Plotting Using PyLab

➢ Plotting Using PyLab :

PyLab is Python library that gives facilities of **MATLAB**. MATLAB is high-level technical computing language and interactive environment for algorithm development, data visualization, data analysis and numeric computation.

Here is a simple example that uses **pylab.plot** to generate plot.

Example

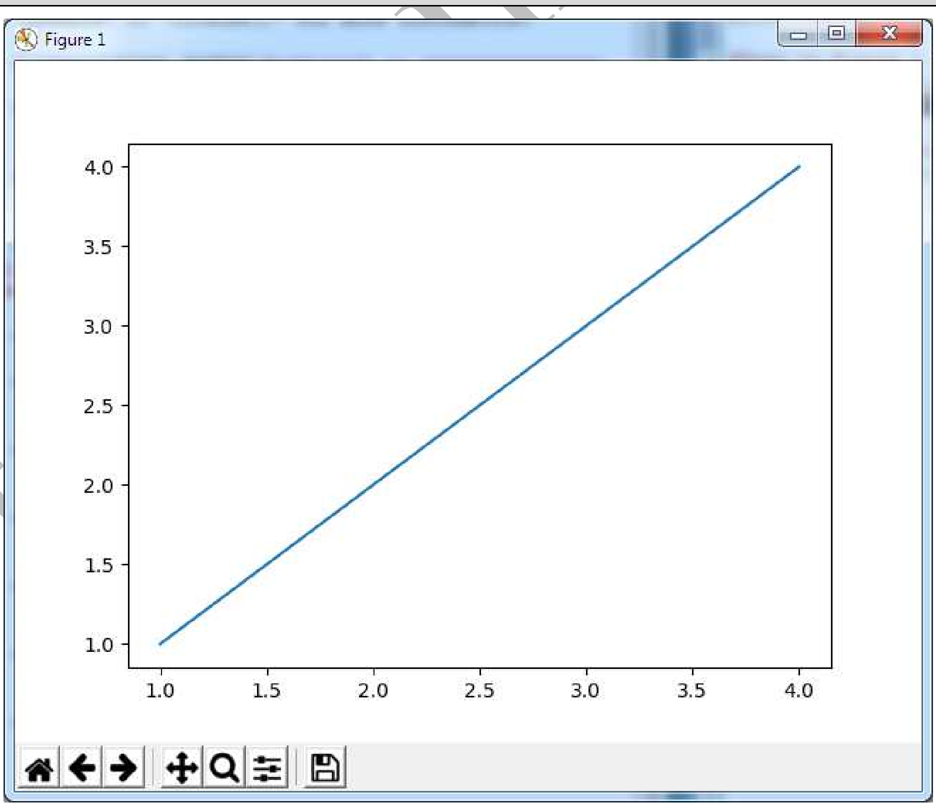
```
import pylab

#Create Figure 1
pylab.figure(1)

#Draw on Figure 1
pylab.plot([1,2,3,4],[1,2,3,4])

#Show Figure on Screen
pylab.show()
```

Out Put



The above chart is the plot generated by the invocation of **pylab.plot**. The two parameters of **pylab.plot** must be sequence of the same length. The first gives the **x-coordinates** of the points to be plotted and the second the **y-coordinates**. Combination of these two(x,y) provides of FOUR <x , y> coordinated pairs **`[(1,1), (2,2), (3,3), (4,4)]`**.

