object which is set by default for any object.
- The syntax of **hash()** method is:

```
hash(object)
```

Hashmap Implementation - Search and Remove

Go through the given implementation of searching for a key in a map and removing it:

```
def remove(self,key):
      hc = hash (key) #Getting hashcode
      index= self.getBucketIndex(hc) #Finding index corresponding to hc
      head = self.buckets[index] #Finding head pointer
      prev= None
      while head is not None:
            if head.key == key:#If found
                  if prev is None: #Removing
                        self.buckets[index] = head.next
                  else:
                        prev.next= head.next
                  return head.value
            prev =head
            head =head.next
      return None
```

Time Complexity and Load Factor

Let's define a few specific terms before moving forward:

1. **n** = Number of entries in our map.
2. **l** = length of the word (in case of strings)

3. **b** = number of buckets. On average, each box contains **(n/b)** entries. This is known as **load factor** (This means **b** boxes contain **n** entries). We also need to ensure that the load factor is always less than 0.7, i.e., **`(n/b) < 0.7(This will ensure that each bucket does not contain too many entries in it).`**
4. To make sure that load factor < 0.7, we can't reduce the number of entries, but we can increase the bucket size comparatively to maintain the ratio. This process is known as **Rehashing**.

This ensures that time complexity is on an average **O(1)** for insertion, deletion, and search operations each.

Rehashing

Now, we will try to implement the rehashing in our map. After inserting each element into the map, we will check the load factor. If the load factor's value is greater than 0.7, then we will rehash. Refer to the code below for better understanding.

```
def rehash(self):
      temp= self.buckets #To store the old bucket
      self.buckets = [None for i in range(2*self.bucketSize))
      self.bucketSize = 2*self.bucketSize #doubling the size
      self.count =0
      for head in temp:#inserting each value of old bucket to new one
            while head is not None:
                  self.insert(head.key,head.value)
                  head = head.next
```

```
      def insert(self,key,value):#Modify the insert function

            hc = hash(key)
            index = self.getBucketIndex(hc)
            head = self.buckets[index]
            while head is not None:
```

```
                if head.key == key:
                        head.value = value
                        return
                head = head.next

        newNode = MapNode(key,value)
        newNode.next  = head
        self.buckets[index] = newNode
        self.count+=1
        # Now we will check the load factor after insertion.
        loadFactor = self.count/self.bucketSize
        if loadFactor>=0.7:
                self.refash()
```

Practice problems:

- https://www.codechef.com/problems/STEM
- https://codeforces.com/problemset/problem/525/A
- https://www.spoj.com/problems/ADACLEAN/

Priority Queues- 1

Introduction

- Priority Queues are abstract data structures where each data/value in the queue has a certain priority.
- A priority queue is a special type of queue in which each element is served according to its priority.
- If elements with the same priority occur, they are served according to their order in the queue.
- Generally, the value of the element itself is considered for assigning the priority.
- **For example**, the element with the highest value is considered as the highest priority element. However, in some cases, we may assume the element with the lowest value to be the highest priority element. In other cases, we can set priorities according to our needs.

Difference between Priority Queue and Normal Queue

In a queue, the **First-In-First-Out(FIFO)** rule is implemented whereas, in a priority queue, the values are removed based on priority. The element with the highest priority is removed first.

Main Priority Queues Operations

- **Insert (key, data)**: Inserts data with a key to the priority queue. Elements are ordered based on key.
- **DeleteMin/DeleteMax**: Remove and return the element with the smallest/largest key.
- **GetMinimum/GetMaximum**: Return the element with the smallest/largest key without deleting it.

Auxiliary Priority Queues Operations

- **kth - Smallest/kth – Largest**: Returns the kth -Smallest/kth –Largest key in the priority queue.
- **Size**: Returns the number of elements in the priority queue.
- **Heap Sort**: Sorts the elements in the priority queue based on priority (key).

Priority Queue Applications

Priority queues have many applications - a few of them are listed below:

- **Data compression:** Huffman Coding algorithm
- **Shortest path algorithms**: Dijkstra's algorithm
- **Minimum spanning tree algorithms**: Prim's algorithm
- **Event-driven simulation**: Customers in a line
- **Selection problem:** Finding the kth- smallest element

Priority Queue Implementations

Before discussing the actual implementation, let us enumerate the possible options.

Unordered Array Implementation

- Elements are inserted into the array without bothering about the order. Deletions (DeleteMax) are performed by searching the key and then deleting.
- Insertions complexity: **O(1)**.
- DeleteMin complexity: **O(n)**

Unordered List Implementation

- It is very similar to array implementation, but instead of using arrays, linked lists are used.
- Insertions complexity: **O(1)**.
- DeleteMin complexity: **O(n)**.

Ordered Array Implementation

- Elements are inserted into the array in sorted order based on the key field. Deletions are performed at only one end.
- Insertions complexity: **O(n)**; DeleteMin complexity: **O(1)**.

Ordered List Implementation

- Elements are inserted into the list in sorted order based on the key field. Deletions are performed at only one end, hence preserving the status of the priority queue. All other functionalities associated with a linked list ADT are performed without modification.
- Insertions complexity: **O(n)**; DeleteMin complexity: **O(1)**.

Binary Search Trees Implementation

- Both insertions and deletions take **O(log(n))** on average if insertions are random (refer to Trees chapter).

Balanced Binary Search Trees Implementation

- Both insertions and deletion take **O(log(n))** in the worst case (refer to Trees chapter).

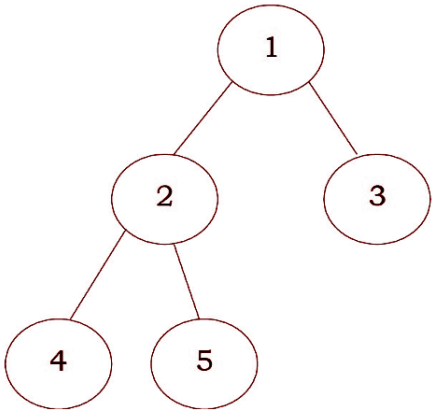
Binary Heap Implementation

In subsequent sections, we will discuss this in full detail.

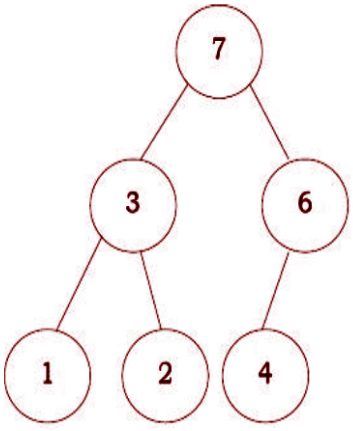
Implementation	Insertion	Deletion (DeleteMax)	Find Min
Unordered array	1	n	n
Unordered list	1	n	n
Ordered array	n	1	1
Ordered list	n	1	1
Binary Search Trees	$\log n$ (average)	$\log n$ (average)	$\log n$ (average)
Balanced Binary Search Trees	$\log n$	$\log n$	$\log n$
Binary Heaps	$\log n$	$\log n$	1

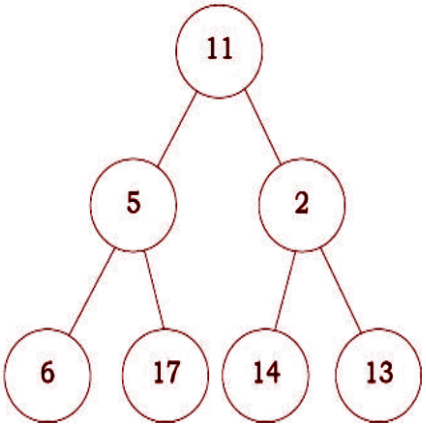
Heaps

- A heap is a tree with some special properties.
- The basic requirement of a heap is that the value of a node must be ≥ (or ≤) than the values of its children. This is called the **heap property**.
- A heap also has the additional property that all leaf nodes should be at **h** or **h – 1** level (where h is the height of the tree) for some h > 0 (complete binary trees).
- That means the heap should form a complete binary tree (as shown below).



In the examples below, the left tree is a heap (each element is greater than its children) and the right tree is not a heap (since 11 is greater than 2).

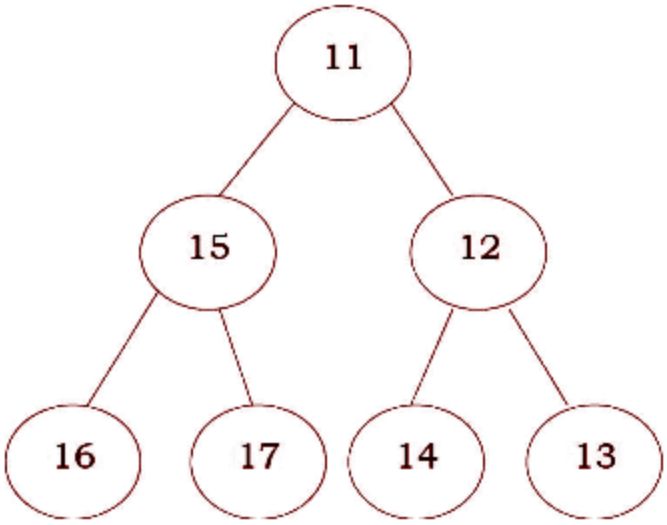




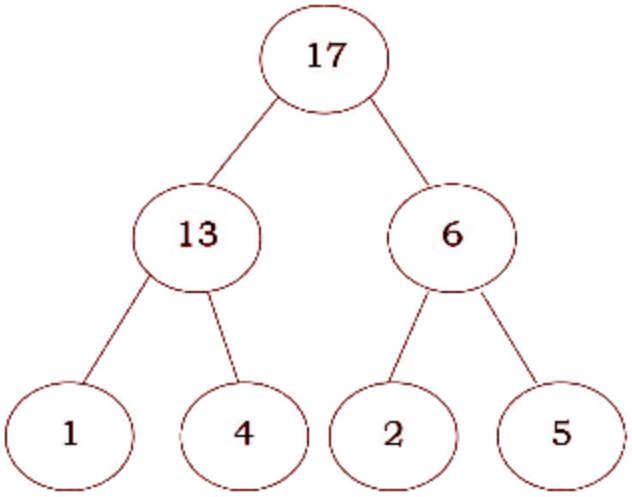
Types of Heaps

Based on the property of a heap we can classify heaps into two types:

- **Min heap:** The value of a node must be less than or equal to the values of its children.



- **Max heap:** The value of a node must be greater than or equal to the values of its children.



Binary Heaps

- In a binary heap, each node may have up to two children.

- In practice, binary heaps are enough and we concentrate on binary min heaps and binary max heaps for the remaining discussion.

Representing Heaps: Before looking at heap operations, let us see how heaps can be represented. One possibility is using arrays. Since heaps are forming complete binary trees, there will not be any wastage of locations. For the discussion below let us assume that elements are stored in arrays, which starts at index 0. The previous max heap can be represented as:

Heap Operations

Some of the important operations performed on a heap are described below along with their algorithms.

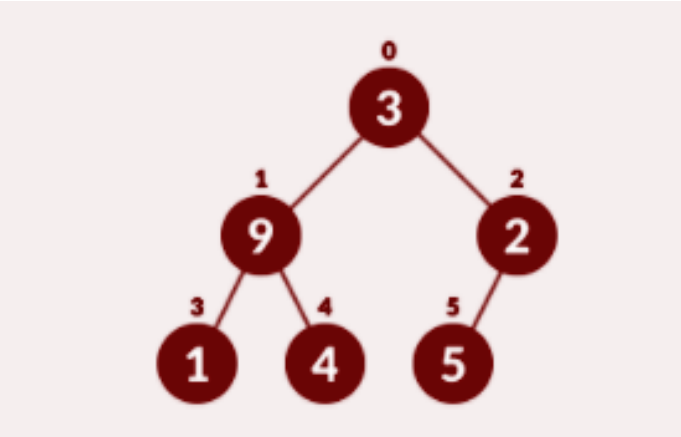
Heapify

Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.

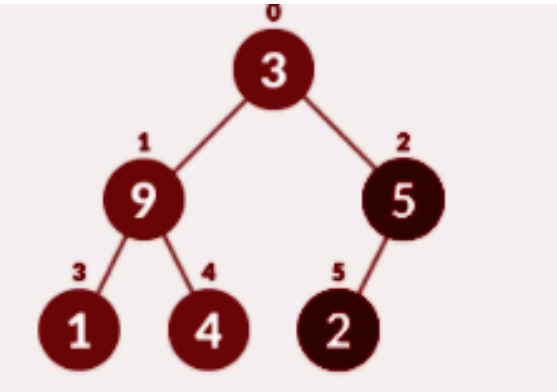
- Let the input array be

- Create a complete binary tree from the array
- Start from the first index of the non-leaf node whose index is given by **n/2 - 1**.

- Set current element **i** as **largest**.



- The index of the left child is given by **2i + 1** and the right child is given by **2i + 2**.
- If **leftChild** is greater than **currentElement** (i.e. element at the ith index), set **leftChildIndex** as largest. `#Condition1`



- If **rightChild** is greater than element in **largest**, set **rightChildIndex** as **largest**. `#Condition2`
- Swap **largest** with **currentElement.** `#Condition3`
- Repeat steps 3-7 until the subtrees are also heapified.
- For Min-Heap, both **leftChild** and **rightChild** must be smaller than the parent for all nodes.

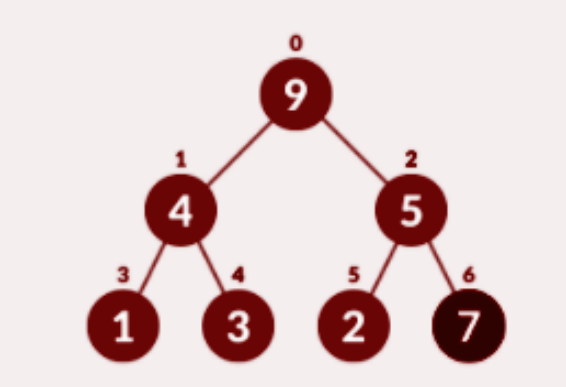
Python Code

```python
def heapify(arr, n, i):
    largest = i
    l = 2 * i + 1 #Index of Left Child
    r = 2 * i + 2 #Index of Right Child

    if l < n and arr[i] < arr[l]: #Condition1
        largest = l

    if r < n and arr[largest] < arr[r]: #Condition2
        largest = r

    if largest != i: #Condition3
        arr[i],arr[largest] = arr[largest],arr[i]
        heapify(arr, n, largest)
```

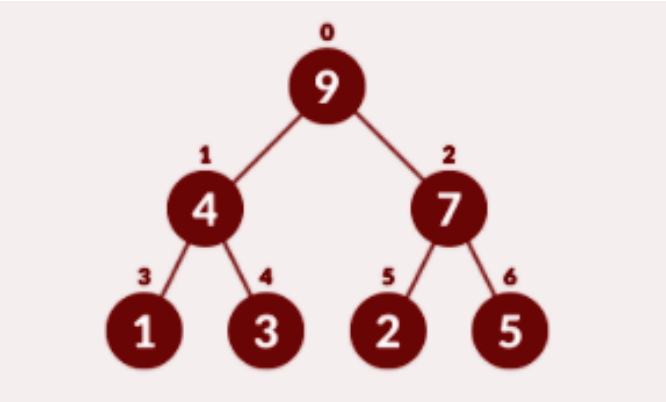
Insert Element into Heap

Insertion into a heap can be done in two steps:

- Insert the new element at the end of the tree. `#Step1`



- Heapify the tree. `#Step2`



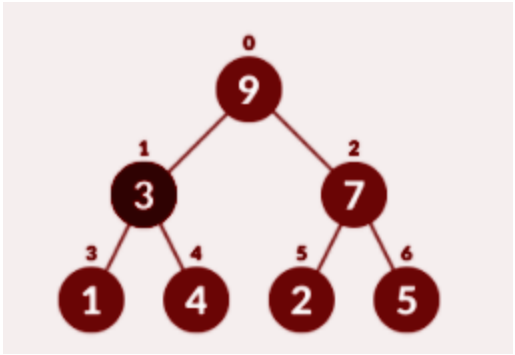
Python Code

```
def insert(array, newNum):
    size = len(array)
    if size == 0:#If empty heap initially
        array.append(newNum) #Simply add the newNum
    else:
        array.append(newNum);#Step1
        for i in range((size//2)-1, -1, -1):
            heapify(array, size, i) #Step2
```

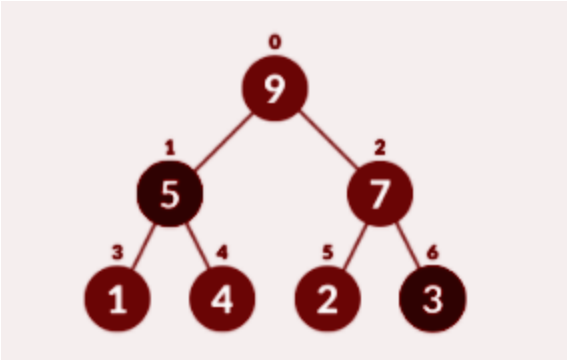
Delete Element from Heap

Follow the given steps to delete an element from a Heap:

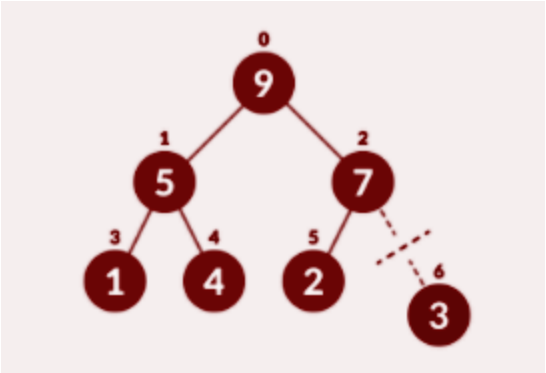
- Select the element to be deleted. `#Step1`



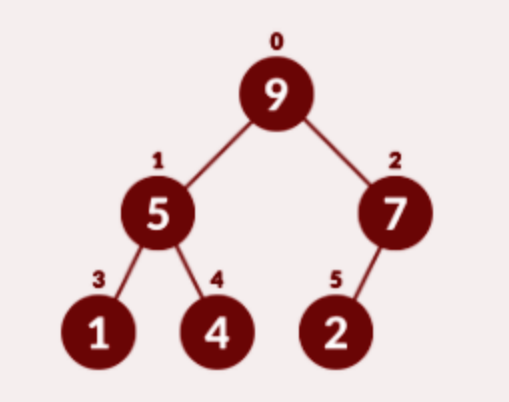
- Swap it with the last element.`#Step2`



- Remove the last element .`#Step3`



- Heapify the tree. `#Step4`



Python Code

```
def deleteNode(array, num):
    size = len(array)
    i = 0
    for i in range(0, size):
        if num == array[i]: #Step1
            break

    array[i], array[size-1] = array[size-1], array[i] #Step2
    array.remove(size-1) #Step3
    for i in range((len(array)//2)-1, -1, -1):
        heapify(array, len(array), i) #Step4
```

Implementation of Priority Queue

- Priority queue can be implemented using an array, a linked list, a heap data structure, or a binary search tree.
- Among these data structures, heap data structure provides an efficient implementation of priority queues.
- Hence, we will be using the heap data structure to implement the priority queue in this tutorial.

Insertion and Deletion in a Priority Queue

The first step would be to represent the priority queue in the form of a max/min-heap. Once it is heapified, the insertion and deletion operations can be performed similar to that in a Heap. Refer to the codes discussed above for more clarity.

