An Internship report

Submitted in Partial Fulfillment of the Requirement for the Award of the Degree of

BACHELOR OF TECHNOLOGY

COMPUTER SCIENCE AND ENGINEERING

To

Dr. A.P.J. ABDUL KALAM TECHNICAL UNIVERSITY LUCKNOW

SUBMITTED BY

Harsh Gupta

University Roll No. 1903420100054

UNDER THE SUPERVISION OF

Mr. Dhananjay sir

Assistant Professor

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

UNITED COLLEGE OF ENGINEERING AND RESEARCH, PRAYAGRAJ

DECEMBER 2022

CANDIDATE'S DECLARATION

I, **Harsh Gupta**(1903420100054), students of B.Tech of Computer Science and Engineering hereby declared that I own the full responsibility for the information, results etc provided in this Internship titled **"Competitive Programming"** submitted to Dr. A.P.J Abdul Kalam Technical University, Lucknow for award of B.Tech (Computer Science and Engineering) degree. I have taken care in all respect to honour the intellectual property right and have acknowledged the contributions of others for using them in this academic purpose. I further declared that in case of any violation of intellectual property right or copyright, I as the candidate would be fully responsible for the same. My supervisor and institute should not be held for full or partial violation of copy right if found at any stage of our degree.

Date: Harsh Gupta

Place: (1903420100054)

CERTIFICATE

This is to certify that the mini project work entitled **“Competitive Programming”,** submitted by Harsh Gupta (1903420100054) to the Dr. A.P.J. Abdul Kalam Technical University, Lucknow, for the partial fulfilment of the requirement for the award of **Bachelor of Technology (Computer Science and Engineering)** degree, is a record of student’s own study carried out under my supervision and guidance.

This mini project has not been submitted to any other university or institution for the award of any other degree.

<u>SUPERVISOR</u>

(Mr. Dhananjay Sir)

ACKNOWLEDGEMENT

I, Harsh Gupta(1903420100054) am grateful to the management of United College of Engineering & Research for providing me an opportunity to undertake my internship. I am grateful and thankful to **Mr. Dhananjay sir** and all the other senior staff of the college, as they have helped me in every way possible. They put me under good supervision which helped in learning a lot of new things about the project and its various applications. They also provided me all the necessary information needed. I also take the opportunity to offer our sincere thanks and deep sense of gratitude to **Mr. Dhananjay sir** for attending us throughout the course of this project. I must make special mention of our **H.O.D. Dr. Vijay Kumar Dwivedi**, for providing me a platform to complete my mini project successfully. I would thank all the lab maintenance staff for providing assistance in various H/W & S/W problem encountered during course of my project. I am also very thankful to respected principal sir who gave me an opportunity to present this project.

INDEX

Language Tools for Python

List

A list is a collection that is ordered and changeable. In Python, lists are written with square brackets.

list = ["Apple" , "Banana" , "Cherry"]

- **You access the list items by referring to the index number** print (list[1])
  // Start with 0th index so Output is Banana

- **To change the value of a specific item, refer to the index number** list[1] = "Orange"

- **You can loop through the list items by using a for loop**
  ```
  for x in list:
          print (x) // Output is Apple , Orange , Cherry
  ```

- **To determine if a specified item is present in a list use the in keyword** if "Apple" in list:
  ```
          print (“Yes”)     // Yes if Apple is present in list else:
          print (“No”)              // No if it is not Present
  ```

- **To determine how many items a list have, use the len() method** print (len (list))
  // Output is 3 as contains 3 elements

- **To add an item to the end of the list, use the append() method** list.append("Mango") // Append at the end of list

- **To add an item at the specified index, use the insert() method** list.insert(1, "Mango") // insert at index 1 of list

- **The remove() method removes the specified item**
  ```
  list.remove( "Banana" )                  // Remove the element Banana if present
  ```

- **The pop() method removes the specified index, (or the last item if index is not specified)**
  ```
  list.pop()
  ```

- **The del keyword removes the specified index**
  ```
  del list[ 0 ]                // removes the specified index
  ```

Tuple

A tuple is a collection that is ordered and unchangeable. In Python, tuples are written with round brackets.

tuple = ("Apple" , "Banana" , "Cherry")

- **You can access tuple items by referring to the index number, inside square brackets:**

  print (tuple[1]) // Output is Banana the specified index

- **Once a tuple is created, you cannot change its values. Tuples are unchangeable .** tuple[1] = "Orange" // Gives error the value remain unchanged

- **You can loop through the tuple items by using a for a loop.** for x in a tuple:

  print (x) // Generate all element present in tuple

- **To determine if a specified item is present in a tuple use the in keyword:** if "Apple" in a tuple:

  print ("Yes") // Output is Yes if Apple is present in tuple

- **To determine how many items a list have, use the len() method** print (len (tuple)) // Output is 3 as 3 element are in tuple

- **Tuples are unchangeable, so you cannot add or remove items from it, but you can delete the tuple completely:**

- **Python has two built-in methods that you can use on tuples.**
  - **count()** Returns the number of times a specified value occurs in a tuple
  - **index()** Searches the tuple for a specified value and returns the position of where it was found

Set

A set is a collection which is unordered and unindexed. In Python sets are written with curly brackets.

set = { "apple" , "banana" , "cherry" }

- **You cannot access items in a set by referring to an index, since sets are unordered the items has no index. But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.** for x in set:

  print (x) // Output contains all element present in set
- **Once a set is created, you cannot change its items, but you can add new items. To add one item to a set use the add() method.**

  set.add("Orange") // Add one element at end

To add more than one item to a set use the update() method. set.update(["Orange" , "Mango" , "Grapes"]) // Add all element in the end

- **To determine how many items a set have, use the len() method.**

  print (len (set)) // output is length of set

- **To remove an item in a set, use the remove() , or the discard() method.** s et.remove("Banana") //Remove element if present else raise error set.discard("Banana") // Remove element if present else don’t raise error

- **Remove last element by using pop() method:** x = set.pop() //Remove and Return last element from the set print (x) // print the last element of set

Dictionary

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and Values. dict = {

```
        "brand" : "Ford" ,
        "model" : "Mustang" ,
        "year" : 1964
    }
```

- **You can access the items of a dictionary by referring to its key name, inside square brackets:**

  x = dict["model"] // Return the value of the key

- **You can change the value of a specific item by referring to its key name:** dict["year"] = 2018

- **You can loop through a dictionary by using a for loop. When looping through a dictionary, the return value are the keys of the dictionary, but there are methods to return the values as well.**

```
for x in dict:
            print (x)                // Print all key names in the dictionary
for x in dict:
            print (dict[x])          // Print all values of the dictionary
for x, y in dict.items():
            print (x, y)             // Print both keys and value of the dictionary
```

- **Adding an item to the dictionary is done by using a new index key and assigning a value to it:** dict["color"] = "red"

  print (dict) // Add new key and value to dictionary

- **The pop() method removes the item with specified key name:** dict.pop("model") // Removes model key/value pair in dictionary

Complexity Analysis

What is Running Time Analysis?