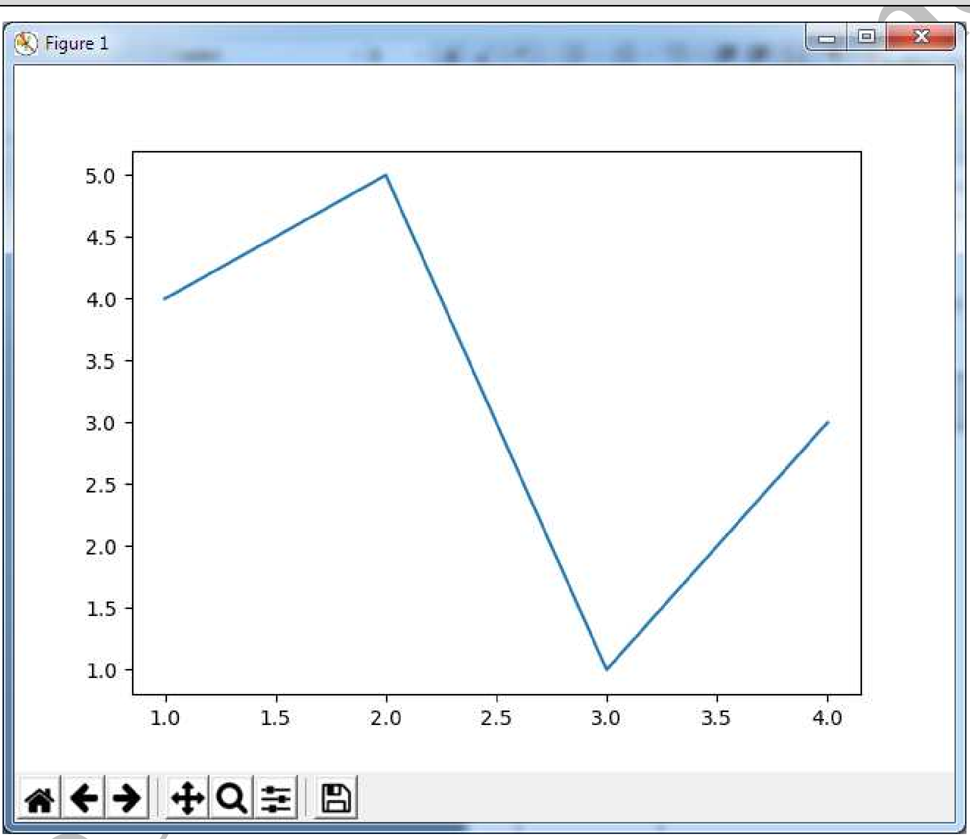
Example : 2

```
from matplotlib import pyplot as plt


#Plotting to Canvas
plt.plot([1,2,3,4],[4,5,1,3])


#Showing what we plotted
plt.show()
```

Out Put : 2

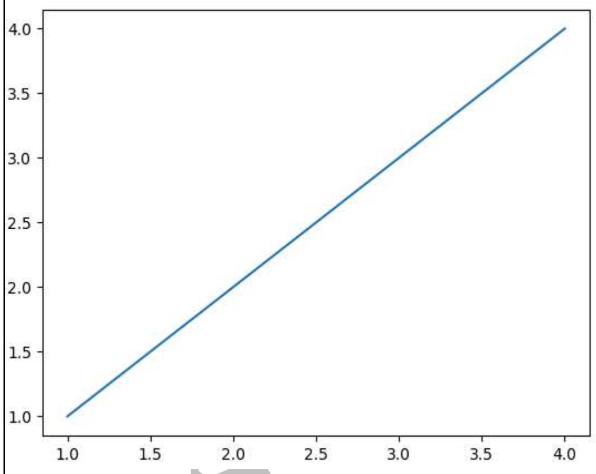


FOUR <x , y> coordinated pairs **`[(1,4), (2,5), (3,1), (4,3)]`**.