Priority Queues- 2

Heap Sort

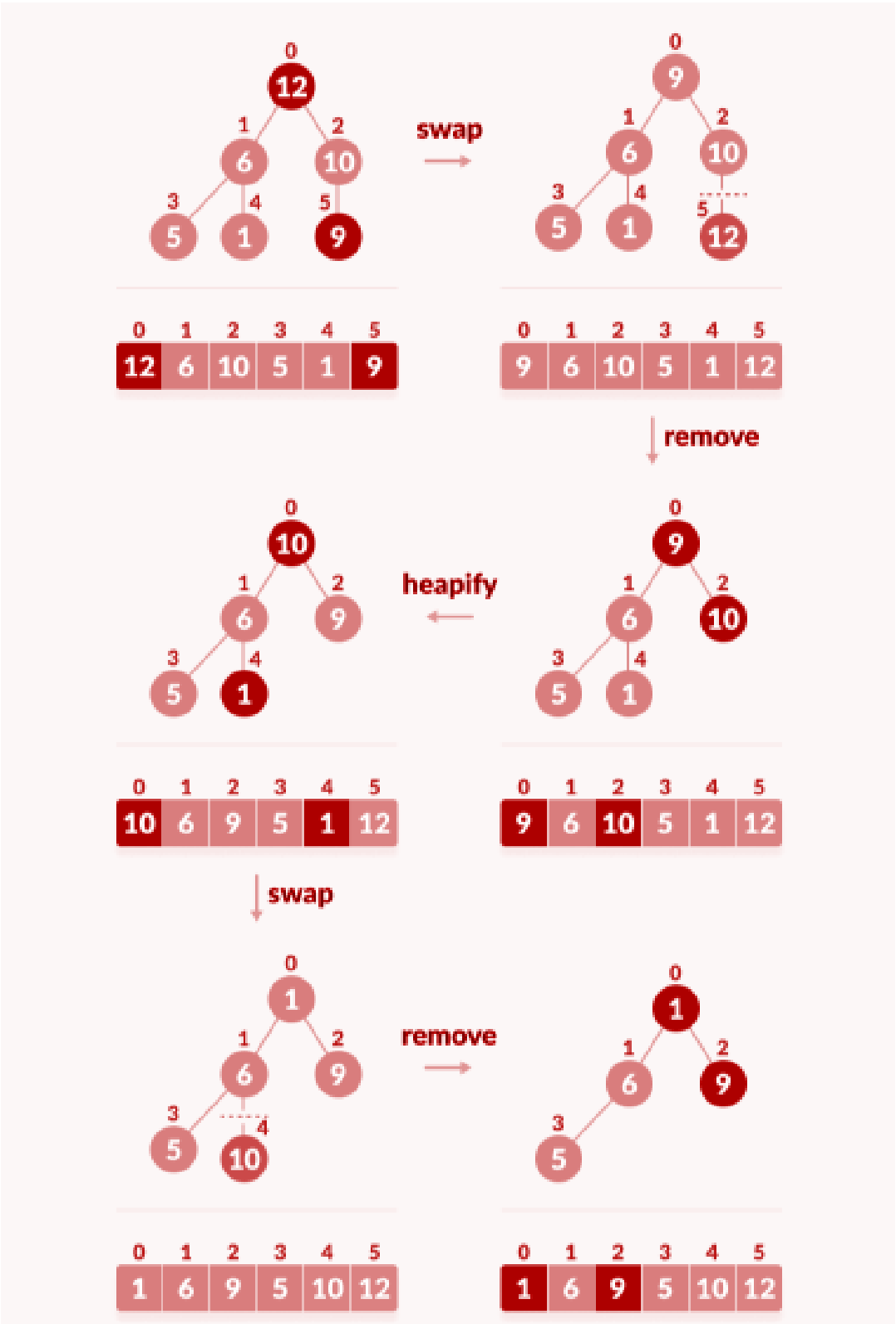
- Heap Sort is another example of an efficient sorting algorithm. Its main advantage is that it has a great worst-case runtime of **O(nlog(n))** regardless of the input data.
- As the name suggests, Heap Sort relies heavily on the heap data structure - a common implementation of a Priority Queue.
- Without a doubt, Heap Sort is one of the simplest sorting algorithms to implement, and coupled with the fact that it's a fairly efficient algorithm compared to other simple implementations, it's a common one to encounter.

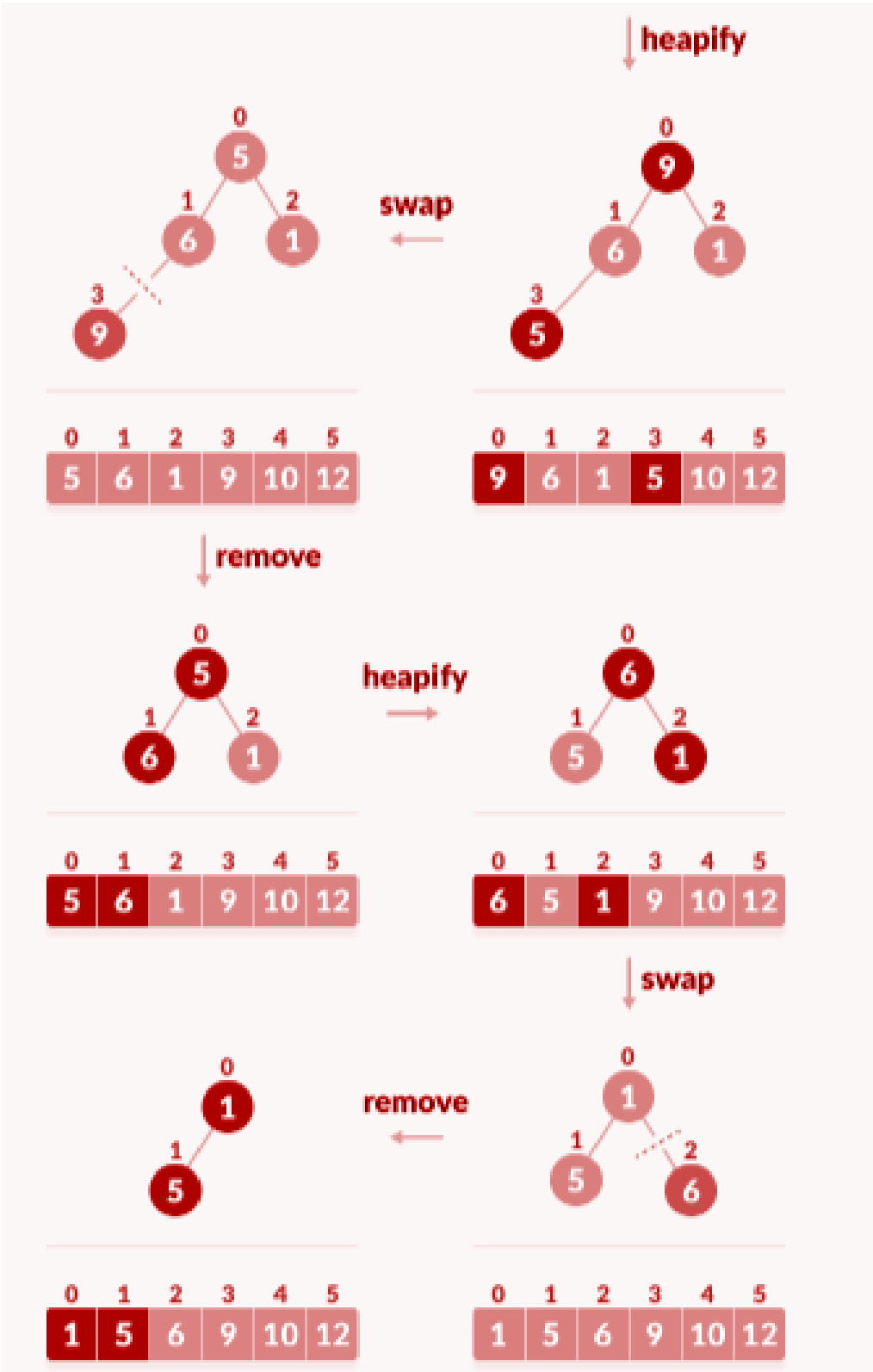
Algorithm

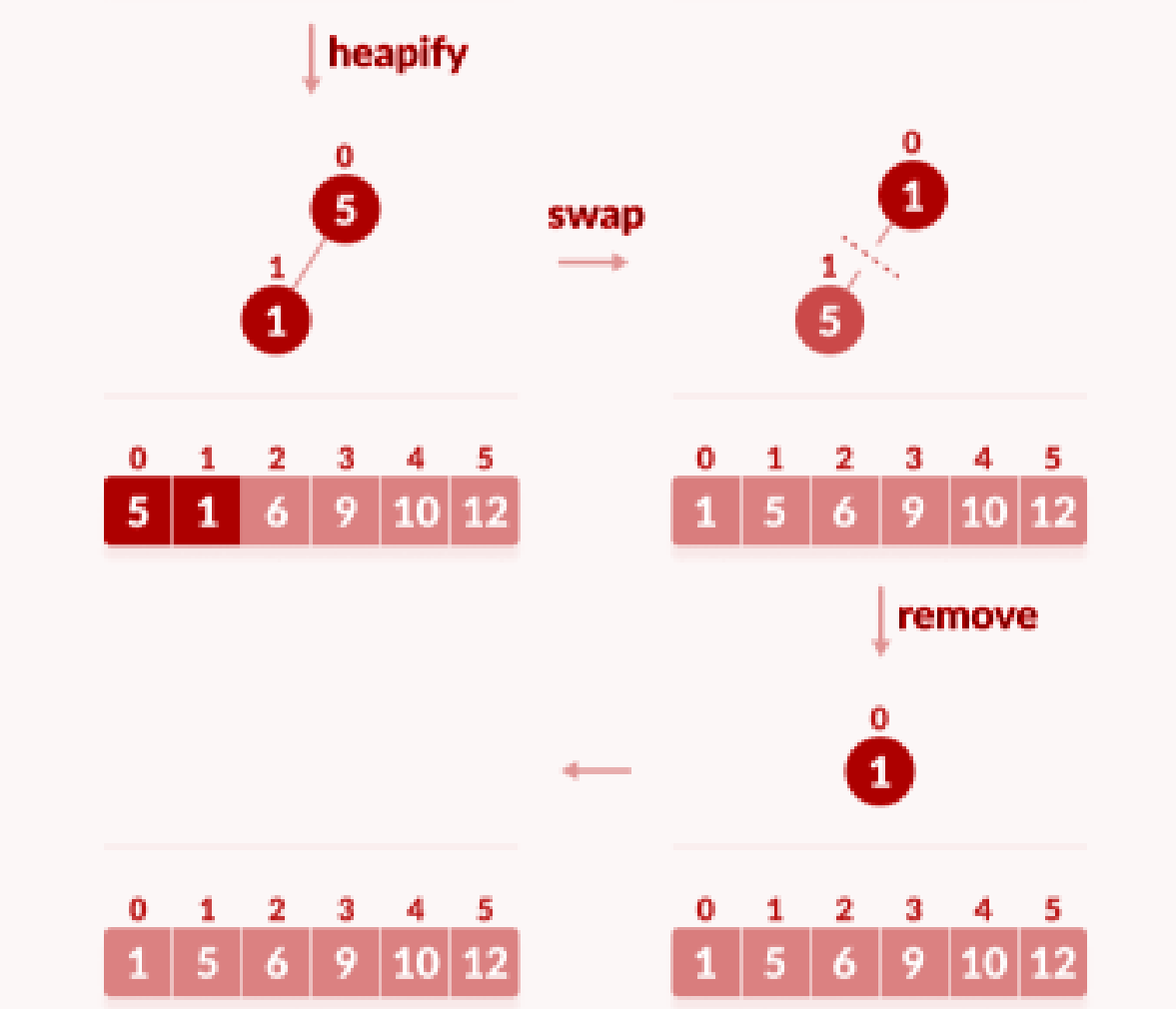
- The heap-sort algorithm inserts all elements (from an unsorted array) into a maxheap.
- Note that heap sort can be done **in-place** with the array to be sorted.
- Since the tree satisfies the Max-Heap property, then the largest item is stored at the root node.
- **Swap**: Remove the root element and put at the end of the array (nth position)
- Put the last item of the tree (heap) at the vacant place.
- **Remove**: Reduce the size of the heap by 1.
- **Heapify:** Heapify the root element again so that we have the highest element at root.
- The process is repeated until all the items in the list are sorted.

Consider the given illustrated example:

-> Applying heapsort to the unsorted array **[12, 6, 10, 5, 1, 9]**


swap
remove
heapify
swap
remove


heapify
swap
remove
heapify
swap
remove



Go through the given **Python Code** for better understanding:

```
def heapSort(arr):
    n = len(arr)
    # Build a maxheap. last parent will be at ((n//2)-1)
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # One by one extract the max elements
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]   # swap
        heapify(arr, i, 0)
```

In-built Min-Heap in Python

A heap is created by using python's inbuilt library named **heapq**. This library has the relevant functions to carry out various operations on a **min-heap** data structure. Below is a list of these functions.

- **heapify** - This function converts a regular list to a heap. In the resulting heap, the smallest element gets pushed to index position 0. But the rest of the data elements are not necessarily sorted.
- **heappush** – This function adds an element to the heap without altering the current heap.
- **heappop** - This function returns the smallest data element from the heap.
- **heapreplace** – This function replaces the smallest data element with a new value supplied in the function.

Creating a Min-Heap

A heap is created by simply using a list of elements with the **heapify** function. In the below example we supply a list of elements and the heapify function rearranges the elements bringing the smallest element to the first position.

```
import heapq
H = [21,1,45,78,3,5]
# Use heapify to rearrange the elements
heapq.heapify(H)
print(H)
```

When the above code is executed, it produces the following result −

```
[1, 3, 5, 78, 21, 45]
```

Inserting into heap

Inserting a data element to a heap always adds the element at the last index. But you can apply the heapify function again to bring the newly added element to the

first index only if it is the smallest in value. In the below example we insert the number 8.

```
import heapq
H = [21,1,45,78,3,5]
# Convert to a heap
heapq.heapify(H)
print(H)
# Add element
heapq.heappush(H,8)
print(H)
```

When the above code is executed, it produces the following result −

```
[1, 3, 5, 78, 21, 45]
[1, 3, 5, 78, 21, 45, 8]
```

Removing from heap

You can remove the element at the first index by using this function. In the below example the function will always remove the element at the index position 1.

```
import heapq
H = [21,1,45,78,3,5]
# Create the heap
heapq.heapify(H)
print(H)
# Remove element from the heap
heapq.heappop(H)

print(H)
```

When the above code is executed, it produces the following result −

```
[1, 3, 5, 78, 21, 45]
[3, 21, 5, 78, 45]
```

Replacing in a Heap

The **heapreplace** function always removes the smallest element of the heap and inserts the new incoming element at some place not fixed by any order.

```
import heapq
H = [21,1,45,78,3,5]
# Create the heap
heapq.heapify(H)
print(H)
# Replace an element
heapq.heapreplace(H,6)
print(H)
```

```
[1, 3, 5, 78, 21, 45]
[3, 6, 5, 78, 21, 45]
```

In-built Max-Heap in Python

To implement a max-heap, the **heapq** library has the following functions:

- **_heapify_max** - This function converts a regular list to a max-heap.
- **_heappop_max** - This function returns the largest data element from the heap.
- **_heapreplace_max** – This function replaces the maximum data element with a new value supplied in the function.
- **_siftdown_max-** Pushes a new element, but compares with all its parents, and pushes all the parents down until it finds a place where the new item fits.

K-Smallest Elements in a List

This is a good example of problem-solving via a heap data structure. The basic idea here is to create a min-heap of all n elements and then extract the minimum element K times (We know that the root element in a min-heap is the smallest element).

Approach

- Build a min-heap of size **n** of all elements.
- Extract the minimum elements **K** times, i.e. delete the root and perform heapify operation **K** times.
- Store all these K smallest elements.

Note: The code written using these insights can be found in the solution tab of the problem itself.

Huffman Coding

Introduction

Huffman Coding is one approach followed for **Text Compression.** Text compression means reducing the space requirement for saving a particular text.

Huffman Coding is a lossless data compression algorithm, ie. it is a way of compressing data without the data losing any information in the process. It is useful in cases where there is a series of frequently occurring characters.

Working of Huffman Algorithm:

Suppose, the given string is:

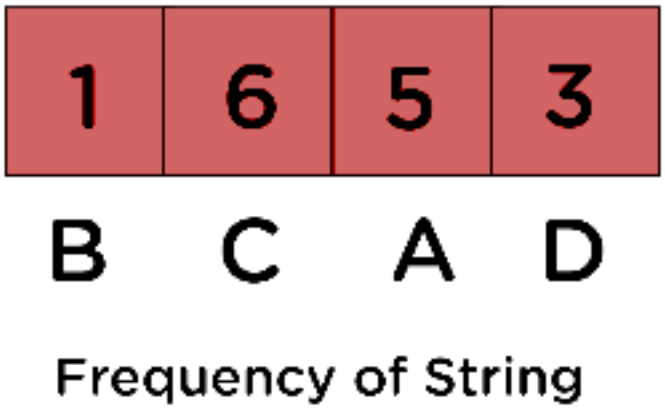
Intial String

Here, each of the characters of the string takes 8 bits of memory. Since there are a total of 15 characters in the string so the total memory consumption will be 15*8 = 120 bits. Let's try to compress its size using the Huffman Algorithm.

First-of-all, Huffman Coding creates a tree by calculating the frequencies of each character of the string and then assigns them some unique code so that we can retrieve the data back using these codes.

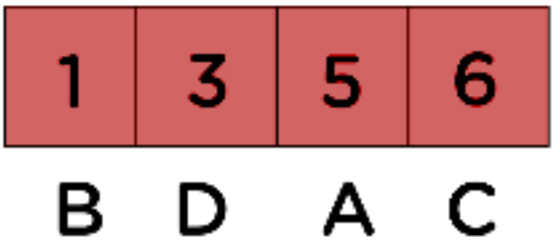
Follow the steps below:

1. Begin with calculating the frequency of each character value in the given string.



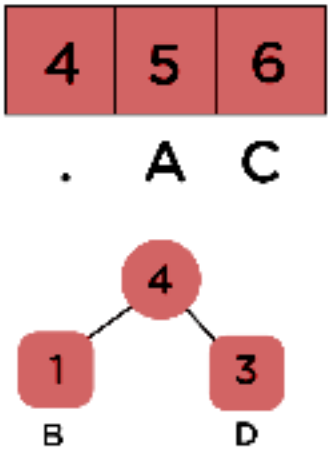
Frequency of String

2. Sort the characters in ascending order concerning their frequency and store them in a priority queue, say **Q.**
3. Each character should be considered as a different leaf node.



Characters sorted according to the frequency

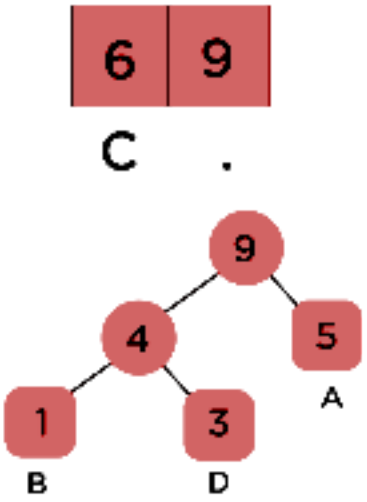
4. Make an empty node, say **z**. The left child of z is marked as the minimum frequency and the right child, the second minimum frequency. The value of z is calculated by summing up the first two frequencies.



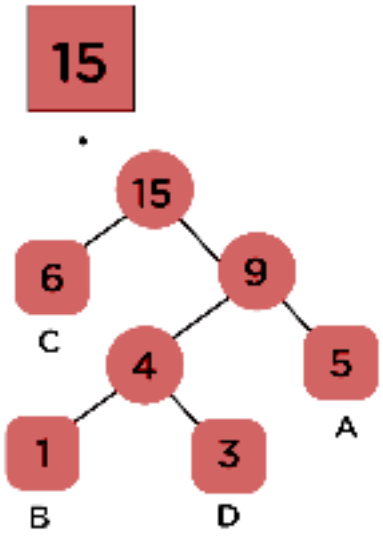
Getting the sum of the least numbers

Here, "." denote the internal nodes.

5. Now, remove the two characters with the lowest frequencies from the priority queue **Q** and append their sum to the same.
6. Simply insert the above node **z** to the tree.
7. For every character in the string, repeat steps 3 to 5.

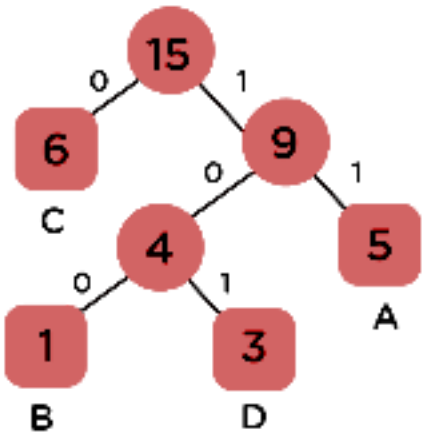


Repeat steps 3 to 5 for all the characters.



Repeat steps 3 to 5 for all the characters.

8. Assign 0 to the left side and 1 to the right side except for the leaf nodes.



Assign 0 to the left edge and 1 to the right edge

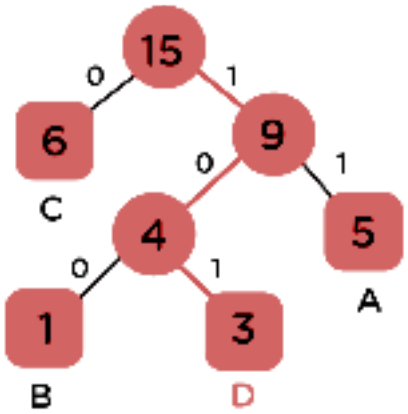
The size table is given below:

Character	Frequency	Code	Size
A	5	11	5*2 = 10
B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
4*8 = 32 bits	15 bits		28 bits

Size before encoding: 120 bits

Size after encoding: 32 + 15 + 28 = 75 bits

To decode the code, simply traverse through the tree (starting from the root) to find the character. Suppose we want to decode 101, then:



Decoding

Time complexity:

In the case of encoding, inserting each character into the priority queue takes **O(log n**) time. Therefore, for the complete array, the time complexity becomes **O(nlog(n))**.

Similarly, extraction of the element from the priority queue takes **O(log n)** time. Hence, for the complete array, the achieved time complexity is **O(nlog n)**.