It is the process of determining how processing time of a problem increases as the size of the problem (input size) increases. Input size is the number of elements in the input, and depending on the problem type, the input may be of different types. Example:

- Size of an array
- Polynomial degree
- Number of elements in matrix
- Number of bits in the binary representation of the input
- Vertices and edges in the graph

What is the Rate of Growth?

The rate at which running time increases as a function of input is called rate of growth.

Commonly used rate of growths

Time Complexity	Name
1	Constant
logn	Logarithmic
n	Linear
nlogn	Linear Logarithmic
n^2	Quadratic
n^3	Cubic
2^n	Exponential
n!	Factorial

Types of analysis

To analyse the given algorithm, we need to know with which inputs does the algorithm take less time and with which inputs the algorithm takes a long time. We have already seen that an algorithm can be represented in the form of an expression. That means we represent the algorithm with multiple expressions:one for the case where it takes less time and another for the case where it takes more 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.

What is Space Complexity?

Space complexity is a measure of the total amount of memory including the space of input values with respect to the input size, that an algorithm needs to run and produce the result.

Auxiliary Space and Space complexity

- **Auxiliary Space:** It is the temporary or extra space used by an algorithm apart from the input size in order to solve a problem.
- **Space complexity:** It is the total space used by an algorithm in order to solve a problem including the input size. It includes both auxiliary space and space taken by the input size.

Space complexity = Auxiliary Space + Space taken by the input size

Calculating Space Complexity

The calculation of space complexity is necessary for determining the algorithm's efficiency. However, the space complexity also depends on the programming language, the compiler used, and the machine on which it is executed.

- Let us consider a program for calculating the sum of two numbers:

```
// Function to print the sum of two integers
function SumOfTwoIntegers()
  // Reading integers num1 and num2
  read(num1);
  read(num2);

  // Calculating the sum of two integers
  sum = num1 + num2;
  print("The sum of two integers");
  print(sum);
```

We have declared three variables 'num1', 'num2',' 'sum', considering them of data type 'int' let the space occupied by 'int' data type be 4 bytes, hence the total space consumed is 4*3 = 12 bytes.

Hence we can say that the space complexity of the above program is O(1) as 12 is a constant.

The trade-off between Time and Space Complexity

The best algorithm to solve a particular problem is no doubt the one that requires less memory space and takes less time to execute. However, designing such an algorithm is not a trivial task, there can be more than one algorithm to solve a given problem one may require less memory space while the other may require less time to solve the problem.

So, it is quite common to observe a tradeoff between time and space consumed while designing an algorithm, where one needs to be sacrificed for the other. So, if space is a constraint, one might choose an algorithm that takes less space at the cost of more CPU time and vice-versa. Hence, we must choose an algorithm according to the requirements and the environment in which it needs to be executed.

Sorting

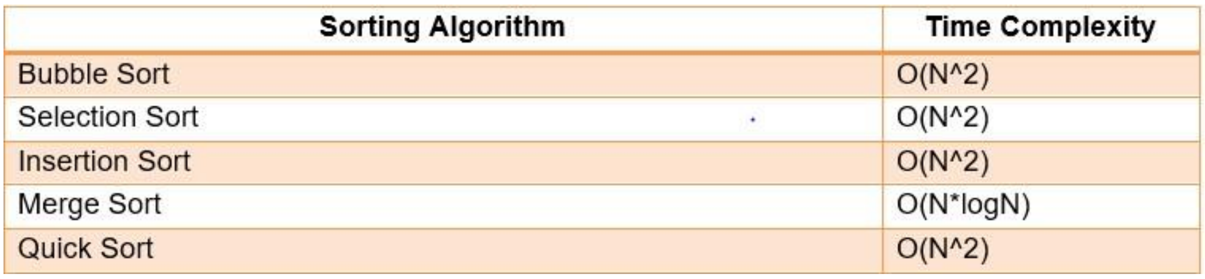
Introduction to Sorting

Sorting means an arrangement of elements in an ordered sequence either in increasing(ascending) order or decreasing(descending) order. Sorting is very important and many software and programs use this. The major difference is the amount of space and time they consume while being performed in the program.

Different languages have their own in-built functions for sorting, which are basically hybrid sorts that are a combination of different basic sorting algorithms. For example, C++ uses introsort, which runs quicksort and switches to heapsort when the recursion gets too deep. This way, you get the fast performance of quicksort in practice while guaranteeing a worst-case time complexity of O(N*logN), where N is the number of elements.

The time complexity of Sorting Algorithms:

The time complexities of various sorting algorithms in the worst case, where N represents the number of elements in the array -



Sorting Algorithm	Time Complexity
Bubble Sort	O(N^2)
Selection Sort	O(N^2)
Insertion Sort	O(N^2)
Merge Sort	O(N*logN)
Quick Sort	O(N^2)

In-place sorting and Out place sorting

1. **In-place sorting:** In-place sorting does not require any extra space except the input space excluding the constant space which is used for variables or iterators. It also doesn't include the space used for stack in the recursive algorithms.
   For Example, Bubble Sort, Selection Sort, Insertion Sort, Quicksort, etc.
2. **Out-place sorting:** Out-place sorting requires extra space to sort the elements.
   For Example, Merge Sort, Counting sort, etc.