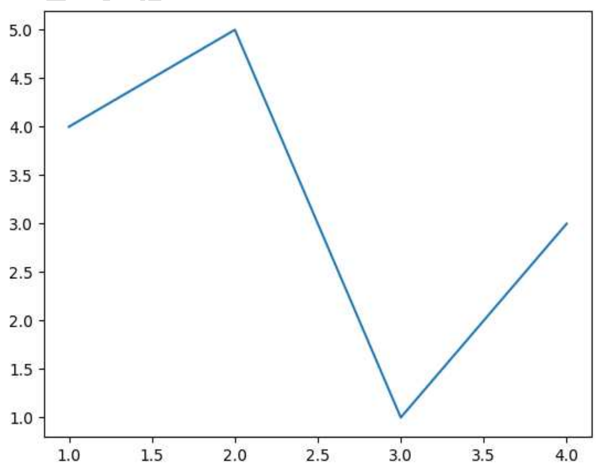
Example : 3

```
import pylab
#Create Figure 1
pylab.figure(1)
#Draw on Figure 1
pylab.plot([1,2,3,4],[1,2,3,4])
#Save Figure 1
pylab.savefig('myFig_01')
#Create Figure 2
pylab.figure(2)
#Draw on Figure 2
pylab.plot([1,2,3,4],[4,5,1,3])
#Save Figure 2
pylab.savefig('myFig_02.pdf')
#Show Figures on Screen
pylab.show()
```

Out Put : 3



myFig_01.png



myFig_02.pdf

Above code will generate two plots. First plot will be saved as "**myFig_01.png**" and Second plot will be saved as "**myFig_02.pdf**". Supported formats are : eps, pdf, pgf, png, ps, raw, rgba, svg, svgz.

➢ Plotting Mortgages :

What is Mortgage? : A mortgage is a debt instrument, secured by the guarantee of specified property, which the borrower is obliged to pay back with a fixed set of payments.
In Python, Mortgage follows same functionality. Means, one class or object will be debt (Mortgage) to get some functionality for different objects. Logically Mortgage is "**User Defined Class / Package / Module"** that is useful to get or set some other functionality.

Example

```
'''calculate the monthly payment on a mortgage loan'''
import math
def calc_mortgage(principal, interest, years):
    '''
    given mortgage loan principal, interest(%) and years to pay
    calculate and return monthly payment amount
    '''
    # monthly rate from annual percentage rate
    interest_rate = interest/(100 * 12)

    # total number of payments
    payment_num = years * 12

    # calculate monthly payment
    payment = principal * \
        (interest_rate/(1-math.pow((1+interest_rate), (-payment_num))))
    return payment

# mortgage loan principal
principal = 100000

# percent annual interest
interest = 7.5

# years to pay off mortgage
years = 30

# calculate monthly payment amount
monthly_payment = calc_mortgage(principal, interest, years)

# calculate total amount paid
total_amount = monthly_payment * years * 12

# show result ...
# {:,} uses the comma as a thousand separator
sf = '''
For a {} year mortgage loan of Rs.{:,}
at an annual interest rate of {:.2f}%
you pay Rs.{:.2f} monthly'''
print(sf.format(years, principal, interest, monthly_payment))
print('='*50)
print("Total amount paid will be Rs.{:,.2f}".format(total_amount))
```

Out Put

```
For a 30 year mortgage loan of Rs.100,000
at an annual interest rate of 7.50%
you pay Rs.699.21 monthly
==================================================
Total amount paid will be Rs.251,717.22
```

Above example is simple calculation for LOAN payments.

Now, Let’s Create our **Mortgage** Class and other **LoanIntrestType Class** to extend our Class.

Example : Mortgage base class

```
def findPayment(loan, r, m):
    """Assumes: loan and r are floats, m an int
    Returns the monthly payment for a mortgage of size
    loan at a monthly rate of r for m months"""
    return loan*((r*(1+r)**m)/((1+r)**m - 1))

class Mortgage(object):
      """Abstract class for building different kinds of mortgages"""

      def __init__(self, loan, annRate, months):
          """Create a new mortgage"""
          self.loan = loan
          self.rate = annRate/12.0
          self.months = months
          self.paid = [0.0]
          self.owed = [loan]
          self.payment = findPayment(loan, self.rate, months)
          self.legend = None #description of mortgage

      def makePayment(self):
          """Make a payment"""
          self.paid.append(self.payment)
          reduction = self.payment - self.owed[-1]*self.rate
          self.owed.append(self.owed[-1] - reduction)

      def getTotalPaid(self):
          """Return the total amount paid so far"""
          return sum(self.paid)
          def __str__(self):
          return self.legend
```

