In [1]:

```
# Program to display the Fibonacci sequence up to n-th term
nterms = int(input("How many terms? "))
# first two terms
n1, n2 = 0, 1
count = 0
# check if the number of terms is valid
if nterms <= 0:
    print("Please enter a positive integer")
# if there is only one term, return n1
elif nterms == 1:
    print("Fibonacci sequence upto",nterms,":")
    print(n1)
# generate fibonacci sequence
else:
    print("Fibonacci sequence:")
    while count < nterms:
        print(n1)
        nth = n1 + n2
        # update values
        n1 = n2
        n2 = nth
        count += 1
```

```
How many terms? 10
Fibonacci sequence:
0
1
1
2
3
5
8
13
21
34
```

In [2]:

```
# Python program to display the Fibonacci sequence
def recur_fibo(n):
    if n <= 1:
        return n
    else:
        return(recur_fibo(n-1) + recur_fibo(n-2))
    nterms = 7
# check if the number of terms is valid
if nterms <= 0:
    print("Plese enter a positive integer")
else:
    print("Fibonacci sequence:")
    for i in range(nterms):
        print(recur_fibo(i))
```

```
Fibonacci sequence:
0
1
1
2
3
5
8
13
21
34
```

In [6]:

```
# Node of a Huffman Tree

class Nodes:
    def __init__(self, probability, symbol, left = None, right = None):
        # probability of the symbol
        self.probability = probability
        # the symbol
        self.symbol = symbol
        # the left node
        self.left = left
        # the right node
        self.right = right
        # the tree direction (0 or 1)
        self.code = ''
""" A supporting function in order to calculate the probabilities of symbols in specifie
def CalculateProbability(the_data):
    the_symbols = dict()
    for item in the_data:
        if the_symbols.get(item) == None:
            the_symbols[item] = 1
        else:
            the_symbols[item] += 1
    return the_symbols
""" A supporting function in order to print the codes of symbols by travelling a Huffman
the_codes = dict()
def CalculateCodes(node, value = ''):
    # a huffman code for current node
    newValue = value + str(node.code)
    if(node.left):
        CalculateCodes(node.left, newValue)
    if(node.right):
        CalculateCodes(node.right, newValue)
    if(not node.left and not node.right):
        the_codes[node.symbol] = newValue
    return the_codes
""" A supporting function in order to get the encoded result """
def OutputEncoded(the_data, coding):
    encodingOutput = []
    for element in the_data:
        # print(coding[element], end = '')
        encodingOutput.append(coding[element])
    the_string = ''.join([str(item) for item in encodingOutput])
    return the_string
""" A supporting function in order to calculate the space difference between compressed
def TotalGain(the_data, coding):
    # total bit space to store the data before compression
    beforeCompression = len(the_data) * 8
    afterCompression = 0
    the_symbols = coding.keys()
    for symbol in the_symbols:
        the_count = the_data.count(symbol)
        # calculating how many bit is required for that symbol in total
        afterCompression += the_count * len(coding[symbol])
    print("Space usage before compression (in bits):", beforeCompression)
    print("Space usage after compression (in bits):", afterCompression)
def HuffmanEncoding(the_data):
    symbolWithProbs = CalculateProbability(the_data)
    the_symbols = symbolWithProbs.keys()
    the_probabilities = symbolWithProbs.values()
```

```
        print("symbols: ", the_symbols)
        print("probabilities: ", the_probabilities)
        the_nodes = []
        # converting symbols and probabilities into huffman tree nodes
        for symbol in the_symbols:
            the_nodes.append(Nodes(symbolWithProbs.get(symbol), symbol))
        while len(the_nodes) > 1:
            # sorting all the nodes in ascending order based on their probability
            the_nodes = sorted(the_nodes, key = lambda x: x.probability)
            # for node in nodes:
            # print(node.symbol, node.prob)
            # picking two smallest nodes
            right = the_nodes[0]
            left = the_nodes[1]
            left.code = 0
            right.code = 1
            # combining the 2 smallest nodes to create new node
            newNode = Nodes(left.probability + right.probability, left.symbol + right.symbol
            the_nodes.remove(left)
            the_nodes.remove(right)
            the_nodes.append(newNode)
        huffmanEncoding = CalculateCodes(the_nodes[0])
        print("symbols with codes", huffmanEncoding)
        TotalGain(the_data, huffmanEncoding)
        encodedOutput = OutputEncoded(the_data,huffmanEncoding)
        return encodedOutput, the_nodes[0]
    def HuffmanDecoding(encodedData, huffmanTree):
        treeHead = huffmanTree
        decodedOutput = []
        for x in encodedData:
            if x == '1':
                huffmanTree = huffmanTree.right
            elif x == '0':
                huffmanTree = huffmanTree.left
            try:
                if huffmanTree.left.symbol == None and huffmanTree.right.symbol == None:
                    pass
            except AttributeError:
                decodedOutput.append(huffmanTree.symbol)
                huffmanTree = treeHead
        string = ''.join([str(item) for item in decodedOutput])
        return string
    the_data = "AAAAAAABBCCCCCCDDDEEEEEEEEE"
    print(the_data)
    encoding, the_tree = HuffmanEncoding(the_data)
    print("Encoded output", encoding)
    print("Decoded Output", HuffmanDecoding(encoding, the_tree))
```

```
AAAAAAABBCCCCCCDDDEEEEEEEEE
symbols:  dict_keys(['A', 'B', 'C', 'D', 'E'])
probabilities:  dict_values([7, 2, 6, 3, 9])
symbols with codes {'E': '00', 'A': '01', 'C': '10', 'D': '110', 'B': '111'}
Space usage before compression (in bits): 216
Space usage after compression (in bits): 59
Encoded output 01010101010101111111101010101010110110110000000000000000000
Decoded Output AAAAAAABBCCCCCCDDDEEEEEEEEE
```

In [10]:

```
class Item:
    def __init__(self, value, weight):
        self.value = value
        self.weight = weight
def fractionalKnapsack(W, arr):
    # Sorting Item on basis of ratio
    arr.sort(key=lambda x: (x.value/x.weight), reverse=True)
    # Result(value in Knapsack)
    finalvalue = 0.0
    # Looping through all Items
    for item in arr:
        # If adding Item won't overflow,
        # add it completely
        if item.weight <= W:
            W -= item.weight
            finalvalue += item.value
            # If we can't add current Item,
            # add fractional part of it
        else:
            finalvalue += item.value * W / item.weight
            break
                # Returning final value
    return finalvalue
# Driver Code
if __name__ == "__main__":
    W = 50
    arr = [Item(60, 10), Item(100, 20), Item(120, 30)]
    # Function call
    max_val = fractionalKnapsack(W, arr)
    print(max_val)
```

```
240.0
```