Stable and Unstable sorting

1. **Stable sorting:** A sorting algorithm when two objects with equal keys appear in the same order in the sorted output as they appear in the unsorted input.
   For Example, Insertion Sort, Merge Sort, Bubble Sort, Counting sort, etc.
2. **Unstable sorting:** A sorting algorithm is called unstable sorting if there are two or more objects with equal keys which don't appear in the same order before and after sorting.
   For Example, Quick Sort, Heap Sort, Selection Sort, etc.

Bubble Sort

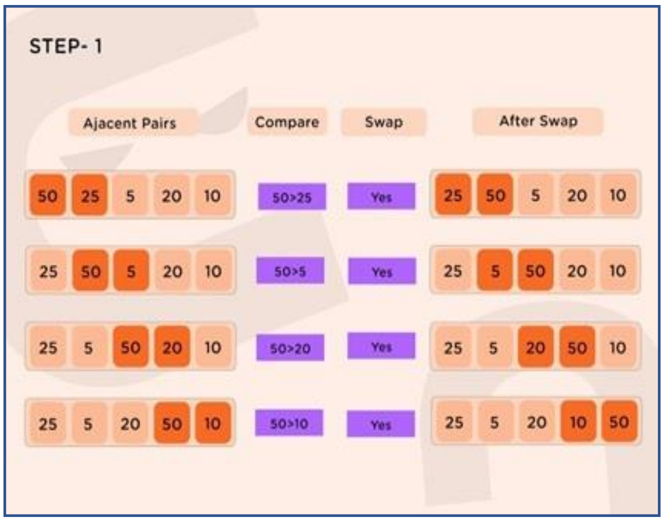
In selection sort, the elements from the start get placed at the correct position first and then the further elements, but in the bubble sort, the elements start to place correctly from the end.

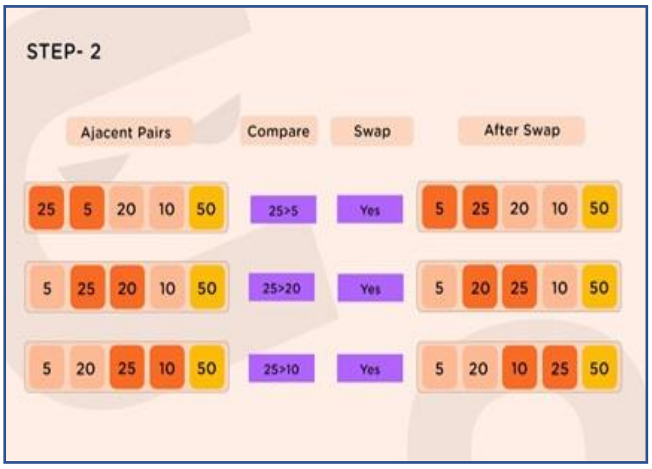
In this technique, we just compare the two adjacent elements of the array and then sort them manually by swapping if not sorted. Similarly, we will compare the next two elements (one from the previous position and the corresponding next) of the array and sort them manually. This way the elements from the last get placed in their correct position. This is the difference between selection sort and bubble sort.

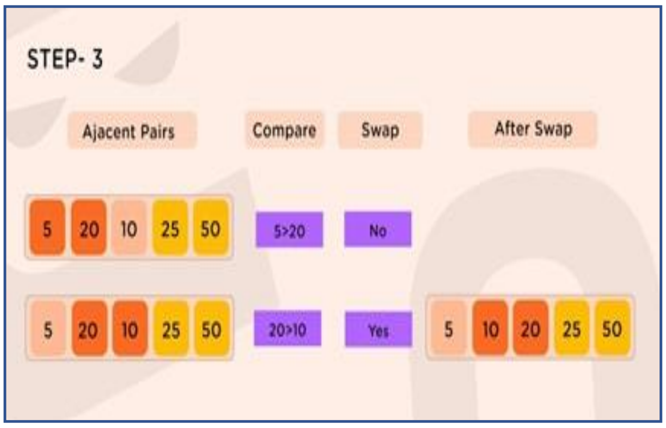
Consider the following depicted array as an example. You want to sort this array in increasing order.

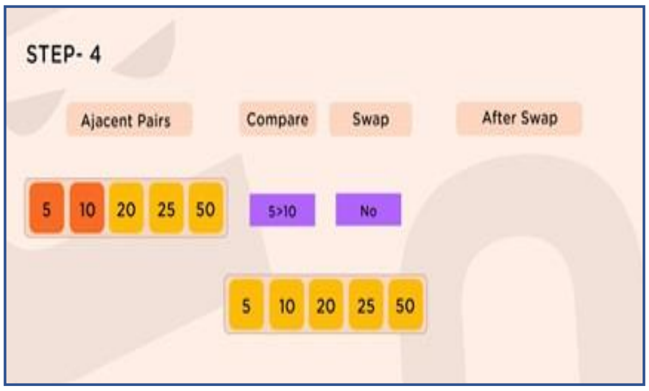
50	**25**	**5**	**20**	**10**

Following is the pictorial diagram for a better explanation of how it works:









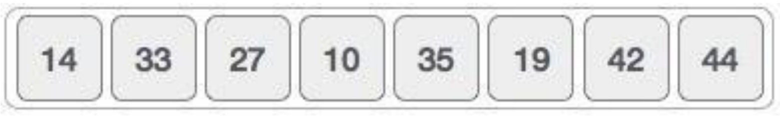
Selection Sort

Steps: (sorting in increasing order)

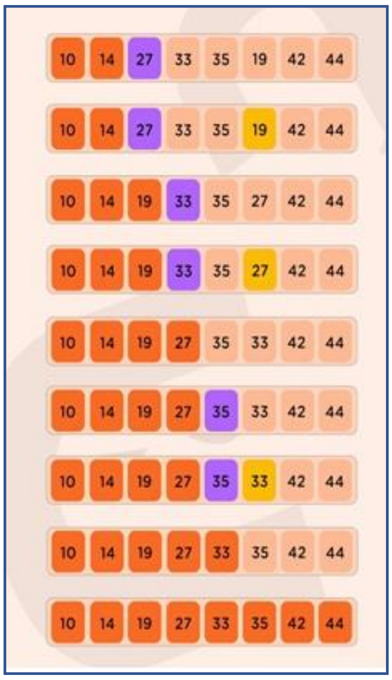
1. First-of-all, we will find the smallest element of the array and swap it with index 0.
2. Similarly, we will find the second smallest and swap that with the element at index 1 and so on…
3. Ultimately, we will be getting a sorted array in increasing order only.