Looking at **__init__**, we see that all Mortgage instances will have instance variables corresponding to the initial loan amount, the monthly interest rate, the duration of the loan in months, a list of payments that have been made at the start of each month (the list starts with 0.0, since no payments have been made at the start of the first month), a list with the balance of the loan that is outstanding at the start of each month, the amount of money to be paid each month (initialized using the value returned by the function findPayment), and a description of the mortgage (which initially has a value of None). The __init__ operation of each subclass of Mortgage is expected to start by calling **Mortgage.__init__**, and then to initialize **self.legend** to an appropriate description of that subclass.

The method “**makePayment()**” is used to record mortgage payments. Part of each payment covers the amount of interest due on the outstanding loan balance, and the remainder of the payment is used to reduce the loan balance. That is why **“makePayment()”** updates both **self.paid** and **self.owed**.

The method “**getTotalPaid()”** uses the built-in Python function sum, which returns the sum of a sequence of numbers. If the sequence contains a non-number, an exception is raised.

Example : Fixed - Rate Mortgage classes

```
class Fixed(Mortgage):
      def __init__(self, loan, r, months):
          Mortgage.__init__(self, loan, r, months)
          self.legend = “Fixed, “ + str(r*100) + “%”

class FixedWithPts(Mortgage):
      def __init__(self, loan, r, months, pts):
          Mortgage.__init__(self, loan, r, months)
          self.pts = pts
          self.paid = [loan*(pts/100.0)]
          self.legend = “Fixed, “ + str(r*100) + “%, “\ + str(pts) + “ points”
```

Above Code contains classes implementing two types of mortgage.

Each of these classes overrides **`__init__`** and inherits the other three methods from Mortgage.

Example : Mortgage With Teaser rates

```
class TwoRate(Mortgage):

      def __init__(self, loan, r, months, teaserRate, teaserMonths):
          Mortgage.__init__(self, loan, teaserRate, months)
          self.teaserMonths = teaserMonths
          self.teaserRate = teaserRate
          self.nextRate = r/12.0
          self.legend = str(teaserRate*100)\
          + '% for ' + str(self.teaserMonths)\
          + ' months, then ' + str(r*100) + '%'

      def makePayment(self):
          if len(self.paid) == self.teaserMonths + 1:
             self.rate = self.nextRate
             self.payment = findPayment(self.owed[-1], self.rate,
             self.months - self.teaserMonths)
          Mortgage.makePayment(self)
```

Above Code contains a third subclass of Mortgage. The class **`TwoRate`** treats the mortgage as the concatenation of two loans, each at a different interest rate. (Since **`self.paid`** is initialized with a 0.0, it contains one more element than the number of payments that have been made. That’s why **`makePayment`** compares **`len(self.paid)`** to **`self.teaserMonths + 1`**).

Example : Calculate Mortgages

```
def compareMortgages(amt, years, fixedRate, pts, ptsRate,
                      varRate1, varRate2, varMonths):
    totMonths = years*12
    fixed1 = Fixed(amt, fixedRate, totMonths)
    fixed2 = FixedWithPts(amt, ptsRate, totMonths, pts)
    twoRate = TwoRate(amt, varRate2, totMonths, varRate1, varMonths)
    morts = [fixed1, fixed2, twoRate]

    for m in range(totMonths):
        for mort in morts:
            mort.makePayment()
    for m in morts:
        print m
        print “Total payments = Rs.” + str(int(m.getTotalPaid()))

compareMortgages(amt=200000, years=30, fixedRate=0.07,
pts = 3.25, ptsRate=0.05, varRate1=0.045,
        varRate2=0.095, varMonths=48)
```

Above Code contains a function that computes and prints the total cost of each kind of mortgage for a sample set of parameters. It begins by creating one mortgage of each kind. It then makes a monthly payment on each for a given number of years. Finally, it prints the total amount of the payments made for each loan.