Python Code:

Go through the given Python code, for deeper understanding:

```
# Huffman Coding in python
string = 'BCAADDDCCACACAC' #String similar to the above-taken example


# Creating tree nodes
class NodeTree(object):

    def __init__(self, left=None, right=None):
        self.left = left
        self.right = right

    def children(self): #Return children of a node
        return (self.left, self.right)

    def nodes(self):
        return (self.left, self.right)

    def __str__(self):
        return '%s_%s' % (self.left, self.right)


# Main function implementing huffman coding
def huffman_code_tree(node, left=True, binString=''):
    if type(node) is str:
        return {node: binString}
    (l, r) = node.children()
    d = dict()
    d.update(huffman_code_tree(l, True, binString + '0'))
    d.update(huffman_code_tree(r, False, binString + '1'))
    return d
```

```
# Calculating frequency
freq = {}
for c in string:
    if c in freq:
        freq[c] += 1
    else:
        freq[c] = 1

freq = sorted(freq.items(), key=lambda x: x[1], reverse=True)
nodes = freq

while len(nodes) > 1:
    (key1, c1) = nodes[-1]
    (key2, c2) = nodes[-2]
    nodes = nodes[:-2]
    node = NodeTree(key1, key2)
    nodes.append((node, c1 + c2))

    nodes = sorted(nodes, key=lambda x: x[1], reverse=True)

huffmanCode = huffman_code_tree(nodes[0][0])

print(' Char | Huffman code ')
print('----------------------')
for (char, frequency) in freq:
    print(' %-4r |%12s' % (char, huffmanCode[char]))
```

Applications of Huffman Coding:

- They are used for transmitting fax and text.
- They are used by conventional compression formats like PKZIP, GZIP, etc.

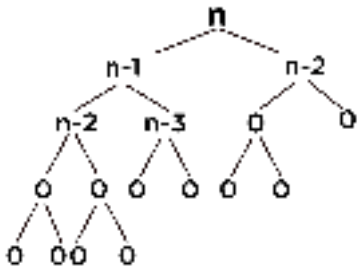
Dynamic Programming- 1

Introduction

Suppose we need to find the n^{th} Fibonacci number using recursion that we have already found out in our previous sections. Let's directly look at its code:

```
def fibo(n):
      if(n <= 1):
            return n
      return fibo(n-1) + fibo(n-2)
```

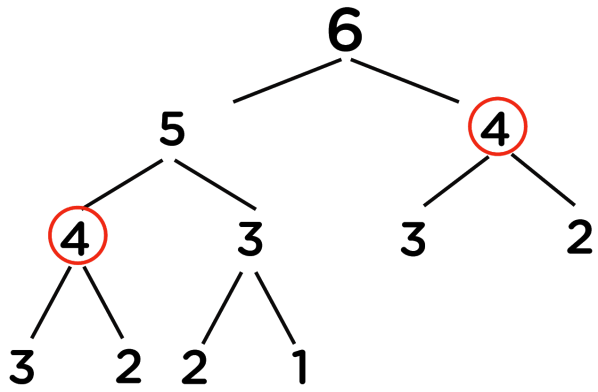
- Here, for every n, we need to make a recursive call to **f(n-1)** and **f(n-2)**.
- For **f(n-1)**, we will again make the recursive call to **f(n-2)** and **f(n-3).**
- Similarly, for **f(n-2)**, recursive calls are made on **f(n-3)** and **f(n-4)** until we reach the base case.
- The recursive call diagram will look something like shown below:



- At every recursive call, we are doing constant work(k)(addition of previous outputs to obtain the current one).
- At every level, we are doing 2^nK work (where n = 0, 1, 2, ...).
- Since reaching 1 from n will take n calls, therefore, at the last level, we are doing $2^{n-1}k$ work.
- Total work can be calculated as:

$$(2^0 + 2^1 + 2^2 + ... + 2^{n-1}) * k \simeq 2^nk$$

- Hence, it means time complexity will be **O(2^n)**.
- We need to improve this complexity. Let's look at the example below for finding the 6^{th} Fibonacci number.



Important Observation:

- We can observe that there are repeating recursive calls made over the entire program.
- As in the above figure, for calculating f(5), we need the value of f(4) (first recursive call over f(4)), and for calculating f(6), we again need the value of f(4)(second similar recursive call over f(4)).
- Both of these recursive calls are shown above in the outlining circle.
- Similarly, there are many other values for which we are repeating the recursive calls.
- Generally, while recursing, there are repeated recursion calls, which increases the time complexity of the program.

To overcome this problem, we will store the output of previously encountered values(preferably in arrays as these are most efficient to traverse and extract data). Next time whenever we will be making the recursive calls over these values, we will directly consider their already stored outputs and then use these in our calculations instead of calculating them over again.

This way, we can improve the running time of our code. This process of storing each recursive call's output and then using them for further calculations preventing the code from calculating these again is called **Memoization**.

- To achieve this in our example we will simply take an answer array, initialized to -1.
- Now while making a recursive call, we will first check if the value stored in this answer array corresponding to that position is -1 or not.
- If it is -1, it means we haven't calculated the value yet and need to proceed further by making recursive calls for the respective value.
- After obtaining the output, we need to store this in the answer array so that next time, if the same value is encountered, it can be directly used from this answer array.

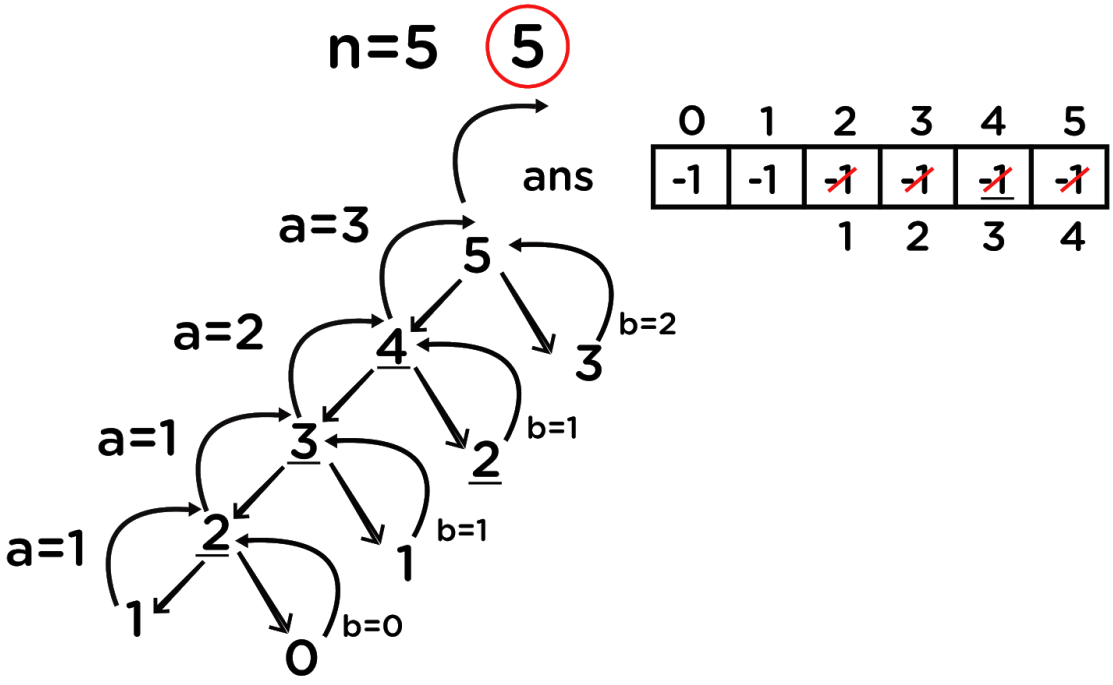
Now in this process of memoization, considering the above Fibonacci numbers example, it can be observed that the total number of unique calls will be at most **(n+1)** only.

Let's look at the memoization code for Fibonacci numbers below:

```
def fibb(n):
      if n==0 or n==1:#Base
            return n

      if dp[n-1] == -1: #checking if has been already calculated
            ans1 = fibb(n-1,dp) # If being calculated first time
            dp[n-1] = ans1
      else:
            ans1 = dp[n-1]

      if dp[n-2] == -1: #checking if has been already calculated
            ans2 = fibb(n-2,dp)
            dp[n-2] = ans2
      else:
            ans2 = dp[n-2]
```

```
        myAns = ans1 + ans2 #final answer
        return myAns

n = int(input())
dp = [-1 for in range(n+1)]
ans = fibb(n,dp)
print(ans)
Le
```

Let's dry run for n = 5, to get a better understanding:



Again, if we observe carefully, we can see that for any number, we are not able to make a recursive call on the right side of it. This means that we can make at most 5+1 = 6 (n+1) unique recursive calls which reduce the time complexity to O(n) which is highly optimized as compared to simple recursion.

Summary

- Memoization is a **top-down approach,** where we save the previous answers so that they can be used to calculate future answers and improve the time complexity to a greater extent.
- Finally, what we are doing is making a recursive call to every index of the answer array and calculating the value for it using previous outputs stored.
- Recursive calls terminate over the base case, which means we are already aware of the answers that should be stored in the base case's indexes.
- In cases of Fibonacci numbers, these indexes are 0 and 1 as f(0) = 0 and f(1) = 1. So we can directly allot these two values to our answer array and then use these to calculate f(2), which is f(1) + f(0), and so on for every other index.
- This can be simply done iteratively by running a loop from i = (2 to n).
- Finally, we will get our answer at the 5^{th} index of the answer array as we already know that the i-th index contains the answer to the i-th value.

We are first trying to figure out the dependency of the current value on the previous values and then using them calculating our new value. Now, we are looking for those values which do not depend on other values, which means they are independent (the base case's values as these are the smallest problems about which we are already aware of). Finally, we will follow a **bottom-up approach** to reach the desired index. This approach of converting recursion into iteration is known as **Dynamic programming(DP).**

Let us now look at the DP code for calculating the n^{th} Fibonacci number:

```
def fibonacci(n):
    f = [0, 1]
    for i in range(2, n+1):
        f.append(f[i-1] + f[i-2]) #Bottom up approach
    return f[n]
```

Note: Generally, memoization is a recursive approach, and DP is an iterative approach.

For all further problems, we will do the following:

1. Figure out the most straightforward approach for solving a problem using recursion.
2. Now, try to optimize the recursive approach by storing the previous answers using memoization.
3. Finally, replace recursion by iteration using dynamic programming. (It is preferred to be done in this manner because recursion generally has an increased space complexity as compared to iteration methods.)

Problem Statement: Min steps to 1

Given a positive integer n, find the minimum number of steps s, that takes n to 1. You can perform any one of the following three steps:

1. Subtract 1 from it. (n = n-1).
2. If its divisible by 2, divide by 2. (if n%2 == 0, then n= n/2).
3. If its divisible by 3, divide by 3. (if n%3 == 0, then n = n / 3).

Example 1: For n = 4:

STEP-1: n = 4/2 = 2

STEP-2: n = 2/2 = 1

Hence, the answer is **2.**

Example 2: For n = 7:

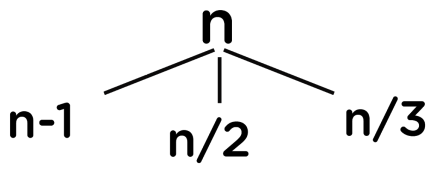
STEP-1: n = 7 - 1 = 6

STEP-2: n = 6/3 = 2

STEP-3: n = 2/2 = 1

Hence, the answer is **3**.

Approach: We are only allowed to perform the above three mentioned ways to reduce any number to 1.



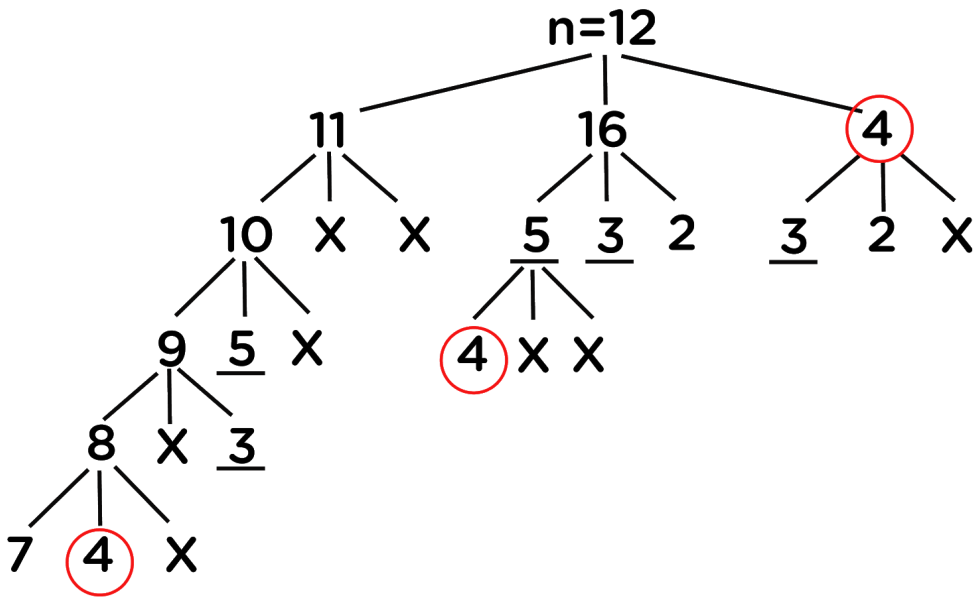
Let's start thinking about the brute-force approach first, i.e., recursion.

We will make a recursive call to each of the three steps keeping in mind that for dividing by 2, the number should be divisible by 2 and similarly for 3 as given in the question statement. After that take the minimum value out of the three obtained and simply add 1 to the answer for the current step itself. Thinking about the base case, we can see that on reaching 1, simply we have to return 0 as it is our destination value. Let's now look at the code:

```
def getMinSteps(n):
    # base case
    if (n <= 1):
        return 0

    x = getMinSteps(n-1) #Recursive call 1
    y = MAX_VALUE #Initialise to infinity to check divisibility
    Z = MAX_VALUE
    if (n%2 == 0):
        y = getMinSteps(n//2) #Recursive call 2
    if (n%3 == 0):
        z = getMinSteps(n//3) #Recursive call 3

    return min(x, min(y,z))+1
```

Now, we have to check if we can optimize the code that we have written. It can be done using memoization. But, for memoization to apply, we need to check if there are any overlapping sub-problems so that we can store the previous values to obtain the new ones. To check this let's dry run the problem for n = 12:

(Here X represents that the calls are not feasible as the number is not divisible by either of 2 or 3)



Here, if we blindly make three recursive calls at each step, then the time complexity will approximately be **O(3^n)**.

From the above, it is visible that there are repeating sub-problems. Hence, this problem can be optimized using memoization.

Now, we need to figure out the number of unique calls, i.e., how many answers we are required to save. It is clear that we need at most n+1 responses to be saved, starting from n = 0, and the final answer will be present at index n.

The code will be nearly the same as the recursive approach; just we will not be making recursive calls for already stored outputs. Follow the code and comments below:

```
def getMinStepsHelper(n, memo):
      # base case
    if (n == 1):
        return 0
    if (memo[n] != -1):
        return memo[n]

    res = getMinSteps(n-1, memo)

    if (n%2 == 0):
        res = min(res, getMinSteps(n//2, memo))
    if (n%3 == 0):
        res = min(res, getMinSteps(n//3, memo))

    # store memo[n] and return
    memo[n] = 1 + res
    return memo[n]

def getMinSteps(n):

    memo = [0 for i in range(n+1)]

    # initialize memoized array
    for i in range(n+1):
        memo[i] = -1

    return getMinStepsHelper(n, memo)
```

Time complexity has been reduced significantly to **O(n)** as there are only **(n+1)** unique iterations. Now, try to code the DP approach by yourself, and for the code, refer to the solution tab.

Problem Statement: Minimum Number of Squares

Given an integer N, find and return the count of minimum numbers, the sum of whose squares is equal to N.

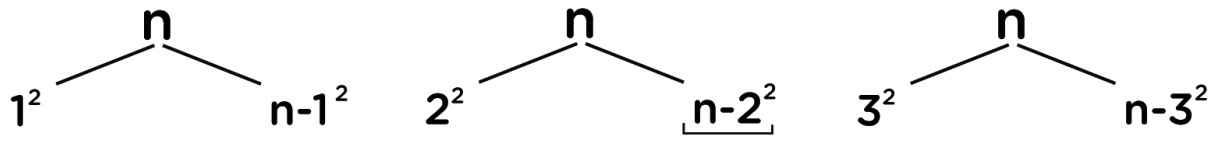
That is, if N is 4, then we can represent it as : {1^2 + 1^2 + 1^2 + 1^2} and {2^2}. The output will be 1, as 1 is the minimum count of numbers required. (x^y represents x raised to the power y.)

Example: For n = 12, we have the following ways:

- 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1
- 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 1^1 + 2^2
- 1^1 + 1^1 + 1^1 + 1^1 + 2^2 + 2^2
- 2^2 + 2^2 + 2^2

Hence, the minimum count is obtained from the 4-th option. Therefore, the answer is equal to 3.

Approach: First-of-all, we need to think about breaking the problems into two parts, one of which will be handled by recursion and the other one will be handled by us(smaller sub-problem). We can break the problem as follows:



And so on...

- In the above figure, it is clear that in the left subtree we are making ourselves try over a variety of values that can be included as a part of our solution.
- The right subtree's calculation will be done by recursion.

- Hence, we will just handle the $\mathbf{i^2}$ part, and $\mathbf{(n-i^2)}$ will be handled by recursion.
- By now, we have got ourselves an idea of solving this problem, the only thinking left is the loop's range on which we will be iterating, i.e., the values of **i** for which we will be deciding to consider while solving or not.
- As the maximum value up to which i can be pushed, to reach **n** is **√n** as **(√n*√n = n).** Hence, we will be iterating over the range **(1 to √n)** and do consider each possible way by sending $\mathbf{(n-i^2)}$ over the recursion.
- This way we will get different subsequences and as per the question, we will simply return the minimum out of it.

This problem is left for you to try out using all the three approaches and for code, refer to the solution tab of the same.

Dynamic Programming- 2

Let us now move to some advanced-level DP questions, which deal with 2D arrays.

Problem Statement: Min Cost Path

Given an integer matrix of size **m*n**, you need to find out the value of minimum cost to reach from the cell **(0, 0) to (m-1, n-1)**. From a cell **(i, j)**, you can move in three directions : **(i+1, j), (i, j+1) and (i+1, j+1).** The cost of a path is defined as the sum of values of each cell through which the path passes.

For example, The given input is as follows-

```
3 4
3 4 1 2
2 1 8 9
4 7 8 1
```

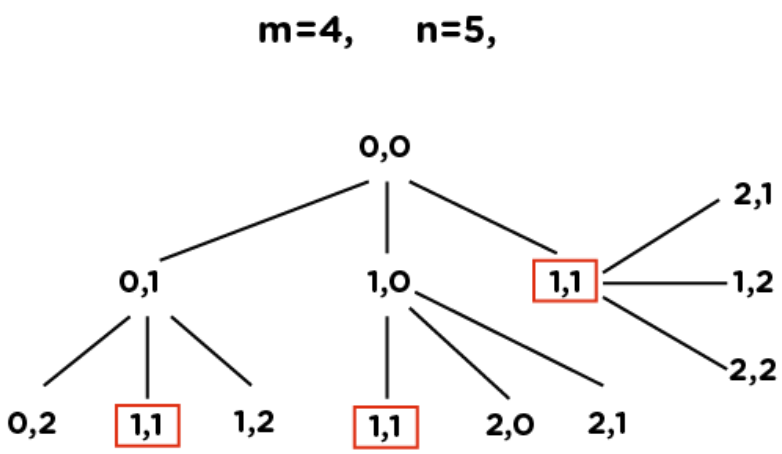
The path that should be followed is **3 -> 1 -> 8 -> 1**. Hence the output is **13**.

Approach:

- Thinking about the **recursive approach** to reach from the cell **(0, 0)** to **(m-1, n-1)**, we need to decide for every cell about the direction to proceed out of three.
- We will simply call recursion over all the three choices available to us, and finally, we will be considering the one with minimum cost and add the current cell's value to it.
- Let's now look at the recursive code for this problem:

```
import sys

# Returns cost of minimum cost path from (0,0) to (m, n) in mat[R][C]
def minCost(cost, m, n):
    if (n < 0 or m < 0):
        return sys.maxsize
    elif (m == 0 and n == 0):
        return cost[m][n]
    else:
        return cost[m][n] + min( minCost(cost, m-1, n-1),
                                 minCost(cost, m-1, n),
                                 minCost(cost, m, n-1) )

#A utility function that returns minimum of 3 integers */
def min(x, y, z):
    if (x < y):
        return x if (x < z) else z
    else:
        return y if (y < z) else z
```

Let's dry run the approach to see the code flow. Suppose, **m = 4 and n = 5**; then the recursive call flow looks something like below:



Here, we can see that there are many repeated/overlapping recursive calls(for example: **(1,1)** is one of them), leading to exponential time complexity, i.e., **O(3^n)**. If we store the output for each recursive call after their first occurrence, we can easily avoid the repetition. It means that we can improve this using memoization.