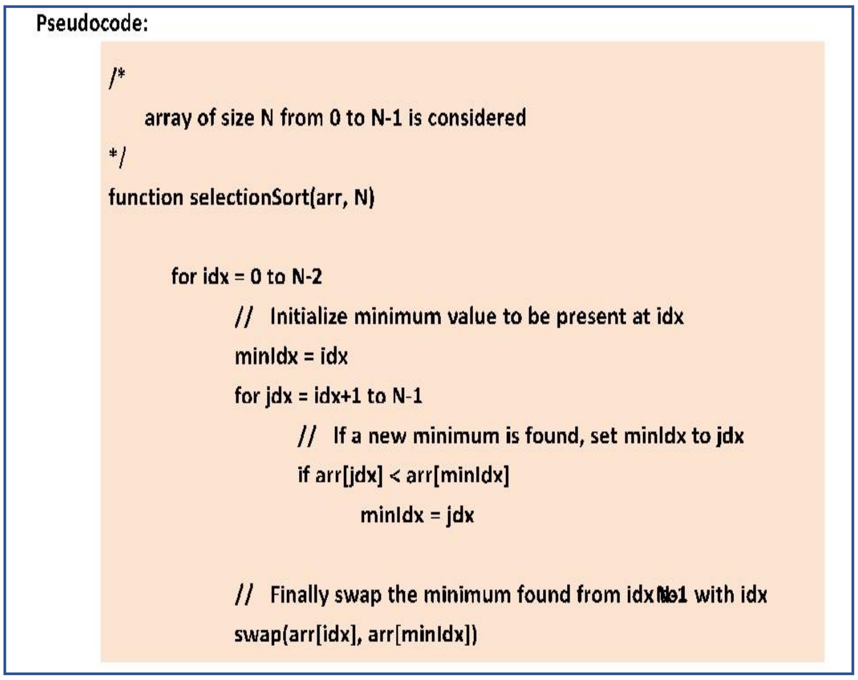
Let us look at the example for better understanding:

Consider the following depicted array as an example. You want to sort this array in increasing order.



Following is the pictorial diagram for a better explanation of how it works:





```
Pseudocode:
    /*
        array of size N from 0 to N-1 is considered
    */
    function selectionSort(arr, N)

        for idx = 0 to N-2
            //  Initialize minimum value to be present at idx
            minIdx = idx
            for jdx = idx+1 to N-1
                //  If a new minimum is found, set minIdx to jdx
                if arr[jdx] < arr[minIdx]
                    minIdx = jdx

            //  Finally swap the minimum found from idx to N-1 with idx
            swap(arr[idx], arr[minIdx])
```

Time complexity: O(N^2), in the worst case.

As to find the minimum element from the array of 'N' elements, we require 'N-1' comparisons, then after putting the minimum element in the correct position, we repeat the same for the unsorted array of the remaining 'N-1' elements, performing 'N-2' comparisons and so on.

So, total number of comparisons : N-1 + N-2 + N-3 + … + 1 = (N*(N-1))/2 and total number of exchanges(swapping): N-1

So, time complexity becomes O(N^2).

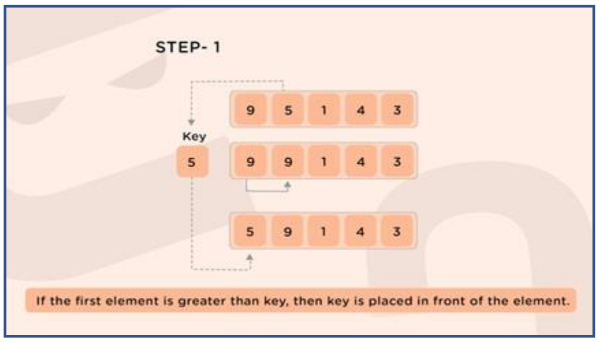
Space complexity: O(1), as no extra space is required.

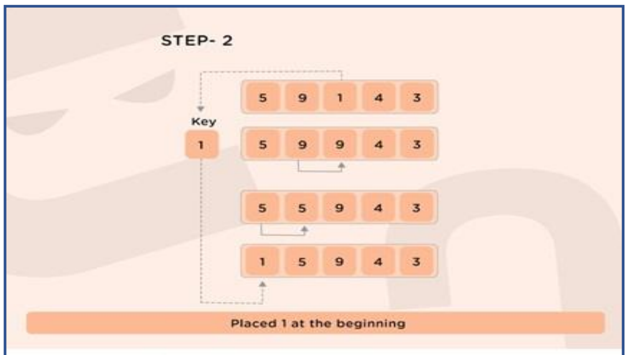
Insertion Sort

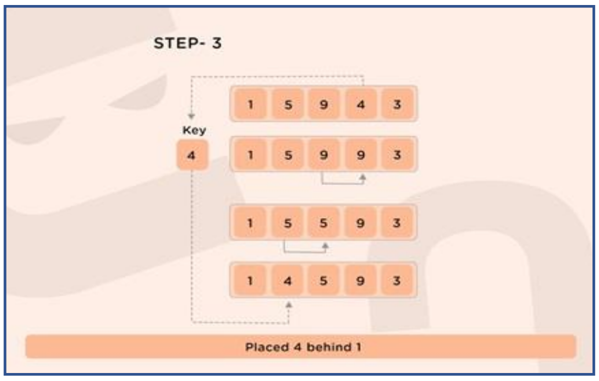
Insertion Sort works similar to how we sort a hand of playing cards.
Imagine that you are playing a card game. You're holding the cards in your hand, and these cards are sorted. The dealer hands you exactly one new card. You have to put it into the correct place so that the cards you're holding are still sorted. Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

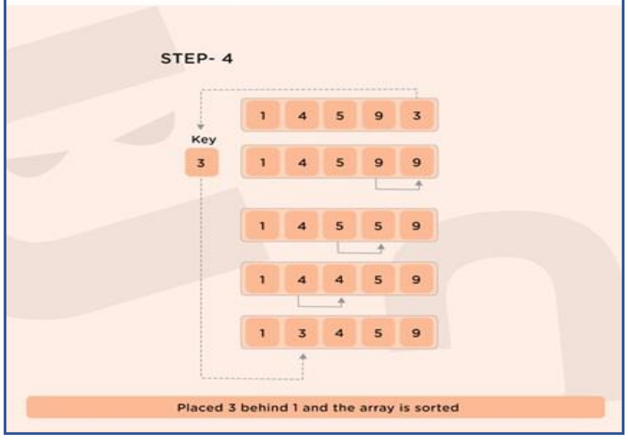
Consider the following depicted array as an example. You want to sort this array in increasing order.