Out Put

```
Fixed, 7.0%
Total payments = Rs. 479017

Fixed, 5.0%, 3.25 points
Total payments = Rs. 393011

4.5% for 48 months, then 9.5%
Total payments = Rs.551444
```

In Python, we can also add PLOTs to above examples to illustrate Graphical Mortgages.

➢ Fibonacci Sequence Revisited :

What is/are Fibonacci Numbers / Sequence? : It's a series of numbers in which each number (Fibonacci number) is the sum of the two preceding numbers.
The simplest is the series 1, 1, 2, 3, 5, 8, etc.

The mathematical equation describing it is $X_{n+2} = X_{n+1} + X_n$

Let's see Python Program to find the Fibonacci series using recursion. The program takes the number of terms and determines the Fibonacci series using recursion up to that term. 1. Take the number of terms from the user and store it in a variable.

Problem Solution

1. Take the number of terms from the user and store it in a variable.
2. Pass the number as an argument to a recursive function named Fibonacci.
3. Define the base condition as the number to be lesser than or equal to 1.
4. Otherwise call the function recursively with the argument as the number minus 1 added to the function called recursively with the argument as the number minus 2.
5. Use a for loop and print the returned value which is the Fibonacci series.
6. Exit.

Example

```
def fibonacci(n):
    if(n < 1):
        return 1
    else:
        return(fibonacci(n-1) + fibonacci(n-2))
n = int(input("Enter number of terms: "))
print("Fibonacci sequence:")
for i in range(n):
    print (fibonacci(i))
```

Out Put

```
Enter number of terms: 5
Fibonacci sequence:
1
2
3
5
8
```

Notice that we are computing the same values over and over again. For example **`Fibonacci()`** gets called with 5 Five times, and each of these calls provokes Six additional calls of **`Fibonacci()`**. It doesn't require a genius to think that it might be a good idea to record the value returned by the first call, and then look it up rather than compute it each time it is needed. This is called "**Memoization (to be remembered)**", and is the key idea behind dynamic programming.

➢ Dynamic programming and the 0/1 Knapsack algorithm :

What is Knapsack? : The **knapsack** or **rucksack** problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible.

Suppose we decide that an approximation is not good enough, i.e., we want the best possible solution to this problem. Such a solution is called optimal, not surprising since we are solving an optimization problem. As it happens, this is an instance of a classic optimization problem, called the **0/1 knapsack problem**.

The 0/1 knapsack problem can be formalized as follows:

1. Each item is represented by a pair, <value, weight>.

2. The knapsack can accommodate items with a total weight of no more than w.

3. A vector, I, of length n, represents the set of available items. Each element of the vector is an item.

4. A vector, V, of length n, is used to indicate whether or not each item is taken by the burglar. If V[i] = 1, item I[i] is taken. If V[i] = 0, item I[i] is not taken.

5. Find a V that maximizes

$$\sum_{i=0}^{n-1} \texttt{V[i]*I[i]value}$$

subject to the constraint that

$$\sum_{i=0}^{n-1} \texttt{V[i]*I[i].weight <= w}$$

Let's see what happens if we try to implement this formulation of the problem in a straightforward way:

1. Enumerate all possible combinations of items. That is to say, generate all subsets112 of the set of items. This is called the power set.
2. Remove all of the combinations whose weight exceeds the allowed weight.
3. From the remaining combinations choose any one whose value is the largest.

This approach will certainly find an optimal answer. However, if the original set of items is large, it will take a very long time to run, because the number of subsets grows exceedingly quickly with the number of items.