Now, let's move on to the **Memoization approach**.
In memoization, we avoid repeated overlapping calls by storing the output of each recursive call in an array. In this case, we will be using a 2D array instead of 1D, as we already discussed in our previous lectures that the storage used for the memoization is generally the same as the one that recursive calls use to their maximum.
Refer to the memoization code (along with the comments) below for better understanding:

```
import sys
def minCost(cost,i,j,n,m,dp):

      # Special Case
      if i == n-1 and j==m-1:
            return cost[i][j]

      # Base Case
      if i>=n or j>=m:
            return sys.maxsize

      if dp[i][j+1] == sys.maxsize:
            ans1 = minCost(cost,i,j+1,n,m,dp)
            dp[i][j+1] = ans1
      else:
            ans1 = dp[i][j+1]

      if dp[i+1][j] == sys.maxsize:
            ans2 = minCost(cost,i+1,j,n,m,dp)
            dp[i+1][j] = ans2
```

```
        else:
                ans2 = dp[i+1][j]

        if dp[i+1][j+1] == sys.maxsize:
                ans3 = minCost(cost,i+1,j+1,n,m,dp)
                dp[i+1][j+1] = ans3
        else:
                ans3 = dp[i+1][j+1]

        ans= cost[i][j]+min(ans1, ans2, ans3)
        return ans

cost = [[1,5,11],[8,13,12],[2,3,7],[15,16,18]]
n=4
m=3
dp= [[sys.maxsize for j in range(m+1) for i in range(n+1)]
ans = minCost(cost, 0,0,4,3,dp)
print(ans)
```

Here, we can observe that as we move from the cell **(0,0) to (m-1, n-1)**, in general, the i-th row varies from 0 to m-1, and the j-th column runs from 0 to n-1. Hence, the unique recursive calls will be a maximum of **(m-1) * (n-1)**, which leads to the time complexity of **O(m*n)**.

To get rid of the recursion, we will now proceed towards the **DP approach**.

The DP approach is simple. We just need to create a solution array (lets name that as **ans**), where:

ans[i][j] = minimum cost to reach from (i, j) to (m-1, n-1)

Now, initialize the last row and last column of the matrix with the sum of their values and the value, just after it. This is because, in the last row or column, we can reach there from their forward cell only (You can manually check it), except the cell **(m-1, n-1)**, which is the value itself.

```
ans[m-1][n-1] = cost[m-1][n-1]
ans[m-1][j] = ans[m-1][j+1] + cost[m-1][j]  (for 0 < j < n)
ans[i][n-1] = ans[i+1][n-1] + cost[i][m-1] (for 0 < i < m)
```

Next, we will simply fill the rest of our answer matrix by checking out the minimum among values from where we could reach them. For this, we will use the same formula as used in the recursive approach:

```
ans[i][j] = min(ans[i+1][j], ans[i+1][j+1], ans[i][j+1]) + cost[i][j]
```

Finally, we will get our answer at the cell (0, 0), which we will return.
The code looks as follows:

```
R = 3
C = 3

def minCost(cost, m, n):

    ans = [[0 for x in range(C)] for x in range(R)]

    ans[0][0] = cost[0][0]

    # Initialize first column of total cost(tc) array
    for i in range(1, m+1):
        ans[i][0] = ans[i-1][0] + cost[i][0]

    # Initialize first row of tc array
    for j in range(1, n+1):
        ans[0][j] = ans[0][j-1] + cost[0][j]

    # Construct rest of the tc array
    for i in range(1, m+1):
        for j in range(1, n+1):
            min_temp = min(ans[i-1][j-1],ans[i-1][j],ans[i][j-1])
            ans[i][j] = min_temp + cost[i][j]

    return ans[m][n]
```

Note: This is the bottom-up approach to solve the question using DP.

Problem Statement: LCS (Longest Common Subsequence)

The longest common subsequence (LCS) is defined as the longest subsequence that is common to all the given sequences, provided that the elements of the subsequence are not required to occupy consecutive positions within the original sequences.

Note: Subsequence is a part of the string which can be made by omitting none or some of the characters from that string while maintaining the order of the characters.

If s1 and s2 are two given strings then z is the common subsequence of s1 and s2, if z is a subsequence of both of them.

Example 1:

```
s1 = "abcdef"
s2 = "xyczef"
```

Here, the longest common subsequence is `"cef"`; hence the answer is 3 (the length of LCS).

Example 2:

```
s1 = "ahkolp"
s2 = "ehyozp"
```

Here, the longest common subsequence is `"hop"`; hence the answer is 3.

Approach: Let's first think of a brute-force approach using **recursion**. For LCS, we have to match the starting characters of both strings. If they match, then simply we can break the problem as shown below:

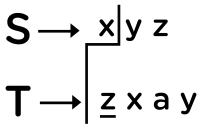
```
s1 = "x|yzar"
s2 = "x|qwea"
```

The rest of the LCS will be handled by recursion. But, if the first characters do not match, then we have to figure out that by traversing which of the following strings, we will get our answer. This can't be directly predicted by just looking at them, so we will be traversing over both of them one-by-one and check for the maximum value of LCS obtained among them to be considered for our answer.

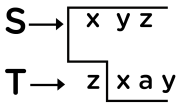
For example:

Suppose, string `s = "xyz"` and string `t = "zxay"`.

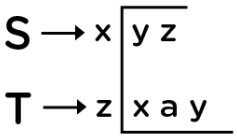
We can see that their first characters do not match so that we can call recursion over it in either of the following ways:



A =



B =



C=

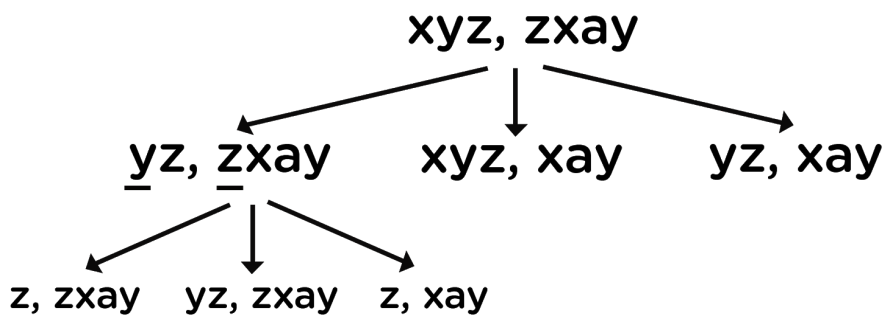
Finally, our answer will be:

```
LCS = max(A, B, C)
```

Check the code below and follow the comments for a better understanding.

```
def lcs(s, t, m, n):

    if m == 0 or n == 0: #Base Case
       return 0;
    elif s[m-1] == t[n-1]:
       return 1 + lcs(s, t, m-1, n-1);
    else:
       return max(lcs(s, t, m, n-1), lcs(s, t, m-1, n));
```

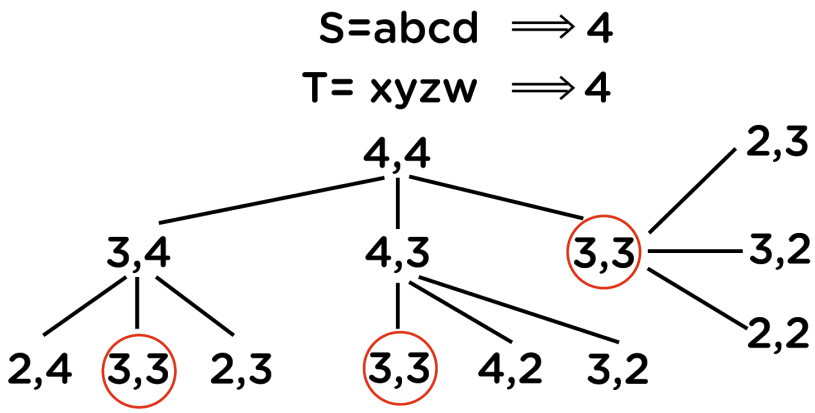
If we dry run this over the example: `s = "xyz"` and `t = "zxay"`, it will look something like below:



Here, as for each node, we will be making three recursive calls, so the time complexity will be exponential and is represented as **$O(2^{m+n})$**, where m and n are the lengths of both strings. This is because, if we carefully observe the above code, then we can skip the third recursive call as it will be covered by the two others.

Now, thinking over improving this time complexity...

Consider the diagram below, where we are representing the dry run in terms of its length taken at each recursive call:



As we can see there are multiple overlapping recursive calls, the solution can be optimized using **memoization** followed by DP. So, beginning with the memoization approach, as we want to match all the subsequences of the given two strings, we have to figure out the number of unique recursive calls. For string s, we can make at most **length(s)** recursive calls, and similarly, for string t, we can make at most **length(t)** recursive calls, which are also dependent on each other's solution. Hence, our result can be directly stored in the form of a 2-dimensional array of size **(length(s)+1) * (length(t) + 1)** as for string s, we have **0 to length(s)** possible combinations, and the same goes for string t.

So for every index 'i' in string *s* and 'j' in string *t*, we will choose one of the following two options:

1. If the character **s[i]** matches **t[j]**, the length of the common subsequence would be one plus the length of the common subsequence till the **i-1** and **j-1** indexes in the two respective strings.
2. If the character **s[i]** does not match **t[j]**, we will take the longest subsequence by either skipping **i-th or j-th character** from the respective strings.

Hence, the answer stored in the matrix will be the LCS of both strings when the length of string s will be 'i' and the length of string t will be 'j'.

Hence, we will get the final answer at the position `matrix[length(s)][length(t)]`.
Moving to the code:

```
N = 0
M = 0

def lcs(s, t, i, j, memo):

    # one or both of the strings are fully traversed

    if i == N or j == M:
        return 0

    # if result for the current pair is already present in
    # the table

    if memo[i][j] != -1:
        return memo[i][j]

    # check if the current characters in both the strings are equal

    if s[i] == t[j]:

        # check for the next characters in both the strings

        memo[i][j] = lcs(s, t, i + 1, j + 1, memo) + 1
    else:

        memo[i][j] = max(lcs(s,t,i,j+1,memo), lcs(s,t,i+1,j,memo))

    return memo[i][j]
```

Now, converting this approach into the **DP** code:

```
def lcs(s , t):
    # find the length of the strings
    m = len(s)
    n = len(t)

    # declaring the array for storing the dp values
    L = [[None]*(n+1) for i in xrange(m+1)]

    for i in range(m+1):
        for j in range(n+1):
            if i == 0 or j == 0 :
                L[i][j] = 0
            elif s[i-1] == t[j-1]:
                L[i][j] = L[i-1][j-1]+1
            else:
                L[i][j] = max(L[i-1][j] , L[i][j-1])

    # L[m][n] contains the length of LCS of X[0..n-1] & Y[0..m-1]
    return L[m][n]
```

Time Complexity: We can see that the time complexity of the DP and memoization approach is reduced to **O(m*n)** where **m** and **n** are the lengths of the given strings.

Problem Statement: Knapsack

Given the weights and values of 'N' items, we are asked to put these items in a knapsack, which has a capacity 'C'. The goal is to get the maximum value from the items in the knapsack. Each item can only be selected once, as we don't have multiple quantities of any item.

For example:

Items: {Apple, Orange, Banana, Melon}
Weights: {2, 3, 1, 4}
Values: {4, 5, 3, 7}
Knapsack capacity: 5

Possible combinations that satisfy the given conditions are:

Apple + Orange (total weight 5) => 9 value
Apple + Banana (total weight 3) => 7 value
Orange + Banana (total weight 4) => 8 value
Banana + Melon (total weight 5) => 10 value

This shows that **Banana + Melon** is the best combination, as it gives us the maximum value, and the total weight does not exceed the capacity.

Approach: First-of-all, let's discuss the brute-force-approach, i.e., the **recursive approach**. There are two possible cases for every item, either to put that item into the knapsack or not. If we consider that item, then its value will be contributed towards the total value, otherwise not. To figure out the maximum value obtained by maintaining the capacity of the knapsack, we will call recursion over these two cases simultaneously, and then will consider the maximum value obtained out of the two.

If we consider a particular weight 'w' from the array of weights with value 'v' and the total capacity was 'C' with initial value 'Val', then the remaining capacity of the knapsack becomes 'C-w', and the value becomes 'Val + v'.

Let's look at the recursive code for the same:

```
def knapSack(W, wt, val, n):

    # Base Case: if the size of array is 0 or we are not able to add
    # any more weight to the knapsack
    if n == 0 or W == 0:
        return 0
    # If the particular weight's value extends the limit of
    # knapsack's remaining capacity, then we have to simply skip it
    if (wt[n-1] > W):
        return knapSack(W, wt, val, n-1)
    else:#Recursive Calls
        return max(val[n-1] + knapSack(W-wt[n-1], wt, val, n-1),
            knapSack(W, wt, val, n-1))
```

Now, the memoization and DP approach is left for you to solve. For the code, refer to the solution tab of the same. Also, figure out the time complexity for the same by running the code over some examples and by dry running it.

Practice problems:

The link provided below contains 26 problems based on Dynamic programming and numbered as A to Z, A being the easiest, and Z being the toughest.

https://atcoder.jp/contests/dp/tasks

Backtracking

Introduction

- A **backtracking** algorithm is a problem-solving algorithm that uses a brute force approach for finding the desired output.
- The Brute force approach tries out all the possible solutions and chooses the desired/best solutions.
- The term **backtracking** suggests that if the current solution is not suitable, then backtrack and try other solutions. Thus, recursion is used in this approach.
- This approach is used to solve problems that have multiple solutions.
- Backtracking is thus a form of recursion.
- We begin by choosing an option and backtrack from it, if we reach a state where we conclude that this specified option does not give the required solution.
- We repeat these steps by going across each available option until we get the desired solution.

There are three types of problems in backtracking:

- **Decision Problem:** In this, we search for a feasible solution
- **Optimization Problem:** In this, we search for the best solution
- **Enumeration Problem:** In this, we find all feasible solutions

Difference between Recursion and Backtracking

In recursion, the function calls itself until it reaches a base case. In backtracking, we use recursion to explore all the possibilities until we get the best result for the problem.

Below is an example of finding all possible order of arrangements of a given set of letters. When we choose a pair we apply backtracking to verify if that exact pair has already been created or not. If not already created, the pair is added to the answer list, else it is ignored.

```
def permute(list, s):
    if list == 1:
        return s
    else:
        return [y+x for y in permute(1, s) for x in permute(list - 1, s)]

print(permute(1, ["a","b","c"]))
print(permute(2, ["a","b","c"]))
```

When the above code is executed, it produces the following result −

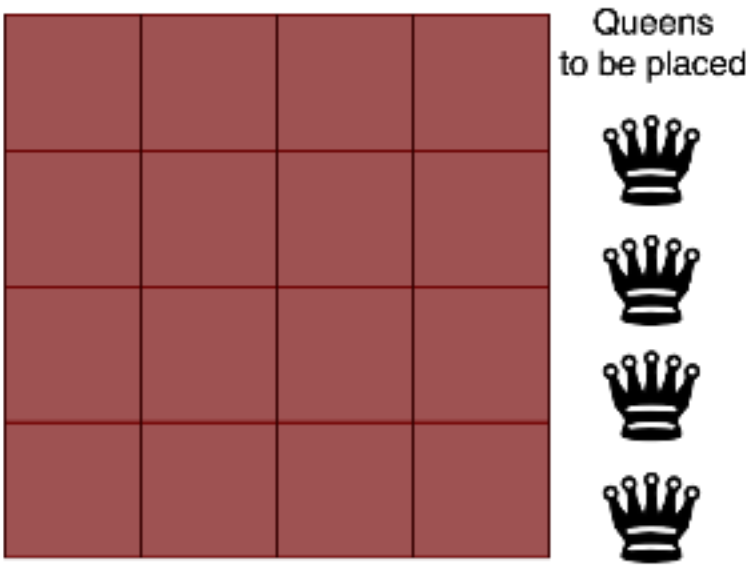
```
['a', 'b', 'c']
['aa', 'ab', 'ac', 'ba', 'bb', 'bc', 'ca', 'cb', 'cc']
```

Problem Statement: N-Queen

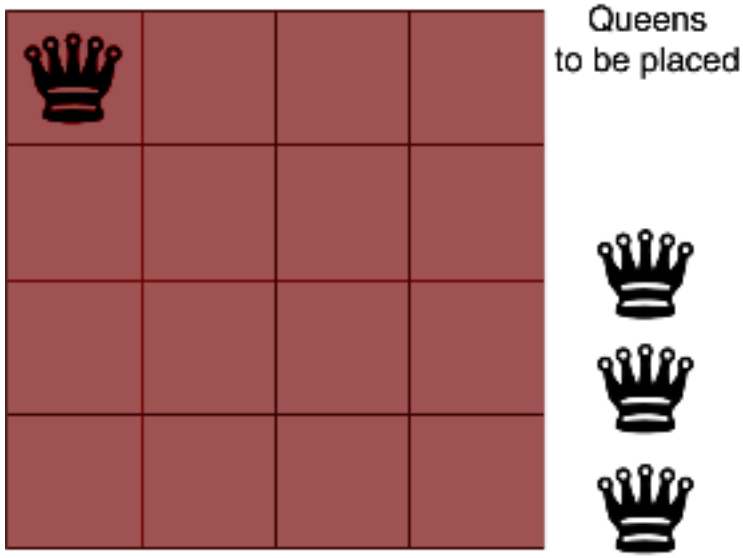
One of the most common examples of the backtracking is to arrange N queens on an NxN chessboard such that no queen can strike down any other queen. A queen can attack horizontally, vertically, or diagonally.

The solution to this problem is also attempted using Backtracking.

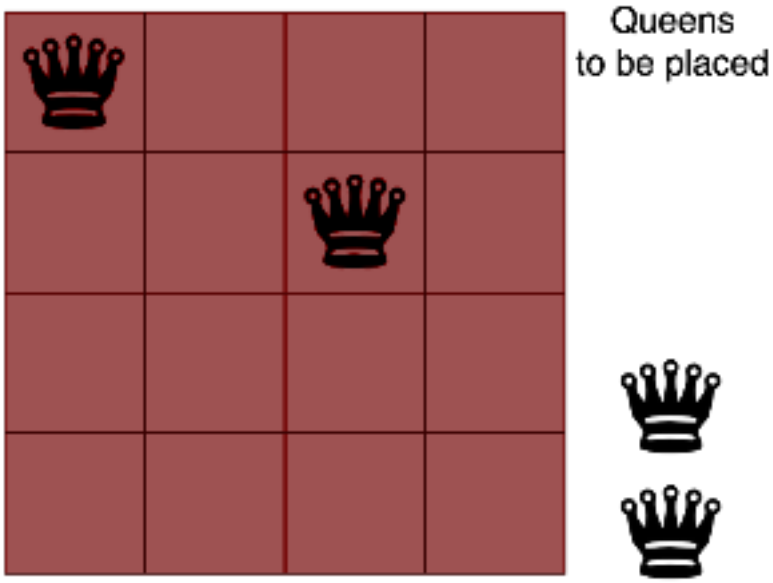
- We first place the first queen anywhere arbitrarily and then place the next queen in any of the safe places.
- We continue this process until the number of unplaced queens becomes zero (a solution is found) or no safe place is left.
- If no safe place is left, then we change the position of the previously placed queen.

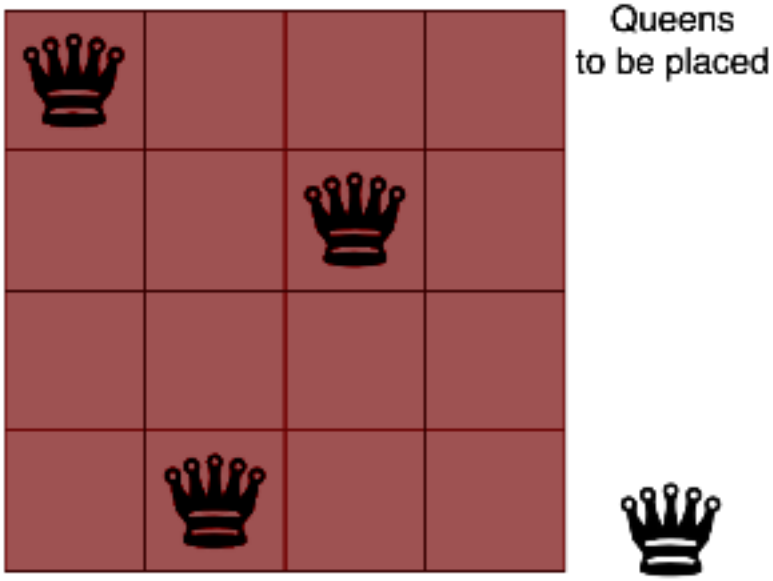


- The above picture shows an NxN chessboard and we have to place N queens on it. So, we will start by placing the first queen.

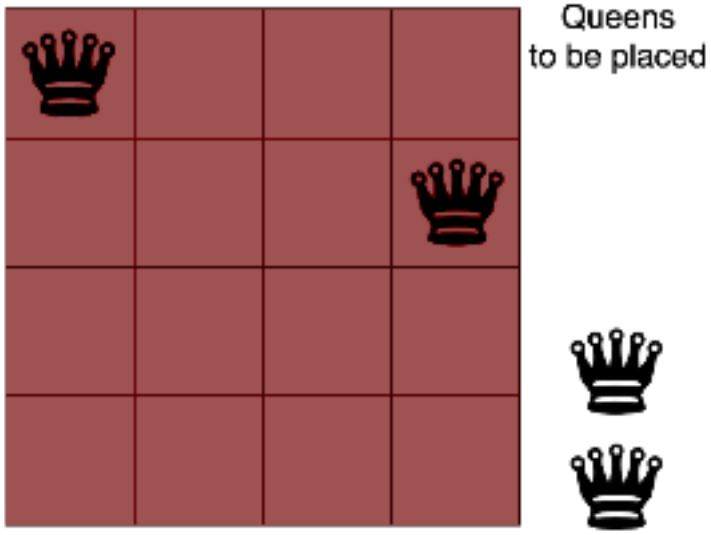


- Now, the second step is to place the second queen in a safe position and then the third queen.