9	5	4	1	3


STEP- 1
9 5 1 4 3
Key
5
9 9 1 4 3
5 9 1 4 3
If the first element is greater than key, then key is placed in front of the element.







Pseudocode:

```
jdx = idx-1 while arr[jdx] > cur and jdx >=0

//  Creating space for the element at idx arr[jdx+1] = arr[jdx] jdx -= 1

//  Placing the element at idx, making the array sorted from 0 to

idx arr[jdx+1] = cur
```

Time complexity: O(N^2), in the worst case. As for finding the correct position for the ith element, we need i iterations from 0 to i-1th index, so the total number of comparisons = 1+2+3+ … + N-1 = (N*(N-1))/2 and the total number of exchanges in the worst case: N-1. So, the overall complexity becomes O(N^2).

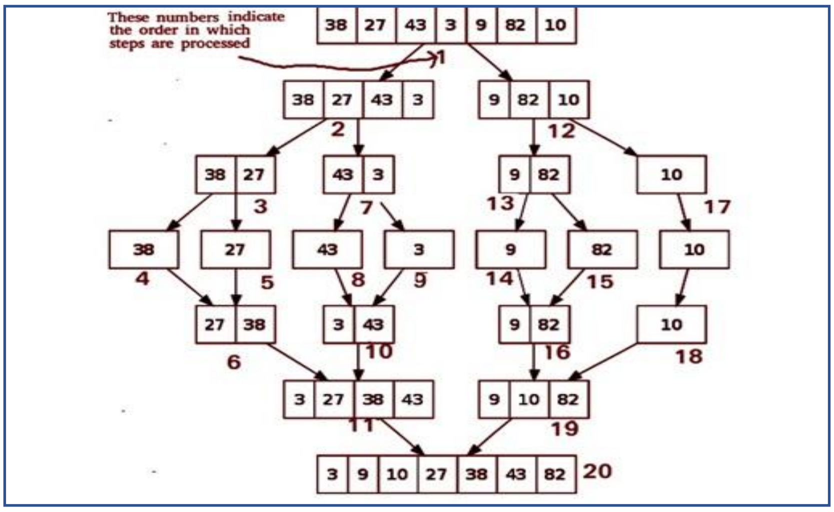
Space complexity: O(1), as no extra space is required.

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/merges the smaller sorted lists keeping the new list sorted too.

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

It has just one disadvantage that it creates a copy of the array and then works on that copy.



Time complexity: O(N*logN), in the worst case.

N elements are divided recursively into 2 parts, this forms a tree with nodes as divided parts of the array representing the subproblems. The height of the tree will be $\log_2 N$ and at each level of the tree, the computation cost of all the subproblems will be N. At each level the merge operation will take O(N) time. So the overall complexity comes out to be O(N*logN).

Space complexity: O(N), as extra space is required to sort the array.

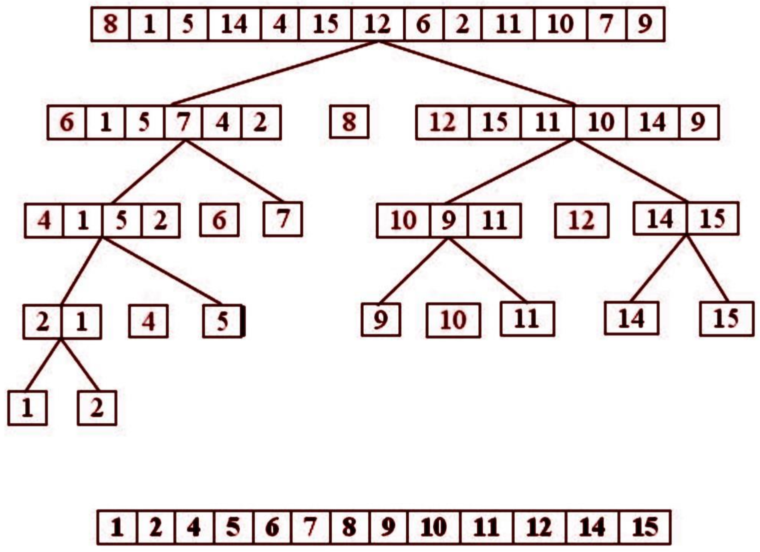
Quick Sort

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.

The following diagram shows the complete quick sort process, by considering the first element as the pivot element, for an example 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.

Time complexity: O(N^2), in the worst case.

But as this is a randomized algorithm, its time complexity fluctuates between $O(N^2)$ and O(N*log N) and mostly it comes out to be O(N*logN).

Space complexity: O(1), as no extra space is required.

Recursion and backtracking

Introduction to Recursion

Any function which calls itself is called recursion. **A recursive method solves a problem by calling a copy of itself to work on a smaller problem.** Each time a function calls itself with a slightly simpler version of the original problem. This sequence of smaller problems must eventually converge on a base case.

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 (Smaller problem):** 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.
- **Self-work:** 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.