Example

```
class Item(object):
     def __init__(self, n, v, w):
          self.name = n
          self.value = float(v)
          self.weight = float(w)
     def getName(self):
         return self.name
     def getValue(self):
         return self.value
     def getWeight(self):
         return self.weight
     def __str__(self):
         result = '<' + self.name + ', ' + str(self.value)\
                  + ', ' + str(self.weight) + '>'
         return result
def value(item):
    return item.getValue()
def weightInverse(item):
    return 1.0/item.getWeight()
def density(item):
    return item.getValue()/item.getWeight()
def buildItems():
    names = ['clock', 'painting', 'radio', 'vase', 'book', 'computer']
    values = [175,90,20,50,10,200]
    weights = [10,9,4,2,1,20]
    Items = []
    for i in range(len(values)):
        Items.append(Item(names[i], values[i], weights[i]))
    return Items
def greedy(items, maxWeight, keyFunction):
    """Assumes Items a list, maxWeight >= 0,
         keyFunction maps elements of Items to floats"""
    itemsCopy = sorted(items, key=keyFunction, reverse = True)
    result = []
    totalValue = 0.0
    totalWeight = 0.0
    for i in range(len(itemsCopy)):
        if (totalWeight + itemsCopy[i].getWeight()) <= maxWeight:
            result.append(itemsCopy[i])
            totalWeight += itemsCopy[i].getWeight()
            totalValue += itemsCopy[i].getValue()
    return (result, totalValue)
def testGreedy(items, constraint, keyFunction):
    taken, val = greedy(items, constraint, keyFunction)
    print (“Total value of items taken = “, val)
    for item in taken:
        print (“ “, item)
def testGreedys(maxWeight = 20):
    items = buildItems()
    print (“Use greedy by value to fill knapsack of size”, maxWeight)
    testGreedy(items, maxWeight, value)
    print (“\nUse greedy by weight to fill knapsack of size”, maxWeight)
    testGreedy(items, maxWeight, weightInverse)
    print (“\nUse greedy by density to fill knapsack of size”, maxWeight)
    testGreedy(items, maxWeight, density)
def getBinaryRep(n, numDigits):
    """Assumes n and numDigits are non-negative ints
       Returns a numDigits str that is a binary
       representation of n"""
    result = “”
    while n > 0:
          result = str(n%2) + result
          n = n//2
    if len(result) > numDigits:
       raise ValueError('not enough digits')
    for i in range(numDigits - len(result)):
       result = '0' + result
    return result
def genPowerset(L):
    """Assumes L is a list
       Returns a list of lists that contains all possible
       combinations of the elements of L. E.g., if
       L is [1, 2] it will return a list with elements
       [], [1], [2], and [1,2]."""
    powerset = []
    for i in range(0, 2**len(L)):
        binStr = getBinaryRep(i, len(L))
        subset = []
        for j in range(len(L)):
            if binStr[j] == '1':
                subset.append(L[j])
        powerset.append(subset)
    return powerset
def chooseBest(pset, maxWeight, getVal, getWeight):
    bestVal = 0.0
    bestSet = None

    for items in pset:
        itemsVal = 0.0
        itemsWeight = 0.0

        for item in items:
            itemsVal += getVal(item)
            itemsWeight += getWeight(item)

        if itemsWeight <= maxWeight and itemsVal > bestVal:
           bestVal = itemsVal
           bestSet = items

    return (bestSet, bestVal)
def testBest(maxWeight = 20):

    items = buildItems()
    pset = genPowerset(items)
    taken, val = chooseBest(pset, maxWeight, Item.getValue,
                               Item.getWeight)
    print (“Total value of items taken =”, val)
    for item in taken:
        print item
```

Above Code contains a straightforward implementation of this **brute-force approach** to solving the 0/1 knapsack problem. It uses the classes and functions. There is a variant of the **knapsack problem**, called the **fractional (or continuous) knapsack problem**, for which a **greedy algorithm** is guaranteed to find an optimal solution. Since the items are infinitely divisible, it always makes sense to take as much as possible of the item with the highest remaining value to- weight ratio. Suppose, for example, that our burglar found only three things of value in the house: a sack of gold dust, a sack of silver dust, and a sack of raisins.