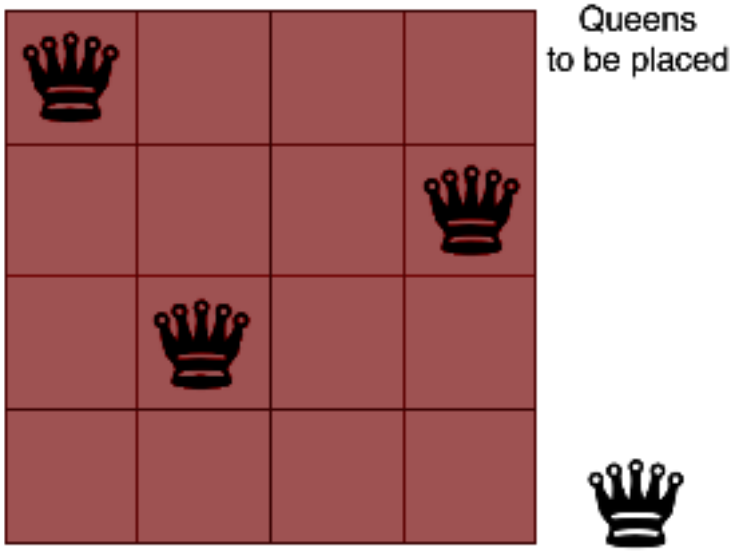




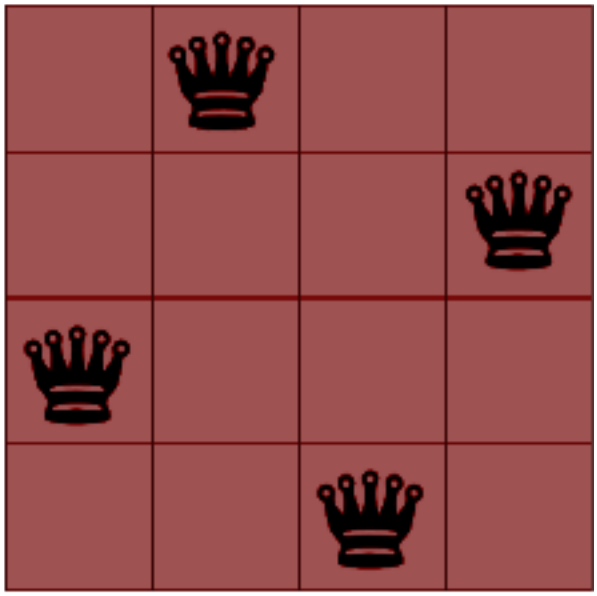
- Now, you can see that there is no safe place where we can put the last queen. So, we will just change the position of the previous queen. And this is backtracking.
- Also, there is no other position where we can place the third queen so we will go back one more step and change the position of the second queen.



- And now we will place the third queen again in a safe position until we find a solution.



- We will continue this process and finally, we will get the solution as shown below.



As now you have understood backtracking, let us now code the above problem of placing N queens on an NxN chessboard using the backtracking method.

```
#Number of queens
print ("Enter the number of queens")
N = int(input())

#NxN matrix with all elements 0
board = [[0]*N for _ in range(N)]

def check_possible(i, j):
    #checking if there is a queen in row or column
    for k in range(0,N):
        if board[i][k]==1 or board[k][j]==1:
            return True
    #checking diagonals
    for k in range(0,N):
        for l in range(0,N):
            if (k+l==i+j) or (k-l==i-j):
                if board[k][l]==1:
                    return True
    return False
```

```
def N_queen(n):
    #if n is 0, solution found
    if n==0:
        return True
    for i in range(0,N):
        for j in range(0,N):
            '''checking if we can place a queen here or not
            queen will not be placed if the place is being attacked
            or already occupied'''
            if (not(check_possible(i,j))) and (board[i][j]!=1):
                board[i][j] = 1
                #recursion
                #check if we can put a queen in this arrangement
                if N_queen(n-1)==True:
                    return True
                board[i][j] = 0

    return False

N_queen(N)
for i in board:
    print (i)
```

Explanation of the code

- `check_possible(int i,int j)` → This is a function to check if the cell (i,j) is under attack by any other queen or not. We are just checking if there is any other queen in the row 'i' or column 'j'. Then we are checking if there is any queen on the diagonal cells of the cell (i,j) or not. Any cell (k,l) will be diagonal to the cell (i,j) if k+l is equal to i+j or k-l is equal to i-j.
- `N_queen` → This is the function where we are implementing the backtracking algorithm.
- `if(n==0)` → If there is no queen left, it means all queens are placed and we have got a solution.

- `if((!check_possible(i,j)) && (board[i][j]!=1))` → We are just checking if the cell is available to place a queen or not. check_possible function will check if the cell is under attack by any other queen and `board[i][j]!=1` is making sure that the cell is vacant. If these conditions are met then we can put a queen in the cell – `board[i][j] = 1`.
- `if(N_queen(n-1)==1)` → Now, we are calling the function again to place the remaining queens and this is where we are doing backtracking. If this function (for placing the remaining queen) is not true, then we are just changing our current move – `board[i][j] = 0` and the loop will place the queen in some other position this time.

Another Example: Rat in A Maze

Go through the given blog to get a deeper understanding of the **Rat in A Maze** Problem:

https://www.codingninjas.com/blog/2020/09/02/backtracking-rat-in-a-maze/

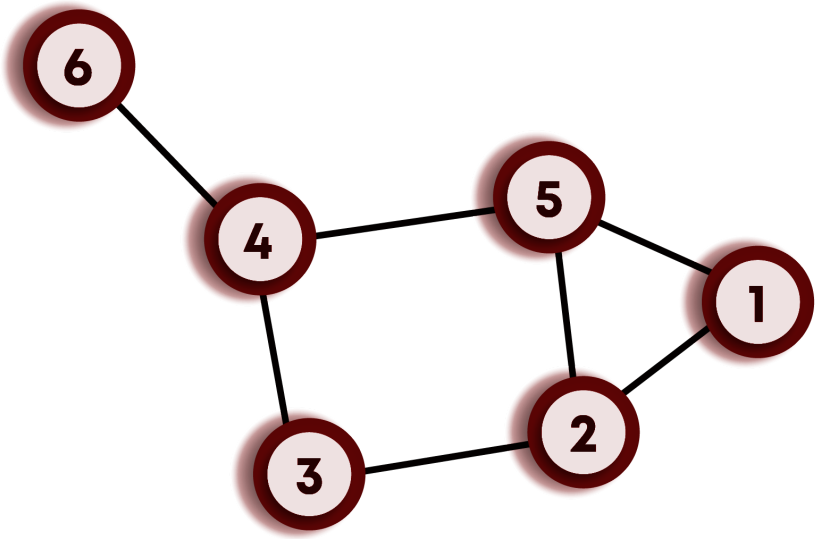
Graphs- 1

Introduction

A **graph** is a pair $G = (V, E)$, where V is a set whose elements are called ***vertices***, and E is a set of two sets of vertices, whose elements are called *edges*.

The vertices x and y of an edge **{*x*, *y*}** are called the ***endpoints*** of the edge. The edge is said to ***join*** x and y and to be ***incident*** on ***x*** and ***y***. A vertex may not belong to any edge.

For example: Suppose there is a road network in a country, where we have many cities, and roads are connecting these cities. There could be some cities that are not connected by some other cities like an island. This structure seems to be non-uniform, and hence, we can't use trees to store it. In such cases, we will be using graphs. Refer to the figure for better understanding.



Relationship between trees and graphs:

- A tree is a special type of graph in which we can reach any node to any other node using some path, unlike the graphs where this condition may or may not hold.
- A tree does not have any cycles in it.

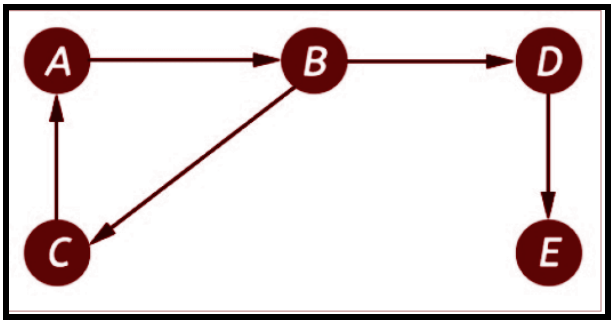
Graphs Terminology

- Nodes are named **vertices,** and the connections between them are called **edges**.
- Two vertices are said to be **adjacent** if there exists a direct edge connecting them.
- The **degree** of a node is defined as the number of edges that are incident to it.
- A **path** is a collection of edges through which we can reach from one node to the other node in a graph.
- A graph is said to be **connected** if there is a path between every pair of vertices.
- If the graph is not connected, then all the connected subsets of the graphs are called **connected components**. Each component is connected within the self, but two different components of a graph are never connected.
- The minimum number of edges in a graph can be zero, which means a graph could have no edges as well.
- The minimum number of edges in a connected graph will be **(N-1)**, where **N** is the number of nodes.

- In a complete graph (where each node is connected to every other node by a direct edge), there are $^{N}C_{2}$ number of edges means **(N * (N-1)) / 2** edges, where n is the number of nodes.
- This is the maximum number of edges that a graph can have.
- Hence, if an algorithm works on the terms of edges, let's say **O(E)**, where **E** is the number of edges, then in the worst case, the algorithm will take **O(N^2)** time, where **N** is the number of nodes.

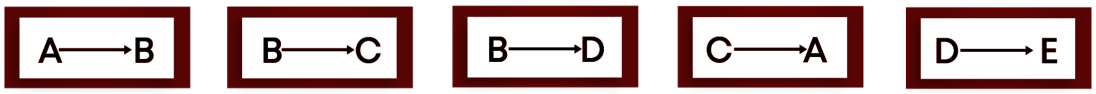
Graphs Implementation

Suppose the graph is as follows:



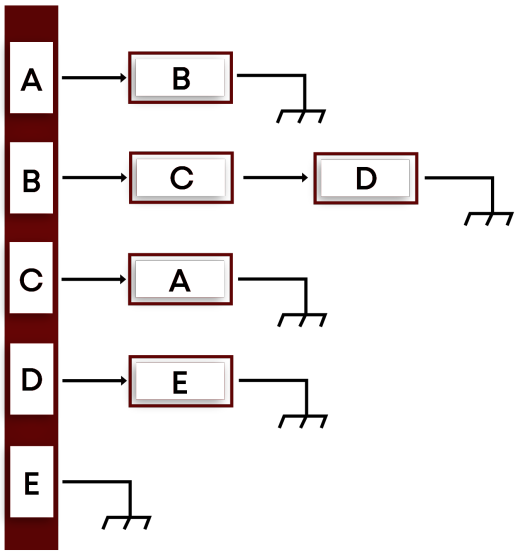
There are the following ways to implement a graph:

1. **Using edge list:** We can create a class that could store an array of edges. The array of edges will contain all the pairs that are connected, all put together in one place. It is not preferred to check for a particular edge connecting two nodes; we have to traverse the complete array leading to O(n^2) time complexity in the worst case. Pictorial representation for the above graph using the edge list is given below:



2. **Adjacency list:** We will create an array of vertices, but this time, each vertex will have its list of edges connecting this vertex to another vertex. Now to check for a particular edge, we can take any one of the nodes and then check

in its list if the target node is present or not. This will take O(n) work to figure out a particular edge.



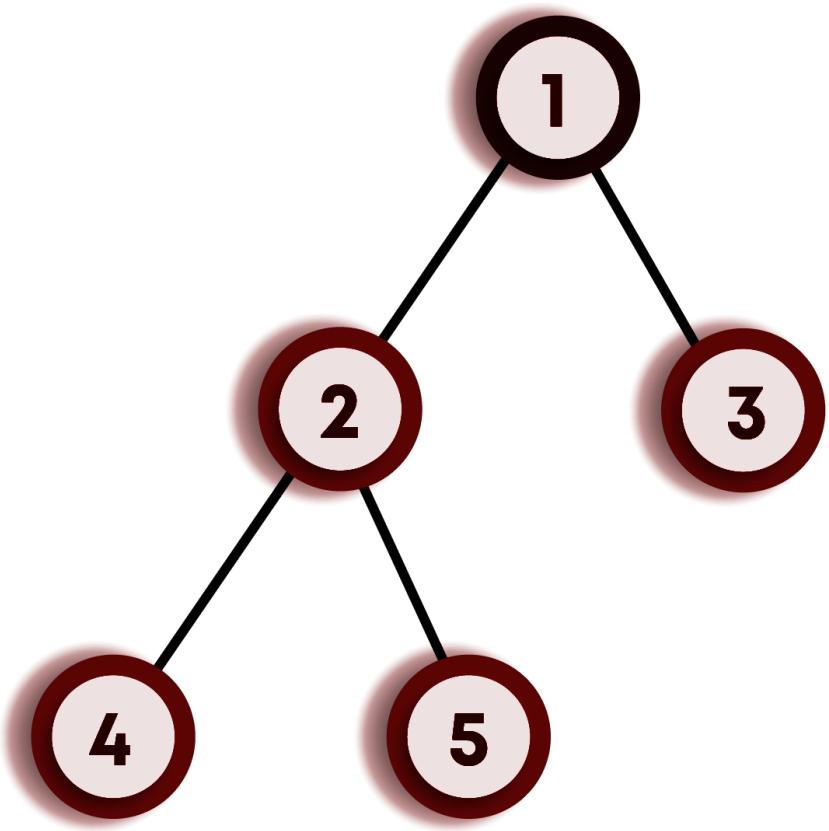
3. **Adjacency matrix:** Here, we will create a 2D array where the cell (i, j) will denote an edge between node i and node j. It is the most reliable method to implement a graph in terms of ease of implementation. We will be using the same throughout the session. The major disadvantage of using the adjacency matrix is vast space consumption compared to the adjacency list, where each node stores only those nodes that are directly connected to them. For the above graph, the adjacency matrix looks as follows:

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	1	0	0	0	0
D	0	0	0	0	1
E	0	0	0	0	0

DFS (Depth First Search)- Take Input and Print Graph

- Suppose we wish to create a graph containing N edges.
- We will store these edges in an Adjacency Matrix.
- We will run the loop from **0** to **number_of_edges**, and at each iteration, we will take input for the two connected nodes and correspondingly update the adjacency matrix. Let's look at the Python code for a better understanding.

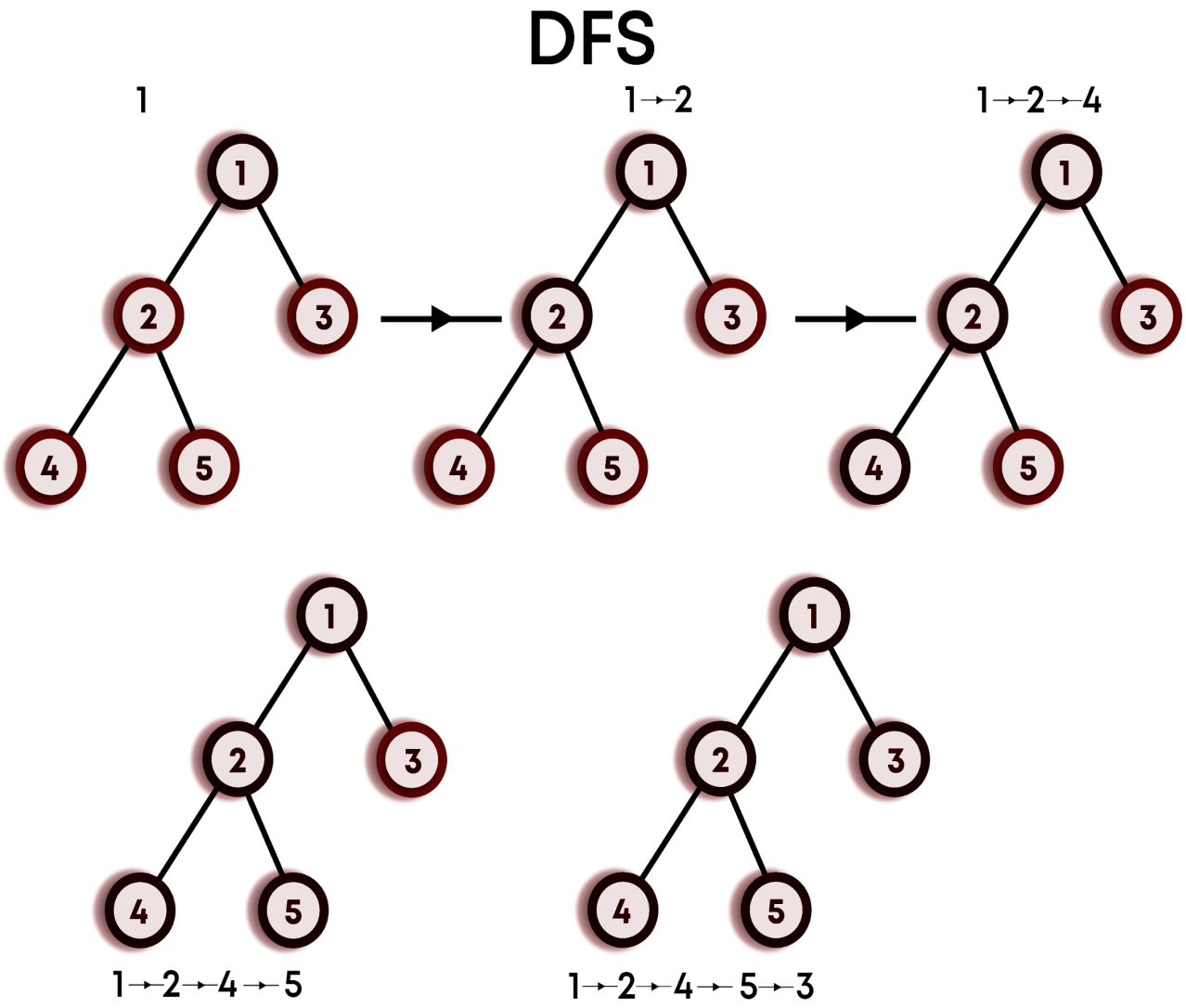
Let's take an example graph:



On dry running the above code, the output will be **1 2 4 5 3.**

Here, we are starting from a node, going in one direction as far as we can, and then we return and do the same on the previous nodes. This method of graph traversal is known as the **depth-first search (DFS)**. As the name suggests, this algorithm

goes into the depth-first and then recursively does the same in other directions. Follow the figure below, for step-by-step traversal using DFS.



```
class Graph:

      def __init__ (self,nVertices):
            self.nVertices = nVertices
            self.adjMatrix = [[0 for i in range(nVertices)] for j in range(nVertices)]

      def addEdge(self,v1,v2):
            self.adjMatrix[v1][v2] = 1
            self.adjMatrix[v2][v1] = 1

      def __dfsHelper(self,sv,visited):
            print(sv)
            visited[sv]= True
            for i in range(self.nVertices):
                  if(self.adjMatrix[sv][i]>0 and visited[i] is False:
                        self.__dfsHelper(i, visited)
      def dfs(self):
            visited = [False for i in range(self.nVertices)]
            self.__dfsHelper(0,visited)

      def removeEdge(self,v1,v2):
            if not self.containsEdge(v1,v2):
                  return
            self.adjMatrix[v1][v2] = 0
            self.adjMatrix[v2][v2] = 0

      def containsEdge(self,v1,v2):
            return True if self.adjMatrix[v1][v2] > 0 else False
```

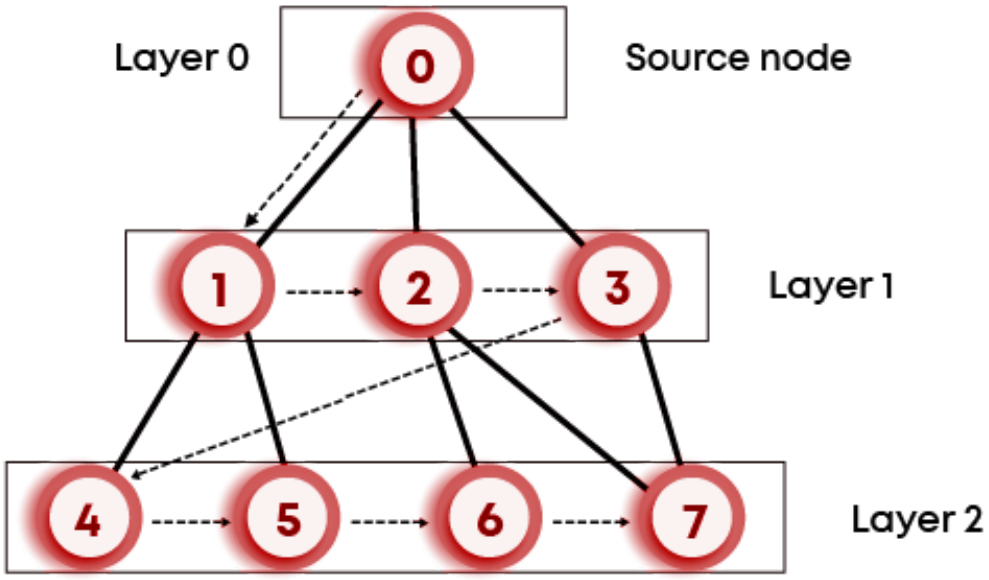
BFS Traversal

Breadth-first search(BFS) is an algorithm where we start from the selected node and traverse the graph level-wise or layer-wise, thus exploring the neighbor nodes

(which are directly connected to the starting node), and then moving on to the next level neighbor nodes.