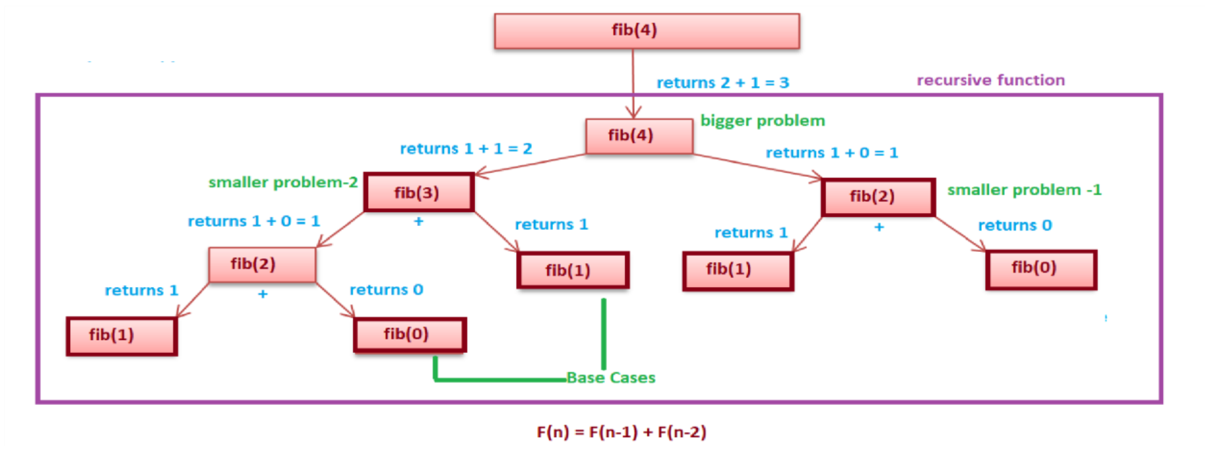
Note*: Recursion uses an in-built stack that stores recursive calls.

Problem Statement - Find nth fibonacci.

We know that Fibonacci numbers are defined as follows

fibo(n) = n for n <= 1 **fibo(n)** = fibo (n - 1) + fibo (n - 2)
otherwise

Let n = 4



As you can see from the above fig and recursive equation that the bigger problem is dependent on 2 smaller problems.

Depending upon the question, the bigger problem can depend on N number of smaller problems.

```
function fibonacci(n)

    //  base case if n equals 1
    OR 0 return n

    //  getting answer of the smaller problem recursionResult1 =
    fibonacci(n - 1) recursionResult2 = fibonacci(n - 2)

    // self work
ans = recursionResult1 + recursionResult2 return ans
```

Divide and Conquer

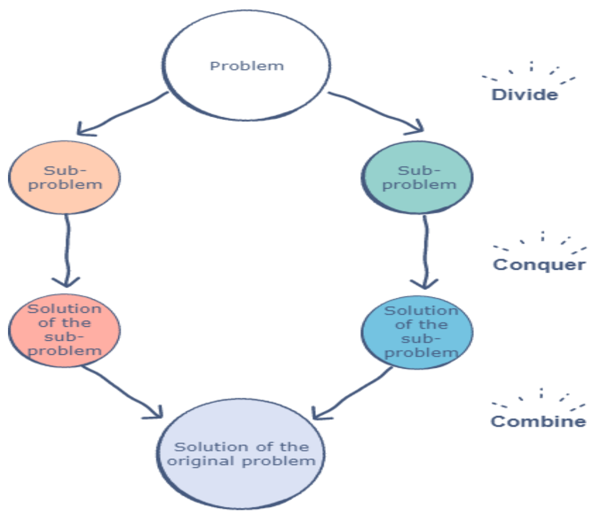
Divide and conquer is an important algorithm design technique based on recursion.

The divide and conquer strategy solves the problem by:

- **Divide:** breaking the problem into smaller subproblems that are themselves smaller instances of the same type of problem.
- **Conquer:** Conquer the subproblems by solving them recursively.
- **Combine:** Combine the solutions to the subproblems into the solution for the original given problem.

Divide and Conquer Visualization

Assume that n is the size of the original problem. As described above we can see that the problem is divided into subproblems with each of size n/b (for some constant b). We solve the subproblem recursively and combine the solutions to get the solution for the original problem.



Problem
Divide
Sub-problem
Sub-problem
Conquer
Solution of the sub-problem
Solution of the sub-problem
Combine
Solution of the original problem

```
DivideAndConquer( P ){
        if(small (P))
                //   P is very small so that the solution is obvious
                return solution( n );

        Divide the problem P into k subproblems P1, P2, Pk,
        return (
                Combine (
                        DivideAndConquer( P1 ),
                        DivideAndConquer( Pk),
```

Introduction to Backtracking

Backtracking is a famous algorithmic-technique for solving/resolving problems recursively by trying to build a solution incrementally. Solving one piece at a time, and removing those solutions that fail to satisfy the constraints of the problem at any point of time (by time, here, is referred to as the time elapsed till reaching any level of the search tree) is the process of backtracking.

In other words, **Backtracking** can be termed as a general algorithmic technique that considers searching **every possible combination** in order to solve a computational problem.

There are generally three types of problems in backtracking -

- **Decision Problems:** In these types of problems, we search for **any feasible** solution.
- **Optimization Problems:** In these types of problems, we search for the **best** possible solution.
- **Enumerations Problems:** In these types of problems, we find **all feasible** solutions.

Backtracking and recursion

Backtracking is based on recursion, where the algorithm makes an effort to build a solution to a computational problem incrementally. Whenever the algorithm needs to choose between multiple possible alternatives, it simply tries **all possible options** for the solution recursively step-by-step.

Implementation

- **Rat Maze problem**

  Given a 2D-grid, for simplicity let us assume that the grid is a **square** matrix of size **N**, having some cells as free and some as blocked. Given source **S** and destination **D**, we need to find whether there exists a path from source to destination in the maze, by traversing through **free cells** only.

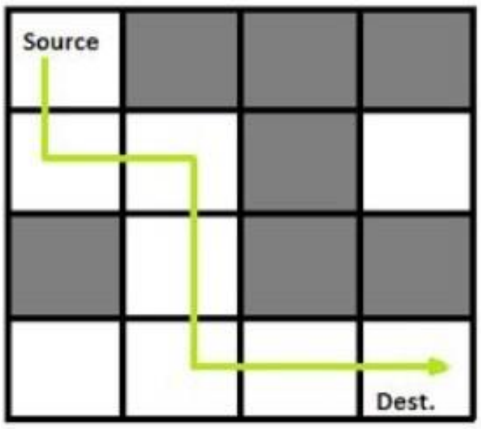
  Let us assume that source S is at the **top-left corner** of the maze and destination D is at the **bottom-right corner** of the maze, and the movements allowed are to either move to the **right** or to **down**.

