#Greedy Algorithm Assume that you have an objective function that needs to be optimized (either maximized or minimized) at a given point. A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized. The Greedy algorithm has only one shot to compute the optimal solution so that it never goes back and reverses the decision.

##When to Use Greedy Algorithms Greedy Algorithms can help you find solutions to a lot of seemingly tough problems. The only problem with them is that you might come up with the correct solution but you might not be able to verify if its the correct one. All the greedy problems share a common property that a local optima can eventually lead to a global minima without reconsidering the set of choices already considered.

###Sample Problems:

1. Lecture Scheduling Problem
2. Student Enrollment Problem

#Brute Force Algorithm Brute Force Algorithms are exactly what they sound like – straightforward methods of solving a problem that rely on sheer computing power and trying every possibility rather than advanced techniques to improve efficiency. For example, imagine you have a small padlock with 4 digits, each from 0-9.

###Sample Problems:

- Solve the same given problems using Brute Force algorithm.

#Dynamic Programming Dynamic Programming(DP) is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to the subproblems.

###Top-Down Approach with Memoization Whenever we solve a subproblem, we cache its result so that we don't end up solving it repeatedly if it's called multiple times. This technique of storing the results of the previously solved subproblems is called Memoization.

In memoization, we solve the bigger problem by recursively finding the solution to the subproblems. It is a top-down approach.

```
def fm(n, memo={}): # Declare function with parameters: Number N and
Dictionary Memo.
    if n <= 0: # Addition
        return str("[INVALID] ||| 0 or Negative integer is not
acceptable.")
    if n == 1: # If n equals 1, return 0
        return 0
    if n == 2: # If n equals 2, return 1
        return 1

    if n in memo:
        return memo[n]

    memo[n] = fm(n -1, memo) + fm(n -2, memo) # If current element is
```

```
not in memo, add to memoy by recursive call for previous function and
add. "
    return memo[n]

num = int(input("Enter a integer: "))
print(f"The {num} in fibonacci is {fm(num)}.")
```

###Bottom-Up Approach with Tabulation Tabulation is the opposite of the top-down approach and does not involve recursion. In this approach, we solve the problem “bottom-up”. This means that the subproblems are solved first and are then combined to form the solution to the original problem.

This is achieved by filling up a table. Based on the results of the table, the solution to the original problem is computed.

```
def ft(n): # Declare the function and take the number whose Fibonacci
Series is to be printed
  if n == 0   : return [0]
  elif n == 1 : return [0, 1]

  fib = [0, 1] # Initialize the list and input the values 0 and 1 in
it.

  for i in range(2, n + 1): # Iterate over the range of 2 to n+1.
    fib.append(fib[i - 1] + fib[i - 2]) # Append the list with the sum
of the previous two values of the list.


  return fib

num = int(input("Enter integer: "))
print(f"The {num} in fibonacci is {ft(num-1)}.") # Return the list as
output [n - 1 for index 0]
```

Through this activity, I learned About these algorithm in which helped me understand the application of these. I realized that these methods can help me solve problems in a more constructive and structured way.