As the name suggests:

- We first move horizontally and visit all the nodes of the current layer.
- Then move to the next layer.



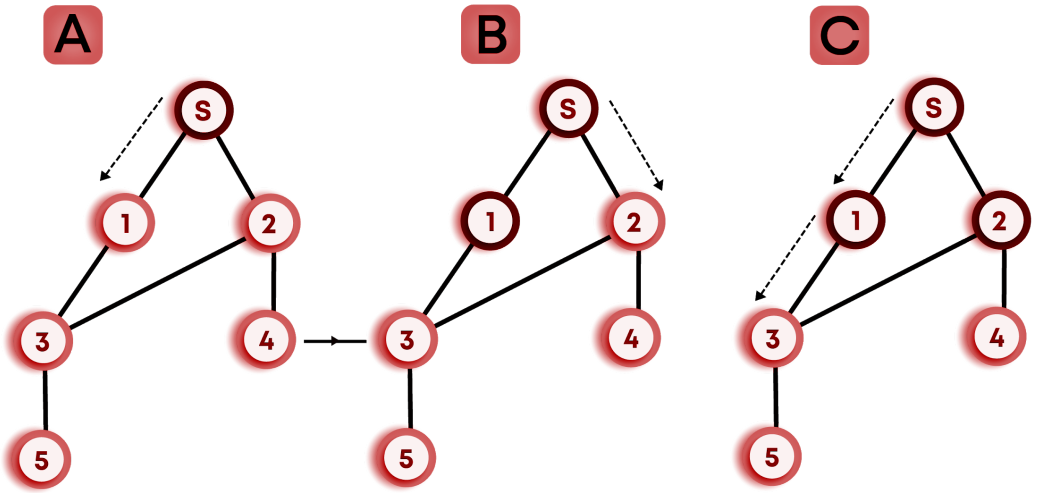
This is an iterative approach. We will use the queue data structure to store the child nodes of the current node and then pop out the current node. This process will continue until we have covered all the nodes. Remember to put only those nodes in the queue which have not been visited.

Let's look at the code below:

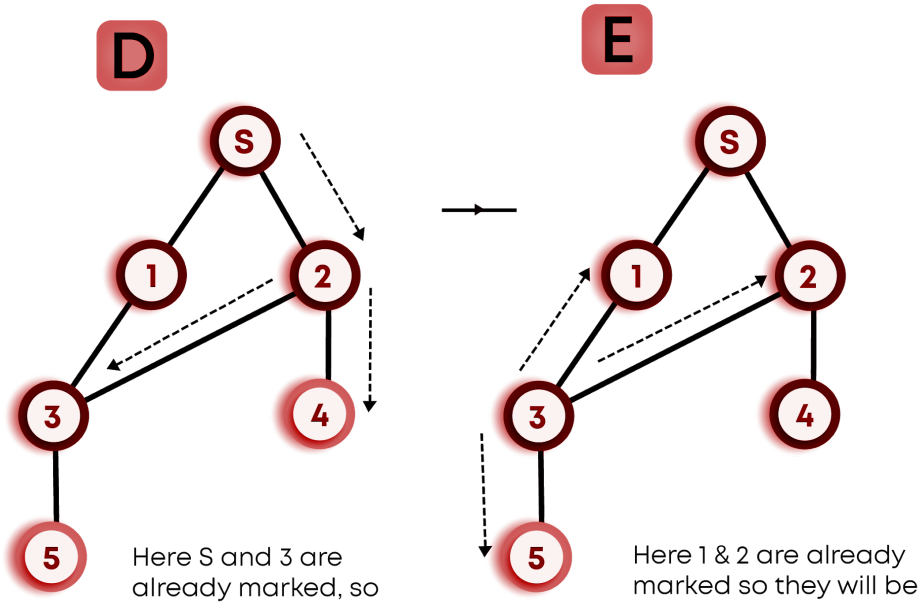
```
class Graph:

      def __init__ (self,nVertices):
            self.nVertices = nVertices
```

```
        self.adjMatrix = [[0 for i in range(nVertices)] for j in range(nVertices)]

    def addEdge(self,v1,v2):
        self.adjMatrix[v1][v2] = 1
        self.adjMatrix[v2][v1] = 1

    def __bfs(self, sv, visited):
        q = queue. Queue()
        q.put(sv)

        visited[sv] = True

        while q.empty() is False :
            u = q.get()
            print(u, end=" ")
            for i in range(self.nVertices) :
                if self.adjMatrix[u][i] > 0 and visited[i] is False :
                    q.put(i)
                    visited[i] = True

    def bfs(self):
        visited = [False for i in range(self.nVertices)]
        for i in range(self.nVertices):
            if visited[i] is False:
                self.__bfs(i, visited)



    def removeEdge(self,v1,v2):
        if not self.containsEdge(v1,v2):
            return
        self.adjMatrix[v1][v2] = 0
        self.adjMatrix[v2][v2] = 0

    def containsEdge(self,v1,v2):
        return True if self.adjMatrix[v1][v2] > 0 else False
```

Consider the dry run over the example graph below for a better understanding of the same:

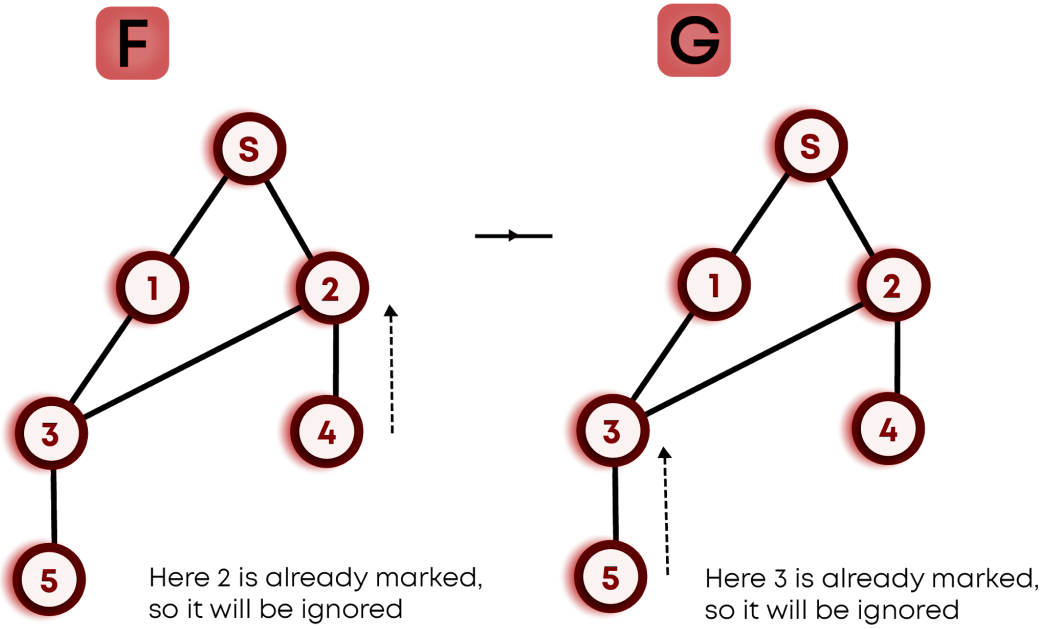


Here S is already marked, so it will be ignored



Here S and 3 are already marked, so they will be ignored

Here 1 & 2 are already marked so they will be ignored



BFS & DFS for Disconnected Graph

Till now, we have assumed that the graph is connected. For the disconnected graph, there will be a minor change in the above codes. Just before calling out the print functions, we will run a loop over each node and check if that node is visited or not. If not visited, then we will call a print function over that node, considering it as the starting vertex. In this way, we will be able to cover up all the nodes of the graph.

Consider the same for the BFS function. Just replace this function in the above code to make it work for the disconnected graph too.

Has Path

Problem statement: Given an undirected graph G(V, E) and two vertices v1 and v2(as integers), check if there exists any path between them or not. Print true or false. V is the number of vertices present in graph G, and vertices are numbered from 0 to V-1. E is the number of edges present in graph G.

Approach: This can be simply solved by considering the vertex v1 as the starting vertex and then run either BFS or DFS as per your choice, and while traversing if we reach the vertex v2, then we will simply return true, otherwise return false.

This problem has been left for you to try yourself. For code, refer to the solution tab of the same.

Get Path - DFS

Problem statement: Given an undirected graph G(V, E) and two vertices v1 and v2(as integers), find and print the path from v1 to v2 (if exists). Print nothing if there is no path between v1 and v2.
Find the path using DFS and print the first path that you encountered irrespective of the length of the path. V is the number of vertices present in graph G, and vertices are numbered from 0 to V-1. E is the number of edges present in graph G. Print the path in reverse order. That is, print v2 first, then intermediate vertices, and v1 at last.
Example: Suppose the given input is:

```
4 4
0 1
```

```
0 3
1 2
2 3
1 3
```

The output should be:

```
3 0 1
```

Explanation: Here, v1 = 1 and v2 = 3. The connected vertex pairs are (0, 1), (0, 3), (1, 2) and (2, 3). So, according to the question, we have to print the path from vertex v1 to v2 in reverse order using DFS only; hence the path comes out to be {3, 0, 1}.

Approach: We have to solve this problem by using DFS. Suppose, if the start and end vertex are the same, then we simply need to put the start in the solution array and return the solution array. If this is not the case, then from the start vertex, we will call DFS on the direct connections of the same. If none of the paths leads to the end vertex, then we do not need to push the start vertex as it is neither directly nor indirectly connected to the end vertex, hence we will simply return NULL. In case any of the neighbors return a non-null entry, it means that we have a path from that neighbor to the end vertex, hence we can now insert the start vertex into the solution array.
Try to code it yourself, and for the answer, refer to the solution tab of the same.

Get Path - BFS

Approach: It is the same problem as the above, just we have to code the same using BFS.

Approach: Using BFS will provide us the shortest path between the two vertices. We will use the queue over here and do the same until the end vertex gets inserted into the queue. Here, the problem is how to figure out the node, which led us to the

end vertex. To overcome this, we will be using a map. In the map, we will store the resultant node as the index, and its key will be the node that led it into the queue.

For example: If the graph was such that 0 was connected to 1 and 0 was connected to 2, and currently, we are on node 0 such that node 1 and node 2 are not visited. So our map will look as follows:

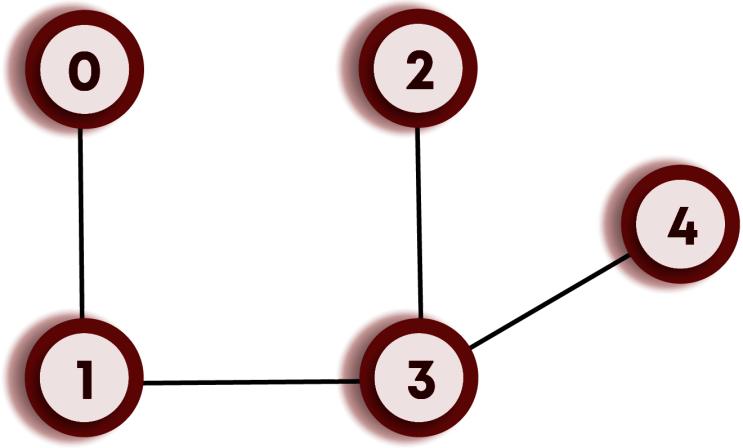
1	0
2	0

This way, as soon as we reach the end vertex, we can figure out the nodes by running the loop until we reach the start vertex as the key value of any node.

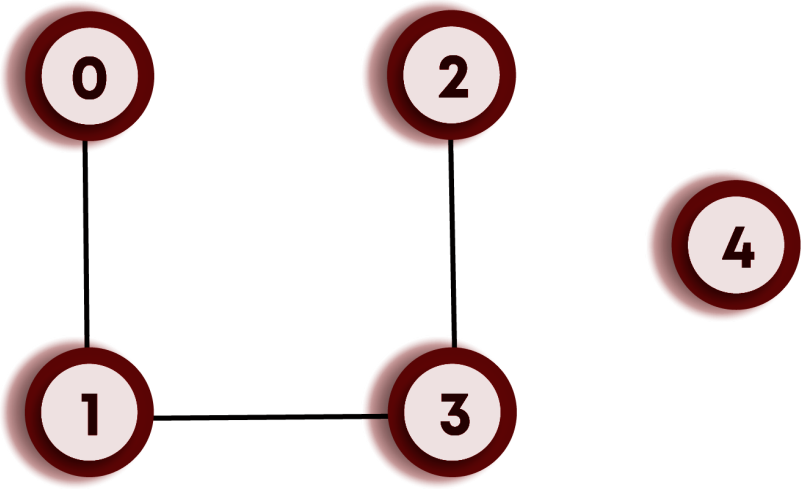
Try to code it yourselves, and for the solution, refer to the specific tab of the same.

Is connected?

Problem statement: Given an undirected graph G(V, E), check if the graph G is a connected graph or not. V is the number of vertices present in graph G, and vertices are numbered from 0 to V-1. E is the number of edges present in graph G.



Connected graph



Disconnected graph

Example 1: Suppose the given input is:

```
4 4
0 1
0 3
1 2
2 3
1 3
```

The output should be: **true**

Explanation: As the graph is connected, so according to the question, the answer will be true.

Example 2: Suppose the given input is:

```
4 3
0 1
1 3
0 3
```

The output should be: **false**

Explanation: The graph is not connected, even though vertices 0,1, and 3 are connected, but there isn't any path from vertices 0,1,3 to vertex 2. Hence, according to the question, the answer will be false.

Approach: This is very start-forward. Take any vertex as the starting vertex as traverse the graph using either DFS or BFS. In the end, check if all the vertices are visited or not. If not, it means that the node was not connected to the starting vertex, which means it is a disconnected graph. Otherwise, it is a connected graph. Try to code it yourselves, and for the code, refer to the solution tab of the same.

Problem statement: Return all Connected Components

Given an undirected graph G(V, E), find and print all the connected components of the given graph G. V is the number of vertices present in graph G, and vertices are numbered from 0 to V-1. E is the number of edges present in graph G.
You need to take input in the main and create a function that should return all the connected components. And then print them in the main, not inside a function.
Print different components in a new line. And each component should be printed in increasing order (separated by space). The order of different components doesn't matter.

Example: Suppose the given input is:

```
4 3
0 1
1 3
0 3
```

The output should be:

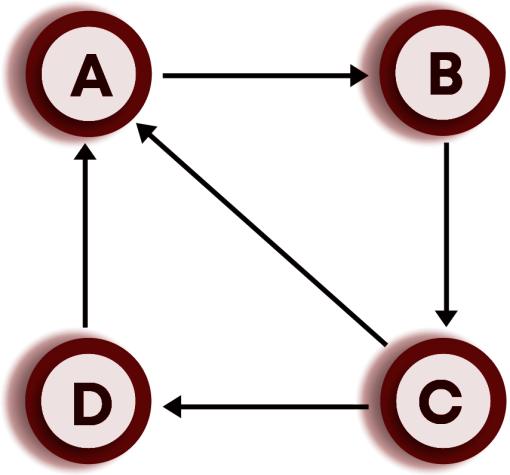
```
0 1 3
2
```

Explanation: As we can see that {0, 1, 3} is one connected component, and {2} is the other one. So, according to the question, we just have to print the same.
Approach: For this problem, start from vertex 0 and traverse until vertex n-1. If the vertex is not visited, then run DFS/BFS on it and keep track of all the connected vertices through that node. This way, we will get all the distinct connected components, and we can print them at last.
This problem is left for you to solve. For the code, refer to the solution tab of the same.

Weighted and Directed Graphs

There are two more variations of the graphs:

Directed graphs: These are generally required when we have one-way routes. Suppose you can go from node A to node B, but you cannot go from node B to node A. Another example could be social media(like Twitter) if you are following someone, it does not mean that they are following you too.



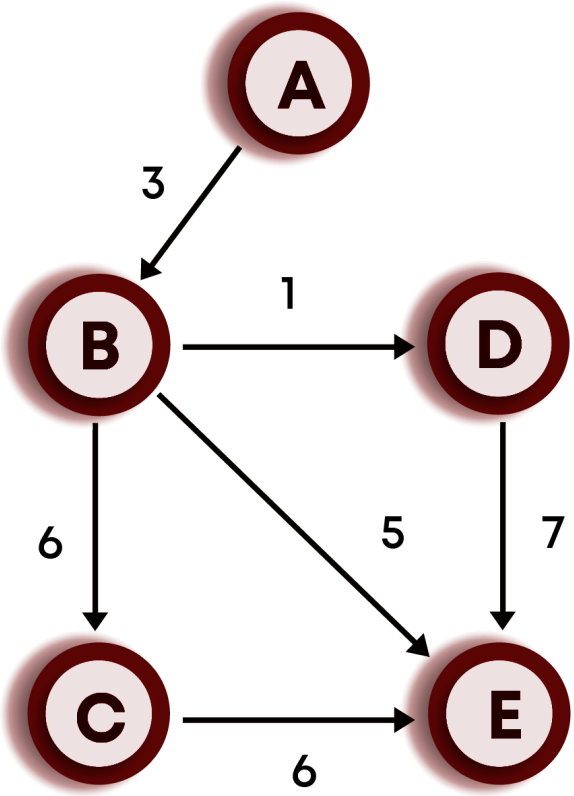
To implement these, there is a small change in the implementation of indirect graphs. In indirect graphs, if there was an edge between node i and j, then we did:

```
edges[i][j] = 1;
edges[j][i] = 1;
```

But, in the case of a directed graph, we will just do the following:

```
edges[i][j] = 1;
```

Weighted graphs: These generally mean that all the edges are not equal, which means somehow, each edge has some weight assigned to it. This weight can be the length of the road connecting the cities or many more.



To implement this, in the edges matrix, we will assign a weight to connected nodes instead of putting it 1 at that position. For example: If node i and j are connected, and the weight of the edge connecting them is 5, then `edges[i][j] = 5`.

Practice problems:

- https://www.codechef.com/problems/CHEFDAG
- https://www.spoj.com/problems/WORDS1/
- https://www.hackerrank.com/challenges/the-quickest-way-up/problem

Graphs- 2

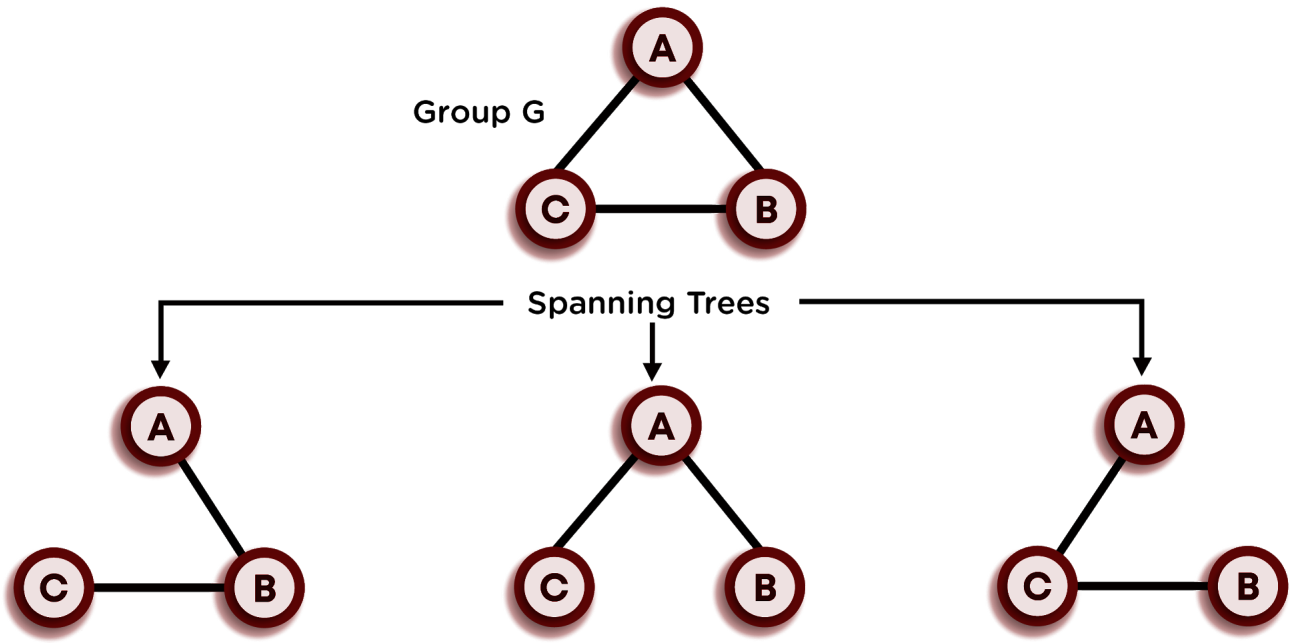
MST(Minimum Spanning Tree) & Kruskal's Algorithm

As discussed earlier, a tree is a graph, which:

- Is always connected.
- Contains no cycle.