  Steps:

  - If the destination point i.e **N** is reached, return true.
  - Else
    - Check if the current position is a **valid** position i.e the position is within the **bounds** of the maze as well as it is a **free** position.
    - Mark the current position as 1, denoting that the position can lead to a possible solution.
    - Move **forward**, and recursively check if this move leads to reach to the destination.
    - If the above move fails to reach the destination, then move **downward**, and recursively check if this move leads to reach the destination.
    - If none of the above moves leads to the destination, then mark the current position as 0, denoting that it is **not possible** to reach the destination from this position.

  Pseudocode:

```
function isValid(x, y, N)

        /*
                Check if the position is within the bounds of the maze and the position does not
                contain a blocked cell.
        */

        if(x <= N and y <= N and x, y is not blocked) return
        true else return false

function RatMaze(maze[][], x, y, N)

        /* x, y is the current position of the rat in the maze.
                Check if the current position is a valid position.
        */

        if isValid(x, y, N)
        mark[x][y] = 1 else
        return false

        /*
                If the current position is the bottom-right corner, i.e N, then we have found a
                solution
        */

        if x equals N and y equals N
                mark[x][y] = 1 return
                true

        /*
                Otherwise, try moving forward or downward to look for other possible
                solutions.
        */ bool found = RatMaze(maze,x+1,y,N) OR RatMaze(maze,x,y+1,N) /*
        If a solution is found from the current position by moving forward or downward, then
        return true, otherwise mark the current position as 0, as it is not possible to reach the
        end of the maze.
*/

if(found) return true else
mark[x][y] = 0 return false
```



Steps:

- If the destination point i.e **N** is reached, return true.
- Else
    - Check if the current position is a **valid** position i.e the position is within the **bounds** of the maze as well as it is a **free** position.
    - Mark the current position as 1, denoting that the position can lead to a possible solution.
    - Move **forward**, and recursively check if this move leads to reach to the destination.
    - If the above move fails to reach the destination, then move **downward**, and recursively check if this move leads to reach the destination.
    - If none of the above moves leads to the destination, then mark the current position as 0, denoting that it is **not possible** to reach the destination from this position.

Greedy Algorithm

Introduction

A greedy algorithm is a simple, intuitive algorithm that is used in optimization problems. Greedy algorithms work in stages. In each stage, a decision is made that is good at that point, without bothering about the future. This means that some local best is chosen. It assumes that a local good selection makes for a globally optimal solution.

Elements of the greedy algorithm

The basic properties of greedy algorithms are

- **Greedy choice property:** This property says that the globally optimal solution can be obtained by making a locally optimal solution. The choice made in a greedy algorithm may depend on earlier choices but never on future choices. It iteratively makes one greedy choice after another and reduces the given problem to a smaller one.
- **Optimal substructure:** A problem exhibits optimal substructure if the optimal solution to the problem contains optimal solutions to the subproblems. That means we can solve subproblems and build up the solution to solve the larger problems.

Example: **Fractional Knapsack Problem**: Given item t1, t2, …, tn (items we might want to carry in a backpack) with associated weights s1, s2, …, sn and benefit values v1, v2, …, vn, how can we maximize the total benefit considering we are subject to an absolute weight limit C?

Algorithm:

1. Compute value per size density for each item di = vi/si.
2. Sort each item by its value density.
3. Take as much as possible of the density item not already in the bag.

Time Complexity: O(nlogn) for sorting and O(n) for greedy selection.

Advantages of Greedy Algorithm

The main advantage of the greedy method is that it is straightforward, easy to understand, and easy to code. In greedy algorithms, once we make a decision, we do not have to spend time re-examining the already computed values.

Disadvantages of Greedy Algorithm

The main disadvantage is that for many problems there is no greedy algorithm. This means that in many cases there is no guarantee that making locally optimal improvements gives the optimal global solution. We also need proof of local optimality as to why a particular greedy solution will work.\

Applications of Greedy Algorithm

- Sorting: Selection sort, Topological sort
- Priority Queues: Heap Sort
- Huffman coding Compression algorithm
- Prim’s and Kruskal’s algorithm
- Shortest path in the weighted graph(Dijkstra’s)
- Coin change problem (for Indian currency)
- Fractional knapsack problem
- Disjoint sets: union by size or union by rank
- Job scheduling algorithm
- The greedy technique can be used as an approximation algorithm for complex problems.

Dynamic Programming

What is Dynamic Programming?

Dynamic Programming is a programming paradigm, where the idea is to memoize the already computed values instead of calculating them again and again in the recursive calls. This optimization can reduce our running time significantly and sometimes reduce it from exponential-time complexities to polynomial time.

Introduction

We know that Fibonacci numbers are defined as follows

fibo(n) = n for n <= 1 **fibo(n)** = fibo (n - 1) + fibo (n - 2) otherwise

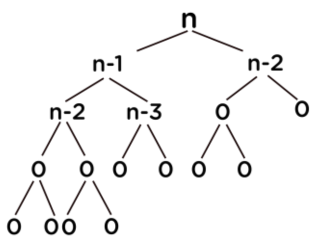
Suppose we need to find the nth Fibonacci number using recursion that we have already found out in our previous sections.

Pseudocode:

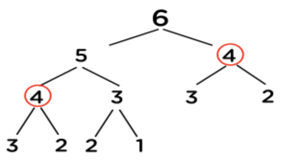
```
function fibo(n):
        if(n <= 1):
                return n
        return fibo(n-1) + fibo(n-2)
```

Time complexity: $O(2^n)$, in the worst case where n represents the nth fibonacci number.

- Here, for every n, we need to make a recursive call to **f(n-1)** and **f(n-2)**.



- We need to improve this complexity. Let's look at the example below for finding the 6th Fibonacci number.



Important Observations

- 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)).
- Generally, while recursing, there are repeated recursion calls, which increases the time complexity of the program.

Memoization

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.**

Pseudocode:

```
function fibo(n, dp)

        //   base case if n equals 0
        or n equals 1 return n

        //   checking if has been already calculated if
        dp[n] not  equals -1 return dp[n]


        //   final ans myAns = fibo(n - 1) +
        fibo(n - 2) dp[n] = myAns return
        myAns
```

Time complexity: O(n), where n represents the nth fibonacci number.

Bottom-up approach

We are first trying to figure out the dependency of the current value on the previous values and then using them to calculate 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.