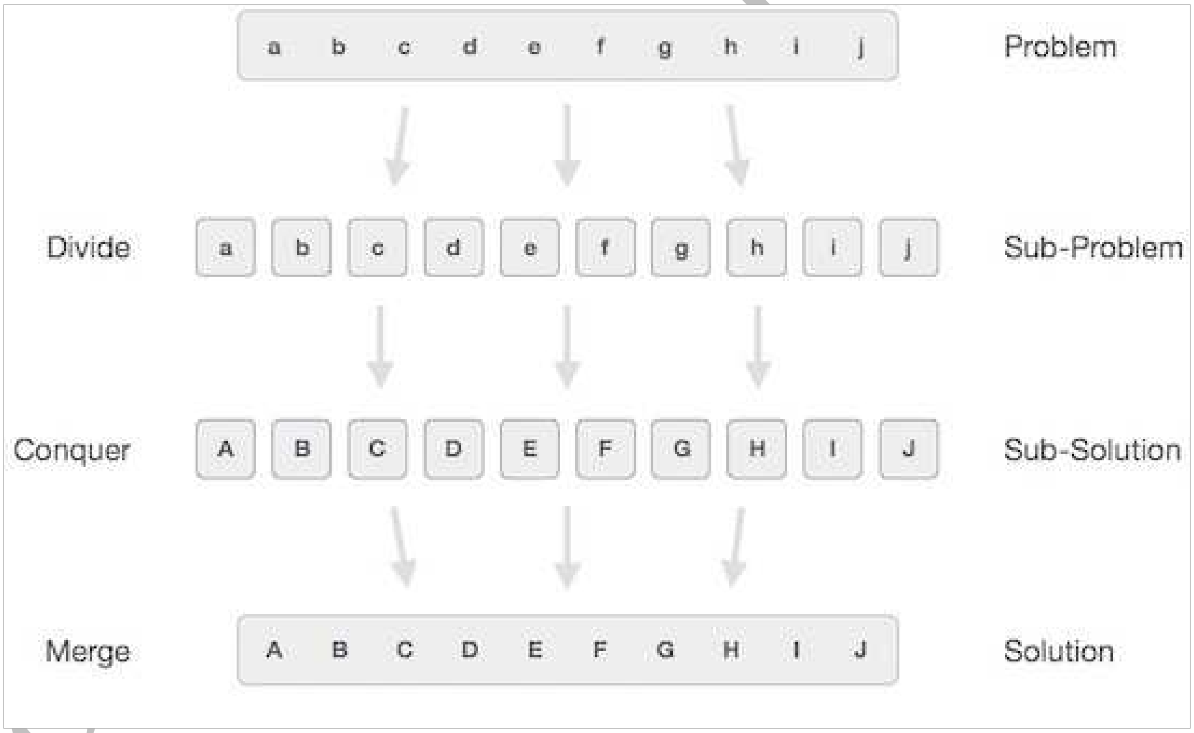
➢ Dynamic programming and divide and conquer :

Dynamic programming is based upon solving independent **subproblems** and then combining those solutions. There are, however, some important differences. **Divide-and-conquer algorithms** are based upon finding **subproblems** that are substantially smaller than the original problem.

For example, merge sort works by dividing the problem size in half at each step. In contrast, dynamic programming involves solving problems that are only slightly smaller than the original problem. The efficiency of **divide-and-conquer algorithms** does not depend upon structuring the algorithm so that the same problems are solved repeatedly. In contrast, dynamic programming is efficient only when the number of distinct **sub-problems** is significantly smaller than the total number of **sub-problems**.

In divide and conquer approach, the problem in hand, is divided into smaller sub-problems and then each problem is solved independently. When we keep on dividing the subproblems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. Those "atomic" smallest possible sub-problems (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.

Divide and conquer uses the Binary Search technique.


a b c d e f g h i j
Problem
Divide
a b c d e f g h i j
Sub-Problem
Conquer
A B C D E F G H I J
Sub-Solution
Merge
A B C D E F G H I J
Solution