If we are given an undirected and connected graph, a **spanning tree** means a tree that contains all the vertices of the same. For a given graph, we can have multiple spanning trees.

Refer to the example below for a better understanding of spanning trees.



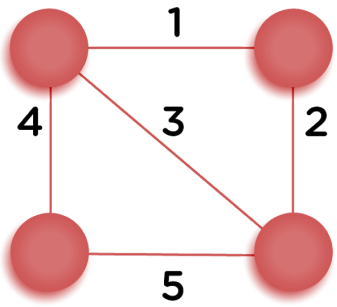
If there are *n* vertices and *e* edges in the graph, then any spanning tree corresponding to that graph contains *n* vertices and *n-1* edges.

Properties of spanning trees:

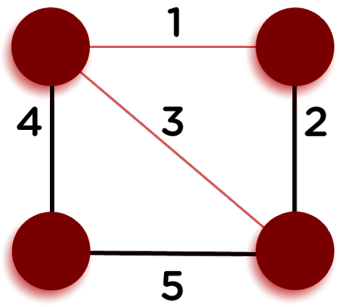
- A connected and undirected graph can have more than one spanning tree.
- The spanning tree is free of loops, i.e., it is acyclic.
- Removing any one of the edges will make the graph disconnected.
- Adding an extra edge to the spanning tree will create a loop in the graph.

Minimum Spanning Tree(MST) is a spanning tree with weighted edges.

In a weighted graph, the MST is a spanning tree with minimum weight than all other spanning trees of that graph. Refer to the image below for better understanding.

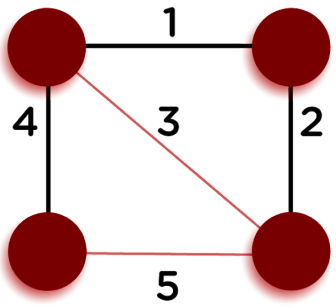


Undirected graph



Spanning tree

Cost=11(=4+5+2)



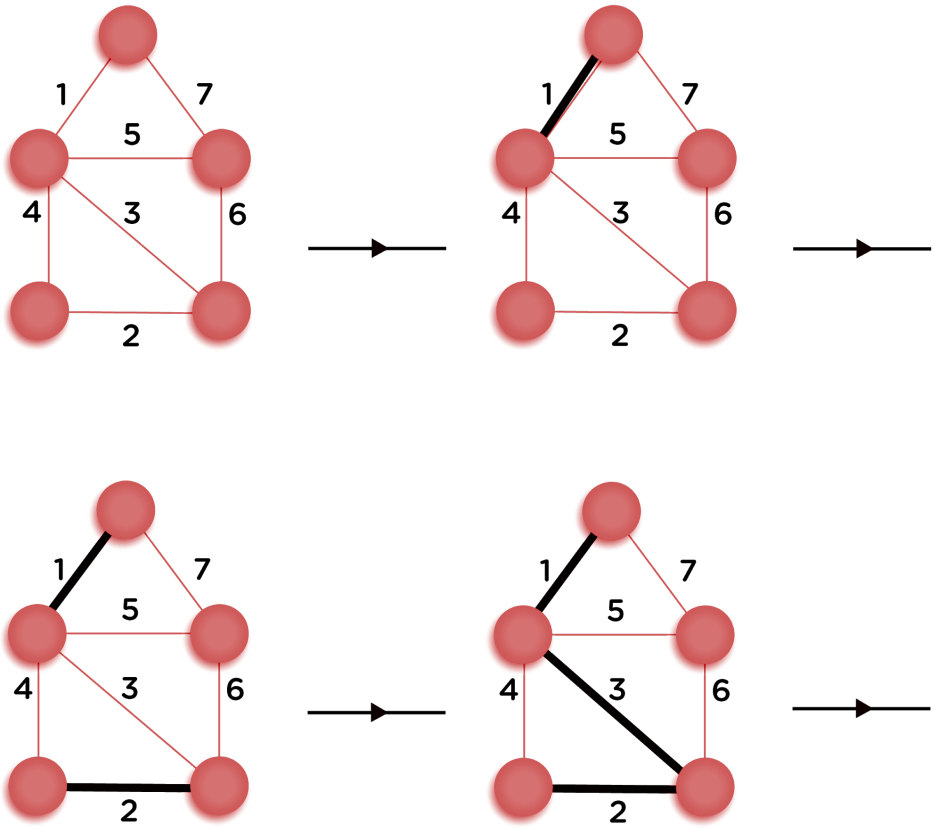
Minimum Spanning tree

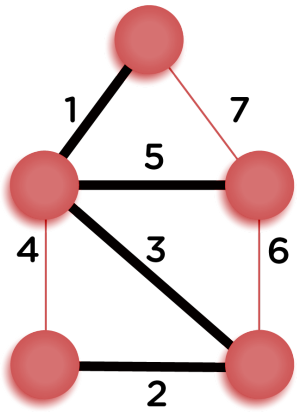
Cost=7(=4+1+2)

Kruskal's Algorithm:

This algorithm is used to find MST for the given graph. It builds the spanning tree by adding edges one at a time. We start by picking the edge with minimum weight, adding that edge into the MST, and increasing the count of edges in the spanning tree by one. Now, we will be picking the minimum weighted edge by excluding the already chosen ones and correspondingly increasing the count. While choosing the edge, we will also make sure that the graph remains acyclic after including the same. This process will continue until the count of edges in the MST reaches *n-1*. Ultimately, the graph obtained will be MST.

Refer to the example below for a better understanding of the same.





This is the final MST obtained using Kruskal's algorithm. It can be checked manually that the final graph is the MST for the given graph.

Cycle Detection

While inserting a new edge in the MST, we have to check if introducing that edge makes the MST cyclic or not. If not, then we can include that edge, otherwise not.

Now, let's figure out a way to detect the cycle in a graph. The following are the possible cases:

- By including an edge between the nodes A and B, if both nodes A and B are not present in the graph, then it is safe to include that edge as including it, will not bring a cycle to the graph.
- Out of two vertices, if any one of them has not been visited (or not present in the MST), then that vertex can also be included in the MST.
- If both the vertices are already present in the graph, they can introduce a cycle in the MST. It means we can't use this method to detect the presence of the cycle.

Let's think of a better approach. We have already solved the **hasPath** question in the previous module, which returns true if there is a path present between two vertices v1 and v2, otherwise false.

Now, before adding an edge to the MST, we will check if a path between two vertices of that edge already exists in the MST or not. If not, then it is safe to add that edge to the MST.

As discussed in previous lectures, the time complexity of the **hasPath** function is O(E+V), where E is the number of edges in the graph and, V is the number of vertices. So, for (n-1) edges, this function will run (n-1) times, leading to bad time complexity, as in the worst case, $E = V^2$.

Now, moving on to a better approach for cycle detection in the graph.

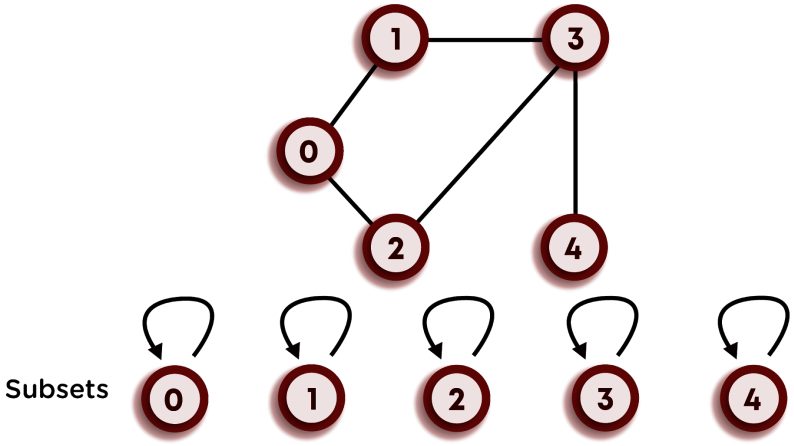
Union-Find Algorithm:

Before adding any edge to the graph, we will check if the two vertices of the edge lie in the same component of MST or not. If not, then it is safe to add that edge to the MST.

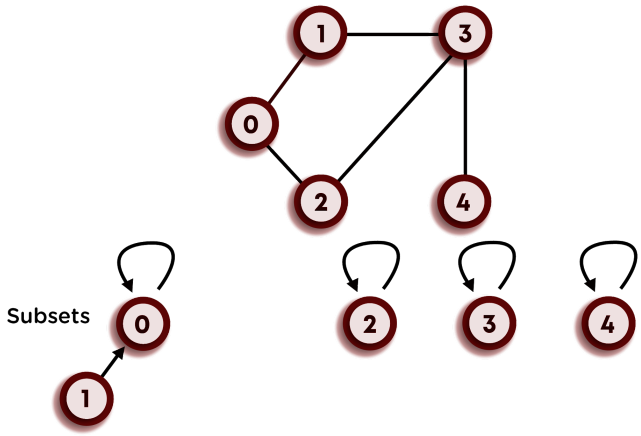
Following the steps of the algorithm:

- We will assume that initially, the total number of disjoint sets is equal to the number of vertices in the graph starting from 0 to n-1.
- We will maintain a parent array specifying the parent vertex of each of the vertex of the graph. Initially, as each vertex belongs to a different disjoint set (connected component), hence each vertex will be its parent.
- Now, before inserting any edge into the MST, we will check the parent of the vertices. If their parent vertices are equal, they belong to the same connected component; hence it is unsafe to add that edge.
- Otherwise, we can add that edge into the MST, and simultaneously update the parent array so that they belong to the same component(Refer to the code on how to do so).

Look at the following example, for better understanding:



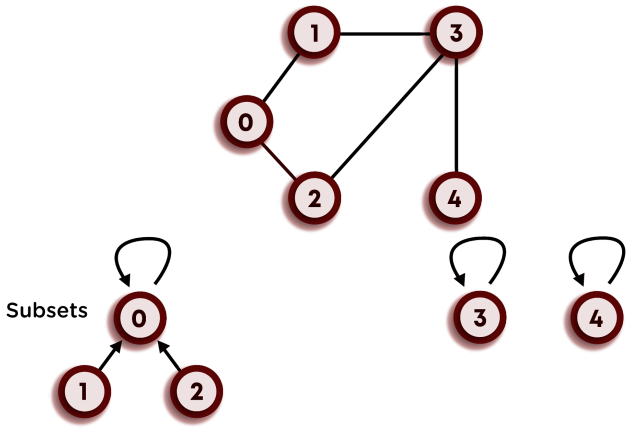
Initially all parent pointers are pointing to self
means only one element in each subset



Edge 0-1

Find: 0 belongs to subset 0 and 1 belongs to subset 1 so they are in different subsets.

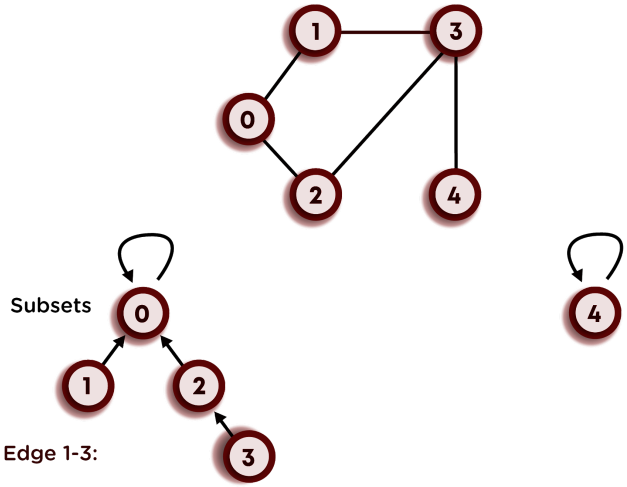
Union: Make 0 as the parent of 1, Updated set is {0,1}. 0 is the set representative since 0 is parent for itself.



Edge 0-2:

Find: 0 belongs to subset 0 and 2 belongs to subset 2 so they are in different subsets.

Union: Make 0 as the parent of 2, Updated set is {0,1,2}. 0 is the set representative since 0 is parent for itself.

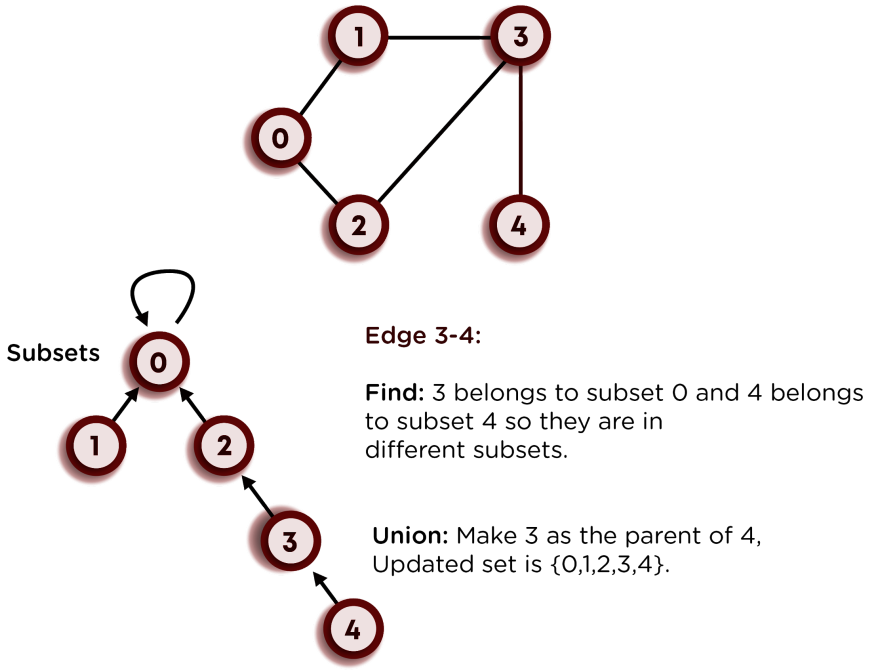


Edge 1-3:

Find: 1 belongs to subset 0 and 3 belongs to subset 3 so they are in different subsets.

Union: Make 1 as the parent of 3, Updated set is {0,1,2,3}. 0 is the set representative since 0 is parent for itself.

Note: While finding the parent of the vertex, we will be finding the topmost parent (Oldest Ancestor). For example: suppose, the vertex 0 and the vertex 1 were



connected, where the parent of 0 is 1, and the parent of 1 is 1. Now, while determining the parent of the vertex 0, we will visit the parent array and check the vertex at index 0. In our case, it is 1. Now we will go to index 1 and check the parent of index 1, which is also 1. Hence, we can't go any further as the index is the parent of itself. This way, we will be determining the parent of any vertex.

The time complexity of the union-find algorithm becomes O(V) for each vertex in the worst case due to skewed-tree formation, where V is the number of vertices in the graph. Here, we can see that time complexity for cycle detection has significantly improved compared to the previous approach.

Kruskal's Algorithm: Implementation

Till now, we have studied the logic behind Kruskal's algorithm for finding MST. Now, let's discuss how to implement it in code.

Consider the code below and follow the comments for a better understanding.

```
class Edge:# Class that store values for each vertex
      def __init__ (self,src,dest,wt):
            self.src = src
            self.dest dest
            self.wt = wt

def getParent(v,parent): # Function to find the parent of a vertex
      if v == parent[v]: #When a vertex is parent of itself
            return v  # Recursively called to find the topmost parent
      return getParent(parent[v], parent)

def kruskal(edges,n,E):
      edges = sorted(edges,key = lambda edge:edge.wt) #Inbuilt sort
      output = []  # Array to store final edges of MST
      parent [i for i in range(n)]
      count = 0
      i = 0
      while count < (n-1):
            currentEdge = edges[i]
            # Figuring out the parent of each edge's vertices
            srcParent = getParent(currentEdge.src,parent)
            destParent = getParent(currentEdge.dest,parent)
            # Parents are not equal-> then add the edge to output
            if srcParent != destParent:
                  output.append(currentEdge)
                  count+=1  # Increased the count
                  parent[srcParent] = destParent #Update Parent array
            i+=1
      return output


li = [int(ele) for ele in input().split()]
```

```
n= li[0]
E= li[1]
edges = []
for i in range(E) :
      curr_input = [int(ele) for ele in input().split()]
      src = curr_input[0]
      dest = curr_input[1]
      wt = curr_input[2]
      edge = Edge(src,dest,wt)
      edges.append(edge)

output = kruskal(edges,n,E)
for ele in output:# Finally, printing the MST obtained.
      if(ele.src < ele.dest):
            print(str(ele.src) +" "+ str(ele.dest)+" "+ str(ele.wt)
      else:
            print(str(ele.dest) + " "+ str(ele.src)+ " "+ str(ele.wt)
```

Time Complexity of Kruskal's Algorithm:

In our code, we have the following three steps: (Here, the total number of vertices is n, and the total number of edges is E)

- Take input in the array of size E.
- Sort the input array based on edge-weight. This step has the time complexity of O(E log(E)).
- Pick (n-1) edges and put them in MST one-by-one. Also, before adding the edge to the MST, we checked for cycle detection for each edge. For cycle detection, in the worst-case time complexity of E edges will be O(E.n), as discussed earlier.

Hence, the total time complexity of Kruskal's algorithm becomes O(E log(E) + n.E). This time complexity is bad and needs to be improved.

We can't reduce the time taken for sorting, but the time taken for cycle detection can be improved using another algorithm named **Union by Rank and Path Compression**. You need to explore this on yourselves. The basic idea in these

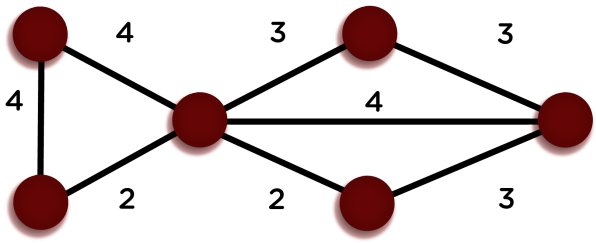
algorithms is that we will be avoiding the formation of skewed-tree structure, which reduces the time complexity for each vertex to O(log(E)).

Prim's Algorithm

This algorithm is used to find MST for a given undirected-weighted graph (which can also be achieved using Kruskal's Algorithm).

In this algorithm, the MST is built by adding one edge at a time. In the beginning, the spanning tree consists of only one vertex, which is chosen arbitrarily from the set of all vertices of the graph. Then the minimum weighted edge, outgoing from this vertex, is selected and simultaneously inserted into the MST. Now, the tree contains two edges. Further, we will be selecting the edge with the minimum weight such that one end is already present there in the MST and the other one from the unselected set of vertices. This process is repeated until we have inserted a total of (n-1) edges in the MST.

Consider the following example for a better understanding.

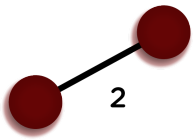


Step: 1

Start with a weighted graph

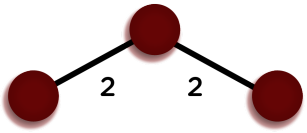
Step: 2

Choose a vertex



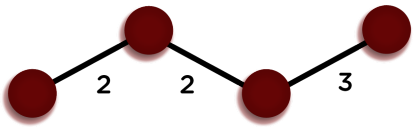
Step: 3

Choose the shortest edge from this vertex add it



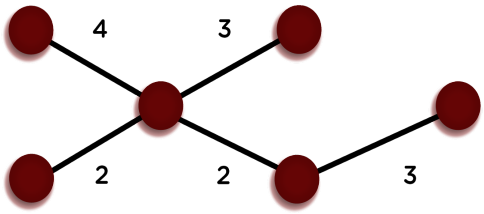
Step: 4

Choose the nearest vertex not yet in the solution



Step: 5

Choose the nearest edge not yet in the solution,
if there are multiple choices, choose one at random



Step: 6

Repeat until you have a spanning tree

Implementation:

- We are considering the starting vertex to be 0 with a parent equal to -1, and weight is equal to 0 (The weight of the edge from vertex 0 to vertex 0 itself).
- The parent of all other vertices is assumed to be NIL, and the weight will be equal to infinity, which means that the vertex has not been visited yet.
- We will mark the vertex 0 as visited and the rest as unvisited. If we add any vertex to the MST, then that vertex will be shifted from the unvisited section to the visited section.
- Now, we will update the weights of direct neighbors of vertex 0 with the edge weights as these are smaller than infinity. We will also update the parent of these vertices and assign them 0 as we reached these vertices from vertex 0.
- This way, we will keep updating the weights and parents, according to the edge, which has the minimum weight connected to the respective vertex.