Let us now look at the DP code for calculating the nth Fibonacci number:

Pseudocode:

```
function fibonacci(n):
        f = array[n+1]
        //  base case
        f[0] = 0 f[1] =
        1

        for i from 2 to n:
        //  calculating the f[i] based on the last two values f[i] =
               f[i-1] + f[i-2]

        return f[n]
```

Time complexity: O(n), where n represents the nth fibonacci number.

Space complexity: O(n), where n represents the nth fibonacci number.

Applications of Dynamic programming

- They are often used in machine learning algorithms, for eg Markov decision process in reinforcement learning.
- They are used for applications in interval scheduling.
- They are also used in various algorithmic problems and graph algorithms like Floyd warshall's algorithms for the shortest path, the sum of nodes in subtree, etc.

Bit Manipulation

Bit manipulation is basically the act of algorithmically manipulating bits (smallest unit of data in a computer).

A bit represents a logical state with one of two possible values '0' or '1', which in other representations can be referred to as 'true' or 'false' / 'on' or 'off'.

Since bits represent logical states efficiently, this makes the binary system suitable for electronic circuits that use logic gates and this is why binary is used in all modern computers.

Decimal and Binary number systems

- **Decimal number system:**

  We usually represent numbers in decimal notation(base 10) that provides ten unique digits - 0,1,2,3,4,5,6,7,8,9. To form any number, we can combine these digits to form a sequence such that each decimal digit represents a value multiplied by a power of 10 in this sequence.

  For example:

  244 = 200 + 40 + 4 = 2*10^2 + 4*10^1 + 4*10^0.

- **Binary number system:**

  The binary system works in a similar manner to that of the decimal number system, the only difference lies in the fact that binary notation(base 2) provides two digits - 0 and 1. A binary number is defined as a sequence of 1s and 0s like '010100', '101', '0','1111','000111', etc. In a binary number, each digit is referred to as a bit, where each such bit represents a power of decimal 2.

  Similar to decimal notation we can convert a given binary sequence to its decimal equivalent.

  For example:

  (base 2)1100: (Base 10) 1*2^3 + 1*2^2 + 0*2^1 + 0*2^0 = 8 + 4 + 0 + 0 = 12.

  A bit is said to be **set** if it is '1' and **unset** if the bit is '0'.

Bitwise Operators

Bitwise operators are used to performing bitwise operations. A bitwise operation operates on a bit string(string consisting of 0s and 1s only) or a bit array(array of 0s and 1s only) at the level of its individual bits. Bit operations are usually fast and are used for calculations and comparisons at the processor level.

1. **Bitwise AND(&)**

A **binary operator** that takes bit representation of its operands and the resulting bit is 1 if both the bits in bits patterns are 1 otherwise, the resulting bit is 0.

For example:

X = 5 $(101)_2$ and Y = 3 $(011)_2$

X & Y = $(101)_2$ & $(011)_2$ = $(001)_2$ = 1

2. **Bitwise OR(|)**

A **binary operator** that takes bit representation of its operands and the resulting bit is 0 if both the bits in bits patterns are 0 otherwise, the resulting bit is 1.

For example:

X = 5 $(101)_2$ and Y = 3 $(011)_2$

X | Y = $(101)_2$ | $(011)_2$ = $(111)_2$ = 7

3. **Bitwise NOT(~)**

A **unary operator** that takes bit representation of its operand and just flips the bits of the bit pattern, i.e if the ith bit is 0 then it will be changed to 1 and vice versa.

Bitwise NOT is the **1's complement** of a number.

For example:

X = 5 $(101)_2$

~X = $(010)_2$ = 2

4. **Bitwise XOR(^)**

A **binary operator** that takes bit representations of its operands and the resulting bit is 0 if the bits in bit patterns are the same i.e the bits in both the patterns at a given position is either 1 or 0 otherwise, the resulting bit is 1 if the bits in bit patterns are different.

For example:

X = 5 $(101)_2$ and Y = 3 $(011)_2$

X ^ Y = $(101)_2$ ^ $(011)_2$ = $(110)_2$ = 6

5. **Left shift operator(<<)**

A **binary operator** that left shifts the bits of the first operand, with the second operand deciding the number of places to shift left, and append that many 0s at the end. We discard the k left most bits.

For example:

X = 5 $(101)_2$

X << 2 = $(00101)_2$ << 4 = $(10100)_2$ = 20

In other words, left shift by k places is equivalent to multiplying the bit pattern by 2^k, as shown in the above example - 5 << 2 = 5 * 2^2 = 20.

So, A << x = A * 2^x that is why (1<< n) is equal to power(2,n) or 2^n.

6. **Right shift operator(>>)**

A **binary operator** that right shifts the bits of the first operand, with the second operand deciding the number of places to shift right. We discard the k rightmost bits.

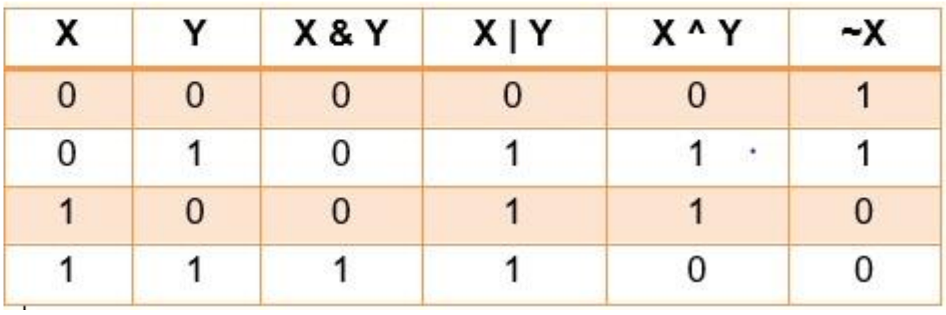
For example:

X = 5 $(101)_2$

X >> 2 = $(101)_2$ = $(001)_2$ = 1

In other words, the right shift by k places is equivalent to dividing the bit pattern by 2^k (integer division), as shown in the above example - 5 >> 2 = 5 / (2^2) = 1.

So, A >> x = A / (2^x) (integer division).



X	Y	X & Y	X | Y	X ^ Y	~X
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0