Let's look at the code now:

```
class Graph:

      def __init__ (self,nVertices):
            self.nVertices = nVertices
            self.adjMatrix = [[0 for 1 in range(nVertices)] for j in range(nVertices)]

      def addEdge(self,v1,v2,wt):
            self.adjMatrix[v1][v2] = wt
            self.adjMatrix[v2][v1] =  wt

      def __getMinVertex(self,visited,weight):
            minVertex = -1  # Initialized to -1 means there is no vertex
            for i in range(self.nVertices):
            # Conditions :
            #the vertex must be unvisited and either minVertex value is -1
            #or if minVertex has some vertex to it, then weight of
            #currentVertex should be less than the weight of the minVertex.
                  a = (minVertex == -1 or (weight[minVertex] > weight[i]))
                  if(visited[i] is False and a:
                        minVertex = i
            return minVertex

      def prims(self):
            # Initially, the visited array is assigned to false
            #and weights array is set to infinity.
            visited = [False for i in range(self.nVertices)]
            parent = [-1 for i in range(self.nVertices)]
            weight = [sys.maxsize for i in range(self.nVertices)]

            for i in range(self.nVertices - 1):
            # Find min vertex
            minVertex = self.__getMinVertex(visited,weight)
            visited[minVertex] = True
            # Explore unvisited neighbors
            for j in range(self.nVertices):
                  if(self.adjMatrix[minVertex][j]>0 and visited[j] is False):
                        if(weight[j] > self.adjMatrix[minVertex][j]):
                              weight[j] = self.adjMatrix[minVertex][j]
                              parent[j] = minVertex
            # Final MST printed
            for i in range(1,self.nVertices):
                  if parent[i] > i:
```

```
                    a = str(i) +" "+ str(parent[i])+" "+ str(weight[i])
                    print(a)
                else:
                    a = str(parent[i]) +" "+ str(i) +" "+ str(weight[i])
                    print(a)

    def removeEdge(self,v1,v2):
        if not self.containsEdge(v1,v2):
            return
        self.adjMatrix[v1][v2] = 0
        self.adjMatrix[v2][v2] = 0

    def containsEdge(self,v1,v2):
        return True if self.adjMatrix[v1][v2] > 0 else False

li = [int(ele) for ele in input().split()]
n = li[0]
E= li[1]
g= Graph(n) #Inbuilt Graph class
for in range(E):
    curr_edge = [int(ele) for ele in input().split()]
    g.addEdge(curr_edge[0], curr_edge[1], curr_edge[2])
g.prims ()
```

Time Complexity of Prim's Algorithm:

Here, n is the number of vertices, and E is the number of edges.

- The time complexity for finding the minimum weighted vertex is O(n) for each iteration. So for (n-1) edges, it becomes $O(n^2)$.
- Similarly, for exploring the neighbor vertices, the time taken is $O(n^2)$.

It means the time complexity of Prim's algorithm is $O(n^2)$. We can improve this in the following ways:

- For exploring neighbors, we are required to visit every vertex because of the adjacency matrix. We can improve this by using an adjacency list instead of a matrix.

- Now, the second important thing is the time taken to find the minimum weight vertex, which is also taking a time of $O(n^2)$. Here, out of the available list, we are trying to figure out the one with minimum weight. This can be optimally achieved using a **priority queue** where the priority will be taken as weights of the vertices. This will take O(log(n)) time complexity to remove a vertex from the priority queue.

These optimizations can lead us to the time complexity of O((n+E)log(n)), which is much better than the earlier one. Try to write the optimized code by yourself.

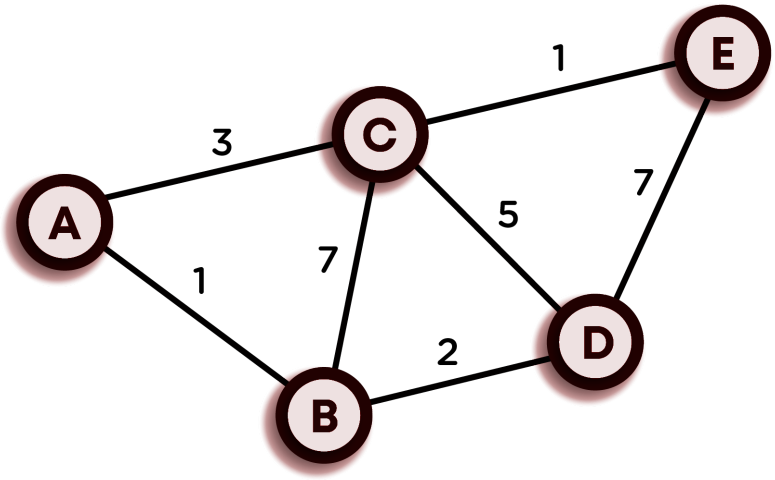
Dijkstra's Algorithm

This algorithm is used to find the shortest distance between any two vertices in a weighted non-cyclic graph.

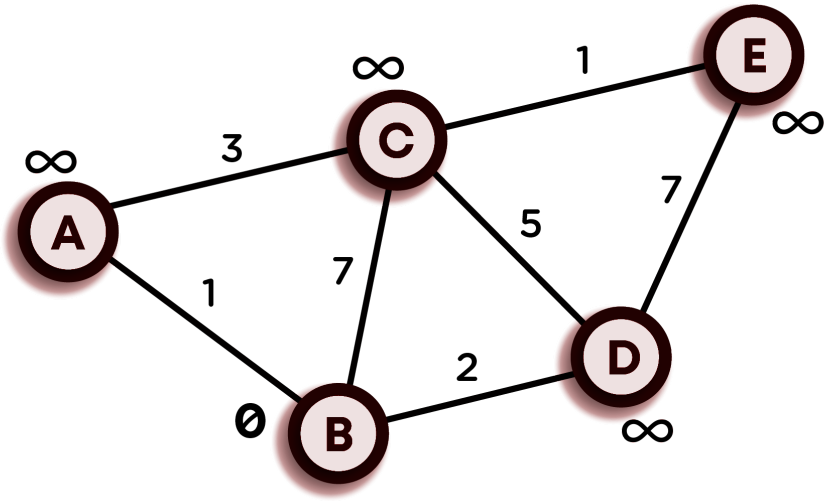
Here, we will be using a slight modification of the algorithm according to which we will be figuring out the minimum distance of all the vertices from the particular source vertex.

Let's consider the algorithm with an example:

1. We want to calculate the shortest path between the source vertex C and all other vertices in the following graph.

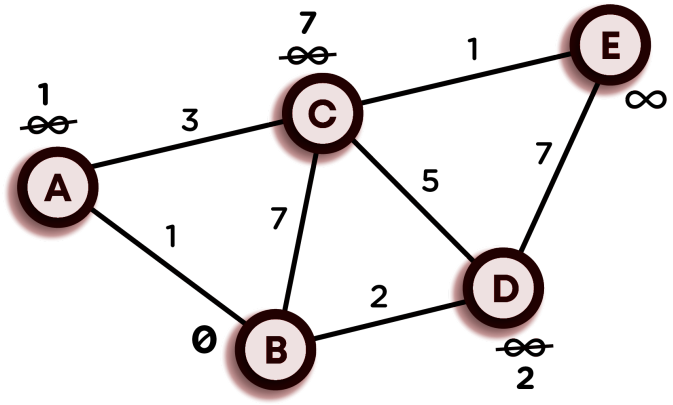


2. While executing the algorithm, we will mark every node with its **minimum distance** to the selected node, which is C in our case. Obviously, for node C itself, this distance will be 0, and for the rest of the nodes, we will assume that the distance is infinity, which also denotes that these vertices have not been visited till now.

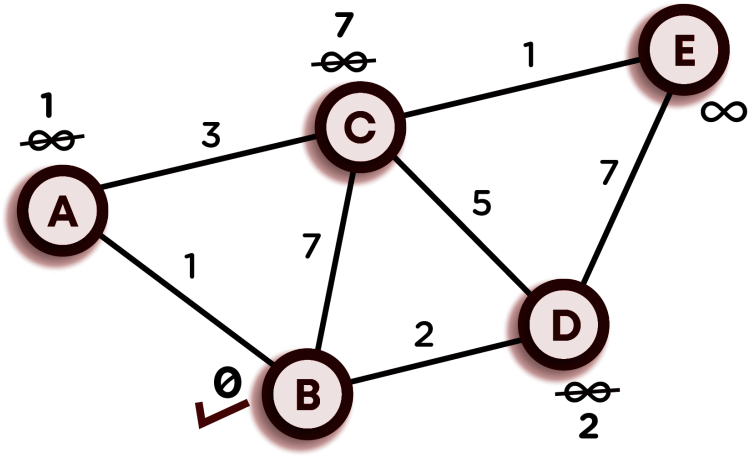


3. Now, we will check for the neighbors of the current node, which are A, B, and D. Now, we will add the minimum cost of the current node to the weight of

the edge connecting the current node and the particular neighbor node. For example, for node B, it's weight will become minimum(infinity, 0+7) = 7. This same process is repeated for other neighbor nodes.

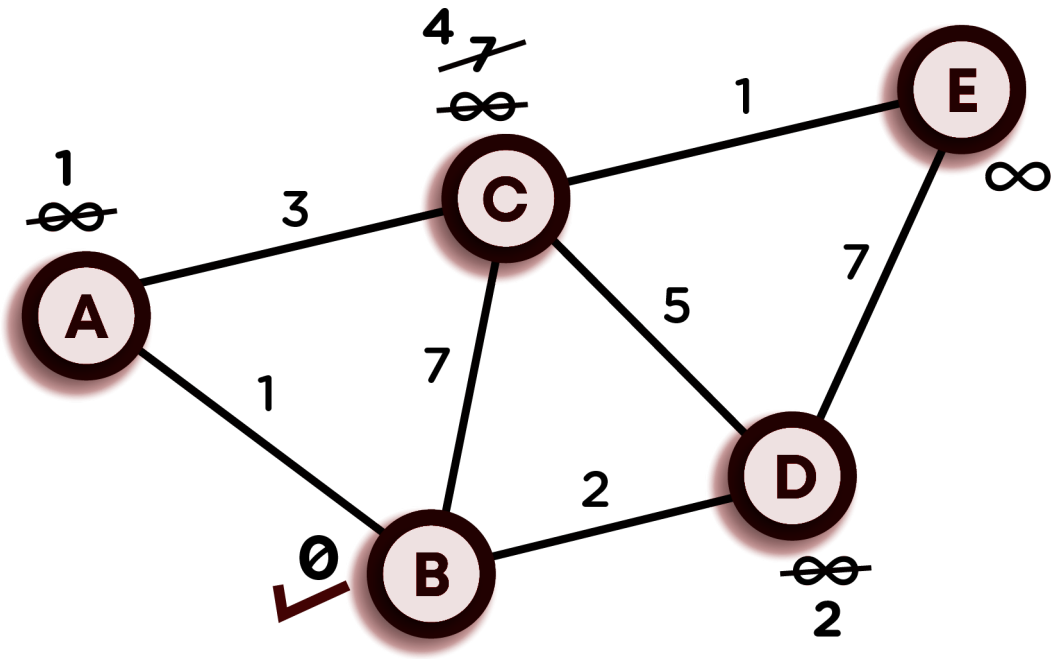


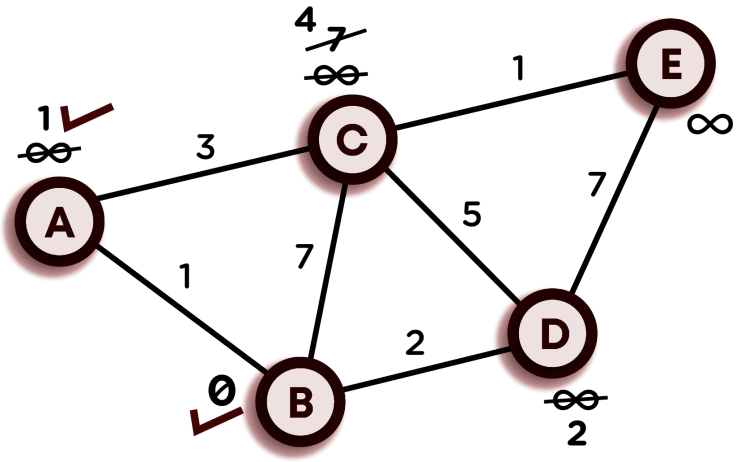
4. Now, as we have updated the distance of all the neighbor nodes of the current node, we will mark the current node as visited.

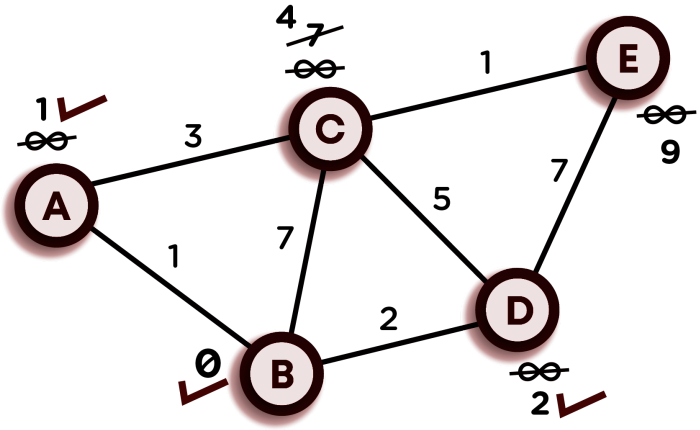


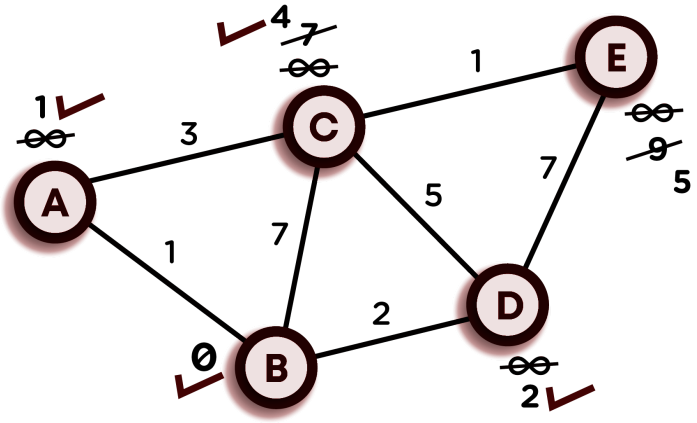
5. After this, we will be selecting the minimum weighted node among the remaining vertices. In this case, it is node A. Take this node as the current node.

6. Now, we will repeat the above steps for the rest of the vertices. The pictorial representation of the same is shown below:

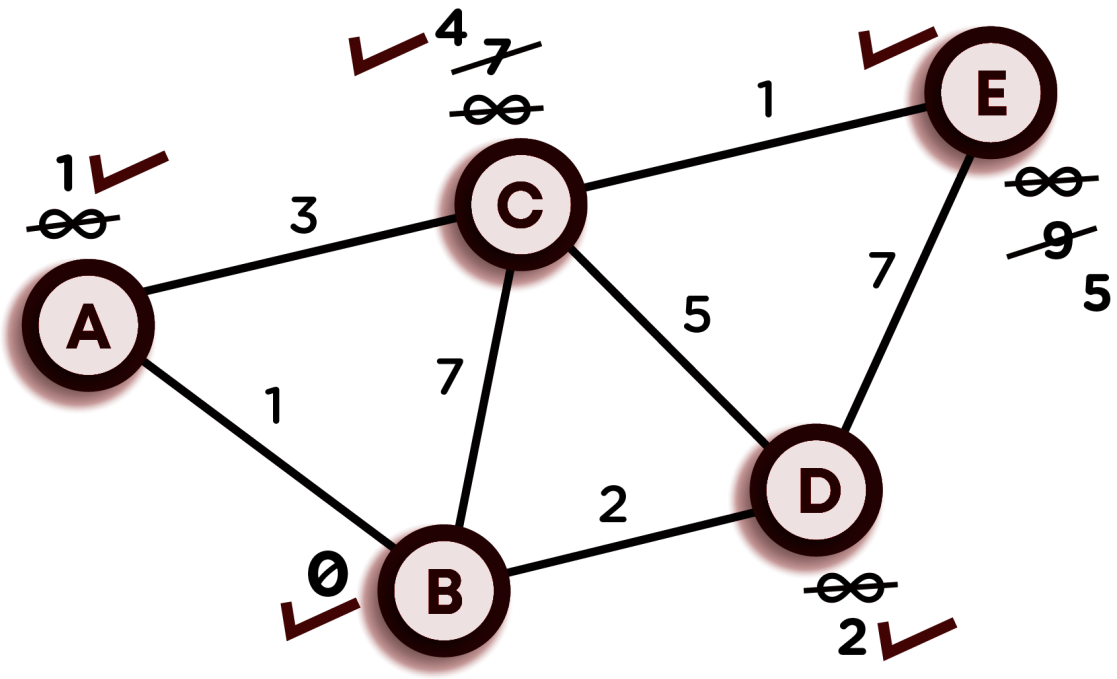








7. Finally, we will get the graph as follows:



The distances finally marked at each node are minimum from node C.

Implementation:

Let's look at the code below for a better explanation:(Code is nearly same as that of Prim's algorithm, just a change while updating the distance)

```
class Graph:

      def __init__ (self,nVertices):
            self.nVertices = nVertices
            self.adjMatrix = [[0 for 1 in range(nVertices)] for j in range(nVertices)]

      def addEdge(self,v1,v2,wt):
            self.adjMatrix[v1][v2] = wt
            self.adjMatrix[v2][v1] =  wt

      def __getMinVertexD(self,visited,weight):
            minVertex = -1  # Initialized to -1 means there is no vertex
            for i in range(self.nVertices):
                  a = (minVertex == -1 or (weight[minVertex] > weight[i]))
                  if(visited[i] is False and a:
                        minVertex = i
            return minVertex

      def djikstra(self):
            # Initially, the visited array is assigned to false
            #and weights array is set to infinity.
            visited = [False for i in range(self.nVertices)]
            dist = [sys.maxsize for i in range(self.nVertices)]
            dist[0]=0
            for i in range(self.nVertices - 1):
                  # Find min vertex
                  minVertex = self.__getMinVertexD(visited,weight)
                  visited[minVertex] = True
                  # Explore unvisited neighbors
                  for j in range(self.nVertices):
                        if(self.adjMatrix[minVertex][j]>0 and visited[j] is False):
                              if(dist[j] > dist[minVertex]+ self.adjMatrix[minVertex][j]):
                              dist[j] = self.adjMatrix[minVertex][j] +dist[minVertex]
                              parent[j] = minVertex

            # Final MST printed
            for i in range(self.nVertices):
                 print(str(i)+" " + str(dist[i]))
```

```
li = [int(ele) for ele in input().split()]
n = li[0]
E= li[1]
g= Graph(n) #Inbuilt Graph class
for in range(E):
      curr_edge = [int(ele) for ele in input().split()]
      g.addEdge(curr_edge[0], curr_edge[1], curr_edge[2])
g.djikstra()
```

Time Complexity of Dijkstra's algorithm:

The time complexity is also the same as that of Prim's algorithm, i.e., **$O(n^2)$**. This can be reduced by using the same approaches as discussed in Prim's algorithm's content.

Recursion-3

In this module, we are going to solve a few advanced problems using recursion.

Problem Statement: Return Subsequences

Given a string (let's say of length n), return all the subsequences of the given string. **Subsequences** contain all the strings of length varying from 0 to n. But the order of characters should remain the same as in the input string.
Note: The order of subsequences is not important.

Approach:

For example, If we have a string : **"abc"**, Then its subsequences are :- **"a", "b", "c", "ab", "ac", "bc", "abc"**

- First, we select the first character i.e. 'a', add it to a list, and fix it. ----> **a**
- Take next char i.e. 'b', add it to a list and fix it ----> **ab**
- Keep taking next chars until string ends ---->**abc**
- Delete up to first char, start taking char from second
- position from first char till string ends ----> **ac**
- Similarly, go for rest chars ---->**b, bc, c**
- If we have a string of N length. Then the number of its non-empty subsequences is (**2^n-1**). In the end, we return the list of subsequences.

Note: The code written from the above insights can be accessed in the solution tab in the question itself.

Problem Statement: Print All Subsequences

Given a string (let's say of length n), print all the subsequences of the given string.
Subsequences contain all the strings of length varying from 0 to n. But the order of characters should remain the same as in the input string.

Approach: This problem is fairly simple and similar in approach to the above problem, with the only change being that instead of adding the subsequences to a list, we directly print them.

Problem Statement: Print Keypad Combinations

Given an integer n, using the phone keypad find out and print all the possible strings that can be made using digits of input n.
Note: The order of strings is not important. Just print different strings in new lines.

The phone keypad looks like this:

Approach:
It can be observed that each digit can represent 3 to 4 different alphabets (apart from 0 and 1). So the idea is to form a recursive function. You can follow the given steps:

- Map the digit with its string of probable alphabets, i.e 2 with "abc", 3 with "def" etc. (i.e. create a database of this mapping)

- Create a recursive function that takes the following parameters: output string, number array, current index, and length of number array.
- **Base Case:** If the current index is equal to the length of the number array then print the output string.
- Extract the string at **digit[current_index]** from the Map, where the digit is the input number array.
- Run a loop to traverse the string from **start** to **end.**
- For every index again call the recursive function with the output string concatenated with the **i^{th}** character of the string and the **current_index + 1**.

Note: The code written from the above insights can be accessed in the solution tab in the question itself.